

Ajax StarterKit

Tutorial Reference Library

Ajax isn't just one single technology, but a combination of several different languages and standards. To help you learn not only Ajax but also some of the technologies that make up Ajax, we've included on this CD some of the best Sams Teach Yourself tutorials on related topics like JavaScript and XML in easily accessible and searchable PDF format.

To jump to a particular title you can just click on a book cover below, or you can browse through the books by topic in the bookmarks on the left.

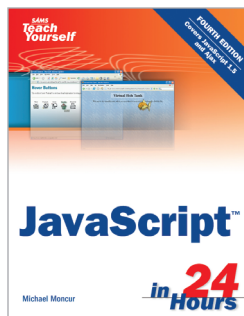


Ajax Starter Kit Quick Start Guide

(based on Sams Teach
Yourself Ajax in 10 Minutes)

Phil Ballard

ISBN 0-672-32868-2



Sams Teach Yourself JavaScript in 24 Hours

Michael Moncur

ISBN 0-672-32879-8



Sams Teach Yourself XML in 10 Minutes

Andrew Watt

ISBN 0-672-32471-7



Sams Teach Yourself HTML in 10 Minutes

Dee Hayes

ISBN 0-672-32878-X



Sams Teach Yourself CSS in 10 Minutes

Russ Weakley

ISBN 0-672-32745-7



Sams Teach Yourself PHP in 10 Minutes

Chris Newman

ISBN 0-672-32762-7



Quick Start Guide



- Ajax Basics
- Constructing Applications
- Using Libraries



Ajax Starter Kit

Quick Start Guide

Phil Ballard

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

Ajax Starter Kit Quick Start Guide

Copyright © 2007 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-10: 0-672-32960-3

ISBN-13: 978-0-672-32960-9

Library of Congress Cataloging-in-Publication data is on file.

Printed in the United States of America

First Printing: June 2007

09 08 07 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Reader Services

Visit our website and register this product at www.sampublishing.com/register for convenient access to any updates, downloads, or errata that may be available.

Table of Contents

Welcome to Ajax!	1
-------------------------	----------

Part I: A Refresher on Web Technologies

1: Anatomy of a Website	7
Workings of the World Wide Web	7
2: Writing Web Pages in HTML	13
Introducing HTML	13
Elements of an HTML Page	15
A More Advanced HTML Page	20
Some Useful HTML Tags	22
Cascading Style Sheets in Two Minutes	23
3: Sending Requests Using HTTP	25
Introducing HTTP	25
The HTTP Request and Response	26
HTML Forms	28
4: Client-Side Coding Using JavaScript	33
About JavaScript	33
In at the Deep End	35
Manipulating Data in JavaScript	44
5: Server-Side Programming in PHP	47
Introducing PHP	47
Embedding PHP in HTML Pages	48
Variables in PHP	49
Controlling Program Flow	51
6: A Brief Introduction to XML	53
Introducing XML	53
XML Basics	54
JavaScript and XML	57
The Document Object Model (DOM)	58

Part II: Introducing Ajax

7: Anatomy of an Ajax Application	61
The Need for Ajax	61
Introducing Ajax	63
The Constituent Parts of Ajax	66
Putting It All Together	68
8: The XMLHttpRequest Object	71
More About JavaScript Objects	71
Introducing XMLHttpRequest	73
Creating the XMLHttpRequest Object	73
9: Talking with the Server	81
Sending the Server Request	81
Monitoring Server Status.	86
The Callback Function	87
10: Using the Returned Data	91
The responseText and responseXML Properties	91
Another Useful JavaScript DOM Property	95
Parsing responseXML.	96
Providing User Feedback.	97
11: Our First Ajax Application	101
Constructing the Ajax Application.	101
The HTML Document	102
Adding JavaScript	103
Putting It All Together	107

Part III: More Complex Ajax Technologies

12: Returning Data as Text	111
Getting More from the responseText Property	111
13: AHAH—Asynchronous HTML and HTTP	119
Introducing AHAH.	119
Creating a Small Library for AHAH.	120
Using myAHAHlib.js.	122

14: Returning Data as XML	129
Adding the “x” to Ajax	129
The responseXML Property	130
Project—An RSS Headline Reader	133
15: Web Services and the REST Protocol	143
Introduction to Web Services	143
REST—Representational State Transfer	144
Using REST in Practice	146
REST and Ajax	150
16: Web Services Using SOAP	151
Introducing SOAP (Simple Object Access Protocol)	151
The SOAP Protocol	152
Using Ajax and SOAP	155
Reviewing SOAP and REST	156
17: A JavaScript Library for Ajax	157
An Ajax Library	157
Reviewing myAHAHlib.js	158
Implementing Our Library	159
Using the Library	163
Extending the Library	166
18: Ajax “Gotchas”	167
Common Ajax Errors	167
The Back Button	167
Bookmarking and Links	168
Telling the User That Something Is Happening	169
Making Ajax Degrade Elegantly	169
Dealing with Search Engine Spiders	170
Pointing Out Active Page Elements	170
Don’t Use Ajax Where It’s Inappropriate	171
Security	172
Test Code Across Multiple Platforms	172
Ajax Won’t Cure a Bad Design	173
Some Programming Gotchas	173

Part IV: Commercial and Open Source Ajax Resources

19: The prototype.js Toolkit	175
Introducing prototype.js	175
Wrapping XMLHttpRequest—the Ajax Object.	178
Example Project—Stock Price Reader	180
20: Using Rico	183
Introducing Rico.	183
Rico's Other Interface Tools.	187
21: Using XOAD	193
Introducing XOAD.	193
XOAD HTML	196
Advanced Programming with XOAD	199
Index	201

Welcome to Ajax!

Ajax is stirring up high levels of interest in the Internet development community. Ajax allows developers to provide visitors to their websites slick, intuitive user interfaces somewhat like those of desktop applications instead of using the traditional page-based web paradigm.

Based on well-known and understood technologies such as JavaScript and XML, Ajax is easily learned by those familiar with the mainstream web design technologies and does not require users to have any browser plug-ins or other special software.

About This Book

Part of the Sams Publishing *Teach Yourself in 10 Minutes* series, this book aims to teach the basics of building Ajax applications for the Internet. Divided into bite-sized lessons, each designed to take no more than about 10 minutes to complete, this volume offers

- A review of the technologies on which the World Wide Web is based
- Basic tutorials/refreshers in HTML, JavaScript, PHP, and XML
- An understanding of the architecture of Ajax applications
- Example Ajax coding projects

After completing all the lessons you'll be equipped to write and understand basic Ajax applications, including all necessary client- and server-side programming.

What Is Ajax?

Ajax stands for *Asynchronous Javascript and XML*. Although strictly speaking Ajax itself is not a technology, it mixes well-known programming techniques in an uncommon way to enable web developers to build Internet applications with much more appealing user interfaces than those to which we have become accustomed.

When using popular desktop applications, we expect the results of our work to be made available immediately, without fuss, and without us having to wait for the whole screen to be redrawn by the program. While using a spreadsheet such as Excel, for instance, we expect the changes we make in one cell to propagate immediately

INTRODUCTION: Welcome to Ajax!

through the neighboring cells while we continue to type, scroll the page, or use the mouse.

Unfortunately, this sort of interaction has seldom been available to users of web-based applications. Much more common is the experience of entering data into form fields, clicking on a button or link, and then sitting back while the page slowly reloads to exhibit the results of the request. In addition, we often find that the majority of the reloaded page consists of elements that are identical to those of the previous page and that have therefore been reloaded unnecessarily; background images, logos, and menus are frequent offenders.

Ajax promises us a solution to this problem. By working as an extra layer between the user's browser and the web server, Ajax handles server communications in the background, submitting server requests and processing the returned data. The results may then be integrated seamlessly into the page being viewed, without that page needing to be refreshed or a new one loaded.

In Ajax applications, such server requests are not necessarily synchronized with user actions such as clicking on buttons or links. A well-written Ajax application may already have asked of the server, and received, the data required by the user—perhaps before the user even knew she wanted it. This is the meaning of the *asynchronous* part of the Ajax acronym.

The parts of an Ajax application that happen “under the hood” of the user's browser, such as sending server queries and dealing with the returned data, are written in *JavaScript*, and *XML* is an increasingly popular means of coding and transferring formatted information used by Ajax to efficiently transfer data between server and client.

We'll look at all these techniques, and how they can be made to work together, as we work through the lessons.

About This Starter Kit

The Ajax Starter Kit includes everything a web developer needs to learn the basics of Ajax and its building-block technologies—HTML, JavaScript, PHP, and XML.

Aimed primarily at web developers seeking to build better interfaces for the users of their web applications, this book also should prove useful to web designers eager to learn how the latest techniques can offer new outlets for their creativity.

Although the nature of Ajax applications means that they require some programming, all the required technologies are explained from first principles within the book, so even those with little or no programming experience should be able to follow the lessons without a great deal of difficulty.

Quick Start Guide

The *Ajax Starter Kit Quick Start Guide* is the best starting point for the would-be Ajax developer. Divided into 21 short, easy-to-read lessons, the booklet offers an overview of the basics and

- A review of the technologies on which the World Wide Web is based
- Basic tutorials/refreshers in HTML, JavaScript, PHP, and XML
- An understanding of the architecture of Ajax applications
- Example Ajax coding projects

After completing all the lessons you'll be equipped to write and understand basic Ajax applications, including all necessary client- and server-side programming.

Reference Library

The *Ajax Starter Kit's* CD-ROM includes—in easy to search and read PDF format — a complete library of tutorials and how-to's on all the main technologies that make up Ajax:

- *Sams Teach Yourself JavaScript in 24 Hours*
- *Sams Teach Yourself HTML in 10 Minutes*
- *Sams Teach Yourself XML in 10 Minutes*
- *Sams Teach Yourself PHP in 10 Minutes*

Toolkit

The CD-ROM also includes a complete toolkit of all the technologies you need to set up a testing environment on your Windows, Mac, or Linux computer, so you can work with the examples from the book and begin to create your own:

- XAMPP for Windows, Mac OS X, and Linux—an easy-to-install package to set up a PHP- and MySQL-enabled Apache server on your computer

- The jEdit programming editor, for Windows, Mac, and Linux
- Prototype, Rico, and XOAD—three JavaScript and Ajax libraries that help simplify the tasks of the developer in creating Ajax applications
- The source code for all of the examples from the tutorials

Who This Book Is For

This volume is aimed primarily at web developers seeking to build better interfaces for the users of their web applications and programmers from desktop environments looking to transfer their applications to the Internet.

It also proves useful to web designers eager to learn how the latest techniques can offer new outlets for their creativity. Although the nature of Ajax applications means that they require some programming, all the required technologies are explained from first principles within the book, so even those with little or no programming experience should be able to follow the lessons without a great deal of difficulty.

What Do I Need To Use This Book?

The main requirement is to have an interest in exploring how people and computers might work better together. Although some programming experience, especially in JavaScript, will certainly be useful it is by no means mandatory because there are introductory tutorials in all the required technologies.

To try out the program code for yourself you need access to a web server and the means to upload files to it (for example, via File Transfer Protocol, usually called FTP). Make sure that your web host allows you to use PHP scripts on the server, though the majority do these days.

To write and edit program code you need a suitable text editor. Windows Notepad does the job perfectly well, though some specialized programmers' editors offer additional useful facilities such as line numbering and syntax highlighting. The appendix contains details of some excellent examples that may be downloaded and used free of charge.

TIPS offer useful shortcuts or easier ways to achieve something.

NOTES are snippets of extra information relevant to the current theme of the text.

CAUTIONS detail traps that may catch the unwary and advise how to avoid them.

Conventions Used in This Book

In addition to the main text of each lesson, you will find a number of boxes labeled as Tips, Notes, and Cautions.

Online Resources and Errata

Visit the Sams Publishing website at www.sampublishing.com where you can download the example code and obtain further information and details of errata.

Anatomy of a Website

We have a lot of ground to cover, so let's get to it. We'll begin by reviewing in this lesson what the World Wide Web is and what are the major components that make it work.

Workings of the World Wide Web

The World Wide Web operates using a client/server networking principle. When you enter the URL (the web address) of a web page into your browser and click on Go, you ask the browser to make an *HTTP request* of the particular computer having that address. On receiving this request, that computer returns ("serves") the required page to you in a form that your browser can interpret and display. Figure 1.1 illustrates this relationship. In the case of the Internet, of course, the server and client computers may be located anywhere in the world.

1: Anatomy of a Website

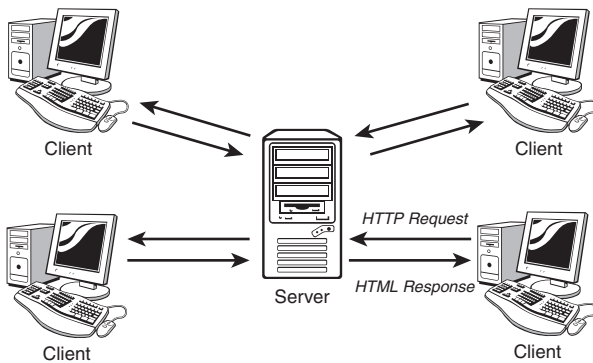


FIGURE 1.1 How web servers and clients (browsers) interact.

Lesson 3, “Sending Requests Using the HTTP Protocol,” discusses the nitty-gritty of HTTP requests in more detail. For now, suffice to say that your HTTP request contains several pieces of information needed so that your page may be correctly identified and served to you, including the following:

- The domain at which the page is stored (for example, mydomain.com)
- The name of the page (This is the name of a file in the web server’s file system—for example, mypage.html.)
- The names and values of any parameters that you want to send with your request

What Is a Web Page?

Anyone with some experience using the World Wide Web will be familiar with the term *web page*. The traditional user interface for websites involves the visitor navigating among a series of connected *pages* each containing text, images, and so forth, much like the pages of a magazine.

Generally, each web page is actually a separate file on the server. The collection of individual pages constituting a website is managed by a program called a *web server*.

CAUTION: The term *web server* is often used in popular speech to refer to both the web server program—such as Apache—and the computer on which it runs.

Web Servers

A web server is a program that interprets HTTP requests and delivers the appropriate web page in a form that your browser can understand. Many examples are available, most running under either UNIX/Linux operating systems or under some version of Microsoft Windows.

Perhaps the best-known server application is the *Apache Web Server* from the Apache Software Foundation (<http://www.apache.org>), an open source project used to serve millions of websites around the world (see Figure 1.2).

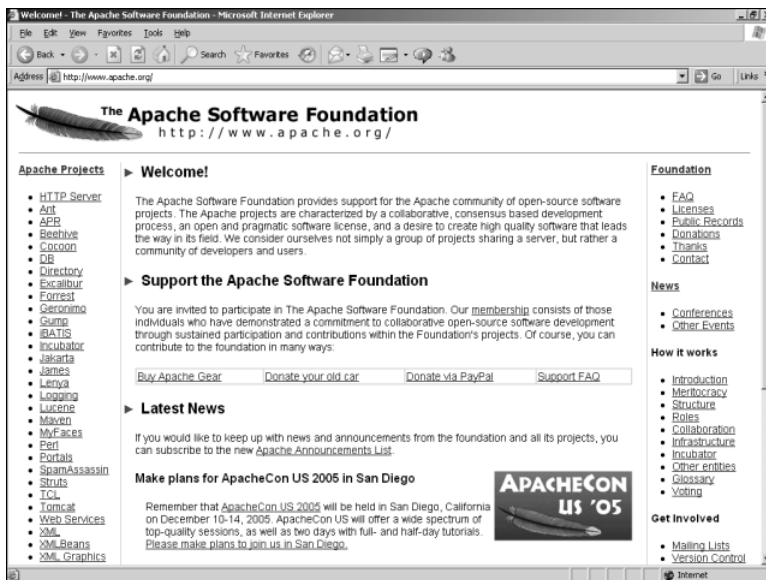


FIGURE 1.2 The Apache Software Foundation home page at <http://www.apache.org/> displayed in Internet Explorer.

ON THE CD: Apache for Windows, Mac, and Linux is included on the *Ajax Starter Kit CD*.

Another example is Microsoft's IIS (Internet Information Services), often used on host computers running the Microsoft Windows operating system.

Server-Side Programming

Server-side programs, scripts, or languages, refer to programs that run on the server computer. Many languages and tools are available for server-side programming, including PHP, Java, and ASP (the latter being available only on servers running the Microsoft Windows operating system). Sophisticated server setups often also include databases of information that can be addressed by server-side scripts.

The purposes of such scripts are many and various. In general, however, they all are designed to preprocess a web page before it is returned to you. By this we mean that some or all of the page content will have been modified to suit the context of your request—perhaps to display train times to a particular destination and on a specific date, or to show only those products from a catalog that match your stated hobbies and interests.

In this way server-side scripting allows web pages to be served with rich and varied content that would be beyond the scope of any design using only static pages—that is, pages with fixed content.

NOTE: Server-side programming in this book is carried out using the popular PHP scripting language, which is flexible, is easy to use, and can be run on nearly all servers. Ajax, however, can function equally well with any server-side scripting language.

Web Browsers

A *web browser* is a program on a web surfer's computer that is used to interpret and display web pages. The first graphical web browser, Mosaic, eventually developed into the famous range of browsers produced by Netscape.

NOTE: By *graphical* web browser we mean one that can display not only the text elements of an HTML document but also images and colors. Typically, such browsers have a point-and-click interface using a mouse or similar pointing device.

There also exist text-based web browsers, the best known of which is Lynx (<http://lynx.browser.org/>), which display HTML pages on character-based displays such as terminals, terminal emulators, and operating systems with command-line interfaces such as DOS.

The Netscape series of browsers, once the most successful available, were eventually joined by Microsoft's Internet Explorer offering, which subsequently went on to dominate the market.

Recent competitive efforts, though, have introduced a wide range of competing browser products including Opera, Safari, Konqueror, and especially Mozilla's Firefox, an open source web browser that has recently gained an enthusiastic following (see Figure 1.3).

Browsers are readily available for many computer operating systems, including the various versions of Microsoft Windows, UNIX/Linux, and Macintosh, as well as for other computing devices ranging from mobile telephones to PDAs (Personal Digital Assistants) and pocket computers.



FIGURE 1.3 The Firefox browser from Mozilla.org browsing the Firefox Project home page.

Client-Side Programming

We have already discussed how server scripts can improve your web experience by offering pages that contain rich and varied content created at the server and inserted into the page before it is sent to you.

Client-side programming, on the other hand, happens not at the server but right inside the user's browser *after* the page has been received. Such scripts allow you to carry out many tasks relating to the data in

the received page, including performing calculations, changing display colors and styles, checking the validity of user input, and much more.

Nearly all browsers support some version or other of a client-side scripting language called JavaScript, which is an integral part of Ajax and is the language we'll be using in this book for client-side programming.

DNS—The Domain Name Service

Every computer connected to the Internet has a unique numerical address (called an *IP address*) assigned to it. However, when you want to view a particular website in your browser, you don't generally want to type in a series of numbers—you want to use the domain name of the site in question. After all, it's much easier to remember `www.somedomain.com` than something like `198.105.232.4`.

When you request a web page by its domain name, your Internet service provider submits that domain name to a DNS server, which tries to look up the database entry associated with the name and obtain the corresponding IP address. If it's successful, you are connected to the site; otherwise, you receive an error.

The many DNS servers around the Internet are connected together into a network that constantly updates itself as changes are made. When DNS information for a website changes, the revised address information is propagated throughout the DNS servers of the entire Internet, typically within about 24 hours.

Summary

In Lesson 1 we discussed the history and development of the Internet and reviewed the functions of some of its major components including web servers and web browsers. We also considered the page-based nature of the traditional website user interface and had a brief look at what server- and client-side scripting can achieve to improve users' web surfing experience.

Writing Web Pages in HTML

In this lesson we introduce HTML, the markup language behind virtually every page of the World Wide Web. A sound knowledge of HTML provides an excellent foundation for the Ajax applications discussed in later lessons.

Introducing HTML

It wouldn't be appropriate to try to give an exhaustive account of HTML (Hypertext Markup Language)—or, indeed, any of the other component technologies of Ajax. Instead we'll review the fundamental principles and give some code examples to illustrate them, paying particular attention to the subjects that will become relevant when we start to develop Ajax applications.

What Is HTML?

The World Wide Web is constructed from many millions of individual pages, and those pages are, in general, written in Hypertext Markup Language, better known as HTML.

That name gives away a lot of information about the nature of HTML. We use it to mark up our text documents so that web browsers know how to display them and to define hypertext links within them to provide navigation within or between them.

Anyone who (like me) can remember the old pre-WYSIWYG word processing programs will already be familiar with text markup. Most of these old applications required that special characters be placed at the beginning and end of sections of text that you wanted to be displayed as (for instance) bold, italic, or underlined text.

ON THE CD: Look for *Sams Teach Yourself HTML in 10 Minutes* on the *Ajax Starter Kit* CD.

What Tools Are Needed to Write HTML?

Because the elements used in HTML markup employ only ordinary keyboard characters, all you really need is a good text editor to construct HTML pages. Many are available, and most operating systems have at least one such program already installed. If you're using some version of Windows, for example, the built-in Notepad application works just fine, or you can use Text Edit on Macs.

ON THE CD: Although Notepad or Text Edit are perfectly serviceable text editors, many so-called *programmers' editors* are available offering useful additional functions such as line numbering and syntax highlighting. A full-featured, cross-platform editor called jEdit is included on the *Ajax Starter Kit* CD.

Our First HTML Document

Let's jump right in and create a simple HTML document. Open Notepad (or whatever editor you've chosen to use) and enter the text shown in Listing 2.1. The HTML markup elements (often referred to as *tags*) are the character strings enclosed by < and >.

LISTING 2.1 testpage.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➡Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>A Simple HTML Document</title>
</head>
<body>
<h1>My HTML Page</h1>
Welcome to my first page written in HTML.<br />
This is simply a text document with HTML markup to show some
words in <b>bold</b> and some other words in <i>italics</i>.
```

CAUTION: Although text editors are ideal for writing program code, the use of word processing software can cause problems due to unwanted markup and other symbols that such programs often embed in the output code. If you choose to use a word processor, make sure that it is capable of saving files as plain ASCII text.

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

Now save the document somewhere on your computer, giving it the name `testpage.html`.

If you now load that page into your favorite browser, such as Internet Explorer or Firefox, you should see something like the window displayed in Figure 2.1.

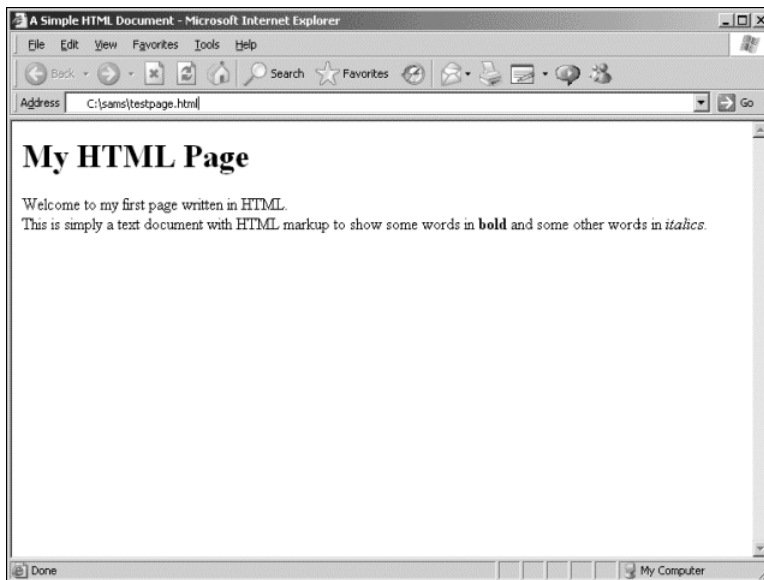


FIGURE 2.1 Our test document displayed in Internet Explorer.

Elements of an HTML Page

Let's look at Listing 2.1 in a little more detail.

The first element on the page is known as the `DOCTYPE` element. Its purpose is to notify the browser of the "flavor" of HTML used in the document. The `DOCTYPE` element used throughout this book refers to *HTML 4.0 Transitional*, a fairly forgiving version of the HTML specification that

allows the use of some earlier markup styles and structures in addition to the latest HTML 4.0 specifications.

The DOCTYPE element must always occur right at the beginning of the HTML document.

Next, note that the remainder of the document is enclosed by the elements `<html>` at the start of the page and `</html>` at the end. These tags notify the browser that what lies between should be interpreted and displayed as an HTML document.

The document within these outer tags is split into two further sections. The first is enclosed in `<head>` and `</head>` tags, and the second is contained between `<body>` and `</body>`. Essentially, the document's head section is used to store information about the document that is not to be displayed in the browser window, whereas the body of the document contains text to be interpreted and displayed to the user via the browser window.

CAUTION: Although many modern browsers correctly display HTML without these tags, it is bad practice to omit them. Even if the page is shown correctly on your own PC, you have no idea what operating system and browser a visitor may be using—he may not be so lucky.

The `<head>` of the Document

From Listing 2.1 we can see that the head section of our simple HTML document contains only one line—the words `A Simple HTML Document` enclosed in `<title>` and `</title>` tags.

Remember that the head section contains information that is not to be displayed in the browser window. This is not, then, the title displayed at the top of our page text, as you can confirm by looking again at Figure 2.1. Neither does the document title refer to the filename of the document, which in this case is `testpage.html`.

In fact, the document title fulfils a number of functions, among them:

- Search engines often use the page title (among other factors) to help them decide what a page is about.
- When you bookmark a page, it is generally saved by default as the document title.
- Most browsers, when minimized, display the title of the current document on their icon or taskbar button.

It's important, therefore, to choose a meaningful and descriptive title for each page that you create.

Many other element types are used in the head section of a document, including `link`, `meta`, and `script` elements. Although we don't give an account of them here, they are described throughout the book as they occur.

The Document `<body>`

Referring again to Listing 2.1, we can clearly see that the content of the document's body section is made up of the text we want to display on the page, plus some tags that help us to define how that text should look.

To define that certain words should appear in bold type, for example, we enclose those words in `` and `` tags. Similarly, to convert certain words into an italic typeface, we can use the `<i>` and `</i>` tags.

The heading, `My HTML Page`, is enclosed between `<h1>` and `</h1>` tags. These indicate that we intend the enclosed text to be a heading. HTML allows for six levels of headings, from `h1` (the most prominent) to `h6`. You can use any of the intermediate values `h2`, `h3`, `h4`, and `h5` to display pages having various levels of subtitles, for instance corresponding to chapter, section, and paragraph headings. Anything displayed within header tags is displayed on a line by itself.

TIP: If you want to write in the body section of the HTML page but *don't* want it to be interpreted by the browser and therefore displayed on the screen, you may do so by writing it as a *comment*. HTML comments start with the character string `<!--` and end with the string `-->` as in this example:

```
<!-- this is just a
comment and won't
be displayed in the
browser -->
```

All the tags discussed so far have been *containers*—that is, they consist of opening and closing tags between which you place the text that you want these tags to act upon. Some elements, however, are not containers but can be used alone. Listing 2.1 shows one such element: the `
` tag, which signifies a line break. Another example is `<hr />` (a horizontal line).

Adding Attributes to HTML Elements

Occasionally there is a need to specify exactly how a markup tag should behave. In such cases you can add (usually within the opening tag) parameter and value pairs, known as *attributes*, to change the behavior of the element:

```
<body bgColor="#cccccc">
... page content goes here ...
</body>
```

2: Writing Web Pages in HTML

In this example, the behavior of the `<body>` tag has been modified by adjusting its `BGCOLOR` (background color) property to a light gray. Figure 2.2 shows the effect this has if applied to our file `testpage.html`:

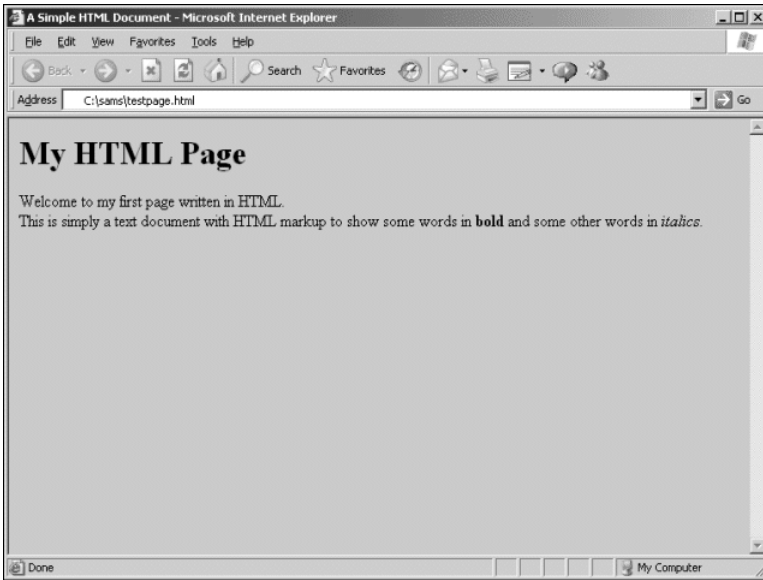


FIGURE 2.2 Our test page with the body color changed to gray.

TIP: Color values in HTML are coded using a hexadecimal system. Each color value is made up from three component values, corresponding to red, green, and blue. Each of the color values can range from hex 00 to hex ff (zero to 255 in decimal notation). The three hex numbers are concatenated into a string prefixed with a hash character #. The color value #000000 therefore corresponds to black, and #ffffff to pure white.

Images

Images can be inserted in our page by means of the `` tag. In this case we specify the source file of the image as a parameter by using the

src attribute. Other aspects of the image display that we can alter this way include the borders, width, and height of the image:

```

```

Border width, image width, and image height are in numbers of *pixels* (the “dots” formed by individual picture elements on the screen).

TIP: A further useful attribute for images is alt, which is an abbreviation of *alternative text*. This specifies a short description of the image that will be offered to users whose browsers cannot, or are configured not to, display images. Alternative text can also be important in making your website accessible to those with visual impairment and other disabilities:

```

```

Tables

Often you want to display information in tabular format, and HTML has a set of elements designed specifically for this purpose:

```
<table>
<tr><th>Column Header 1</th><th>Column Header 2</th></tr>
<tr><td>Data Cell 1</td><td>Data Cell 2</td></tr>
<tr><td>Data Cell 3</td><td>Data Cell 4</td></tr>
</table>
```

The <table> and </table> tags contain a nested hierarchy of other tags, including <tr> and </tr>, which define individual table rows; <th> and </th>, which indicate cells in the table’s header; and <td> and </td>, which contain individual cells of table data.

Look ahead to Figure 2.3 to see an example of how a table looks when displayed in a browser window.

Hyperlinks

Hypertext links (*hyperlinks*) are fundamental to the operation of HTML. By clicking on a hyperlink, you can navigate to a new location, be that to another point on the current page or to some point on a different page on another website entirely.

Links are contained within an `<a>`, or anchor tag, a container tag that encloses the content that will become the link. The destination of the link is passed to this tag as a parameter `href`:

Here is `my hyperlink`

Clicking on the words `my hyperlink` in the above example results in the browser requesting the page `newpage.html`.

TIP: A hyperlink can contain images as well as, or instead of, text. Look at this example:

```
<a href="newpage.html"></a>
```

Here, a user can click on the image `picfile.gif` to navigate to `newpage.html`.

A More Advanced HTML Page

Let's revisit our `testpage.html` and add some extra elements. Listing 2.2 shows `seville.html`, developed from our original HTML page but with different content in the `<body>` section of the document. Figure 2.3 shows how the page looks when displayed, this time in Mozilla Firefox.

Now we have applied a background tint to the body area of the document. The content of the body area has been centered on the page, and that content now includes an image (which we've given a two-pixel-wide border), a heading and a subheading, a simple table, and some text.

LISTING 2.2 seville.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>A Simple HTML Document</title>
</head>
<body bgcolor="#cccccc">
<center>

<h1>Guide to Seville</h1>
<h3>A brief guide to the attractions</h3>
```



```

<table border="2">
  <tr>
    <th bgcolor="#aaaaaa">Attraction</th>
    <th bgcolor="#aaaaaa">Description</th>
  </tr>
  <tr>
    <td>Cathedral</td>
    <td>Dating back to the 15th century</td>
  </tr>
  <tr>
    <td>Alcazar</td>
    <td>The medieval Islamic palace</td>
  </tr>
</table>
<p>Enjoy your stay in beautiful Seville.</p>
</center>
</body>
</html>

```

Let's take a closer look at some of the code.

First, we used the BGCOLOR property of the <body> tag to provide the overall background tint for the page:

```
<body bgcolor="#cccccc">
```

Everything in the body area is contained between the <center> tag (immediately after the body tag) and its partner </center>, immediately before the closing body tag. This ensures that all of our content is centered on the page.

The main heading is enclosed in <h1> ... </h1> tags as previously, but is now followed by a subheading using <h3> ... </h3> tags to provide a slightly smaller font size.

By using the border property in our opening <table> tag, we set a border width of two pixels for the table:

```
<table border="2">
```

Meanwhile we darkened the background of the table's header cells slightly by using the BGCOLOR property of the <th> elements:

```
<th bgcolor="#aaaaaa">Attraction</th>
```



FIGURE 2.3 seville.html shown in Mozilla Firefox.

Some Useful HTML Tags

Table 2.1 lists some of the more popular HTML tags.

TABLE 2.1 Some Common HTML Markup Elements

DOCUMENT TAGS

<code><html>...</html></code>	The entire document
<code><head>...</head></code>	Document head
<code><body>...</body></code>	Document body
<code><title>...</title></code>	Document title

STYLE TAGS

<code><a>...</code>	Hyperlink
<code>...</code>	Bold text
<code>...</code>	Emphasized text
<code>...</code>	Changed font

TIP: The World Wide Web Consortium is responsible for administering the definitions of HTML, HTTP, XML, and many other web technologies. Its website is at <http://www.w3.org/>.

STYLE TAGS

<code><i>...</i></code>	Italic text
<code><small>...</small></code>	Small text
<code><table>...</table></code>	Table
<code><tr>...</tr></code>	Table row
<code><th>...</th></code>	Cell in table header
<code><td>...</td></code>	Cell in table body
<code>...</code>	Bulleted list
<code>...</code>	Ordered (numbered) list
<code>...</code>	List item in bulleted or ordered list

Cascading Style Sheets in Two Minutes

The preceding approach to styling web pages has a few downsides.

First, you need to explicitly state the attributes of each page element. When you want to change the look of the page, you need to go through the source code line by line and change every instance of every attribute. This may be okay with a few simple pages, but as the amount of content increases, the pages become more difficult to maintain. Additionally, the attributes applied to HTML elements allow only limited scope for you to adjust how they are displayed.

Wouldn't it be better to make one change to the code and have that change applied to all HTML elements of a given type? As I'm sure you've already guessed, you can.

To achieve this goal you use *styles*. Styles may be embedded within your HTML document by using style tags in the head of the document:

```
<style type="text/css">
... style definition statements ...
</style>
```

Alternatively, they may be linked from an external file, using a link element, once again placed in the head section of the document:

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

TIP: You can even define styles on-the-fly. These are known as *inline styles* and can be applied to individual HTML elements. Taking the body tag of Listing 2.2 as an example:

```
<body bgcolor="#cccccc">
```

You could achieve the same effect using an inline style:

```
<body style="background-color:#cccccc">
```

Setting Style Sheet Rules

Style sheets allow you to set styling rules for the various HTML elements. A rule has two components: the selector, which identifies which HTML tag the rule should affect, and the declaration, which contains your styling rule. The following example defines a style for the paragraph element, <p>:

```
P {color: #333333}
```

This example determines that any text enclosed in paragraph tags <p> ... </p> should be displayed using dark gray text. You may also specify more than one rule for each tag. Suppose that, in addition to gray text, you want all text in the paragraph element to be displayed in italics:

```
P {color: #333333; font-style: italic}
```

A style sheet can contain as many such rules as you require.

You may also apply a declaration to more than one tag at once, by separating the tag selectors with commas. The following rule determines that all h1, h2, and h3 headings appear in blue text:

```
H1, H2, H3 {color: blue}
```

Summary

This lesson discussed the basics of web page layout using Hypertext Markup Language, including the structure of HTML documents, examples of HTML page elements, and page styling using both element attributes and cascading style sheets.

Sending Requests Using HTTP

Various protocols are used for communication over the World Wide Web, perhaps the most important being HTTP, the protocol that is also fundamental to Ajax applications. This lesson introduces the HTTP protocol and shows how it is used to request and receive information.

Introducing HTTP

HTTP or *Hypertext Transfer Protocol* is the main protocol of the World Wide Web. When you request a web page by typing its address into your web browser, that request is sent using HTTP. The browser is an *HTTP client*, and the web page server is (unsurprisingly) an *HTTP server*.

In essence, HTTP defines a set of rules regarding how messages and other data should be formatted and exchanged between servers and browsers.

Why Do I Need To Know About This?

Ajax sends server requests using the HTTP protocol. It's important to recognize the different types of HTTP requests and the responses that the server may return. Ajax applications need to construct HTTP requests to query the server and will base decisions about what to do next on the content of HTTP responses from the server.

What Is (and Isn't) Covered in This Lesson

It would be possible to fill the whole book with information on the HTTP protocol, but here we simply discuss it in terms of its roles in requesting web pages and passing information between them.

In this lesson you'll look at the construction of HTTP requests and responses and see how HTML forms use such requests to transfer data between web pages.

TIP: For a detailed account of HTTP, see Sams Publishing's *HTTP Developer's Handbook* by Chris Shiflett.

The HTTP Request and Response

The HTTP protocol can be likened to a conversation based on a series of questions and answers, which we refer to respectively as *HTTP requests* and *HTTP responses*.

The contents of HTTP requests and responses are easy to read and understand, being near to plain English in their syntax.

This section examines the structure of these requests and responses, along with a few examples of the sorts of data they may contain.

The HTTP Request

After opening a connection to the intended server, the HTTP client transmits a request in the following format:

- An opening line
- Optionally, a number of *header lines*
- A blank line
- Optionally, a message body

The opening line is generally split into three parts; the name of the *method*, the path to the required *server resource*, and the *HTTP version* being used. A typical opening line might read:

```
GET /sams/testpage.html HTTP/1.0
```

In this line we are telling the server that we are sending an HTTP request of type GET (explained more fully in the next section), we are sending this using HTTP version 1.0, and the server resource we require (including its local path) is

```
/sams/testpage.html.
```

NOTE: In this example the server resource we seek is on our own server, so we have quoted a relative path. It could of course be on another server elsewhere, in which case the server resource would include the full URL.

Header lines are used to send information about the request, or about the data being sent in the message body. One parameter and value pair is sent per line, the parameter and value being separated by a colon. Here's an example:

User-Agent: *[name of program sending request]*

For instance, Internet Explorer v5.5 offers something like the following:

User-agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)

A further example of a common request header is the Accept: header, which states what sort(s) of information will be found acceptable as a response from the server:

Accept: text/plain, text/html

NOTE: HTTP request methods include POST, GET, PUT, DELETE, and HEAD. By far the most interesting in our pursuit of Ajax are the GET and POST requests. The PUT, DELETE, and HEAD requests are not covered here.

By issuing the header in the preceding example, the request is informing the server that the sending application can accept either plain text or HTML responses (that is, it is not equipped to deal with, say, an audio or video file) .

The HTTP Response

In answer to such a request, the server typically issues an HTTP response, the first line of which is often referred to as the *status line*. In that line the server echoes the HTTP version and gives a response status code (which is a three-digit integer) and a short message known as a *reason phrase*. Here's an example HTTP response:

HTTP/1.0 200 OK

The response status code and reason phrase are essentially intended as machine-and human-readable versions of the same message, though the reason phrase may actually vary a little from server to server. Table 3.1 lists some examples of common status codes and reason phrases. The first digit of the status code usually gives some clue about the nature of the message:

- 1**—Information
- 2**—Success
- 3**—Redirected
- 4**—Client error
- 5**—Server error

TABLE 3.1 Some Commonly Encountered HTTP Response Status Codes

STATUS CODE	EXPLANATION
200 - OK	The request succeeded.
204 - No Content	The document contains no data.
301 - Moved Permanently	The resource has permanently moved to a different URL.
401 - Not Authorized	The request needs user authentication.
403 - Forbidden	The server has refused to fulfill the request.
404 - Not Found	The requested resource does not exist on the server.
408 - Request Timeout	The client failed to send a request in the time allowed by the server.
500 - Server Error	Due to a malfunctioning script, server configuration error or similar.

TIP: A detailed list of status codes is maintained by the World Wide Web Consortium, W3C, and is available at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

The response may also contain header lines each containing a header and value pair similar to those of the HTTP request but generally containing information about the server and/or the resource being returned:

```
Server: Apache/1.3.22
Last-Modified: Fri, 24 Dec 1999 13:33:59 GMT
```

HTML Forms

Web pages often contain fields where you can enter information. Examples include select boxes, check boxes, and fields where you can type information. Table 3.2 lists some popular HTML form tags.

TABLE 3.2 Some Common HTML Form Tags

TAG	DESCRIPTION
<form>...</form>	Container for the entire form
<input />	Data entry element; includes text, password, check box and radio button fields, and submit and reset buttons
<select>...</select>	Drop-down select box

TAG	DESCRIPTION
<code><option>...</option></code>	Selectable option within select box
<code><textarea>...</textarea></code>	Text entry field with multiple rows

After you have completed the form you are usually invited to submit it, using an appropriately labeled button or other page element.

At this point, the HTML form constructs and sends an HTTP request from the user-entered data. The form can use either the GET or POST request type, as specified in the `method` attribute of the `<form>` tag.

GET and POST Requests

Occasionally you may hear it said that the difference between GET and POST requests is that GET requests are just for GETting (that is, retrieving) data, whereas POST requests can have many uses, such as uploading data, sending mail, and so on.

Although there may be some merit in this rule of thumb, it's instructive to consider the differences between these two HTTP requests in terms of how they are constructed.

A GET request encodes the message it sends into a *query string*, which is appended to the URL of the server resource. A POST request, on the other hand, sends its message in the *message body* of the request. What actually happens at this point is that the entered data is encoded and sent, via an HTTP request, to the URL declared in the `action` attribute of the form, where the submitted data will be processed in some way.

Whether the HTTP request is of type GET or POST and the URL to which the form is sent are both determined in the HTML markup of the form. Let's look at the HTML code of a typical form:

```
<form action="http://www.sometargetdomain.com/somepage.htm"
  method="post">
Your Surname: <input type="text" size="50" name="surname"
/>
<br />
<input type="submit" value="Send" />
</form>
```

This snippet of code, when embedded in a web page, produces the simple form shown in Figure 3.1.

3 : Sending Requests Using HTTP

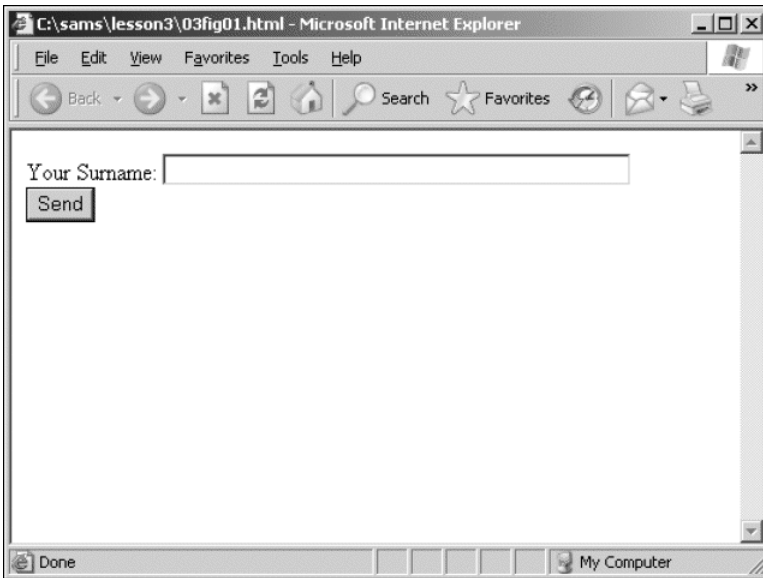


FIGURE 3.1 A simple HTML form.

Let's take a look at the code, line by line. First, we begin the form by using the `<form>` tag, and in this example we give the tag two attributes. The `action` attribute determines the URL to which the submitted form will be sent. This may be to another page on the same server and described by a relative path, or to a remote domain, as in the code behind the form in Figure 3.1.

Next we find the attribute `method`, which determines whether we want the data to be submitted with a GET or a POST request.

Now suppose that we completed the form by entering the value **Ballard** into the surname field. On submitting the form by clicking the Send button, we are taken to `http://www.sometargetdomain.com/somepage.htm`, where the submitted data will be processed—perhaps adding the surname to a database, for example.

The variable `surname` (the name attribute given to the Your Surname input field) and its value (the data we entered in that field) will also have been sent to this destination page, encoded into the body of the POST request and invisible to users.

Now suppose that the first line of the form code reads as follows:

```
<form action="http://www.sometargetdomain.com/somepage.htm"
  ➡ method="get">
```

On using the form, we would still be taken to the same destination, and the same variable and its value would also be transmitted. This time, however, the form would construct and send a GET request containing the data from the form. Looking at the address bar of the browser, after successfully submitting the form, we would find that it now contains:

```
http://www.example.com/page.htm?surname=Ballard
```

Here we can see how the parameter and its value have been appended to the URL. If the form had contained further input fields, the values entered in those fields would also have been appended to the URL as *parameter=value* pairs, with each pair separated by an & character. Here's an example in which we assume that the form has a further text input field called `firstname`:

```
http://www.example.com/page.htm?surname=Ballard&firstname=
Phil
```

Some characters, such as spaces and various punctuation marks, are not allowed to be transmitted in their original form. The HTML form encodes these characters into a form that can be transmitted correctly. An equivalent process decodes these values at the receiving page before processing them, thus making the encoding/decoding operation essentially invisible to the user. We can, however, see what this encoding looks like by making a GET request and examining the URL constructed in doing so.

Suppose that instead of the surname field in our form we have a `full-name` field that asks for the full name of the user and encodes that information into a GET request. Then, after submitting the form, we might see the following URL in the browser:

```
http://www.example.com/page.htm?fullname=Phil+Ballard
```

NOTE: In many cases, you may use either the POST or GET method for your form submissions and achieve essentially identical results. The difference becomes important, however, when you learn how to construct server calls in Ajax applications.

Here the space in the name has been replaced by the + character; the decoding process at the receiving end removes this character and replaces the space.

The XMLHttpRequest object at the heart of all Ajax applications uses HTTP to make requests of the server and receive responses. The content of these HTTP requests are essentially identical to those generated when an HTML form is submitted.

Summary

This lesson covered some basics of server requests and responses using the HTTP protocol, the main communications protocol of the World Wide Web. In particular, we discussed how GET and POST requests are constructed, and how they are used in HTML forms. Additionally, we saw some examples of responses to these requests that we might receive from the server.

Client-Side Coding Using JavaScript

In this lesson we introduce the concept of client-side scripting using JavaScript. Client-side scripts are embedded in web pages and executed by a JavaScript interpreter built into the browser. They add extra functionality to an otherwise static HTML page.

About JavaScript

JavaScript was developed from a language called LiveScript, which was developed by Netscape for use in its early browsers. JavaScript source code is embedded within the HTML code of web pages and interpreted and executed by the browser when the page is displayed.

Using JavaScript, you can add extra functionality to your web pages. Examples include

- Change the way page elements are displayed
- Add animation and other image effects
- Open pop-up windows and dialogs
- Check the validity of user-entered data

Nearly all modern browsers support JavaScript, though with a few differences in some commands. Where these occur, they are described in the text.

Why Do I Need To Know About JavaScript?

The *j* in Ajax stands for JavaScript; you use functions written in this language and embedded within your web pages to formulate Ajax server calls and to handle and process the response returned from the server.

What Is (and Isn't) Covered in This Lesson

There is no room here for an exhaustive guide to all JavaScript's functions. Instead this lesson concentrates on those aspects of the language necessary for later developing Ajax applications.

After completing this lesson, you'll have experience with the following:

- Embedding JavaScript commands and external JavaScript files into web pages
- Using some of the common JavaScript commands
- Using event handlers to launch JavaScript commands
- Working with JavaScript variables and objects
- Abstracting JavaScript commands into functions

JavaScript Basics

JavaScript commands can be embedded directly into HTML pages by placing them between `<script> ...</script>` tags. It is also common for JavaScript functions to be kept in a separate file on the server (usually with a file extension `.js`) and linked to HTML files where required, by placing a line like this into the head of the HTML file:

```
<SCRIPT language="JavaScript" SRC="myJS.js"></SCRIPT>
```

This allows you to call any JavaScript within the file `myJS.js`, just as if that source code had been typed directly into your own web page.

TIP: Placing JavaScript functions into external files allows them to be made available to a number of different web pages without having to retype any code. It also makes them easier to maintain because the latest version is automatically linked into the calling HTML page each time that page is requested.

It is possible to build up substantial JavaScript libraries in this way, linking them into web pages when their particular functions are required.

CAUTION: Although JavaScript is likely to be supported by your browser, it is usually possible for the browser options to be configured so as to disable its use. If you find that you cannot get any JavaScript commands to work, consult your browser's help files to find out how to check whether JavaScript is correctly enabled.

NOTE: Microsoft's Internet Explorer browser actually runs a proprietary Microsoft language called Jscript, instead of JavaScript. The two are, however, virtually identical and therefore largely compatible. Where differences occur, they are described in the text.

ON THE CD: For a much more thorough course in JavaScript, try *Sams Teach Yourself JavaScript in 24 Hours* by Michael Moncur, included on the *Ajax Starter Kit* CD.

In at the Deep End

Let's get right to it and add a JavaScript command to the simple web page we developed in Lesson 2, "Writing Web Pages in HTML."

CAUTION: JavaScript, unlike HTML, is case sensitive. When entering JavaScript commands, be careful not to enter characters in the incorrect case, or errors will occur.

Open your favorite text editor and load up seville.html (Listing 2.2 from Lesson 2). We're going to add the following code to the page, immediately after the `</p>` (the closing paragraph tag) on line 24:

```
<script language="JavaScript" type="text/javascript">
document.writeln("This line was written using
JavaScript!");
</script>
```

The whole of the source code with the extra lines added is shown in Listing 4.1. Make sure that you have added the code correctly; then save the file as testpage3.html and load it into your favorite browser.

LISTING 4.1 Adding JavaScript to an HTML Page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤ Transitional//EN" "http://www.w3.org/TR/html4/
➤ loose.dtd">
<html>
<head>
<title>A Simple HTML Document</title>
</head>
<body bgcolor="#cccccc">
<center>

<h1>A More Advanced HTML Page</h1>
<h3>Welcome to my second page written in HTML.</h3>
<table border="2">
<tr>
<th bgcolor="#aaaaaa">Vegetables</th>
<th bgcolor="#aaaaaa">Fruits</th>
</tr>
<tr>
<td>Carrot</td>
<td>Apple</td>
</tr>
<tr>
<td>Cabbage</td>
<td>Orange</td>
</tr>
</table><br />
<p>... and here's some text in a paragraph.</p>
```

4: Client-Side Coding Using JavaScript

```
<script language="JavaScript" type="text/javascript">
document.writeln("This line was written using JavaScript!")
</script>
</center>
</body>
</html>
```

If all has gone well, the page should now be like that shown in Figure 4.1. You should now be able to see an extra line of text toward the bottom of the page saying "This line was written using JavaScript!"

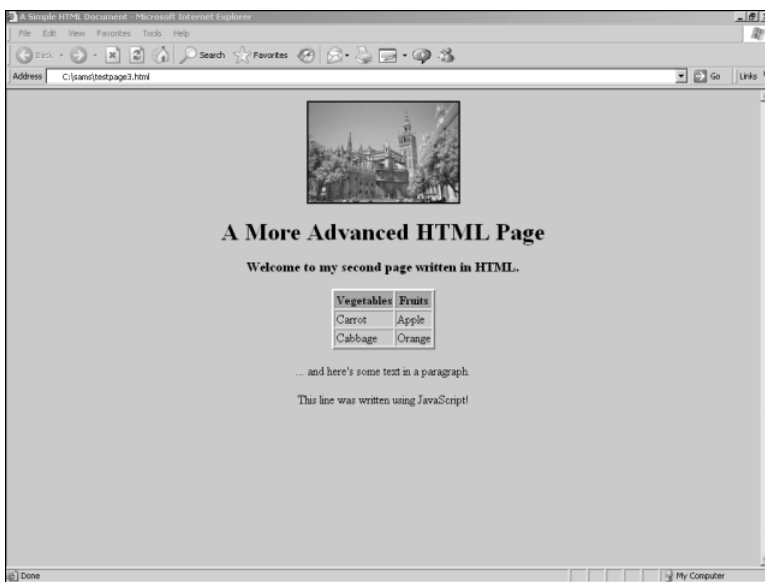


FIGURE 4.1 HTML document including one line written by JavaScript.

Let's look at our JavaScript code. The first item is the `<script>` tag, and here we have included the definition

`Language="JavaScript"`

which tells the browser that the statements contained within this script element should be interpreted as JavaScript.

Also in this tag appears the attribute

`type="text/javascript"`

TIP: There are other possible languages in which such scripts could be written; each has its own type declaration such as

```
type="text/vbscript"
```

or

```
type="text/xml"
```

NOTE: In addition to methods, objects also possess *properties*. Such properties tell you something about the object, as opposed to the object's methods, which perform actions upon it.

This declares that the script enclosed in the element is written in JavaScript.

The script is ended on the next to the last line with the familiar `</script>` tag.

Now for the meat in the sandwich:

```
document.writeln("This line was written using JavaScript!")
```

JavaScript (in common with many other programming languages) uses the concept of objects. The word `document` in this line of code refers to the object on which we want our JavaScript command to operate. In this case, we are dealing with the `document` object, which is the entire HTML document (including any embedded JavaScript code) that we are displaying in the browser. We'll have a further look at objects later in the lesson.

The term `writeln` describes the operation we want JavaScript to perform on the `document` object. We say it is a *method* of the `document` object, in this case one that writes a line of text into the document.

The string within the parentheses we refer to as the *argument* that we pass to the `writeln` method. In this case it tells the method what to write to the `document` object.

Including JavaScript in HTML Pages

We can include as many `<script>...</script>` tags in our page as we need. However, we must pay some attention to where in the document they are placed.

JavaScript commands are executed in the order in which they appear in the page. Note from Listing 4.1 that we entered our JavaScript code at exactly the place in the document where we want the new text to appear.

JavaScript can also be added to the head section of the HTML page. This is a popular place to keep JavaScript functions, which we'll describe shortly.

Event Handlers

Often you want your JavaScript code to be executed because something specific has occurred. In an HTML form, for instance, you may decide to have JavaScript check the validity of the data entered by the

user at the moment when the form is submitted. On another occasion, you may want to alert your user by opening a warning dialog whenever a particular button is clicked.

To achieve these effects you use special interfaces provided by the browser and known as *event handlers*. Event handlers allow you to call JavaScript methods automatically when certain types of events occur. Consider the following code:

```
<form>
<input type="button" value="Click Here"
  ➡   onClick="alert('Thanks for clicking!')">
</form>
```

Here we capture the action of the user clicking the button, using the `onClick` event handler. When the user's click is detected, the script carries out the instructions listed in the `onClick` attribute of the `input` tag: `onClick="alert('Thanks for clicking!')"`

This line calls the JavaScript `alert` method, which pops up a dialog box displaying a message and an OK button. The message to be displayed in the alert dialog is contained in the string passed to the `alert` method as an argument.

Let's add this code to our HTML document, as shown in Listing 4.2. Save the page as `testpage4.html` after you've made the changes and load it into the browser.

LISTING 4.2 Calling `alert()` from the `onClick` Event Handler

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional
➡//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>A Simple HTML Document</title>
</head>
<body bgcolor="#cccccc">
<center>

<h1>A More Advanced HTML Page</h1>
<h3>Welcome to my second page written in HTML.</h3>
<table border="2">
<tr>
  <th bgcolor="#aaaaaa">Vegetables</th>
  <th bgcolor="#aaaaaa">Fruits</th>
</tr>
<tr>
  <td>Carrot</td>
```

```

        <td>Apple</td>
    </tr>
    <tr>
        <td>Cabbage</td>
        <td>Orange</td>
    </tr>
</table><br />
<p>... and here's some text in a paragraph.</p>
<script language="JavaScript" type="text/javascript">
document.writeln("This line was written using JavaScript!")
</script>
<form>
<input type="button" value="Click Here" onClick="alert
➡('Thanks for clicking!')">
</form>
</center>
</body>
</html>

```

Our HTML page should now show our new button, as in Figure 4.2.

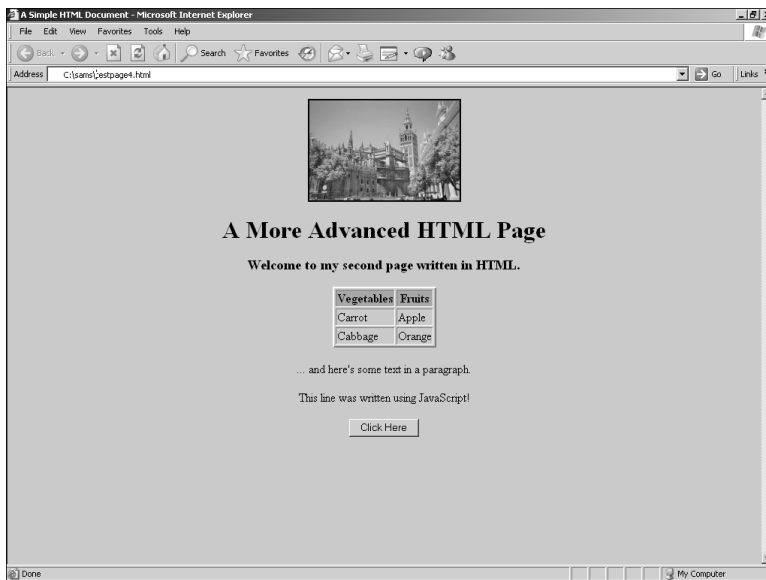


FIGURE 4.2 The new Click Here button in our web page.

Go ahead and click on the button. If everything goes according to plan, an alert dialog pops open as shown in Figure 4.3. You can click OK to clear the dialog.

Creating Functions

Often you will need to combine various JavaScript methods and objects, perhaps using many lines of code. JavaScript allows you to compose such blocks of instructions and name them, making your code easier to write, understand, and maintain.

For example, let's use another event handler, but this time we'll use it to call a function rather than to directly call a JavaScript method.

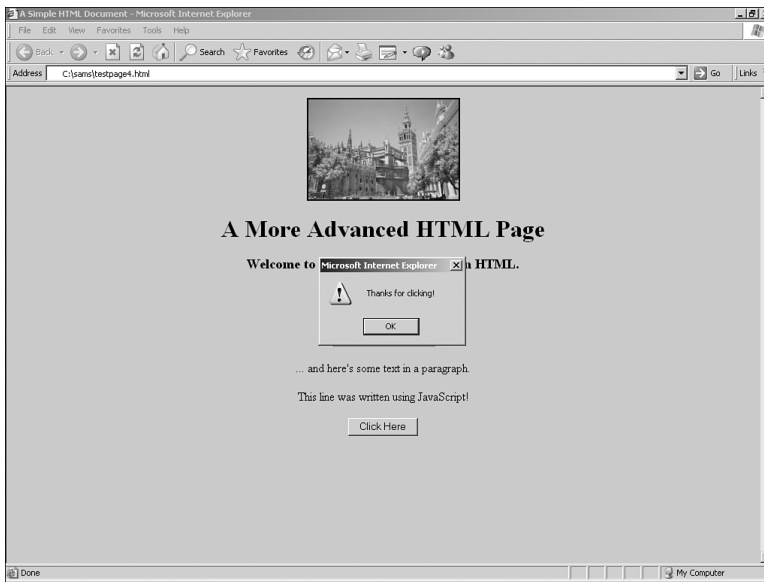


FIGURE 4.3 The dialog that appears after you click on the new button.

Here's the code for our function, which we'll place in the head section of our HTML document:

```
<script language="JavaScript">
function showAlert()
{
    alert("A Picture of Seville")
}
</script>
```

Within the usual `<script>` tags, we have now defined a function called `showAlert`, which carries out the commands contained within the curly braces. In this case, there is only one command, a call to the previously encountered `alert` method.

NOTE: Note that a function definition always starts with the word `function` followed by the function's name. The statements within a function are contained within curly braces `{}`.

We want this alert dialog to appear when the user's mouse passes over the photograph in our web page. We are therefore going to add an attribute to the `` tag that contains the image, as follows:

```

```

This line uses the `onMouseOver` event handler to detect when the cursor enters the area occupied by the photograph. When this happens, our new function `showAlert` is called.

Listing 4.3 shows the revised code.

LISTING 4.3 Using the `onMouseOver` Event Handler

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>A Simple HTML Document</title>
<script language="JavaScript" type="text/javascript">
function showAlert()
{
alert("A Picture of Seville")
}
</script>
</head>
<body bgcolor="#cccccc">
<center>

<h1>A More Advanced HTML Page</h1>
<h3>Welcome to my second page written in HTML.</h3>
<table border="2">
<tr>
<th bgcolor="#aaaaaa">Vegetables</th>
<th bgcolor="#aaaaaa">Fruits</th>
</tr>
<tr>
<td>Carrot</td>
<td>Apple</td>
</tr>
<tr>
<td>Cabbage</td>
<td>Orange</td>
</tr>
</table><br />
<p>... and here's some text in a paragraph.</p>
```

```
<script language="JavaScript" type="text/javascript">
document.writeln("This line was written using JavaScript!")
</script>
<form>
<input type="button" value="Click Here" onClick="alert
➤('Thanks for clicking!')">
</form>
</center>
</body>
</html>
```

With this HTML document loaded into your browser, roll your mouse over the photograph. An alert box should appear with the message "A Picture of Seville".

Passing Arguments to Functions

Of course, we could easily call our function from a wide variety of event handlers within our page and have it pop open an alert dialog. Unfortunately, the alert would always contain the message "A Picture of Seville", which is not very useful!

Wouldn't it be good if we could tell the function what message to display so that we could have different alert messages for different circumstances? We can achieve this by passing the message to our function as an argument:

```
<script language="JavaScript" type="text/javascript">
function showAlert(message)
{
    alert(message)
}
</script>
```

The function now "expects" to find the text for the message defined passed as an argument within the call. Rewrite the `onMouseOver` event handler for the image to provide this:

```

```

We'll also rewrite the button's `onClick` event handler to use this function but with a different message:

```
<input type="button" value="Click Here"
➤    onClick="showAlert('Thanks for clicking!')" />
```

Listing 4.4 shows the revised code.

LISTING 4.4 Calling JavaScript Functions from Event Handlers

```
<html>
<head>
<title>A Simple HTML Document</title>
<script language="JavaScript" type="text/javascript">
function showAlert(message)
{
alert(message)
}
</script>
</head>
<body bgcolor="#cccccc">
<center>

<h1>A More Advanced HTML Page</h1>
<h3>Welcome to my second page written in HTML.</h3>
<table border="2">
<tr>
<th bgcolor="#aaaaaa">Vegetables</th>
<th bgcolor="#aaaaaa">Fruits</th>
</tr>
<tr>
<td>Carrot</td>
<td>Apple</td>
</tr>
<tr>
<td>Cabbage</td>
<td>Orange</td>
</tr>
</table><br />
<p>... and here's some text in a paragraph.</p>
<script language="JavaScript" type="text/javascript">
document.writeln("This line was written using JavaScript!")
</script>
<form>
<input type="button" value="Click Here" onClick=
➤ "showAlert('Thanks for clicking!')">
</form>
</center>
</body>
</html>
```

Other Event Handlers

So far you have seen examples of the `onClick` and `onMouseOver` event handlers. Many others are available for use; Table 4.1 lists a selection of the most popular event handlers.

TABLE 4.1 Some Common JavaScript Event Handlers

EVENT HANDLER	COMMENTS
<code>onChange</code>	Occurs when the value in an input field changes
<code>onClick</code>	Occurs when a user clicks the mouse on the element in question
<code>onLoad</code>	Occurs when the page has finished loading
<code>onMouseOver</code>	Occurs when the mouse pointer enters the screen area occupied by the element in question ...
<code>onMouseOut</code>	... and when it leaves
<code>onSubmit</code>	Occurs at the point a form is submitted

Manipulating Data in JavaScript

You can use JavaScript to achieve much more than popping up dialog boxes. JavaScript gives you the opportunity to define and use variables and arrays, work with date and time arithmetic, and control program flow with loops and conditional branches.

Variables

The concept of a variable might already be familiar to you if you've ever done any algebra, or programmed in just about any computer language. A *variable* is a piece of data given a name by which you can conveniently refer to it later. In JavaScript, you declare variables with the keyword `var`:

```
var speed = 63;
```

The preceding line of code declares the variable `speed` and by using the assignment operator `=` assigns it a value of 63.

We may now use this variable in other statements:

```
var speedlimit = 55;
var speed = 63;
var excess_speed = speed - speedlimit;
```

Variables need not be numeric; the statement

```
var lastname = 'Smith';
```


assigns a string to the variable `lastname`.

Both numeric and string variables may be manipulated within JavaScript statements. Consider the following code:

```
var firstname = 'Susan';
var lastname = 'Smith';
document.writeln('Hello, ' + firstname + ' ' + lastname);
```

This code would write `Hello, Susan Smith` into our document.

Objects

You met the concept of an object earlier in the lesson and saw how objects have both properties that describe them and methods that perform actions on them.

Objects in JavaScript have a hierarchical relationship. References begin with the highest-level object, with subsequent levels appended separated by a period:

```
document.image1.src
```

NOTE: In fact, the object that truly has the highest level in the object hierarchy is `window`, which refers to the browser screen and everything within it. In general, you don't need to include this object; JavaScript assumes it to be there.

This string starts with the object `document`, then refers to an object `image1` within that object, and finally the property `src` (the source file for the image).

Suppose that we have the following HTML code somewhere in our page:

```
<form name="form1" action="somepage.html" method="post">
<input type="text" name="lastname">
<input type="submit" value="Submit">
</form>
```

We can refer, in JavaScript, to the string that the user has typed into the `lastname` field by referring to the property value of the object corresponding to that field:

```
document.form1.lastname.value
```

Example—Form Validation

Let's use this technique to check a user's entered form data for validity. We want to trap the event of the user attempting to submit the form and use this event to trigger our JavaScript function, which checks the data for validity. Here's the HTML code for our form:

```
<form name="form1" method="post" action="otherpage.html">
Enter a number from 1 to 10: <input size="4" type="text"
name="usernumber">
```

4: Client-Side Coding Using JavaScript

```
<input type="submit" value="Enter"  
  onSubmi t="return numcheck()">  
</form>
```

We can see here that the `onSubmit` event handler is called when the Submit button is clicked and calls a JavaScript function called `numcheck()`. We need this function to check what our user has entered for validity, and either submit the form or (if the entry is invalid) issue an error. Note the word `return` prior to the function call. This is here because on this occasion we want the function to tell us whether the `submit` method should be allowed to go ahead. We want our function to return a value of `false` to the form if the form submission is to be stopped. Here's the function:

```
<script language="JavaScript" type="text/javascript">  
function numcheck()  
{  
  var numentered = document.form1.usernumber.value;  
  if((numentered>=1)&&(numentered<=10))  
  {  
    return true;  
  } else  
  {  
    alert("Your entry was invalid. Please try  
again.");  
    return false;  
  }  
}
```

The first action of the function is to assign the user's entered value to the variable `numentered`. We then test that number for validity by checking that it is greater than or equal to 1 *and* less than or equal to 10. Depending on the result, we either return a value of `true` to the calling form (thus allowing the form to be submitted) or pop up a dialog informing of the error. In the latter case, when the user clicks OK to clear the dialog, a value of `false` is returned to the calling form, preventing the form from being submitted until the user enters appropriate data.

Summary

This lesson covered the basics of JavaScript programming. We saw how JavaScript commands may be integrated into HTML pages, discussed grouping JavaScript commands into functions, and learned how event handlers are employed to launch JavaScript commands and functions.

Server-Side Programming in PHP

Ajax applications can work with virtually any server-side language, requiring only that the server should return correctly formatted responses to its HTTP requests. This lesson introduces PHP, a popular open source scripting language used on a huge number of web servers throughout the world.

Introducing PHP

Like JavaScript, PHP is composed of commands that can be embedded within the HTML code of your pages. PHP however is a server-side programming language—that is, it works hand-in-hand with your web server to process the source code of a page *before* that page is sent to the browser.

Why Do I Need To Know This?

As you are already aware, Ajax applications make calls to the web server and subsequently use the returned information within the page currently being viewed. You need a way to run programs on the server to process the Ajax request and return the required data.

Ajax can work with various server-side technologies including PHP, ASP, Java, and others. This book uses PHP, arguably the most popular and easy to use of the available server-side languages.

This lesson provides an introduction to PHP for those who have never encountered it and a refresher of the basics for any who have.

What Is (and Isn't) Covered in This Lesson

As with every lesson in this part of the book, it is neither feasible nor appropriate to give an exhaustive course on every aspect of the subject.

This lesson covers the basics of PHP programming with some practical examples, concentrating mainly on those aspects of PHP most relevant to our explorations of Ajax.

Embedding PHP in HTML Pages

PHP statements are embedded into HTML documents by surrounding the PHP statements with `<?php` and `?>` tags. Anything between such tags is evaluated by the web server and replaced with appropriate HTML code, prior to the page being served to the browser.

You can have as many sets of `<?php` and `?>` tags in your page as you want.

Outputting HTML from PHP

Several PHP commands can help you write text and HTML code directly into your page. Perhaps the simplest is the `echo` command:

```
echo "I wrote this line using PHP";
```

The preceding statement simply places "I wrote this line using PHP" into the HTML document at precisely the place where the PHP statement occurs.

Listing 5.1 shows the source code of a PHP file to print `Hello World` in the browser window.

LISTING 5.1 Printing Hello World in PHP

```
<html>
<head>
<title>A Simple PHP Script</title>
</head>
<body>
<?php echo "<h1>Hello World!</h1>"; ?>
</body>
</html>
```

Note that in this script, the output string also contains some HTML tags, `<h1>` and `</h1>`. As the PHP statements are executed by the web

ON THE CD: PHP for Windows, Mac, and Linux is included on the *Ajax Starter Kit* CD.

TIP: For a more complete course on PHP, try *Sams Teach Yourself PHP in 10 Minutes* by Chris Newman on the *Ajax Starter Kit* CD.

TIP: Web servers normally recognize by the file extension which files contain PHP code and process them accordingly. The most used file extension for PHP files is `.php`, but you may also see `.php3`, `.php4`, `.phtml`, and various others. To make your code portable to as many web server environments as possible, it's best to stick with `.php`.

server before serving the page to us, these tags are written into the document's HTML along with the "Hello World" text and evaluated by our browser along with all other HTML markup in the document. Figure 5.1 shows the browser displaying our "Hello World" page.

If we ask the browser to show us the source of this page, it displays the following code, in which we can see that the PHP elements have been completely evaluated by the web server, which has inserted the relevant HTML into the page:

```
<html>
<head>
<title>A Simple PHP Script</title>
</head>
<body>
<h1>Hello World!</h1></body>
</html>
```

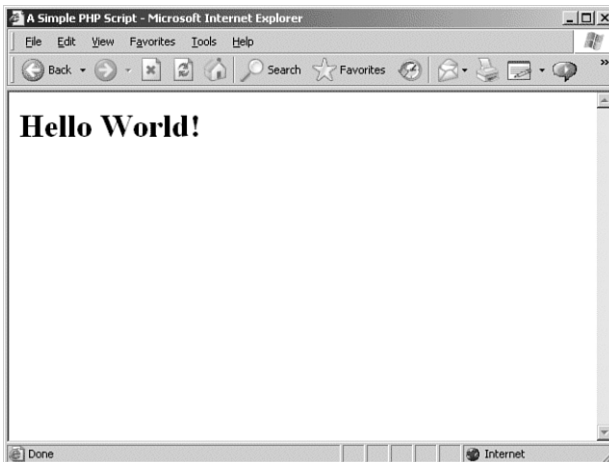


FIGURE 5.1 "Hello World" in PHP.

CAUTION: Variable names are case sensitive in PHP. For example, \$varname and \$VarName represent two distinct variables. Take care to enter the names of variables in the correct case.

Variables in PHP

Variables in PHP, much like in any programming language, are named *pockets* in which pieces of data are stored. All variable names in PHP must begin with a "\$" character, followed by a string made up of letters, numbers, and underscores.

We can assign values to variables in PHP without declaring the variables beforehand:

```
$score = 71;  
$player = 'Harry Scott';
```

Variables can take a number of data types, including strings, integers, floats, and Boolean (true or false). When a variable is assigned a value, such as in the preceding examples, PHP assigns a data type automatically.

Numbers

All the basic mathematical operators are available in PHP, as shown in the following examples:

```
$answer = 13 + 4;  
$answer = 13 * 4;  
$answer = 13 / 4;  
$answer = 13 - 4;
```

You can also calculate the modulus, for which we use the % character:

```
$answer = 13 % 4;
```

Strings

In PHP you enclose strings within single or double quotes:

```
$mystring = "The quick brown fox";
```

Strings may be concatenated using the period character:

```
$newstring = " jumped over the lazy dog";  
$concat = $mystring.$newstring;
```

TIP: PHP offers the `date()` command, which allows you to get the server time and date and format it to your liking; for example, the line `echo date('D F Y H:I');`

outputs the current date in a form similar to `Fri 16 December 2005 11:36`.

Arrays

PHP also supports arrays. An *array* is a variable that can contain a set of values rather than just one. Here's a PHP array containing some of the days of the week:

```
$daynames = array("Monday", "Tuesday", "Wednesday",  
    "Thursday", "Friday");
```

The items in an array are referenced by a key, which is an integer starting at zero and incrementing for each item in the array. The following line outputs Thursday to an HTML page:

```
echo $daynames[3];
```

Note that, because the index value begins at zero, the preceding statement actually echoes the fourth element of the array.

This type of array is known as a *numeric array*, but you may also use *associative arrays*. In this case, the key value of each element is not numeric but instead is a string of your choosing. The syntax to declare such an array and assign values to it is slightly different:

```
$lunch = array("Susan" => "Chicken", "Matthew" => "Beef",  
➡      "Louise" => "Salmon");
```

You can now select the elements of such an array using the key value:

```
echo $lunch["Louise"];
```

This command would output the word `Salmon` to our page.

Controlling Program Flow

PHP contains various structures for controlling the flow of your programs. One of the most useful is the simple `if` statement, which allows you to alter the flow of program execution depending on the outcome of a condition. Let's have a look at a code snippet using the `if` statement:

```
if($temp > 80)  
{  
➡    echo $temp." degrees is too hot. Turn down  
    the thermostat."; }  
}
```

This `if` statement simply evaluates the condition contained in the brackets. If the condition is satisfied, the statements within the curly braces are executed; otherwise, these statements are ignored.

We can also add an `else` clause to our `if` statement:

```
if($temp > 80)  
{  
➡    echo $temp." degrees is too hot. Turn down  
    the thermostat."; }  
else
```

5: Server-Side Programming in PHP

```
{  
    echo $temp." degrees is cool enough."  
}
```

PHP also has loop constructs, which allow you to repeat the same code instructions a number of times until the conditions are satisfied for the loop to be terminated. This is the code for a `while` loop:

```
$x = 1;  
while($x<=12)  
{  
    echo "This is trip number ".$x." through the loop<br  
>";  
    $x++;  
}
```

The statement `$x++` means “increment x by one.” The loop executes over and over until the condition

`$x<=12`

is no longer met (because `$x` has become greater than 12), and the statements within the curly braces will then be ignored. Program execution then carries on from below the closing curly brace.

You can also make a similar loop using PHP’s `for` construct:

```
for($x = 1; $x <= 12; $x++)  
{  
    echo "This is trip number ".$x." through the loop<br />";  
}
```

The `for` statement takes an argument with three components. The first is evaluated before the first loop and provides a starting value for `$x`. The second component of the argument is the condition that will be evaluated on each loop to test whether the loop should be executed, and the third is a statement that will be carried out after each loop, and in this case increments `$x`.

The operation of this loop is identical to that of the `while` example.

Summary

This lesson introduced the principles of programming in the PHP server-side language, including the use of variables and program flow control constructs. You have also seen how PHP statements may be embedded into HTML pages.

A Brief Introduction to XML

The “x” of Ajax stands for XML, a powerful markup language that can allow your Ajax applications to transfer and process complex, structured information. This lesson discusses the basics of creating and using XML documents.

Introducing XML

Anyone who has carried out any HTML markup will already be somewhat familiar with the nature of XML code. XML (eXtensible Markup Language) has many similarities in markup style to HTML.

However, whereas HTML is intended to determine how web pages are displayed, XML has a rather more wide-ranging use. XML documents can be used in all manner of data storage and data exchange applications ranging from document storage and retrieval to roles traditionally fulfilled by database programs.

Why Do I Need To Know This?

One of the many uses of XML is for the transfer of structured information between applications. In Ajax you can use XML to return information from the server to your Ajax application, where it may be parsed and used.

What Is (and Isn't) Covered in This Lesson

In common with the other lessons in this section of the book, we do not attempt to offer a complete and thorough treatise on XML. Rather, this lesson covers the basics of the language and its application, concentrating mainly on those aspects relevant to your work with Ajax.

ON THE CD: If you want a more in-depth tutorial in XML, see *Sams Teach Yourself XML in 10 Minutes* by Andrew H. Watt, on the *Ajax Starter Kit* CD.

XML Basics

XML is a markup language that allows data to be stored and transmitted in a structured, hierarchical manner. It has similarities in markup style to HTML, but whereas HTML has a fixed list of element definitions and is designed primarily to allow you to define how a document should be displayed, XML elements may be defined within a particular XML document to suit the data being described there.

In common with HTML, markup elements (normally referred to as *tags*) enclosed by < and > are used to annotate the contents of a text file, describing the information it contains.

Unlike the tags in HTML, though, whose definitions are fixed, XML tags can be defined to be anything you want, allowing you to describe virtually any kind of data. Consider this example of an XML document:

```
<race>
  <yacht raceNo='74'>
    <name>Wanderer</name>
    <skipper>Walter Jeffries</skipper>
    <helm>Sally Jacobs</helm>
  </yacht>
  <yacht raceNo='22'>
    <name>Free Spirit</name>
    <skipper>Jennifer Scully</skipper>
    <helm>Paul Thomas</helm>
  </yacht>
</race>
```

This short XML document describes a yacht race, including the two competing yachts and their respective personnel. Note how the tag names are descriptive of the data they contain, and how the tag structures are hierarchical. You may also notice that XML tags, like those of HTML, can also have *attributes*. The end effect is that the XML file is quite readable—that is, the meaning of the data may be readily inferred by a human reader.

NOTE: The similarities between XML and HTML are not purely accidental. Both are based on SGML (Standard Generalized Markup Language), a system for organizing the elements of a document. SGML was developed and standardized by the International Organization for Standards (ISO).

CAUTION: Unlike HTML, tagnames in XML are case sensitive, so <yacht> and <Yacht> would be treated as two distinct elements.

TIP: XML uses the same syntax as HTML for the display of comments. Any information beginning with the character string `<!--` and ending with the string `-->` will be ignored:

```
<!-- This is a comment -->
```

XML Document Structure

The permitted structure of an XML document has only one mandatory element, the so-called *document element*. In the preceding yacht race example, this would be the `<race>` element.

NOTE: The document element need not necessarily have elements nested within it; the following is an allowable XML document:

```
<competition>Farlington Summer Cup</competition>
```

Document Prolog

CAUTION: If such a declaration exists, it must be the first thing in the document. Not even white space is allowed before it.

Other information may be optionally included before the document element, forming the document's *prolog*. An example is the XML declaration:

```
<?xml version="1.0" ?>
```

The prolog may also contain, in addition to various comments and processing instructions, a *Document Type Declaration*.

Document Type Declaration

CAUTION: Take care not to confuse the Document Type (DOCTYPE) Declaration with the *Document Type Definition* (DTD). The DTD is comprised of both the markup declarations contained in the DOCTYPE Declaration and those contained in any external file to which the DOCTYPE Declaration refers.

The optional Document Type Declaration (often referred to as a *DOCTYPE declaration*) is a statement of the permitted structure of an XML document. It usually contains (or refers to another file that contains) information about the names of the elements in the document and the relationships between those elements.

Let's look at an example Document Type Declaration for the yacht race document:

```
<!DOCTYPE race SYSTEM race.dtd>
```

This declaration, which would appear in the document before the `<race>` element, specifies that the document element will be called `<race>` and that document structure definitions may be found in an

external file, `race.dtd`, which would perhaps contain something like the following:

```
<!ELEMENT race (yacht+) >
<!ELEMENT yacht (name, skipper, helm) >
<!ATTLIST yacht raceNo #CDATA #REQUIRED >
<!ELEMENT name (#PCDATA) >
<!ELEMENT skipper (#PCDATA) >
<!ELEMENT helm (#PCDATA) >
```

Alternatively, this information could be quoted in the DOCTYPE Declaration itself, placed between [and] characters:

```
<!DOCTYPE race [
<!ELEMENT race (yacht+) >
<!ATTLIST yacht raceNo #CDATA #REQUIRED >
<!ELEMENT yacht (name, skipper, helm) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT skipper (#PCDATA) >
<!ELEMENT helm (#PCDATA) >
]>
```

In either case we define four *elements*—namely, `race`, `yacht`, `skipper`, and `helm`—and one *attribute list*.

TIP: DOCTYPE

Declarations can contain both internal and external references, known as the *internal and external subsets* of the DTD.

Element Declarations

The line

```
<!ELEMENT race (yacht+) >
```

declares that the `<race>` element will contain elements of type `<yacht>`, whereas the `+` character indicates that there may be any number of occurrences from one upward of such `<yacht>` elements. Alternatively, we could use the character `*` to indicate any number of occurrences *including zero*, or the character `?` to indicate *zero or one occurrence*. The absence of all of these characters indicates that there should be exactly one `<yacht>` element within `<race>`.

The `<yacht>` element is declared to contain three further elements, `<name>`, `<skipper>`, and `<helm>`. The `#PCDATA` term contained in the declarations for those elements stands for *parsed character data* and indicates that these elements must contain character-based data and may not contain further elements. Other possible content types include MIXED (text and elements) and ANY (any valid content).

Attribute List Declarations

Our example also contains the line

```
<!ATTLIST yacht raceNo #CDATA #REQUIRED >
```

Such declarations are used to specify what attributes are permitted or required for any given element. In our example, we specify that the `<yacht>` element has an attribute called `raceNo`, the value of which is comprised of `#CDATA` (character data).

The term `#REQUIRED` indicates that, in this example, the `<yacht>` element *must* have such an attribute. Other possibilities include `#IMPLIED`, specifying that such an attribute is optional; `#DEFAULT` followed by a value in quotation marks, specifying a default value for the attribute should none be declared in the XML document; or `#FIXED` followed by a value in quotation marks, fixing the value of the attribute to that quoted.

Valid XML

If an XML document contains a DOCTYPE Declaration and complies fully with the declarations it contains, it is said to be a *valid* XML document.

JavaScript and XML

Most modern browsers already contain some tools to help you deal with XML documents.

A JavaScript object must exist to contain the XML document. Creating a new instance of such an object is done slightly differently depending on whether you use a non-Microsoft browser, such as Mozilla's Firefox, or Microsoft Internet Explorer:

For Firefox and other non-Microsoft browsers, use the following code to create a JavaScript XML document object:

```
<script type="text/javascript">
var myxmlDoc =
    document.implementation.createDocument("", "", null);

myxmlDoc.load("exampleDoc.xml");
```

Program statements

```
</script>
```

To create a JavaScript XML document object with Internet Explorer, use this code:

```
<script type="text/javascript">
var myxmlDoc=new ActiveXObject("Microsoft.XMLDOM")
myxmlDoc.async="false"
myxmlDoc.load("exampleDoc.xml")
```

Program statements

```
</script>
```

After you have an object to represent the XML document, you may use the properties and methods of that object to gain access to the XML data contained within the document. Effectively, the hierarchical structure and data of the XML document now have equivalents in the JavaScript hierarchy of objects, the Document Object Model (DOM).

The Document Object Model (DOM)

Let's take a look at some of the methods and properties that help you access and manipulate this information, often called *Walking The DOM*.

Nodes

Suppose that our JavaScript object `myxmlDoc` contains the XML listing of the yacht race. The document element, `<race>`, contains two elements of type `<yacht>`; we say it has two *children*.

In general, you can get the number of children belonging to a particular element by using the `childNodes.length` property of the object. Because `<race>` is the document element, it is at the top of the object hierarchy, and we can refer to it simply with the variable `myxmlDoc`:

```
var noYachts = myxmlDoc.childNodes.length;
```

We can also determine information about individual children by appending the node number in parentheses:

```
myxmlDoc.childNodes(0)
```

The preceding line refers to the first `<yacht>` element appearing in the document.

CAUTION: As in many programming constructs, the first element has the number zero, the second element has the number one, and so forth.

We can test for the presence of children for a particular element by using the `hasChildNodes()` method:

```
myxmlDoc.childNodes(1).hasChildNodes()
```

This line returns `true` because the second yacht in the document has three children (with tag names `name`, `skipper`, and `helm`). However, `myxmlDoc.childNodes(1).childNodes(0).hasChildNodes()`

returns `false` because the `<name>` element within that `<yacht>` element has no children.

Getting Tagnames

The `tagname` property allows you to find the tagname associated with a particular element:

```
myxmlDoc.childNodes(0).childNodes(1).tagName
```

The preceding line returns `skipper`.

Getting Element Attributes

The method `getAttribute("AttributeName")` can be used to return the attribute values for a given element:

```
myxmlDoc.childNodes(0).getAttribute("raceNo")
```

This line returns `74`.

Tag Contents

The `text` property can be used to return the contents of a particular element. The line

```
myxmlDoc.childNodes(0).childNodes(1).text
```

would return `Walter Jeffries`.

You'll learn about these and similar methods in more detail in Lesson 14, "Returning Data as XML."

Summary

This lesson discussed the basics of XML, including XML document structures and Document Type Declarations. We also briefly examined how JavaScript may be used to deal with XML data using object properties and methods, much like using any other JavaScript object. This knowledge will be useful when we use Ajax to retrieve XML data from the server.

Anatomy of an Ajax Application

In this lesson you will learn about the individual building blocks of Ajax and how they fit together to form the architecture of an Ajax application. Subsequent lessons here in Part II, "Introducing Ajax," examine these components in more detail, finally assembling them into a working Ajax application.

The Need for Ajax

In Part I, "A Refresher on Web Technologies," we reviewed the core technologies that form the components of an Ajax application. By now, you will hopefully have at least a rudimentary knowledge of JavaScript, PHP, and XML, all of which we'll use here in Part II.

Before discussing the individual components, let's look in more detail at what we want from our Ajax application.

Traditional Versus Ajax Client-Server Interactions

Lesson 1, "Anatomy of a Website," discussed the traditional page-based model of a website user interface. When you interact with such a website, individual pages containing text, images, data entry forms, and so forth are presented one at a time. Each page must be dealt with individually before navigating to the next.

For instance, you may complete the data entry fields of a form, editing and re-editing your entries as much as you want, knowing that the data will not be sent to the server until the form is finally submitted.

Figure 7.1 illustrates this interaction.

7: Anatomy of an Ajax Application

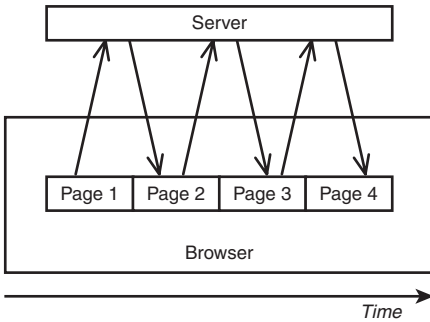


FIGURE 7.1 Traditional client-server interactions.

After you submit a form or follow a navigation link, you then must wait while the browser screen refreshes to display the new or revised page that has been delivered by the server.

As your experience as an Internet user grows, using this interface becomes almost second nature. You learn certain rules of thumb that help to keep you out of trouble, such as “don’t press the Submit button a second time,” and “don’t press the Back button after submitting a form.”

Unfortunately, interfaces built using this model have a few drawbacks. First, there is a significant delay while each new or revised page is loaded. This interrupts what we, as users, perceive as the “flow” of the application.

Furthermore, a *whole* page must be loaded on each occasion, even when most of its content is identical to that of the previous page. Items common to many pages on a website, such as header, footer, and navigation sections, can amount to a significant proportion of the data contained in the page.

Figure 7.2 illustrates a website displaying pages before and after the submission of a form, showing how much identical content has been reloaded and how relatively little of the display has actually changed.

This unnecessary download of data wastes bandwidth and further exacerbates the delay in loading each new page.

NOTE: *Bandwidth* refers to the capacity of a communications channel to carry information. On the Internet, bandwidth is usually measured in bps (bits per second) or in higher multiples such as Mbps (million bits per second).

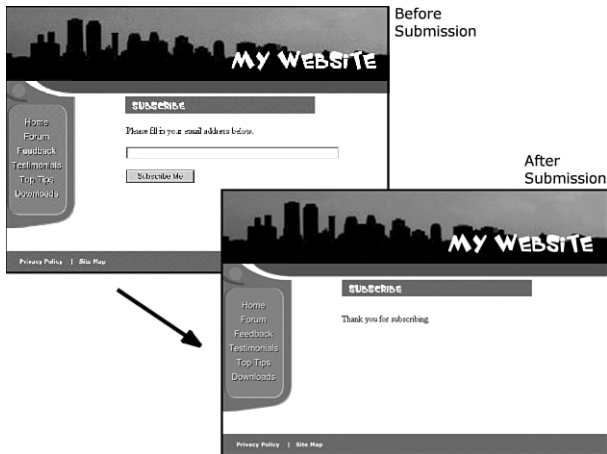


FIGURE 7.2 Many page items are reloaded unnecessarily.

The Rich User Experience

The combined effect of the issues just described is to offer a much inferior user experience compared to that provided by the vast majority of desktop applications.

On the desktop, you expect the display contents of a program to remain visible and the interface elements to respond to commands while the computing processes occur quietly in the background. As I write this lesson using a word processor, for example, I can save the document to disk, scroll or page up and down, and alter font faces and sizes without having to wait on each occasion for the entire display to be refreshed.

Ajax allows you to add to your web application interfaces some of this functionality more commonly seen in desktop applications and often referred to as a *rich user experience*.

Introducing Ajax

To improve the user's experience, you need to add some extra capabilities to the traditional page-based interface design. You want your user's page to be interactive, responding to the user's actions with revised content, and be updated without any interruptions for page loads or screen refreshes.

To achieve this, Ajax builds an extra layer of processing between the web page and the server.

This layer, often referred to as an *Ajax Engine* or *Ajax Framework*, intercepts requests from the user and in the background handles server communications quietly, unobtrusively, and *asynchronously*. By this we mean that server requests and responses no longer need to coincide with particular user actions but may happen at any time convenient to the user and to the correct operation of the application. The browser does not freeze and await the completion by the server of the last request but instead lets the user carry on scrolling, clicking, and typing in the current page.

The updating of page elements to reflect the revised information received from the server is also looked after by Ajax, happening dynamically while the page continues to be used.

Figure 7.3 represents how these interactions take place.

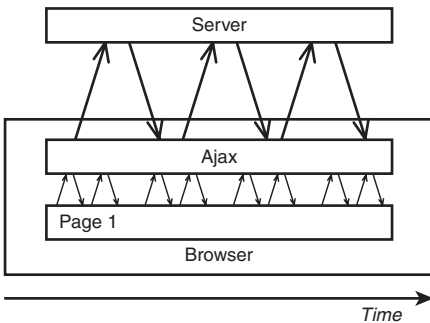


FIGURE 7.3 Ajax client-server interaction.

A Real Ajax Application—Google Suggest

To see an example of an Ajax application in action, let's have a look at *Google Suggest*. This application extends the familiar Google search engine interface to offer the user suggestions for suitable search terms, based on what he has so far typed.

With each key pressed by the user, the application's Ajax layer queries Google's server for suitably similar search phrases and presents the returned data in a drop-down box. Along with each suggested phrase

is listed the number of results that would be expected for a search conducted using that phrase. At any point the user has the option to select one of these suggestions instead of continuing to type and have Google process the selected search.

Because the server is queried with every keypress, this drop-down list updates dynamically as the user types—with no waiting for page refreshes or similar interruptions.

Figure 7.4 shows the program in action. You can try it for yourself by following the links from Google's home page at <http://www.google.com/webhp?complete=1&hl=en>.

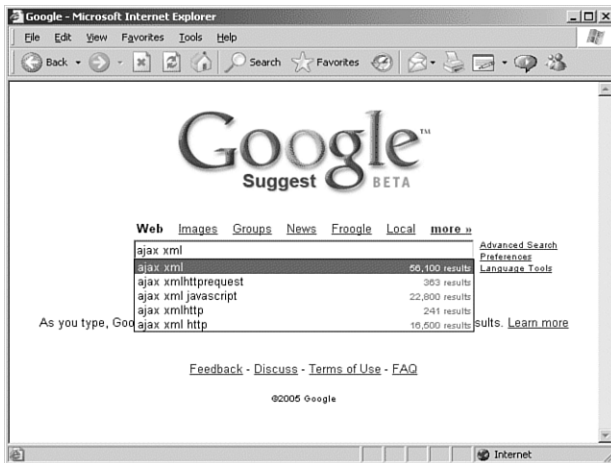


FIGURE 7.4 An example of an Ajax application—Google Suggest.

Next let's identify the individual components of such an Ajax application and see how they work together.

NOTE: Google has presented other Ajax-enabled applications that you can try, including the *gmail* web mail service and the *Google Maps* street mapping program. See the Google website at <http://www.google.com/> for details.

The Constituent Parts of Ajax

Now let's examine the components of an Ajax application one at a time.

The XMLHttpRequest Object

When you click on a hyperlink or submit an HTML form, you send an HTTP request to the server, which responds by serving to you a new or revised page. For your web application to work asynchronously, however, you must have a means to send HTTP requests to the server *without* an associated request to display a new page.

You can do so by means of the XMLHttpRequest *object*. This JavaScript object is capable of making a connection to the server and issuing an HTTP request without the necessity of an associated page load.

In following lessons you will see how an instance of such an object can be created, and how its properties and methods can be used by JavaScript routines included in the web page to establish asynchronous communications with the server.

Lesson 8, "The XMLHttpRequest Object," discusses how to create an instance of the XMLHttpRequest object and reviews the object's properties and methods.

TIP: As a security measure, the XMLHttpRequest object can generally only make calls to URLs within the same domain as the calling page and cannot directly call a remote server.

Talking with the Server

In the traditional style of web page, when you issue a server request via a hyperlink or a form submission, the server accepts that request, carries out any required server-side processing, and subsequently serves to you a new page with content appropriate to the action you have undertaken.

While this processing takes place, the user interface is effectively frozen. You are made quite aware of this, when the server has completed its task, by the appearance in the browser of the new or revised page.

With asynchronous server requests, however, such communications occur in the background, and the completion of such a request does not necessarily coincide with a screen refresh or a new page being loaded. You must therefore make other arrangements to find out what progress the server has made in dealing with the request.

The XMLHttpRequest object possesses a convenient property to report on the progress of the server request. You can examine this property

using JavaScript routines to determine the point at which the server has completed its task and the results are available for use.

Your Ajax armory must therefore include a routine to monitor the status of a request and to act accordingly. We'll look at this in more detail in Lesson 9, "Talking with the Server."

What Happens at the Server?

So far as the server-side script is concerned, the communication from the XMLHttpRequest object is just another HTTP request. Ajax applications care little about what languages or operating environments exist at the server; provided that the client-side Ajax layer receives a timely and correctly formatted HTTP response from the server, everything will work just fine.

It is possible to build simple Ajax applications with no server-side scripting at all, simply by having the XMLHttpRequest object call a static server resource such as an XML or text file.

Ajax applications may make calls to various other server-side resources such as web services. Later on we'll look at some examples of calling web services using protocols such as SOAP and REST.

NOTE: Here we'll be using the popular PHP scripting language for our server-side routines, but if you are more comfortable with ASP, JSP, or some other server-side language, go right ahead and use it in your Ajax applications.

Dealing with the Server Response

Once notified that an asynchronous request has been successfully completed, you may then utilize the information returned by the server.

Ajax allows for this information to be returned in a number of formats, including ASCII text and XML data.

Depending on the nature of the application, you may then translate, display, or otherwise process this information within the current page.

We'll look into these issues in Lesson 10, "Using the Returned Data."

Other Housekeeping Tasks

An Ajax application will be required to carry out a number of other duties too. Examples include detecting error conditions and handling them appropriately, and keeping the user informed about the status of submitted Ajax requests.

You will see various examples in later lessons.

Putting It All Together

Suppose that you want to design a new Ajax application, or update a legacy web application to include Ajax techniques. How do you go about it?

First you need to decide what page events and user actions will be responsible for causing the sending of an asynchronous HTTP request. You may decide, for example, that the `onMouseOver` event of an image will result in a request being sent to the server to retrieve further information about the subject of the picture; or that the `onClick` event belonging to a button will generate a server request for information with which to populate the fields on a form.

You saw in Lesson 4, “Client-Side Coding Using JavaScript,” how JavaScript can be used to execute instructions on occurrences such as these, by employing event handlers. In your Ajax applications, such methods will be responsible for initiating asynchronous HTTP requests via `XMLHttpRequest`.

Having made the request, you need to write routines to monitor the progress of that request until you hear from the server that the request has been successfully completed.

Finally, after receiving notification that the server has completed its task, you need a routine to retrieve the information returned from the server and apply it in the application. You may, for example, want to use the newly returned data to change the contents of the page’s body text, populate the fields of a form, or pop open an information window.

Figure 7.5 shows the flow diagram of all this.

In Lesson 11, “Our First Ajax Application,” we’ll use what we have learned to construct a complete Ajax application.

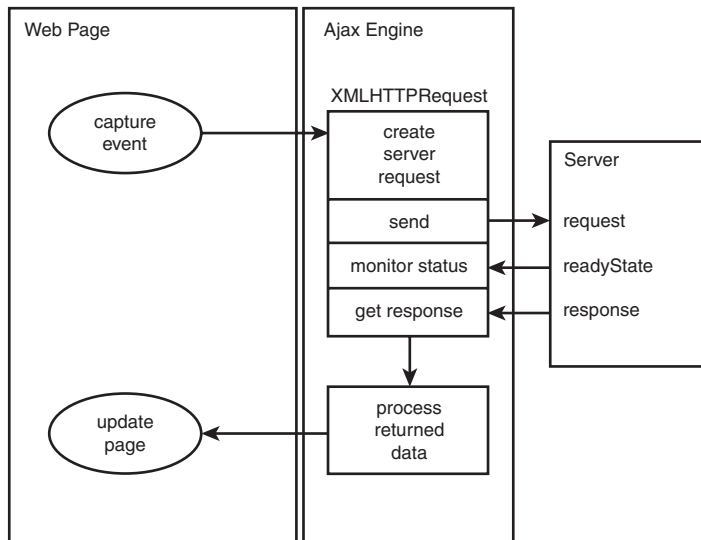


FIGURE 7.5 How the components of an Ajax application work together.

Summary

This lesson discussed the shortcomings of the traditional web interface, identifying specific problems we want to overcome. We also introduced the various building blocks of an Ajax application and discussed how they work together.

In the following lessons of Part II, we will look at these components in more detail, finally using them to build a complete Ajax application.

The XMLHttpRequest Object

In this lesson you will learn how to create an instance of the XMLHttpRequest object regardless of which browser your user may have. The object's properties and methods will be introduced.

More About JavaScript Objects

Lesson 7, "Anatomy of an Ajax Application," introduced the building blocks of an Ajax application and discussed how these pieces fit together.

This lesson examines the object at the heart of every Ajax application—the XMLHttpRequest object.

NOTE: You briefly met *objects* in Lesson 4, "Client-Side Coding Using JavaScript," when we discussed the document object associated with a web page. The XMLHttpRequest object, after it has been created, becomes a further such object within the page's object hierarchy and has its own properties and methods.

An object can be thought of as a single package containing a set of *properties*, which contain and classify data, and a set of *methods* with which the object can perform actions on that data.

Suppose, for example, that we had an object of type `wheelbarrow`. Such an object might have a property `contents`, which describes how many items the wheelbarrow holds at any given moment. Methods might include `fill()`, `tip()`, `forward()`, and `stop()`. When using JavaScript you can design such objects as you see fit.

However, in addition to user-defined objects, JavaScript has a range of ready-made objects for use in scripts. These are referred to as *native* objects. Examples of JavaScript's native objects include `Math()`, `String()`, and `Date()`.

Creating an Instance of an Object

Many objects, such as the document object that you saw in Lesson 4, already exist and therefore do not need you to create an instance of them. Others, however, require you to create an instance of the object in question before you can use it.

You can create an instance of an object by calling a method known as the object's *constructor*, using the `new` keyword:

```
var myBarrow = new Wheelbarrow();
```

Having created an instance `myBarrow` of the object `wheelbarrow`, properties and methods for the object may be manipulated using a simple syntax:

```
myBarrow.contents = 20;
myBarrow.forward();
myBarrow.stop();
myBarrow.tip();
```

Of course, you are at liberty to create other instances of the same object and have them exist concurrently:

```
var myBarrow = new Wheelbarrow();
var yourBarrow = new Wheelbarrow();
myBarrow.contents = 20;
yourBarrow.contents = 50;
```

The Document Object Model or DOM

We mentioned briefly in Lesson 4 the hierarchy of objects “built in” to a web page and known as the *Document Object Model*. You access these objects and their properties and methods in the same way as native objects and objects that you devise and create yourself.

NOTE: The Document Object Model or DOM is really not a part of JavaScript but a separate entity existing outside it. Although you can use JavaScript to manipulate DOM objects, other scripting languages may equally well access them too.

CAUTION: Some browsers may require attention to their security settings to allow the XMLHttpRequest object to operate correctly. See your browser's documentation for details.

TIP: Although the object's name begins with *XML*, in fact, any type of document may be returned from the server; ASCII text, HTML, and XML are all popular choices, and we will encounter all of these in the course of the book.

In later lessons you'll see how the XMLHttpRequest object can use XML data returned from the server in response to XMLHttpRequest calls to create additional DOM objects that you can use in your scripts.

Introducing XMLHttpRequest

XMLHttpRequest is supported by virtually all modern browsers, including Microsoft's Internet Explorer 5+ and a variety of non-Microsoft browsers, including Mozilla, Firefox, Konqueror, Opera, and Safari, and is supported on a wide range of platforms, including Microsoft Windows, UNIX/Linux, and Mac OS X.

The purpose of the XMLHttpRequest object is to allow JavaScript to formulate HTTP requests and submit them to the server. Traditionally programmed web applications normally make such requests *synchronously*, in conjunction with a user-initiated event such as clicking on a link or submitting a form, resulting in a new or updated page being served to the browser.

Using XMLHttpRequest, however, you can have your page make such calls *asynchronously* in the background, allowing you to continue using the page without the interruption of a browser refresh and the loading of a new or revised page.

This capability underpins all Ajax applications, making the XMLHttpRequest object the key to Ajax programming.

Creating the XMLHttpRequest Object

You cannot make use of the XMLHttpRequest until you have created an instance of it. Creating an instance of an object in JavaScript is usually just a matter of making a call to a method known as the object's constructor. In the case of XMLHttpRequest, however, you must change this routine a little to cater for the peculiarities of different browsers, as you see in the following section.

Different Rules for Different Browsers

Microsoft first introduced the XMLHttpRequest object, implementing it in Internet Explorer 5 as an *ActiveX object*.

Most other browser developers have now included into their products an equivalent object, but implemented as a native object in the browser's JavaScript interpreter.

Because you don't know in advance which browser, version, or operating system your users will have, your code must adapt its behavior on-the-fly to ensure that the instance of the object will be created successfully.

For the majority of browsers that support XMLHttpRequest as a native object (Mozilla, Opera, and the rest), creating an instance of this object is straightforward. The following line creates an XMLHttpRequest object called request:

```
var request = new XMLHttpRequest();
```

Here we have declared a variable request and assigned to it the value returned from the statement `new XMLHttpRequest()`, which is invoking the constructor method for the XMLHttpRequest object.

To achieve the equivalent result in Microsoft Internet Explorer, you need to create an ActiveX object. Here's an example:

```
var request = new ActiveXObject("Microsoft.XMLHTTP");
```

Once again, this assigns the name request to the new object.

To complicate matters a little more, some versions of Internet Explorer have a different version of MSXML, the Microsoft XML parser, installed; in those cases you need to use the following instruction:

```
var request = new ActiveXObject("Msxml2.XMLHTTP");
```

TIP: ActiveX is a proprietary Microsoft technology for enabling active objects into web pages. Among the available web browsers, it is currently only supported in Microsoft's Internet Explorer. Internet Explorer uses its built-in XML parser, MSXML, to create the XMLHttpRequest object.

A Solution for All Browsers

You need, therefore, to create a script that will correctly create an instance of a XMLHttpRequest object regardless of which browser you are using (provided, of course, that the browser supports XMLHttpRequest).

A good solution to this problem is to have your script try in turn each method of creating an instance of the object, until one such method succeeds. Have a look at Listing 8.1, in which such a strategy is used.

LISTING 8.1 Using Object Detection for a Cross-Browser Solution

```
function getXMLHttpRequest()
{
    var request = false;
    try
    {
        request = new XMLHttpRequest(); /* e.g. Firefox */
    }
    catch(err1)
    {
        try
        {
            vrequest = new ActiveXObject("Msxml2.XMLHTTP");
            /* some versions IE */
        }
        catch(err2)
        {
            try
            {
                request = new ActiveXObject("Microsoft.XMLHTTP");
                /* some versions IE */
            }
            catch(err3)
            {
                request = false;
            }
        }
    }
    return request;
}
```

Listing 8.1 uses the JavaScript statements `try` and `catch`. The `try` statement allows us to attempt to run a piece of code. If the code runs without errors, all is well; however, should an error occur we can use the `catch` statement to intervene before an error message is sent to the user and determine what the program should then do about the error.

TIP: Note the syntax:

```
catch(identifier)
```

Here `identifier` is an object created when an error is caught. It contains information about the error; for instance, if you wanted to alert the user to the nature of a JavaScript runtime error, you could use a code construct like this:

```
catch(err)
{
    alert(err.description);
}
```

to open a dialog containing details of the error.

An alternative, and equally valid, technique would be to detect which type of browser is in use by testing which objects are defined in the browser. Listing 8.2 shows this technique.

LISTING 8.2 Using Browser Detection for a Cross-Browser Solution

```
function getXMLHttpRequest()
{
    var request = false;
    if(window.XMLHttpRequest)
    {
        request = new XMLHttpRequest();
    } else {
        if(window.ActiveXObject)
        {
            try
            {
                request = new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch(err1)
            {
                try
                {
                    request =
new ActiveXObject("Microsoft.XMLHTTP");
                }
                catch(err2)
                {
                    request = false;
                }
            }
        }
    }
}
```



```

    }
    return request;
}

```

In this example we've used the test

```
if(window.XMLHttpRequest) { ... }
```

to determine whether XMLHttpRequest is a native object of the browser in use; if so, we use the constructor method

```
request = new XMLHttpRequest();
```

to create an instance of the XMLHttpRequest object; otherwise, we try creating a suitable ActiveX object as in the first example.

Whatever method you use to create an instance of the XMLHttpRequest object, you should be able to call this function like this:

```
var myRequest = getXMLHttpRequest();
```

NOTE: JavaScript also makes available a navigator object that holds information about the browser being used to view the page. Another method we could have used to branch our code is to use this object's appName property to find the name of the browser:

```
var myBrowser = navigator.appName;
```

This would return "Microsoft Internet Explorer" for IE.

Methods and Properties

Now that we have created an instance of the XMLHttpRequest object, let's look at some of the object's properties and methods, listed in Table 8.1.

TABLE 8.1 XMLHttpRequest Objects and Methods

PROPERTIES	DESCRIPTION
onreadystatechange	Determines which event handler will be called when the object's readyState property changes
readyState	Integer reporting the status of the request:
	0 = uninitialized
	1 = loading

TABLE 8.1 Continued

PROPERTIES	DESCRIPTION
	2 = loaded
	3 = interactive
	4 = completed
responseText	Data returned by the server in text string form
responseXML	Data returned by the server expressed as a document object
status	HTTP status code returned by server
statusText	HTTP reason phrase returned by server
METHODS	DESCRIPTION
abort()	Stops the current request
getAllResponseHeaders()	Returns all headers as a string
getResponseHeader(x)	Returns the value of header x as a string
open('method', 'URL', 'a')	specifies the HTTP method (for example, GET or POST), the target URL, and whether the request should be handled asynchronously (If yes, a='true'—the default; if no, a='false'.)
send(content)	Sends the request, optionally with POST data
setRequestHeader('x', 'y')	Sets a parameter and value pair x=y and assigns it to the header to be sent with the request

Over the next few lessons we'll examine how these methods and properties are used to create the functions that form the building blocks of Ajax applications.

For now, let's examine just a few of these methods.

The open() Method

The `open()` method prepares the `XMLHttpRequest` object to communicate with the server. You need to supply at least the two mandatory arguments to this method:

- First, specify which HTTP method you intend to use, usually GET or POST. (The use of GET and POST HTTP requests was discussed in Lesson 3, “Sending Requests Using HTTP.”)
- Next, the destination URL of the request is included as the second argument. If making a GET request, this URL needs to be suitably encoded with any parameters and their values as part of the URL.

CAUTION: A common mistake is to reference your domain as `mydomain.com` in a call made from `www.mydomain.com`. The two will be regarded as different by the JavaScript interpreter, and connection will not be allowed.

NOTE: A Boolean data type has only two possible values, 1 (or `true`) and 0 (or `false`).

For security reasons, the XMLHttpRequest object is allowed to communicate only with URLs within its own domain. An attempt to connect to a remote domain results in a “permission denied” error message.

Optionally you may include a third argument to the `send` request, a Boolean value to declare whether the request is being sent in asynchronous mode. If set to `false`, the request will not be sent in asynchronous mode, and the page will be effectively locked until the request is completed. The default value of `true` will be assumed if the parameter is omitted, and requests will then be sent asynchronously.

The `send()` Method

Having prepared the XMLHttpRequest using the `open()` method, you can send the request using the `send()` method. One argument is accepted by the `send()` function.

If your request is a GET request, the request information will be encoded into the destination URL, and you can then simply invoke the `send()` method using the argument `null`:

```
objectname.send(null);
```

However, if you are making a POST request, the content of the request (suitably encoded) will be passed as the argument.

```
objectname.setRequestHeader('Content-Type',
➡ 'application/x-www-form-urlencoded');
objectname.send(var1=value1&var2=value2);
```

In this case we use the `setRequestHeader` method to indicate what type of content we are including.

Summary

This lesson introduced the XMLHttpRequest object, the driving force behind any Ajax application, and illustrated how an instance of such an object is created both for Internet Explorer and for other, non-Microsoft browsers. We also briefly examined some of the object's properties and methods.

Following lessons will show how more of the object's methods and properties are used.

Talking with the Server

In this lesson you'll learn how to use the properties and methods of the XMLHttpRequest object to allow the object to send requests to and receive data from the server.

Sending the Server Request

Lesson 8, “The XMLHttpRequest Object,” discussed at some length the JavaScript XMLHttpRequest object and how an instance of it may be created in various different browsers.

Now that we have our XMLHttpRequest object, let's consider how to create and send server requests, and what messages we might expect to receive back from the server.

We're going to jump right in and first write some code using what you learned in Lesson 8 to create an XMLHttpRequest object called `myRequest`. We'll then write a JavaScript function called `callAjax()` to send an asynchronous request to the server using that object. Afterward we'll break down the code line by line to see what it's doing.

Listing 9.1 shows our prototype function to prepare and send an Ajax request using this object.

LISTING 9.1 Sending a Server Request

```
function getXMLHttpRequest()
{
    var req = false;
    try
    {
        req = new XMLHttpRequest(); /* e.g. Firefox */
    }
    catch(err1)
    {
        try
        {
            req = new ActiveXObject("Msxml2.XMLHTTP");
            /* some versions IE */
        }
        catch(err2)
        {
            try
            {
                req = new ActiveXObject("Microsoft.XMLHTTP");
                /* some versions IE */
            }
            catch(err3)
            {
                req = false;
            }
        }
    }
    return req;
}

var myRequest = getXMLHttpRequest();

function callAjax() {
    // declare a variable to hold some information
    // to pass to the server
    var lastname = 'Smith';
    // build the URL of the server script we wish to call
    var url = "myserverscript.php?surname=" + lastname;
    // ask our XMLHttpRequest object to open a
    // server connection
    myRequest.open("GET", url, true);
    // prepare a function responseAjax() to run when
    // the response has arrived
    myRequest.onreadystatechange = responseAjax;
    // and finally send the request
    myRequest.send(null);
}
```

TIP: Lines starting with `//` are treated as comments by JavaScript. You may use lines like these to document your code or add other useful notes, and your browser's JavaScript interpreter will ignore them when executing code instructions.

First, we need to create an instance of an XMLHttpRequest object and call it myRequest. You'll no doubt recognize the code for this from Lesson 8.

Next we'll look at the function callAjax().

The first line simply declares a variable and assigns a value to it:

```
var lastname = 'Smith';
```

This is the piece of data that our function intends to send to the server, as the value of a variable called surname that is required by our server-side script. In reality, of course, the value of such data would usually be obtained dynamically by handling a page event such as a mouse click or a keyboard entry, but for now this will serve as a simple example.

The server request we intend to make is a GET request, so we must construct a suitable target URL having our parameter and value pairs suitably coded on the end; the next line carries this out:

```
var url = "myserverscript.php?surname=" + lastname;
```

We dealt briefly with the open() method in Lesson 8. We use it in the next line to prepare our server request:

```
myRequest.open("GET", url, true);
```

This line specifies that we are preparing a GET request and passes to it the destination URL complete with the appended content of the GET request.

The third parameter, true, indicates that we want our request to be handled asynchronously. In this case it could have been omitted because the default value of true is assumed in such cases. However, it does no harm to include it for clarity.

Next, we need to tell our XMLHttpRequest object myRequest what it should do with the "progress reports" it will receive from the server. The XMLHttpRequest object has a property onreadystatechange that contains information about what JavaScript function should be called whenever the server status changes, and in the next line

```
myRequest.onreadystatechange = responseAjax;
```

we assign the function responseAjax() to do this job. We will write this function later in the lesson.

Dealing with the Browser Cache

All browsers maintain a so-called *cache* of visited web pages, a local record of page contents stored on the hard disk of the browser's computer. When you request a particular web page, the browser first tries to load the page from its cache, rather than submitting a new HTTP request.

Although this can sometimes be advantageous in terms of page load times, it creates a difficulty when trying to write Ajax applications. Ajax is all about talking to the server, not reloading information from cache; so when you make an asynchronous request to the server, a new HTTP request must be generated every time.

It is possible to add HTTP headers to the data returned by server-side routines, intended to tell the browser not to cache a particular page. Examples include

`"Pragma: no-cache"`

and

`"Cache-Control: must-revalidate"`

among others.

Unfortunately such strategies vary widely in their effectiveness. Different browsers have different cache handling strategies and support different header declarations, making it difficult to ensure that pages are not cached.

A commonly used trick to work around this problem involves the adding of a parameter with a random and meaningless value to the request data. In the case of a GET request, this necessitates adding a further parameter and value pair to the end of the URL.

If the random part of the URL is different each time, this effectively "fools" the browser into believing that it is to send the asynchronous request to an address not previously visited. This results in a new HTTP request being sent on every occasion.

Let's see how to achieve this. In JavaScript, you can generate random numbers using the `Math.random()` method of the native `Math()` object. Listing 9.2 contains a couple of changes to our `callAjax()` function.

NOTE: This appears to be more of a problem with IE than with the non-Microsoft browsers. Only GET requests are affected; POST requests are not cached.

LISTING 9.2 Dealing with the Browser Cache

```

function getXMLHttpRequest()
{
    var req = false;
    try
    {
        req = new XMLHttpRequest(); /* e.g. Firefox */
    }
    catch(err1)
    {
        try
        {
            req = new ActiveXObject("Msxml2.XMLHTTP");
            /* some versions IE */
        }
        catch(err2)
        {
            try
            {
                req = new ActiveXObject("Microsoft.XMLHTTP");
                /* some versions IE */
            }
            catch(err3)
            {
                req = false;
            }
        }
    }
    return req;
}

var myRequest = getXMLHttpRequest();

function callAjax() {
    // declare a variable to hold some information
    // to pass to the server
    var lastname = 'Smith';
    // build the URL of the server script we wish to call
    var url = "myserverscript.php?surname=" + lastname;
    // generate a random number
    var myRandom=parseInt(Math.random()*999999999);
    // ask our XMLHttpRequest object to open
    // a server connection
    myRequest.open("GET", url + "&rand=" + myRandom, true);
    // prepare a function responseAjax() to run when
    // the response has arrived

```

LISTING 9.2 Continued

```
myRequest.onreadystatechange = responseAjax;  
// and finally send the request  
myRequest.send(null);  
}
```

We can see from Listing 9.2 that the script will now generate a destination URL for our Ajax request that looks something like this:

```
myserverscript.php?surname=Smith&rand=XXXX
```

where XXXX will be some random number, thereby preventing the page from being returned from cache and forcing a new HTTP request to be sent to the server.

NOTE: Some programmers prefer to add the current timestamp rather than a random number. This is a string of characters derived from the current date and time. In the following example, the JavaScript `Date()` and `getTime()` methods of the native `Date()` object are used:

```
myRand= + new Date().getTime()
```

Monitoring Server Status

The `XMLHttpRequest` object contains mechanisms by which we can stay informed of the progress of our Ajax request and determine when the information returned by the server is ready to use in our application.

Let's now have a look at the relevant properties.

The `readyState` Property

The `readyState` property of the `XMLHttpRequest` object gives you information from the server about the current state of a request you have made. This property is monitored by the `onreadystatechange` property, and changes in the value of `readyState` cause `onreadystatechange` to become true and therefore cause the appropriate function (`responseAjax()` in our example) to be executed.

TIP: The function called on completion of the server request is normally referred to as the *callback function*.

readyState can take the following values:

- 0 = uninitialized
- 1 = loading
- 2 = loaded
- 3 = interactive
- 4 = completed

When a server request is first made, the value of readyState is set to zero, meaning uninitialized.

As the server request progresses, data begins to be loaded by the server into the XMLHttpRequest object, and the value of the readyState property changes accordingly, moving to 1 and then 2.

An object readyState value of 3, interactive, indicates that the object is sufficiently progressed so that certain interactivity with it is possible, though the process is not yet fully complete.

TIP: Not all of the possible values may exist for any given object. The object may “skip” certain states if they bear no relevance to the object’s content type.

When the server request has completed fully and the object is available for further processing, the value of readyState changes finally to 4.

In most practical cases, you should look for the readyState property to achieve a value of 4, at which point you can be assured that the server has finished its task and the XMLHttpRequest object is ready for use.

Server Response Status Codes

In addition to the readyState property, you have a further means to check that an asynchronous request has executed correctly: the HTTP server response status code.

HTTP responses were discussed in Lesson 3, “Sending Requests Using HTTP.” If you refer to Table 3.1 you’ll see that a response status code of 200 corresponds to an OK message from the server.

We’ll see how to test for this as we further develop our callback function.

The Callback Function

By now, then, you have learned how to create an instance of an XMLHttpRequest object, declare the identity of a callback function, and prepare and send an asynchronous server request. You also know which property tells you when the server response is available for use.

9: Talking with the Server

Let's look at our callback function, `responseAjax()`.

First, note that this function is called every time there is a change in the value of the `onreadystatechange` property. Usually, then, when this function is called, it is required to do absolutely nothing because the value of the `readyState` property has not yet reached 4 and we therefore know that the server request has not completed its processing.

We can achieve this simply by using a JavaScript `if` statement:

```
function responseAjax() {  
    // we are only interested in readyState of 4,  
    // i.e. "completed"  
    if(myRequest.readyState == 4) {  
        ... program execution statements ...  
    }  
}
```

In addition to checking that the server request has completed, we also want to check the HTTP response status code to ensure that it is equal to 200, indicating a successful response to our asynchronous HTTP request.

Referring quickly back to Table 8.1, we can see that our `XMLHttpRequest` object `myRequest` has two properties that report the HTTP status response. These are

`myRequest.status`

which contains the status response code, and

`myRequest.statusText`

containing the reason phrase.

We can employ these properties by using a further loop:

```
function responseAjax() {  
    // we are only interested in readyState of 4,  
    // i.e. "loaded"  
    if(myRequest.readyState == 4) {  
        // if server HTTP response is "OK"  
        if(myRequest.status == 200) {  
            ... program execution statements ...  
        } else {  
            // issue an error message for any  
            // other HTTP response  
            alert("An error has occurred: "  
➡+ myRequest.statusText);
```

```

    }
  }
}

```

This code introduces an `else` clause into our `if` statement. Any server status response other than 200 causes the contents of this `else` clause to be executed, opening an alert dialog containing the text of the reason phrase returned from the server.

Using the Callback Function

So how do we go about calling our `callAjax()` function from our HTML page? Let's see an example. Here's the code for a simplified form in an HTML page:

```

<form name='form1'>
Name: <input type='text' name='myname'><br>
Tel: <input type='text' name='telno'><br>
<input type='submit'>
</form>

```

We'll launch the function using the `onBlur` event handler of a text input field in a form:

```

<form name='form1'>
Name: <input type='text' name='myname'
➔onBlur='callAjax()'><br>
Tel: <input type='text' name='telno'><br>
<input type='submit'>
</form>

```

The `onBlur` event handler is activated when the user leaves the field in question. In this case, when the user leaves the field, `callAjax()` will be executed, creating an instance of the `XMLHttpRequest` object and making an asynchronous server request to `myserverscript.php?surname=Smith`

That doesn't sound very useful. However, what if we were to now make a slight change to the code of `callAjax()`?

```

function callAjax() {
// declare a variable to hold some
// information to pass to the server
var lastname = document.form1.myname.value;
...

```

Now we can see that, as the user leaves the form field `myname`, the value she had typed into that field would be passed to the server via our asynchronous request. Such a call may, for example, check a database to verify the existence of the named person, and if so return information to populate other fields on the form.

The result, so far as the user is concerned, is that she sees the remaining fields magically populated with data before submitting—or even completing—the form.

How we might use the returned data to achieve such a result is discussed in Lesson 10, “Using the Returned Data.”

Summary

This lesson looked at the ways in which our `XMLHttpRequest` object can communicate with the server, including sending asynchronous requests, monitoring the server status, and executing a callback function.

In Lesson 10, you will see how Ajax applications can deal with the data returned by the server request.

Using the Returned Data

In this lesson you will learn how to process the information returned from the server in response to an Ajax request.

The `responseText` and `responseXML` Properties

Lesson 9, “Talking with the Server,” discussed the server communications that allow you to send and monitor asynchronous server requests. The final piece of the Ajax jigsaw is the information returned by the server in response to a request.

This lesson discusses what forms that information can take, and how you can process it and use it in an application. We will use two of the `XMLHttpRequest` object’s properties, namely `responseText` and `responseXML`.

Table 8.1 listed several properties of the `XMLHttpRequest` object that we have yet to describe. Among these are the `responseText` and `responseXML` properties.

Lesson 9 discussed how we could use the `readyState` property of the `XMLHttpRequest` object to determine the current status of the `XMLHttpRequest` call. By the time our server request has completed, as detected by the condition `myRequest.readyState == 4` for our `XMLHttpRequest` object `myRequest`, then the two properties `responseText` and `responseXML` will respectively contain text and XML representations of the data returned by the server.

In this lesson you’ll see how to access the information contained in these two properties and apply each in an Ajax application.

The responseText Property

The `responseText` property tries to represent the information returned by the server as a text string.

Let's look again at the callback function prototype:

```
function responseAjax() {
  // we are only interested in readyState of 4, i.e.
  "loaded"
  if(myRequest.readyState == 4) {
    // if server HTTP response is "OK"
    if(myRequest.status == 200) {
      ... program execution statements ...
    } else {
      // issue an error message for any other HTTP ➡re-
      sponse
      alert("An error occurred: " +
myRequest.statusText);
    }
  }
}
```

TIP: If the `XMLHttpRequest` call fails with an error, or has not yet been sent, `responseText` will have a value `null`.

Let's add a program statement to the branch of the `if` statement that is executed on success, as in Listing 10.1.

LISTING 10.1 Displaying the Value of `responseText`

```
function responseAjax() {
  // we are only interested in readyState of 4,
  // i.e. "completed"
  if(myRequest.readyState == 4) {
    // if server HTTP response is "OK"
    if(myRequest.status == 200) {
      alert("The server said: "
➡+ myRequest.responseText);
    } else {
      // issue an error message for
      // any other HTTP response
      alert("An error has occurred: "
➡+ myRequest.statusText);
    }
  }
}
```

In this simple example, our script opens an alert dialog to display the text returned by the server. The line

```
alert("The server said: " + myRequest.responseText);
```

takes the text returned by the server-side routine and appends it to the string "The server said: " before presenting it in a JavaScript alert dialog.

Let's look at an example using a simple PHP file on the server:

```
<?php echo "Hello Ajax caller!"; ?>
```

A successful XMLHttpRequest call to this file would result in the `responseText` property containing the string `Hello Ajax caller!`, causing the callback function to produce the dialog shown in Figure 10.1.



FIGURE 10.1 Output generated by Listing 10.1.

CAUTION: The `responseText` property is read-only, so there's no point in trying to manipulate its value until that value has first been copied into another variable.

Because the `responseText` contains a simple text string, we can manipulate it using any of JavaScript's methods relating to strings. Table 10.1 includes some of the available methods.

TABLE 10.1 Some JavaScript String Manipulation Methods

METHOD	DESCRIPTION
<code>charAt(<i>number</i>)</code>	Selects the single character at the specified position within the string
<code>indexOf(<i>substring</i>)</code>	Finds the position where the specified substring starts
<code>lastIndexOf(<i>substring</i>)</code>	Finds the last occurrence of the substring within the string
<code>substring(<i>start</i>,<i>end</i>)</code>	Gets the specified part of the string
<code>toLowerCase()</code>	Converts the string to lowercase
<code>toUpperCase()</code>	Converts the string to uppercase

We'll be looking at how `responseText` may be used in real situations in Lesson 12, "Returning Data as Text," and Lesson 13, "AHAH—Asynchronous HTML and HTTP."

The `responseXML` Property

Now suppose that the PHP script we used on the server in the previous example had instead looked like Listing 10.2.

LISTING 10.2 A Server-Side Script to Return XML

```
<?php
header('Content-Type: text/xml');
echo "<?xml version='1.0' ?><greeting>
  Hello Ajax caller!</greeting>";
?>
```

Although this is a short script, it is worthwhile to look at it in some detail.

The first line inside the `<?php` and `?>` delimiters uses PHP's header instruction to add an HTTP header to the returned data.

The header returned is the parameter and value pair

```
Content-Type: text/xml
```

which announces to our `XMLHttpRequest` object to expect that the following data from the server will be formatted as XML.

The next line is a PHP echo statement that outputs this simple, but complete, XML document:

```
<?xml version="1.0" ?>
<greeting>
Hello Ajax caller!
</greeting>
```

NOTE: In PHP you need to escape any quotes that occur within a quoted string to ensure that the meaning of the statement is unambiguous. You do so using a backslash character, hence the PHP command

```
echo "<img src=\"picture.gif\">";
```

produces the output:

```

```

CAUTION: Make sure that your PHP script does not output anything—even white space characters such as spaces and line returns—prior to issuing a `header()` instruction; otherwise, an error will occur.

TIP: It is important to note that the `responseXML` property does not contain just a string that forms a text representation of the XML document, as was the case with the `responseText` property; instead, the entire data and hierarchical structure of the XML document has been stored as a DOM-compatible object.

When the server call is completed, we now find this XML document loaded into the `responseXML` property of our `XMLHttpRequest` object.

We can now access the content of the XML document via JavaScript's DOM methods and properties.

Another Useful JavaScript DOM Property

You will no doubt recall that we described some of these methods in Lesson 6, "A Brief Introduction to XML." Let's now examine one more of these methods, namely `getElementsByTagName()`.

The `getElementsByTagName()` Method

This useful method allows you to build a JavaScript array of all the elements having a particular tagname. You can then access elements of that array using normal JavaScript statements. Here's an example:

```
var myElements = object.getElementsByTagName('greeting');
```

This line creates the array `myElements` and populates it with all the elements with tagname `greeting`. As with any other array, you can find out the length of the array (that is, the number of elements having the declared tagname) by using the `length` property:

```
myElements.length
```

You can access a particular element individually if you want; the first occurring element with tagname `greeting` can be accessed as `myElements[0]`, the second (if there is a second) as `myElements[1]`, and so:

```
var theElement = myElements[0];
```

TIP: You could also access these individual array elements directly:

```
var theElement = object.getElementsByTagName  
    ➡('greeting')[0];
```

Parsing responseXML

Listing 10.3 gives an example of how we can use `getElementsByTagName()`, alongside some other methods discussed in Lesson 6, to return the text of our greeting in an alert dialog.

LISTING 10.3 Parsing responseXML using

```
getElementsByTagName()
function responseAjax() {
    // we are only interested in readyState
    // of 4, i.e. "completed"
    if(myRequest.readyState == 4) {
        // if server HTTP response is "OK"
        if(myRequest.status == 200) {
            var greetNode = http.responseXML
            ➤.getElementsByTagName("greeting")[0];
            var greetText = greetNode.childNodes[0]
            ➤.nodeValue;
            alert("Greeting text: " + greetText);
        } else {
            // issue an error message for
            // any other HTTP response
            alert("An error has occurred: "
            ➤+ myRequest.statusText);
        }
    }
}
```

After the usual checks on the values of the `readyState` and `status` properties, the code locates the required element from `responseXML` using the `getElementsByTagName()` method and then uses `childNodes[0].nodeValue` to extract the text content from this element, finally displaying the returned text in a JavaScript alert dialog.

Figure 10.2 shows the alert dialog, showing the text string recovered from the `<greeting>` element of the XML document.



FIGURE 10.2 Displaying the returned greeting.

Providing User Feedback

In web pages with traditional interfaces, it is clear to the user when the server is busy processing a request; the interface is effectively unusable while a new page is being prepared and served.

The situation is a little different in an Ajax application. Because the interface remains usable during an asynchronous HTTP request, it may not be apparent to the user that new information is expected from the server. Fortunately there are some simple ways to warn that a server request is in progress.

Recall that our callback function is called each time the value of `readyState` changes, but that we are only really interested in the condition `myRequest.readyState == 4`, which indicates that the server request is complete.

Let's refer again to Listing 10.3. For all values of `readyState` other than 4, the function simply terminates having done nothing. We can use these changes to the value of `readyState` to indicate to the user that a server request is progressing but has not yet completed. Consider the following code:

```
function responseAjax() {
    if(myRequest.readyState == 4) {
        if(myRequest.status == 200) {
            ... [success - process the server response] ...
        } else {
            ... [failed - report the HTTP error] ...
        }
    }
    } else {
        // if readyState has changed
        // but readyState <> 4
        ... [do something here to provide user feedback] ...
    }
}
```

A commonly used way to do this is to modify the contents of a page element to show something eye-catching, such as a flashing or animated graphic, while a request is being processed and then remove it when processing is complete.

The getElementById() Method

JavaScript's `getElementById()` method allows you to locate an individual document element by its `id` value. You can use this method in your user feedback routine to temporarily change the contents of a particular page element to provide the visual clue that a server request is in progress.

Suppose that we have, say, a small animated graphic file `anim.gif` that we want to display while awaiting information from the server. We want to display this graphic inside a `<div>` element within the HTML page. We begin with this `<div>` element empty:

```
<div id="waiting"></div>
```

Now consider the code of the callback function:

```
function responseAjax() {
    if(myRequest.readyState == 4) {
        document.getElementById('waiting').innerHTML = '';
        if(myRequest.status == 200) {
            ... [success - process the server response] ...
        } else {
            ... [failed - report the HTTP error] ...
        }
    }
    else { // if readyState has changed
           // but readyState <> 4
        document.getElementById('waiting')
        ➡.innerHTML = '';
    }
}
```

On each change in value of the property `readyState`, the callback function checks for the condition `readyState == 4`. Whenever this condition fails to be met, the `else` condition of the outer loop uses the `innerHTML` property to ensure that the page element with `id` `waiting` (our `<div>` element) contains an image whose source is the animated GIF. As soon as the condition `readyState == 4` is met, and we therefore know that the server request has concluded, the line

```
document.getElementById('waiting').innerHTML = '';
```

once more erases the animation.

We'll see this technique in action in Lesson 11, "Our First Ajax Application," when we create a complete Ajax application.

TIP: Elements within a page that have had `id` values declared are expected to each have a unique `id` value. This allows you to identify a unique element. Contrast this with the `class` attribute, which can be applied to any number of separate elements in a page and is more commonly used to set the display characteristics of a group of objects.

Summary

This lesson examined the last link in the Ajax chain: how to deal with server responses containing both text and XML information.

We also introduced a further JavaScript DOM method, `getElementsByTagName()`.

In the next lesson, the last in Part II, we use this knowledge along with that gained from earlier lessons, to construct a complete and working Ajax application.

Our First Ajax Application

In this lesson you will learn how to construct a complete and working Ajax application using the techniques discussed in previous lessons.

Constructing the Ajax Application

The previous lessons have introduced all the techniques involved in the design and coding of a complete Ajax application. In this lesson, we're going to construct just such an application.

Our first application will be simple in function, merely returning and displaying the time as read from the server's internal clock; nevertheless it will involve all the basic steps required for any Ajax application:

- An HTML document forming the basis for the application
- JavaScript routines to create an instance of the XMLHttpRequest object and construct and send asynchronous server calls
- A server-side routine (in PHP) to configure and return the required information
- A callback function to deal with the returned data and use it in the application

Let's get to it, starting with the HTML file that forms the foundation for our application.

The HTML Document

Listing 11.1 shows the code for our HTML page.

LISTING 11.1 The HTML Page for Our Ajax Application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤ Transitional//EN"
➤ "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Ajax Demonstration</title>
<style>
.displaybox {
width:150px;
background-color:#ffffff;
border:2px solid #000000;
padding:10px;
font:24px normal verdana, helvetica, arial, sans-serif;
}
</style>
</head>
<body style="background-color:#cccccc;
➤ text-align:center">

<h1>Ajax Demonstration</h1>
<h2>Getting the server time without page refresh</h2>
<form>
<input type="button" value="Get Server Time" />
</form>
<div id="showtime" class="displaybox"></div>

</body>
</html>
```

This is a simple HTML layout, having only a title, subtitle, button, and <div> element, plus some style definitions.

Figure 11.1 shows what the HTML page looks like.

TIP: In HTML the <div> ... </div> element stands for division and can be used to allow a number of page elements to be grouped together and manipulated in a block.

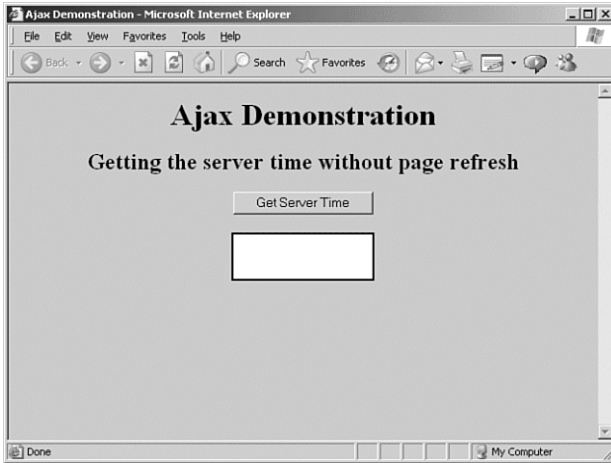


FIGURE 11.1 The HTML file of Listing 11.1.

Adding JavaScript

We can now add our JavaScript routines to the HTML page. We'll do so by adding them inside a `<script> ... </script>` container to the `<head>` section of the page.

TIP: Alternatively we could have added the routines in an external JavaScript file (ajax.js, say) and called this file from our document by using a statement like:

```
<script language="JavaScript" type="text/javascript"
  src="ajax.js"></script>
```

in the `<head>` section of the document.

The XMLHttpRequest Object

First, let's add our function to create our XMLHttpRequest object:

```
function getXMLHttpRequest() {
  try {
    req = new XMLHttpRequest();
  } catch(err1) {
    try {
      req = new ActiveXObject("Msxml2.XMLHTTP");
```

```
} catch (err2) {  
    try {  
        req = new XMLHttpRequest("Microsoft.XMLHTTP");  
    } catch (err3) {  
        req = false;  
    }  
}  
}  
return req;  
}
```

It's now a simple matter to create our XMLHttpRequest object, which on this occasion we're going to call http:

```
var http = getXMLHttpRequest();
```

The Server Request

Now we need a function to construct our server request, define a callback function, and send the request to the server. This is the function that will be called from an event handler in the HTML page:

```
function getServerTime() {  
    var myurl = 'telltimeXML.php';  
    myRand = parseInt(Math.random()*9999999999999999);  
    // add random number to URL to avoid cache problems  
    var modurl = myurl+"?rand="+myRand;  
    http.open("GET", modurl, true);  
    // set up the callback function  
    http.onreadystatechange = useHttpResponse;  
    http.send(null);  
}
```

Once again we have added a parameter with a random value to the URL to avoid any cache problems. Our callback function is named `useHttpResponse` and is called each time a change is detected in the value of `http's readyState` property.

Our PHP Server-Side Script

Before explaining the operation of the callback function, we need to refer to the code of the simple PHP server routine `telltimeXML.php`, shown in Listing 11.2.

LISTING 11.2 telltimeXML.php

```

<?php
header('Content-Type: text/xml');
echo "<?xml version=\"1.0\" ?><clock1><timenow>"
➔ .date('H:i:s')."</timenow></clock1>";
?>

```

This short program reports the server time using PHP's `date()` function. The argument passed to this function defines how the elements of the date and time should be formatted. Here we've ignored the date-related elements completely and asked for the time to be returned as Hours:Minutes:Seconds using the 24-hour clock.

Our server script returns an XML file in the following format:

```

<?xml version="1.0" ?>
<clock1>
  <timenow>
    XX:XX:XX
  </timenow>
</clock1>

```

with `XX:XX:XX` replaced by the current server time. We will use the callback function to extract this time information and display it in the `<div>` container of the HTML page.

The Callback Function

Here is the code for the callback function `useHttpResponse`:

```

function useHttpResponse() {
  if (http.readyState == 4) {
    if(http.status == 200) {
      var timeValue = http.responseXML
➔ .getElementsByTagName("timenow")[0];
      document.getElementById('showtime').innerHTML
➔ = timeValue.childNodes[0].nodeValue;
    }
    } else {
      document.getElementById('showtime').innerHTML
➔ = '';
    }
  }
}

```

Once again we have used the `getElementsByTagName` method, this time to select the `<timenow>` element of the XML data, which we have stored in a variable `timeValue`. However, on this occasion we're not

going to display the value in an alert dialog as we did in Lesson 10, “Using the Returned Data.”

This time we want instead to use the information to update the contents of an element in the HTML page. Note from Listing 11.1 how the `<div>` container is defined in our HTML page:

```
<div id="showtime" class="displaybox"></div>
```

In addition to the `class` declaration (which is used in the `<style>` definitions to affect how the `<div>` element is displayed), we see that there is also defined an `id` (identity) for the container, with a value set to `showtime`.

Currently the `<div>` contains nothing. We want to update the content of this container to show the server time information stored in `timeValue`. We do so by selecting the page element using JavaScript’s `getElementById()` method, which we met in Lesson 10. We’ll then use the JavaScript `innerHTML` property to update the element’s contents:

```
document.getElementById('showtime').innerHTML  
➡ = timeValue.childNodes[0].nodeValue;
```

Employing Event Handlers

Finally, we must decide how the server requests will be triggered. In this case we shall slightly edit the HTML document to use the `onClick()` event handler of the `<button>` object:

```
<input type="button" value="Get Server Time"  
➡ onClick="getServerTime()">
```

This will correctly deal with the occasion when the Get Server Time button is clicked. It does, however, leave the `<div>` element empty when we first load the page.

To overcome this little problem, we can use the `onLoad()` event handler of the page’s `<body>` element:

```
<body style="background-color:#cccccc"  
➡ onLoad="getServerTime()">
```

This event handler fires as soon as the `<body>` area of the page has finished loading.

Putting It All Together

Listing 11.3 shows the complete client-side code for our Ajax application.

LISTING 11.3 The Complete Ajax Application

```
<html>
<head>
<title>Ajax Demonstration</title>
<style>
.displaybox {
width:150px;
background-color:#ffffff;
border:2px solid #000000;
padding:10px;
font:24px normal verdana, helvetica, arial, sans-serif;
}
</style>
<script language="JavaScript" type="text/javascript">
function getXMLHttpRequest() {
try {
req = new XMLHttpRequest();
} catch(err1) {
try {
req = new ActiveXObject("Msxml2.XMLHTTP");
} catch (err2) {
try {
req = new ActiveXObject("Microsoft.XMLHTTP");
} catch (err3) {
req = false;
}
}
}
return req;
}

var http = getXMLHttpRequest();

function getServerTime() {
var myurl = 'telltimeXML.php';
myRand = parseInt(Math.random()*9999999999999999);
var modul1 = myurl+"?rand="+myRand;
http.open("GET", modul1, true);
http.onreadystatechange = useHttpResponse;
http.send(null);
}
```

LISTING 11.3 Continued

```

}

function useHttpResponse() {
    if (http.readyState == 4) {
        if(http.status == 200) {
            var timeValue = http.responseXML
            ↪.getElementsByTagName("timenow")[0];
            document.getElementById('showtime').innerHTML
            ↪ = timeValue.childNodes[0].nodeValue;
        }
        } else {
            document.getElementById('showtime').innerHTML
            ↪ = '';
        }
    }
</script>
</head>
<body style="background-color:#cccccc"
↪ onLoad="getServerTime()">
<center>
<h1>Ajax Demonstration</h1>
<h2>Getting the server time without page refresh</h2>
<form>
<input type="button" value="Get Server Time"
↪ onClick="getServerTime()">
</form>
<div id="showtime" class="displaybox"></div>
</center>
</body>
</html>

```

Loading the page into our browser, we can see that the server time is displayed in the <div> container, indicating that the onLoad event handler for the <body> of the page has fired when the page has loaded.

User Feedback

Note also that we have provided user feedback via the line

```

document.getElementById('showtime').innerHTML
↪ = '';

```


which executes on each change to the value `readyState` until the condition

```
readyState == 4
```

is satisfied. This line loads into the time display element an animated GIF with a rotating pattern, indicating that a server request is in progress, as shown in Figure 11.2. This technique was described in more detail in Lesson 10.

TIP: If you have a fast server and a good Internet connection, it may be difficult to see this user feedback in action because the time display is updated virtually instantaneously. To demonstrate the operation of the animated GIF image, we can slow down the server script to simulate the performance of a more complex script and/or an inferior connection, by using PHP's `sleep()` command:

```
<?php
header('Content-Type: text/xml');
sleep(3);
echo "<?xml version='1.0' ?><clock1><timenow>"
    .date('H:i:s')."</timenow></clock1>";
?>
```

The line

```
sleep(x);
```

Forces the server to pause program execution for *x* seconds.

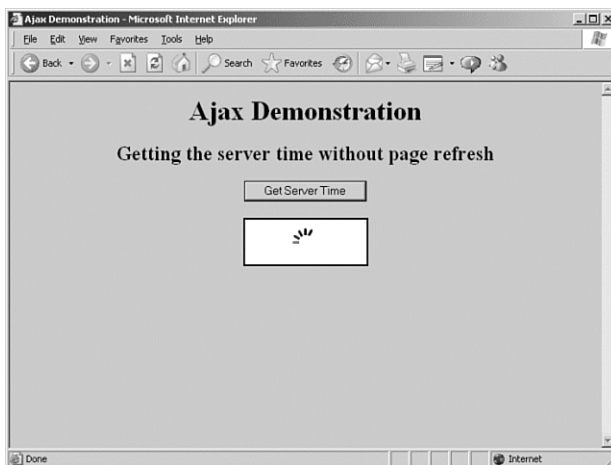


FIGURE 11.2 An animated image provides user feedback.

Now, each time we click on the Get Server Time button, the time display is updated. Figure 11.3 shows the completed application.

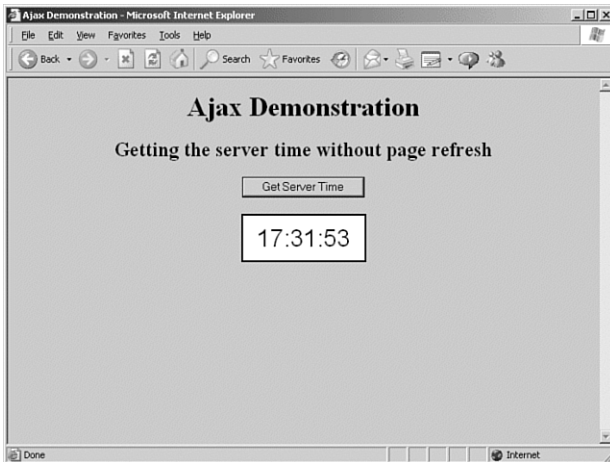


FIGURE 11.3 Our completed Ajax application.

Summary

In this lesson, we constructed a simple yet complete Ajax application that does the following:

- Creates an instance of the XMLHttpRequest object
- Reacts to JavaScript event handlers built into an HTML page
- Constructs and sends asynchronous server requests
- Parses XML received from the server using JavaScript DOM methods
- Provides user feedback that a request is in progress
- Updates the displayed page with the received data

This completes Part II of the book. Part III, "More Complex Ajax Technologies," investigates some more advanced Ajax techniques.

Returning Data as Text

In this lesson you will learn some more techniques for using the `responseText` property to add functionality to Ajax applications.

Getting More from the `responseText` Property

The lessons of Part II, “Introducing Ajax,” discussed the individual components that make Ajax work, culminating in a complete Ajax application. In Part III, “More Complex Ajax Technologies,” each lesson examines how you can extend what you know to develop more sophisticated Ajax applications.

For this lesson, we’ll look a little more closely at the `responseText` property of the `XMLHttpRequest` object and see how we can give our application some extra functionality via its use.

As you have seen in previous lessons, the `XMLHttpRequest` object provides two properties that contain information received from the server, namely `responseText` and `responseXML`. The former presents the calling application with the server data in string format, whereas the latter provides DOM-compatible XML that can be parsed using JavaScript methods.

Although the `responseXML` property allows you to carry out some sophisticated programming tasks, much can be achieved just by manipulating the value stored in the `responseText` property.

Returning Text

The term *text* is perhaps a little misleading. The `responseText` property contains a character string, the value of which you can assign to a JavaScript variable via a simple assignment statement:

```
var mytext = http.responseText;
```

There is no rule saying that the value contained in such a string must be legible text; in fact, the value can contain complete gibberish provided that the string contains only characters that JavaScript accepts in a string variable.

This fact allows a degree of flexibility in what sorts of information you can transfer using this property.

Using Returned Text Directly in Page Elements

Perhaps the simplest example is to consider the use of the value held in `responseText` in updating the textual part of a page element, say a `<div>` container. In this case you may simply take the returned string and apply it to the page element in question.

Here's a simple example. The following is the HTML code for an HTML page that forms the basis for an Ajax application:

```
<html>
<head>
  <title>My Ajax Application</title>
</head>
<body>
  Here is the text returned by the server:<br />
  <div id="myPageElement"></div>
</body>
</html>
```

Clearly this is a simple page that, as it stands, would merely output the line "Here is the text returned by the server:" and nothing else.

Now suppose that we add to the page the necessary JavaScript routines to generate an instance of a `XMLHttpRequest` object (in this case called `http`) and make a server request in response to the `onLoad()` event handler of the page's `<body>` Element. Listing 12.1 shows the source code for the revised page.

LISTING 12.1 A Basic Ajax Application Using the responseText Property

```

<html>
<head>
<title>My Ajax Application</title>
<script Language="JavaScript">
function getXMLHttpRequest() {
try {
req = new XMLHttpRequest();
} catch(err1) {
try {
req = new ActiveXObject("Msxml2.XMLHTTP");
} catch (err2) {
try {
req = new ActiveXObject("Microsoft.XMLHTTP");
} catch (err3) {
req = false;
}
}
}
return req;
}

var http = getXMLHttpRequest();

function getServerText() {
var myurl = 'textserver.php';
myRand = parseInt(Math.random()*9999999999999999);
var modurl = myurl+"?rand="+myRand;
http.open("GET", modurl, true);
http.onreadystatechange = useHttpResponse;
http.send(null);
}

function useHttpResponse() {
if (http.readyState == 4) {
if(http.status == 200) {
var mytext = http.responseText;
document.getElementById('myPageElement')
➡.innerHTML = mytext;
}
} else {
document. getElementById('myPageElement')
➡.innerHTML = "";
}
}
}

```

LISTING 12.1 Continued

```
</head>
<body onLoad="getServerText()">
Here is the text returned by the server:<br>
<div id="myPageElement"></div>
</body>
</html>
```

Most, and probably all, of this code will be familiar from previous lessons. The part that interests us here is the callback function `useHttpResponse()`, which contains these lines:

```
var mytext = http.responseText;
document.getElementById('myPageElement').innerHTML =
mytext;
```

Here we have simply assigned the value received in `responseText` to become the content of our chosen `<div>` container.

Running the preceding code with the simple server-side script

```
<?php
echo "This is the text from the server";
?>
```

produces the screen display of Figure 12.1.

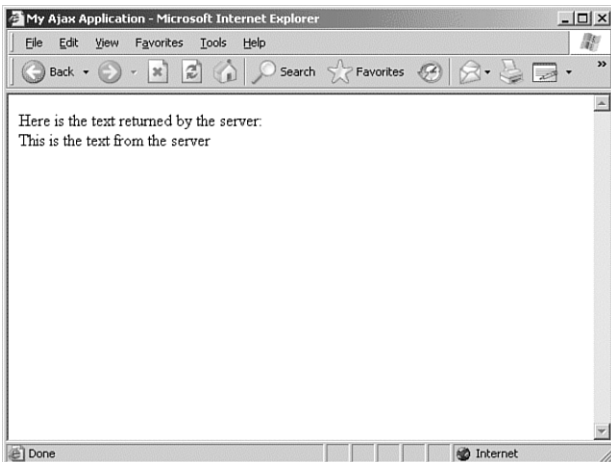


FIGURE 12.1 Displaying text in a page element via `responseText`.

Including HTML in responseText

Now let's modify the code from the preceding example.

As you know from previous lessons, HTML markup is entirely composed of tags written using text characters. If the value contained in the responseText property is to be used for modifying the display of the page from which the server request is being sent, there is nothing to stop us having our server script include HTML markup in the information it returns.

Suppose that we once again use the code of Listing 12.1 but with a modified server script:

```
<?php
echo "<h3>Returning Formatted Text</h3>";
echo "<hr />";
echo "We can use HTML to <strong>format</strong>
➤ text before we return it!";
?>
```

Figure 12.2 shows the resulting browser display.

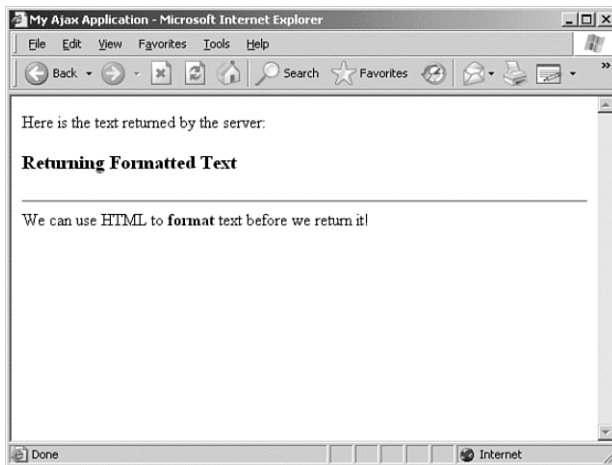


FIGURE 12.2 Display showing HTML formatted at the server.

As a slightly more involved example, consider the case where the server script generates more complex output. We want our application to take this server output and display it as the contents of a table.

12: Returning Data as Text

This time we'll use our server-side PHP script to generate some tabular information:

```
<?php
$days = array('Monday', 'Tuesday', 'Wednesday',
    ➤ 'Thursday', 'Friday', 'Saturday', 'Sunday');
echo "<table border='2'>";
echo "<tr><th>Day Number</th><th>Day Name</th></tr>";
for($i=0;$i<7;$i++)
{
    echo "<tr><td>".$i."</td><td>".$days[$i]."</td></tr>";
}
echo "</table>";
?>
```

Once again using the code of Listing 12.1 to call the server-side script via XMLHttpRequest, we obtain a page as displayed in Figure 12.3.

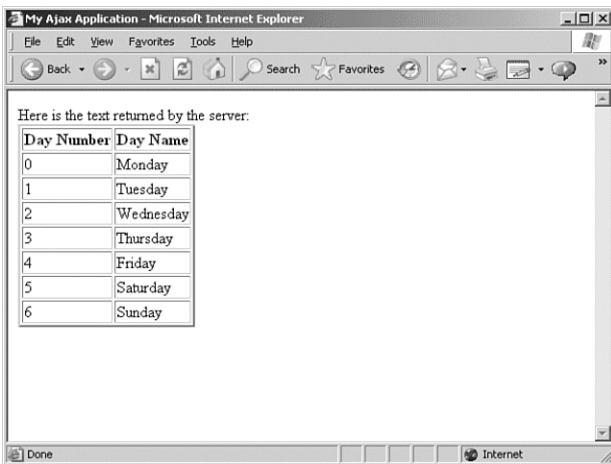


FIGURE 12.3 Returning more complex HTML.

More Complex Formatted Data

So far we have demonstrated ways to return text that may be directly applied to an element on a web page. So far, so good. However, if you are willing to do a little more work in JavaScript to manipulate the returned data, you can achieve even more.

Provided that the server returns a string value in the `responseText` property of the `XMLHttpRequest` object, you can use any data format you may devise to encode information within it.

NOTE: Note the use of the PHP `sizeof()` function to determine the number of items in the array. In PHP, as in JavaScript, array keys are numbered from 0 rather than 1.

Consider the following server-side script, which uses the same data array as in the previous example:

```
<?php
$days = array('Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday', 'Sunday');
$numdays = sizeof($days);
for($i=0;$i<($numdays - 1);$i++)
{
    echo $days[$i]."|";
}
echo $days[$numdays-1];
?>
```

The string returned in the `responseText` property now contains the days of the week, separated—or *delimited*—by the pipe character `|`. If we copy this string into a JavaScript variable `mystring`,

```
var mystring = http.responseText;
```

we will find that the variable `mystring` contains the string

```
Monday|Tuesday|Wednesday|Thursday|Friday|Saturday|Sunday
```

We may now conveniently divide this string into an array using JavaScript's `split()` method:

```
var results = http.responseText.split("|");
```

TIP: The JavaScript `split()` method slices up a string, making each cut wherever in the string it locates the character that it has been given as an argument. That character need not be a pipe; popular alternatives are commas or slashes.

We now have a JavaScript array `results` containing our data:

```
results[0] = 'Monday'
results[1] = 'Tuesday'
etc...
```

Rather than simply displaying the received data, we now can use it in JavaScript routines in any way we want.

TIP: For complex data formats, XML may be a better way to receive and handle data from the server. However, it is remarkable how much can be done just by using the `responseText` property.

Summary

With little effort, the `XMLHttpRequest` object's `responseText` property can be persuaded to do more than simply return some text to display in a web page.

For all but the most complex data formats, it may prove simpler to manipulate `responseText` than to deal with the added complexity of XML.

In this lesson you saw several examples of this technique, ranging from the simple update of text content within a page element, to the manipulation of more complex data structures.

AHAH—Asynchronous HTML and HTTP

In this lesson you will learn how to use AHAH (Asynchronous HTML and HTTP) to build Ajax-style applications without using XML.

Introducing AHAH

You saw in Lesson 12, “Returning Data as Text,” just how much can be achieved with an Ajax application without using any XML at all. Many tasks, from simply updating the text on a page to dealing with complicated data structures, can be carried out using only the text string whose value is returned in the XMLHttpRequest object’s `responseText` property.

It is possible to build complete and useful applications without any XML at all. In fact, the term *AHAH (Asynchronous HTML and HTTP)* has been coined for just such applications.

This lesson takes the concepts of Lesson 12 a little further, examining in more detail where—and how—AHAH can be applied.

NOTE: This technique, a kind of subset of Ajax, has been given various acronyms. These include AHAH (asynchronous HTML and HTTP), JAH (Just Asynchronous HTML), and HAJ (HTML And JavaScript). In this book we’ll refer to it as AHAH.

Why Use AHAH Instead of Ajax?

There is no doubt that XML is an important technology with diverse and powerful capabilities. For complex Ajax applications with sophisticated data structures it may well be the best—or perhaps the only—option. However, using XML can sometimes complicate the design of an application, including:

- Work involved in the design of custom schemas for XML data.
- Cross-browser compatibility issues when using JavaScript's DOM methods.
- Performance may suffer from having to carry out processor-intensive XML parsing.

Using AHAH can help you avoid these headaches, while offering a few more advantages too:

- Easy reworking of some preexisting web pages.
- HTML can be easier to fault-find than XML.
- Use of CSS to style the returned information, rather than having to use XSLT.

NOTE: *XSLT* is a transformation language used to convert XML documents into other formats—for example, into HTML suitable for a browser to display.

In the following sections we'll package our AHAH scripts into a neat external JavaScript file that we can call from our applications.

Creating a Small Library for AHAH

The Ajax applications examined in the last couple of lessons, although complete and functional, involved embedding a lot of JavaScript code into our pages. As you have seen, each application tends to contain similar functions:

- A method to create an instance of the `XMLHttpRequest` object, configure it, and send it
- A callback function to deal with the returned text contained in the `responseText` property

You can abstract these functions into simple JavaScript function calls, especially in cases where you simply want to update a single page element with a new value returned from the server.

Introducing myAJAXlib.js

Consider Listing 13.1; most of this code will be instantly recognizable to you.

LISTING 13.1 myAJAXlib.js

```
function callAJAX(url, pageElement, callMessage) {
    document.getElementById(pageElement)
    ➔.innerHTML = callMessage;
    try {
        req = new XMLHttpRequest(); /* e.g. Firefox */
    } catch(e) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
        /* some versions IE */
        } catch (e) {
            try {
                req = new ActiveXObject("Microsoft.XMLHTTP");
            /* some versions IE */
            } catch (E) {
                req = false;
            }
        }
    }
    req.onreadystatechange
    ➔ = function() {responseAJAX(pageElement)};
    req.open("GET",url,true);
    req.send(null);
}

function responseAJAX(pageElement) {
    var output = '';
    if(req.readyState == 4) {
        if(req.status == 200) {
            output = req.responseText;
            document.getElementById(pageElement)
            ➔.innerHTML = output;
        }
    }
}
```

The function `callAJAX()` encapsulates the tasks of creating an instance of the `XMLHttpRequest` object, declaring the callback function, and sending the request.

Note that instead of simply declaring

```
req.onreadystatechange = responseAHAH;
```

we instead used the JavaScript construct

```
req.onreadystatechange  
= function() {responseAHAH(pageElement)};
```

This type of declaration allows us to pass an argument to the declared function, in this case identifying the page element to be updated.

`callAHAH()` also accepts an additional argument, `callMessage`. This argument contains a string defining the content that should be displayed in the target element while we await the outcome of the server request. This provides a degree of feedback for the user, indicating that something is happening on the page. In practice this may be a line of text, such as

```
'Updating page; please wait a moment ....'
```

Once again, however, you may choose to embed some HTML code into this string. Using an animated GIF image within an `` element provides an effective way of warning a user that a process is underway.

The callback function `responseAHAH()` carries out the specific task of applying the string returned in the `responseText` property to the `innerHTML` property of the selected page element `pageElement`:

```
output = req.responseText;  
document.getElementById(pageElement).innerHTML = output;
```

This code has been packaged into a file named `myAHAHlib.js`, which you can call from an HTML page, thus making the functions available to your AHAH application. The next section shows some examples of its use.

Using myAHAHlib.js

In Lesson 4, “Client-Side Coding Using JavaScript,” we encountered the concept of JavaScript functions being located in an external file that is referred to within our page.

That’s how we’ll use our new file `myAHAHlib.js`, using a statement in this form:

```
<SCRIPT language="JavaScript" SRC="myAHAHlib.js"></SCRIPT>
```

We will then be at liberty to call the functions within the script whenever we want.

The following is the skeleton source code of such an HTML page:

```
<html>
<head>
<title>Another Ajax Application</title>
<SCRIPT language="JavaScript" SRC="myAHAHlib.js"></SCRIPT>
</head>
<body>
<form>
<input type="button" onClick=
➡ "callAAHAH('serverscript.php?parameter=x',
➡ 'displaydiv', 'Please wait - page updating ...')">
This is the place where the server response
will be posted:<br>
<div id="displaydiv"></div>
</form>
</body>
</html>
```

In this simple HTML page, a button element is used to create the event that causes the callAAHAH() method to be called. This method places the text string 'Please wait - page updating ...'

in the <div> element having id displaydiv and sends the asynchronous server call to the URL serverscript.php?parameter=x.

When responseAAHAH() detects that the server has completed its response, the <div> element's content is updated using the value stored in responseText; instead of showing the "please wait" message, the <div> now displays whatever text the server has returned.

Applying myAHAHlib.js in a Project

We can demonstrate these techniques with a further simple Ajax application. This time, we'll build a script to grab the 'keywords' metatag information from a user-entered URL.

NOTE: *Metatags* are optional HTML container elements in the <head> section of an HTML page. They contain data about the web page that is useful to search engines and indexes in deciding how the page's content should be classified. The 'keywords' metatag, where present, typically contains a comma-separated list of words with meanings relevant to the site content. An example of a 'keywords' metatag might look like this:

```
<meta name="keywords" content="programming, design,
➡ development, Ajax, JavaScript, XMLHttpRequest,
➡ script">
```

Listing 13.2 shows the HTML code.

LISTING 13.2 getkeywords.html

```
<html>
<head>
<title>A 'Keywords' Metatag Grabber</title>
<SCRIPT language="JavaScript" SRC="myAHAHlib.js">
</SCRIPT>
</head>
<body>
<script type="text/javascript" src="ahahLib.js">
</script>
<form>
<table>
<tr>
<td>
URL: http://
</td>

<td>
<input type="text" id="myurl" name="myurl" size=30>
<input type="button" onclick =
➤"callAHAH('keywords.php?url='+document
➤.getElementById('myurl').value,'displaydiv',
➤'Please wait; loading content ...')" value="Fetch">
</td>
</tr>
<tr><td colspan=2 height=50 id="displaydiv"></td></tr>
</table>
</form>
</body>
</html>
```

Finally, consider the server-side script:

```
<?php
$tags = @get_meta_tags('http://'.$url);
$result = $tags['keywords'];
if(strlen($result) > 0)
{
    echo $result;
} else {
    echo "No keywords metatag is available";
}
?>
```


TIP: The @ character placed before a PHP method tells the PHP interpreter not to output an error message if the method should encounter a problem during execution. We require it in this instance because not all web pages contain a 'keywords' metatag; in the cases where none exists, we would prefer the method to return an empty string so that we can add our own error handling.

We present the selected URL to the PHP method `get_meta_tags()` as an argument:

```
$tags = @get_meta_tags('http://'.$url);
```

This method is specifically designed to parse the metatag information from HTML pages in the form of an associative array. In this script, the array is given the name `$tags`, and we can recover the 'keywords' metatag by examining the array entry `$tags['keywords']`; we can then check for the presence or absence of a 'keywords' metatag by measuring the length of the returned string using PHP's `strlen()` method.

When the file `getkeywords.html` is first loaded into the browser, we are presented with the display shown in Figure 13.1.

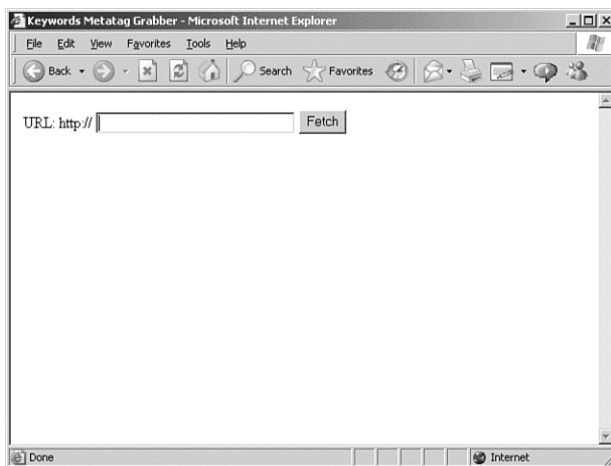


FIGURE 13.1 The browser display after first loading the application.

Here we are invited to enter a URL. When we then click on the Fetch button, `callTAHAH()` is executed and sends our chosen URL as a parameter to the server-side script. At the same time, the message "Please wait; loading content ..." is placed in the `<div>` container. Although possibly only visible for a fraction of a second, we now have a display such as that shown in Figure 13.2.

13: AHAH—Asynchronous HTML and HTTP

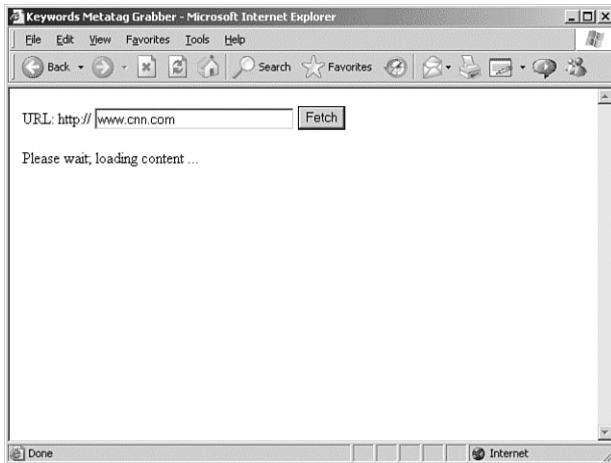


FIGURE 13.2 Awaiting the server response.

Finally, when the server call has concluded, the contents of the `responseText` property are loaded into the `<div>` container, producing the display of Figure 13.3.

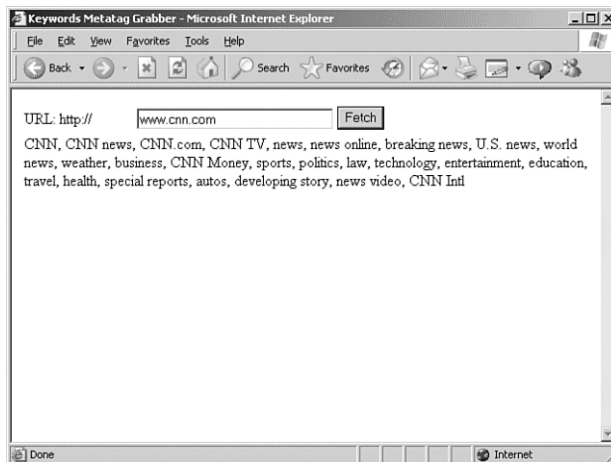


FIGURE 13.3 The keywords are successfully returned.

Extending myAHAHlib.js

As it stands, myAHAHlib.js is a simple implementation of AHAH. There are many ways it could be improved and extended, depending on how it is to be used. Rather than cover these in this lesson, we'll leave these for your own experimentation. Here's a few suggestions to get you started:

- Currently only GET requests are supported. How might the functions be modified to allow POST requests too?
- Much of the user feedback discussed in Lesson 11, "Our First Ajax Application," is not yet implemented in `responseAHAH()`.
- Is it possible for `callAHAH()` to be modified to accept an array of page elements for updating and (with the aid of a suitable server-side script) process them all at once?

TIP: One option we haven't yet considered is the idea of passing back JavaScript code within `responseText`. Because JavaScript source code (like everything else in an HTML page) is made up of statements written in plain text, you can return JavaScript source from the server in the `responseText` property.

You can then execute this JavaScript code using JavaScript's `eval()` method:

```
eval(object.responseText);
```

Consider the situation where your server script returns the string:

```
"alert('Hello World!');"
```

In this case the `eval()` method would execute the content as a JavaScript statement, creating a dialog saying 'Hello World!' with an OK button.

Summary

It will hopefully have become clear, in the course of this lesson and Lesson 12, that Ajax can achieve a lot of functionality without using any XML at all.

By carefully using combinations of client-side coding in JavaScript and server-side scripting in your chosen language, you can create data schemes of high complexity.

In simpler applications, where all you want to do is update the text of page elements, the `XMLHttpRequest` object's functionality may be abstracted into a JavaScript function library and called from an HTML page via straightforward methods.

For some tasks, however, you need to leverage the power of XML. We'll look at this subject in Lesson 14, "Returning Data as XML."

Returning Data as XML

In this lesson you will learn to use XML data returned from the server via the `responseXML` property of the `XMLHttpRequest` object.

Adding the “x” to Ajax

Lesson 12, “Returning Data as Text,” and Lesson 13, “AHAH—Asynchronous HTML and HTTP,” dealt at some length with the string value contained in `responseText` and looked at several techniques for using this information in applications. These examples ranged from simple updates of page element text to applications using more sophisticated data structures encoded into string values that can be stored and transferred in the `responseText` property.

The *X* in Ajax does, of course, stand for XML, and there are good reasons for using the power of XML in your applications. This is particularly true when you need to use highly structured information and/or perform complex translations between different types of data representation.

As discussed previously, the `XMLHttpRequest` object has a further property called `responseXML`, which can be used to transfer information from the server via XML, rather than in text strings.

You saw in Lesson 11, “Our First Ajax Application,” how JavaScript’s document object model (DOM) methods can help you process this XML information. This lesson looks at these techniques in a little more detail and hopefully gives you a taste of what Ajax applications can achieve when leveraging the power of XML.

The responseXML Property

Whereas the `responseText` property of the `XMLHttpRequest` object contains a string, `responseXML` can be treated as if it were an XML document.

NOTE: Like the `responseText` property, the value stored in `responseXML` is read-only, so you cannot write directly to this property; to manipulate it you must copy the value to another variable:

```
var myobject = http.responseXML;
```

CAUTION: You need to make sure that your server presents valid and well-formed XML to be returned via the `responseXML` property. In situations where XML cannot be correctly parsed by the `XMLHttpRequest` object, perhaps due to well-formedness errors or problems with unsupported character encoding, the content of the `responseXML` is unpredictable and also likely to be different in different browsers.

The complete structure and data contained in the XML document can now be made available by using JavaScript's DOM methods. Later in the lesson we'll demonstrate this with another working Ajax application, but first let's revisit the JavaScript DOM methods and introduce a few new ones.

More JavaScript DOM Methods

You met some of the JavaScript DOM methods, such as `getElementById` and `getElementsByTagName`, in previous lessons. In those cases, we were mostly concerned with reading the values of the nodes to write those values into HTML page elements.

This lesson looks at the DOM methods that can be used to actually create elements, thereby changing the structure of the page.

The Document Object Model can be thought of as a treelike structure of nodes. As well as reading the values associated with those nodes, you can create and modify the nodes themselves, thereby changing the structure and content of your document.

To add new elements to a page, you need to first create the elements and then attach them to the appropriate point in your DOM tree. Let's look at a simple example using the following HTML document:

```
<html>
<head>
  <title>Test Document</title>
</head>
<body>
  We want to place some text here:<br />
  <div id="displaydiv"></div>
</body>
</html>
```

In this example, we want to add the text “Hello World!” to the `<div>` container in the document body. We’ll put our JavaScript routine into a function that we’ll call from the body’s `onLoad()` event handler.

First, we’ll use the JavaScript DOM method `createTextNode()` to, well, create a text node:

```
var textnode = createTextNode('Hello World!');
```

We now need to attach `textnode` to the DOM tree of the document at the appropriate point.

You first learned about child nodes in Lesson 4, “Client-Side Coding Using JavaScript”; hopefully, you recall that nodes in a document are said to have *children* if they contain other document elements. JavaScript has an `appendChild()` method, which allows us to attach our new text node to the DOM tree by making it a child node of an existing document node.

NOTE: Compare this DOM-based method of writing content to the page with the `innerHTML` method used in the project in Lesson 11.

In this case, we want our text to be inside the `<div>` container having the `id displaydiv`:

```
var textnode = document.createTextNode('Hello World!');
document.getElementById('displaydiv').appendChild(textnode);
```

Let’s look at the complete source of the page, after wrapping up this JavaScript code into a function and adding the `onLoad()` event handler to execute it:

NOTE: If you display the source code of this document in your browser, you won’t see the ‘Hello World!’ text inside the `<div>` container. The browser builds its DOM representation of the HTML document and then uses that model to display the page. The amendments made by your code are made to the DOM, not to the document itself.

```
<html>
<head>
  <title>Test Document</title>
  <script Language="JavaScript">
    function hello()
    {
      var textnode = document.createTextNode('Hello World!');
      document.getElementById('displaydiv').appendChild(textnode);
    }
  </script>
</head>
<body onload="hello()">
  We want to place some text here:<br />
  <div id="displaydiv"></div>
</body>
</html>
```

Figure 14.1 shows the browser display after loading this page.

14: Returning Data as XML

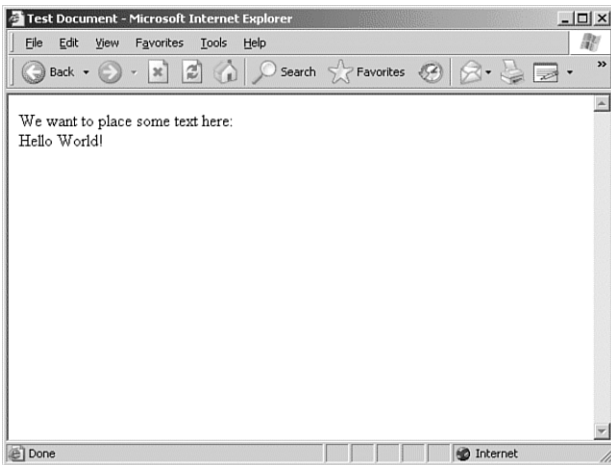


FIGURE 14.1 The DOM says “Hello World!”

When you want to create other page elements besides text nodes, you can do so using the `createElement()` method, which works pretty much like `createTextNode()`. We could, in fact, have used `createElement()` to create the `<div>` container itself, prior to adding our ‘Hello World!’ text node:

```
var newdiv = document.createElement("div");
```

In general, you simply pass the type of the required page element as an argument to `createElement()` to generate the required type of element.

An Overview of DOM Methods

This book is about Ajax, not just about JavaScript DOM techniques, so we’re not going to reproduce here a comprehensive guide to all the available methods and properties. However, Table 14.1 itemizes some of the more useful ones.

TABLE 14.1 Some JavaScript DOM Properties and Methods

NODE PROPERTIES

<code>childNodes</code>	Array of child nodes
<code>firstChild</code>	The first Child node
<code>lastChild</code>	The last Child node

TIP: If you need a more comprehensive account of the JavaScript DOM methods and properties, Andrew Watt gives a useful list in his excellent book *Sams Teach Yourself XML in 10 Minutes*, which is on the *Ajax Starter Kit* CD.

NODE PROPERTIES

nodeName	Name of the node
nodeType	Type of node
nodeValue	Value contained in the node
nextSibling	Next node sharing the same parent
previousSibling	Previous node sharing same parent
parentNode	Parent of this node

NODE METHODS

AppendChild	Add a new child node
HasChildNodes	True if this node has children
RemoveChild	Deletes a child node

DOCUMENT METHODS

CreateAttribute	Make a new attribute for an element
CreateElement	Make a new document element
CreateTextNode	Make a text item
GetElementsByTagName	Create an array of tagnames
GetElementsById	Find an element by its ID

Project—An RSS Headline Reader

Let's now take what we've learned about returning XML data from the server and use these techniques to tackle a new project.

XML data is made available on the Internet in many forms. One of the most popular is the RSS feed, a particular type of XML source usually containing news or other topical and regularly updated items. RSS feeds are available from many sources on the Web, including most broadcast companies and newspaper publishers, as well as specialist sites for all manner of subjects.

We'll write an Ajax application to take a URL for an RSS feed, collect the XML, and list the titles and descriptions of the news items contained in the feed.

The following is part of the XML for a typical RSS feed:

```
<rss version="0.91">
<channel>
<title>myRSSfeed.com</title>
```

14: Returning Data as XML

```
<link>http://www.*****.com/</link>
<description>My RSS feed</description>
<language>en-us</language>
<item>
<title>New Store Opens</title>
<link>http://www.*****.html</link>
<description>A new music store opened today in Canal Road.
  ↳The new business, Ajax Records, caters for a wide range of
  ↳musical tastes.</description>
</item>
<item>
<title>Bad Weather Affects Transport</title>
<link>http://www.*****.html</link>
<description>Trains and buses were disrupted badly today
  ↳due to sudden heavy snow. Police advised people not to
  ↳travel unless absolutely necessary.</description>
</item>
<item>
<title>Date Announced for Mayoral Election</title>
<link>http://www.*****.html</link>
<description>September 4th has been announced as the date
  ↳for the next mayoral election. Watch local news for more
  ↳details.</description>
</item>
</channel>
</rss>
```

From the first line

```
<rss version="0.91">
```

we see that we are dealing with RSS version 0.91 in this case. The versions of RSS differ quite a bit, but for the purposes of our example we only care about the `<title>`, `<link>`, and `<description>` elements for the individual news items, which remain essentially unchanged from version to version.

The HTML Page for Our Application

Our page needs to contain an input field for us to enter the URL of the required RSS feed and a button to instruct the application to collect the data. We also will have a `<div>` container in which to display our parsed data:

```
<html>
<head>
<title>An Ajax RSS Headline Reader</title>
```

```

</head>
<body>
<h3>An Ajax RSS Reader</h3>
<form name="form1">
URL of RSS feed: <input type="text" name="feed" size="50"
➡value="http://"><input type="button" value="Get Feed">
<br /><br />
<div id="news"><h4>Feed Titles</h4></div>
</form>
</html>

```

If we save this code to a file `rss.htm` and load it into our browser, we see something like the display shown in Figure 14.2.

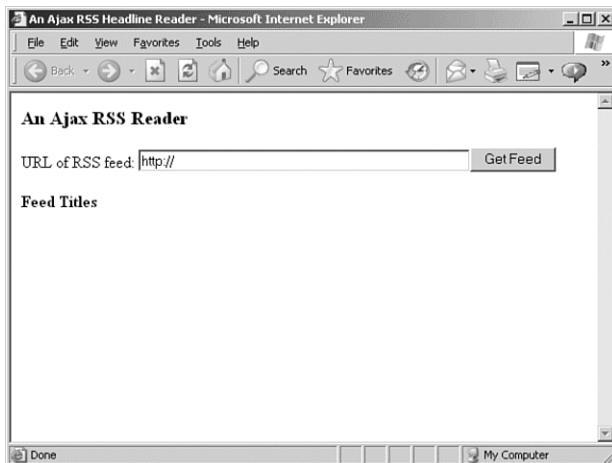


FIGURE 14.2 Displaying the base HTML document for our RSS headline reader.

Much of the code for our reader will be familiar by now; the means of creating an instance of the `XMLHttpRequest` object, constructing and sending a server request, and checking when that request has been completed are all carried out much as in previous examples.

This time, however, instead of using `responseText` we will be receiving data in XML via the `responseXML` property. We'll use that data to modify the DOM of our HTML page to show the news items' titles and descriptions in a list within the page's `<div>` container. Each title and description will be contained in its own paragraph element (which we'll also construct for the purpose) and be styled via a style sheet to display as we want.

The Code in Full

Let's jump right in and look at the code, shown in Listing 14.1.

LISTING 14.1 Ajax RSS Headline Reader

```
<html>
<head>
<title>An Ajax RSS Headline Reader</title>
</head>
<style>
.title {
font: 16px bold helvetica, arial, sans-serif;
padding: 0px 30px 0px 30px;
text-decoration:underline;
}
.descrip {
font: 14px normal helvetica, arial, sans-serif;
text-decoration:italic;
padding: 0px 30px 0px 30px;
background-color:#cccccc;
}
.link {
font: 9px bold helvetica, arial, sans-serif;
padding: 0px 30px 0px 30px;
}
.displaybox {
border: 1px solid black;
padding: 0px 50px 0px 50px;
}
</style>
<script language="JavaScript" type="text/javascript">
function getXMLHttpRequest() {
try {
req = new XMLHttpRequest(); /* e.g. Firefox */
} catch(e) {
try {
req = new ActiveXObject("Msxml2.XMLHTTP");
/* some versions IE */
} catch (e) {
try {
req = new ActiveXObject("Microsoft.XMLHTTP");
/* some versions IE */
} catch (E) {
req = false;
}
}
}
}
```

```

}
return req;
}

var http = getXMLHttpRequest();

function getRSS() {
    var myurl = 'rssproxy.php?feed=';
    var myfeed = document.form1.feed.value;
    myRand = parseInt(Math.random()*9999999999999999);
    // cache buster
    var modurl = myurl+escape(myfeed)+"&rand="+myRand;
    http.open("GET", modurl, true);
    http.onreadystatechange = useHttpResponse;
    http.send(null);
}

function useHttpResponse() {
    if (http.readyState == 4) {
        if(http.status == 200) {
            // first remove the childnodes
            // presently in the DM
            while (document.getElementById('news'))
➡.hasChildNodes())
            {
                document.getElementById('news').removeChild(document
➡.getElementById('news').firstChild);
            }
            var titleNodes = http.responseXML
➡.getElementsByTagName("title");
            var descriptionNodes = http.responseXML
➡.getElementsByTagName("description");
            var linkNodes = http.responseXML
➡.getElementsByTagName("link");
            for(var i =1;i<titleNodes.length;i++)
            {
                var newtext = document
➡.createTextNode(titleNodes[i]
➡.childNodes[0].nodeValue);
                var newpara = document.createElement('p');
                var para = document.getElementById('news')
➡.appendChild(newpara);
                newpara.appendChild(newtext);
                newpara.className = "title";

                var newtext2 = document

```

LISTING 14.1 Continued

```

➤.createTextNode(descriptionNodes[i]
➤.childNodes[0].nodeValue);
    var newpara2 = document.createElement('p');
    var para2 = document
➤.getElementById('news').appendChild(newpara2);
    newpara2.appendChild(newtext2);
    newpara2.className = "descrip";
    var newtext3 = document
➤.createTextNode(linkNodes [i]
➤.childNodes[0].nodeValue);
    var newpara3 = document.createElement('p');
    var para3 = document.getElementById('news')
➤.appendChild(newpara3);
    newpara3.appendChild(newtext3);
    newpara3.className = "link";
    }
  }
}
</script>
<body>
<center>
<h3>An Ajax RSS Reader</h3>
<form name="form1">
URL of RSS feed: <input type="text" name="feed"
➤size="50" value="http://"><input type="button"
➤onClick="getRSS()" value="Get Feed"><br><br>
<div id="news" class="displaybox">
➤<h4>Feed Titles</h4></div>
</form>
</center>
</html>

```

Mostly we are concerned with describing the workings of the callback function `useHttpResponse()`.

The Callback Function

In addition to the usual duties of checking the `XMLHttpRequest.readyState` and status properties, this function undertakes for us the following tasks:

- Remove from the display `<div>` any display elements from previous RSS listings.

- Parse the incoming XML to extract the title, link, and description elements.
- Construct DOM elements to hold and display these results.
- Apply CSS styles to these elements to change how they are displayed in the browser.

To remove the DOM elements installed by previous news imports (where they exist), we first identify the `<div>` element by using its ID and then use the `hasChildNodes()` DOM method, looping through and deleting the first child node from the `<div>` element each time until none remain:

```
while (document.getElementById('news').hasChildNodes())
{
    document.getElementById('news')
➡.removeChild(document.getElementById('news').firstChild);
}
```

The following explanation describes the processing of the title elements, but, as can be seen from Listing 14.1, we repeat the process identically to retrieve the description and link information too.

To parse the XML content to extract the item titles, we build an array `titleNodes` from the XML data stored in `responseXML`:

```
var titleNodes
➡ = http.responseXML.getElementsByTagName("title");
```

We can then loop through these items, processing each in turn:

```
for(var i =1;i<titleNodes.length;i++)
    { ... processing instructions ... }
```

For each title, we need to first extract the title text using the `nodeValue` property:

```
var newtext = document.createTextNode(titleNodes[i]
➡.childNodes[0].nodeValue);
```

We can then create a paragraph element:

```
var newpara = document.createElement('p');
```

append the paragraph as a child node of the `<div>` element:

```
var para = document.getElementById('news')
➡.appendChild(newpara);
```

and apply the text content to the paragraph element:

```
newpara.appendChild(newtext);
```

Finally, using the `className` property we can define how the paragraph is displayed. The class declarations appear in a `<style>` element in the document head and provide a convenient means of changing the look of the RSS reader to suit our needs.

```
newpara.className = "title";
```

Each time we enter the URL of a different RSS feed into the input field and click the button, the `<div>` content is updated to show the items belonging to the new RSS feed. This being an Ajax application, there is of course no need to reload the whole page.

The Server-Side Code

Because of the security constraints built into the `XMLHttpRequest` object, we can't call an RSS feed directly; we must use a script having a URL on our own server, and have this script collect the remote XML file and deliver it to the Ajax application.

In this case, we do not require that the server-side script `rssproxy.php` should *modify* the XML file but simply route it back to us via the `responseXML` property of the `XMLHttpRequest` object. We say that the script is acting as a proxy because it is retrieving the remote resource on behalf of the Ajax application.

Listing 14.2 shows the code of the PHP script.

LISTING 14.2 Server Script for the RSS Headline Reader

```
<?php
$mysession = curl_init($_GET['feed']);
curl_setopt($mysession, CURLOPT_HEADER, false);
curl_setopt($mysession, CURLOPT_RETURNTRANSFER, true);
$out = curl_exec($mysession);
header("Content-Type: text/xml");
echo $out;
curl_close($mysession);
?>
```

The script uses the cURL PHP library, a set of routines for making Internet file transfer easier to program. A full description of cURL would not be appropriate here; suffice to say that this short script first receives the URL of the required RSS feed by referring to the feed variable sent

by the Ajax application. The two lines that call the `curl_setopt()` function declare, respectively, that we don't want the headers sent with the remote file, but we do want the file contents. The `curl_exec()` function then makes the data transfer.

TIP: For a full description of using cURL with PHP, see the PHP website at <http://uk2.php.net/curl> and/or the cURL site at <http://curl.haxx.se/>.

After that it's simply a matter of adding an appropriate header by using the familiar PHP `header()` command and returning the data to our Ajax application.

Figure 14.3 shows the RSS reader in action, in this case displaying content from a CNN newsfeed.

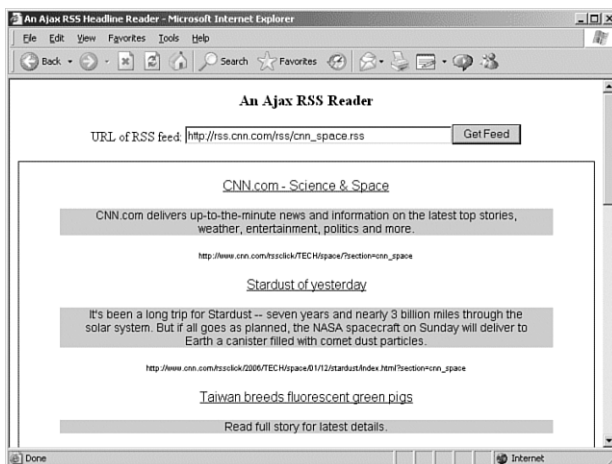


FIGURE 14.3 The Ajax RSS reader in action.

Summary

The JavaScript DOM methods, when used with the XMLHttpRequest object and XML data, provide a powerful means of transferring, organizing, and either displaying or otherwise processing data that has a sophisticated structure.

In this lesson you saw how DOM elements can be added, deleted, and manipulated to restructure an application's DOM in accordance with XML data received in the XMLHttpRequest object's `responseXML` property.

Web Services and the REST Protocol

In this lesson you will learn the basics of web services and how to implement them using the REST (Representational State Transfer) protocol.

Introduction to Web Services

So far you have seen several example applications in which we have called server-side scripts to carry out tasks. In each case we devised data structures to transfer the information and written routines to handle data transfer both to and from the server.

Suppose, though, that you wanted to make your server-side programs more generally available. Perhaps you can imagine that several different web applications might interface with such scripts for their own purposes. As well as browsers requesting pages directly, perhaps other applications (for example Ajax applications operating via XMLHttpRequest calls) might also make data requests and expect to receive, in response, data that they can understand and manipulate.

In such cases it would be beneficial to have some form of standardization in the interfaces that your program makes available. This principle provides the basis of what have come to be known as *web services*.

As an example, suppose that our server application produces XML-formatted weather forecast data in response to a request containing geographical information.

The nature of this type of service makes it broadly applicable; such an application might have a wide variety of “clients” ranging from simple web pages that present weather forecasts in their local area to complex aviation or travel planning applications that require the data for more demanding uses.

This type of service is just one small example of what a web service might be capable of doing. Thousands of web services are active on the Internet, providing a mind-boggling array of facilities including user authentication, payment processing, content syndication, messaging, and a host of others.

In general, a web service makes available an application programming interface (API), which allows client applications to build interfaces to the service. Although any Internet protocol might be used to create web services, XML and HTTP are popular options.

A number of protocols and techniques have emerged that help you to create and utilize web services. This lesson looks at perhaps the simplest of those, called *REST (Representational State Transfer)*, and Lesson 16, “Web Services Using SOAP,” discusses another protocol, this time called *SOAP (the Simple Object Access Protocol)*. Each lesson highlights in particular how they may be useful in Ajax applications.

REST—Representational State Transfer

REST is centered on two main principles for generalized network design:

- Resources are represented by URLs—A resource can be thought of as a “noun” and refers to some entity we want to deal with in the API of a web service; this could be a document, a person, a meeting, a location, and so on. Each resource in a REST application has a unique URL.
- Operations are carried out via standard HTTP methods—HTTP methods such as GET, POST, PUT, and DELETE are used to carry out operations on resources. In this way we can consider such operations as “verbs” acting on resources.

A Hypothetical REST Example

To understand how and why we might apply these ideas, let's look at a hypothetical example.

Suppose that we have a web service that allows writers to submit, edit, and read articles. Applying so-called RESTful principles to the design of this application, the following occurs:

TIP: Although REST requires that URLs be unique, it does not follow that each resource must have a corresponding physical page. In many cases the resource is generated by the web service at the time of the request—for example, by reference to a database.

- Each submitted article has a unique URL, for example:

`http://somedomain.com/articles/173`

We only require that the URL be unique for each article; for instance

`http://somedomain.com/articles/list.php?id=173`

also fulfils this requirement.

- To retrieve an article to read or edit, our client application would simply use an HTTP GET request to the URL of the article in question.
- To upload a new article, a POST request would be used, containing information about the article. The server would respond with the URL of the newly uploaded article.
- To upload an edited article, a PUT request would be used, containing the revised content.
- HTTP DELETE would be employed to delete a particular article.

NOTE: The World Wide Web itself is a REST application.

In this way, the web service is using an interface familiar to anyone who has used the World Wide Web. We do not need to devise a library of API methods for sending or retrieving information; we already have them in the form of the standard HTTP methods.

Query Information Using GET

An important issue concerning the use of the HTTP GET request in a RESTful application is that it should never change the server state. To put it another way: We only use GET requests to ask for information from the server, never to add or alter information already there.

POST, PUT, and DELETE calls can all change the server status in some way.

Stateless Operation

All server exchanges within a RESTful application should be *stateless*. By stateless we mean that the call itself must contain all the information required by the server to carry out the required task, rather than depending on some state or context currently present on the server. We cannot, for example, require the server to refer to information sent in previous requests.

Using REST in Practice

Let's expand on the example quoted earlier involving our articles web service.

Reading a List of Available Articles

The list of available articles is a resource. Because the web service conforms to REST principles, we expect the service to provide a URL by which we can access this resource, for instance:

`http://somedomain.com/articles/list.php`

Because we are querying information, rather than attempting to change it, we simply use an HTTP GET request to the preceding URL. The server may return, for example, the following XML:

```
<articles>
  <article>
    <id>173</id>
    <title>New Concepts in Ajax</title>
    <author>P.D. Johnstone</author>
  </article>
  <article>
    <id>218</id>
    <title>More Ajax Ideas</title>
    <author>S.N. Braithwaite</author>
  </article>
  <article>
    <id>365</id>
    <title>Pushing the Ajax Envelope</title>
    <author>Z.R. Lawson</author>
  </article>
</articles>
```

Retrieving a Particular Article

Because this is another request for information, we are again required to submit an HTTP GET request. Our web service might perhaps allow us to make a request to

```
http://somedomain.com/articles/list.php?id=218
```

and receive in return

```
<article>
  <id>218</id>
  <title>More Ajax Ideas</title>
  <author>S.N. Braithwaite</author>
</article>
```

Uploading a New Article

In this instance we need to issue a POST request rather than a GET request. In cases similar to the hypothetical one outlined previously, it is likely that the server will assign the `id` value of a new article, leaving us to encode parameter and value pairs for the `title` and `author` elements:

```
var articleTitle = 'Another Angle on Ajax';
var articleAuthor = 'K.B. Schmidt';
var url = '/articles/upload.php';
var poststring = "title="+encodeURIComponent(articleTitle)
➔+"&author="+encodeURIComponent(articleAuthor);
http.onreadystatechange = callbackFunction();
http.open('POST', url, true);
http.setRequestHeader("Content-type",
➔"application/x-www-form-urlencoded");
http.setRequestHeader("Content-length", poststring.length);
http.send(poststring);
```

Real World REST—the Amazon REST API

Leading online bookseller Amazon.com makes available a set of REST web services to help developers integrate Amazon browsing and shopping facilities into their web applications.

By first creating a URL containing parameter/value pairs for the required search parameters (such as publisher, sort order, author, and so on) and then submitting a GET request to this URL, the Amazon web service can be persuaded to return an XML document containing product details. We may then parse that XML to create DOM objects for display in a web

NOTE: Amazon.com often refers to the REST protocol as *XML-over-HTTP* or *XML/HTTP*.

page or to provide data for further processing as required by our application.

TIP: Amazon requires that you obtain a *developer's token* to develop client applications for its web services. You will need this token in constructing REST requests to Amazon's web services. You can also obtain an Amazon Associate's ID to enable you to earn money by carrying Amazon services on your website. See <http://www.amazon.com> for details.

Let's see this in practice by developing a REST request to return a list of books. Many types of searches are possible, but in this example, we request a list of books published by Sams.

We start to construct the GET request with the base URL:

```
$url = 'http://xml.amazon.com/onca/xml3';
```

We then need to add a number of parameter/value pairs to complete the request:

```
$assoc_id = "XXXXXXXXXX"; // your Amazon Associate's ID
$dev_token = "ZZZZZZZZZZ"; // Your Developer Token
$manuf = "Sams";
$url = "http://xml.amazon.com/onca/xml3";
$url .= "?t=".$assoc_id;
$url .= "&dev-t=".$dev_token;
$url .= "&ManufacturerSearch=".$manuf;
$url .= "&mode=books";
$url .= "&sort=+salesrank";
$url .= "&offer=All";
$url .= "&type=1ite";
$url .= "&page=1";
$url .= "&f=xml";
```

Submitting this URL, we receive an XML file containing details of all matching books. I won't reproduce the whole file here (there are more than 5,000 titles!), but Listing 15.1 shows an extract from the XML file, including the first book in the list.

LISTING 15.1 Example of XML Returned by Amazon Web Service

```
<?xml version="1.0" encoding="UTF-8" ?>
<ProductInfo xmlns:xsi="http://www.w3.org/
  2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation
```



```

=>"http://xml.amazon.com/schemas3/dev-lite.xsd">
  <Request>
    <Args>
      <Arg value="Mozilla/4.0 (compatible; MSIE 6.0;
=>Windows NT 5.1; SV1; .NET CLR 1.1.4322)"
=> name="UserAgent" />
      <Arg value="OG2CGCT7MRWB37PXAS4B" name="RequestID" />
      <Arg value="All" name="offer" />
      <Arg value="us" name="locale" />
      <Arg value="1" name="page" />
      <Arg value="ZZZZZZZZZZ" name="dev-t" />
      <Arg value="XXXXXXXXXX" name="t" />
      <Arg value="xml" name="f" />
      <Arg value="books" name="mode" />
      <Arg value="Sams" name="ManufacturerSearch" />
      <Arg value="lite" name="type" />
      <Arg value="salesrank" name="sort" />
    </Args>
  </Request>
  <TotalResults>5051</TotalResults>
  <TotalPages>506</TotalPages>
  <Details url="http://www.amazon.com/exec/obidos/ASIN/
=>0672327236/themousewhisp-20?dev-t=
=>1WPTTG90FS816BXMNFG2%26camp=2025%26link_code=xm2">
    <Asin>0672327236</Asin>
    <ProductName>Sams Teach Yourself Microsoft SharePoint
=>2003 in 10 Minutes (Sams Teach Yourself
=>in 10 Minutes)</ProductName>
    <Catalog>Book</Catalog>
    <Authors>
      <Author>Colin Spence</Author>
      <Author>Michael Noel</Author>
    </Authors>
    <ReleaseDate>06 December, 2004</ReleaseDate>
    <Manufacturer>Sams</Manufacturer>
    <ImageUrlSmall>http://images.amazon.com/images/P/
=>0672327236.01.THUMBZZZ.jpg</ImageUrlSmall>
    <ImageUrlMedium>http://images.amazon.com/images/P/
=>0672327236.01.MZZZZZZZ.jpg</ImageUrlMedium>
    <ImageUrlLarge>http://images.amazon.com/images/P/
=>0672327236.01.LZZZZZZZ.jpg</ImageUrlLarge>
    <Availability>Usually ships in 24 hours</Availability>
    <ListPrice>$14.99</ListPrice>
    <OurPrice>$10.19</OurPrice>
    <UsedPrice>$9.35</UsedPrice>
  </Details>

```

Clearly we can now process this XML document in any way we want. For example, Lesson 14, “Returning Data as XML,” discussed how to use JavaScript DOM methods to select information from the XML document and place it in page elements added to the DOM of our document.

REST and Ajax

You know already that the `XMLHttpRequest` object has methods that allow you to directly deal with HTTP request types and URLs.

Accessing RESTful web services is therefore simplified to a great extent. Because you know that each resource exposed by the web service API has a unique URL, and that the methods made available by the service are standard HTTP methods, it becomes a simple matter to construct the required `XMLHttpRequest` calls.

The prospect of being able to access a wide variety of web services from within Ajax applications, and use the returned information within those applications, is attractive—even more so if you can use a consistent and simple interface protocol.

Summary

This lesson introduced the concept of web services and the principles underlying the REST protocol.

REST requires that all resources be made accessible via unique URLs and that all required actions can be carried out on those resources by means of the standard HTTP methods. This makes RESTful web services interface comfortably with Ajax applications, due to the `XMLHttpRequest` object having methods that directly reference URLs and HTTP methods to create server requests.

Lesson 16 discusses a different style of web service using SOAP and how it relates to Ajax development.

Web Services Using SOAP

In this lesson you will learn about using web services with the SOAP protocol.

Introducing SOAP (Simple Object Access Protocol)

In Lesson 15, “Web Services and the REST Protocol,” we discussed web services and in particular saw how the REST (Representational State Transfer) protocol can be used to provide a consistent application programming interface (API) to such services.

REST is a good example of a protocol designed to operate with *resource-oriented* services, those that provide a simple mechanism to locate a resource and a set of basic methods that can manipulate that resource. In a resource-oriented service, those methods normally revolve around creating, retrieving, modifying, and deleting pieces of information.

In the case of REST, the methods are those specified in the HTTP specifications—GET, POST, PUT, and DELETE.

In certain cases, however, we are more interested in the *actions* a web service can carry out than in the resources it can control. We might perhaps call such services *action-oriented*. In these situations the resources themselves may have some importance, but the key issues concern the details of the activities undertaken by the service.

Perhaps the most popular and widely used protocol for designing action-oriented web services is SOAP, the Simple Object Access Protocol.

This lesson looks at SOAP, comparing and contrasting it where appropriate with the REST protocol discussed in Lesson 15.

The Background of the SOAP Protocol

SOAP began in the late 1990s when XML was itself a fledgling web technology and was offered to the W3C in 2000. SOAP and another XML-based web service protocol, called XML-RPC, had a joint upbringing.

SOAP was designed essentially as a means of packaging remote procedure calls (requests to invoke programs on remote machines) into XML wrappers in a standardized way.

Numerous enterprises contributed to the early development of SOAP, including IBM, Microsoft, and Userland. The development of SOAP later passed to the XML Protocols Working Group of the W3C.

NOTE: The full name Simple Object Access Protocol has been dropped in the later versions of the SOAP specifications, as it was felt that the direction of the project had shifted and the name was no longer appropriate. The protocol continues to be referred to as SOAP.

TIP: You can get the latest information on the SOAP specification from the W3C website at <http://www.w3.org/2000/xp/Group/>.

The SOAP Protocol

SOAP is an XML-based messaging protocol. A SOAP request is an XML document with the following main constituents:

- An *envelope* that defines the document as a SOAP request
- A body element containing information about the call and the expected responses
- Optional header and fault elements that carry supplementary information

Let's look at a skeleton SOAP request:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    ... various commands . . .
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    ... various commands . . .
    <SOAP-ENV:Fault>
      ... various commands . . .
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note that the SOAP request is an XML file, which has as its root the `Envelope` element.

The first line of the `Envelope` is

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV =
➡"http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle=
➡"http://schemas.xmlsoap.org/soap/encoding/">
```

TIP: A *namespace* is an identifier used to uniquely group a set of XML elements or attributes, providing a means to qualify their names, so that names in other schemas do not conflict with them.

This line declares the `xmlns:soap` namespace, which must always have the value `xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"`.

The `encodingStyle` attribute contains information defining the data types used in the message.

Next appears the `Header` element, which is optional but must, if present, be the first element in the message. Attributes defined in the `Header` element define how the message is to be processed by the receiving application.

The body element of the SOAP message contains the message intended for the final recipient.

The serialized method arguments are contained within the SOAP request's body element. The call's XML element must immediately follow the opening XML tag of the SOAP body and must have the same name as the remote method being called.

The body may also contain a `Fault` element (but no more than one). This element is defined in the SOAP specification and is intended to carry information about any errors that may have occurred. If it exists, it must be a child element of the body element. The `Fault` element has various child elements including `faultcode`, `faultstring`, and `detail`, which contain specific details of the fault condition.

Code Example of a SOAP Request

Let's see how a typical SOAP request might look:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
➡"http://schemas.xmlsoap.org/soap/envelope/" SOAP
➡ENV:encodingStyle="http://schemas.xmlsoap.org/
➡soap/encoding/">
<SOAP-ENV:Body>
  <m:GetInvoiceTotal xmlns:m=
➡"http://www.somedomain.com/invoices">
    <m:Invoice>77293</m:Invoice>
  </m:GetInvoiceTotal>
```

```
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the preceding example, the `m:GetInvoiceTotal` and `m:Invoice` elements are specific to the particular application, and are not part of SOAP itself. These elements constitute the message contained in the SOAP envelope.

Let's see what the SOAP response from the web service might look like:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
  ↪ "http://schemas.xmlsoap.org/soap/envelope/" SOAP-
  ↪ ENV:encodingStyle="http://schemas.xmlsoap.org/
  ↪ soap/encoding/">
  <SOAP-ENV:Body>
    <m:ShowInvoiceTotal xmlns:m=
  ↪ "http://www.somedomain.com/invoices">
      <m:InvoiceTotal>3295.00</m:InvoiceTotal>
    </m:ShowInvoiceTotal>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Sending the SOAP Request Via HTTP

A SOAP message may be transmitted via HTTP GET or HTTP POST. If sent via HTTP POST, the SOAP message requires at least one HTTP header to be set; this defines the `Content-Type`:

`Content-Type: text/xml`

After a successful SOAP exchange, you would expect to receive the SOAP response preceded by an appropriate HTTP header:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: yyy
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
  ↪ "http://schemas.xmlsoap.org/soap/envelope/" SOAP-
  ↪ ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encod-
  ↪ ing/">
  <SOAP-ENV:Body>
    <m:ShowInvoiceTotal xmlns:m=
  ↪ "http://www.somedomain.com/invoices">
      <m:InvoiceTotal>3295.00</m:InvoiceTotal>
    </m:ShowInvoiceTotal>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using Ajax and SOAP

To use SOAP with Ajax, you need to perform a number of separate steps:

1. Create the SOAP envelope.
2. Serialize the application-specific information into XML.
3. Create the SOAP body containing a serialized version of your application-specific code.
4. Send an HTTP request via the XMLHttpRequest object, containing the SOAP message as a payload.

The callback function then needs to be responsible for unpacking the SOAP response and parsing the XML contained inside it.

Code Example

How might the resulting code look? Let's see an example using the fictitious SOAP web service of the previous example:

```
var invoiceno = '77293';
http.open("POST", "http://somedomain.com/invoices", true);
http.onreadystatechange=function() {
    if (http.readyState==4) {
        if(http.status==200) {
            alert('The server said: '+ http.responseText)
        }
    }
}
http.setRequestHeader("Content-Type", "text/xml")
var mySOAP = '<?xml version="1.0"?>'
    + '<SOAP-ENV:Envelope xmlns:SOAP-ENV='
➡ "http://schemas.xmlsoap.org/soap/envelope/"'
    + ' SOAP-ENV:encodingStyle='
➡ "http://schemas.xmlsoap.org/soap/encoding/">'
    + '<SOAP-ENV:Body>'
    + '<m:GetInvoiceTotal xmlns:m='
➡ "http://www.somedomain.com/invoices">'
    +
    '<m:Invoice>'+invoiceno+'</m:Invoice></m:GetInvoiceTotal>'
    + '</SOAP-ENV:Body></SOAP-ENV:Envelope>';
http.send(mySOAP);
```

Here we have constructed the entire SOAP envelope in a JavaScript string variable, before passing it to the send() function of the XMLHttpRequest object.

The value returned from the server needs to be parsed first to remove the SOAP response wrapper and then to recover the application data from the body section of the SOAP message.

Reviewing SOAP and REST

Over the course of this lesson and Lesson 15, we've looked at the REST and SOAP approaches to using web services.

Although other web services protocols exist, a significant REST versus SOAP argument has been waged among developers over the last couple of years.

I don't intend to join that argument in this book. Instead, let's summarize the similarities and differences between the two approaches:

- REST leverages the standard HTTP methods of PUT, GET, POST, and DELETE to create remote procedure calls having comparable functions. Web service implementations using the REST protocol seem particularly suited toward resource-based services, where the most-used methods generally involve creating, editing, retrieving, and deleting information. On the downside, REST requires a little more knowledge about the HTTP protocol.
- The SOAP protocol adds substantial complexity, with the necessity to serialize the remote call and then construct a SOAP envelope to contain it. Further work arises from the need to "unpack" the returned data from its SOAP envelope before parsing the data. These extra steps can also have an impact on performance, with SOAP often being a little slower in operation than REST for a similar task. SOAP does, however, make a more complete job of separating the remote procedure call from its method of transport, as well as add a number of extra features and facilities, such as the `Fault` element and type checking via namespaces.

Summary

In this lesson we considered SOAP, the Simple Object Access Protocol. SOAP is a popular web service protocol with a rather different approach to the REST protocol utilized in Lesson 15.

Either style of web service can be used via `XMLHttpRequest` requests, though they differ somewhat in the complexity of the code involved.

A JavaScript Library for Ajax

In this lesson you will learn how to encapsulate some of the techniques studied up to now into a small JavaScript library that you can call from your applications.

An Ajax Library

Through the lessons and code examples up to now, we have developed a number of JavaScript code techniques for implementing the various parts of an Ajax application. Among these methods are:

- A method for generating an instance of the XMLHttpRequest object, which works across the range of currently popular browsers
- Routines for building and sending GET and POST requests via the XMLHttpRequest object
- Techniques for avoiding unwanted caching of GET requests
- A style of callback function that checks for correct completion of the XMLHttpRequest call prior to carrying out your wishes
- Methods of providing user feedback
- Techniques for dealing with text data returned in responseText
- Techniques for dealing with XML information returned in responseXML

In addition, you saw in Lesson 13, “AHAH—Asynchronous HTML and HTTP,” how some of these methods could be abstracted into a small JavaScript “library” (in that case containing only two functions).

This lesson extends that idea to build a more fully featured library that allows Ajax facilities to be added simply to an HTML page with minimal additional code.

Of necessity, our Ajax library will not be as complex or comprehensive as the open source projects described later on; however, it will be complete enough to use in the construction of functional Ajax applications.

Reviewing myAHAHlib.js

Listing 17.1 shows the code of myAHAHlib.js, reproduced from Lesson 13.

LISTING 17.1 myAHAHlib.js

```
function callAHAH(url, pageElement, callMessage) {
    document.getElementById(pageElement).innerHTML
    ➔ = callMessage;
    try {
        req = new XMLHttpRequest(); /* e.g. Firefox */
    } catch(e) {
        try {
            req = new ActiveXObject("Msxml2.XMLHTTP");
            /* some versions IE */
        } catch (e) {
            try {
                req = new ActiveXObject("Microsoft.XMLHTTP");
                /* some versions IE */
            } catch (E) {
                req = false;
            }
        }
    }
    req.onreadystatechange =
    ➔function() {responseAHAH(pageElement)};
    req.open("GET",url,true);
    req.send(null);
}

function responseAHAH(pageElement) {
    var output = '';
    if(req.readyState == 4) {
        if(req.status == 200) {
```

```

        output = req.responseText;
        document.getElementById(pageElement).innerHTML
➡ = output;
    }
}
}

```

Let's consider how we may extend the capabilities of this library:

- There is currently support only for HTTP GET requests. It would be useful to be able to support at least the HTTP POST request too, especially if you intend to build applications using the REST protocol (as described in Lesson 15, "Web Services and the REST Protocol").
- The library currently only deals with text information returned via `responseText` and has no means to deal with `responseXML`.

Implementing Our Library

Having identified what needs to be done, we'll now put together a more capable Ajax library.

Creating XMLHttpRequest Instances

Let's turn our attention first to the routine for creating instances of the `XMLHttpRequest` object.

Currently this function is coupled tightly with the routine for constructing and sending HTTP GET requests. Let's decouple the part responsible for the creation of the `XMLHttpRequest` instance and put it into a function of its own:

```

function createREQ() {
try {
    req = new XMLHttpRequest(); /* e.g. Firefox */
} catch(err1) {
    try {
        req = new ActiveXObject("Msxml2.XMLHTTP");
        /* some versions IE */
    } catch (err2) {
        try {
            req = new ActiveXObject("Microsoft.XMLHTTP");
            /* some versions IE */
        } catch (err3) {

```

```

        req = false;
    }
}
}
return req;
}

```

We can now create XMLHttpRequest object instances by simply calling the following function:

```
var myreq = createREQ();
```

HTTP GET and POST Requests

We'll start with the GET request because we already support that type of request:

```

function requestGET(url, query, req) {
myRand=parseInt(Math.random()*99999999);
req.open("GET",url+'?' +query+'&rand='+myRand,true);
req.send(null);
}

```

To this request we must pass as arguments the URL to which the request will be sent and the identity of the XMLHttpRequest object instance.

We could exclude the query argument because, in a GET request, it's encoded into the URL. We keep the two arguments separate here to maintain a similar interface to the function for making POST requests.

The query argument must be suitably encoded prior to calling the function, though the cache-busting random element is added by the function.

Next, the POST function:

```

function requestPOST(url, query, req) {
req.open("POST", url,true);
req.setRequestHeader('Content-Type',
    'application/x-www-form-urlencoded');
req.send(query);
}

```

The Callback Function

How do we deal with the callback function? We are going to add a further function:

```
function doCallback(callback,item) {
    eval(callback + '(item)');
}
```

This function uses JavaScript's `eval()` function to execute another function whose name is passed to it as an argument, while also passing to that function an argument of its own, via `item`.

Let's look at how these functions might interact when called from an event handler:

```
function doAjax(url,query,callback,reqtype,getxml) {
    // create the XMLHttpRequest object instance
    var myreq = createREQ();
    myreq.onreadystatechange = function() {
        if(myreq.readyState == 4) {
            if(myreq.status == 200) {
                var item = myreq.responseText;
                if(getxml==1) {
                    item = myreq.responseXML;
                }
                doCallback(callback, item);
            }
        }
    }
    if(reqtype=='post') {
        requestPOST(url,query,myreq);
    } else {
        requestGET(url,query,myreq);
    }
}
```

Our function `doAjax` now takes five arguments:

- `url`—The target URL for the Ajax call
- `query`—The encoded query string
- `callback`—Identity of the callback function
- `reqtype`—'post' or 'get'
- `getxml`—1 to get XML data, 0 for text

Listing 17.2 shows the complete JavaScript source code.

LISTING 17.2 The Ajax Library myAJAXlib.js

```

function createREQ() {
  try {
    req = new XMLHttpRequest(); /* e.g. Firefox */
  } catch(err1) {
    try {
      req = new ActiveXObject("Msxml2.XMLHTTP");
    } /* some versions IE */
    catch (err2) {
      try {
        req = new ActiveXObject("Microsoft.XMLHTTP");
      } /* some versions IE */
      catch (err3) {
        req = false;
      }
    }
  }
  return req;
}

function requestGET(url, query, req) {
  myRand=parseInt(Math.random()*99999999);
  req.open("GET",url+'?' +query+'&rand='+myRand,true);
  req.send(null);
}

function requestPOST(url, query, req) {
  req.open("POST", url,true);
  req.setRequestHeader('Content-Type', 'application/
  x-www-form-urlencoded');
  req.send(query);
}

function doCallback(callback,item) {
  eval(callback + '(item)');
}

function doAjax(url,query,callback,reqtype,getxml) {
  // create the XMLHttpRequest object instance
  var myreq = createREQ();

  myreq.onreadystatechange = function() {
    if(myreq.readyState == 4) {
      if(myreq.status == 200) {
        var item = myreq.responseText;
        if(getxml==1) {

```

```

        item = myreq.responseXML;
    }
    doCallback(callback, item);
}
}
}
if(reqtype=='post') {
    requestPOST(url,query,myreq);
} else {
    requestGET(url,query,myreq);
}
}
}

```

Using the Library

To demonstrate the use of the library, we're going to start with another simple HTML page, the code for which is shown here:

```

<html>
<head>
</head>
<body>
<form name="form1">
<input type="button" value="test">
</form>
</body>
</html>

```

This simple page displays only a button labeled "Test". All the functionality on the form will be created in JavaScript, using our new Ajax library.

The steps required to "Ajaxify" the application are

1. Include the Ajax library myAJAXlib.js in the <head> area of the page.
2. Write a callback function to deal with the returned information.
3. Add an event handler to the page to invoke the server call.

We'll start by demonstrating a GET request and using the information returned in the `responseText` property. This is similar to the situation we faced when dealing with AHAAH in Lesson 13.

Including the Ajax library is straightforward:

```

<head>
<script Language="JavaScript" src="myAJAXlib.js"></script>

```

Next, we need to define our callback function to deal with the value stored in the `responseText` property. For these examples, we'll simply display the returned text in an alert:

```
<head>
<script Language="JavaScript" src="myAJAXlib.js"></script>
<script Language="JavaScript">
function cback(text) {
  alert(text);
}
</script>
```

Finally, we need to add an event handler call to our button:

```
onClick="doAjax('libtest.php', 'param=hello',
  ↪ 'cback', 'get', '0')"
```

Our server-side script `libtest.php` simply echoes back the parameter sent as the second argument:

```
<?php
echo "Parameter value was ".$param;
?>
```

Meanwhile the remaining parameters of the function call declare that the callback function is called `cback`, that we want to send an HTTP GET request, and that we expect the returned data to be in `responseText`. Listing 17.3 shows the complete code of our revised HTML page.

LISTING 17.3 HTML Page Rewritten to Call `myAJAXlib.js`

```
<html>
<head>
<script Language="JavaScript" src="myAJAXlib.js">
↪ </script>
<script Language="JavaScript">
function cback(text) {
  alert(text);
}
</script>
</head>
<body>
<form name="form1">
<input type="button" value="test" onClick=
  ↪ "doAjax('libtest.php', 'param=hello',
  ↪ 'cback', 'get', '0')">
</form>
</body>
</html>
```

Figure 17.1 shows the result of running the program.



FIGURE 17.1 Returning text following an HTTP GET request.

To use the same library to retrieve XML data, we'll once again use the server-side script of Lesson 11, "Our First Ajax Application," which you may recall delivers the current server time in a small XML document:

```
<?php
header('Content-Type: text/xml');
echo "<?xml version='1.0' ?><clock1><timenow>"
➔.date('H:i:s')."</timenow></clock1>";
?>
```

Our callback function must be modified because we now need to return the parsed XML. We'll use some DOM methods that should by now be familiar:

```
<script>
function cback(text) {
var servertime = text.getElementsByTagName("timenow")[0]
➔.childNodes[0].nodeValue;
alert('Server time is '+servertime);
}
</script>
```

The only other thing we need to change is the call to our `doAjax()` function:

```
onClick="doAjax('telltmeXML.php','','cback','post','1')"
```

Here we have decided to make a POST request. Our server-side script `telltmeXML.php` does not require a query string, so in this case the second argument is left blank. The final parameter has been set to '1' indicating that we expect the server to respond with XML in the property `responseXML`.

Figure 17.2 shows the result of running the program.

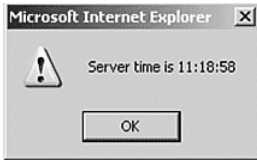


FIGURE 17.2 Returning the server time in XML via a POST request.

Extending the Library

The current library might be improved in a number of ways. These will be left as an exercise for the reader, though in many cases the techniques have been covered elsewhere in the book.

User feedback, for example, has not been addressed; we previously discussed how the display of suitable text or a graphic image can alert the user that a request is currently in progress. It would be useful to revise the library to include the techniques discussed in Lesson 11 and elsewhere.

Error handling, too, has been excluded from the code and would prove a useful addition. For example, it should not be too difficult to modify the library to detect XMLHttpRequest status properties other than 200 and output a suitable error message to the user.

Feel free to experiment with the code and see what you can achieve.

Summary

This lesson combined many of the techniques discussed to date to produce a compact and reusable JavaScript library that can be called simply from an HTML page.

The code supports both HTTP GET and HTTP POST requests and can deal with data returned from the server as text or XML.

Using such a library allows Ajax to be introduced to web pages using relatively small additions to the HTML markup. This not only keeps the code clean and easy to read but also simplifies the addition of Ajax facilities to upgrade legacy HTML.

In Lesson 18, “Ajax ‘Gotchas,’” the last lesson of Part III, we’ll discuss some potential problems and pitfalls awaiting the programmer in developing Ajax applications.

Ajax “Gotchas”

In this lesson you’ll learn about some of the common Ajax mistakes and how to avoid them.

Common Ajax Errors

Ajax has some common pitfalls waiting to catch the unwary developer. In this lesson, the last lesson of Part III, we’ll review some of these pitfalls and discuss possible approaches to finding solutions.

The list is not exhaustive, and the solutions offered are not necessarily appropriate for every occasion. They should, however, provide some food for thought.

The Back Button

All browsers in common use have a Back button on the navigation bar. The browser maintains a list of recently visited pages in memory and allows you to step back through these to revisit pages you have recently seen.

Users have become used to the Back button as a standard part of the surfing experience, just as they have with the other facets of the page-based web paradigm.

Ajax, as you have learned, does much to shake off the idea of web-based information being delivered in separate, page-sized chunks; with an Ajax application, you may be able to change page content over and over again without any thought of reloading the browser display with a whole new page.

TIP: JavaScript has its own equivalent of the Back button written into the language. The statements

```
onClick = "history.back()"
```

and

```
onClick = "history.go(-1)"
```

both mimic the action of clicking the Back button once.

What then of the Back button?

This issue has caused considerable debate among developers recently. There seem to be two main schools of thought:

- Create a means of recording state programmatically, and use that to re-create a previous state when the Back button is pressed.
- Persuade users that the Back button is no longer necessary.

Artificially re-creating former states is indeed possible but adds a great deal of complexity to Ajax code and is therefore somewhat the province of the braver programmer!

Although the latter option sounds a bit like it's trying to avoid the issue, it does perhaps have some merit. If you use Ajax to re-create desktop-like user interfaces, it's worthy of note that desktop applications generally don't have—or need—a Back button because the notion of separate “pages” never enters the user's head!

Bookmarking and Links

This problem is not unrelated to the Back button issue.

When you bookmark a page, you are attempting to save a shortcut to some content. In the page-based metaphor, this is not unreasonable; although pages can have some degree of dynamic content, being able subsequently to find the page itself usually gets us close enough to seeing what we saw on our previous visit.

Ajax, however, can use the same page address for a whole application, with large quantities of dynamic content being returned from the server in accordance with a user's actions.

What happens when you want to bookmark a particular screen of information and/or pass that link to a friend or colleague? Merely using the URL of the current page is unlikely to produce the results you require.

Although it may be difficult to totally eradicate this problem, it may be possible to alleviate it somewhat by providing permanent links to specially chosen states of an application.

Telling the User That Something Is Happening

This is another issue somewhat related to the change of interface style away from separate pages.

The user who is already familiar with browsing web pages may have become accustomed to program activity coinciding with the loading of a new or revised page.

Many Ajax applications therefore provide some consistent visual clue that activity is happening; perhaps a stationary graphic image might be replaced by an animated version, the cursor style might change, or a pop-up message appear. Some of these techniques have been mentioned in some of the lessons in this book.

Making Ajax Degrade Elegantly

The lessons in this book have covered the development of Ajax applications using various modern browsers. It is still possible, though, that a user might surprise you by attempting to use your application with a browser that is too old to support the necessary technologies. Alternatively, a visitor's browser may have JavaScript and/or ActiveX disabled (for security or other reasons).

It is unfortunate if an Ajax application should break down under these conditions.

At the least, the occurrence of obvious errors (such as a failure to create an instance of the XMLHttpRequest object) should be reported to the user. If the Ajax application is so complex that it cannot be made to automatically revert to a non-Ajax mode of operation, perhaps the user can at least be redirected to a non-Ajax version of the application.

TIP: You can detect whether JavaScript is unavailable by using the `<noscript> ... </noscript>` tags in your HTML page. Statements between these tags are evaluated only if JavaScript is NOT available:

```
<noscript>
JavaScript is not available in this browser.<br />
Please go <a href="otherplace.htm">HERE</a> for
the HTML-only version.<br />
</noscript>
```

Dealing with Search Engine Spiders

Search engines gather information about websites through various means, an important one being the use of automated programs called *spiders*.

Spiders, as their name suggests, “crawl the web” by reading web pages and following links, building a database of content and other relevant information about particular websites. This database, better known as an *index*, is queried by search engine visitors using their key words and phrases and returns suggestions of relevant pages for them to visit.

This can create a problem for highly dynamic sites, which rely on user interaction (rather than passive surfing) to invoke the loading of new content delivered on-demand by the server. The visiting spider may not have access to the content that would be loaded by dynamic means and therefore never gets to index it.

The problem can be exacerbated further by the use of Ajax, with its tendency to deliver even more content in still fewer pages.

It would seem wise to ensure that spiders can index a static version of all relevant content somewhere on the site. Because spiders follow links embedded in pages, the provision of a hypertext linked site map can be a useful addition in this regard.

Pointing Out Active Page Elements

Without careful design, it may not be apparent to users which items on the page they can click on or otherwise interface with to make something happen.

It is worth trying to use a consistent style throughout an application to show which page elements cause server requests or some other dynamic activity. This is somewhat reminiscent of the way that hyper-text links in HTML pages tend to be styled differently than plain text so that it's clear to a user that they perform an additional function.

At the expense of a little more coding effort, instructions and information about active elements can be incorporated in ToolTip-style pop-ups. This is, of course, especially important when a click on an active link can have a major effect on the application's state. Figure 18.1 shows an example of such a pop-up information box.

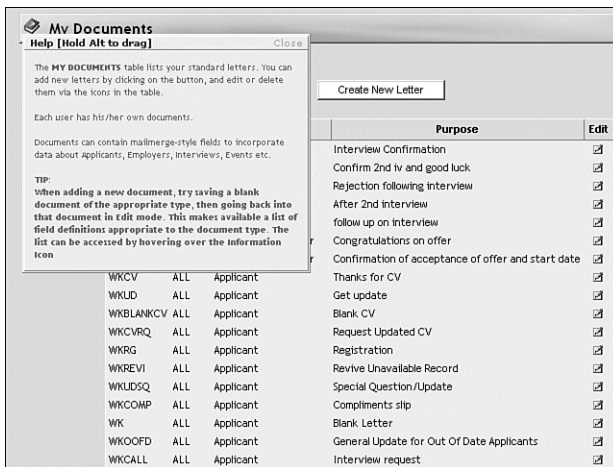


FIGURE 18.1 Pop-up information helps users to understand interfaces.

Don't Use Ajax Where It's Inappropriate

Attractive as Ajax undoubtedly is for improving web interfaces, you need to accept that there are many situations where the use of Ajax detracts from the user experience instead of adding to it.

This is especially true where the page-based interface metaphor is perfectly adequate for, perhaps even of greater relevance to, the content and style of the site. Text-based sites with subjects split conveniently into chapter-styled pages can often benefit as much from intelligently designed hyperlinking as they can from the addition of Ajax functionality.

Small sites in particular may struggle to get sufficient benefit from an Ajax interface to balance the associated costs of additional code and added complexity.

Security

Ajax does not itself seem to present any security issues that are not already present when designing web applications. It is notable, however, that Ajax-enhanced applications tend to contain more client-side code than they did previously.

Because the content of client-side code can be viewed easily by any user of the application, it is important that sensitive information not be revealed within it. In this context, sensitive information is not limited to such things as usernames and passwords (though they are, of course, sensitive), but also includes business logic. Make the server-side scripts responsible for carrying out such issues as database connection. Validate data on the server before applying it to any important processing.

Test Code Across Multiple Platforms

It will be clear from the content of this book that the various browsers behave differently in their implementation of JavaScript. The major difference in the generation of XMLHttpRequest object instances between Microsoft and non-Microsoft browsers is a fundamental example, but there is a host of minor differences, too.

The DOM, in particular, is handled rather differently, not only between browsers but also between different versions of the same browser. CSS implementation is another area where minor differences still proliferate.

Although it has always been important to test new applications on various browsers, this is perhaps more important than ever when faced with the added complexity of Ajax applications.

Hopefully browsers will continue to become more standards-compliant, but until then test applications on as many different platforms and with as many different browsers as possible.

Ajax Won't Cure a Bad Design

All the dynamic interactivity in the world won't correct a web application with a design that is fundamentally flawed.

All the tenets of good web design still apply to Ajax applications:

TIP: The W3C offers a free online validator at <http://validator.w3.org/>.

- Write for multiple browsers and validate your code.
- Comment and document your code well so that you can debug it later.
- Use small graphics wherever possible so that they load quickly.
- Make sure that your choices of colors, backgrounds, font sizes, and styles don't make pages difficult to read.

Some Programming Gotchas

Some of these have been alluded to in various lessons, but it's worth grouping them here. These are probably the most common programming issues that Ajax developers bump up against at some time or other!

Browser Caching of GET Requests

Making repeated GET requests to the same URL can often lead to the response coming not from the server but from the browser cache. This problem seems especially significant when using Internet Explorer.

Although in theory this can be cured with the use of suitable HTTP headers, in practice the cache can be stubborn.

An effective way of sidestepping this problem is to add a random element to the URL to which the request is sent; the browser interprets this as a request to a different page and returns a server page rather than a cached version.

In the text we achieved this by adding a random number. Another approach favored by many is to add a number derived from the time, which will of course be different every time:

```
var url = "serverscript.php"+"?rand="+new Date().getTime();
```

Permission Denied Errors

Receiving a Permission Denied error usually means that you have fallen foul of the security measure preventing cross-domain requests from being made by an XMLHttpRequest object.

Calls must be made to server programs existing in the same domain as the calling script.

CAUTION: Be careful that the domain is written in exactly the same way. Somedomain.com may be interpreted as referring to a different domain from www.somedomain.com, and permission will be denied.

Escaping Content

When constructing queries for GET or POST requests, remember to escape variables that could contain spaces or other nontext characters. In the

following code, the value `idValue` has been collected from a text input field on a form, so we escape it to ensure correct encoding:

```
http.open("GET", url + escape(idValue) + "&rand=" +  
myRandom, true);
```

Summary

Ajax undoubtedly has the potential to greatly improve web interfaces. However, the paradigm change from traditional page-based interfaces to highly dynamic applications has created a few potholes for developers to step into. In this lesson we’ve tried to round up a few of the better-known ones.

Some of these issues have already been encountered in the other lessons, whereas others will perhaps not become apparent until you start to develop real-world applications.

This lesson concludes Part III, “More Complex Ajax Technologies.” If you have followed the lessons through to this point, you will by now have a good grip on the fundamentals of the XMLHttpRequest object, JavaScript, XML, and the Document Object Model, and be capable of creating useful Ajax applications from first principles.

Fortunately, you don’t have to always work from first principles. Many open source and commercial projects on the Internet offer a wide variety of Ajax frameworks, tools, and resources.

Part IV, “Commercial and Open Source Ajax Resources,” concludes our journey through Ajax development by looking at some of these resources and their capabilities.

The prototype.js Toolkit

In this lesson you will learn about the `prototype.js` JavaScript library and how it can reduce the work required for building capable Ajax applications.

Introducing `prototype.js`

Part IV, “Commercial and Open Source Ajax Resources,” looks at some available code libraries and frameworks for Ajax development.

We begin this lesson with Sam Stephenson’s *prototype.js*, a popular JavaScript library containing an array of functions useful in the development of cross-browser JavaScript routines, and including specific support for Ajax. You’ll see how your JavaScript code can be simplified by using this library’s powerful support for DOM manipulation, HTML forms, and the XMLHttpRequest object.

The latest version of the `prototype.js` library can be downloaded from <http://prototype.conio.net/>.

Including the library in your web application is simple, just include in the `<head>` section of your HTML document the line:

```
<script src="prototype.js" Language="JavaScript"  
  type="text/javascript"></script>
```

prototype.js contains a broad range of functions that can make writing JavaScript code quicker, and the resulting scripts cleaner and easier to maintain.

The library includes general-purpose functions providing shortcuts to regular programming tasks, a wrapper for HTML forms, an object to encapsulate the XMLHttpRequest object, methods and objects for simplifying DOM tasks, and more.

Let's take a look at some of these tools.

ON THE CD: Version 1.5.1 of Prototype is included on the *Ajax Starter Kit CD*. If you download a different version, check the documentation to see whether there are differences between your version and the one described here.

The \$() Function

`$()` is essentially a shortcut to the `getElementById()` DOM method. Normally, to return the value of a particular element you would use an expression such as

```
var mydata = document.getElementById('someElementID');
```

The `$()` function simplifies this task by returning the value of the element whose ID is passed to it as an argument:

```
var mydata = $('someElementID');
```

Furthermore, `$()` (unlike `getElementById()`) can accept multiple element IDs as an argument and return an array of the associated element values. Consider this line of code:

```
mydataArray = $('id1','id2','id3');
```

In this example:

- `mydataArray[0]` contains value of element with ID `id1`.
- `mydataArray[1]` contains value of element with ID `id2`.
- `mydataArray[2]` contains value of element with ID `id3`.

The \$F() Function

The `$F()` function returns the value of a form input field when the input element or its ID is passed to it as an argument. Look at the following HTML snippet:

```
<input type="text" id="input1" name="input1">
<select id="input2" name="input2">
  <option value="0">Option A</option>
  <option value="1">Option B</option>
  <option value="2">Option C</option>
</select>
```

Here we could use

```
$F('input1')
```

to return the value in the text box and

```
$F('input2')
```

to return the value of the currently selected option of the select box.

The `$F()` function works equally well on check box and text area input elements, making it easy to return the element values regardless of the input element type.

The Form Object

prototype.js defines a `Form` object having several useful methods for simplifying HTML form manipulation.

You can return an array of a form's input fields by calling the `getElements()` method:

```
inputs = Form.getElements('thisform');
```

The `serialize()` method allows input names and values to be formatted into a URL-compatible list:

```
inputlist = Form.serialize('thisform');
```

Using the preceding line of code, the variable `inputlist` would now contain a string of serialized parameter and value pairs:

```
field1=value1&field2=value2&field3=value3...
```

`Form.disable('thisform')` and `Form.enable('thisform')` each do exactly what it says on the tin.

The Try . these() Function

Previous lessons discussed the use of exceptions to enable you to catch runtime errors and deal with them cleanly. The `Try . these()` function provides a convenient way to encapsulate these methods to provide a cross-browser solution where JavaScript implementation details differ:

```
return Try.these(function1(),function2(),function3(), ...);
```

The functions are processed in sequence, operation moving on to the next function when an error condition causes an exception to be thrown. Operation stops when any of the functions completes successfully, at which point the function returns `true`.

Applying this function to the creation of an XMLHttpRequest instance shows the simplicity of the resulting code:

```
return Try.these(
  function() {return new ActiveXObject('Msxml2.XMLHTTP')},
  function() {return new ActiveXObject('Microsoft.XML-
HTTP')},
  function() {return new XMLHttpRequest()}
)
```

NOTE: You may want to compare this code snippet with Listing 8.1 to see just how much code complexity has been reduced and readability improved.

Wrapping XMLHttpRequest—the Ajax Object

prototype.js defines an Ajax object designed to simplify the development of your JavaScript code when building Ajax applications. This object has a number of classes that encapsulate the code you need to send server requests, monitor their progress, and deal with the returned data.

Ajax.Request

Ajax.Request deals with the details of creating an instance of the XMLHttpRequest object and sending a correctly formatted request. Calling it is straightforward:

```
var myAjax = new Ajax.Request( url, {method: 'post',
  ➤parameters: mydata, onComplete: responsefunction} );
```

In this call, `url` defines the location of the server resource to be called, `method` may be either `post` or `get`, `mydata` is a serialized string containing the request parameters, and `responsefunction` is the name of the callback function that handles the server response.

The `onComplete` parameter is one of several options corresponding to the possible values of the XMLHttpRequest `readyState` properties, in this case a `readyState` value of 4 (`Complete`). You might instead specify that the callback function should execute during the prior phases `Loading`, `Loaded`, or `Interactive`, by using the associated parameters `onLoading`, `onLoaded`, or `onInteractive`.

There are several other optional parameters, including `asynchronous: false`

to indicate that a server call should be made synchronously. The default value for the `asynchronous` option is `true`.

TIP: The second argument is constructed using a notation often called *JSON* (*JavaScript Object Notation*). The argument is built up from a series of `parameter: value` pairs, the whole contained within braces. The parameter values themselves may be JSON objects, arrays, or simple values.

JSON is popular as a data interchange protocol due to its ease of construction, ease of parsing, and language independence. You can find out more about it at <http://www.json.org>.

Ajax.Updater

On occasions when you require the returned data to update a page element, the `Ajax.Updater` class can simplify the task. All you need to do is to specify which element should be updated:

```
var myAjax = new Ajax.Updater(elementID, url, options);
```

The call is somewhat similar to that for `Ajax.Request` but with the addition of the target element's ID as the first argument. The following is a code example of `Ajax.Updater`:

```
<script>
  function updateDIV(mydiv)
  {
    var url = 'http://example.com/serverscript.php';
    var params = 'param1=value1&param2=value2';
    var myAjax = new Ajax.Updater
      (
        mydiv,
        url,
        {method: 'get', parameters: params}
      );
  }
</script>
<input type="button" value="Go"
onclick="updateDIV(targetDiv)">
<div id="targetDiv"></div>
```

Once again, several additional options may be used when making the call. A noteworthy one is the addition of

```
evalScripts:true
```

to the options list. With this option added, any JavaScript code returned by the server will be evaluated.

Ajax.PeriodicalUpdater

The `Ajax.PeriodicalUpdater` class can be used to repeatedly create an `Ajax.Updater` instance. In this way you can have a page element updated after a certain time interval has elapsed. This can be useful for such applications as a stock market ticker or an RSS reader because it ensures that the visitor is always viewing reasonably up-to-date information.

`Ajax.PeriodicalUpdater` adds two further parameters to the `Ajax.Updater` options:

- frequency—The delay in seconds between successive updates. Default is two seconds.
- decay—The multiplier by which successive delays are increased if the server should return unchanged data. Default value is 1, which leaves the delay constant.

Here's an example call to `Ajax.PeriodicalUpdater`:

```
var myAjax = new Ajax.PeriodicalUpdater(elementID, url,  
    ↪ {frequency: 3.0, decay: 2.0});
```

Here we elected to set the initial delay to 3 seconds and have this delay double in length each time unchanged data is returned by the server.

Example Project—Stock Price Reader

Let's use the prototype.js library to build a simple reader that updates periodically to show the latest value returned from the server. In this example, we'll use a simple server-side script `rand.php` to simulate a changing stock price:

```
<?php  
srand ((double) microtime( ) * 1000000);  
$price = 50 + rand(0, 5000) / 100;  
echo "$price";  
?>
```

This script first initializes PHP's random number routine by calling the `srand()` function and passing it an argument derived from the current time. The `rand(0, 5000)` function is then used to generate a random number that is manipulated arithmetically to produce phony "stock prices" in the range 50.00 to 100.00.

Now let's build a simple HTML page to display the current stock price. This page forms the basis for our Ajax application:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01  
Transitional//EN"  
↪ "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
<script src="prototype.js" Language="JavaScript"  
↪ type="text/javascript"></script>  
<title>Stock Reader powered by Prototype.js</title>  
</head>
```



```

<body>
<h2>Stock Reader</h2>
<h4>Powered by Prototype.js</h4>
<p>Current Stock Price:</p>
<div id="price"></div>
</body>
</html>

```

Note that we included the `prototype.js` library by means of a `<script>` tag in the document head. We also defined a `<div>` with `id` set to “price”, which will be used to display the current stock price.

We now need to implement the `Ajax.PeriodicalUpdater` class, which we’ll attach to the document body element’s `onLoad` event handler. Listing 19.1 shows the complete script.

LISTING 19.1 Ajax Stock Price Reader Using `prototype.js`

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤ Transitional//EN"
➤ "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<script src="prototype.js" Language="JavaScript"
➤ type="text/javascript"></script>
<script>
function checkprice()
{
var myAjax = new Ajax.PeriodicalUpdater('price',
➤ 'rand.php', {method: 'post', frequency: 3.0,
➤ decay: 1});
}
</script>
<title>Stock Reader powered by Prototype.js</title>
</head>
<body onLoad="checkprice()">
<h2>Stock Reader</h2>
<h4>Powered by Prototype.js</h4>
<p>Current Stock Price:</p>
<div id="price"></div>
</body>
</html>

```

Look how simple the code for the application has become through using `prototype.js`. Implementing the application is merely a matter of defining a one-line function `checkprice()` to instantiate our repeating Ajax call and calling that function from the body element’s `onLoad` event handler.

From the arguments passed to `Ajax.PeriodicalUpdater`, you'll see that a 3-second repeat interval has been specified. This period does not change with subsequent calls because the decay value has been set to 1.

Figure 19.1 shows the application running. What cannot be seen from the figure, of course, is the stock price updating itself every 3 seconds to show a new value.

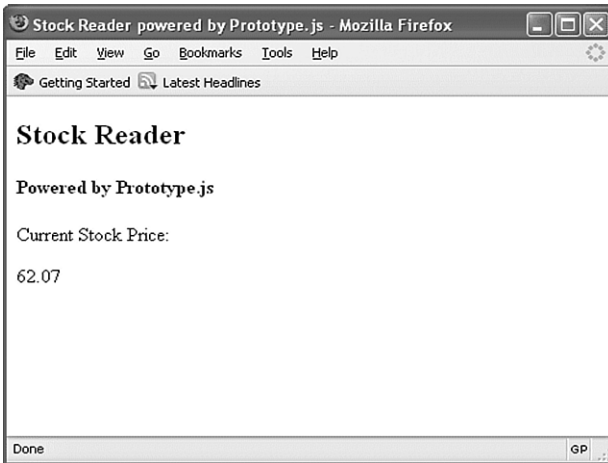


FIGURE 19.1 Ajax stock reader.

This simple example does not come close to showing off the power and versatility of the `prototype.js` library. Rather, it is intended to get you started with your own experiments by offering an easy point of access to this great resource.

Summary

In this first lesson in Part IV, we discussed the use of the powerful and elegant `prototype.js` JavaScript library.

The functions made available by this library greatly simplify some of the trickier programming tasks when developing Ajax applications.

The library offers good support for the `XMLHttpRequest` object, along with time-saving shortcuts for DOM handling, HTML forms, and many other techniques relevant to Ajax development.

Using Rico

In this lesson you will learn the basics of using Rico, a powerful Ajax and user interface development framework.

Introducing Rico

In Lesson 19, “The prototype.js Toolkit,” we looked at prototype.js, a powerful and useful JavaScript library that simplifies many of the programming tasks facing the Ajax developer.

In this lesson we’ll take a look at using Rico, a sophisticated Ajax framework employing the prototype.js library.

Rico is an open source library that extends the capabilities of prototype.js to provide a rich set of interface development tools. In addition to the Ajax development techniques discussed so far, Rico offers a whole range of tools such as drag-and-drop, cinematic effects, and more.

Tip *Rico* is the Spanish word for *rich*, which seems appropriate for a toolkit designed for building rich user interfaces!

Using Rico in Your Applications

To start using Rico to build applications with rich user interfaces, you need to include both Rico and prototype.js libraries in the <head>...</head> section of your web pages.

```
<script src="scripts/prototype.js"></script>
<script src="scripts/rico.js"></script>
```

Rico's AjaxEngine

The inclusion of `rico.js` causes an instance called `ajaxEngine` of an `AjaxEngine` object to be created automatically ready for you to use. The `AjaxEngine` is Rico's mechanism for adding Ajax capabilities to your web pages.

The `AjaxEngine` requires a three-step process to update page elements via Ajax:

1. Register the request handler. Registering the request handler associates a unique name with a particular URL to be called via Ajax.
2. Register the response handler. Rico can deal with the return of both HTML data and JavaScript code within the XML returned from the server. In the former case, the response handler identifies a page element that is to be updated using the returned data; in the latter case, a JavaScript object that handles the server response.
3. Make the Ajax call from the page by using an appropriate event handler.

We first register our request handler by making a call to the `registerRequest()` method of `ajaxEngine`:

```
ajaxEngine.registerRequest('getData', 'getData.php');
```

We have now associated the name `getData` with a request to the server routine `getData.php`. That server-side routine is required to return a response in well-formed XML. The following is an example of a typical response:

```
<ajax-response>
  <response type="element" id="showdata">
    <div class="datadisplay">
      The <b>cat</b> sat on the <b>mat</b>
    </div>
  </response>
</ajax-response>
```

Such responses always have a root element `<ajax-response>`. The `<response>` element it contains in this example has two attributes, `type` and `id` `showdata`. These signify, respectively, that the response contains HTML, and that this HTML is to be used to update the page element having `id` `showdata`. This element is updated via its `innerHTML` property.

TIP: Rico is capable of updating multiple page elements from one request. To achieve this, the `<ajax-response>` element may contain multiple `<response>` elements.

The other form of response that Rico can return is a JavaScript object. Here's an example:

```
<ajax-response>
  <response type="object" id="myHandler">
    <sentence>The cat sat on the mat.</sentence>
  </response>
</ajax-response>
```

Here the type has been set to `object`, indicating that the content is to be dealt with by a JavaScript object, the identity of which is contained in the `id` value (here `myHandler`). The content of the response is always passed to the `ajaxUpdate` method of this object.

How the response handler is registered depends on which type of response we are dealing with. For responses of type `element`, you can simply call:

```
ajaxEngine.registerAjaxElement('showdata');
```

In the case of responses containing a JavaScript object, you will need:

```
ajaxEngine.registerAjaxObject('myHandler', new
myHandler());
```

Whereas responses of type `element` are simply intended for the updating of HTML page elements, responses of type `object` can have handlers to process responses in any way they want. This allows Rico applications to be built ranging from simple to sophisticated.

A Simple Example

We can see Rico in action by using the simple script of Listing 20.1. This application updates two HTML elements with a single call to Rico's `ajaxEngine` object. The script for the application is in Listing 20.1.

LISTING 20.1 A Simple Rico Application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤ Transitional//EN"
➤"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Testing OpenRico</title>
<script src="prototype.js"></script>
<script src="rico.js"></script>
<script type="text/javascript">
function callRICO()
{
ajaxEngine.registerRequest('myRequest', 'ricotest.php');
```

LISTING 20.1 Continued

```

ajaxEngine.registerAjaxElement('display');
ajaxEngine.registerAjaxElement('heading');
}
</script>
</head>
<body onload=" callRICO();">
<div id="heading"><h3>Demonstrating Rico</h3></div>
<input type="button" value="Get Server Data"
  ↪ onclick="ajaxEngine.sendRequest('myRequest');" />
<div id="display"><p>This text should be replaced with
  ↪ server data ...</p></div>
</body>
</html>

```

You will see from the code that the single function `callRICO()` is used to register both the single request handler `myRequest` and two response handlers. The response handlers are used to update two `<div>` containers; one of these contains the page's heading, the other a short text message. On making the Rico request, the contents of both are updated, leaving the page with a new title and now displaying some server information instead of the previous text message. Figure 20.1 shows before and after screenshots.

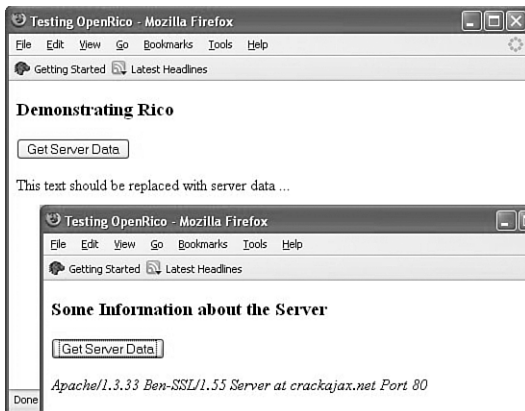


FIGURE 20.1 Updating multiple page elements with Rico.

The server routine is a simple PHP script that outputs the required XML data. The script uses PHP's `$_SERVER['SERVER_SIGNATURE']` global variable. Note that the script constructs and returns two separate

<response> elements, each responsible for updating a particular element in the HTML page.

Listing 20.2 shows the server script.

LISTING 20.2 The Server Script for Generating

```
<ajax-response>
<?php
header("Content-Type:text/xml");
header("Cache-Control:no-cache");
header("Pragma:no-cache");
echo "<ajax-response><response type=\"element\"
➡id=\"display\"><p>
➡".$_SERVER['SERVER_SIGNATURE']
➡.\"</p></response>
➡<response type=\"element\" id=\"heading\">
➡<h3>Some Information about the Server</h3>
➡</response></ajax-response>";
?>
```

TIP: Lesson 9, “Talking with the Server,” discussed problems that can occur due to the browser cache. In that lesson we used a workaround involving adding a parameter of random value to the URL of the server resource that we wanted to call.

This script example uses another technique, including the header commands

```
header("Cache-Control:no-cache");
header("Pragma:no-cache");
```

instructing the browser not to cache this page, but to collect a new copy from the server each time.

CAUTION: PHP’s `$_SERVER` global array variable was introduced in PHP 4.1.0. If you have an older version of PHP installed, you’ll need the global variable `$HTTP_SERVER_VARS` instead.

Rico’s Other Interface Tools

Rico’s capabilities aren’t limited to aiding the development of Ajax applications. Let’s now look at some other capabilities you can add to your user interfaces using the Rico toolkit. Although these techniques do not themselves use Ajax, it takes little imagination to realize what they might achieve when combined with Rico’s Ajax tools.

Drag-and-Drop

Both desktop applications and the operating systems on which they run make widespread use of drag-and-drop to simplify the user interface. The JavaScript techniques required to implement drag-and-drop can be tricky to master, not least because of the many cross-browser issues that arise.

Drag-and-drop using Rico, however, is simple.

Including the `rico.js` file in your application automatically causes the creation of an object called `dndMgr`, Rico's Drag and Drop Manager. Using the `dndMgr` object is much like using `AjaxEngine`; this time, though, we need to register not Ajax requests and responses, but *draggable* items and *drop zones* (page elements that can receive dragged items).

These tasks are carried out via the `registerDraggable` and `registerDropZone` methods:

```
dndMgr.registerDraggable( new Rico.Draggable('test',
➤ 'dragElementID') );
dndMgr.registerDropZone( new Rico.Dropzone
➤ ('dropElementID') );
```

These two simple commands declare, respectively, a page element with ID `dragElementID` as being draggable, and another element with ID `dropElementID` as a drop zone. The argument `'test'` of the `registerDraggable()` method defines a type for the draggable item, which can be tested and used by subsequent code, if required.

Example of a Drag-and-Drop Interface

Listing 20.3 shows how simple it is to implement drag-and-drop using Rico. The displayed HTML page is shown in Figure 20.2.

LISTING 20.3 Simple Drag-and-Drop Using Rico

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
➤ Transitional//EN"
➤ "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <script src="prototype.js"></script>
  <script src="rico.js"></script>
  <style>
    body {
      font: 10px normal arial, helvetica, verdana;
      background-color: #dddddd;
```



```

}

div.simpleDropPanel {
  width      : 260px;
  height     : 180px;
  background-color: #ffffff;
  padding    : 5px;
  border     : 1px solid #333333;
}

div.box {
  width      : 200px;
  cursor     : hand;
  background-color: #ffffff;
  -moz-opacity : 0.6;
  filter     : alpha(Opacity=60);
  border: 1px solid #333333;
}
</style>
</head>
<body>
<table width="550">
<tr>
  <td><h3>Drag and Drop</h3>
    <p>Drag and drop data items into the target fields
    ➡ using the left mouse button in the usual way.
    ➡ Note how available target fields change colour
    ➡ during the drag operation.</p>
    <p>Reload the page to start again.</p>
    <div class="box" id="draggable1">This is a piece
    ➡ of draggable data</div>
    <div class="box" id="draggable2">
    ➡ This is another</div>
    <div class="box" id="draggable3">
    ➡ And this is a third</div>
    <br/>
  </td>
</tr>
<tr>
  <td>
    <div id="droponme" class="simpleDropPanel">
      <b>Drop Zone 1</b><br />A simple text area
    </div>
  </td>
  <td>
    <b>Drop Zone 2</b><br />
    A form text entry field.
    <form><textarea name="dropzone" id="droponme2"
    ➡ rows="6" cols="30"></textarea></form>
  </td>
</tr>
</table>

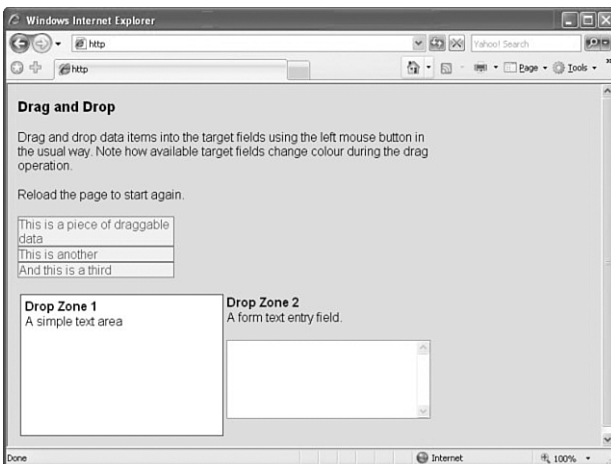
```

LISTING 20.3 Continued

```

    </td>
  </tr>
</table>
</td>
</tr>
</table>
<script>
  dndMgr.registerDraggable( new
➤ Rico.Draggable('foo','draggable1') );
  dndMgr.registerDraggable( new
➤ Rico.Draggable('foo','draggable2') );
  dndMgr.registerDraggable( new Rico.
➤ Draggable('foo','draggable3') );
  dndMgr.registerDropZone( new Rico.Dropzone
➤ ('dropzone') );
  dndMgr.registerDropZone( new Rico.Dropzone
➤ ('dropzone2') );
</script>
</body>
</html>

```

**FIGURE 20.2** The simple drag-and-drop application.

The two JavaScript libraries `rico.js` and `prototype.js` are included in the `<head>` of the document along with style definitions for various page elements.

Note that two page elements in particular, a `<div>` container and a `<textarea>` input field, have been given IDs of `dropzone1` and `dropzone2`. Further down the listing, these two elements are defined as drop zones for our drag-and-drop operations by the lines

```
dndMgr.registerDropZone( new Rico.Dropzone('droponme') );
dndMgr.registerDropZone( new Rico.Dropzone('droponme2') );
```

You'll see too that three small `<div>` containers have been defined in the page and given IDs of `draggable1`, `draggable2`, and `draggable3`. As you have no doubt guessed, they are to become draggable page elements and are defined as such by the following code lines:

```
dndMgr.registerDraggable( new Rico.Draggable('foo',
➡'draggable1') );
dndMgr.registerDraggable( new Rico.Draggable('foo',
➡'draggable2') );
dndMgr.registerDraggable( new Rico.Draggable('foo',
➡'draggable3') );
```

That's all there is to it! Rico takes care of all the details, even changing the look of the available drop zones while something is being dragged, as shown in Figure 20.3.

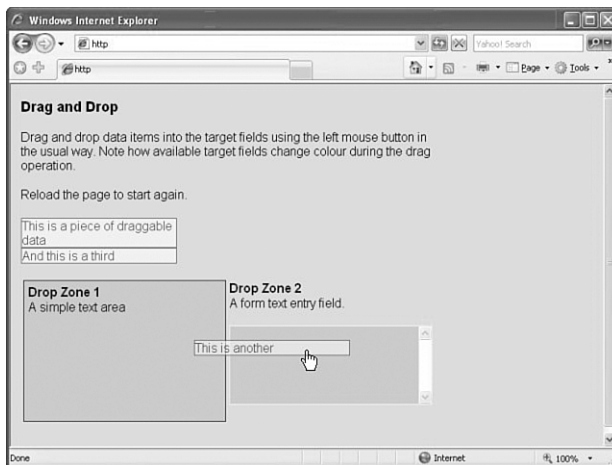


FIGURE 20.3 Drop zones highlighted during drag operation.

When released above an available drop zone, draggable items position themselves inline with the HTML code of the drop zone element, as can be seen in Figure 20.4.

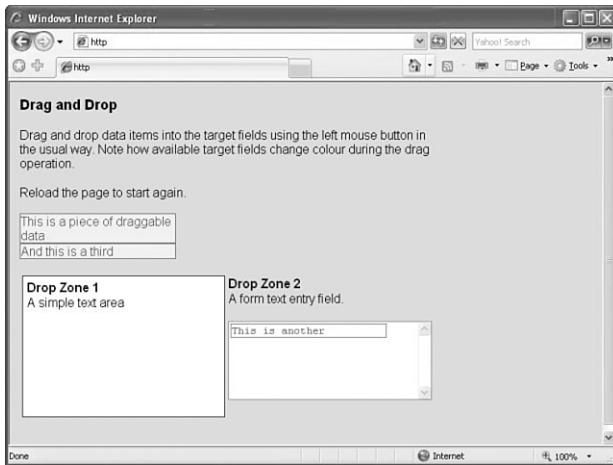


FIGURE 20.4 After completing the drag-and-drop.

Cinematic Effects

In addition to Ajax and drag-and-drop tools, Rico also makes available a host of user interface gadgets known collectively as *cinematic effects*.

NOTE: Rico's cinematic effects are extensions to the `Effect` class found in `prototype.js`.

These effects include animation of page elements (changing their sizes and/or shapes), fading effects (altering the opacity of page elements), applying rounded corners to objects, and manipulating object colors.

Used alongside the interface techniques previously discussed, these effects can help you to build sophisticated, eye-catching, and user-friendly interfaces much more reminiscent of desktop applications than of web pages.

Summary

Following our examination of the `prototype.js` library in the Lesson 19, this lesson moved on to experiment with Rico. Rico is an open source framework based on `prototype.js` that offers a simple way to integrate Ajax, along with drag-and-drop and other visual effects, into user interface designs.

Finally, in Lesson 21, "Using XOAD," we will look into an Ajax framework that uses an alternative approach—the server-side, PHP-based XOAD.

Using XOAD

In this lesson you will learn about XOAD, a server-side framework with Ajax support written by Stanimir Angeloff.

Introducing XOAD

So far in this part of the book we have looked at the prototype.js and Rico libraries and how they can help you to develop Ajax applications. Unlike these client-side libraries, which are written in JavaScript, XOAD is a *server-side* Ajax toolkit written in PHP.

This lesson discusses some of the concepts behind XOAD and the basics of its use.

All our work so far has concentrated on the use of JavaScript to handle both the server request and the returned data in Ajax applications. XOAD is a server-based solution written in PHP that takes a slightly different approach.

XOAD applications make server-based PHP functions available to the client-side JavaScript interpreter by passing serialized versions of them as JavaScript objects.

Downloading and Installing XOAD

XOAD is made up of many PHP and supporting scripts and can be downloaded as an archive file from <http://sourceforge.net/projects/xoad>. To install XOAD successfully, you need FTP access to a web server that supports PHP and (to use the more advanced features of XOAD) the MySQL database. Detailed instructions for installing XOAD can be found in the downloaded material, and there is a public forum at <http://forums.xoad.org/>.

A Simple XOAD Page

Let's take a look at an example of the simplest XOAD page. Suppose that you have a PHP class that you want to use in your XOAD application. This class is stored in the PHP file `myClass.class.php`:

```
<?php
class myClass {
    function stLength($mystring) {
        return strlen($mystring);
    }
    function xoadGetMeta() {
        XOAD_Client::mapMethods($this, array('stLength'));
        XOAD_Client::publicMethods($this, array('stLength'));
    }
}
?>
```

This simple class has only one function, `stLength()`, which merely returns the length of a string variable. We also added some *metadata* to the class in the form of the function `xoadGetMeta()`. This information tells XOAD which methods from the class are available to be exported to the main application. In this case there is just one, `stLength()`.

Now you need to start constructing the main application script `xoad.php`.

Listing 21.1 shows the XOAD application. This is a fairly pointless program that simply returns the length of a string, "My XOAD Application." Nevertheless, it demonstrates the concept of methods from server-side PHP classes being made available on the client side as JavaScript objects.

LISTING 21.1 A Simple XOAD Application

```
<?php
require_once('myClass.class.php');
require_once('xoad.php');
XOAD_Server::allowClasses('myClass');
if (XOAD_Server::runServer()) {
    exit;
}
?>
```

ON THE CD: All the needed tools—XOAD, PHP, MySQL, and Apache—are included on the *Ajax Starter Kit* CD.

CAUTION: It is not absolutely necessary to include metadata in the class, but it is recommended. Without metadata, all methods will be public, and method names will be converted to lowercase.

TIP: The Ajax applications developed in previous lessons were HTML files with file extensions `.htm` or `.html`. Because our XOAD application contains PHP code, it must have a suitable file extension. Most web server and PHP implementations will accept a file extension of `.php`, and some will allow other extensions such as `.php4` or `.phtml`.

```

<?= XOAD_Uilities::header('.') ?>
<script type="text/javascript">
var myobj = <?= XOAD_Client::register(new myClass()) ?>;
var mystring = 'My XOAD Application';
myobj.onStLengthError = function(error) {
    alert(error.message);
    return true;
}
myobj.stLength(mystring, function(result) {
    document.write('String: ' + mystring
➡ + '<br />Length: ' + result);
});
</script>

```

On loading the preceding document into a browser, the page simply says:

```
String: My XOAD Application
Length: 19
```

I won't go into much detail about how the PHP code works; this is after all about Ajax, not advanced PHP. It's important, though, to understand the concepts that underpin the code, so let's step through Listing 21.1 and try to understand what's happening:

```

<?php
require_once('myClass.class.php');
require_once('xoad.php');
XOAD_Server::allowClasses('myClass');
if (XOAD_Server::runServer()) {
    exit;
}
?>
<?= XOAD_Uilities::header('.') ?>

```

The first part of the script includes both `xoad.php` and the required class file `myClass.class.php`, and informs XOAD which classes it may access (in this case only one).

The XOAD function `runServer()` checks whether the XOAD request is a client callback, and if so handles it appropriately. The `header()` function is used to register the client header files.

Now let's look at the remainder of the script:

```

<script type="text/javascript">
var myobj = <?= XOAD_Client::register(new myClass()) ?>;
var mystring = 'My XOAD Application';

```

21: Using XOAD

```
myobj.onStLengthError = function(error) {  
    alert(error.message);  
    return true;  
}  
myobj.stLength(mystring, function(result) {  
    document.write('String: ' + mystring  
    ➤+ '<br />Length: ' + result);  
    });  
</script>
```

See how the remainder of the script is a `<script>...</script>` element?

The line

```
var myobj = <?= XOAD_Client::register(new myClass()) ?>;
```

exports the public methods declared in `myClass.class.php` to a JavaScript object. We now have a JavaScript object with a method `stLength()` that allows us to use the method of the same name from the PHP class `myClass`.

XOAD HTML

XOAD HTML is an extension that allows for the easy updating of HTML page elements using XOAD. The following examples show the use of the `XOAD_HTML::getElementById()` and `XOAD_HTML::getElementsByTagName()` methods, which do exactly the same thing as their equivalent JavaScript DOM methods.

XOAD_HTML::getElementById()

You will recognize the layout of the code in Listing 21.2 as being similar in structure to the basic XOAD program discussed earlier.

Rather than include an external class file, in this example we have defined a class, `Updater`, within the application itself. The class contains a single function, `change()`.

The first line in that function uses `XOAD_HTML::getElementById()` to identify the page element with and ID of `display`. Subsequent program lines proceed to change the text and background color of the page element.

The function `change()` is made available as a method of the JavaScript object `myobj` and can then be called like any other JavaScript method:

```
<a href="#server" onclick="myobj.change();  
➤return false;">Change It!</a>
```


Figure 21.1 shows the program's operation.

LISTING 21.2 Application to Use

XOAD_HTML::getElementById

```
<?php
class Updater
{
    function change()
    {
        $mytext =& XOAD_HTML::getElementById('display');
        $mytext->style['backgroundColor'] = 'yellow';
        $mytext->innerHTML = 'My background
➤ color has changed.';
    }
}
define('XOAD_AUTOHANDLE', true);
require_once('xoad.php');
?>
<?= XOAD_Uilities::header('.') ?>
<div id="display">My background color is white.</div>
<script type="text/javascript">
var myobj = <?= XOAD_Client::register(new Updater()) ?>;
</script>
<a href="#server" onclick="myobj.change();
➤return false;">Change It!</a>
```

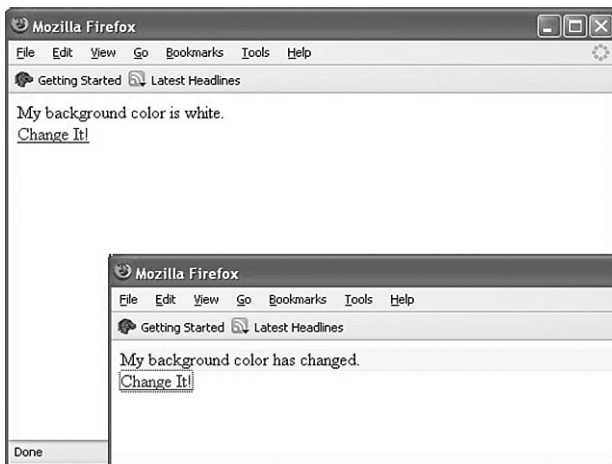


FIGURE 21.1 Using XOAD_HTML::getElementById().

XOAD_HTML::getElementsByTagName()

The `XOAD_HTML::getElementsByTagName()` method, like its JavaScript equivalent, returns an array of elements with a certain element type.

Listing 21.3 identifies all page elements of type `<div>` and changes some of their style attributes.

LISTING 21.3 Changing All Page Elements of a Given Type

```
<?php
class Updater
{
    function change()
    {
        $mydivs =& XOAD_HTML::getElementsByTagName('div');
        $mydivs->style['height'] = '60';
        $mydivs->style['width'] = '350';
        $mydivs->style['backgroundColor'] = 'lightgreen';
        $mydivs->innerHTML =
        ➤ 'Size and color changed by XOAD';
    }
}
define('XOAD_AUTOHANDLE', true);
require_once('xoad.php');
?>
<?= XOAD_Uilities::header('.') ?>
<script type="text/javascript">
var myobj = <?= XOAD_Client::register(new Updater()) ?>;
</script>
<style>
div {
border:1px solid black;
height:80;
width:150
}
</style>
<div>Div 1</div>
<br />
<div>Div 2</div>
<br />
<div>Div 3</div>
<a href="#"server" onclick="myobj.change();
➤return false;">Update All Divs</a>
```

The three `<div>` elements in the page are identified by `XOAD_HTML::getElementsByTagName()` and have their styles and sizes changed.

Figure 21.2 shows the program in operation.

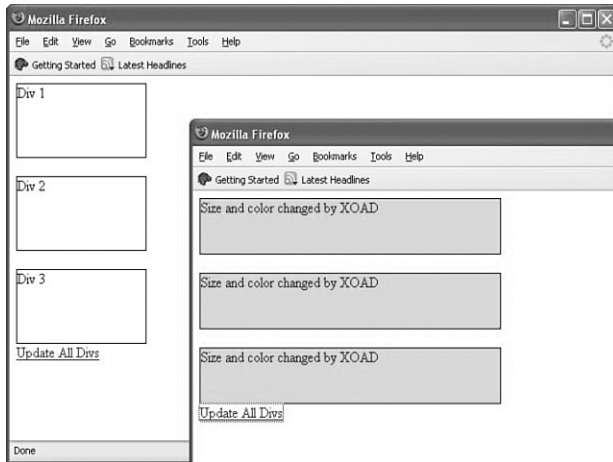


FIGURE 21.2 Selecting multiple page elements with XOAD_HTML.

TIP: XOAD_HTML has many other capabilities. Details of all the functions available within XOAD_HTML are in the XOAD download.

Advanced Programming with XOAD

XOAD has a range of advanced capabilities over and above those discussed in this lesson. In case you want to investigate the limits of what is possible using XOAD, here is an overview of the currently supported techniques.

XOAD Events

The XOAD framework also has support for *events*. A XOAD event instigated on one client's computer can be stored on the server and subsequently detected by other clients, making it possible to build complex applications in which users can interact. Such applications might, for instance, include chat, groupware, or similar collaborative tools.

Cache Handling with XOAD

XOAD allows for the caching on the server using the `XOAD_Cache` class. Caching results gives significant performance improvements, especially when server-side routines are time-intensive (such as sorting a large data set or performing queries on a sizeable database table).

XOAD Controls

You can define custom client controls in XOAD using the `XOAD_Controls` class.

Summary

This lesson examined a server-side implementation of an Ajax toolkit, in the form of XOAD.

XOAD allows the methods contained within PHP classes stored on the server to be made available to client programs using JavaScript. This forms an interesting contrast in approach compared to the client-side techniques discussed in Lessons 19 and 20.

This concludes Part IV. You should now have a good understanding of Ajax application architecture and the coding techniques on which it is based.

Good luck with your experiments in Ajax!

TIP: At the time of writing, the current version of XOAD is 0.6.0.0. If the version you download is different, consult the documentation included in the download.

INDEX

SYMBOLS

@ characters, PHP methods, 125
\$ SERVER global array variable, 187
\$() function, 176
\$F() function, 176-177
<ajax-response> elements, Rico, 184, 187
<div>...<div> elements, 102
<div> containers, 105
<response> elements, Rico, 184, 187
<script>...<script> elements, 103

A

abort method, 78
active page elements, designing, 171
AJAH (Asynchronous HTML and HTTP). *See also*
HTML; HTTP
 advantages of, 120
 callAJAH() functions, 121-123
 myAJAHlib.js, 121-123
 metatag information, retrieving from URL,
 124-125
 responseText property, 127
 responseAJAH() functions, 122-123
Ajax
 application examples, Google Suggest, 64-65
 application flow diagram, 68
 client-server interaction, 61-64

objects
 Ajax.PeriodicalUpdater class, 179-180
 Ajax.request class, 178
 Ajax.Updater class, 179
 using, inappropriate situations for, 171
Ajax Engines, 64
Ajax.PeriodicalUpdater class, 179-180
Ajax.request class, 178
Ajax.Updater class, 179
AjaxEngine objects, instances in Rico, 184-185
alt attribute (image tags), 19
Amazon.com REST API, 147-150
anchor tags (HTML), 20
Apache Web Server website, 9
AppendChild method, 131-133
applications
 designing, 101, 171
 callback functions, 105-106
 completed application, 107-108
 event handlers, 106
 HTML document, 102
 PHP scripts, 104-105
 server requests, 104
 troubleshooting, 173
 user feedback, 109-110
 XMLHttpRequest objects, 103-104
 flow diagrams, 68
 prototype.js, adding to, 175
 Rico, adding to, 183
arguments (JavaScript), 37, 42

- articles, REST
 - lists of available articles, reading, 146
 - particular articles, retrieving, 147
 - uploading, 147
- ASCII text, server responses, 67
- asynchronous servers
 - communications, 64
 - requests, 66, 81-86
- ATTLIST declarations (XML), 57

B

- Back button, 167
- bandwidth, defining, 62
- body tags (HTML), 16-18
- bookmarks, troubleshooting, 168-169
- browser caches
 - callAjax() functions, 84-86
 - GET requests, 173
 - server requests, 84-86
- browsers
 - availability of, 11
 - defining, 10
 - graphical browsers, 10
 - HTML documents, loading, 15
 - text-based browsers, 10
 - unsupported browsers, troubleshooting, 169-170
 - web server interaction, 7

C

- caches (browser)
 - GET requests, 173
 - server requests, 84-86
- callAHAAH() functions, 121-123
- callAjax() function, 83
 - browser caches, 84-86
 - launching, 89-90

- callback functions, 86-88
 - AHAH, 122-123
 - basic application creation example, 105-106
 - JavaScript libraries, 161-162
 - launching, 89-90
 - myAJAXlib.js, 164
 - RSS headline readers, creating, 138-140
- callRICO() function, 186
- center tags (HTML), 21
- change() function, 196-197
- character strings, split() method, 117
- charAt method, responseText property, 93
- child nodes, adding to DOM, 131
- childNodes property, 132
- cinematic effects (Rico), 192
- client-server interactions, traditional interactions versus Ajax, 61-62
- client-side programming, defining, 11
- code, troubleshooting, 172
- color, HTML, 18
- comments (HTML), 17
- constructors, creating instances, 72
- CreateAttribute method, 133
- CreateElement method, 132-133
- CreateTextNode method, 131-133

D

- data() function, 105
- date command (PHP), 50
- DELETE requests, 145
- design applications, troubleshooting, 173
- developer's tokens, 148
- DNS (Domain Name Service) servers, 12
- doAjax function, 161, 164-165
- DOCTYPE declarations (XML), 55-56
- DOCTYPE elements, 15-16
- document elements (XML), 55
- DOM (Document Object Model), 72-73
 - appendChild() method, 131
 - child nodes, adding to, 131

- createElement() method, 132
- createTextNode() method, 131
- document methods table, 133
- elements, deleting, 139
- getAttribute method, 59
- getElementById method, 130
- getElementsByName method, 130
- node methods table, 133
- node properties table, 132
- nodes, 58-59
- tagname properties, 59
- text properties, 59

DTD (Document Type Definitions) versus DOCTYPE declarations, 55

E

ELEMENT declarations (XML), 56

Engines (Ajax), 64

error handling

- application design, 173
- Back button codes, 167
- bookmarks, 168-169
- browser caches, 173
- code, platform tests, 172
- GET requests, 173-174
- JavaScript libraries, 166
- links, 168-169
- page design, 171
- Permission Denied errors, 174
- POST requests, 174
- security, 172
- spiders, 170
- unsupported browsers, 169-170
- user feedback, 169

eval() function, JavaScript libraries, 161-162

event handlers

- basic application creation example, 106
- JavaScript functions, calling, 43
- myAJAXlib.js, calls for, 164

- onChange event handler, 44
- onClick event handler, 38-39, 44
- onLoad event handler, 44
- onMouseOut event handler, 44
- onMouseOver event handler, 41-44
- onSubmit event handler, 44-46

F

feedback (user)

- basic application creation example, 109-110
- JavaScript libraries, 166
- server requests, 97
- troubleshooting, 169

file extensions, PHP files, 48

firstChild property, 132

for loops, 52

Form objects, prototype.js, 177

form tags (HTML), 28-30

form validation example (JavaScript), 45-46

Frameworks (Ajax), 64

functions

- \$(), 176
- \$F(), 176-177
- callAHAH(), 121-123
- callAjax, 83
 - browser caches, 84-86*
 - launching, 89-90*
- callback, 86-88
 - AHAH, 122-123*
 - basic application creation example, 105-106*
 - JavaScript libraries, 161-162*
 - launching, 89-90*
 - myAJAXlib.js, 164*
 - RSS headline readers, creating, 138-140*
- callIRICO(), 186
- change(), 196-197
- date(), 105
- doAjax, 161, 164-165

- eval(), JavaScript libraries, 161-162
- header(), 195
- JavaScript
 - arguments, passing to*, 42
 - calling*, 41
 - event handlers, calling from*, 43
 - numcheck*, 46
 - structure of*, 40
- responseAHAH(), 122-123
- responseAjax(), 83, 88
- runServer(), 195
- sizeof(), 117
- Try.these(), 177

G

- GET requests, 83
 - browser caches, 84-86, 173
 - HTTP requests, 29-31
 - JavaScript libraries, 160
 - myAJAXlib.js, 163
 - query strings, 29
 - REST, 145-147
 - troubleshooting, 174
- getAllResponseHeaders method, 78
- getAttribute method (DOM), 59
- getElementById method, 98, 106, 130
- getElementsByTagName method, 105-106
- getElements() method, prototype.js, 177
- GetElementsById method, 133
- getElementsByTagName method, 95-96, 130, 133
- getResponseHeader method, 78
- gmail web mail service (Google), 65
- Google Maps, 65
- Google Suggest, 64-65
- graphics web browsers, 10

H

- HasChildNodes method, 133
- head tags (HTML), 16

- header lines (HTTP)
 - requests, 27
 - responses, 28
- header() function, 195
- headers, outputting prior to issuing PHP scripts, 94
- Hello World example, printing in PHP, 48
- hexadecimal numbering system, HTML color values, 18
- HTML (Hypertext Markup Language), 13. **See also** AHAH
 - <div>...<div> elements, 102
 - <div> containers, 105
 - <script>...<script> elements, 103
 - advanced document example, 20-21
 - basic application creation example, 102
 - basic document example, 14
 - body tags*, 16-17
 - DOCTYPE elements*, 15
 - head tags*, 16
 - HTML tags*, 16
 - tags, adding attributes to*, 18
 - title tags*, 16
 - color values, 18
 - comments, 17
 - defining, 14
 - DOCTYPE elements, 15-16
 - forms
 - attributes*, 30
 - attributes methods*, 30
 - parameter values*, 31
 - processing*, 30
 - simple form example*, 30-31
 - special characters, transmitting*, 31
 - tags*, 28-30
 - variables*, 30
 - GET requests, 29-31
 - hyperlinks, 19-20
 - JavaScript, 34-37
 - loading documents, 15
 - metatags
 - keywords*, 123-125
 - myAHAHlib.js*, 124-125

- myAJAXlib.js, 164
- PHP, 48
- POST requests, 29-31
- responseText property, 115-116
- RSS headline readers, creating, 134
- saving documents, 15
- script tags, 34
- seville.html document example, 20-21
- styles, 23
 - inline styles*, 24
 - style sheet rules, setting*, 24
- tags, 14
 - adding attributes to*, 18
 - anchor tags*, 20
 - as containers*, 17
 - body tags*, 16-17
 - body tags, adding attributes to*, 18
 - center tags*, 21
 - common tags table*, 22-23
 - head tags*, 16
 - images tags*, 19
 - table tags*, 19-21
 - title tags*, 16
- testpage.html document example, 14
 - body tags*, 16-17
 - body tags, adding attributes to*, 18
 - DOCTYPE elements*, 15-16
 - head tags*, 16
 - HTML tags*, 16
 - loading*, 15
 - saving*, 15
 - title tags*, 16
- tool requirements, 14
- word processors, 14
- XML, similarities to, 54
- XOAD, 199
 - change() function*, 196-197
 - XOAD HTML::getElementById() method*, 196-197
 - XOAD HTML::getElementsByTagName() method*, 198

HTTP (Hypertext Transfer Protocol), 25. **See also**

- AHAH
 - requests, 7
 - GET requests*, 29-31
 - header lines*, 27
 - opening lines*, 26
 - POST requests*, 29-31
 - responses
 - header lines*, 28
 - reason phrases*, 27
 - status lines*, 27
 - server response status codes, 87
 - SOAP requests, sending, 154
 - versions (HTTP requests), 26
- hyperlinks, HTML, 19-20

I - J

- id values, 98
- if statements (PHP), 51
- image tags (HTML), 19
- images, defining pixels, 19
- indexOf method, responseText property, 93
- inline styles, 24
- instances (objects), creating, 72-77
- Internet, HTTP requests, 7
- Internet Explorer (MS), Jscript, 34
- IP addresses, defining, 12

- JavaScript, 33
 - arguments, 37, 42
 - Back button codes, 167
 - case sensitivity, 35
 - commands, execution order, 37
 - enabling, 34
 - form validation example, 45-46
 - functions
 - arguments, passing to*, 42
 - calling*, 41

- event handlers, calling from, 43*
 - numcheck, 46*
 - structure of, 40*
- HTML pages, adding to, 35-37
- methods, 37-42
- objects, 37, 45, 57-58
- script tags, 34
- variables, 44
- XML, 57-59

JavaScript libraries, 158

- callback functions, 161-162
- doAjax functions, 161, 164-165
- error handling, 166
- eval() function, 161-162
- GET requests, 160
- myAJAXlib.js, 158-162
 - callback functions, 164*
 - event handler calls, 164*
 - GET requests, 163*
 - HTML pages, 164*
 - PHP scripts, 164*
 - responseText properties, 164*
 - usage example, 163-165*
 - XML data, retrieving, 165*
- POST requests, 160, 165

prototype.js

- \$() function, 176*
- \$F() function, 176-177*
- Ajax.PeriodicalUpdater class, 179-180*
- Ajax.request class, 178*
- Ajax.Updater class, 179*
- download website, 175*
- Form objects, 177*
- getElements() method, 177*
- Rico, 183-192*
- serialize() method, 177*
- Stock Price Reader build example, 180-182*
- Try.these() function, 177*
- web applications, adding to, 175*

- user feedback, 166*
 - XMLHttpRequest instances, creating, 159-160*
- Jscript, 34
- JSON (JavaScript Object Notation) website, 178

K - L

keywords metatag, 123-125

lastChild property, 132

lastIndexOf method, responseText property, 93

libraries (JavaScript)

- callback functions, 161-162
- doAjax functions, 161, 164-165
- error handling, 166
- eval() function, 161-162
- GET requests, 160
- myAHAHlib.js, 158-159
- myAJAXlib.js, 161-162
 - callback functions, 164*
 - event handler calls, 164*
 - GET requests, 163*
 - HTML pages, 164*
 - PHP scripts, 164*
 - responseText properties, 164*
 - usage example, 163-165*
 - XML data, retrieving, 165*
- POST requests, 160, 165

prototype.js

- \$() function, 176*
- \$F() function, 176-177*
- Ajax.PeriodicalUpdater class, 179-180*
- Ajax.request class, 178*
- Ajax.Updater class, 179*
- download website, 175*
- Form objects, 177*
- getElements() method, 177*
- Rico, 183-192*
- serialize() method, 177*
- Stock Price Reader build example, 180-182*

- Try.these()* function, 177
- web applications, adding to*, 175
- user feedback, 166
- XMLHttpRequest instances, creating, 159-160
- libraries (open source), Rico, 183
- <response> elements, 184, 187
- AjaxEngine instances, 184-185
- callRICO() function, 186
- cinematic effects, 192
- drag-and-drop, 188-191
- multiple page element updates, 184
- usage example, 185-186
- web applications, adding to, 183
- links, troubleshooting, 168-169
- loading HTML documents, 15
- loop constructs (PHP), 52
- Lynx text-based web browsers, 10

M

- markup elements (HTML). **See** tags, HTML
- Math.random() method, 84
- metatags
 - keywords, 123-125
 - myAHAHlib.js, retrieving metatag information, 124-125
- methods, 71
 - abort, 78
 - AppendChild, 131-133
 - charAt, responseText property, 93
 - CreateAttribute, 133
 - CreateElement, 132-133
 - CreateTextNode, 131-133
 - getAllResponseHeaders, 78
 - getElementById, 98, 106, 130
 - getElementsByTagName, 105-106
 - getElements(), prototype.js, 177
 - GetElementsById, 133
 - getElementsByTagName, 95-96, 130, 133

- getResponseHeader, 78
- HasChildNodes, 133
- HTTP requests, 26-27
- indexOf, responseText property, 93
- JavaScript, event handlers, 37
 - onClick event handler*, 38-39
 - onMouseOver event handler*, 41-42
- lastIndexOf, responseText property, 93
- Math.random(), 84
- open, 78-79
- registerDraggable, 188
- registerDropZone, 188
- RemoveChild, 133
- send, 78-79
- serialize(), prototype.js, 177
- setRequestHeader, 78-79
- split(), 117
- substring, responseText property, 93
- toLowerCase(), responseText property, 93
- toUpperCase(), responseText property, 93
- XMLHttpRequest object, list of, 78
- XOAD HTML::getElementById(), 196-197
- XOAD HTML::getElementByTagName(), 198
- multiplatform code tests, 172
- myAHAHlib.js, 121-123, 158-159
 - metatag information, retrieving from URL, 124-125
 - responseText property, 127
- myAJAXlib.js, 161-162
 - callback functions, 164
 - event handler calls, 164
 - GET requests, 163
 - HTML pages, 164
 - PHP scripts, 164
 - responseText properties, 164
 - usage example, 163-165
 - XML data, retrieving, 165

N - O

namespaces, SOAP, 153

native objects, 72

nextSibling property, 133

nodeName property, 133

nodes

- child nodes, adding to DOM, 131

- DOM, 58-59

- DOM document methods table, 133*

- DOM node methods table, 133*

- DOM node properties table, 132*

nodeType property, 133

nodeValue property, 133

numcheck function (JavaScript), 46

numeric arrays, 51

objects

- Ajax

- Ajax.PeriodicalUpdater class, 179-180*

- Ajax.request class, 178*

- Ajax.Updater class, 179*

- AjaxEngine, instances in Rico, 184-185

- constructors, 72

- DOM, 72-73

- Form, prototype.js, 177

- instances, creating, 72-77

- JavaScript, 45

- methods, 37*

- XML, 57-58*

- methods, 71

- native objects, 72

- properties, 71

- XMLHttpRequest

- basic application creation example, 103-104*

- callAjax() function, 83*

- instances, creating, 74-77*

- JavaScript libraries, creating, 159-160*

- methods*

open, 78-79

send, 79

- methods, list of, 78*

- properties, list of, 77*

- responseAjax() function, 83*

- server requests, browser caches, 84-86*

- server requests, callback functions, 88-90*

- server requests, sending, 81-83*

- server requests, status monitoring, 86-87*

- server requests, timestamps, 86*

- status property, 88*

- statusText property, 88*

- uses of, 73*

XMLHttpRequest, readyState property, 86-87

onBlur event handler, 89-90

onChange event handler, 44

onClick event handler, 38-39, 44

onLoad event handler, 44, 106

onMouseOut event handler, 44

onMouseOver event handler, 41-44

onreadystatechange property, 77

onSubmit event handler, 44-46

open method, 78-79

open source libraries

- Rico, 183

- <response> elements, 184, 187*

- AjaxEngine instances, 184-185*

- callRICO() function, 186*

- cinematic effects, 192*

- drag-and-drop, 188-191*

- multiple page element updates, 184*

- usage example, 185-186*

- web applications, adding, 183*

opening lines (HTTP requests)

- HTTP versions, 26

- methods, 26-27

- server resources, 26

P

- page design, troubleshooting, 171
- page elements, designing, 171
- parentNode property, 133
- parsing, responseXML property, 96
- Permission Denied errors, troubleshooting, 174
- PHP, 47
 - \$ SERVER global array variable, 187
 - date command, 50
 - file extensions, 48
 - Hello World example, printing, 48
 - HTML, 48
 - if statements, 51
 - loop constructs, 52
 - methods, @ characters, 125
 - php tags, 48
 - program flow, controlling, 51-52
 - resource websites, 48
 - scripts
 - basic application creation example, 104-105*
 - header() instructions, outputting prior to issues, 94*
 - myAJAXlib.js, 164*
 - quotes, escaping, 94*
 - tags, 48
 - variables
 - arrays, 50*
 - case sensitivity, 49*
 - naming conventions, 49*
 - numbers, 50*
 - strings, 50*
 - values, assigning, 49*
- XOAD
 - cache handling, 200*
 - client controls, customizing, 200*
 - downloading/installing, 193*
 - events, 199*
 - header() function, 195*
 - runServer() function, 195*
 - simple page example, 194-196*
 - XOAD Controls class, 200*
 - XOAD HTML, 196-199*
- PHP interpreter, @ characters, 125
- pixels, defining, 19
- platform code tests, 172
- pop-ups, 171
- POST requests, 145-147, 165
 - HTTP requests, 29-31
 - JavaScript libraries, 160
 - message bodies, 29
 - troubleshooting, 174
- previousSibling property, 133
- programmer's editors, HTML, 14
- prologs (XML), 55
- properties, 71
 - childNodes, 132
 - DOM document methods table, 133
 - DOM node methods table, 133
 - DOM node properties table, 132
 - firstChild, 132
 - lastChild, 132
 - nextSibling, 133
 - nodeName, 133
 - nodeType, 133
 - nodeValue, 133
 - onreadystatechange, 77
 - parentNode, 133
 - previousSibling, 133
 - readyState, 77, 86-87
 - responseText, 78, 111
 - character strings, 112-114*
 - formatted data, 117*
 - HTML, 115-116*
 - manipulation methods list, 93-94*
 - myAHAHlib.js, 127*
 - myAJAXlib.js, 164*
 - null values, 92*
 - returned text, 112-114*
 - values, displaying, 92-93*
 - values, manipulating, 93*

properties

- responseXML, 78, 94-95, 130
 - parsing*, 96
 - stored values*, 130
 - web pages, adding elements to*, 130-132
- status, 78, 88
- statusText, 78, 88
- XMLHttpRequest object, list of, 77
- prototype.js
 - \$() function, 176
 - \$F() function, 176-177
 - Ajax objects
 - Ajax.PeriodicalUpdater class*, 179-180
 - Ajax.request class*, 178
 - Ajax.Updater class*, 179
 - download website, 175
 - Form objects, 177
 - getElements() method, 177
 - Rico
 - <response> elements*, 184, 187
 - AjaxEngine instances*, 184-185
 - callRICO() function*, 186
 - cinematic effects*, 192
 - drag-and-drop*, 188-191
 - multiple page element updates*, 184
 - usage example*, 185-186
 - web applications, adding to*, 183
 - serialize() method, 177
 - Stock Price Reader build example, 180-182
 - Try.these() function, 177
 - web applications, adding to, 175
- PUT requests, 145

Q - R

- query strings, GET requests, 29
- quotes, escaping in PHP scripts, 94

- readystate property, 77, 86-87
- reason phrases (HTTP responses), 27
- registerDraggable method, 188

- registerDropZone method, 188
- RemoveChild method, 133
- requests (HTTP), opening lines
 - GET requests, 29-31
 - header lines, 27
 - HTTP versions, 26
 - methods, 26-27
 - POST requests, 29-31
 - server resources, 26
- responseAHAH() functions, 122-123
- responseAjax() function, 83, 88
- responses (HTTP)
 - header lines, 28
 - reason phrases, 27
 - status lines, 27
- responseText property, 78, 111
 - character strings, using in page elements, 112-114
 - formatted data, 117
 - HTML, 115-116
 - manipulation methods list, 93-94
 - myAHAHlib.js, 127
 - myAJAXlib.js, 164
 - null values, 92
 - returned text, using in page elements, 112-114
 - values
 - displaying*, 92-93
 - manipulating*, 93
- responseXML property, 78, 94-95
 - parsing*, 96
 - stored values*, 130
 - web pages, adding elements to*, 130-132
- REST (Representational State Transfer)
 - Amazon.com REST API, 147-150
 - articles, uploading, 147
 - DELETE requests, 145
 - example of, 145
 - GET requests, 145-147
 - lists of available articles, reading, 146
 - particular articles, retrieving, 147

- POST requests, 145-147
- principles of, 144
- PUT requests, 145
- SOAP versus, 156
- stateless operations, 146
- Rico
 - <response> elements, 184, 187
 - AjaxEngine instances, 184-185
 - callRICO() function, 186
 - cinematic effects, 192
 - drag-and-drop, 188-191
 - multiple page element updates, 184
 - usage example, 185-186
 - web applications, adding to, 183
- RSS feeds, 133
- RSS headline readers, creating, 133, 136-137
 - callback functions, 138-140
 - HTML page, 134
 - server scripts, 140-141
- runServer() function, 195

S

- saving HTML documents, 15
- script tags (HTML), 34
- search engine spiders, troubleshooting, 170
- security
 - troubleshooting, 172
 - XMLHttpRequest objects, 66
- send method, 78-79
- serialize() method, prototype.js, 177
- server-side programming, defining, 10
- server-side scripts, 67
- servers
 - asynchronous communications, 64
 - requests
 - asynchronous requests, 66*
 - basic application creation example, 104*
 - browser caches, 84-86*
 - callback functions, 86*

- GET requests, 83*
- in progress notifications, 97*
- readyState property, 86-87*
- sending, XMLHttpRequest objects, 81-86*
- status, monitoring, 86-87*
- timestamps, 86*
- user feedback, 97*
- resources (HTTP requests), 26
- responses, 67
 - getElementsByTagName() method, 95*
 - in progress notifications, 97*
 - responseText property, 92-93*
 - responseXML property, 94-96*
 - user feedback, 97*
- scripts
 - RSS headline readers, creating, 140-141*
 - server-side scripts, 67*
- setRequestHeader method, 78-79
- seville.html document example, 20-21
- sizeof() function, 117
- SOAP (Simple Object Access Protocol), 151
 - development of, 152
 - namespaces, 153
 - requests
 - Ajax usage example, 155*
 - code example, 153-154*
 - components of, 152-153*
 - HTTP, sending via, 154*
 - REST versus, 156
 - specification information website, 152
- spiders, troubleshooting, 170
- split() method, 117
- src attribute (image tags), 19
- status codes table (HTTP responses), 27
- status lines (HTTP responses), 27
- status property, 78, 88
- statusText property, 78, 88
- Stock Price Reader build example, 180-182
- style sheets, setting rules, 24

styles

styles

- HTML documents, 23

- inline styles, 24

- style sheet rules, setting, 24

- substring method, `responseText` property, 93

T

table tags (HTML), 19-21

tagname properties (DOM), 59

tags

- HTML, 14-15

- adding attributes to*, 18

- anchor tags*, 20

- as containers*, 17

- body tags*, 16-18

- center tags*, 21

- common tags table*, 22-23

- head tags*, 16

- image tags*, 19

- table tags*, 19-21

- title tags*, 16

- XML, 54

testpage.html document example, 14

- body tags, 16-18

- DOCTYPE elements, 15-16

- head tags, 16

- HTML tags, 16

- loading, 15

- saving, 15

- title tags, 16

text editors, HTML, 14

text properties (DOM), 59

text-based web browsers, 10

timestamps, server requests, 86

title tags (HTML), 16

`toLowerCase()` method, `responseText` property, 93

`toUpperCase()` method, `responseText` property, 93

troubleshooting

- application design, 173

- Back button codes, 167

- bookmarks, 168-169

- browser caches, 173

- code, platform tests, 172

- GET requests, 173-174

- links, 168-169

- page design, 171

- Permission Denied errors, 174

- POST requests, 174

- security, 172

- spiders, 170

- unsupported browsers, 169-170

- user feedback, 169

- `Try.these()` function, 177

U - V

unsupported browsers, troubleshooting, 169-170

URL

- RSS headline readers, creating, 133, 136-138

- callback functions*, 138-140

- HTML page*, 134

- server scripts*, 140-141

user feedback

- basic application creation example, 109-110

- JavaScript libraries, 166

- server requests, 97

- troubleshooting, 169

valid XML documents, defining, 57

variables

- JavaScript, 44

- PHP

- arrays*, 50

- case sensitivity*, 49

- naming conventions*, 49

- numbers*, 50

- strings*, 50

- values, assigning*, 49

W

- W3C validator website, 173
- W3C website, 152
- Web (World Wide), HTTP requests, 7
- web browsers
 - availability of, 11
 - defining, 10
 - graphical web browsers, 10
 - HTML documents, loading, 15
 - text-based web browsers, 10
 - unsupported browser, troubleshooting, 169-170
 - web server interaction, 7
- web pages
 - defining, 8
 - elements, adding to via responseXML property, 130-132
 - HTTP requests, 7
 - id values, 98
- web servers
 - Apache Web Server website, 9
 - defining, 9
 - web browser interaction, 7
- web services
 - example of, 144
 - REST
 - Amazon.com REST API, 147-150*
 - articles, uploading, 147*
 - DELETE requests, 145*
 - example of, 145*
 - GET requests, 145-147*
 - lists of available articles, reading, 146*
 - particular articles, retrieving, 147*
 - POST requests, 145-147*
 - principles of, 144*
 - PUT requests, 145*
 - SOAP versus, 156*
 - stateless operations, 146*
 - SOAP, 151
 - development of, 152*
 - namespaces, 153*

- requests, Ajax usage example, 155*
- requests, code example, 153-154*
- requests, components of, 152-153*
- requests, sending via HTTP, 154*
- REST versus, 156*
- specification information website, 152*

- while loops, 52
- word processors, HTML, 14

X - Y - Z

- XML (eXtensible Markup Language), 53
 - ATTLIST declarations, 57
 - comments, displaying, 55
 - data, retrieving via myAJAXlib.js, 165
 - DOCTYPE declarations, 55-56
 - document elements, 55
 - ELEMENT declarations, 56
 - HTML, similarities to, 54
 - JavaScript, 57-59
 - prologs, 55
 - responseXML property, 130-132
 - RSS headline readers, creating, 133, 136-137
 - callback functions, 138-140*
 - HTML page, 134*
 - server scripts, 140-141*
 - server responses, 67
 - tags, 54
 - valid documents, defining, 57
- XMLHttpRequest object
 - basic application creation example, 103-104
 - callAjax() function, 83
 - instances, creating, 74-77
 - JavaScript libraries, creating, 159-160
 - methods
 - list of, 78*
 - open, 78-79*
 - send, 79*

XMLHttpRequest object

properties

readyState, 86-87

status, 88

statusText, 88

responseAjax() function, 83

security, 66

server requests, 66

browser caches, 84-86

callback functions, 88-90

sending, 81-83

status, monitoring, 86-87

timestamps, 86

server-side scripts, 67

uses of, 73

XOAD (XMLHTTP Object-oriented Application Development)

cache handling, 200

client controls, customizing, 200

downloading/installing, 193

events, 199

header() function, 195

runServer() function, 195

simple page example, 194-196

XOAD Controls class, 200

XOAD HTML, 196-199

XOAD Controls class, 200

XOAD HTML, 196

change() function, 196-197

XOAD HTML::getElementById() method,
196-197

XOAD HTML::getElementsByTagName() method,
198

XOAD HTML::getElementById() method, 196-197

XOAD HTML::getElementsByTagName() method,
198

XSLT, 120

SAMS
**Teach
Yourself**



**Fourth
Edition**

Covers HTML 4.01
and XHTML 2.0

HTML

Deidre Hayes

in **10**
Minutes

SAMS
**Teach
Yourself**

HTML

in **10**
Minutes

Deidre Hayes

Fourth Edition

SAMS

800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself HTML in 10 Minutes

Fourth Edition

Copyright © 2006 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32878-x

Library of Congress Catalog Card Number: 2005909313

Printed in the United States of America

First Printing: May 2006

09 08 07 06 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

ACQUISITIONS EDITOR
Betsy Brown

DEVELOPMENT EDITOR
Songlin Qiu

MANAGING EDITOR
Charlotte Clapp

PROJECT EDITOR
Andy Beaster

COPY EDITOR
Ben Berg

INDEXER
Heather McNeill

PROOFREADER
Mike Henry

TECHNICAL EDITOR
Gary Rebholz

**PUBLISHING
COORDINATOR**
Vanessa Evans

INTERIOR DESIGNER
Gary Adair

COVER DESIGNER
Aren Howell

PAGE LAYOUT
TnT Design, Inc.

Contents

1	What's It All About?	5
	What Is the Internet?	5
	What Is HTML?	6
	Then, What's XHTML?	7
	How Do They Work?	7
	Using Web Browsers	8
	Getting Connected	9
2	Creating Your First Page	11
	Getting Started	11
	Required Elements	12
	Saving and Viewing the Page	13
	XHTML Requirements	14
	Using Good Form	17
3	Adding Text and More	19
	Paragraphs	19
	Text Emphasis	21
	Headings	23
	Special Characters	24
	Math and Science Notations	25
	English Isn't the Only Language	26
	Meta Tags	27
4	Linking Text and Documents	32
	What Is a URL?	32
	Hyperlinks	33
	Linking to Other Files and Email	33
	Linking Within the Same Page	35

5	Adding Your Own Style	39
	Style Sheets	39
	Defining the Rules	40
	Add a Little class	41
	Applying Styles	43
	Formatting Text with Styles	46
	Adding Lines	52
6	Creating Lists	57
	Types of Lists	57
	Bulleted (Unordered) Lists	57
	Numbered (Ordered) Lists	59
	Definition Lists	62
7	Creating Tables	64
	Simple Tables	64
	Formatting Tables	65
	Advanced Tables	68
	Using Tables for Layout	68
8	Using Graphics	71
	Adding Images	71
	Adding Alternate Text	73
	Image Attributes	75
	Using Images as Links	78
	Image Etiquette	79
9	Mapping Images	82
	What Are Image Maps?	82
	Creating Client-Side Image Maps	86
	Adding Text Links for Older Browsers	87
10	Creating Frames	89
	Simple Frames	89
	Nested Frames	95
	Linking Between Frames	98
	The Two Biggest Problems with Frames	100
	Using Frames Effectively	103

11	Building Online Forms	105
	Creating Forms	105
	Form Fields	108
	Receiving Form Data	114
12	Making It Sing: Sound and Video	116
	Adding Sound and Video	116
	Finding Plug-ins	120
13	Designing with HTML	122
	Design Basics	122
	Layout, Content, and Navigation	124
	Fonts and Colors	126
	Images	129
14	Creating Active Web Pages	134
	What Are Active Web Pages?	134
	DHTML	135
	Java and ActiveX	137
	JavaScript and VBScript	138
15	Using Web Authoring Tools	142
	Why Use a Tool?	142
	Microsoft FrontPage	143
	Macromedia Dreamweaver	146
	Other Popular Web Tools	149
16	Making a Name for Yourself	151
	Web Hosting	151
	Search Pages and Indexes	152
	Adding Your Web Site to the Search Engine	155
	Advertising	157
17	Planning for the Future	158
	The Future of the Internet	158
	What Is XML?	158
	Being Prepared	163

A	HTML/XHTML Quick Reference	169
	Required Elements	170
	Text Phrases and Paragraphs	173
	Text Formatting Elements	175
	Lists	176
	Links <code><a>...</code>	177
	Tables	178
	Frames	180
	Embedded Content	182
	Style <code><style>...</style></code>	184
	Forms	184
	Scripts	189
	Common Attributes and Events	190
B	Style Sheet Quick Reference	192
	Text and Fonts	193
	Typography	196
	Colors and Backgrounds	198
	Borders and Tables	201
	List	204
	Layout	205
C	Special Characters	205
	Symbol Entities	205
	Character Entities	207
	Greek Entities	210
	Other Entities	213
	Index	215

About the Author

Deidre Hayes is an information architect with a Web services group that created and manages a very successful corporate intranet. She is continually looking for ways to increase productivity with online workflow technologies and has spoken to national audiences on her favorite Web-related topics: information design and usability. She is a member of the Society for Technical Communications, the Usability Professionals Association, and the HTML Writers' Guild.

Acknowledgments

I would like to thank my family—especially my beautiful daughter, Alexandra—for their patience. Thanks to my friends and colleagues for challenging me everyday. And, thanks also to the editing staff at Sams Publishing for their dedicated help.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: webdev@sampublishing.com

Mail: Mark Taber
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.sampublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

If you're reading this book, you must have some idea of what HTML is, right? Maybe you already know that HTML is the language of the Internet and that far from being a complex programming language requiring years to perfect, HTML is actually a simple markup language that you can learn very quickly. XHTML is the latest version of HTML. You'll learn more about how these two standards work together to create Web pages in later chapters.

You're probably thinking that if you knew how to create documents in HTML, you could help your company earn more money, or better yet, help you earn more money.

What you probably don't know is how to get started. How do you learn that language and what's it going to cost?

What You Need to Know Before Using This Book

Guess what? You can create HTML documents on any computer system because HTML works the same on any type of computer. Even better, you can use software that you already own to do it. Any kind of text editor (such as Microsoft Notepad) can be used.

Because we're covering a lot in 10 minutes, it will certainly help as you go through this book if you already have some basic computer skills (including the ability to use a word processor, some understanding of how to use directories and filenames on your computer system, and some experience using a Web browser such as Netscape or Internet Explorer).

About the *Sams Teach Yourself in 10 Minutes* Series

Sams Teach Yourself HTML in 10 Minutes uses a series of lessons that walk you through the basics of HTML, and then moves on to more advanced features of the language. Each lesson is designed to take about

10 minutes and each is limited to a particular feature, or several related features, of the HTML language. There are plenty of examples and screen shots to show you what things look like. By the time you finish this book, you should feel confident in creating your own HTML documents for the World Wide Web. You can even use HTML to provide unique and valuable services to your organization, or to tell the world about yourself.

Special Sidebars

In addition to the explanatory text and other helpful tidbits in this book, you will find icons that highlight special kinds of information.



Plain English sidebars appear whenever a new term is defined. If you aren't familiar with some of the terms and concepts, watch for these flagged paragraphs.



Caution sidebars alert you to common mistakes and tell you how to avoid them. These paragraphs also explain how to undo certain features, and highlight differences between HTML and XHTML.



Tip sidebars explain shortcuts (for example, key combinations) for performing certain tasks.



Note sidebars present pertinent pieces of information related to the surrounding discussion.

Conventions Used in This Book

The creation and editing of HTML documents can be done using any one of a wide variety of editing tools. You'll find many excerpts from HTML documents that illustrate the points being made. These fragments look like this:

```
<html>
<head><title>This is the Title of Your Page</title></head>
<body>This is the document text surrounded
by HTML tags.</body>
</html>
```

If you're working along with the examples, you might want to enter the HTML fragments into your own HTML documents as you move through the lessons.

Web Browser Screen Shots

Web browsers (such as Internet Explorer and Netscape) are used to interpret HTML documents for your computer. There are many different types of Web browsers (some with more bells and whistles, some with less), but they all do essentially the same thing. You'll find out about some of these differences (and how to avoid problems) as we move through the lessons in this book. To avoid confusion, all the Web browser screen shots in this book were taken from Internet Explorer.

This page intentionally left blank

LESSON 1

What's It All About?



In this lesson, you will learn how the Internet works and why HTML and XHTML are so important.

What Is the Internet?

Like many inventions, the Internet began as the solution to a problem. It began with the government's need to find a way to link several computer networks together so that files could be shared. In other words, it created a network of networks. These computer networks were located all over the world and sharing information the old-fashioned way took a long time. Today, the idea of sharing files with people around the world doesn't sound like such a big deal when almost everyone has the modem, e-mail, and dial-up connections that make *Wide Area Networks (WANs)* commonplace. Back then, however, no one had even considered the idea. So, how did they do it? Well, researchers working for the *Advanced Research Projects Agency (ARPA)* created ARPAnet, which became the first WAN. Eventually, this led to an *Internet Protocol (IP)*—a common computer language—enabling all computers to talk to each other.



Internet Protocol (IP) A predefined set of rules used to enable computers to communicate with each other, regardless of which operating system they are running.

This protocol and the new network of networks made exchanging information much easier than ever before, but it still wasn't simple. To find information on the Internet, you had to know where it was stored. You first had to understand how all the computers were connected, and then you had to navigate through the network to find the data you were looking for.

All that changed in the early 1990s. At that time, a new protocol was created. That protocol, the *Hypertext Transfer Protocol (HTTP)*, enabled information on the Internet to be accessed from *anywhere*, by *anyone*. It's what allows you to jump from one Web page to another by pointing and clicking. The code that makes up the HTTP was a breakthrough, but it can't do everything by itself. The information stored on the computers in the network (the documents and data) must include its own set of communication tools so that the other computers in the network can interpret it. In the case of the World Wide Web, the communication tool is *HTML*.



HTML Stands for *Hypertext Markup Language*. Most documents that appear on the World Wide Web were written in HTML.

What Is HTML?

In the Introduction, you learned that HTML is a markup language, not a programming language. In fact, the term *HTML* is an acronym that stands for *Hypertext Markup Language*. You can apply this markup language to your pages to display text, images, sound and movie files, and almost any other type of electronic information. You use the language to format documents and link them together, regardless of the type of computer with which the file was originally created.

Why is that important? You know that if you write a document in your favorite word processor and send it to a friend who doesn't have that same word processor, your friend can't read the document, right? The same is true for almost any type of file (including spreadsheets, databases, and bookkeeping software). Rather than using some proprietary programming code that can be interpreted by only a specific software program, HTML is written as plain text that any Web browser or word processing software can read. The software does this by identifying specific elements of a document (such as heading, body, and footer), and then defining the way those elements should behave. These elements, called *tags*, are created by the *World Wide Web Consortium (W3C)*. You'll learn more about tags in upcoming lessons.



Tags These are elements of a Web page that are used to define how those pages should behave. They are most often used in pairs, which surround the element they are defining.

World Wide Web Consortium (W3C) Members of this group develop the protocols that make up the World Wide Web. Currently, the W3C has 180 members from commercial, academic, and governmental organizations worldwide.

Then, What's XHTML?

XHTML, an acronym for *eXtensible Hypertext Markup Language*, is the first big change to HTML in years. With it, the W3C is trying to add the structure and extensibility of *XML* to HTML pages. By adding a few simple structural elements to existing HTML pages, you can be assured that your Web pages are compatible with later versions of HTML, and even with XML. Lesson 2, "Creating Your First Page," has all the information you need to get started.



XHTML Stands for *eXtensible Hypertext Markup Language*. It is the next generation of HTML.

XML Stands for *eXtensible Markup Language*. It is the newest language being developed by the W3C, and is also the most flexible. You'll learn more about it in Lesson 17, "Planning for the Future."

How Do They Work?

Markup languages such as HTML and XHTML serve another important purpose when it comes to sharing information over long distances: Information comes to you faster because your computer (using a Web browser) does the work of interpreting the format of the information after you receive the page. Sound confusing? Well, let's look at it another way.

Your computer has a Web browser, such as Internet Explorer or Netscape Navigator, installed on it. When you are looking for information on the Web, your browser has to find the computer that is storing that information. It does this using the HTTP. The storage computer, or server, then sends the new Web page (as a plain text file) back to your computer using the same HTTP. Your browser sees the new Web page and interprets the text and HTML tags to show you the formatting, graphics, and text that appear on the page.



Tip HTTP isn't the only protocol used on the Internet. Each protocol is used for a specific network service, such as electronic mail or file transfers.

Using Web Browsers

As great as Web browsers are, you should be aware of some limitations. Although all HTML commands are the same, not all browsers interpret the commands in the same way. Some browsers, such as Lynx, can display only text (even if the HTML author added images to the document). Some older browsers do not understand the newer HTML commands and might produce errors rather than text. What's more, some of the newest browsers enable viewers to determine which window display sizes, fonts, and colors they prefer when viewing Web pages (even if those settings are different from what you, the Web author, want them to see). Don't despair; there is good news. Most Web pages look the same, or almost the same, on every browser regardless of the computer system: PC, Macintosh, or UNIX. With each lesson in this book, you'll find tips to help ensure that your pages are viewed as you intended. Keep these tips in mind as you create your own Web pages, and you'll avoid the disappointment that many novice Web authors face as they realize that the page they worked so hard on looks awful on another computer or browser.



Tip The Web itself offers Web page designers the opportunity to preview their pages on a number of different browsers at one time. Web sites, such as AnyBrowser (www.anybrowser.com), show you exactly how each browser will interpret your page. You can use this information to redesign your page and help ensure that most people see it the way you intend.

Getting Connected

This might be apparent, but sometimes it pays to state the obvious: Although you can create Web page files in any plain text editor and view them in any browser, you have to decide how you are going to store the files. You already know that you can't surf the Net without having an *Internet Service Provider (ISP)*. In the same way, you need a *Web Presence Provider (WPP)*—or your own Web server—to store your pages before they can be viewed from the Web. Other ways to view Web pages also exist. Table 1.1 describes the methods you can use to store your files.

TABLE 1.1 Storing and Viewing Your Documents

If You Store Your Files On	They Can Be Viewed by People with Access To
Your own computer	Your computer (or an intranet)
A disk or CD-ROM	That disk or CD-ROM
A Web host server	The World Wide Web



Internet Service Provider (ISP) A company that provides you with access to the Internet.

Intranet This is like your own private Internet in that it uses the same HTTP as the World Wide Web, but it is accessible only by people within your own network.

Web Host A company that stores (hosts) information that can be accessed from the Internet using the HTTP. A Web host may also be called a *Web Presence Provider (WPP)*.

In this lesson, you've learned:

- HTML and XHTML are markup languages that define the structure, rather than the format, of the text elements in your documents.
- HTML is platform independent. As long as they have a browser, your Web site visitors can see the same Web page on a PC, a Macintosh, or a UNIX computer.
- XHTML, the latest version of HTML, requires more structure than HTML.
- You need a Web server or a Web Presence Provider to store your pages before they can be viewed from the Web.

LESSON 2

Creating Your First Page



In this lesson, you will learn to create, save, and view simple Web pages.

Getting Started

I think you'll find that the best way to learn is to follow along with the examples in this book and create your own Web pages as you read. As you learned in the introduction of this book, you can create Web pages or HTML documents with any text editor (including Microsoft Notepad, DOS edit, Mac SimpleText, and UNIX vi). You probably already have at least one of these editors installed on your computer, even if you have never used it before.



Caution Although you can also create Web pages using some word processors (such as Microsoft Word) and some WYSIWYG programs (such as Microsoft FrontPage), I suggest that you ignore these programs for now and concentrate on learning HTML. HTML authoring tools are discussed in Lesson 15, "Using Web Authoring Tools."



WYSIWYG An acronym for *What You See Is What You Get*. It generally refers to software programs that enable you to see what the page looks like without seeing all the program's formatting codes.

Required Elements

To see what HTML looks like and learn the most basic HTML tags, let's look at a very simple HTML document. Figure 2.1 shows a simple Web page in Microsoft Notepad. You can type the same text in your own editor to follow along with the lesson.



FIGURE 2.1 The `<html>` and `</html>` tags are all you need to identify your file as an HTML file.

Every HTML document must begin with the `<html>` tag and end with its complement, the `</html>` tag. In addition to the `<html>` tag, this document includes three other pairs of tags that should be included in any HTML document:

- The `<head>` and `</head>` tag pair is used to indicate any information about the document itself. You'll learn how to add some of this information in later lessons.
- The `<title>` and `</title>` tags are used to add a title to your browser's Title bar. The Title bar is the colored band at the top of any application that gives the name of the application.
- The `<body>` and `</body>` tags are used to surround any text that appears in the HTML page.

All HTML documents are separated into two parts: the head and the body. Because the title contains information about the document, the `<title>` and `</title>` tags are placed within the `<head>` and `</head>` tags.

One More Page

If you were to create another simple HTML page, you would see that the same four tags are present in this document as well. Only the text that appears between tags is changed.

```
<html>
<head>
<title>My Second Web Page</title>
</head>
<body>
<p>This is my second Web page.</p>
</body>
</html>
```



Tip Most HTML tags come in pairs. You use the first tag in the pair (for example, `<html>`) to tell the computer to start applying the format. The second tag (for example, `</html>`) requires a slash in front of the tag name that tells the computer to stop applying the format. The first tag is usually referred to by the name within the bracket (for example, *HTML*). You can refer to the second tag as the end, or the close, tag (for example, *end HTML*).

Saving and Viewing the Page

To view your own page in a browser, you must first save it. Because you've created an HTML document, you want to save your file with an .htm extension (first.htm, for example) so that you recognize it quickly.



Caution Some people prefer to name their HTML files with an .html extension (for example, first.html). Some older computer systems, however, still require the file extension to be three characters or fewer and might have trouble reading (or storing) a file with a longer extension.

You can preview any HTML file in your browser, even when that file is stored on your computer rather than on a Web server. In Internet Explorer, you can view your new file by selecting Open from the File menu. Figure 2.2 shows how Internet Explorer displays the first.htm file that you created in Figure 2.1.



Tip Although you don't see them, HTML commands are sitting behind the scenes of every document that you open in your Web browser. You can see the HTML commands by selecting Source from the View menu of Internet Explorer (other browsers might use different menu commands). When you find a page on the Web that you like, you can view the source code to learn how you can use HTML to create something similar.

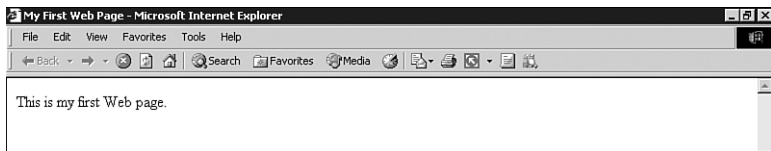


FIGURE 2.2 My First Web Page as it appears in the Internet Explorer browser. Notice that the Title bar contains the text between the `<title>` and `</title>` tags and the body of the browser contains the text between the `<body>` and `</body>` tags.



Caution Some Web pages use frames to display more than one HTML page at the same time. (See Lesson 10, "Creating Frames.") To view the source code for this type of page, make sure that you use your mouse to highlight some portion of the page you're interested in before selecting Source from the View menu.

XHTML Requirements

XHTML, the latest revision of HTML, adds another required element to your Web pages: the `<!DOCTYPE>` tag. This tag appears at the top of the file and identifies the file as an HTML document conforming to the XHTML requirements. If you were to create an XHTML-conforming document, it would look like the following:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<title>My XHTML Page</title>
</head>
<body>
<p>This is my first XHTML page.</p>
</body>
</html>
```

The `<!DOCTYPE>` tag has three variations: Strict, Transitional, and Frameset. You declare which one you are using in the top of the file.

- **Strict** Declare this variation when you are certain that your viewers will be accessing your pages from newer browsers that interpret style sheets correctly. You'll learn more about style sheets in Lesson 5, "Adding Your Own Style." The Strict variation looks like this:

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

- **Transitional** Declare this variation when you are not certain how your viewers will be accessing your pages.

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

- **Frameset** Declare this variation when you are working in frames. You will learn more about frames in Lesson 10.

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



Caution This book uses the Transitional variation of the `<!DOCTYPE>` tag throughout. It's a good habit to get into, allowing you to conform to W3C rules, but offering more flexibility than the Strict variation.

You might have noticed one more change from the HTML required elements: The `<html>` tag has some new attributes: `xmlns`, `xml:lang`, and `lang`. In HTML, you only have to include the `<html>` tag to identify the document as an HTML file, but XHTML requires that you use the `xmlns` attribute to link your document to the W3C's definition of XHTML, which continues to evolve. You will learn more about this evolution and how to prepare for it in Lesson 17, "Planning for the Future." For now, just remember to include the `<!DOCTYPE>` tag and the full `<html>` tag (shown in the following sample) in all your Web pages. Figure 2.3 demonstrates how the XHTML page, created previously, would appear in the browser.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
```



FIGURE 2.3 Notice that adding the XHTML declaration does not affect your page's appearance.



Caution The `<!DOCTYPE>` tag is the only tag that appears in uppercase. All other HTML tags should be lowercase as explained in the next section.

Using Good Form

Tags aren't the only things that make a good Web page. As you continue through the lessons in this book, you'll discover that while HTML was very forgiving, XHTML must conform to the rules. Though current versions of the most popular browsers will recognize your intentions even if you use incorrect tags (or enter the correct tags in the wrong order), later versions will not. You'll want to move beyond the novice level now and follow some basic Web coding principles to conform to XHTML's standards. Following is a brief list of those principles, but you'll learn more in later lessons:

- *Include all the required XHTML elements that you learned in this lesson* You might want to create a template for yourself that already includes these tags. You can use the XHTML document created in the "XHTML Requirements" section as a template. Whenever you create a new Web page; open your template file, add your new text, and save the new file.
- *Use lowercase for all tags* To the browser, <HEAD>, <Head>, and <head> all mean the same thing. (That won't always be true.) Use the same lowercase spelling for all your commands and you won't be caught having to recode your pages as the standard evolves.
- *Never use spaces in filenames* Older computer systems have trouble reading filenames that include spaces (for example, *my first page.htm*). Instead, you can use a couple of file management tricks to replace the spaces:
 - Use an underscore (_) to represent spaces (for example, *my_first_page.htm*).
 - Use initial capital letters to indicate new words in a filename (for example, *MyFirstPage.htm*).

Table 2.1 shows a list of the tags that you learned in this lesson. A similar table of new HTML tags appears at the end of other lessons.

TABLE 2.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<!DOCTYPE>		Begins each XHTML document and includes a reference to the Strict, Transitional, or Frameset variation.
<html>	</html>	Surrounds all the text in an HTML file. XHTML documents must include the xmlns, xml:lang, and lang attributes.
<head>	</head>	Contains information about the document. Must include the <title> tag.
<title>	</title>	Identifies the title of the page.
<body>	</body>	Surrounds the text of the page.

In this lesson, you've learned:

- Any text editor, including Microsoft Notepad, can be used to create Web pages (or HTML documents).
- All HTML documents are separated into two parts: the head and the body.
- Every HTML document must include the <html> tag and end with its complement, the </html> tag.
- Every XHTML document must include a valid variation of the <!DOCTYPE> tag before the <html> tag.
- All HTML tags (except the <!DOCTYPE> tag) should be typed in lowercase.

LESSON 3

Adding Text and More



In this lesson, you will learn how to use HTML to add text and headings in your Web pages. You'll also learn how to add mathematical notations, information about your Web page, and special characters (such as ampersands).

Paragraphs

You might not realize it, but you already learned how to create an HTML paragraph in Lesson 2, “Creating Your First Page.” In HTML, a paragraph is created whenever you insert text between the `<p>` tags. Look at the code from Lesson 2 again:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title>My XHTML Page</title>
</head>
<body>
<p>This is my first XHTML page.</p>
</body>
</html>
```

Web browsers see that you want text and they display it. Web browsers don't pay any attention to how many blank lines you put in your text; they only pay attention to the HTML tags. In the following HTML code, you see several lines of text and even a blank line, but the browser only recognizes paragraphs surrounded by the `<p>` and `</p>` tags (or paragraph tags). The `<p>` tag tells the browser to add a blank line before displaying any text that follows it, as shown in Figure 3.1.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
  transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">

<head>
<title>Typing Paragraphs in HTML</title>
</head>
<body>
<p>This is the first line.

But is this the second?</p>
<p>No, this is.</p>
</body>
</html>
```

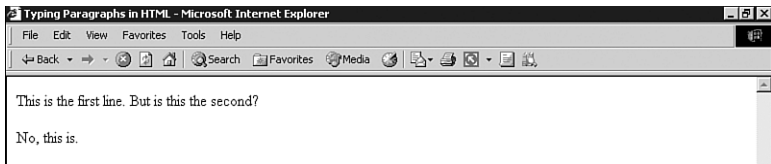


FIGURE 3.1 The browser ignores the blank line that I inserted and puts the line break before the `<p>` tag instead.

Web browsers do something else with paragraph text that you should be aware of: They wrap the text at the end of the browser window. In other words, when the text in your Web page reaches the edge of the browser window, it automatically continues on the next line regardless of where the `<p>` is located. The `<p>` tag always adds a blank line, but you might not always want a blank line between lines of text. Sometimes you just want your text to appear on the next line (such as the lines of an address or a poem). You can use a new tag for this—the line break, or `
` tag, shown in Figure 3.2.

This new tag forces the browser to move any text following the tag to the next line of the browser, without adding a blank line in between. Figure 3.3 shows how the browser uses these two tags to format your text.

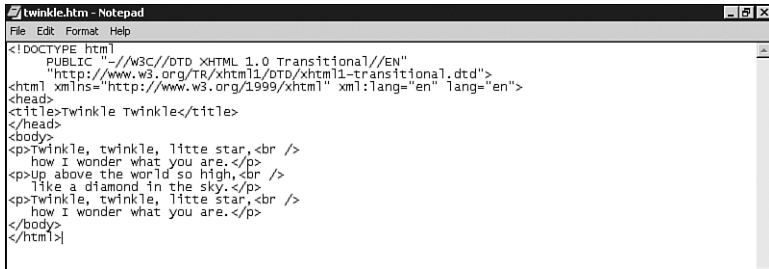


FIGURE 3.2 The <p> and
 tags help to separate your text into lines and paragraphs.

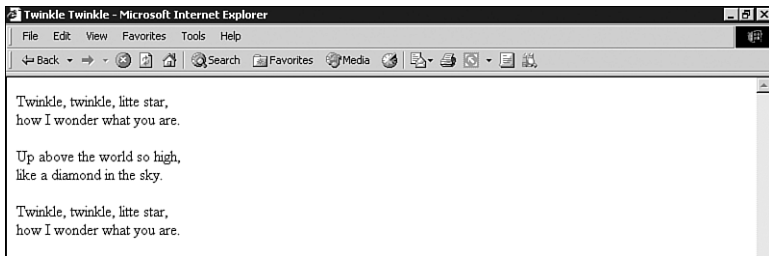


FIGURE 3.3 The browser inserts line breaks and blank paragraph separators only where you place the correct HTML tags.

Text Emphasis

So far you've learned how to add text, but here you will learn how to format it. You will occasionally want to add emphasis to your text to make it stand out. HTML enables you to quickly apply some standard formats, such as boldface and italic, using a predefined set of tags. All these tags occur in pairs (corresponding opening and closing tags) and must surround the text that they are emphasizing. Use the code that follows in your own Web page to see how each of these tags appears in the browser.

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

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<title>Emphasizing Text</title>
</head>
<body>
<p>Make your text one size larger with the
    <big>big tag.</big></p>
<p>Try the <b>bold tag</b> or the <strong>strong tag</strong>
    to make an impact.</p>
<p>The <i>italics tag</i> and the <em>emphasis tag</em>
    create a different impact.</p>
<p>Use the <tt>teletype tag</tt> to imitate a typewriter.</p>
<p>Make your text one size smaller with the
    <small>small tag.</small></p>
</body>
</html>
```



Caution Other formatting tags exist in HTML, but their use is discouraged in HTML and deprecated in XHTML in favor of style sheets. The World Wide Web Consortium (W3C) has determined that HTML should be used to identify types of information (text, headings, tables, and so on), but should not be used to format that information.



Deprecated Some older HTML tags, specifically related to formatting, have been replaced by the formatting capabilities of style sheets.

Style Sheets Web developers use style sheets to specify formatting instructions for a single document or a group of documents.

You'll learn how to create style sheets in Lesson 5, "Adding Your Own Style." Throughout the book, however, you'll see how styles can enhance your Web pages.

Headings

Separating your text into paragraphs isn't the only way to format your Web pages. HTML enables you to add six different heading tags to your pages by using the tags `<h1>`–`<h6>`. These tags are very simple to use. Look at the following line of code:

```
<h1>This is Heading 1</h1>
```

The closing heading tags also create an automatic paragraph break. In other words, all headings automatically include a blank line to separate them from the text. Heading 1, the `<h1>` tag, has the largest font of the heading tags and Heading 6, the `<h6>` tag, has the smallest. In fact, you usually only see Web page authors use the `<h1>`–`<h3>` tags because the remaining tags, `<h4>`–`<h6>`, are actually smaller than normal text. Figure 3.4 shows a sample of all the heading tags compared to normal text.



FIGURE 3.4 Notice that HTML's Heading 4 is the same size as normal text, but Headings 5 and 6 are actually smaller.



Tip Unless you or the people viewing your pages have adjusted the browser's default settings, normal HTML body text appears in 12 point Times New Roman font on most computer systems.

Special Characters

You might find that you sometimes need to use symbols on your Web pages. Symbols (such as +, −, %, and &) are used frequently in our everyday writing, so it's easy to understand that they would appear on a Web page as well. Unfortunately, not all Web browsers display these symbols correctly. HTML uses a little computer shorthand, either using a numerical code or a text code (called an *entity character reference*) to tell the browser how to interpret these symbols. Table 3.1 shows some of the most frequently used codes.

TABLE 3.1 Special Character Codes

Char	Code	Description
&	&	Ampersand
<	<	Less than
>	>	Greater than
©	©	Copyright
®	®	Registered trademark
±	&plusmin;	Plus or minus
²	²	Superscript 2
³	³	Superscript 3
´	´	Acute accent
`	`	Grave accent
#	#	Number
%	%	Percent

Appendix C, “Special Characters,” contains a more complete list of the characters supported by HTML. You can see how many of these symbols are easy to understand (for example, & for the ampersand and > for the greater than symbol). Some of the characters, such as number and

percent, require that you memorize numbered codes. Yuck. The best thing you can do is to make sure that you preview your Web pages in a variety of browsers before publishing them.



Tip Here's a special character that you should remember: ` `. The symbol stands for *nonbreaking space* and is used to insert a space inside an HTML document. Because HTML ignores extra spaces between words and tags, you'll need to have a way of including extra spaces. You can do that with the ` ` character.

Math and Science Notations

Although HTML was first designed and used by scientists, it has yet to support mathematical and scientific notation with any degree of complexity. HTML does give you two tags to help write simple equations. Together with the codes for special characters, the `<sub>` (subscript) and `<sup>` (superscript) tags go a long way toward creating equations, as shown in Table 3.2.

TABLE 3.2 `<sup>` and `<sub>` Tags

You Type	The Browser Displays
<code>A<sup>2</sup> + B<sup>2</sup> = C<sup>2</sup></code>	$A^2 + B^2 = C^2$
<code>CO<sub>2</sub> = Carbon Dioxide</code>	$CO_2 = \text{Carbon Dioxide}$

If you are looking to write more complex equations, you need to be a little more creative. The obvious answer is to write your equation in the program that you usually use, and then use a graphics program to turn it into an image. You can insert that image into any HTML page, as you've already learned. That works, but the solution is limited. Because the equation is graphical, you are not able to index or search for text within the equation. That's a big drawback, but so is the fact that images slow down

your page's load time and the fact that your equation cannot be viewed by nongraphical Web browsers.



Tip Some commercial products are available to help you notate mathematical expressions. You can see a list of them on the W3C Web site (www.w3.org/Math/).

English Isn't the Only Language

You can use HTML even if you don't write in English. URLs, hyperlinks, HTML tags, and document formatting elements are language neutral, but text requires a specification all its own. If you write in standard U.S. English, you don't need to make any changes to the way you create your HTML documents. If you are writing text in any other language, however, you should specify the language for the browser. The following HTML samples show the designations for German and French.

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

and

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

The language attributes (`xml:lang` and `lang`) support the same values as ISO, the International Standards Organization. You can see the full list of supported languages and their codes at www.loc.gov/standards/iso639-2/langcodes.html.



Tip Why is language important? Browsers do not recognize the language you type unless you use the `lang` attribute. Some search engines use the `lang` attribute to return only pages written in a specific language. Speech synthesizers use this information to aid in pronunciation. Even some spelling checkers can use the information to recognize misspellings.

Mixing Languages in a Single Page

Although the preceding example shows the `lang` attribute used as part of the `<html>` tag at the top of your document, it's possible that you would want to include text of one language within a document of another language—for example, including a paragraph in French within a document in English. You can assign the `lang` attribute to the `<p>` tag to solve this problem. Look at the following sample:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
    transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<title>Multi-Language Document</title>
</head>
<body>
<p>Put your English text here.</p>
<p lang="fr">Mettez votre texte français ici.</p>
<p lang="en">Put the rest of your English text here.</p>
</body>
</html>
```

Meta Tags

Finally, you get to do something with the `<head>` tag. So far, you've only seen the `<title>` tag used to give information about the document, but you can do a lot more with the `<head>` tag. What's more, aside from the `<title>` tag, information in the `<head>` tag doesn't usually appear in your document. You can use the meta information tag (`<meta>`) to identify the page's author, keywords used for searching, or a brief description to appear in search results. You also can use the `<meta>` tag to give commands to the browser. You can use as many `<meta>` tags as you like in your page. You'll learn how in the sections that follow.

Improved Searching

Search engines (as you'll find in Lesson 17, "Planning for the Future") add the content of your Web pages to their indexes. When a potential

visitor enters a search phrase, the search engine checks its index to find that word and returns any pages that include that word. It works great. But, what if you were a realtor and you worked hard at creating a Web page that included the words *houses*, *housing*, *sale*, and *buy*; but didn't include the phrase *real estate*? If that was the phrase your visitor was looking for, they would never find your page. You can use the `<meta>` tag to include product names, geographic locations, industry terms, and synonyms that people might be searching for. There are three `<meta>` tags that work to help improve your chances of being found by a search engine:

- **Keywords** Keywords are words that you feel people might use to search for your Web page, or synonyms for words that appear in your document.
- **Description** This is usually a paragraph of information about your page. Some search engines use the information in this tag to summarize your page, but other search engines use the first few lines of text in your actual document.
- **Author** This is your opportunity to shine. Just in case someone is searching for your name, they will find your page if you enter that information into the `<meta>` tag.

Meta information for search engines comes in pairs: name and contents. The following HTML code includes meta information pairs for each of the preceding `<meta>` tags.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
  transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>
<title>Your HTML Page</title>
<meta name="keywords" contents="words that people might
  use to search for your page." />
<meta name="description" contents="a brief paragraph describing
  your document." />
<meta name="author" contents="your name" />
</head>
<body>
```

```
<p>Insert your text here.</p>
</body>
</html>
```



Caution Remember, `<meta>` tags only appear in the `<head>` section of an HTML file.

Refresh and Redirect

There might be times when you want to replace one page with another or want to redirect a link. You might, for example, choose to include a *splash page* on your Web site. You can use the meta information to force the page to change within a given time span using the sample code that follows:

```
<meta http-equiv="refresh" content="time in seconds,
                                   URL of the new page" />
```



Splash Page The introductory page used by some Web page authors to show flashy graphics or a product logo before continuing to the rest of the site's contents.

If you have a page that you update several times a day and you want to make sure that people always see the most recent version, you can enter the page's own URL in the refresh tag. When the browser sees the refresh tag, it presents the requested URL in the specified time.

```
<meta http-equiv="refresh" content="time in seconds,
                                   URL for this page" />
```



Caution Because not all Web browsers support this attribute, authors should include some content on the splash page that enables users to move to the next page on their own.

Expiration Dates

If you have a page that you change frequently, you can specify an expiration date in the `<meta>` tag to ensure that the Web browser looks for a newer version (rather than displaying an older version, which might still be stored in the browser's memory). Look at the example that follows:

```
<meta http-equiv="expires"
      contents="Wed, 04 December 2006 00:00:00 GMT" />
```

When you enter the URL for this page in your browser, it checks its history files to see whether a copy is stored there. If so, it checks the meta information to see whether this page is still valid. If the expiration date has passed, the browser looks to the Web for a more recent copy before displaying the page.

Table 3.3 reminds you of the formatting tags you learned in this lesson.

TABLE 3.3 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code></code>	<code></code>	Text appears boldface.
<code><big></code>	<code></big></code>	Text appears one size larger than normal.
<code>
</code>		Line break. Forces text to the next line.
<code></code>	<code></code>	Text appears emphasized (italic). Usually the same as <code><i></code> .
<code><h1></code>	<code></h1></code>	A first-level heading.
<code><h2></code>	<code></h2></code>	A second-level heading.
<code><h3></code>	<code></h3></code>	A third-level heading.
<code><h4></code>	<code></h4></code>	A fourth-level heading. Rarely used.
<code><h5></code>	<code></h5></code>	A fifth-level heading. Rarely used.
<code><h6></code>	<code></h6></code>	A sixth-level heading. Rarely used.
<code><i></code>	<code></i></code>	Text appears emphasized (italic).
<code><meta /></code>		Identifies information about the document.

HTML Tag	Closing	Description of Use
<code><p></code>	<code></p></code>	Paragraph break. Forces a blank line.
<code><small></code>	<code></small></code>	Text appears one size smaller than normal.
<code></code>	<code></code>	Text appears boldface. Same as <code></code> .
<code><sub></code>	<code></sub></code>	Text appears in subscript.
<code><sup></code>	<code></sup></code>	Text appears in superscript.
<code><tt></code>	<code></tt></code>	Text appears monospaced, as if typed.

In this lesson, you've learned:

- The `<p>` tag, or paragraph tag, tells the browser to add a blank line before it displays any text that follows. The `
` tag moves your text to the next line without adding a blank line.
- HTML enables you to add emphasis to your text with several predefined formatting tags.
- Symbols such as +, −, and % require a little computer shorthand to tell the browser how to interpret these symbols. This shorthand begins with an ampersand (&) and ends with a semicolon (;). A more complete list can be found in Appendix C.
- You can add other languages to your HTML documents by using the `lang` attribute on the `<html>` tag.
- Meta information for search engines comes in pairs: name and contents, and the `<meta />` tags always appear between the `<head>` tags.



LESSON 4

Linking Text and Documents

In this lesson, you will learn how to use HTML's most valuable feature: hyperlinks.

What Is a URL?

Ask anyone and they'll tell you that (far and away) the feature that makes HTML so worthwhile is the ability to *hyperlink* from one place to another. All Web pages, Internet resources, files, and so on, have an address. That address is known as a *Uniform Resource Locator*, or *URL*. Before you can link to another page (or resource), you have to know its address. You can find the URL for any resource in the Address box (or Location box) of your browser.



Hyperlink The text that enables you to jump from a Web document to another location.



Caution Although *URL* is the commonly accepted term to describe the location of Internet resources, a new term, *URI (Uniform Resource Identifier)*, will likely replace it as XML becomes the standard. You'll find out more about XML in Lesson 17, "Planning for the Future."

The `<a>` tag (called an *anchor*) is used to define hyperlinks. Unlike most other HTML tags, the `<a>` tag *requires* an attribute. When you use the `<a>` tag, you must specify whether you want the enclosed text to link *to*

someplace (with the `` tag) or be linked *from* someplace (with the `` tag). The following section provides some examples.

Hyperlinks

The easiest link to learn is the hyperlink to another location. The `<a>` tag with the `href` attribute and its closing tag, ``, surround any text that you want to highlight. The default hyperlink highlighting in HTML is underlined blue text. In the following example, you would click on the words `click here` to jump (hyperlink) to the document found at the URL inside the quotes (`http://www.microsoft.com`).

Please `click here` to open the Microsoft Web site.



Tip Did you know that you can copy the URL of any Web page from your browser? Just highlight the address in the Address box (or Location box) and select Edit, Copy (or press Ctrl+C). Then, select Edit, Paste (or press Ctrl+V) to paste the address between the quotes of the `href` attribute.

Linking to Other Files and Email

You can link to more than just other people's Web sites. You can use the same `href` attribute to link to email addresses for other pages of your own Web site, or even to other files on your own computer. The hyperlink to point to another file (`second.htm`) on *my* own computer, for example, is shown in the following code. In this example, the `second.htm` file is stored in the same directory as the page linking to it.

Please `click here` to open my second Web page.

If, however, my `second.htm` file was stored in another directory (for example, the `Links` directory), the hyperlink would need to include the directory name too, as in the following:

Please `click here` to open my other page.



Tip Did you know that you can force your hyperlink to open a new browser window? This is especially handy if you want to link to someone else's Web site without directing traffic away from your own site. Use the `target="_blank"` attribute, as in the following example. Try it!

```
<a href="http://www.somewhere.com/page.htm"
target="_blank">
    Click here</a> to open a related Web site.
```

The `href` attribute changes slightly if you want to link to a file that is not part of your Web site. You need to tell the Web browser that the file is not located on the Web server. You can see how that is accomplished in the following example:

```
<a href="file:\\servername\\foldername\\filename.extension">
    Click here</a> to open my favorite file.
```

If I want to link to my `dogs.doc` file in the `4legs` folder of my animals server, for example, my hyperlink looks like the following:

```
<a href="file:\\animals\\4legs\\dogs.doc">click here</a>
    to open my favorite file.
```



Caution Did you notice that the direction of my slashes changes when I change my link type from `http://` to `file:\\`? The forward slash (/) is always used to separate directory folders on a Web file server. The backslash (\) is used to separate directory folders in Windows and DOS.

You also can link to an email address by using the `mailto:` prefix, as shown in the following code line. When you click on the words `click`

here, an email window that enables you to type your message to Mickey Mouse appears.

```
<a href="mailto:mickey.mouse@disney.com">Click here</a>  
to send mail to Mickey.
```

Linking Within the Same Page

Now that you know how to link to other resources, you might want your hyperlinks to be more meaningful. HTML enables you to use hyperlinks to point to a specific spot (or anchor) in an HTML document, instead of just pointing to the entire document. As an example, suppose that you have a list of headlines at the top of your HTML document that points to a more complete article at the bottom of your document. This is easy in HTML. Remember that anchor tags come with three attributes: href (which has already been discussed), and name and id (which always appear together).



Anchor A named point on a Web page. The same tag is used to create hyperlinks and anchors.



Caution In the new XHTML and XML standards (which will eventually replace HTML), the W3C is calling for the use of a new attribute for the <a> tag (called id) to replace the name attribute. The smart thing to do (to make sure that you comply with the new standard when it is released) is to use both attributes in your documents. To avoid problems, use the same value for both attributes, for example, if name="dogs" then id="dogs" as well.

The <a> tag also enables you to name an anchor (or bookmark) in your document with the name and id attributes. HTML then enables you to use the anchor tag to point directly to that bookmark. Figure 4.1 demonstrates how the example in the previous paragraph might look in HTML. Figure 4.2 shows that same document in the browser.

```

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Anchor Tags</title>
</head>
<body>
<h1>Click a Headline to Read the Full Text</h1>
<p><em><a href="#red">Red, white, and Blue</a></em></p>
<p><em><a href="#green">Green: All the Rage</a></em></p>
<h1>Read the Full Article</h1>
<h2><a name="red" id="red">Red, white, and Blue</a></h2>
<p>Red, white, and blue are making a comeback int he world of decorating. Recently,
  many designers have realized that their clients enjoy the feeling of nostalgia
  and patriotism those color inspire.</p>
<h2><a name="green" id="green">Green: All the Rage</a></h2>
<p>jail cells everywhere are seeing green. Once jail officials noticed the calming
  effect the color green had on inmates, they started painting everything green--
  from the walls of the cells to the walls of the cafeteria.</p>
</body>
</html>

```

FIGURE 4.1 Notice how the href attribute points to the location named by the name and id attributes.



Caution The `<a href>` tag includes the same URL format you've seen before, but also includes the # symbol to separate the filename from the named anchor.

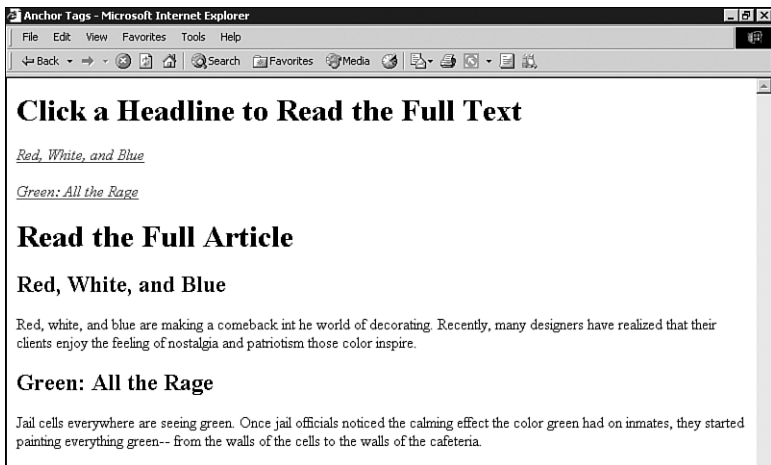


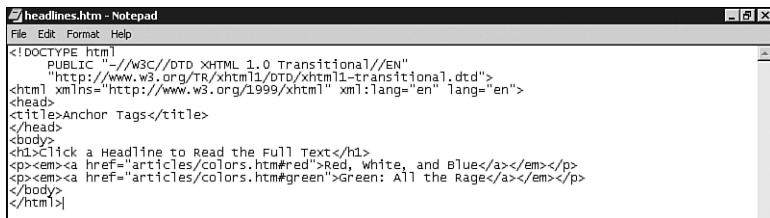
FIGURE 4.2 The `<a>` tag with the href attribute is highlighted, but the `<a>` tag with the name and id attributes is not.



Tip When naming anchors, remember to keep the names short and not to use spaces. These aren't HTML requirements, but following these guidelines certainly makes linking easier. Look at the example in Figure 4.1 again. The named anchor for the Red, White, and Blue article is the abbreviated red.

Linking to an Anchor in Another Page

Creating a hyperlink to an anchor in another page requires only one more element: the URL. As you learned before, you can link to an anchor in a file on your own Web site, as shown in Figure 4.3, or you can link to a known anchor in a file on another Web site. The keyword in that sentence is *known*. You can't link to a specific spot on a file unless that spot is already recognized by the Web browser as a named anchor.



```
<!DOCTYPE html>
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Anchor Tags</title>
</head>
<body>
<h1>Click a Headline to Read the Full Text</h1>
<p><em><a href="articles/colors.htm#red">Red, White, and Blue</a></em></p>
<p><em><a href="articles/colors.htm#green">Green: All the Rage</a></em></p>
</body>
</html>
```

FIGURE 4.3 Notice that each href attribute includes a folder name (articles), a filename (colors.htm), and the specific anchor name (red).



Tip You can use style sheets to add visual interest to your hyperlinks. Lesson 5, "Adding Your Own Style," will show you how.

Table 4.1 lists the HTML tags that were discussed in this lesson.

TABLE 4.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code></code>	<code></code>	Surrounds text that links to another location.
<code></code>	<code></code>	Surrounds text that is linked to.
<code></code>	<code></code>	Same as <code><a name></code> , but might soon replace it. Use them together as: <code></code> .

In this lesson, you've learned:

- Anchor tags come with three attributes: `href` (which links *to* someplace), and `name` and `id` (which link *from* someplace).
- You can copy the URL of any Web page from your browser and paste it between the quotes of the `href` attribute in your `<a>` tag.
- The same `href` attribute links to email addresses, to other pages of your own Web site, or even to other files on your own computer.

LESSON 5

Adding Your Own Style



In this lesson, you will learn how to create style sheets, apply them to your HTML pages, and wow your audience with your creativity.

Style Sheets

As you've already learned, HTML was written as a markup language for defining the structure of a document (paragraphs, headings, tables, and so on). Although it was never intended to become a desktop publishing tool, it does include some basic formatting attributes, such as `bgcolor`, `font-size`, and `align`. In 1996, the W3C first recommended the idea of Cascading Style Sheets (CSS) to format HTML documents. The recommendation, which was updated in mid-1998, enables Web developers to separate the structure and format of their documents.



Style Sheet A set of rules that determine how the styles are applied to the HTML tags in your documents.

The CSS recommendation describes the following three types of style sheets:

- **Embedded** The style properties are included (within the `<style>` tags) at the top of the HTML document. A style assigned to a particular tag applies to all those tags in this type of document. In this book, you'll see embedded style sheets most often.

- **Inline** The style properties are included throughout the HTML page. Each HTML tag receives its own style attributes as they occur in the page.
- **Linked** The style properties are stored in a separate file. That file can be linked to any HTML document using a `<link>` tag placed within the `<head>` tags.

In the following sections, you'll learn how to construct these style sheets and how to apply them to your documents.



Tip Even without all the formatting benefits that style sheets provide, Web developers can rejoice in knowing that using style sheets will no doubt be the biggest timesaver they've ever encountered. Because you can apply style sheets to as many HTML documents as you like, making changes takes a matter of minutes rather than days.

Before the advent of style sheets, if you wanted to change the appearance of a particular tag in your Web site, you would have to open each document, find the tag you wanted to change, make the change, save the document, and continue on to the next document. With style sheets, you can change the tag in a single style sheet document and have the changes take effect immediately in all the pages linked to it.

Defining the Rules

Style sheet rules are made up of selectors (the HTML tags that receive the style) and declarations (the style sheet properties and their values). In the following example, the selector is the body tag and the declaration is made up of the style property (background) and its value (black). This example sets the background color for the entire document to black.

```
body {background:black}
```

You can see that, in a style sheet, the HTML tag is not surrounded by brackets as it would be in the HTML document, and the declaration is surrounded by curly braces. Declarations can contain more than one property. The following example also sets the text color for this page to white. Notice that the two properties are separated by a semicolon.

```
body {background:black; color:white}
```

You can format this style rule in a number of ways to make it easier to read. For example, the following rule produces exactly the same results as the preceding style:

```
body {background:black;
      color:white}
```

So does this:

```
body {
    background:black;
    color:white
}
```

If you want to apply the same rules to several HTML tags, you could group those rules together, as in the following example:

```
body, td, h1 {
    background:black;
    color:white
}
```

Add a Little **class**

As the old saying goes, rules are made to be broken. What if you don't want every single h1 heading in your document to be white on a black background? Maybe you want every other h1 heading to be yellow on a white background. Let me introduce you to the **class** attribute. You can apply this attribute to almost every HTML tag, and it's almost like creating your own tags.

Figure 5.1 shows a fairly standard HTML page that uses an aqua table at the top of the page to hold the navigation links, and places other tabular content in yellow tables throughout the document. You can see the HTML document for that page in Figure 5.2.

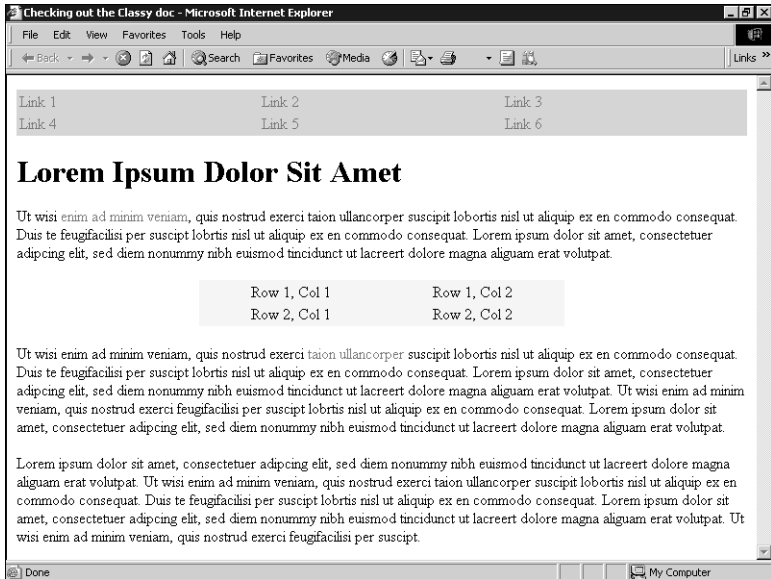


FIGURE 5.1 An HTML page that formats two tables differently.



FIGURE 5.2 The HTML document for the page in Figure 5.1. Notice the class attribute in each <table> tag.

Take a closer look at the style properties in Figure 5.2. This document defines two table styles within the `<style>` tags. The HTML tag name `table` is followed by a period (.) and the class names (`nav` and `rest`).

```
table.nav {background:aqua}
table.rest {background:yellow;
            text-align:center;
            color:black}
```

When the table is referenced in the body of the document, you must apply the `class` attribute to tell the browser which style properties should be applied. The HTML markup for each table in this example appears in the following HTML code. You can see that the `class` name appears within quotations just like the other HTML attributes (and as with the `width` attribute shown here).

```
<table class="nav" width="100%">
<table class="rest" width="50%">
```

Applying Styles

Before moving on, we'll quickly cover how to apply style properties to your documents. Remember, you have three methods to add style sheets: embedded, linked, and inline. We'll discuss each one in turn.



Tip In designing your Web site, use linked style sheets to describe your most frequently used styles (the ones that will be formatted in the same fashion for all of the pages in your Web site), such as the heading tags and link tags. Use embedded style sheets to describe the formatting of tags that will remain the same within a single document, or set of documents, such as special table settings or page margins. Use the inline style sheets to describe the formatting of tags that vary from the site-wide formatting applied with the other style sheets, such as for a special callout or sidebar.

Embedded Styles

All the styles are defined at the top of the HTML document within the `<head>` tags because they contain information about the entire document. The styles defined here apply only to the one document in which they appear. If you plan to use these same styles in another document, you need to add them there as well.

```
<head>
<style type="text/css">
table.nav {background:aqua}
table.rest {background:yellow;
            text-align:center;
            color:black}
a:link {color:red;
        text-decoration:none}
</style>
</head>
```



Note The `<style>` tag almost always includes the `type="text/css"` attribute, so you should get used to adding it.

Linked Styles

Linked style sheets hold all the style properties in a separate file. You then link the file into each HTML document where you want those style properties to appear.

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

With this method, I've created a separate file called `mystyles.css` (for cascading style sheet) that contains all my style properties. You can see that the same `type="text/css"` attribute shows up here. Following are the entire contents of the `mystyles.css` file. These are the same styles that showed up in the preceding embedded styles example, but now they appear in a separate text file.

```
table.nav {background:aqua}
table.rest {background:yellow;
            text-align:center;
            color:black}
a:link {color:red;
        text-decoration:none}
```



Tip Well-designed Web sites (with more than one page) contain repeated page elements and styles. The linked style sheet is most appropriate for this type of Web authoring. You'll learn more about designing effective Web sites in Lesson 13, "Designing with HTML."

Inline Styles

With inline styles, the style properties are added to the HTML tag as the tag is entered. This means that if I want the same style to appear on all the <h1> tags in my document, I would have to type those styles in all the <h1> tags. Look at the following example. I am still using the same style properties, as in the previous examples, but now you can see how the two tables would be created using inline styles.

```
<table style="background:aqua" width="100%">

<table style="background:yellow; text-align:center;
            color:black" width="100%">
```

Using inline styles, the <style> tag becomes the style attribute. Multiple style properties are still separated by semicolons, but the entire group of properties for each tag is grouped within each HTML tag. This type of style sheet is fine for documents in which you need to apply styles to only one or two elements, but you wouldn't want to do all that work when you have a lot of styles to add.

Cascading Precedence

You've got one more thing to learn before moving on. These three styles are not treated equally by the browsers, nor are they supposed to be.

Web browsers give precedence to the style that appears closest to the tag. So, inline styles (which appear as attributes within the tag itself) are most important. Embedded styles (which appear at the top of the HTML file) are applied next, and linked styles (which appear in another file altogether) are applied last.

Imagine that you have created an embedded style for the `<h1>` tag, but want to change that style for one occurrence of the `<h1>` tag in that document. You would create an inline style for that new `<h1>` tag. The browsers recognize that fact and change the style for that tag to reflect the inline style.



Caution Style sheet precedence is supposed to place more importance on embedded styles than on linked style sheets. In actual practice, however, you'll find that both Internet Explorer and Netscape treat linked sheets as more important than embedded sheets (but they do treat inline styles as more important than either of the other two). You'll find that you have better luck if you use either linked or embedded styles, but not both.

Formatting Text with Styles

Text is the most important element of any Web page. Without text, there is nothing on the page to help people decide whether it's worth coming back.

Text on an HTML page is structured by the `<body>`, `<p>`, `<td>`, `<tr>`, `<th>`, `<h1>` `<h6>`, and `` tags (among others). You can add your own style preferences to each of these tags using the style properties shown in Table 5.1.



Note Unless you (or the people viewing your pages) have adjusted the browser's default settings, normal HTML body text appears in 12 point Times New Roman font on most computer systems.

In the following example, we've added some embedded style elements that set the font, font size, and font color for the body text of the basic HTML document we created in Lesson 2, "Creating Your First Page." In Figure 5.3, you can see how those styles change the appearance of the document in the browser.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<title>My First Web Page</title>
<style type="text/css">
body {font-family:"Arial";
      font-size:"12pt";
      color:red}
</style>
</head>
<body>
<p>This is my <b><i>first</i></b> Web page.</p>
</body>
</html>
```



FIGURE 5.3 The browser applies the style attributes to the text in the `<body>` tags.

Table 5.1 lists the many style properties that you can use to format your text.

TABLE 5.1 Style Properties for Text

Property	Description of Use and Values
background	Sets the background color for the text.
color	Sets the text color for the text.

continues

TABLE 5.1 Continued

Property	Description of Use and Values
font-family	Sets the font for the text.
font-size	Can be a point size, a percentage of the size of another tag, or xx-small to xx-large.
font-style	normal (which is assumed) or italic.
font-weight	extra-light to extra-bold.
text-align	left, right, center, or justify (full).
text-indent	Can be a fixed length or a percentage.
text-decoration	underline, overline, strikethrough, and none.

Microsoft maintains a brief tutorial for style sheets on its typography site (<http://www.microsoft.com/typography/default.mspx>). The tutorial teaches Web page authors how style sheets can enhance their documents. The <style> tag for one of those examples is shown in the following code. This is impressive because of the many different styles and classes defined in this document. You can see that you are only limited by your own imagination. You can see the page this style code created in Figure 5.4.

```
<style type="text/css">
body {background: coral}
.copy {color: Black;
      font-size: 11px;
      line-height: 14px;
      font-family: Verdana, Arial, Helvetica, sans-serif}
a:link {text-decoration: none;
      font-size: 20px;
      color: black;
      font-family: Impact, Arial Black, Arial,
                  Helvetica, sans-serif}
.star {color: white;
      font-size: 350px;
      font-family: Arial, Arial, helvetica, sans-serif}
.subhead {color: black;
      font-size: 28px;
      margin-top: 12px;
      margin-left: 20px;
```




FIGURE 5.4 The preceding style code produced this page, found at <http://www.microsoft.com/typography/css/gallery/slide3.htm>.



Caution None of the most popular Web browsers react the same way to all the style sheet properties. Your best bet is to remember to test everything before you publish it. Webmaster Stop maintains a table of style sheet properties mapped to the most popular browsers. Check out this table (<http://www.webmasterstop.com/118.html>) to find out whether the style sheet properties you plan to use are supported by specific browsers.

Link Styles

You have probably seen those bright blue underlined hyperlinks on the Web. Style sheets have the following selectors to help you change the look of them:

- `a:link` Sets the styles for unvisited links.
- `a:visited` Sets the styles for visited links.

- `a:active` Sets the styles for the link while it is linking.
- `a:hover` Sets the style for the link while your mouse is hovering.

Table 5.2 shows some of the style properties you can assign to your links.

TABLE 5.2 Style Properties for the Anchor Styles

Property	Description of Use and Values
<code>background-color</code>	Sets the background color for the link.
<code>color</code>	Sets the text color for the link.
<code>font-family</code>	Sets the font for the text of the link.
<code>text-decoration</code>	<code>underline</code> , <code>overline</code> , <code>strikethrough</code> , and <code>none</code> .



Tip One of the most popular style sheet effects on the Web right now is to remove the underlining on hyperlinks. To do this on your pages, just add the `text-decoration:none` declaration to the `a` styles, as shown in the following example:

```
a:link {color:yellow;
        text-decoration:none}
```

If you like the look of the underlined hyperlink, you're in luck. You don't have to specify anything at all. Underlining is assumed for all `a` styles.

Color Styles

As you can see in Table 5.3, you can apply color to your HTML tags in two different ways: with `color` or with `background`.



Tip Check out <http://wv1.internet.com/Graphics/Colour/> for a quick tune-up of Web color selections.

TABLE 5.3 Style Properties for Color

Property	Description of Use and Values
color	Sets the color of the text.
background	Sets the background of the page or text.



Caution Don't forget to test your pages before you publish them. Not all colors work together. If you've specified a black background color and a black text color, you've got a problem because no one will be able to see your text.

Adding Lines

A horizontal line, or horizontal rule as it is named in HTML, is one of the easiest tags to use. You can insert the `<hr />` tag anywhere in your document to insert a horizontal line that extends across the space available. Take a look at the following sample HTML. It shows three `<hr>` tags: two used as a section break between text and the other used inside a table cell. Figure 5.5 shows how they appear in the browser.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title>Horizontal Lines</title>
<style type="text/css">
td {text-align:center}
</style>
</head>
<body>
<p>This is a horizontal line.</p>
<hr />
<p>This is another horizontal line.</p>
<hr />

<table width="50%" rules=cols>
```

```

<tr>
  <td>This is also a<hr />horizontal line.</td>
  <td>There is <br />no line on this<br />side
    of the table.</td>
</tr>
</table>
</body>
</html>

```

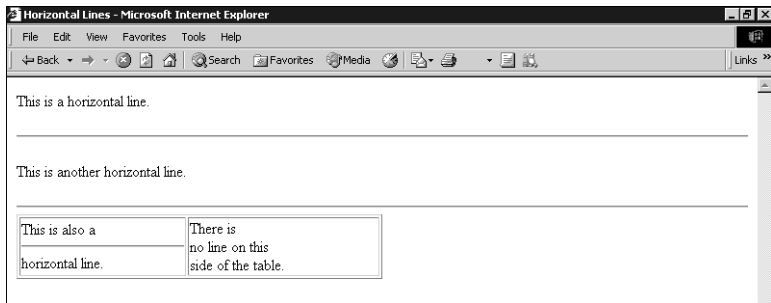


FIGURE 5.5 The `<hr />` tag inserts a horizontal line that stretches across the available horizontal space.

Adding Style to Horizontal Lines

As with other HTML tags, you can use style sheet properties to design your own horizontal rules. You can set the height, width, and color of the line to match the design of your Web page. The following HTML sample shows two different styles attached to the `<hr />` tag. If I use the `hr.red` style, I see a red line that takes up 50% of the horizontal space. If I use the `hr.purple` style, I see a purple line that is 4 pixels high and takes up 75% of the horizontal space.

```

<style type="text/css">
hr.red {color:red;
        width:50%}
hr.purple {color:purple;
           height:4;
           width:75%}
</style>

```

I've used both of those styles in the following sample HTML. Figure 5.6 shows you how those examples look in the browser.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title>Horizontal Lines</title>
<style type="text/css">
td {text-align:center}
hr.red {color:red;
        width:50%}
hr.purple {color:purple;
           height:4;
           width:75%}
</style>
</head>
<body>
<p>This is a plain horizontal line.</p>
<hr />

<p>This is a purple horizontal line.</p>
<hr class="purple" />

<table width="50%" rules=cols>
  <tr>
    <td>This is a red <hr class="red" />horizontal line.</td>
    <td>There is <br />no line on this<br />
      side of the table.</td>
  </tr>
</table>
</body>
</html>
```

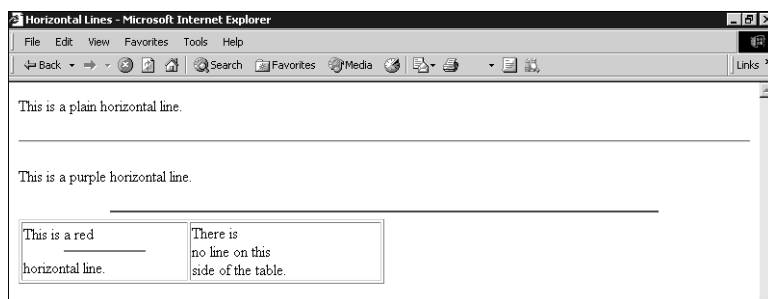


FIGURE 5.6 Applying styles to the `<hr />` tag changes the appearance of the horizontal line.

Margin Styles

Style sheets give you another important advantage: You can specify the margins of almost any HTML element. The margins can be defined in pt, in, cm, or px sizes.

```
body {margin-left: 100px;
      margin-right: 100px;
      margin-top: 50px}
```

You can set the `margin-left`, `margin-right`, and `margin-top` properties individually or combine them into one property called `margin` that applies the sizes to the top, right, and left margins.

```
body {margin: 100px 100px 50px}
```

The sample CSS document from Microsoft's CSS Gallery (which you looked at earlier) also specifies margins for the text elements. Try it on your documents.

```
<style type="text/css">
body {background: coral }
.subhead { color: black;
    font-size: 28px;
    margin-top: 12px;
    margin-left: 20px;
    line-height: 32px;
    font-family: Impact, Arial Black, Arial,
                Helvetica, sans-serif}
</style>
```



Tip Check out the following style sheet references for more help:

- http://webdeveloper.com/html/html_css_1.html
Web Developer's CSS tutorial
- www.w3.org/TR/REC-CSS2/propidx.html W3C's list of CSS properties
- <http://www.microsoft.com/typography/default.msp> Microsoft's CSS tutorial

Table 5.4 lists the HTML tags that were discussed in this lesson.

TABLE 5.4 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code><hr /></code>		Creates a horizontal line.
<code><style></code>	<code></style></code>	Surrounds style sheet properties, or references to external style sheets. The standard open tag should be <code><style type= "text/css"></code> .

In this lesson, you've learned:

- The CSS recommendation describes three types of style sheets: embedded, inline, and linked.
- Three different style sheets exist in HTML: embedded, inline, and linked.
- If multiple style sheets are applied to your HTML document, the browser applies the styles of the inline style sheet first, and then the linked style sheets, and then embedded style sheets.
- The `<hr />` tag adds a horizontal line to your HTML document. Use style sheet properties to adjust the color, width, and height.
- Remove the underlining on your hyperlinks by adding the `text-decoration:none` declaration to your a style tags.

LESSON 6

Creating Lists



In this lesson, you will learn to use HTML to organize your text into lists.

Types of Lists

One way to organize the text in your Web pages is with lists. In addition to the obvious benefit of being able to *list* items on a page, they also provide a design benefit by enabling you to break up long pages of ordinary paragraphs. HTML recognizes the following list types and has tags that correspond to each:

- Bulleted (unordered) lists
- Numbered/lettered (ordered) lists
- Definition lists



Tip You should use ordered lists when the items in the list must be followed in a specific order, and use unordered lists when they do not. You generally use definition lists for terms and their definitions, but they can have other uses as well.

Bulleted (Unordered) Lists

A bullet (usually a solid circle) appears in front of each item in an unordered list. HTML automatically creates the bullet when you use the unordered list tag (``) together with the list item tag (``). Although the following sample HTML shows each list item as a single line of text, your list items can be as long as you want:

```
<ul><li>first item in the list</li>
<li>second item in the list</li>
<li>third item in the list</li></ul>
```


Figure 6.1 shows how the Web browser displays an ordered and an unordered list. The figure includes list examples from many of the following sections. When your list items are longer than a single line of text, the Web browser indents the second line (and any following lines) so that the text lines up.

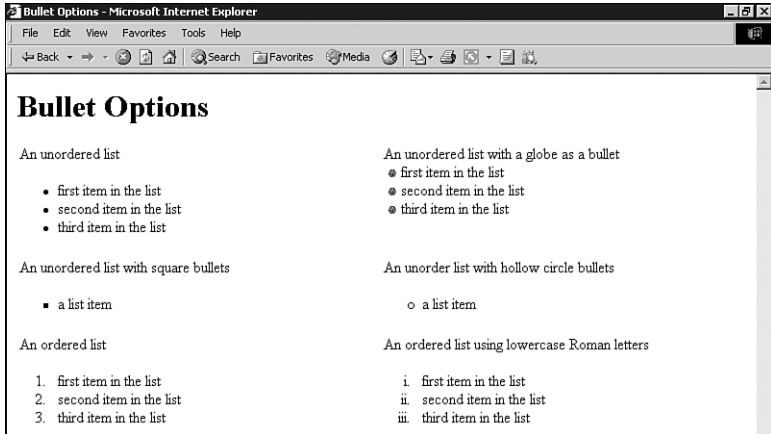


FIGURE 6.1 Ordered and unordered lists shown in the Web browser.

Formatting Bulleted Lists

HTML automatically adds a solid circle in front of each list item as a bullet, but you have two other choices. Using style sheet tags (which you learned about in Lesson 5, "Adding Your Own Style"), you can select one of two other bullet types: a square or a hollow circle. You can see how your HTML document would look if you chose to use a square bullet instead of the standard solid circle. Figure 6.1 shows how the Web browser displays this bullet type.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">

<head>
```

```
<title>Bullet Options</title>
<style type="text/css">
ul.square {list-style-type:square}
ul.image {list-style-
image:url("http://www.xeroxblankmedia.co.uk/graphics/globe.gif")}
</style>
</head>
<body>
<ul class="square">
<li>a list item</li>
<li>another list item</li>
</ul>

<ul class="image">
<li>a list item</li>
<li>another list item</li>
</ul>
</body>
</html>
```

You'll notice that this sample HTML also includes a style (`list-style-image`). This style enables you to replace the plain HTML bullets with your own image. In this example, I replaced the bullets with an image of a globe. Try changing the URL for one of your own images. You can see the globe image I chose in Figure 6.1.

Numbered (Ordered) Lists

If the items in your list should follow a specific order, as in recipes or instructions, you want to use the ordered list tag (``). With this tag, HTML automatically numbers or letters your items for you. Here's an example:

```
<ol><li>first item in the list</li>
<li>second item in the list</li>
<li>third item in the list</li></ol>
```

Notice how similar the two list samples are. Both the `` and `` tags call for the individual list items to be identified with the `` tag. Like the `` tag, HTML has an automatic style for the list items within the `` tag. HTML automatically numbers the items with the familiar Arabic

numerals (1, 2, 3, and so on). What's more, it automatically rennumbers the list items if you decide to add or delete items later. Once again, Figure 6.1 has an example of this type of list.

Formatting Numbered Lists

You can use style sheets for formatting ordered lists. In addition to the standard Arabic numerals, there are four other styles that can be applied to your ordered list. Table 6.1 describes each of those types, and the following sample HTML shows how you can use style sheets to create a list ordered by lowercase roman numerals. Figure 6.1 shows an example of such a list in the Web browser.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>
<title>Bullet Options</title>
<style type="text/css">
ol.lwroman {list-style-type:lower-roman}
</style>
</head>
<body>
<ol class="lwroman">
<li>a list item</li>
<li>another list item</li>
</ol>
</body>
</html>
```

TABLE 6.1 List Style Types

Sample	Style Syntax	Definition
a, b, c	lower-alpha	Lowercase letters
A, B, C	upper-alpha	Uppercase letters
i, ii, iii	lower-roman	Small roman numerals
I, II, III	upper-roman	Large roman numerals

Setting a Start Value

There might be times when you'd like to start an ordered list with a number other than one. Many times when writing instructions, you need to interrupt a numbered list with some other material (such as text or examples), and then continue with the numbered list. To do this in HTML, close the first list, add the additional materials that you need, and then start a new list, using the list item's value attribute to set the beginning number for the new list. Figure 6.2 demonstrates how the following code is interpreted by the browser.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>
<title>Ordered Lists</title>
<style type="text/css">
</style>
</head>
<body>
<p>It's Payday!</p>
<ol>
<li>Turn in your timecard.</li>
<li>Receive your paycheck.</li>
<li>Endorse your paycheck.</li>
</ol>
<p>Congratulations! You're almost there.</p>
<ol>
<li value="4">Put the check in the bank.</li>
</ol>
</body>
</html>
```



Caution The value attribute requires that you use Arabic numbering to specify the start value, even if you've chosen roman numerals or letters for your list type.

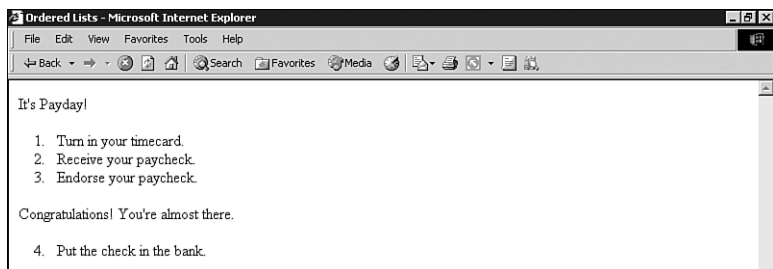


FIGURE 6.2 The Web browser shows an ordered list that was interrupted and started again using the value attribute.

Definition Lists

If you need it, HTML has one more type of list available to you: the definition list, which uses the `<dl>` tag. Rather than using the usual `` tag to specify the items in the list, this type of list uses the `<dt>` tag (for definition terms) and the `<dd>` tag for their definitions. Following is an example of the HTML for a definition list, and Figure 6.3 shows how the Web browser formats a definition list.

```
<dl><dt>The Definition Term</dt>  
<dd>Is defined below the term.</dd></dl>
```

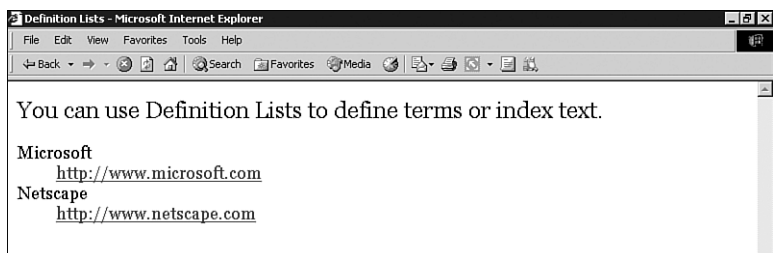


FIGURE 6.3 A definition list displayed in the browser.

Table 6.2 lists the HTML tags that were discussed in this lesson.

TABLE 6.2 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<d1>	</d1>	Definition list.
		List item. Used with and tags.
<dt>	</dt>	Definition term. The list item of a <d1>.
<dd>	</dd>	Definition data. Describes definition terms.
		Ordered, or numbered/lettered, list.
		Unordered, or bulleted, list.

In this lesson, you've learned:

- HTML recognizes three different list types: unordered (bulleted), ordered (numbered), and definition.
- Rather than the default bullet style (a solid circle), style sheets enable you to select three other bullet types: a square, a hollow circle, or an image of your own.
- The value attribute of the tag sets the beginning number for your list.



LESSON 7

Creating Tables

In this lesson, you will learn how to build tables using HTML, and how to control the layout and appearance of a Web page using tables.

Simple Tables

Traditionally, *tables* have been used for displaying tabular data (such as numbers) in rows and columns. The flexibility of HTML, however, enables Web developers to create tables that display more than just numbers. In fact, as important as the capability to display tabular data is, even more important to Web designers is the capability to control the layout of other document elements (such as text and images).



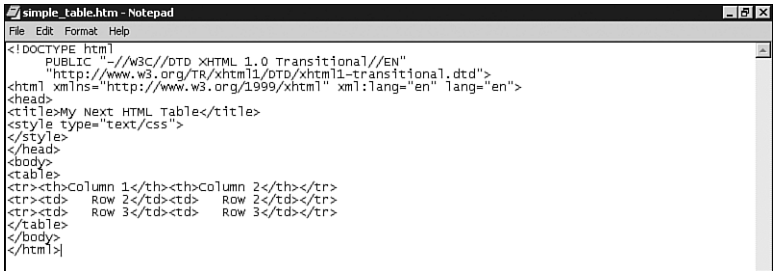
Table An arrangement of horizontal rows and vertical columns. The intersection of a row and a column is called a *cell*.

Caution Although HTML tables look similar to your favorite spreadsheet, HTML tables don't perform mathematical functions.

HTML tables are not difficult to create, but they do require some organization. All HTML tables begin with the `<table>` tag and end with the `</table>` tag. In between those tags are three other tags to be aware of, as follows:

- `<tr>` defines a horizontal row.
- `<td>` defines a data cell within that row.
- `<th>` specifies a data cell as a table heading. In newer browsers, a table heading cell is formatted as centered and bold.

Remember that Web browsers ignore any spaces, tabs, and blank lines that you include in your HTML document. So, feel free to use spacing to help you keep track of the table tags. Figure 7.1 shows enough blank spaces between the tags so that you can see the rows and columns lining up. It makes it easier to ensure that you don't forget any tags. Figure 7.2 shows how that table looks in a browser.



```
<!DOCTYPE html>
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<title>My Next HTML Table</title>
<style type="text/css">
</style>
</head>
<body>
<table>
<tr><th>Column 1</th><th>Column 2</th></tr>
<tr><td>  Row 2</td><td>  Row 2</td></tr>
<tr><td>  Row 3</td><td>  Row 3</td></tr>
</table>
</body>
</html>
```

FIGURE 7.1 A simple two-column, three-row HTML table.

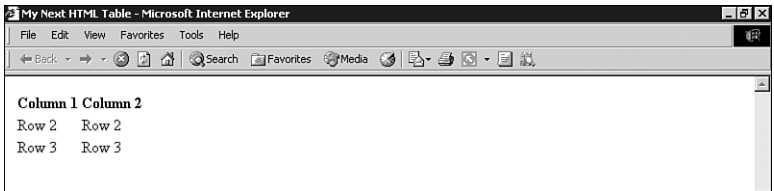


FIGURE 7.2 That same HTML table as it appears in the browser.

Formatting Tables

Now you can add some pizzazz to your simple table. In Table 7.1, you see some of the different style attributes you can apply to HTML tables. Figure 7.3 shows how you can use these attributes to create an HTML table with a little more character. Figure 7.4 shows the way the table appears in a browser.



Tip The World Wide Web Consortium's Web site (www.w3.org/TR/REC-html40/struct/tables.html) has detailed descriptions of all the attributes available for tables, as well as examples of how you can use them.

TABLE 7.1 Table Style Attributes

Attribute	Default	Use With	Values
align	left	All	Horizontal alignment of cell contents: left, right, center, and char (which aligns around a specific character, usually a decimal or comma).
bgcolor		All	Background color.
border	0	<table>	Width of the border (in <i>pixels</i>).
cellpadding	0	<td>, <th>	Space between border and content (in <i>pixels</i>).
cellspacing	0	<td>, <th>	Space between cells (in <i>pixels</i>).
colspan	1	<td>, <th>	Number of columns that a cell should span (merge).
rowspan	1	<td>, <th>	Number of rows that a cell should span (merge).
rules	none	<table>	Where the lines (rules) appear between cells: rows, cols, or all.
valign	center	<td>, <tr>, <th>	Vertical alignment of cell contents: top, bottom, or baseline.
width	to fit	All	Width of table or cells (in <i>pixels</i> or as a percentage of the page).



Pixel A pixel is the size of a single dot of color on your monitor. The monitor's display resolution affects the size of a pixel. A display resolution of 800×600 means that your monitor displays 800 pixels in width by 600 pixels in height. The pixel size on a monitor that displays at a resolution of 1024×800 would be much smaller than one on a monitor with a resolution of 800×600.

A table with a width attribute fixed at 800 pixels fills a screen that is set to a resolution of 800×600, but only fills a portion of a screen that is set to 1024×800.

```

table_attributes.htm - Notepad
File Edit Format Help
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>My Next HTML Table</title>
<style type="text/css">
th {background-color:lightblue;
  color:red;
  font-family:Arial}
td {color:blue;
  font-family:Times New Roman}
</style>
</head>
<body>
<table width="50%" border="1" rules="all">
<tr><th>Column 1</th><th>Column 2</th></tr>
<tr><td align="center"> Row 2</td><td align="center"> Row 2</td></tr>
<tr><td align="center"> Row 3</td><td align="center"> Row 3</td></tr>
</table>
</body>
</html>
  
```

FIGURE 7.3 Table attributes in HTML.

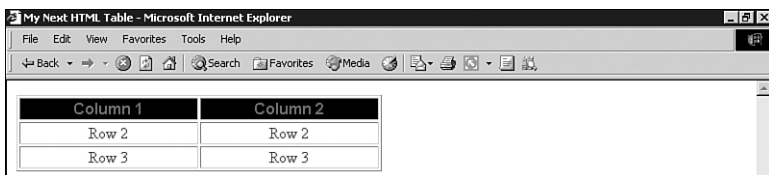


FIGURE 7.4 That same HTML table as it appears in the browser.

Advanced Tables

HTML contains two more attributes that you should be aware of when formatting tables. The `colspan` (which causes a cell to span two or more columns) and `rowspan` (which causes a cell to span two or more rows) attributes are invaluable when creating complex tables, although, as you can tell from the HTML in Figure 7.5, using them makes it harder to keep your HTML document organized. Figure 7.6 shows how the table looks in a browser.

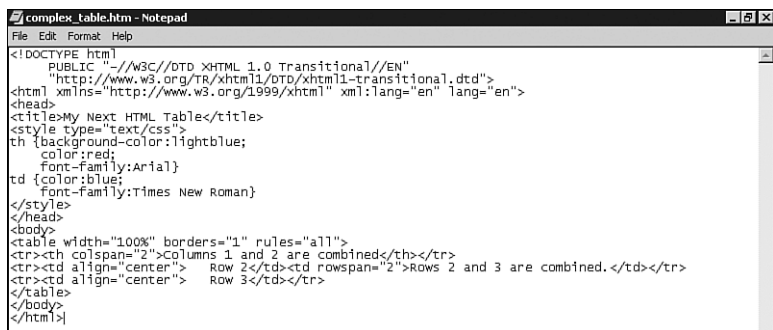


FIGURE 7.5 Using the `colspan` and `rowspan` attributes to create complex tables.



FIGURE 7.6 That same HTML table as it appears in the browser.

Using Tables for Layout

Look at the source code for some of your favorite Web pages and I bet that you'll find they were created using tables. Following are some of my favorite Web pages that use tables to control the page layout:

- www.ibm.com/us/ The columns of search categories are created with tables.
- www.cnn.com/ This site is essentially a three column table.
- www.microsoft.com/office/editions/prodinfo/default.mspx Microsoft, too, uses tables to design the layout of its Web site.
- www.idolonfox.com/ The *American Idol* Web site demonstrates another creative use of tables in layout.



Tip Even if you don't plan to place a border around the cells in your table, it's much easier to see how your HTML commands are interpreted by your Web browser when you have the borders turned on (`<table border="1">`). After you are satisfied that the table is formatted correctly and your content is where you want it to be, you can remove the border attribute, leaving just the `<table>` tag.

Table 7.2 lists the HTML tags that were discussed in this lesson.

TABLE 7.2 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code><table></code>	<code></table></code>	Identifies the beginning and ending of a table. All table content must be contained within this tag.
<code><td></code>	<code></td></code>	Table data cell. Similar to a column.
<code><th></code>	<code></th></code>	Table heading.
<code><tr></code>	<code></tr></code>	Table row. Surrounds table cells (<code><td></code>) and headings (<code><th></code>).

In this lesson, you've learned:

- Tables can control the layout of HTML document elements (such as text, navigation, and images).
- Extra spaces in your HTML documents help you keep track of the table tags. Web browsers ignore any extra spaces.
- The `colspan` and `rowspan` attributes merge cells so that you can create complex tables.

LESSON 8

Using Graphics



In this lesson, you'll learn how to add pizzazz to your Web pages with graphic images.

Adding Images

If the Web were nothing but text, it would still be technologically impressive, but it wouldn't be nearly as much fun. Adding images to your pages is easy; adding images that make your Web pages look professional just takes a little know-how. Luckily, you'll learn that here—and it shouldn't take longer than 10 minutes.

The two most frequently used graphics file formats found on the Web are GIF and JPEG. The *Joint Photographic Experts Group (JPEG)* format is used primarily for realistic, photographic-quality images. The *Graphics Interface Format (GIF)* is used for almost everything else. One new file format is gaining popularity among designers and will soon be making its presence known: The *Portable Network Graphics* format (*PNG*) is expected to replace the GIF format someday. Don't rush out to replace all your graphics, however; not all browsers support it fully yet.



Tip Sound like a pro—learn how to pronounce the graphic formats you use. GIF is pronounced “jif” (like the peanut butter), JPEG is pronounced “jay-peg,” and PNG is pronounced “ping.”

Let's get down to business. You add all images by using a single HTML tag, the image source tag, ``. By now you probably recognize that this tag is actually an `` tag with an attribute

(src) and attribute value (*location*), but because all images require a src attribute, it's easier to refer to it as a single tag. You'll also notice that the image tag does not have a corresponding closing tag. It is a single tag and you'll need to remember to add the closing slash at the end: ``. The result of the following sample HTML appears in Figure 8.1.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
      transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<head>
<title>First Images</title>
<style type="text/css">
</style>
</head>
<body>
<p>This is an image in my first Web page.</p>

</body>
</html>
```

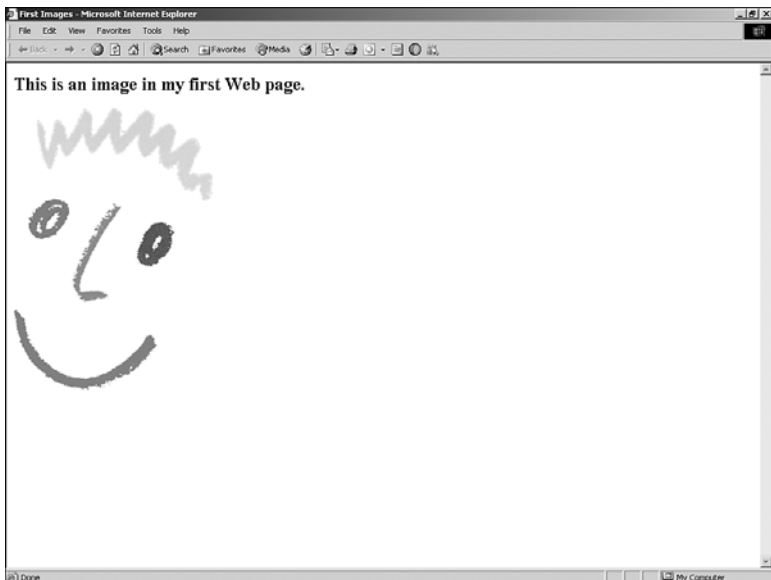


FIGURE 8.1 The `` tag inserts an image into your HTML document.



Caution Be aware that the World Wide Web Consortium, the standards-setting body for HTML, is considering replacing the `` tag with the more generic `<object>` tag. To add an image using the `<object>` tag, follow this format:

```
<object data="location" type="image/gif"> text  
describing the image... </object>
```

Adding Alternate Text

When browsing the Web, you might have noticed that many times when you move your mouse pointer over an image, you see a text pop-up that describes the image, or tells you something more about the area of the Web site that image represents. You can see an example of that type of text pop-up in Figure 8.2.

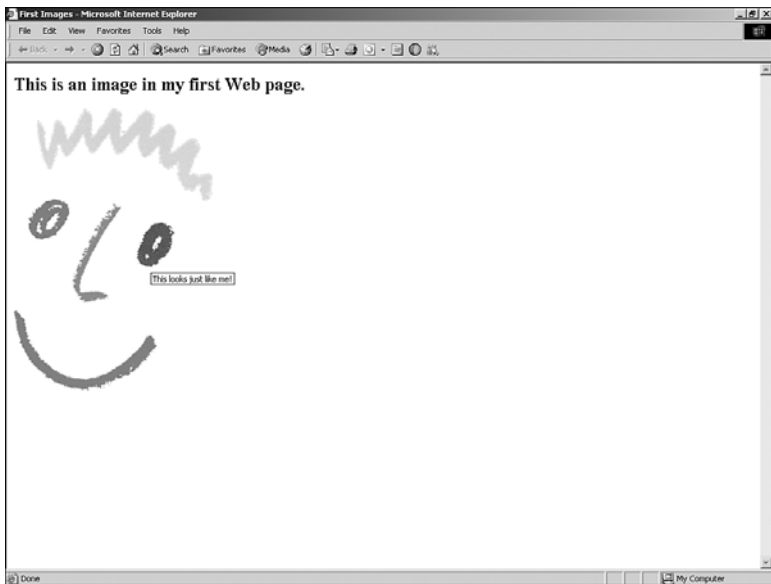


FIGURE 8.2 The `alt` attribute adds a text pop-up to your image.

The following HTML sample shows how the `alt` attribute is added into the `` tag. Like the `src` attribute, the `alt` attribute tells the browser more information about the image. And, like the `src` attribute, you should always use the `alt` attribute with the `` tag.

```

```



alt attribute Sets the *alternate text* for a graphic. It was named `alt` because it describes the text some people would see as an alternative to the image that others would see.

The `alt` attribute has another very important purpose. Many people with slower modem connections to the Web decide to customize their browser settings to ignore graphics because loading graphics into a Web browser can sometimes take a long time. Remember, too, that not all browsers enable you to view graphics. Some browsers, such as Lynx, have no graphics capabilities at all. The `alt` attribute ensures that people who can't view your graphics can still understand their context.



Caution Although you should use the `alt` attribute whenever you use the `` tag, make sure that you don't specify irrelevant text. For example, there is no point in specifying alternate text for a decorative image (such as a bullet or a line); instead, specify an empty value (`alt=" "`).

Without any other attributes, the browser displays the image at its original size and aligns the bottom of the graphic with the bottom of the text. You can adjust both those settings using style sheet tags.

Image Attributes

You can use other attributes of the `` tag to change the image size. Table 8.1 shows some of these attributes, and the following sections provide some examples for adding these attributes to your documents.

TABLE 8.1 Attributes Used with the `` Tag

Attribute	Values	Description of Use
height	Pixel or percent	Specifies the height of an image.
width	Pixel or percent	Specifies the width of an image.

Adjusting the Height and Width

You can adjust the size of your image using the height and width attributes. You can set these attributes to a fixed pixel size or a percentage of the page size. Look at the following sample HTML lines. The first line sets the happy face image from Figure 8.1 to a fixed pixel size of 60 pixels high and 60 pixels wide. The second line sets the same image to 6% of the page width and 10% of the page height. Figure 8.3 shows how both of these look in the browser.

```


```

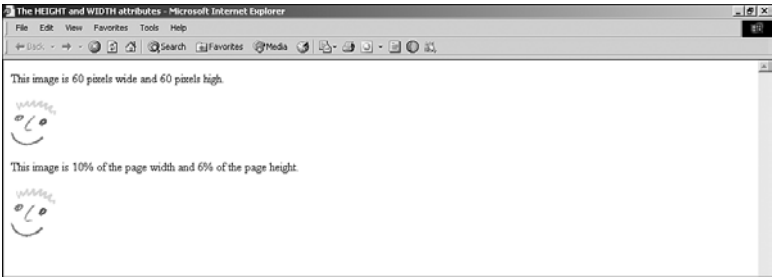


FIGURE 8.3 The height and width attributes control the size of the image.

The Web browser interprets pixels and percentages equally well when rendering an image. You need to remember, however, that your Web visitors may not use the same monitor display settings that you do. What does this mean to you? My monitor is set to 800 pixels wide. In the preceding HTML sample, I set the happy face image to 6% of the page width, or 48 pixels wide. If I viewed the same page on a monitor set to 1024 pixels wide, that same 6% of the page width would now equal 61 pixels, which is much wider than I wanted.

If you truly want the image to be a certain percentage of the page (as you might for a graphical line), then use percentages. Using percentages ensures that the image will take up the space you want it to. If you want the image to appear a specific size, use the pixel setting.



Caution Be sure to change both the height and width of your image if you plan to resize them. Adjusting only one of them will stretch the image out of proportion. An alternative is to resize the image in your image editor.



Tip Create the illusion of faster image loading. Regardless of whether you're resizing an image or not, you should always include the height and width attributes because they give the browser important information about how much space will be required to show the image on the page. This way, the browser can set that space aside and continue building other aspects of the page even while the image downloads. This gives the impression that the page loads faster since the viewer doesn't have to wait for the entire image to download before looking at other areas of the page.

Aligning Text and Images

You can use the `align` attribute of the `` tag to force an image to appear on the left or right of a section of text. You can see an example of this attribute in action in Figure 8.4.

```

```

You also can use the align attribute to vertically align an image with the text. The align attribute has three more values: top, bottom, and center, which are discussed in the following list. Figure 8.4 shows you a sample HTML document using the vertical alignment properties.

- Setting the align attribute to top aligns the top of the image with the top of any surrounding text.
- Setting the align attribute to bottom aligns the bottom of the image with the bottom of any surrounding text.
- Setting the align attribute to center aligns the center of the image with the center of any surrounding text.

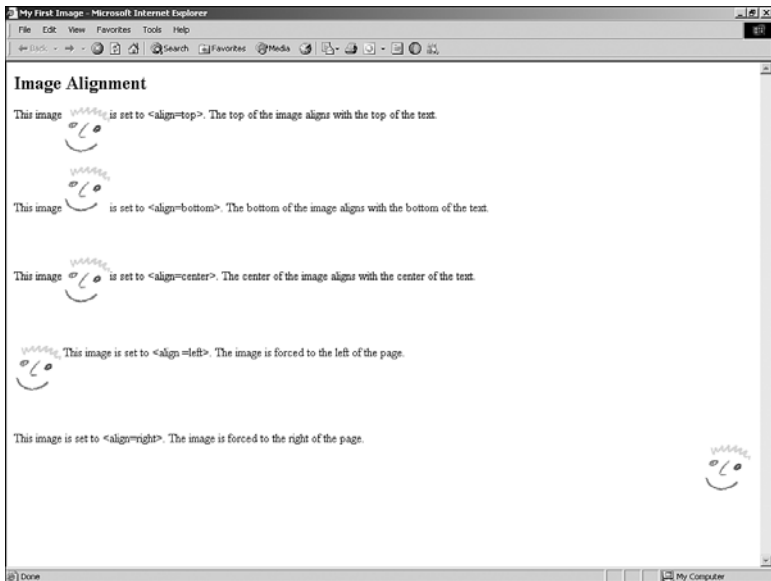


FIGURE 8.4 Notice how the align attribute forces the image to align with the text.



Caution Be sure to preview your HTML documents in the browser (or in several browsers) to make sure that you are happy with how they look before you publish them. Not all browsers treat these `align` attributes in the same way.

Using Images as Links

Images are good for more than just looks. You can use them to provide creative hyperlinks to other documents. HTML makes this easy because using an image as a link is exactly the same as using text. You are still using the anchor tag (the `<a>` tag you learned about in Lesson 4, “Linking Text and Documents”) to surround the item you want to act as the hyperlink to another document. When you link from an image, the anchor tags must surround the image tag. Following is an example of the HTML you would use:

```
<a href="DOC2.htm">  
  
</a>
```

When your visitors move their mouse pointers over the face image in this sample, they will see a pop-up that says, “This looks just like me!” When the visitors click on the image, they will open the `DOC2.htm` file referenced by the anchor tag.

Thumbnail Images

Another popular use of the hyperlinking capability of HTML is to link from one image to another. Why would you want to do that? Well, many times the image you want to display is so large that it takes longer to load into the browser than you think people would like to wait. If that’s so, you can create a smaller version of the file, called a *thumbnail*, that will load more quickly into the browser. The visitor simply clicks the thumbnail if he wants to open the larger file. Here’s how it’s done.

```
<a href="large_image.jpg">  
  
</a>
```

As you can see, clicking the thumbnail.jpg image will open another image (large_image.jpg). The alt attribute in this sample tells the visitor how to open the larger image.



Tip Many image editor programs provide tools to help you create thumbnail images of your large graphics. You can also use standalone products, such as Cerious Software's Thumbs Plus available for download at <ftp://ftp.cerious.com/pub/cerious/thmpls32.exe>.

Image Etiquette

Images are fun and colorful and easy to add to your HTML, but following are some etiquette rules to follow if you want your visitors to be happy with your site.

- The larger an image's file size, the longer it will take to load into the browser. Because most visitors to the World Wide Web still use a slow speed modem to connect from home, their time is precious. If you remember that and make sure to use small images whenever possible, you'll find that your visitors are happier.
- Not only is the file size of the individual image important, but also is the total file size of your HTML document. The more images you add—even small images—the larger your file size will become. Previewing your page in several browsers will help you determine how long your page will take to load in the browser. If you find the time too slow, so will your visitors.
- While the alt attribute is one of the most important attributes (because it should be used every time you use the `` tag), it pays to remember a simple guideline: Make sure that the text for the alt attribute is relevant to the image—anything less will frustrate your visitors.

- On the subject of relevance: Be sure that your images are relevant to the text. An image of an airplane works great if you're talking about travel plans, but means nothing if you're talking about wildlife.
- You can find images all over the Internet, and saving them to your own computer for use later is easy (see the following Tip). Just as in the publishing world, however, graphic designers can protect their images by copyright. If you've found an image you like on a commercial Web site, look for a copyright notice or other legal statement that indicates whether the image is free for the taking. There are plenty of free images available on the Internet without using copyrighted material.



Tip You can copy any Web image to your own computer, as long as it isn't protected by copyright. Just right-click on the image (or hold down the mouse button if you are on a Macintosh computer) and select Save Image As from the pop-up menu. Save the file on your own computer and use it as you would any other image file.

Table 8.2 lists the HTML tags that were discussed in this lesson.

TABLE 8.2 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code></code>		Adds an image to an HTML document.
<code><object></code>	<code></object></code>	Adds an object (can be used for images) to a HTML document.

In this lesson, you've learned:

- The two most frequently used graphics file formats found on the Web are GIF and JPEG. JPEG is used primarily for realistic, photographic-quality images; GIF is used for almost everything else. PNG is expected replace GIF sometime in the future.
- All images are added to HTML documents with the image tag and the source attribute, ``.
- You can use the `<a>` tag to link an image to another document.
- Images are part of the fun of Web pages, but they are also part of the problem; larger file sizes mean longer page load times.



LESSON 9

Mapping Images

In this lesson, you'll learn how to use image maps to link one image to many pages.

What Are Image Maps?

You've learned how to use an image to link to another page, but did you know that you can subdivide a single image and link each part of that image to another page? This type of subdivided image is called an *image map*.



Image Map An image that is divided into pieces that are linked to (or, mapped to) more than one resource, such as an HTML page, a file, or another image.

You've probably seen image maps on the Web, even if you didn't know what they were. Rather than creating a different image for each button in the navigation bar, many Web designers create a single image that contains all the buttons and then use image maps to link each button to the appropriate page. Figure 9.1 shows one example of such a navigation bar. Look at the following additional examples:

- The Amazon Web site (<http://www.amazon.com>) also uses image maps in its navigation bar.
- The map on the Travel Alberta Web site (<http://www1.travelalberta.com/content/maps/>) uses an image map to direct visitors to information about specific regions of the province.

- The Johnson's Baby Soft Web site (<http://johnsons.babysoft.com/>) uses an image map to link to specific product information for the products shown on its home page.



FIGURE 9.1 This image has been subdivided into four parts (one for each button on my navigation bar). Notice that some pieces of the image will not be linked to anything. Do not draw the boxes on your own image; I included them for demonstration purposes.

Finding the Coordinates

Like any other map, image maps have coordinates. In an image map, the coordinates, which are written as pixels, mark the corners of the piece of the image that will be linked to a specific URL. Before you can create any image map, you have to know the coordinates for your image.

Many image editors are available that can help you determine these coordinates and give them to you in a file so that you can cut and paste them into your HTML document. However, it's a lot easier to let your image editor do the work for you. Just type "create image map" into your favorite Web search engine to find several applications.



Tip A free trial version of Paint Shop Pro can be downloaded from the Corel® Web site at <http://www.corel.com/servlet/Satellite?pagename=Corel3/Trials/DownloadContainer>.

If you want to create your own image map, use the image editor to find the coordinates, write them down, and add those coordinates to your HTML file. Figure 9.2 shows you how Paint Shop Pro displays the coordinates for an image. I highlighted the portion of my navigation bar that I wanted to map to my home page and Paint Shop Pro told me which coordinates to use. As the figure shows, the highlighted section is a rectangle with corners at 1, 107 and 189, 167.



Tip You can divide your image into rectangular, circular, or irregular polygon shapes. The rectangle is the easiest shape to use when you're getting started and that's the shape used in Figure 9.2.



FIGURE 9.2 Paint Shop Pro displays the coordinates of a selected region of the image in the lower-left corner.



Caution The pixel coordinates for an image mark the corners of the portion of the image you are highlighting. The coordinates are relative to the entire image, not to the position of the image on the Web page. Use your image editor to gather the coordinates and you won't get confused.

Client Side Versus Server Side

With HTML, you can create image maps that work on the client side and the server side. The following list indicates the differences:

- *Client side* When you click on a client-side image map, the Web browser does all the work to bring you to the new location. The browser selects the link that was specified for the activated region and follows it.
- *Server side* When you click on a server-side image map, the server that stores the map interprets the commands and brings you to the page to which you are linked.

So how do you know which one to use? Most Web page authors only use client-side image maps because they are faster and anyone with a version 3.0 browser or higher can view them. You can always provide text links for older browsers that don't recognize the client-side image maps. Because client-side image mapping is the type of image map used most often, it is the one you'll learn about in the following section.



Tip When you are planning your Web page design, remember that you might not need to use an image map at all. You can place several smaller images close together for the same look. As long as the areas you want to link are primarily rectangular, this process is very easy with HTML and the `` tag you learned in Lesson 8, "Using Graphics."

Creating Client-Side Image Maps

Let's get started. After you have an image and have determined the coordinates for each piece of the image, you can begin mapping your image in HTML. The following HTML sample shows the image map I created for the navigation bar shown in Figure 9.1:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
<title>Image Maps</title>
<style type="text/css">
body {text-align="center"}
</style>
</head>
<body>
<map name="NavBar" id="NavBar">
<area shape="rect" coords="270, 91, 416, 138"
href="resources.htm" alt="resources" " />
<area shape="rect" coords="139, 117, 287, 166"
href="services.htm" alt="services" "" />
<area shape="rect" coords="139, 61, 290, 111"
href="about.htm" alt="about us" />
<area shape="rect" coords="5, 84, 157, 139"
href="default.htm" alt="home page" />
</map>

</body>
</html>
```

Look at the HTML example for image maps a little closer:

- `<map name="x" id="x">` Every image map needs a name and an id. It works just like the named anchor tag `<a>` you saw in Lesson 4, “Linking Text and Documents.” It identifies the section of the HTML document that you want to reference from your image.
- `<area shape="w" coords="x" href="y" alt="z" />` An `<area />` tag is required for each portion of an image that will be linked. It identifies the shape of that portion, the coordinates for it, and the URL to which it will lead.

- `</map>` This tag closes the preceding `<map name>` tag.
- `usemap="#Map Name"` `usemap` is an attribute of the `` tag. It points the Web browser to the correct image map for this image. Notice the `#` sign that precedes the map name; it works just like creating a hyperlink to a named anchor within the current document.

The Web browser sees the image map and knows that the image will be linked. In Figure 9.3, you can see that the mouse pointer changes into a hand when it hovers over a portion of the image that is mapped, as it does when placed over any other hyperlink.



FIGURE 9.3 The image from Figure 9.2 displayed in the Web browser.

Adding Text Links for Older Browsers

Because client-side image maps can be interpreted only by version 3.0 or later Web browsers, you'll need to provide another way for your visitors to get to the other pages in your Web site. The easiest way to do this is to provide text links under your image, as shown in the following HTML sample and in Figure 9.4:

```

<p align="center" />
<a href="default.htm">Home Page</a> |
<a href="about.htm">About Us</a> |
<a href="resources.htm">Resources</a> |
<a href="services.htm">Services</a>
```

As you can see, text links are standard HTML `<a href>` links. They will follow the `` tag and direct the viewers to the same pages they could reach with the image map. Figure 9.4 shows you how these links will look in the Web browser.



FIGURE 9.4 The image from Figure 9.2 displayed in the Web browser, with additional text links provided for older browsers.

Table 9.1 lists the HTML tags that were discussed in this lesson.

TABLE 9.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code><area /></code>		Identifies the shape (circle, rect, or poly), coordinates (in pixels), and URL for each section of the image.
<code><map name></code>	<code></map></code>	Surrounds the image map and gives a reference name to be used with the <code>usemap="#map name"</code> attribute.

In this lesson, you've learned:

- Image maps link a single image to multiple Internet resources. The most popular examples of image maps on the Web are for navigation bars.
- Paint Shop Pro and other graphics programs enable you to create your image, and determine the coordinates for the image map.
- Image maps can be contained within your HTML document or in a separate file.

LESSON 10

Creating Frames



In this lesson, you'll learn to create frames. You'll also learn why some people don't like them and how you can use them effectively.

Simple Frames

HTML *frames* give you a way to display two or more HTML documents at once. Each frame in the browser window displays its own HTML document. Those documents can link to each other or remain completely separate entities.



Frame A complete HTML document that appears inside of, or alongside, one or more other HTML documents within the same browser window.

Most often, as in Figure 10.1, you'll see frames used as a navigation bar on a Web site. The navigation frame can appear on any side of the document, but you'll find it most often on the top or left margins because English (the language of the majority of Web pages) is oriented from top to bottom and left to right.

To create frames, you'll need to create a new type of HTML document, called a *frameset*. A *frameset* is a special type of HTML document that defines how many frames will be displayed and which HTML documents will appear in each frame. The frameset document for the page is displayed in Figure 10.1.



FIGURE 10.1 A simple two-frame document as it appears in the browser. The left frame contains the site's navigation bar and the right frame contains the pages to which the navigation buttons link.

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="en" lang="en">
<head>
<title>First Frameset</title>
<style type="text/css">
</style>
</head>
<frameset cols="20%,*">
<frame src="toc.htm" name="left" id="left" />
<frame src="latin.htm" name="main" id="main" />
  <noframes>
    <body>
      <p><a href="toc.htm">Table of Contents</a></p>
    </body>
  </noframes>
</frameset>
</html>
```



Tip Did you notice that the `<!DOCTYPE>` tag changed? In Lesson 2, “Creating Your First Page,” you learned that XHTML has three variations. The frameset variation is used whenever you create framed pages.

If you compare this document with a regular HTML document, you should notice right away that the `<body>` tag is missing. A frameset document uses a new tag, `<frameset>`, to replace the `<body>` tag. Within the `<frameset>` tag, you’ll see the `<frame />` tag, which is used to describe the contents of each frame, and the `<noframes>` tag, which is used to instruct the browser what to display in the event the viewer’s browser does not support frames, including adding the `<body>` tag again. Confused? Let’s take a closer look at each of these tags.

`<frameset>`

Within the `<frameset>` tag, you will need to define the orientation of the frames—in vertical columns, `cols`, or in horizontal rows, `rows`. This orientation attribute also requires you to define the size of each of your frames. For example, if you have three vertical frames in your frameset, you will need to specify three size attributes. Look again at the `<frameset>` tag in the preceding HTML sample.

```
<frameset cols="20%,*">
```

This tag defines two vertical columns. The first column is 20% of the screen width; the second column fills the remainder of the screen—80%. The asterisk (*) tells the browser to fill the remainder of the screen. You can use the same trick if you are defining more than two frames. Although it shows only two values, the following `<frameset>` tag will actually be used to define *three* horizontal rows. The first row has been set to 20% of the length of the screen; the asterisk forces the browser to equally divide the remainder of the screen between the other two rows.

```
<frameset rows="20%,*">
```

You don't have to let the browser figure out the size of your frames. If you are a perfectionist, you can do your own math and specify the size yourself. Just remember that the total value of the sizes can't be more than 100% of the screen. Now that makes sense, doesn't it?



Tip You can specify the size of your frames in pixels or as a percentage of the browser window by using the % sign as in the following tag:

```
<frameset cols="50%,50%">.
```

You don't have to use the % sign, however. You can use a forward slash (/) as an abbreviation of the % sign, as in the following tag:

```
<frameset cols="50/,50/">.
```

<frame />

Like the tag you learned about in Lesson 8, "Using Graphics," the <frame /> tag uses the src (source) attribute to tell the browser where to find the document to display. The important thing to remember when you are setting up your frameset document is that you are defining the start page for your Web site, or the first framed page in your site. You don't have to figure out every possible combination of pages that might appear, you only have to specify the first one.

The <frame /> tag also requires the name and id attributes. Most people name their frames by their location on the browser window. The <frame /> tags in the following example, for instance, call the frame that appears on the left of the screen, left, and the other frame main because it will hold the main pages of the Web site.

```
<frame src="toc.htm" name="left" id="left" />
<frame src="latin.htm" name="main" id="main" />
```



Tip You could name the frames anything (Dog, Cat, Red, or Blue), but you'll find them easier to remember if you stick to something simple.

Following are a few more attributes of the `<frame />` tag that might come in handy:

- **frameborder** With this attribute, you can remove the small border line that separates the frames. In Figure 10.2, the border has been removed from the sample frameset.

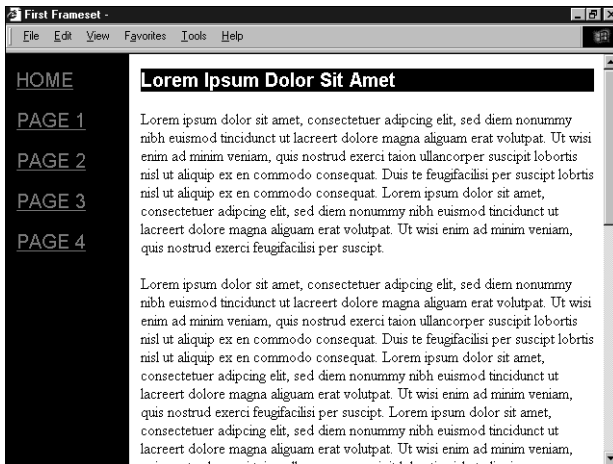


FIGURE 10.2 Figure 10.1 with the `frameborder` attribute set to "0".

- **marginwidth** or **marginheight** These attributes specify (in pixels) the space between the border and the text in the frame.
- **scrolling** Using the values of `yes`, `no`, or `auto`, you can tell the browser whether or not to add a scrollbar next to the frame. Don't worry, however; even if you've specified `scrolling="no"`, the browser will display a scrollbar if the content of the frame exceeds the size of the frame.
- **noresize** Just like any other window, you can resize frames manually by dragging the frame's border (even when the `frameborder="0"` attribute has been specified). You can prevent that capability by specifying the `noresize` attribute in your frameset.

<noframes>

The `<noframes>` tag that appears in the preceding example tells the browser what to do if it doesn't know how to display frames, or if your visitor has adjusted his browser's settings to refuse frames.

```
<noframes>
<body>
<p><a href="toc.htm">Table of Contents</a></p>
</body>
</noframes>
```

The `<noframes>` tag is not required and many Web page authors choose to ignore it, but it takes very little effort to add it and it makes good sense if you want to be certain that everyone will be able to view your Web site.

While it's true that many authors ignore the `<noframes>` tag, you'll find that just as many authors choose to create an entire non-framed version of their Web site. I happen to think that's overkill.

If you are using frames as a navigation bar, you could make a couple simple changes to your main HTML pages to help people who can't see the frames navigate your site. In Figure 10.3, I've made the same frameset document from Figure 10.1 work for people who can't see frames. By adding a simple one-row table to hold a duplicate set of navigation elements, someone who stumbles upon your page, but can't see the frame on the left, can still navigate the site.



FIGURE 10.3 Adding a navigation bar to the top of each of the main pages will make this site work for those people who can't view framed pages.

Nested Frames

You might want to be more creative with your frame layout. You can use the `<frameset>` tag more than once in a single frameset document. This feature enables you to nest frames within each other. Following is an example of a nested frame. I indented the second `<frameset>` to make it easier to read.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title>First Frameset</title>
<style type="text/css">
</style>
</head>
<frameset rows="15%,*,10%">
<frame src="sitename.htm" name="top" id="top" />
  <frameset cols="20%,*,11%">
    <frame src="toc.htm" name="left" id="left" />
    <frame src="latin.htm" name="main" id="main" />
    <frame src="motto.htm" name="right" id="right" />
  </frameset>
<frame src="contacts.htm" name="bottom" id="bottom" />
<noframes>
  <body>
    <p><a href="toc.htm">Table of Contents</a></p>
  </body>
</noframes>
</frameset>
</html>
```

The first `<frameset>` tag defines three horizontal frames, but the second `<frameset>` tag divides the middle row into three column frames. Figure 10.4 shows you how this nested frameset will appear in the browser.



FIGURE 10.4 All the borders have been left showing to help you see where the nested frames are in this example.

<iframe>

You can create a frame one more way by using the `<iframe>`, or inline frame, tag. Rather than creating a separate frameset document, you define an inline frame within a regular HTML document because it appears in the middle of another document. Figure 10.5 shows the same content as the sample in Figure 10.4, but this page was created using an inline frame. You can see the HTML document for this page in Figure 10.6.

You can apply all the same attributes for regular frames to the `<iframe>` tag except the `noresize` attribute because unlike regular frames, inline frames cannot be resized.



Caution As of right now, the `<iframe>` tag only works with Internet Explorer 4 and higher browsers, so don't try to use it unless you know your audience has one of those browsers.

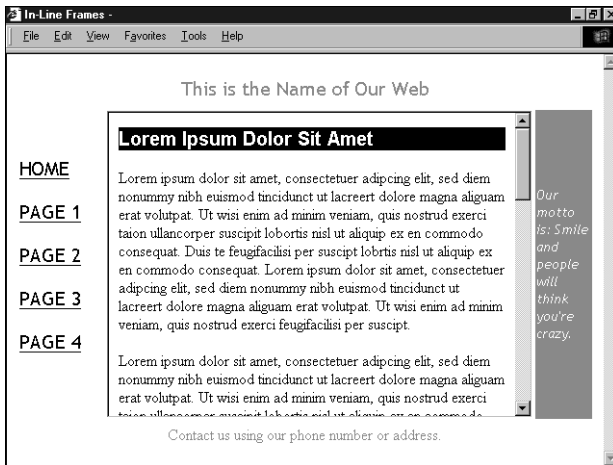


FIGURE 10.5 The scrollable document in the center of this page was added with an inline frame.

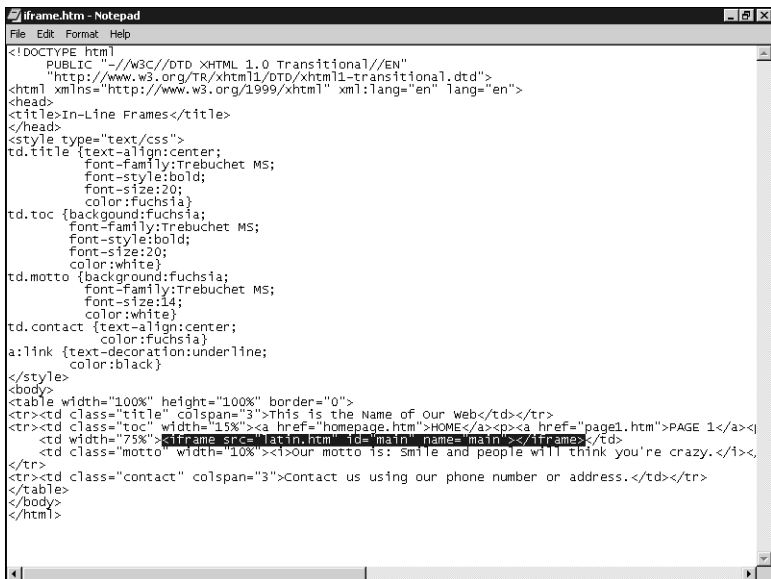


FIGURE 10.6 Style sheet properties are used to help define the colors and fonts for this document. The `<iframe>` tag is actually embedded inside a table to achieve the page layout shown in Figure 10.5.



Tip Throughout the book, you have been cautioned that not all browsers support all the HTML tags you've learned. For a fee, WebSideStory's StatMarket® tracks browser usage and other fun statistics on its Web site (<http://www.websidestory.com/services-solutions/datainsights/statmarket/overview.html>). You can use that knowledge to decide whether you are willing to take the risk of using a tag such as `<iframe>`, which is not supported by all browsers.

Linking Between Frames

Think back to Lesson 4, "Linking Text and Documents," when you learned how to create hyperlinks. You'll remember that you can use the `` tag to name an anchor, or target, within a document that could be linked to directly, as shown in the following code.

```
<a name="PointA" id="PointA">Point A</a>
```

You'll remember that you need to use the anchor tag, ``, to surround the text that you want to highlight, as shown in the following example:

```
<a href="DOC2.htm#PointA">Click Here to go to Point A</a>
```

All frames also have the `name` and `id` attributes assigned to them. You can use that name to specify which frame you want your hyperlink to open in. Let's look at the HTML code for the `toc.htm` file used in the preceding examples.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">

<head>
<title>Table of Contents</title>
<style type="text/css">
```

```
</style>
</head>
<body>
<p><a href="homepage.htm" target="main">HOME</a></p>
<p><a href="page1.htm" target="main">Page 1</a></p>
<p><a href="page2.htm" target="main">Page 2</a></p>
<p><a href="page3.htm" target="main">Page 3</a></p>
<p><a href="page4.htm" target="main">Page 4</a></p>
</body>
</html>
```

Now you see a new attribute has been attached to the `<a href=` tag: `target`. The `target` attribute refers to the *target frame* for the hyperlink. Besides the frame names that you've specified in your frameset document, you can target the following other three names:

- `` This tag opens the hyperlink in a new browser window.
- `` This tag opens the hyperlink in the same window where the hyperlink was. If the hyperlink was in a frame, the link will open in the same frame, replacing that frame document.
- `` This tag opens the hyperlink in the same browser window. If the hyperlink was in a frame, the link will open in the same frame, replacing the entire frameset.



Target Frame The name of the frame in which a hyperlink will open.



Caution Always specify the `target` attribute whenever you are working with frames. If you don't specify the target frame, the browser generally will replace the current frame with the target document, which might not be what you'd intended.

The Two Biggest Problems with Frames

Mention frames to any Web site developer, and you'll be sure to get an earful. Good or bad, people always have an opinion. You've already seen how useful they can be at providing navigational information, but let's see why so many people dislike them.



Tip Jakob Nielsen, one of the Web's most respected usability experts, maintains a Web page called (and pardon my French), *Why Frames Suck (Most of the Time)*, in which he discusses some of the many problems users have with frames. You can read his original 1996 article at www.useit.com/alertbox/9612.html, which was later updated in 1999 (<http://www.useit.com/alertbox/990502.html>). Though not a recent article, the concerns he raises are still valid today.

So Many Pages, So Few URLs

When you load your frameset document into your browser, you are telling the browser to load all the pages into this same document, following the selected hyperlinks. So? Take a look at the URL for your frameset document. My URL is `C:\Webshare\wwwroot\sams\frame2.htm`. No matter how many times I click the hyperlinks in my framed pages, the URL stays the same because all those framed pages are loading into the same frameset document.

Why is that a problem? Suppose that my best customer is browsing my site and she is looking at the *wonderful* information on `Page4.htm`. She decides to save the URL in her Favorites (or bookmark it) so that she doesn't have to search for the information again. The URL she saves is `C:\Webshare\wwwroot\sams\frame2.htm`, not `C:\Webshare\wwwroot\sams\page4.htm`, which is what she was actually looking at. There is no guarantee that when she opens that URL in the browser, it will open on `Page4.htm`, as she wanted in the first place. How frustrating!

If you right-click the mouse (or hold the mouse button if you have a Macintosh), you can click Properties in the shortcut menu. On the Properties dialog box is the URL for that particular page (see Figure 10.7). You can highlight and copy that URL into the Address field of your browser to open later. When you do this, however, you will not see the framed version of the site; you will only see the single document that you saved, with no additional navigation to help you.

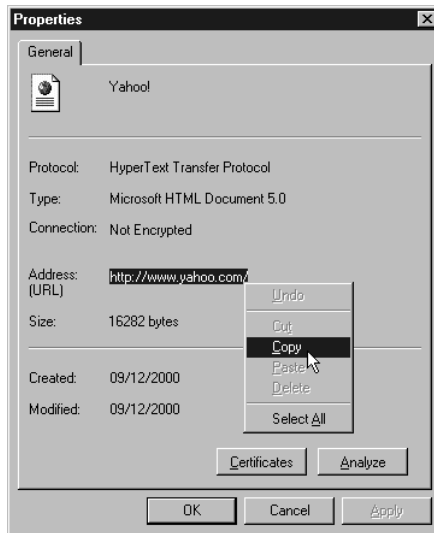


FIGURE 10.7 The Properties dialog box contains the actual URL for a framed page. You can copy the URL using the shortcut menu after you highlight the text.

Printing

Another huge problem for users of your framed Web site is *printing*. Why should printing be a problem? As much as we like to think that we are headed toward paperless offices and online commerce, people still like to print documents. When most people see a page that they want to print, they click the Print button on their browser. With older browsers, the Print button only prints the *active frame*.

**Active Frame** The last frame that you clicked.

On my sample framed site, if you clicked a link in the left frame to open a new document in the main frame and then clicked the Print button without making any other mouse clicks, you would actually print the navigation bar on the left frame, not the document in the main frame that you wanted. Unfortunately, the browser doesn't know which frame you want, only which frame was last active.

The newer browser versions have included a new feature in their Print dialog boxes—a Print Frames option that allows you to specify whether you want to print the active frame or the entire frameset (see Figure 10.8). A big improvement, but you can't guarantee which browser your visitor is using.

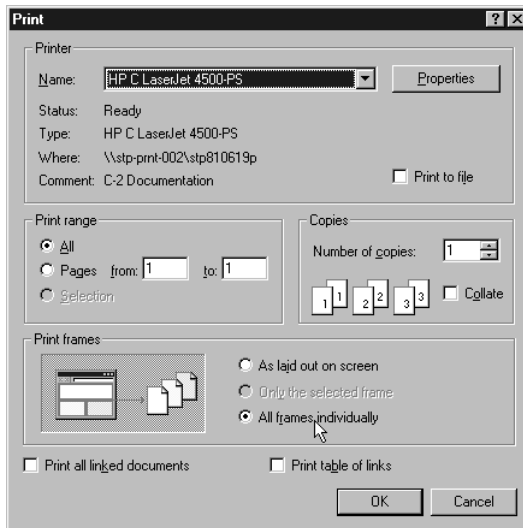


FIGURE 10.8 The Print dialog box from Internet Explorer.



Tip If you want to print a single frame, you can click the right-mouse button, or hold down the mouse button if you are using a Macintosh, and select the Print option from the shortcut menu. The printer will print only the selected frame, not the entire frameset.

Using Frames Effectively

Although frames have some usability problems, there are some obvious advantages for using them. Just make sure that you use them the right way. Here are some tips to help.

- *Frames are not a toy* Frames work best when used as a navigation tool, or when it makes sense to show two or more elements of a document at the same time.
- *Remember the target attribute* Nothing is worse than clicking a hyperlink in a framed document and breaking out of frames unintentionally. Worse, each hyperlink in a framed document that does not include the target attribute has the potential of opening in a new browser window. You could end up with a real mess.
- *Include the <noframes> tag* Always remember that there are people who can't see frames (either because of older browsers, or because they set their browser preferences to ignore them). Provide alternate content with the <noframes> tag.
- *Include alternate navigation within the main frame* With the Web, there is no guarantee that your visitors will always arrive at your home page and see the frameset as you intended them to see it. Sometimes, they will arrive on an individual frame. If you provide additional navigation links within those pages, your visitors will still be able to move within your site.
- *Never frame other framed pages* Not as frequent anymore on the Web, but when frames first became available, Web page authors framed everything, including other framesets. This compound-framing is very confusing to users.

Table 10.1 lists the HTML tags that were discussed in this lesson.

TABLE 10.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<code><a href></code>	<code></code>	Creates hyperlinks to other documents. Always use the <code>target</code> attribute with frames.
<code><frameset></code>	<code></frameset></code>	Replaces the <code><body></code> tag in a frameset document and surrounds the <code><frame></code> and <code><noframes></code> tags. This tag must include the attribute to describe the orientation of the frames and their size.
<code><frame /></code>		Includes the frame's name and <code>id</code> and a URL for the content (<code>src</code>). It also might include attributes to define the border and scrolling.
<code><iframe></code>	<code></iframe></code>	Embeds a frame inside another document. It only works with Internet Explorer.
<code><noframes></code>	<code></noframes></code>	Defines an alternate viewing page for browsers that don't support frames.

In this lesson, you've learned:

- A frameset document defines the number of frames and their sizes; standard HTML documents will be contained in the frames.
- Each frame of a frameset document must be named so that you can direct your hyperlink to appear in a specific *target* frame.
- Despite their obvious advantage for organizing your site's navigation elements, many people dislike frames because of usability problems associated with them.

LESSON 11

Building Online Forms



In this lesson, you'll learn how to create Web forms that enable you to get input from your visitors.

Creating Forms

You've seen forms on the Web, but I'll bet you didn't know they were so easy to create. I want to point out a couple of things for you to keep track of as you read this section and then you'll create a form.

- Forms are made up of fields (that you want the user to fill out) and buttons (to perform actions such as submit and reset).
- Every field (`<input type="type" />`) should have name and id attributes as well.
- Every field can be set to have a default value (a pre-selected option that the users can overwrite if they want); many also can be set to validate the data the user enters.
- Every form requires a Submit button that sends the form data to the address specified in the action attribute of the `<form>` tag. It has its own `<input />` tag and you can read more about it in the "Buttons" section later in this lesson.

One more thing: A form isn't a form until it is enclosed within the `<form>` tag. The `<form>` tag always includes an action and a method attribute. To make it simple, a form's method is almost always set to post and the action can only be one of two values: an e-mail address of the person who will be receiving the form's data, or a URL of a file that will be receiving the form's data. We're going to use the e-mail option because

it's easier for you to practice with. Figure 11.1 demonstrates an online form using the following simple `<form>` tag. Figure 11.2 shows the full HTML document for the form shown in Figure 11.1.

```
<form method="mailto:youremail@yourisp.com" action="post">
```

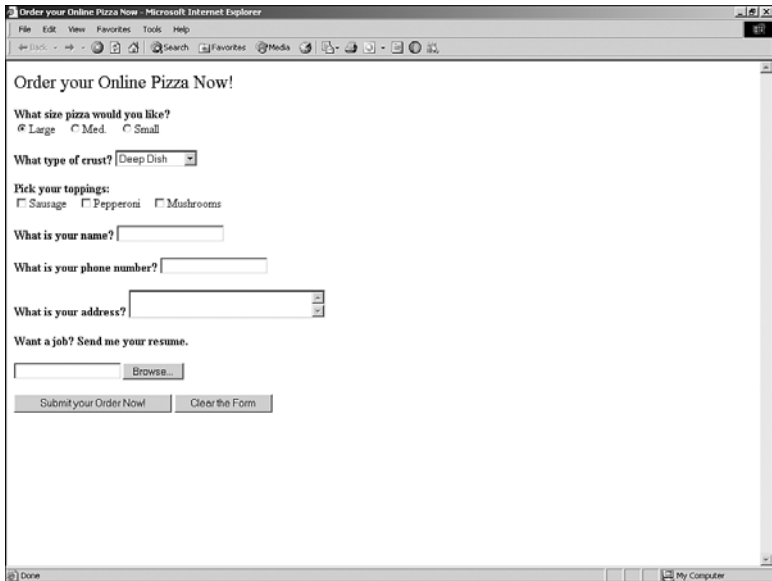
A screenshot of a Microsoft Internet Explorer browser window displaying a web form titled "Order your Online Pizza Now!". The form contains several input fields: a radio button group for pizza size (Large, Med., Small), a dropdown menu for crust type (Deep Dish), a checkbox group for toppings (Sausage, Pepperoni, Mushrooms), and text input fields for name, phone number, and address. There is also a checkbox for "Want a job? Send me your resume." and a "Browse..." button. At the bottom, there are "Submit your Order Now!" and "Clear the Form" buttons. The browser's address bar shows "http://localhost:8080/" and the status bar shows "Done" and "My Computer".

FIGURE 11.1 This Web form contains each of the input types (fields) discussed in this lesson.

Don't forget that an HTML form is just like any other HTML document; it doesn't recognize extra spaces. If you want to line up your fields for a more professional-looking form, line up your form fields in tables, as shown in Figure 11.3, and use style sheet properties to define your fonts and add images.

```

pizza_form.html - Notepad
File Edit Format Help
<!DOCTYPE html>
<html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Order your online Pizza now</title>
</head>
<body>
<p><bfg><bfg>order your online #Pizza Now!</bfg></bfg></p>
<form method="POST" action="mailto:pizza@pizza.com" name="pizza_order" id="pizza_order">
<p><strong>What size pizza would you like?</strong>
<input type="radio" value="large" name="size" id="size" checked="checked" tabindex="1" />Large<br>
<input type="radio" value="med" name="size" id="size" tabindex="2" />Med.<br>
<input type="radio" value="small" name="size" id="size" tabindex="3" />Small</p>
<p><strong>What type of crust?</strong>
<select name="crust" id="crust" size="1">
<option value="deep dish" tabindex="4">Deep Dish</option>
<option value="hand-tossed" tabindex="5">Hand-Tossed</option>
<option value="thin & crisp" tabindex="6">Thin & Crisp</option>
</select></p>
<p><strong>Pick your toppings:</strong><br>
<input type="checkbox" name="toppings" id="toppings" value="sausage" tabindex="7" />Sausage<br>
<input type="checkbox" name="toppings" id="toppings" value="pepperoni" tabindex="8" />Pepperoni<br>
<input type="checkbox" name="toppings" id="toppings" value="mushrooms" tabindex="9" />Mushrooms</p>
<p><strong>What is your name?</strong>
<input type="text" name="name" id="name" size="20" tabindex="10" /></p>
<p><strong>What is your phone number?</strong>
<input type="text" name="phone" id="phone" size="20" tabindex="11" /></p>
<p><strong>What is your address?</strong>
<textarea rows="2" name="address" id="address" cols="30" tabindex="12">Enter the address here.</p>
<p><strong>Want a job? Send me your resume.</strong>
<input type="file" name="resume" id="resume" tabindex="13" /></p>
<p><input type="submit" value="Submit your Order Now!" name="b1" tabindex="14" />
<input type="reset" value="Clear the Form" name="b2" tabindex="15" /></p>
</form>
</body>
</html>

```

FIGURE 11.2 Here's the HTML document for the form shown in Figure 11.1.

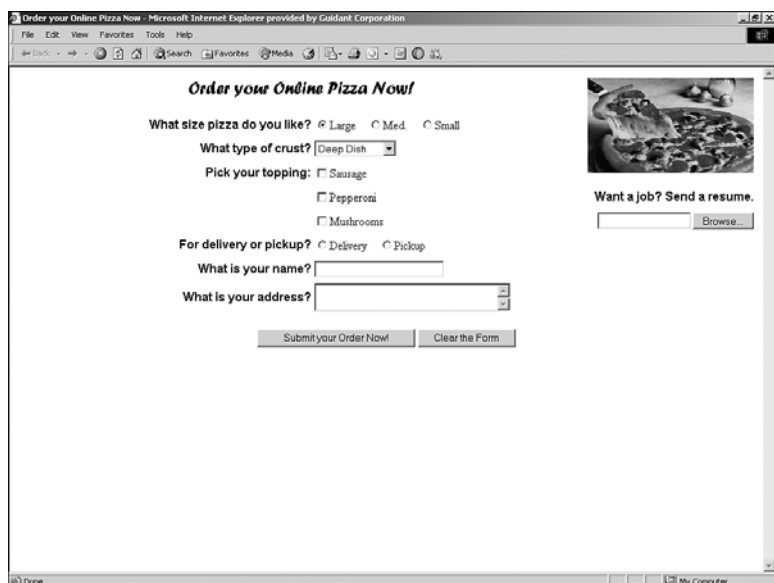


FIGURE 11.3 This version of the form took a little longer to create, but the results are worth it.

Form Fields

The main reason to create a form is to collect data. The fields on a form help you do that. The following sections describe each of the field types and give you some hints for how each one can be customized to suit your needs.

Text Box

The simplest form of data collection is an empty box. Your form poses a question (“What is your name?”) and your visitor fills in the answer in the space provided. In HTML, this type of field is called a *text box*. HTML uses the `<input />` command to identify a form field. The following example is a complete HTML form with one field: a text box that is 40 pixels wide and is called Name.

```
<form action="mailto:youremail@yourisp.com" method="post">
<p><b>What is your name?</b></p>
<p><input type="text" name="Name" id="Name" size="40" /></p>
<p><input type="submit" value="Submit" name="submit"
      id="submit" /></p>
</form>
```

The form field’s attributes (type, name, id, and size) help to customize the form field. name, id, and size are obvious, but the type attribute could use some explanation. Although this type of field traditionally is called a *text box*, you also can set the type attribute to password (which displays an asterisk when the user types his or her password). If you know that your visitors will be using Internet Explorer, you could also set the type attribute to integer (which is a whole number without decimals) or number (which can include decimals). One more type, file, is explained in the “File Select” section found later.



Tip The `tabindex` attribute, shown within the `<input />` tags on Figure 11.2, sets the order in which the user can navigate through the form elements using the Tab key. The `tabindex` attribute and the index number increases toward the bottom of the form.

list. The following code sample demonstrates how this option would look. Figure 11.4 illustrates this option in an online form.

```
<option value="Pick">Pick One</option>
```

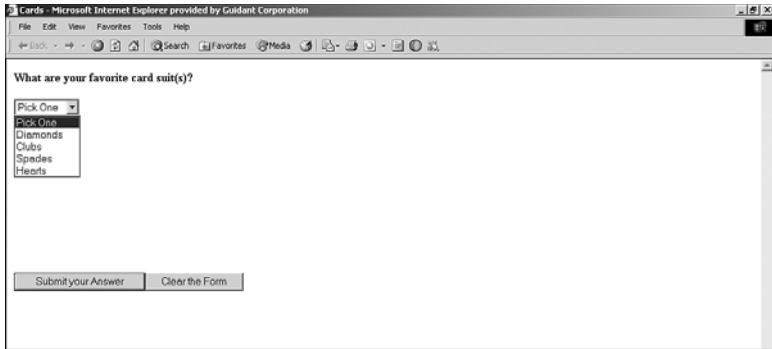


FIGURE 11.4 This drop-down menu includes an extra `<option>` tag for the Pick One statement.

Like check boxes, your user can hold the Ctrl key to select multiple options in the drop-down menu if you add the `multiple` attribute to the `<select>` tag. This change (shown in the following HTML sample) enables users to select multiple options by pressing and holding down the Ctrl key while clicking on the options in the menu. (See Figure 11.5.)

```
<form action="mailto:youremail@yourisp.com" method="post">
<h3>What are your favorite card suit(s)?</h3>
<select name="suit" id="suit" size="1" multiple="multiple">
  <option value="Hearts">Hearts</option>
  <option value="Diamonds">Diamonds</option>
  <option value="Clubs">Clubs</option>
  <option value="Spades">Spades</option>
</select>
<p><input type="submit" value="Submit your Answer"
name="submit" id="submit" />
<input type="reset" value="Clear the Form" name="reset"
id="reset" /></p>
</form>
```

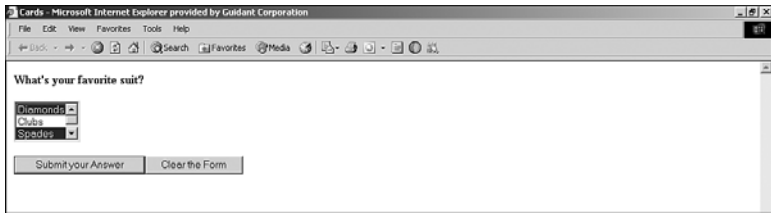


FIGURE 11.5 Changing the `size="1"` attribute from the drop-down menu code sample to `size="3"` enables the visitor to see three options at once. Here, two options have been selected.

File Select

Another useful way to gather information from your visitors is to allow them to send you files. You might use this input option, the `file` type, to collect résumés, orders, or any other type of file. The following HTML sample demonstrates how this option is created. As you can see in Figure 11.6, the browser creates a Browse button to help your visitors send their files.

```
<form action="mailto:youremail@yourisp.com" method="post">
<p>Send me your resume.</p>
<p><input type="file" enctype="multipart/form-data"
      name="resume" id="resume" /></p>
<p><input type="submit" value="send now"
      name="submit" id="submit" /></p>
</form>
```



FIGURE 11.6 The visitors will click the Browse button to browse their own file system in search of the appropriate file.



Caution Check with your Web host before creating this type of form; it might have additional requirements for you.

Buttons

The Submit and Reset buttons are special types of form elements. Although they are created using the `<input />` tag (see Figure 11.2), they are not data collection tools, but actually are data submission tools.

- The Submit button collects all the data from the form and *posts* (sends) it to the location specified in the action portion of the `<form>` tag.
- The Reset button clears the form of any data that might have already been completed. The Reset button *resets* the form to the original pre-selected values.

The Submit button is required on all forms, but the Reset button is optional. The browser's Refresh button has the same effect as the Reset button on a form. It reloads the page and deletes everything except the initial values of the form.



Tip There is one more input type to be aware of: the hidden type. A hidden field is not displayed on the form, but returns results anyway. You might want to collect the date and time the visitor submitted the form, the version of the form that was submitted, or the name of the person who should receive the data. Create the field based on the HTML example that follows:

```
<input type="hidden" name="version"
      id="version" value="B" />
```


Receiving Form Data

When your visitors click the Submit button on a form on your Web site, the data they entered into the form will be sent to you using the action you specified in your `<form>` tag. In Figure 11.2, we selected an e-mail action. Figure 11.7 shows you how my e-mail software returns the form data to me. You should now see why it is so important to include the name and id attributes associated with every form field.

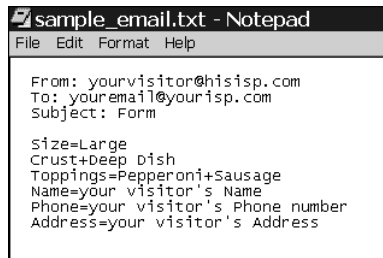


FIGURE 11.7 Your e-mail software may format the responses differently, but they will all show the field names (Size and Toppings, for example), along with the data your visitor entered into those fields.

It is not always convenient to receive form data via e-mail, particularly if you expect to receive a lot of responses. Reading, and responding to, that many e-mail messages can become tiresome. Your ISP also might prefer that you do not use its mail servers in this manner.

Another action that you can assign to your forms is a script to handle the responses for you. Scripts are automated form handlers and can be used to collect all the responses in a single file and respond to the visitors for you. This book can't begin to explain how to write the scripts, or find them, but your ISP, or your network administrator, probably will have several scripts available for you to choose from and can help you attach them to your form.



Caution All ISPs handle forms differently. Always check with your ISP or network administrator before attempting to create forms for your Web site.

Table 11.1 lists the HTML tags that were discussed in this lesson.

TABLE 11.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<form>	</form>	Encloses all form elements.
<input />		Identifies a form field.
<option>	</option>	Identifies the contents of a drop menu.
<select>	</select>	Encloses a drop-down menu field.
<textarea>	</textarea>	Identifies a multi-lined text field.

In this lesson, you've learned:

- All form fields should have name, id, and tabindex attributes.
- The <form> tag always includes a method attribute (which is usually post) and an action attribute. The action can be either an e-mail address or a URL of a file that will be receiving the form's data.
- The six form field types are text box, text area, radio buttons, check boxes, drop-down option menus, and the file browse box.



LESSON 12

Making It Sing: Sound and Video

In this lesson, you'll learn how to add sound and video to your Web pages, and find the plug-ins required to use them.

Adding Sound and Video

Used correctly, sound and video clips can greatly enhance the content in your Web pages. Imagine a Web page about Dr. Martin Luther King, Jr. that didn't include something about his famous "I Have a Dream" speech. The text of the speech is moving, but the delivery is what made it so powerful. You can add sound and video clips to your own Web pages using some HTML tags you've already learned.



Tip You can hear the "I Have a Dream" speech at <http://www.americanrhetoric.com/speeches/Ihaveadream.htm>.

This lesson discusses several methods for adding sound and video clips. You should know that the one method that is sure to work with every browser on every platform is also the simplest: the `<a>` tag.

```
<a href="http://www.americanrhetoric.com/mp3clips/
politicalspeeches/mlkihavedream35348.mp3">
Listen to the I Have a Dream speech.</a>
```

When your visitors click on the words *Listen to the I Have a Dream speech*, the `mlkihavedream35348.mp3` file will download to their computers and begin playing. If the visitors do not have the correct plug-in to

hear the sound clip, the browser should prompt them to save the file for later. It will not, however, prompt them to download the correct plug-in. You will need to provide that information on your page.



Tip Try typing "sound clips" into your favorite search engine to find sound files you could use on your own pages.

Video clips can be handled in exactly the same way:

```
<a href="
  http://www.lucidcafe.com/library/96jan/96jangifs/
  MLK630828Video.ram">
```

Watch a video of Dr. King's greatest speech.



Caution Use sound and video sparingly and make the wait worthwhile. Even short clips can have a large file size and may take a very long time to load. Make sure that you give your visitors some idea of the content of the clip so that they can decide whether to wait for the download.

<embed>

Netscape invented a new tag called <embed> to enable you to include a sound or video clip on a Web page. Microsoft's Internet Explorer browser also accepts this non-standard tag. The following sample shows how the <embed> tag works to add a video clip.



Tip If you'd like to duplicate the following sample, you can download the video clip from http://download.microsoft.com/download/0/9/d/09d051c4-decc-4d39-9c57-f520187213a1/Amazing_Caves_720.exe and save it to your own desktop. Then just change the src attribute to suit your needs.

```

<p>Click the video to see a clip from the IMAX movie Journey
into Amazing Caves.</p>
<embed src="C:\Documents and
Settings\qc04818\Desktop\Amazing_Caves_720.wmv"
width="360" controls="true" autostart="true"
loop="false">
<noembed>
  <a href="C:\Documents and
  Settings\qc04818\Desktop\Amazing_Caves_720.wmv">
    Click for a surprise.</a>
</noembed>
</embed>
<p>You will need the <a href="http://www.microsoft.com/
windows/windowsmedia/download/
AllDownloads.aspx?displang=en&qstechnology=">
Windows Media Player</a> to see this clip.</p>

```

The browser displays the control panel for the media device you use, as shown in Figure 12.1. You can add width and height attributes to the `<embed>` tag to control the size of the video still. The `<noembed>` tag provides an alternate way for visitors to download the video if their browser doesn't recognize the `<embed>` tag.

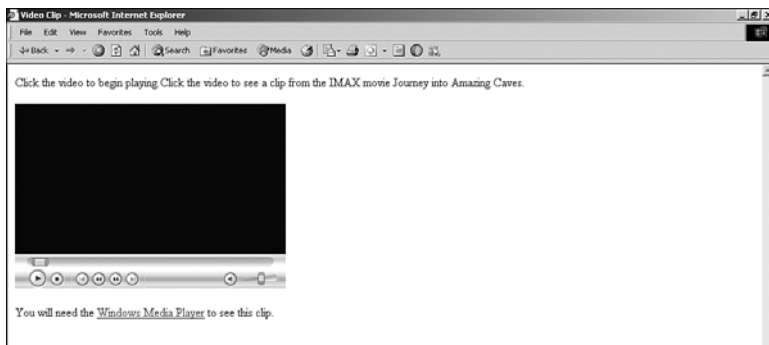


FIGURE 12.1 When embedded, a control panel for the device used to play the video or sound clip, appears in the browser.

Notice that I've included four attributes for the `<embed>` tag in this example: width, controls, autostart and loop. Width sets the width of the video. It is not necessary to set the height since the video will size the image proportionally. Controls ensures that the video player controls are

included. Autostart, which you can set to true or false, tells the browser whether to begin playing the clip immediately upon loading the page. The loop attribute tells the browser how many times in a row to play the clip before stopping. You can set the loop attribute to any whole number.

Another attribute you can set is autorewind. This attribute is automatically set to true, but if for some reason you *don't* want to rewind the clip after it plays, you can set it to false. The hidden attribute hides the player's VCR controls from the user. Hiding the controls gives you, the developer, more control over the use of the clip, but may annoy your visitors if they are not allowed to turn your clip on or off.

<object>

The <embed> tag is nonstandard, which means that the W3C doesn't recognize it as a legitimate HTML markup tag. The W3C prefers that Web page developers include sound and video clips using the <object> tag. You will learn more about this tag in Lesson 14, "Creating Active Web Pages."

```
<p>Click the video to begin playing.</p>
  <object classid="C:\Documents and
    Settings\qc04818\Desktop\Amazing_Caves_720.wmv">
  </object>
<a href="C:\Documents and
  Settings\qc04818\Desktop\Amazing_Caves_720.wmv">Surprise!</a>
```



Caution Unfortunately, the <object> tag does not yet work consistently in all browsers, although it is the W3C-preferred tagging method. Be prepared to either always include an alternate <a href> tag for sound and video elements, or duplicate your work by including the <embed> tag as well.

The <object> tag also comes with a multitude of attributes that you can set to help control the use of the clip in your Web page. You can add a border around the object by using the appearance attribute and setting the value to 1. The <object> tag also has an autostart and autorewind attribute.

Finding Plug-ins

We've mentioned before that it's never a good idea to include any items on your Web page that require a plug-in (such as Flash or RealAudio) without also providing a link to the plug-in. You don't want to assume that your visitors will have the required software to view your page because they might not.

Microsoft generally believes that the browser itself should contain enough code to run any scripts, applications, and embedded items without loading plug-ins, and it uses ActiveX controls to handle these types of events. Netscape, however, agrees with the idea that browsers should be light and plug-ins should be used to handle outside events. Partially because of this belief, Netscape maintains one of the best plug-in archives on the Web at <http://browser.netscape.com/ns8/community/plugin.jsp>.



Tip If you're looking for a browser-neutral source of downloadable plug-ins, check out CNET at <http://www.download.com/sort/3120-20-0-1-5.html?qt=browser+plug-ins>.

Table 12.1 lists the HTML tags that were discussed in this lesson.

TABLE 12.1 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<embed>	</embed>	Netscape's nonstandard, although largely supported, tag for including sound and video clips.
<noembed>	</noembed>	Netscape's tag that provides an alternate method of downloading the clip for browsers that don't recognize the <embed> tag.
<object>	</object>	W3C's preferred, although largely unsupported, tag for including sound and video clips.

In this lesson, you've learned:

- Sound and video clips can be added to your Web page with the `<a>` tag.
- Both the `<object>` and the `<embed>` tags enable you to add a video clip with the video controls (start, stop, and so on) to your documents.



LESSON 13

Designing with HTML

In this lesson, you will learn some designer tricks of the trade to make your pages look as good as they work.

Design Basics

Web design may have had its roots in traditional paper design, but online design is different. One of the biggest differences when designing for online is the capability to *hyperlink*. Adding hyperlinks in your Web pages gives you the capability to quickly direct your viewers to the information you want them to see, including reference material on, or off, your Web site. Unfortunately, the capability to link also is one of the biggest disadvantages to online design. Occasionally, viewers get so caught up in clicking on all those “for additional information click here” links that they forget what they were looking for in the first place; in effect, they get lost in cyberspace.

Web site designers have a number of design elements available to help them make it easy for their users to recognize which Web pages are part of the same Web site. These elements can also help set the mood for their Web site. The layout, images, navigation buttons, bullets, lines, colors, and even the fonts you choose should support the overall design theme of your site. In the following sections, you’ll learn how each of these elements works together.

To design an effective Web page, you’ll need to be aware of the differences in moving from traditional design to online design. Table 13.1 summarizes some of the differences. Knowing the problems you’ll face is only half the battle; the rest is knowing how to avoid them. You’ll learn that in the sections that follow.

TABLE 13.1 Paper Design Versus Online Design

Paper Design	Online Design
Viewers follow content along a linear path with a beginning, middle, and an end.	Using search tools or hyperlinks, viewers can access the content at any point. The only way for you to control that movement is to provide hyperlinks and navigation.
Viewers can see an entire page (text and graphics) at the same time.	With larger graphics (or non-graphical browsers), viewers often have time to read the entire text before they ever see any images.
Serif fonts (such as Times Roman) usually are used for content; sans-serif fonts (such as Arial) usually are used for headings.	Sans-serif fonts usually are used for content; serif fonts usually are used for headings.
Viewers see an entire page (or multiple pages in a book or magazine layout). The size of the page, and the amount of content presented on it, are controlled by you, the author.	Viewers see only the amount of content that will fit on their monitor at one time, which often is only a couple of paragraphs of text. The viewer controls the presentation of the content with the size of their monitor and the browser settings.

Two whole fields of study, Information Design and Usability, are devoted to finding the most effective methods of communicating your message. Researchers in these fields have come up with some standard design guidelines that can help you make the most of the material you have to present. Following are some facts I'll bet you didn't know:

- Red, yellow, orange, and green are the most difficult colors of text to read online. It's best not to use them or to use them sparingly. You'll learn more about colors and fonts in the subsequent sections.

- Your visitors read almost 50% slower online than on paper. You can counter that by keeping your page length short (no more than two to three screen lengths) and providing tables and bulleted lists to give their eyes a rest from large blocks of text.
- Animated images and moving text catch the eye of potential visitors, but most people find them annoying if they continue to move while the visitor is trying to read or search for content on the page. You'll learn about these features in Lesson 14, "Creating Active Web Pages."
- If your visitors are looking for a particular piece of information, they will search your site for less than a minute before moving on to some other site, unless they are confident that you have the information they are looking for. A well-designed Web site will help your visitors find their information quickly. You'll learn how to do this in the "Layout, Content, and Navigation" section.

Layout, Content, and Navigation

Because people tend to read online text more slowly than paper text, Web site designers use *page layout* techniques to help make content more readable.



Page Layout The arrangement of text, graphics, and *whitespace* on a page.

Whitespace Refers to the background of a page. Note that this space does not have to be white.

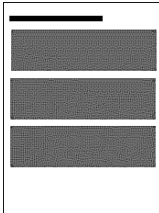
In general, when designing a Web site, you will need to keep the following key tips in mind. Figure 13.1 shows you how some of these layout tricks work to emphasize your content.

- *Keep paragraphs short and include a margin* Keep your paragraphs under ten lines and include a margin. If you want viewers

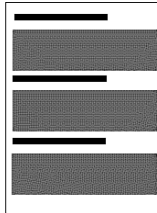
to read your text, you'll need to make it easy for them. You learned how to use style sheets to create margins in Lesson 5, "Adding Your Own Style."

- *Break up long sections of text with bullets, tables, and headings* Information design research has shown that online readers scan text, rather than read it, until they find what they're looking for. Bullets and headings help users find things more quickly.
- *Don't underline any text unless it is a hyperlink* Online viewers expect anything underlined to be clickable. If you use underlining for another purpose, such as formatting your headings, you will confuse your readers.

A plain text document can be boring and intimidating.



Headings break up the text, but it's still boring.



Adding bullets and a paragraph separator helps add diversity and interest to your page.

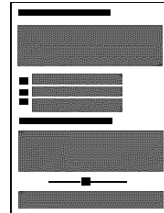


FIGURE 13.1 Adding diversity to your page layout helps enhance its readability.

If your Web site contains more than one page, you'll want to include some way for your visitors to find the other pages in your site. A good navigation system is more than a table of contents; it is a defined structure that gives your visitors information about your site. Your navigation system can consist of text links or image links (refer to Lesson 8, "Using Graphics"). Whichever link type you choose, your navigation system should appear on every page of your site to help orient your users.



Tip Many designers use *frames* as a navigation tool. A frame is a portion of your HTML document that displays a separate HTML. You learned how to create frames, and use them effectively, in Lesson 10, “Creating Frames.”

Fonts and Colors

Color is an exciting way to add interest to your Web pages. In addition to the obvious splash of color that images provide, you can add color to your fonts and page backgrounds. Be creative in your choices, but use a critical eye to review the results. Some colors are very difficult to read online and some color combinations are nearly impossible to decipher. Always provide some contrast in your color choice: use a light-colored font on a dark background and a dark-colored font on a light background.



Tip In HTML, some colors are defined by name (such as navy, red, and black), others by a hexadecimal number. The six-digit number represents the amount of RGB (red, green, and blue) in the color. Lynda Weinmann, a well known graphic designer, has created a couple of color charts specifically for online use: <http://www.lynda.com/hex.html>.

So, how do you add color? With style sheet properties, of course. HTML does have a `` tag that enables you to specify a font (such as Arial or Times Roman) and colors and sizes, but according to the W3C, users are not supposed to use it. Instead, they’ve given you the font-family, font-size, color, and background properties for your style sheets. The following code provides an example of how you can specify your fonts for the `<body>` and `<h1>` tags.



Caution Just because you can specify a font doesn't mean that your visitor will have that font on his or her computer. To be on the safe side, always specify at least one alternate font, as I did in the following example. All but the most basic computers will have Arial and Times New Roman, so it's not a bad idea to use one of those two as your alternate font. Appendix B, "Style Sheet Quick Reference," contains a list of Web-safe fonts.

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title>Fonts and Colors</title>
<style type="text/css">
body {font-family:Trebuchet MS, Arial;
    color:navy;
    font-size:12;
    background:white}
h1 {font-family:Bookman Old Style, Times New Roman;
    color:white;
    font-size:14;
    background:navy}
</style>
</head>
<body>
<h1>Fonts and Colors</h1>
<p>This text is navy on a white background, but the heading
    above is white on a navy background.</p>
</body>
</html>
```

By changing the values in the style properties, you change the results you see in the browser. Look at Figure 13.2 to see how the following changes affect what you see. By not adding a separate font-color and background property to the <h1> tag, the properties assigned in the <body> tag continue.

```

<style type="text/css">
body {font-family:Trebuchet MS;
      color:black;
      font-size:12;
      background:#FFFF80}
h1    {font-family:Bookman Old Style;
      font-size:14}
</style>

```

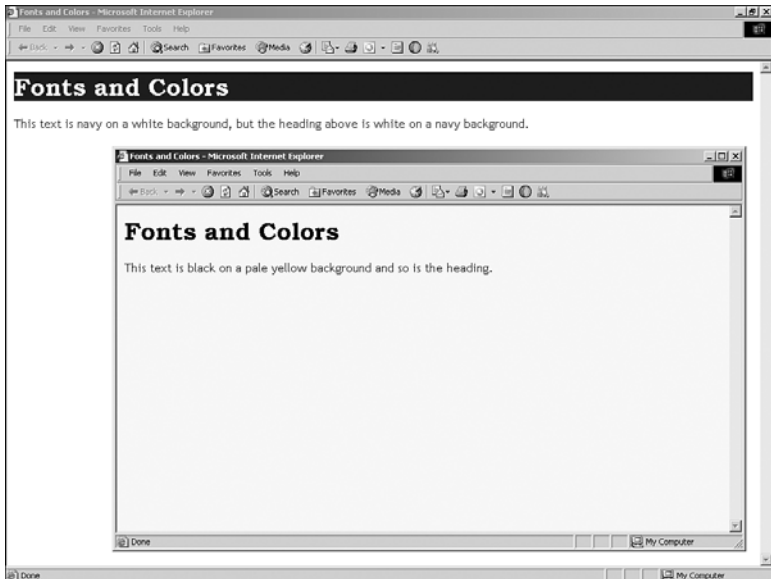


FIGURE 13.2 The background property sets the background color of the entire tag, so using the property on the <body> tag sets the color for the entire page.



Tip Don't get carried away with your font selections. A good rule of thumb is to use no more than three different fonts on each page: one font for the headings, one for the body text, and one for any special text, such as captions and pull-quotes.

Images

Like the other design elements discussed in this lesson, you should use images sparingly when they support the theme you've already established. In Lesson 8, you learned how to add images to your Web pages and use HTML and style sheet properties to align them with your text. Figure 13.3 shows the difference that balance and diversity make to your overall Web page layout.

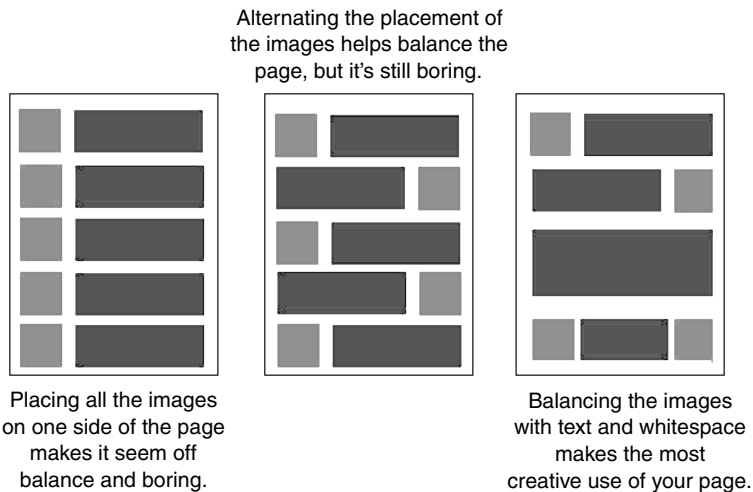


FIGURE 13.3 Adding diversity to your graphical layout helps add interest.



Caution Whenever you are working with graphics on a Web page, you need to be mindful of the overall size of the page. Most people will visit your Web site using a slow modem connection and might not be willing to wait for your page to finish loading. When you open your page using a modem, it should take no longer than five seconds to load. If your pages take much longer to load, you might try to reduce the image size, add thumbnails, or include some type of warning as to the fact that the page will take longer to load.

Background Images

Earlier, you learned how to add background color to your pages, but sometimes you'll want to add an image to the background of your page. The most prolific example of a background image is the page border (see Figure 13.4).

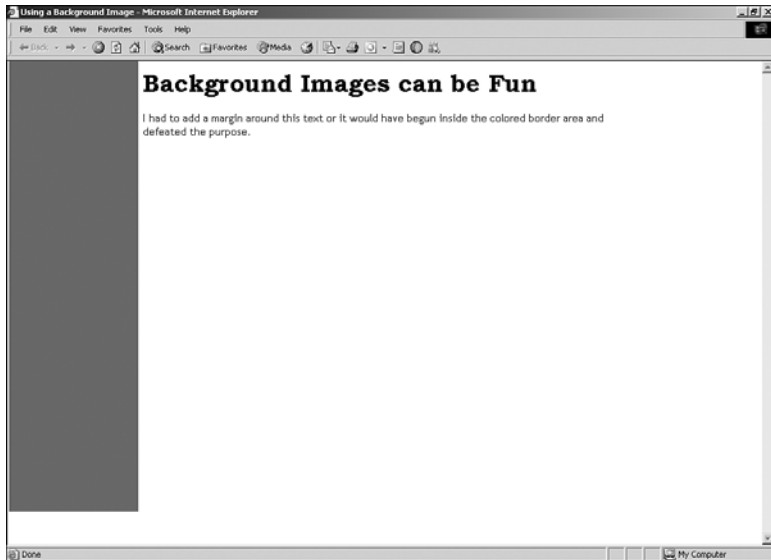


FIGURE 13.4 Use the background-image and margin style sheet properties to create a colored border on your Web page.

Here's how the source code looks for that page:

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
<title>Using a Background Image</title>
<style type="text/css">
body {font-family:Trebuchet MS, Arial;
      color:black;
```

```

        font-size:14;
        background-image:url(images\background.gif);
        background-repeat:no repeat);
        background-position:left top;
        margin:10px,160px}
h1 {font-family:Bookman Old Style, Times New Roman;
    color:#A00068;
    font-size:18;
    background:white}
</style>
</head>
<body>
<h1>Background Images can be Fun</h1>
<p>I had to add a margin around this text or it would have
    begun inside the colored border area and defeated
    the purpose.</p>
</body>
</html>

```

The background.gif image is the colored border background. It is a GIF file, which makes it small (only 5KB) so that it will load quickly. I made it in Paint Shop Pro by drawing a rectangle down the side of the page, as shown in Figure 13.5.

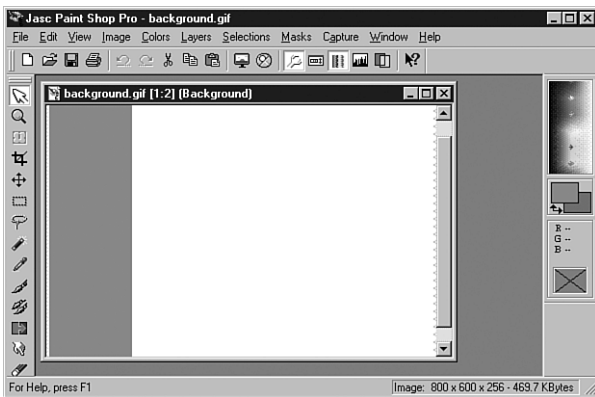


FIGURE 13.5 The background.gif file was created in Paint Shop Pro.

You've already seen how you can use style sheet properties to set the fonts and colors of the page, but look at the previous HTML source code

for Figure 13.4 again to see that we've added four new style sheet properties for you to learn:

- **Background-image:url (*URL of file*)** The property tells the browser where to find the background image you want to use on your page. It must be used as part of the body style.
- **Background-position** This property tells the browser where to place the background image. This is assumed to be the top left (or left top), but you can specify any combination of the following vertical values: top, bottom, or center, and these horizontal values: left, right, or center.
- **Background-repeat** This property determines whether or not the background image will repeat in a tiled manner to fill the entire browser window or appear on the background just once.
- **Margin** You can specify the margin property in inches (in), centimeters (cm), ems (em), points (pt), or pixels (px). If no unit of measure is specified, the pixel unit is assumed. You can set the top, right, bottom, and left margins. I only set two of the margins for my background (0, 160). The browser knows that I wanted top=0px, right=160px, bottom=0px, and left=160px. The browser copied the first two values and applied the same values to the last two options. If I had entered only one value, the browser would have applied the same value to all four options.

If I hadn't set the margin property, the text on my page would have overlapped my image, as shown in Figure 13.6.



Tip Don't forget to check out Lesson 5 if you want to know more about HTML style sheets.

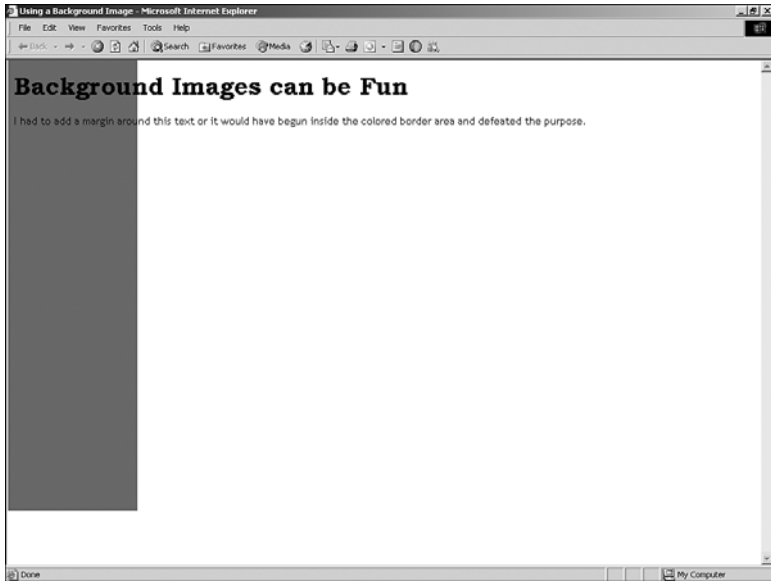


FIGURE 13.6 Without the `margin` property, text starts at the left edge of the page.

In this lesson, you've learned:

- The layout, images, navigation buttons, bullets, lines, colors, and even the fonts that you choose should support the overall design theme of your site.
- Create interest in your Web pages by alternating the alignment of text and images, and by adding bulleted lists and tables.
- Just because you can specify a font doesn't mean that your visitor will have that font on his or her computer. To be on the safe side, always use Arial or Times New Roman as your alternate font.



LESSON 14

Creating Active Web Pages

In this lesson, you will learn about some of the advanced scripting tools that can enhance your Web pages. You'll also find some resources to help you learn more.

What Are Active Web Pages?

Normal HTML pages—everything you've created so far—are considered to be *static*. The page you create is the page that your visitors see, and (assuming that the pages are created without browser-specific code) all your visitors see the same thing.

Scripting languages and DHTML provide the capability to make any HTML element respond to user events, such as mouse clicks. This capability can be used for something as simple as displaying a menu of navigational choices when a word is clicked, to something as complex as a Web application, like online ordering.

Don't be scared off by the word *application*. There is some programming knowledge that is required to implement these elements into your Web pages, but not much. Unfortunately, I can't teach you everything there is to know about programming, but I can tell you where to find additional information.

The most popular ways of including active elements in your Web pages are described in the sections that follow. Table 14.1 gives a quick overview of the information in this lesson.

TABLE 14.1 Scripting and Programming

Technique	Comments
ActiveX	ActiveX works natively on Microsoft Internet Explorer and on Netscape with a plug-in.

Technique	Comments
DHTML	Microsoft and Netscape disagree on how to implement DHTML. You might end up creating two sets of code to make everyone happy.
Java	Platform independent and can be used to create complex applications.
JavaScript	The best of everything. Works on all the major browsers, can be called from a plain HTML page, and you can find plenty of examples to copy into your own Web pages.
VBScript	An easy to learn scripting language, VBScript is the most commonly used language in creating Active Server Pages.

DHTML

DHTML is an acronym for *Dynamic HTML*. DHTML combines all the elements you've already learned (HTML, style sheets, and scripting) to create Web pages that are interactive and easy to update. Unfortunately, Microsoft, Netscape, and the World Wide Web Consortium (W3C) all disagree on how to accomplish this feat. The W3C doesn't even list the acronym on its Web site when discussing HTML standards.



Tip See what Microsoft has to say about DHTML at <http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/dhtml.asp>.

You can find Netscape's description at <http://archive.devx.com/dhtml/articles/sl011701/sl011701.asp>.

Microsoft and Netscape do agree that DHTML should enable you to alter the appearance of a Web page after it has been loaded in the browser. They also agree that DHTML should enable developers to position any HTML element on a page. The elements can even be positioned in the same location so that, in effect, the elements appear on top of each other, but that's where the agreement ends.

Microsoft and Netscape have each developed their own browser-specific codes to achieve this type of interactivity. Using Microsoft's coding standards means that Netscape viewers might not be able to see the dynamic elements. The same is true if you use Netscape's coding standards. This diversity means that developers are forced to choose either to ignore a whole subset of their users, or to double-code all their pages to ensure that doesn't happen.

The most popular use of DHTML on the Internet is in navigational menus. You've seen it in action even if you didn't recognize it. When you hover over a menu name and a list of submenus appears, the menu was probably generated using DHTML. One of my favorite examples (shown in Figure 14.1) includes images as well as text links in the submenus.



Tip You can find plenty of information and coding examples online at <http://www.DynamicDrive.com>.



FIGURE 14.1 This Web site uses style sheets, absolute positioning, and JavaScript to dynamically alter the page.

Java and ActiveX

Java and ActiveX are both programming languages used to create Web applications. Java (created by Sun Microsystems) is platform independent (meaning that PCs, Macs, and UNIX systems can interpret the commands in the application). ActiveX (created by Microsoft), however, only works with the Internet Explorer browser. It is nearly impossible to guarantee that your visitors will use this browser unless you employ the annoying tactic that so many developers have chosen: preceding any application with a note warning your visitors that they must download the *correct* browser before they can view your pages.

How Do They Work?

Both Java and ActiveX work under the principle of object-oriented programming. The idea is that each piece of code should be treated as a separate entity, which can be used repeatedly in many types of environments, including the Web.

Both elements can be embedded in your Web pages using HTML's `<object>` tag. An `<object>`, in HTML, can be an image, an application, or another HTML document. The attributes are the important distinction. The first example (which follows) would be used to include a Java applet. The second example would be used to include an ActiveX control (or application).

```
<object codetype="application/java-archive"
      codebase="http://www.myWeb.com/apps/"
      classid="java:my.program.start">
</object>

<object codebase="http://www.myWeb.com/apps/"
      data="my.activex.program">
      classid="CLSID:613C8CCE-1FF8-41CF-A3DB-052336C14002"
</object>
```

In the first example, the `classid` attribute is the name of the Java applet being called by the `<object>` tag. This same information appears in the `data` attribute for ActiveX programs. In both examples, the `codebase` attribute indicates the directory in which the application can be found. However, the `codebase` attribute itself is not necessary. The entire URL

(including the base directory information) could be included in the `classid` (for Java) and `data` (for ActiveX) attributes rather than including the separate `codebase` attribute.

The ActiveX `classid` attribute deserves some explanation. Other than telling you that the string of letters and numbers actually represents a URL, the best help I can give is to inform you that any ActiveX control that you choose to use in your Web page includes the appropriate `classid` information so that you can copy it into your tag.



Tip Find information about Java and download some fun Java applets at

<http://java.sun.com/applets/index.html>.

You can download ActiveX controls at

<http://www.download.com/2001-2206-0.html>.

JavaScript and VBScript

Scripting is another type of programming, but it's easier to learn, which is a plus. Scripts can be added to an HTML document using the `<script>` tag. The tag can appear within the `<head>` or `<body>` of the document.

A script might be contained in a separate document that is called by the `<script>` tag (much as a linked style sheet is a separate document called by the `<style>` tag). A script might also be contained within the `<script>` tags in the HTML document itself. The decision is yours, based on how often you plan to use the script. If the script appears in only one page, incorporate the script into the document, as in the first of the following HTML samples. If the script appears on more than one page, make it a separate file so that you don't have to duplicate it, as in the second HTML sample.

```
<script type="text/vbscript">enter your script here.</script>
<script type="text/javascript"
    src="http://www.myweb.com/scripts/myscript.jss">
</script>
```

Although the `src` (source) attribute is only required when the script is contained in a separate file, the `type` attribute is always required. This attribute tells the browser which language the script is written in: `text/javascript`, `text/vbscript`, or `text/tcl`. If you are using the same scripting language throughout your HTML document, you can include a `<meta>` tag that defines the default script type for the entire document. The `<meta>` tag (as you learned in Lesson 3, “Adding Text and More”) is placed inside the `<head>` tag and gives the browser information about the document.

```
<meta http-equiv="Content-Script-Type" content="type">
```

What Can Scripting Do?

The easy answer to this question is anything. If you look at some of the script collections on the Web, you find that people are using script for all kinds of things—including adding table values, creating rollover effects, and even games.



Tip WebDeveloper has more than 7,000 downloadable JavaScript samples at <http://webdeveloper.com/javascript/>.

It is possible to associate a script with a certain event that occurs when the page appears on the browser. Figure 14.2 is an HTML page with some very simple JavaScript code that changes the background color of the page with the press of a button.

You might wonder where the code is because the `<script>` tag is empty. The code, in this case, is embedded in the `<input>` tag with the `onClick` command. The tag responds to each of the events shown in Table 14.2 and a few more.



FIGURE 14.2 Simple JavaScript code to change the background color with the onclick command.

TABLE 14.2 Script Calls

Event	With Tags	The Script Runs When...
onload	<body>, <frameset>	The document opens.
onunload	<body>, <frameset>	The document closes.
onclick	Anything	The mouse is clicked over a particular item (button, image, and so on).
ondblclick	Anything	The mouse is double-clicked over a particular item.
onmouseover	Anything	The mouse is moved onto an item.
onmouseout	Anything	The mouse moves away from an item.
onmousemove	Anything	The mouse is moved while on an item.
onsubmit	Submit button	The form is submitted.
onreset	Reset button	The form is reset.



Tip The event handlers (script calls) that tags support differ between browsers. Netscape supports many fewer event handlers than Internet Explorer does. Use Web Developer Journal's compatibility table (http://webdevelopersjournal.com/articles/javascript_limitations.html) to choose the right handler for your pages.

The scope of this book does not allow for coverage of all these topics in any depth, but I hope you have some idea of the possibilities and can take the time to learn more on your own. Table 14.3 lists the HTML tags that were discussed in this lesson.

TABLE 14.3 HTML Tags Used in This Lesson

HTML Tag	Closing	Description of Use
<object>	</object>	Embeds an object, such as an application, into a Web page.
<script>	</script>	Embeds a script into a Web page.

In this lesson, you've learned:

- How to add Java and ActiveX applications to your HTML documents with the <object> tag.
- How to add JavaScript and VBScript to your HTML documents with the <script> tag.
- That DHTML combines all HTML, style sheets, and scripting to create Web pages that are interactive and easy to update.



LESSON 15

Using Web Authoring Tools

In this lesson, you will learn where to find some of the most popular Web page authoring tools and how you can use them to create your Web site.

Why Use a Tool?

You've just spent the last 14 lessons learning how to create Web pages by yourself, so why would you want to use a tool? The biggest reasons are time and ease of use. The Web is a very visual medium and staring at HTML code in a text editor is not very visually stimulating. It's easy to forget to be creative when you are concentrating so hard on making sure that you're using the correct HTML tags and putting them in the proper place in the document.

Web page authoring tools come with enough bells and whistles to get your Web site started in no time, and feature excellent site management and reporting tools. There are differences, though. FrontPage, for example, comes with a set of design themes that you can apply to your own pages to give your site a professional look. Dreamweaver has extensive support for style sheets. You can easily create style sheet declarations and apply them to the other pages in your Web site.

FrontPage and Dreamweaver are consistently voted best of the best and you might find that one of them could help you, too. In the following sections, I want to introduce you to some of the most interesting features of both products.

Microsoft FrontPage

FrontPage contains a little bit of everything. It has site management features, pre-designed Web themes, starter Webs that are ready for your content, and advanced features (such as navigation buttons and search bots); all with a familiar interface that feels like a combination of Microsoft Word and the Windows Explorer. Figure 15.1 shows a sample page in FrontPage.

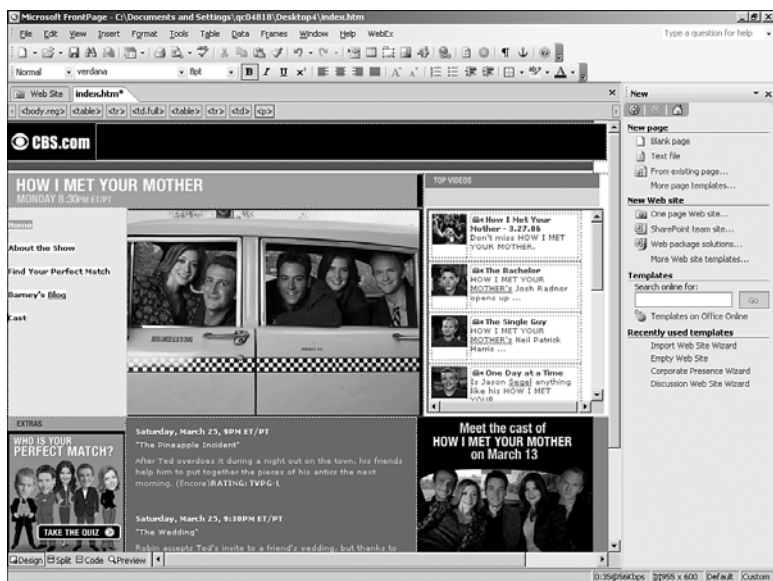


FIGURE 15.1 A sample page in FrontPage.

One of the best features FrontPage has going for it, aside from the fact that the interface is so familiar to most computer users, is that it includes a variety of Web wizards that you can use to define the type of site and the type of pages you want to have. Select a design theme and the program creates all your pages and adds Navigation bars with the hyperlinks already in place. You just add the basic content in the middle of the page.

The Navigation view of FrontPage is exciting as well. You can add, remove, and rearrange the pages in your Web site and FrontPage automatically updates the Navigation bar. (See Figure 15.2.)

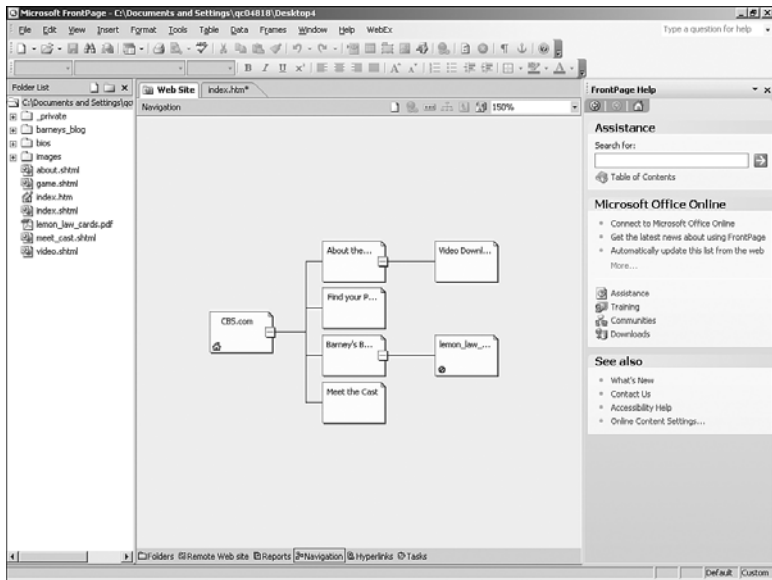


FIGURE 15.2 The Navigation view in FrontPage. You can add, remove, and rearrange pages in this view and the Navigation bar is automatically updated.

FrontPage is available with the Professional Special Edition version of Office; it can also be purchased separately. It integrates well with the other Microsoft Office software packages, such as Word, Excel, and Access. You can even create a Web page in Microsoft Word, save it as a Web page in your FrontPage Web site, and apply the FrontPage Web theme to the finished product without losing your original formatting.

FrontPage supports basic database interactivity well. It can create and update an Access database from a form. In addition, extensive toolbars, menu choices, and shortcut menus make it easy to add your content without cluttering up the editing window. The tabs at the bottom of the

FrontPage window enable you to see the actual HTML source code for your page or a split version of the screen so that you can see both the design and code view. (See Figure 15.3.)

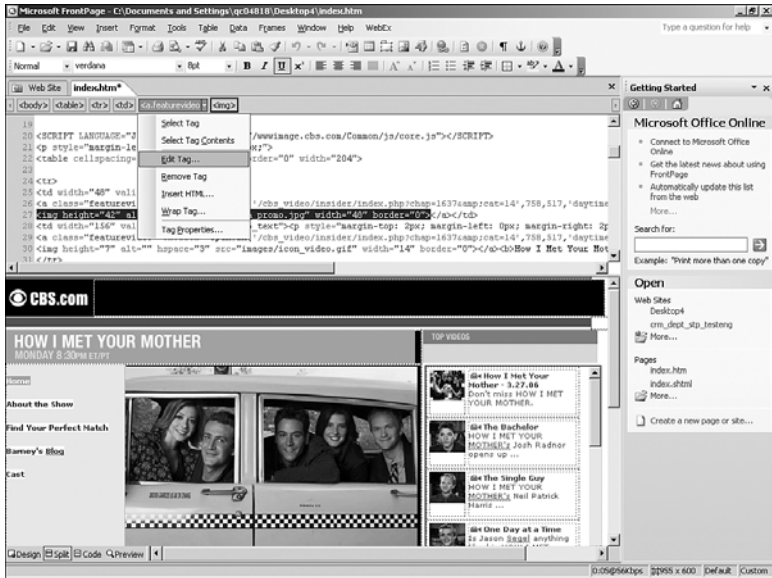


FIGURE 15.3 The Split view of FrontPage.

FrontPage has another feature that you might enjoy using on your Web sites. It includes the capability to add content from some of Microsoft's most popular Web sites (such as Expedia, MSNBC, and bCentral). In Figure 15.4, for instance, you can see that the MSNBC component includes headlines from the News, Living and Travel, Business, and three other sections.



Tip You can find out more about FrontPage at Microsoft's FrontPage Web site (<http://office.microsoft.com/en-us/FX010858021033.aspx>).

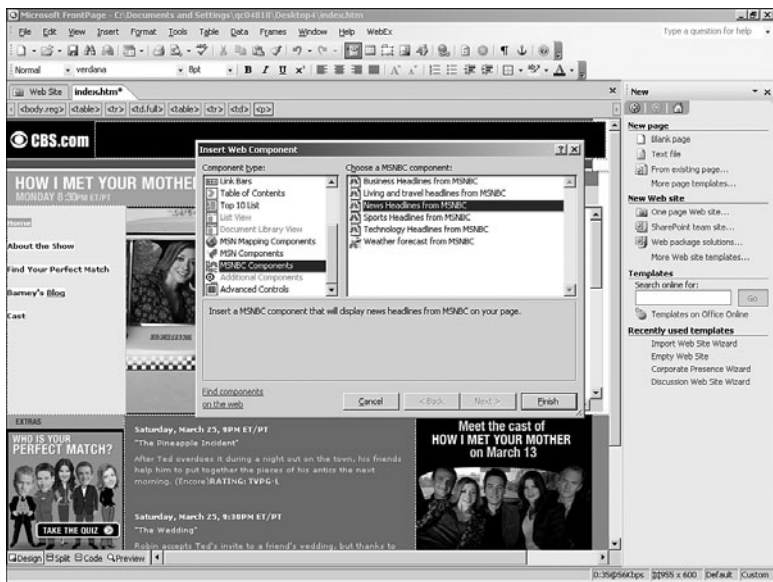


FIGURE 15.4 Inserting a FrontPage Web component.

FrontPage does provide some fairly advanced page components that include an ad banner manager, hit counter, search bot, hover buttons, and scheduled image and page substitutions. Most of these features, however, require FrontPage server extensions that you can download free from Microsoft's site.



Caution Be aware that if you create a Web site in FrontPage and want to publish it to the World Wide Web, you need to find a Web host that supports these server extensions. You'll learn more finding Web hosts in Lesson 16, "Making a Name for Yourself."

Macromedia Dreamweaver

Dreamweaver, too, comes packaged with an impressive array of page designs and layout templates to help you get started, although it does not feature a "looks like Microsoft Word" interface. (See Figure 15.5.) Instead,

the interface includes a variety of toolbars, called *panels*, but those same panels offer a variety of ways to enhance the pages that you create.

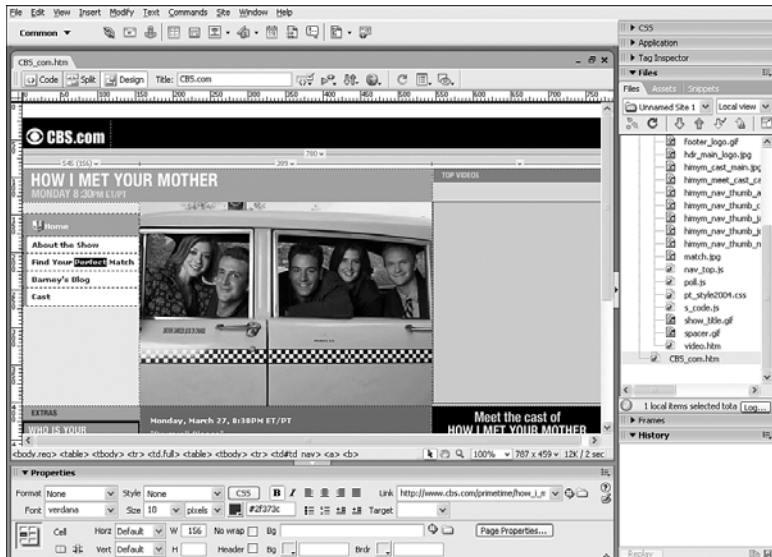


FIGURE 15.5 A sample page in Dreamweaver.

Once you've designed a page to your specifications, you can create a template with editable and non-editable regions, which you then can use to build other pages on your site. What's more, if you change your template somewhere down the line, Dreamweaver automatically updates any page created with the template—a handy shortcut to ensure that the appearance of your site remains intact. You can use Dreamweaver's CSS Styles panel to create style sheet declarations by example and store those styles for recall in other pages. (See Figure 15.6.)

The Properties Inspector, at the bottom of the Dreamweaver workspace, enables you to review existing properties or set new properties for any page element (text, tables, images). Available properties are alignment, color, anchor tags, etc. You set the properties and Dreamweaver writes the HTML/CSS code for you.

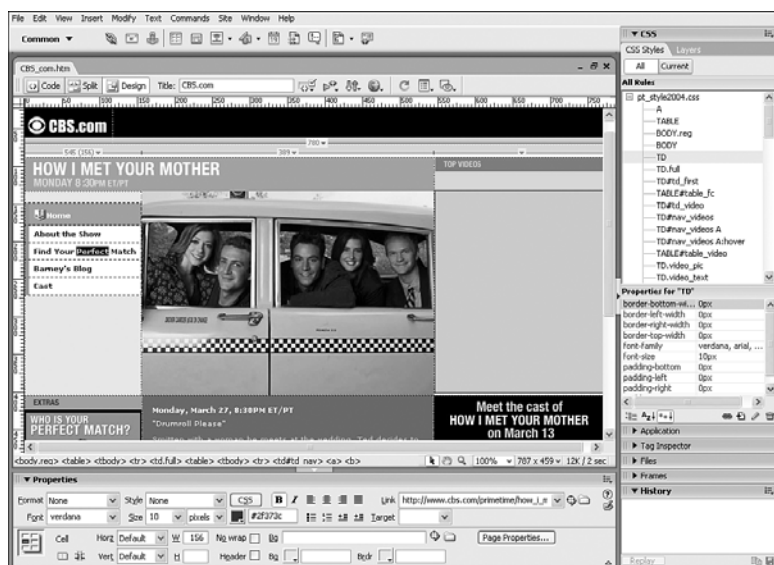


FIGURE 15.6 The CSS Styles panel easily modifies styles. The Properties Inspector displays the properties of any selected tag.

Two of my favorite features are the Insert toolbar and Dreamweaver's new Panel groups, which appear to the right of your workspace. The panels are completely customizable, allowing you to display or hide those panels that are most frequently used by you.

- The Insert toolbar contains buttons for creating the most commonly used objects (such as images, tables, and layers).
- The CSS Styles panel contains details about the properties associated with a selected style.
- The Application panel allows you to configure databases, set up links to your remote site, and create ColdFusion components.
- The Tag Inspector panel shows you all of the attributes associated with the selected tags.
- The Files panel is similar to the Windows Explorer because it contains all of the files and folders in your Web site.

- The Frames panel visually displays the frames in your frameset helping keep you organized.
- The History panel keeps track of every action, including arrow movements, copy/paste, and text insertion that takes place on your Web page.

Dreamweaver excels at the advanced features, such as layering and XML compliance. Even though most browsers can't agree on how to support these features, Dreamweaver can check your page for compatibility with several versions of both Internet Explorer and Netscape Navigator, and verify all your internal links.

Unlike FrontPage, Dreamweaver doesn't require special server extensions or add extraneous code to your pages, which means that you are able to use your pages with any Web host.

Other Popular Web Tools

FrontPage and Dreamweaver aren't the only Web authoring and site management tools available. Several more appear in the following list:

- **Adobe GoLive** <http://www.adobe.com/products/golive/main.html>
- **NetObjects Fusion** <http://www.netobjects.com/>
- **BEdit** <http://www.barebones.com/products/bbedit/index.shtml>

If you don't like the idea of a WYSIWYG tool and you prefer to continue working directly with the HTML source code, you probably will like HomeSite. With HomeSite, you can drag and drop HTML code, insert links, and modify existing tags. You can also search and replace HTML code, and when you're confused about which attributes apply to which tags, HomeSite can offer suggestions. All this makes HomeSite the choice of many professional developers. You can read more about HomeSite at <http://www.macromedia.com/software/homesite/>.

In this lesson, you've learned:

- Microsoft FrontPage is perfect for beginners. It looks like Microsoft Word and comes with many preformatted designs.
- Macromedia's Dreamweaver has advanced features, such as style sheet controls and layering, that make it perfect for professionals.
- Both tools enable you to create Web pages without knowing the HTML tags that you're learning in this book.

LESSON 16

Making a Name for Yourself



In this lesson, you will learn where to find a Web host to publish your Web site and tips for making sure your site is found.

Web Hosting

When you finally finish creating your Web pages, you're going to want to put them on the Internet and make sure they're found. Unless you plan to set up your own Web server, you'll probably be looking for a *Web hosting* service.



Web Host A company that provides space on its Web servers to store your Web files.

Web hosting services offer help in a variety of ways, and at a variety of costs. Some Web hosts offer design services, customizable scripts, visitor logs, database support, and more (in addition to the standard disk space). Use the information in Table 16.1 to find a Web host that meets your needs.

TABLE 16.1 Web Hosting Resources

Host Name	Comments	URL
The List	The official list of Internet service providers	http://www.thelist.com/
Web Hosting Buying Guide	CNET describes popular features offered by Web hosts so that you can decide which host is best for you	http://reviews.cnet.com/Web_hosting_buying_guide/4520-6540_7-5138854-1.html?tag=dir
Yahoo! Geocities	Free Web hosting with authoring resources	http://geocities.yahoo.com/home/
Tripod	Free Web hosting	http://www.tripod.lycos.com/
FreeWebs	Free Web hosting	http://members.freewebs.com/
Register FrontPage Hosts	Web hosts that support Microsoft FrontPage, and who have registered with Microsoft	http://www.microsoft.com/office/frontpage/prodinfo/partner/wpp.asp

Search Pages and Indexes

After your HTML documents are up and running on a Web server, you need to make sure that people can find them. Because most people look to *search engines* when they want to find something on the Internet, we'll start there.



Search Engines Searchable indexes of Web resources. Some search engines (called *indexes*) also categorize information enabling people to search by categories and keywords.

Two types of search engines exist on the Web: spiders and indexes.

- A *spider* (also called a Web crawler, or *bot*, which is short for robot) is an automated script that crawls through Web pages following hyperlinks to find related pages, and then builds a database with the contents of all the pages it visits. Google.com is an example of spider technology.
- A *search directory* is a categorized list of sites on the Internet. The search directory's administrative personnel review the content of user-submitted Web sites and populate the categories. The Yahoo! directory is the best known example of this type of technology.

Search Bots

Search engine bots (also called robots, spiders, and crawlers) search through all Web pages and then index them according to the information they find. You can help the indexing portion be more accurate by using <meta> tags. Without <meta> tags, these bots treat every word in a document exactly alike. If you add keywords and descriptions to your documents, you increase the possibility that your Web pages will be found. You learned how to do this in Lesson 3, "Adding Text and More," but let's try a quick refresher. The following example shows the correct format for adding the <meta> tag to your documents. Figure 16.1 is an actual example of the <meta> tags used on the WebReference.com site.

```
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xml:lang="en" lang="en">
<head>
<title>Your HTML Page</title>
<meta name="keywords" contents="keywords that
  people might use to search
  for your page.">
<meta name="description" contents="a brief
  paragraph describing your
  document.">
<meta name="author" contents="your name">
```



```

</head>
<style type="text/css">
</style>
<body>
    insert your document here.
</body>
</html>

```

```

www.microsoft[1] - Notepad
File Edit Format Help
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html dir="ltr" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" >
<!--TOOLBAR_EXEMPT-->
<meta http-equiv="PRCS-Label" content="(PRCS-1.1 &quot;http://www.rsac.org/ratingsvol.htm&quot;)" >
<meta name="KEYWORDS" content="products; headlines; downloads; news; web site; what's new; solutions" >
<meta name="DESCRIPTION" content="The entry page to Microsoft's web site. Find software, solutions, and more." >
<meta name="MS.LOCALE" content="en-US" >
<meta name="CATEGORY" content="home page" >
<title>Microsoft Corporation</title>
<base href="http://g.msn.com/mh_mshp/98765" >
<style type="text/css" media="all">
@import "http://i.microsoft.com/h/en-us/r/hp.css";
</style>
<script type="text/javascript" src="http://i2.microsoft.com/h/all/s/hp.js"></script>
<script type="text/javascript" src="http://i1.microsoft.com/h/en-us/r/SiteRecruit_PageConfiguration"></script>
</head>
<body>
<script type="text/javascript">
<!--
var isw; isw=(document&&document.body.clientWidth&&document.body.clientWidth)>895&&document.getElementById("mainPage")>0;
</script>
<a href="http://www.microsoft.com/default.aspx#cArea" class="hide">Click here to jump to main page</a>
<div id="dpage" class="page">
<table cellpadding="0" width="100%"><tr><td colspan="2">
<table cellpadding="0" width="100%" style="height: 22px"><tr>
<td width="30%" style="filter: progid:DXImageTransform.Microsoft.Gradient(startColorStr='#4B92D9', endColorStr='#CEDFF6');">
</td><td width="70%" style="filter: progid:DXImageTransform.Microsoft.Gradient(startColorStr='#CEDFF6', endColorStr='#4B92D9');">
</td></tr></table>
<div id="msvGlobalToolbar" height="22" nowrap align="left">
<table cellpadding="0"><tr>
<td class="gt0" nowrap onmouseover="this.className='gt1'" onmouseout="this.className='gt0'"><p><a href="http://www.microsoft.com/default.aspx#cArea" class="hide">Click here to jump to main page</a></p></td>
<td class="gt0" nowrap onmouseover="this.className='gt1'" onmouseout="this.className='gt0'"><p><a href="http://www.microsoft.com/default.aspx#cArea" class="hide">Click here to jump to main page</a></p></td>
</tr></table>
<table cellpadding="0" width="100%" bgcolor="#FFFFFF"><tr valign="top">
<td colspan="2"><script type="text/javascript">rtt</script></td></tr></table>
</div>
</body>

```

FIGURE 16.1 The HTML source code for <http://www.microsoft.com> with the `<meta>` tags highlighted.



Tip If you don't add any other `<meta>` tags, be sure to add the description. Not all search engines rank pages in the same manner, but they all display a description of the pages found. The right description can lure visitors to your site.

Adding Your Web Site to the Search Engine

All search engines enable Web authors to add the URL of their own Web site to the search engine. Most of them do this with some type of online form. A link to Google's Advertising Programs page (http://www.google.com/submit_content.html) appears at the bottom of every Google page. (See Figure 16.2.) As you can see, Google offers several ways to advertise your Web site through its free index, or you can submit your site to specific programs for a fee. Just choose the right option for you.



FIGURE 16.2 Google's Advertising page enables you to add your Web page to the Google index.

Yahoo!'s Suggest a Site page (<http://search.yahoo.com/info/submit.html>) offer similar flexibility (see Figure 16.3).



Tip Some Web sites offer to add your URL to many (if not all) search sites with one form. Companies such as Add Me (<http://www.addme.com/submission.htm>), for instance, charge a fee for this convenience. However, you probably want to add your own site information to the most popular search sites (Yahoo!, Excite, AltaVista, Lycos, LookSmart, and Go.com) to assure yourself the best chance of being found.

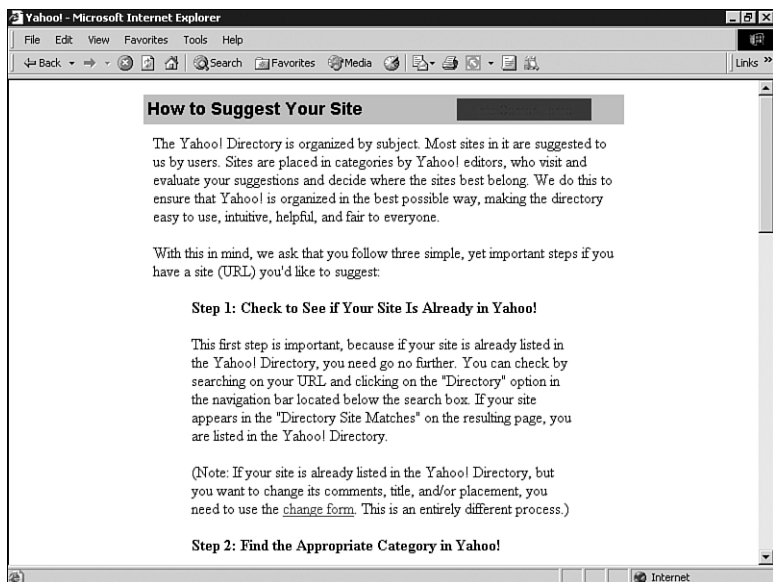


FIGURE 16.3 Yahoo!'s Suggest a Site page enables provides option for submitting your Web site.

Advertising

Don't forget that you can advertise on the Web, too. The following list of Web sites offers some form of advertising or announcement service. The flashy banner ads and annoying pop-ups page that proliferate the Web are there for a reason: People actually click on them. These advertising services can help you create your own ad and place it on pages that relate to your site. The Interactive Advertising Bureau (<http://www.iab.net/standards/index.asp>) maintains a set of standards for many types of Internet advertising options.



Tip Remember to include your URL on your business cards, letterhead, and e-mail auto-signature. Unless you tell people where to look, they can't find you.

In this lesson, you've learned:

- Two types of search engines exist on the Web: spiders and indexes.
- To add keywords for the search engines with <meta> tags. Different search engines search for different <meta> tags, so use several (including keywords, descriptions, and author).
- Most search engines have their own site-submittal forms. Fill out these forms so that the search engine can find your site.



LESSON 17

Planning for the Future

In this lesson, you will learn what's next for the Internet and what you can do now to prepare for the coming changes.

The Future of the Internet

The extraordinary growth of the Internet since the early 1990s has come about chiefly because HTML is so easy to learn. Companies can distribute information to their employees, customers, and business partners quickly and inexpensively. Unfortunately (or fortunately, depending on your point of view), the first blush of Internet and Web development has passed and companies are already beginning to look for new ways to disseminate the information they want to share.

Hearing this cry for help, the World Wide Web Consortium has developed an eXtensible Markup Language (XML) that can be used by Web page authors whose needs extend beyond the capabilities of HTML.



eXtensible Markup Language (XML) The newest language being developed by the World Wide Web Consortium, XML has been described as a language for defining other languages. It is more flexible than HTML.

What Is XML?

To understand XML, you need to step back and remember what HTML is. HTML is a markup language that uses a predefined set of tags to describe a document's structure in terms of paragraphs, headings, and so on. Like HTML, XML describes the structure of the document, but unlike HTML,

XML is flexible enough (or *extensible* enough) to define the same tag name (such as <title>) in several different ways depending on which *Document Type Definition (DTD)* is called.

In addition, XML takes the concept of tagging one step further by enabling developers to create custom tags and attributes. Both markup languages use style sheets to define the format of each tag with color, fonts, and emphasis.



Document Type Definition (DTD) A file defining the set of tags that can be used within a particular file. XHTML uses three DTDs: strict, transitional, and frameset.

The following examples show how a single entry from an address book might be marked up in both HTML and XML, respectively:

- HTML:

```
<p>The White House<br />
1600 Pennsylvania Avenue NW<br />
Washington, DC 20500</p>
```

- XML:

```
<contact>
  <name>The White House</name>
  <address>1600 Pennsylvania Avenue NW</address>
  <city>Washington</city>
  <state>DC</state>
  <zip>20500</zip>
</contact>
```

Why is this difference important? It's important because, in essence, your document becomes a giant database of information.

Suppose that I am the owner of a chain of multiplex theaters and I want to put information on the Web about the movies I'm showing. In traditional Web publishing (if something as young as the World Wide Web can be said to even have a traditional method), I could do one of the following two things:

- Create a series of Web pages that would need to be updated frequently.
- Create a database that held all the information and hire a Java programmer to write an application that would enable people to perform searches on my database to see what was showing in their neighborhood.

With the advent of XML, I have a third option. I can create a single Web page that contains all the information for all my theaters, and then use style sheets and templates to present the right information to the right people.



Tip The W3C and industry experts are creating industry-specific versions of the XML standard. So, you can create your own tags in XML, and you can also take advantage of the fact that others in your industry are using the same standard.

Analyze the Data

The first thing I have to do is analyze my data. What information do I need to share? I probably would want to share the name of the movie, a brief description, the names of the stars in the movie, links to promotional information for the film, the name of the theater in which it's playing, the address of the theater, my phone number, the time the movie is showing, the price of the ticket, whether discounts are available, and a lot more.

After you know the type of data you need to collect, you can create your XML input document. You can see an example of two of these input documents in Figure 17.1. Each data type is represented by a pair of tags (such as `<movies>` and `</movies>`). Related data types are nested within a parent tag. For example, the `<title>` and `<star-male>` tags are related to the `<movies>` tag. Unlike with HTML, I made up my own XML tags based on the information I wanted to present.

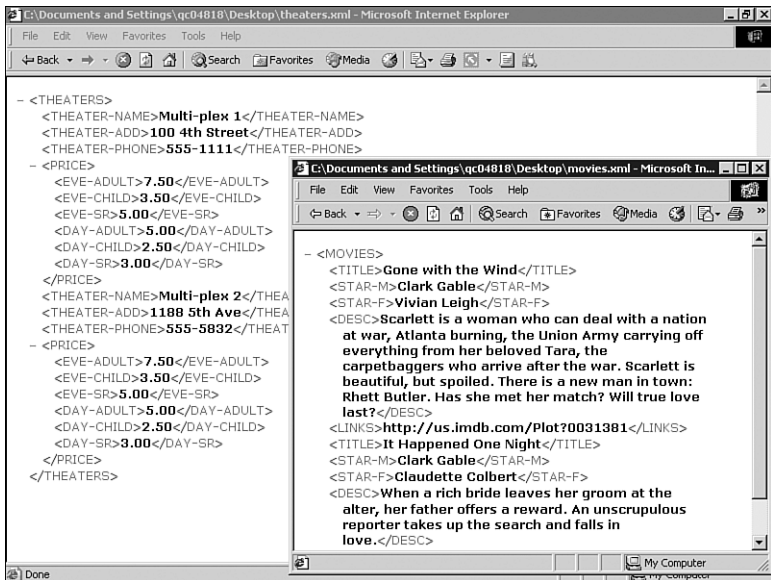


FIGURE 17.1 Without a style sheet, an XML-enabled browser can only render text.



Caution Don't rush out to convert all your HTML documents to XML just yet. Most browsers can't process XML documents yet. However, you can start preparing now by creating XHTML documents. These documents enable you to use HTML and XHTML now, and will be easy to convert to XML in the future.

Create a Style Sheet Template

After you complete the input document, you need to create a style sheet template that determines how you present your information. You learned about style sheets in Lesson 5, "Adding Your Own Style." XML style sheet templates are very similar, but also define the structure of the document (tables, lists, paragraphs, and so on).



Tip You can learn more about XML style sheet templates from the W3C at www.w3.org/TR/xsl/. Another excellent resource for XML information is www.xml.com.

The real fun with XML documents comes from the fact that the content of the page is separated from its format. In the movie theater example, suppose that I own two movie theaters. Multiplex 1 is a downtown art theater. It only shows artsy films attended by serious film students and it likes to promote itself as a dark, almost somber, environment. Multiplex 2 is in a posh part of uptown and shows mostly revivals to an older, more conservative crowd. Now imagine that I'm planning to show the same movie, *Citizen Kane*, at both theaters.

My input document, which holds the content that appears on the Web site for both theaters, includes the following tags for *Citizen Kane*:

```
<title>Citizen Kane</title>
<star-male>Orson Wells</star-male>
<desc>Powerful newspaper owner Charles Foster Kane was many
      things to many people, both in life and, as seen in
      retrospective, in death.</desc>
<links>http://us.imdb.com/Plot?0033467</links>
```

Using XML style sheet templates, I can create two completely different pages for my theaters. For Multiplex 1, the artsy theater, I might choose to have a black background with the title in a dramatic gothic-looking font and the other elements (<star-male>, <desc>, and <links>) placed in a bulleted list below. For Multiplex 2, the revival theater, I might create a background image of a film canister for my page. Then, I might choose to place all the elements of the movie into a horizontal table for a more conservative feel.

I can do that because style sheet properties reference the element they are defining, not the content of that element. Rather than placing the content (*Citizen Kane*) on the style sheet template, I would place the following tag, which tells your computer to insert the information in the <title> tag.

```
<xsl:value-of select="title" />
```

XML promises to be a platform-independent, software-independent language. Web developers and other programmers will be able to use the same data input documents to present information on the Web, in business automation tools (such as spreadsheets and word processors), and even on paper. That can save us all a lot of time and money.

Being Prepared

More and more, computer application developers are choosing to create their applications using Web technology. Whereas just 10 years ago, schools were busy teaching their students how to write BASIC programs and type DOS commands at the appropriate prompts, now they are teaching students HTML and learning to browse the Internet is a requirement. Some schools even offer homework help on the Internet. The Internet and Web technology are not going away, and they are going to continue to grow and change.

Already we are seeing the emergence of cell phones, pagers, and other hand-held devices that can display some Internet sites. The release of the XML standard will enable the Internet to become available in any number of new media. That's why it is important to understand what you can do now to make sure that you aren't caught off guard the next time the standard changes.

Check Your Code

Microsoft and Netscape, the two largest competitors in the browser wars, continue to try and outdo each other with new browser features. Both browsers have been known to create new tags that work only on their own browser. If you use those tags when you are creating your Web site, you end up forcing your viewers to choose a browser, or lose important features that you intended to share with them. Don't put them in that position. You can use tools, such as NetMechanic (<http://www.netmechanic.com>), to ensure that your site is the best it can be.

Be sure to test your pages on different browsers and older browser versions. Not everyone uses the newest version of a browser and some older versions do not support as many tags. The Browser Photo feature at the NetMechanic site does this for a small fee. Figure 17.2 shows what you can expect from their service. For each site, Browser Photo tells you which browsers (and at what resolutions) the site was tested.

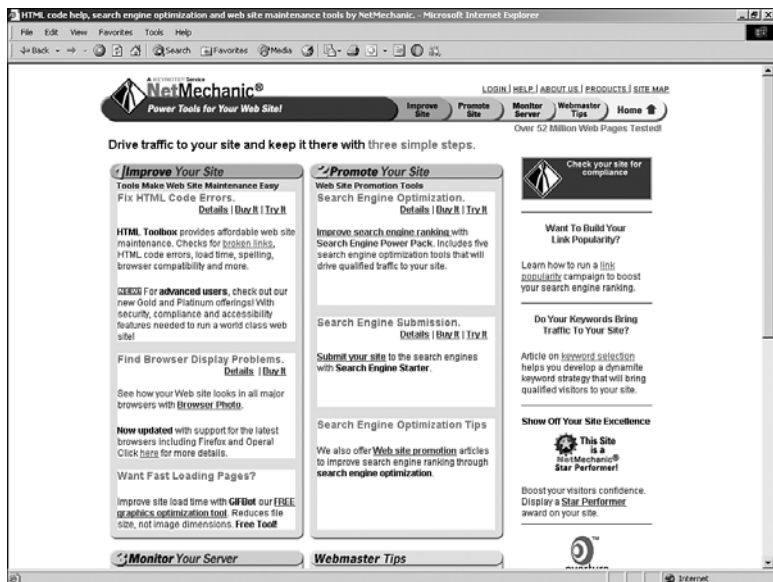


FIGURE 17.2 NetMechanic can test your pages for browser compatibility.

Use Correct Syntax

XHTML must be well-formed; in other words, tags must be nested properly (see the “Nest Tags Properly” section later in this lesson) and tags must be closed. For example, if you forget to close your `` (list item) tag within a `` (unordered or bulleted list), the browser knows that when you add the next `` tag, you want the last one to close. In fact, you want all of your tags to close. In HTML, the following:

```
<ul>
<li>One ring-y, ding-y</li>
<li>Two ring-y, ding-ys</li>
</ul>
```

is the same as this:

```
<ul>
<li>One ring-y, ding-y
<li>Two ring-y, ding-ys
</ul>
```

and the same as this:

```
<UL>
<LI>One ring-y, ding-y</LI>
<LI>Two ring-y, ding-ys</LI>
</UL>
```

and the same as this:

```
<ul>
<li>One ring-y, ding-y</li>
<li>Two ring-y, ding-ys</li>
</ul>
```

With XHTML documents, browsers can differentiate between those examples. Only the first example is well-formed. Learn now to use the proper syntax for your documents and you won't find yourself reworking them later.



Tip Did you notice in the examples that capitalization of the tags makes a difference in XHTML? It's all part of the syntax.

Always Quote Attributes

All tag attributes must be quoted. In the past, you could add attributes, as in the following HTML sample:

```
<img src=/images/trial/gavel.jpg />
```

However, the new XHTML standard (in an effort to prepare us for the transition to XML) requires us to enclose all the attribute specifications in quotes, as in the following HTML sample:

```

```

These are minor differences, sure, but if you get into the habit of doing this correctly from the start, it will save a tremendous amount of rework as the standard is fine-tuned.

Use Style Sheets

In previous versions of HTML, Web page authors controlled the color, format, and layout of their documents with formatting tags (such as `` and `<body bgcolor="color">`). With XHTML, the W3C is recommending that all these format attributes be controlled with style sheets instead.

This book has focused on the XHTML preferences, which might mean that older browsers won't always show what you intend. You can add older HTML tags to your documents without affecting your style sheets, as shown in Figure 17.3. Just remember that the HTML format tags and the style sheet properties cannot conflict, or you will have problems. If your style sheet property tells the browser that the `<body>` tag should have a yellow background, for example, be sure that the `<body>` tag also calls for a yellow background. If you choose conflicting attributes by mistake (as is done in the following example where the style sheet requires the background color to be #FFFF80, but the `<body>` tag requires the background color to be white), the style sheet property takes precedence.

Nest Tags Properly

Because XHTML and XML are more structured than HTML, you should get into the habit of paying attention to the details. You've seen in previous lessons that you can nest one HTML tag inside another. If you want to have text within a table cell (or any tagged element, such as a ``, ``, and so on) to be both bold and italic, remember to close the tags in the opposite order from which they were opened. The following example shows that `` was opened first and closed last.

```
<table>
<tr>
<td>
<b>This text is only bolded. <i>This text is bolded and
italicized.</i></b>
</td>
</tr>
</table>
```

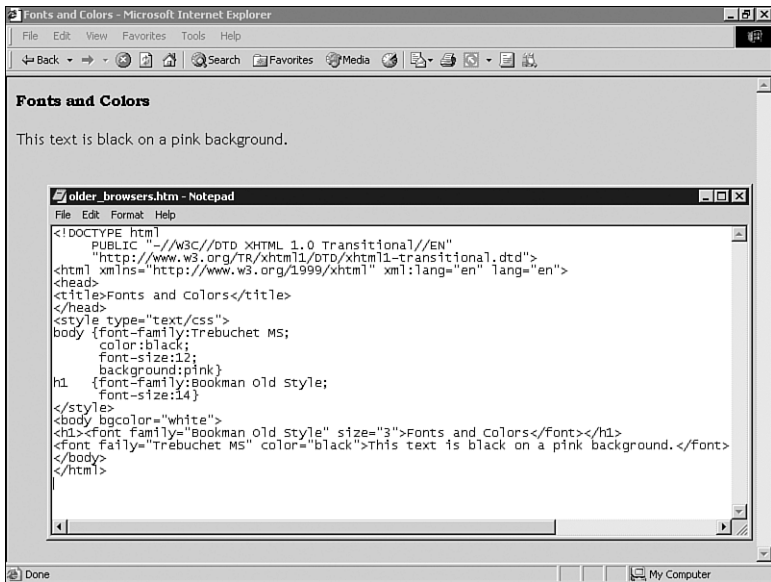


FIGURE 17.3 The HTML document seen earlier now has formatting tags added for older browsers.

As shown above, you might nest tags within a paragraph. In that code, the two sentences are both bold, although only the second is also italicized.

Check It Twice

It's such a simple thing that we often overlook it, but your pages appear more professional and your visitors have more respect for the information you provide, if your content is spelled correctly.

By the same token, don't publish broken links. Nothing is worse than clicking a link that goes nowhere, or leads to the dreaded 404 error. Make sure you verify that all your links go where you want them to go.



Caution Whatever you do, don't forget to verify that your document includes the correct DTD: Strict, Transitional, or Frameset. The document is not XHTML-compliant if it doesn't include the DTD.

Learn All You Can

The Internet is a great place to learn about HTML, XML, and the World Wide Web. Check out some of the following great resources:

- W3C's HTML and XHTML Specifications
www.w3.org/MarkUp/
- W3C's XML 1.0 Recommendation www.w3.org/XML/
- XML, Java, and the Future of the Web
<http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm>
- XML Resource Center www.xml.com
- W3 Schools <http://www.w3schools.com/default.asp>

In this lesson, you've learned:

- XML goes beyond HTML. Rather than just assigning a structure to the text (with paragraphs, headings, tables, and so on), XML adds meaning and order.
- The XML standard isn't complete yet, but it will take over the Internet when it is.
- There are things you can do now to ensure that you are ready for the future of the Internet: use the correct syntax for all tags, use lowercase HTML tags, nest your tags appropriately, and use style sheets rather than the HTML formatting codes.

APPENDIX A

HTML/ XHTML

Quick Reference



HTML and XHTML are markup languages that define the structure, rather than the format, of the elements of your documents. XHTML, is the latest version of that language, is more restrictive than previous versions.



Tip This appendix is based on the information provided in the *XHTML Specification W3C Recommendation* of January 26, 2000 and revised on August 1, 2002. The latest versions of these standards can be found at <http://www.w3.org/TR/html1/>.

To make the information readily accessible, this appendix organizes HTML elements by their function in the following order:

- Required elements
- Text phrases and paragraphs
- Text formatting elements
- Lists
- Links
- Tables
- Frames

- Embedded content
- Style
- Forms
- Scripts

The elements are listed alphabetically within each section, and the following information is presented:

- **Usage** A general description of the element.
- **Attributes** Lists the attributes of the element with a short description of their effect.
- **Notes** Relates any special considerations when using the element.



Caution Several elements and attributes of HTML have been deprecated by the current XHTML specification. They have been outdated and you should avoid using them. Those deprecated elements and attributes have been eliminated in this appendix.

Following this, the common attributes and intrinsic events are summarized.

Required Elements

HTML relies on several elements to define the document as well as to provide information that is used by the browser or search engine.



Tip Several common attributes used for structure, internationalization, and events are abbreviated as `core`, `i18n`, and `events` in the following quick reference sections. The description for each of these abbreviations can be found later in the “Common Attributes and Events” section.

<body>...</body>

Usage	Contains the document's content.
Attributes	<p>core, i18n, events.</p> <p>onload="..." Intrinsic event triggered when the document loads.</p> <p>onunload="..." Intrinsic event triggered when the document unloads.</p>
Notes	There can be only one <body>, and it must follow the <head>. The <body> element can be replaced by a <frameset> element.

<!DOCTYPE>

Usage	Version information appears on the first line of an HTML document and is an SGML declaration rather than an element.
Attributes	<p>html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"> Used for documents following the Strict XHTML requirements.</p> <p>html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"> Used for documents following the XHTML requirements, but also including some deprecated elements.</p> <p>html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd"> Used for framed documents.</p>

<head>...</head>

Usage	This is the document header and contains other elements that provide information to users and search engines.
Attributes	i18n. profile="..." URL specifying the location of meta data.
Notes	In general, there can be only one <head> per document. It must follow the opening <html> tag and precede the <body>. The <head> must include a <title>.

<html>...</html>

Usage	The html element contains the entire document.
Attributes	i18n.

<meta />

Usage	Provides information about the document.
Attributes	i18n. http-equiv="..." HTTP response header name. name="..." Name of the meta information. content="..." Content of the meta information. -scheme="..." Assigns a scheme to interpret the meta data.

<title>...</title>

Usage	This is the name you give your Web page. The <title> element is located in the <head> element and is displayed in the browser window title bar.
Attributes	i18n.

Text Phrases and Paragraphs

Text phrases (or blocks) can be structured to suit a specific purpose, such as creating a paragraph. This should not be confused with modifying the formatting of the text.

<address>...</address>

Usage	Provides a special format for author or contact information.
Attributes	core, i18n, events.
Notes	The <code>
</code> element is commonly used inside the <code><address></code> element to break the lines of an address.

<blockquote>...</blockquote>

Usage	Used to display long quotations.
Attributes	core, i18n, events. <code>cite="..."</code> The URL of the quoted text.

**
**

Usage	Forces a line break.
Attributes	core, i18n, events. <code>clear="..."</code> Sets the location where the next line begins after a floating object (none, left, right, all).

<div>...</div>

Usage	The division element is used to add structure to a block of text.
Attributes	core, i18n, events.
Notes	Cannot be used within a <code><p></code> element.

...

Usage Emphasized text.
Attributes core, i18n, events.

<h1>...</h1>—<h6>...</h6>

Usage The six headings (h1 is uppermost, or most important) are used in the body to structure information in a hierarchical fashion.
Attributes core, i18n, events.
Notes Visual browsers display the size of the headings in relation to their importance, <h1> being the largest and <h6> the smallest.

<p>...</p>

Usage Defines a paragraph.
Attributes core, i18n, events.

<pre>...</pre>

Usage Displays preformatted text.
Attributes core, i18n, events.
 width="..." The width of the formatted text.

...

Usage Stronger emphasis.
Attributes core, i18n, events.

_{...}

Usage Creates subscript.

Attributes core, i18n, events.

^{...}

Usage Creates superscript.

Attributes core, i18n, events.

Text Formatting Elements

Text characteristics such as the size, weight, and style can be modified using these elements, but the XHTML specification encourages you to use style sheets instead.

...

Usage Bold text.

Attributes core, i18n, events.

<big>...</big>

Usage Large text.

Attributes core, i18n, events.

<hr />

Usage Horizontal rules are used to separate sections of a Web page.

Attributes core, events.

noshade="..." Displays the rule as a solid color.

<i>...</i>

Usage Italicized text.

Attributes core, i18n, events.

<small>...</small>

Usage Small text.

Attributes core, i18n, events.

<tt>...</tt>

Usage Teletype (or monospaced) text.

Attributes core, i18n, events.

Lists

You can organize text into a more structured outline by creating lists. Lists can be nested.

<dd>...</dd>

Usage The definition description used in a <dl> (definition list) element.

Attributes core, i18n, events.

Notes Can contain block-level content, such as the <p> element.

<dl>...</dl>

Usage Creates a definition list.

Attributes core, i18n, events.

compact="compact" Deprecated. Compacts the displayed list.

Notes Must contain at least one <dt> or <dd> element in any order.

<dt>...</dt>

- Usage The definition term (or label) used within a <dl> (definition list) element.
- Attributes core, i18n, events.
- Notes Must contain text (which can be modified by text markup elements).

...

- Usage Defines a list item within a list.
- Attributes core, i18n, events.
- type="..." Changes the numbering style (1, a, A, i, I) in ordered lists, or the bullet style (disc, square, circle) in unordered lists.
- value="..." Sets the numbering to the given integer beginning with the current list item.

...

- Usage Creates an ordered list.
- Attributes core, i18n, events.
- start="..." Sets the starting number to the chosen integer.
- Notes Must contain at least one list item.

...

- Usage Creates an unordered list.
- Attributes core, i18n, events.
- Notes Must contain at least one list item.

Links `<a>...`

Hyperlinking is fundamental to HTML. These elements enable you to link to other documents, other locations within a document, or external files.

Usage	Used to define links and anchors.
Attributes	<code>core</code> , <code>i18n</code> , <code>events</code> .
	<code>charset="..."</code> Character encoding of the resource.
	<code>name="..."</code> Defines an anchor.
	<code>href="..."</code> The URL of the linked resource.
	<code>target="..."</code> Determines where the resource is displayed (user-defined name, <code>_blank</code> , <code>_parent</code> , <code>_self</code> , <code>_top</code>).
	<code>rel="..."</code> Forward link types.
	<code>rev="..."</code> Reverse link types.
	<code>accesskey="..."</code> Assigns a hotkey to this element.
	<code>shape="..."</code> Enables you to define client-side imagemaps using defined shapes (default, <code>rect</code> , <code>circle</code> , <code>poly</code>).
	<code>coords="..."</code> Sets the size of the shape using pixel or percentage lengths.
	<code>tabindex="..."</code> Sets the tabbing order between elements with a defined <code>tabindex</code> .

Tables

Tables are meant to display data in a tabular format. Tables are also widely used for page layout purposes.

`<caption>...</caption>`

Usage	Displays a table caption.
Attributes	<code>core</code> , <code>i18n</code> , <code>events</code> .

<table>...</table>

Usage Creates a table.

Attributes core, i18n, events.

width="..." Table width.

cols="..." The number of columns.

border="..." The width in pixels of a border around the table.

frame="..." Sets the visible sides of a table (void, above, below, hside, lhs, rhs, vside, box, border).

rules="..." Sets the visible rules within a table (none, groups, rows, cols, all).

cellspacing="..." Spacing between cells.

cellpadding="..." Spacing in cells.

<td>...</td>

Usage Defines a cell's contents.

Attributes core, i18n, events.

axis="..." Abbreviated name.

axes="..." axis names listing row and column headers pertaining to the cell.

rowspan="..." The number of rows spanned by a cell.

colspan="..." The number of columns spanned by a cell.

char="..." Sets a character on which the column aligns.

charoff="..." Offset to the first alignment character on a line.

<th>...</th>

Usage Defines the cell contents of the table header.

Attributes core, i18n, events.

axis="..." Abbreviated name.

axes="..." axis names listing row and column headers pertaining to the cell.

rowspan="..." The number of rows spanned by a cell.

colspan="..." The number of columns spanned by a cell.

char="..." Sets a character on which the column aligns.

charoff="..." Offset to the first alignment character on a line.

<tr>...</tr>

Usage Defines a row of table cells.

Attributes core, i18n, events.

char="..." Sets a character on which the column aligns.

charoff="..." Offset to the first alignment character on a line.

Frames

Frames create new panels in the Web browser window that are used to display content from different source documents.

<frame />

Usage	Defines a frame.
Attributes	<p><code>name="..."</code> The name of a frame.</p> <p><code>src="..."</code> The source to be displayed in a frame.</p> <p><code>frameborder="..."</code> Toggles the border between frames (0, 1).</p> <p><code>marginwidth="..."</code> Sets the space between the frame border and content.</p> <p><code>marginheight="..."</code> Sets the space between the frame border and content.</p> <p><code>noresize</code> Disables sizing.</p> <p><code>scrolling="..."</code> Determines scrollbar presence (auto, yes, no).</p>
Notes	Use the Frameset DTD.

<frameset>...</frameset>

Usage	Defines the layout of frames within a window.
Attributes	<p><code>rows="..."</code> The number of rows.</p> <p><code>cols="..."</code> The number of columns.</p> <p><code>onload="..."</code> The intrinsic event triggered when the document loads.</p> <p><code>onunload="..."</code> The intrinsic event triggered when the document unloads.</p>
Notes	Use the Frameset DTD.

<iframe>...</iframe>

Usage	Creates an inline frame.
Attributes	<code>name="..."</code> The name of the frame.

`src="..."` The source to be displayed in a frame.

`frameborder="..."` Toggles the border between frames (0, 1).

`marginwidth="..."` Sets the space between the frame border and content.

`marginheight="..."` Sets the space between the frame border and content.

`scrolling="..."` Determines scrollbar presence (auto, yes, no).

`height="..."` Height.

`width="..."` Width.

<noframes>...</noframes>

Usage	Alternative content when frames are not supported.
Attributes	None.
Notes	Use the Frameset DTD. The <body> element must be included inside the <noframes> tag.

Embedded Content

Also called inclusions, embedded content applies to Java applets, imagemaps, and other multimedia or programmed content that is placed in a Web page to provide additional functionality.

Comments <!-- ... -->

Usage	Used to insert notes or scripts that are not displayed by the browser.
Attributes	None.
Notes	Comments are not restricted to one line and can be any length. The end tag is not required to be on the same line as the start tag.

Usage Includes an image in the document.

Attributes core, i18n, events.

src="..." The URL of the image.

alt="..." Alternative text to display.

height="..." The height of the image.

width="..." The width of the image.

border="..." Border width.

hspace="..." The horizontal space separating the image from other content.

vspace="..." The vertical space separating the image from other content.

usemap="..." The URL to a client-side imagemap.

ismap="ismap" Identifies a server-side imagemap.

<map>...</map>

Usage When used with the <area> element, it creates a client-side imagemap.

Attributes core.

name="..." The name of the imagemap to be created.

<object>...</object>

Usage Includes an object or applet.

Attributes core, i18n, events.

declare="declare" A flag that declares, but doesn't create an object.

`classid="..."` The URL of the object's location.

`codebase="..."` The URL for resolving URLs specified by other attributes.

`data="..."` The URL to the object's data.

`type="..."` The Internet content type for data.

`codetype="..."` The Internet content type for the code.

`standby="..."` Shows message while loading.

`height="..."` The height of the object.

`width="..."` The width of the object.

`border="..."` Displays the border around an object.

`hspace="..."` The space between the sides of the object and other page content.

`vspace="..."` The space between the top and bottom of the object and other page content.

`usemap="..."` The URL to an imagedmap.

`shapes=` Enables you to define areas to search for hyperlinks if the object is an image.

`name="..."` The URL to submit as part of a form.

`tabindex="..."` Sets the tabbing order between elements with a defined `tabindex`.

Style **<style>...</style>**

Style sheets (both embedded and linked) are incorporated into an HTML document through the use of the `<style>` element.

Usage Creates an internal style sheet.

Attributes `i18n`.

`type="..."` The Internet content type.

`media="..."` Defines the destination medium
(screen, print, projection, braille, speech, all).

`title="..."` The title of the style.

Notes Located within the `<head>` element.

Forms

Forms create an interface for the user to select options and return data to the Web server.

`<button>...</button>`

Usage Creates a button.

Attributes `core, i18n, events`.

`name="..."` The button name.

`value="..."` The value of the button.

`type="..."` The button type (button, submit, reset).

`disabled="..."` Sets the button state to disabled.

`tabindex="..."` Sets the tabbing order between elements with a defined `tabindex`.

`onfocus="..."` The event that occurs when the element receives focus.

`onblur="..."` The event that occurs when the element loses focus.

`<form>...</form>`

Usage Creates a form that holds controls for user input.

Attributes `core, i18n, events`.

`action="..."` The URL for the server action.

`method="..."` The HTTP method (get, post). get is deprecated.

`enctype="..."` Specifies the MIME (Internet media type).

`onsubmit="..."` The intrinsic event that occurs when the form is submitted.

`onreset="..."` The intrinsic event that occurs when the form is reset.

`target="..."` Determines where the resource is displayed (user-defined name, `_blank`, `_parent`, `_self`, `_top`).

`accept-charset="..."` The list of character encodings.

<input />

Usage Defines controls used in forms.

Attributes `core`, `i18n`, `events`.

`type="..."` The type of input control (text, password, checkbox, radio, submit, reset, file, hidden, image, button).

`name="..."` The name of the control (required except for submit and reset).

`value="..."` The initial value of the control (required for radio and checkboxes).

`checked="checked"` Sets the radio buttons to a checked state.

`disabled="..."` Disables the control.

`readonly="..."` For text password types.

`size="..."` The width of the control in pixels except for text and password controls, which are specified in number of characters.

`maxlength="..."` The maximum number of characters that can be entered.

`src="..."` The URL to an image control type.

`alt="..."` An alternative text description.

`usemap="..."` The URL to a client-side imagemap.

`tabindex="..."` Sets the tabbing order between elements with a defined `tabindex`.

`onfocus="..."` The event that occurs when the element receives focus.

`onblur="..."` The event that occurs when the element loses focus.

`onselect="..."` Intrinsic event that occurs when the control is selected.

`onchange="..."` Intrinsic event that occurs when the control is changed.

`accept="..."` File types allowed for upload.

<label>...</label>

Usage Labels a control.

Attributes `core`, `i18n`, `events`.

`for="..."` Associates a label with an identified control.

`disabled="..."` Disables a control.

`accesskey="..."` Assigns a hotkey to this element.

`onfocus="..."` The event that occurs when the element receives focus.

`onblur="..."` The event that occurs when the element loses focus.

<legend>...</legend>

Usage Assigns a caption to a fieldset.

Attributes core, i18n, events.

accesskey="..." Assigns a hotkey to this element.

<option>...</option>

Usage Specifies choices in a <select> element.

Attributes core, i18n, events.

selected="selected" Specifies whether the option is selected.

disabled="disabled" Disables the control.

value="..." The value submitted if a control is submitted.

<select>...</select>

Usage Creates choices for the user to select.

Attributes core, i18n, events.

name="..." The name of the element.

size="..." The height in number of visible rows.

multiple="multiple" Allows multiple selections.

disabled="disabled" Disables the control.

tabindex="..." Sets the tabbing order between elements with a defined tabindex.

onfocus="..." The event that occurs when the element receives focus.

onblur="..." The event that occurs when the element loses focus.

`onselect="..."` Intrinsic event that occurs when the control is selected.

`onchange="..."` Intrinsic event that occurs when the control is changed.

<textarea>...</textarea>

Usage Creates an area for user input with multiple lines.

Attributes `core`, `i18n`, `events`.

`name="..."` The name of the control.

`rows="..."` The height in number of rows.

`cols="..."` The width in number of columns.

`disabled="disabled"` Disables the control.

`readonly="readonly"` Sets the displayed text to read-only status.

`tabindex="..."` Sets the tabbing order between elements with a defined `tabindex`.

`onfocus="..."` The event that occurs when the element receives focus.

`onblur="..."` The event that occurs when the element loses focus.

`onselect="..."` Intrinsic event that occurs when the control is selected.

`onchange="..."` Intrinsic event that occurs when the control is changed.

Notes Text to be displayed is placed within the start and end tags.

Scripts

Scripting language is made available to process data and performs other dynamic events through the `<script>` element.

`<script>...</script>`

Usage	The <code><script></code> element contains client-side scripts that are executed by the browser.
Attributes	<code>type="..."</code> Script language Internet content type. <code>src="..."</code> The URL for the external script.
Notes	You can set the default scripting language in the <code><meta /></code> element.

`<noscript>...</noscript>`

Usage	Provides alternative content for browsers unable to execute a script.
Attributes	None.

Common Attributes and Events

Four attributes are abbreviated as core. They are

- `id="..."` A global identifier.
- `class="..."` A list of classes separated by spaces.
- `style="..."` Style information.
- `title="..."` Provides more information for a specific element, as opposed to the `<title>` element, which entitles the entire Web page.

Two attributes for internationalization (i18n) are abbreviated as i18n:

- `lang="..."` The language identifier.
- `dir=c` The text direction (`ltr`, `rtl`).

The following intrinsic events are abbreviated events:

- `OnClick="..."` A pointing device (such as a mouse) was single-clicked.
- `OnDb1Click="..."` A pointing device (such as a mouse) was double-clicked.
- `OnMouseDown="..."` A mouse button was clicked and held down.
- `OnMouseUp="..."` A mouse button that was clicked and held down was released.
- `OnMouseOver="..."` A mouse moved the cursor over an object.
- `OnMouseMove="..."` The mouse was moved.
- `OnMouseOut="..."` A mouse moved the cursor off an object.
- `OnKeyPress="..."` A key was pressed and released.
- `OnKeyDown="..."` A key was pressed and held down.
- `OnKeyUp="..."` A key that was pressed has been released.



APPENDIX B

Style Sheet Quick Reference

Cascading Style Sheets enable Web authors to attach styles (for example, fonts, spacing, and colors) to HTML documents. By separating the presentation style of documents from the content of documents, the style sheet specification simplifies Web authoring and site maintenance. This appendix provides a quick reference to some of the most common style sheet properties.



Note This appendix is based on the information provided in the *CSS 2.1 Specification W3C Recommendation* (revised on June 15, 2005). The latest version of this standard can be found at www.w3.org/TR/CSS21/.

To make the information readily accessible, this appendix organizes style sheet properties by their function in the following order:

- Text and fonts
- Typography
- Colors and backgrounds
- Borders and tables
- Lists
- Layout

The elements are listed alphabetically within each section, and the following information is presented:

- **Usage** A general description of the property.
- **Values** Lists the values of the property with a short description of their effect.
- **Initial Value** Lists the default value of the property. It is not necessary to set this value.
- **Notes** Relates any special considerations when using the property.

Text and Fonts

Sets the fonts, colors, positioning, and other styles for the text elements.



Tip Links can use all the same properties as other text elements, but you need to remember the link types:

- `a:link`
- `a:visited`
- `a:active`
- `a:hover`

font-family

Usage Sets the font to be used.

Values `<family name>` The name of the font family (such as Arial) as it appears in your editor.

`<generic family>` Sets a generic font set dependent on the user's computer (such as, serif, sans-serif, cursive, fantasy, and monospace).

`inherit` The same as the parent element.

Initial Value Depends on the browser setting.



Tip Because not all computer systems ship with the same fonts, there are only a few sure-fire fonts worth specifying:

- Arial
- Arial Black
- Comic Sans
- Courier New
- Georgia
- Impact
- Times New Roman
- Trebuchet
- Verdana

font-size

Usage	Sets the font size.
Values	<code><absolute-size></code> The size of the font expressed in points (pt), inches (in), centimeters (cm), pixels (px), or one of the absolute keywords (xx-small, x-small, small, medium, large, x-large, xx-large, smaller, or larger) which determines the size of the font relative to the default value. <code>inherit</code> The same as the parent element.
Initial Value	medium; typically 12 points.

font-style

Usage	Sets the style of the font.
Values	<code>normal</code> No special format to the font. <code>italic</code> The font appears in italics. <code>inherit</code> The same as the parent element.
Initial Value	<code>normal</code> .

font-weight

Usage	Sets the weight of the font.
Values	<code><weight></code> The weight of the font may be extra-light, light, demi-light, medium, normal, demi-bold, bold, or extra-bold. <code><relative-weight></code> The weight of the font may be relative to some inherited value (for example, bolder or lighter) <code>100–900</code> The weight of the font as a specific numerical value, where <code>400</code> is approximately the same as normal and <code>700</code> is approximately the same as bold. <code>inherit</code> The same as the parent element.
Initial Value	normal.

font-variant

Usage	Changes the appearance of the font.
Values	<code>small-caps</code> Transforms all text into small caps. <code>inherit</code> The same as the parent element.
Initial Value	normal.

text-decoration

Usage	Sets the format of the text element.
Values	<code>underline</code> A line appears under the text. <code>overline</code> A line appears over the text. <code>line-through</code> A line appears horizontally through the center of the text. <code>blink</code> The text blinks. <code>inherit</code> The same as the parent element.
Initial Value	none (or underline for links).

text-transform

Usage	Sets the format of the text element.
Values	<code>capitalize</code> Transforms the case of the first letter of every word to uppercase. <code>uppercase</code> All text appears uppercase. <code>lowercase</code> All text appears lowercase. <code>inherit</code> The same as the parent element.
Initial Value	<code>none</code> .

Typography

This sets the style that determines the spacing and alignment of text elements.

word-spacing

Usage	Sets the spacing between words.
Values	<code><length></code> The spacing may be expressed in inches (in), centimeters (cm), points (pt), and pixels (px). <code>inherit</code> The same as the parent element.
Initial Value	<code>normal</code>

letter-spacing

Usage	Sets the spacing between letters, also known as kerning.
Values	<code><length></code> The spacing may be expressed in inches (in), centimeters (cm), points (pt), and pixels (px). <code>inherit</code> The same as the parent element.
Initial Value	<code>normal</code>

line-height

Usage	Sets the total height of a line of text, including the space above and below the text.
Values	<code><length></code> The spacing may be expressed in inches (in), centimeters (cm), points (pt), and pixels (px). <code>inherit</code> The same as the parent element.
Initial Value	<code>normal</code>



Tip Each of these properties may also be expressed as a negative (for example, `letter-spacing: -10px`), which decreases the normal spacing to create interesting effects.

text-align

Usage	Sets the alignment of the text element.
Values	<code>left</code> The text is aligned on the left side. <code>right</code> The text is aligned on the right side. <code>center</code> The text is centered. <code>justify</code> The text is aligned on both the left and right sides. <code>inherit</code> The same as the parent element.
Initial Value	Depends on the browser settings.

vertical-align

Usage	Sets the vertical alignment of the element relative to the surrounding elements.
Values	<code>baseline</code> The element aligns with the baseline of surrounding elements. <code>sub</code> or <code>super</code> The element is subscripted or superscripted respectively.

`top, middle, bottom` The element is aligned with the top, middle, or bottom of surrounding elements respectively.

`text-top` or `text-bottom` The element is aligned with the top or bottom of surrounding text elements.

`inherit` The same as the parent element.

Initial Value `baseline`.

text-indent

Usage Sets the spacing before the text element relative to the surrounding elements.

Values `<length>` The indent spacing is a fixed length expressed in inches (`in`), centimeters (`cm`), points (`pt`), or pixels (`px`).

`<percentage>` The indent spacing is a percentage of the containing element (usually the browser window or the table cell).

`inherit` The same as the parent element.

Initial Value `0`.

Colors and Backgrounds

This sets the style for the background of the text, page, table, or other elements.



Caution As with other elements, color does not appear the same on all computers; however, most monitors will display a 256-color palette (with 216 of those colors consistent across all platforms). Lynda Weinmann's website (www.lynda.com/hex.html) contains two charts sorting these browser-safe colors sorted by both value and hue. They are well worth checking out.

background-color

Usage	Sets the background color of an element.
Values	<code><color></code> The hex number (or text equivalent) of the preferred color.
	<code>transparent</code> The same background as the underlying element.
Initial Value	<code>transparent</code>

background-image

Usage	Sets the background image for an element.
Values	<code>url("...")</code> The URL for the background image.
	<code>none</code> No image.
Initial Value	<code>none</code>

background-position

Usage	Sets the position of the background image. The values always appear in pairs—the first number refers to the horizontal positioning of the image; the second to the vertical positioning. For example, <code>background-position: 0% 100%</code> will place the image at the top right of the browser window.
Values	<code><percentage></code> The position of the background image.
	<code>top</code> Corresponds to 0%.
	<code>bottom</code> Corresponds to 100%.
	<code>left</code> Corresponds to 0%.
	<code>right</code> Corresponds to 100%.
	<code>center</code> Corresponds to 50%.
Initial Value	<code>none</code> .

background-repeat

Usage	Sets the pattern of repeats for the background image.
Values	<p><code><no-repeat></code> The background image does not repeat.</p> <p><code><repeat></code> The background image repeats both horizontally and vertically (as normal) to fill the entire space.</p> <p><code><repeat-x></code> The background image repeats horizontally only to fill the entire space.</p> <p><code><repeat-y></code> The background image repeats vertically only to fill the entire space.</p>
Initial Value	repeat.

color

Usage	Sets the color of the font.
Values	<p><code><color></code> The hex number (or text equivalent) of the preferred color.</p> <p><code>inherit</code> The same as the parent element.</p>
Initial Value	Depends on the browser setting.



Tip Some colors have been defined in the style sheet specification by a name, not a hexadecimal code. The following 16 are guaranteed to appear the same in all resolutions.

- | | | | |
|-----------|----------|----------|----------|
| • aqua | • gray | • navy | • silver |
| • black | • green | • olive | • teal |
| • blue | • lime | • purple | • white |
| • fuchsia | • maroon | • red | • yellow |

Borders and Tables

These set the border styles for the page, text, table, and image elements.

border-color

Usage	Sets the border color for an element.
Values	<code><color></code> The hex number (or text equivalent) of the preferred color. <code>inherit</code> The same as the parent element. <code>none</code> No image.
Initial Value	transparent.
Notes	Set the color for specific borders using the <code>border-top-color</code> , <code>border-right-color</code> , <code>border-bottom-color</code> , and <code>border-left-color</code> properties.

border-style

Usage	Sets the style of the border for an element.
Values	<code>dotted</code> A series of small dots form the border. <code>dashed</code> A series of dashes form the border. <code>solid</code> A narrow, solid line forms the border. <code>double</code> Double, narrow, solid lines form the border. <code>groove</code> A narrow carved line forms the border. <code>ridge</code> A narrow raised line forms the border. <code>inset</code> The border makes the entire element appear to be embedded. <code>outset</code> The border makes the entire element appear to be raised. <code>inherit</code> The same as the parent element. <code>none</code> No border.
Initial Value	none.
Notes	Set the position of the border with the <code>border-top-style</code> , <code>border-right-style</code> , <code>border-bottom-style</code> , and <code>border-left-style</code> properties.

border-width

Usage	Sets the width of the border for an element.
Values	<code><width></code> The width is a fixed length expressed in inches (in), centimeters (cm), points (pt), or pixels (px). <code>thin</code> A thin line forms the border. <code>medium</code> A medium line forms the border. <code>thick</code> A thick line forms the border. <code>inherit</code> The same as the parent element.
Initial Value	<code>medium</code> .
Notes	Set the width of specific borders using the <code>border-top-width</code> , <code>border-right-width</code> , <code>border-bottom-width</code> , and <code>border-left-width</code> properties.

caption-side

Usage	Sets the position of the caption relative to the table.
Values	<code>top</code> The caption appears at the top of the table. <code>bottom</code> The caption appears at the bottom of the table. <code>inherit</code> The same as the parent element.
Initial Value	<code>0</code> .

empty-cells

Usage	Specifies the treatment of empty table cells.
Values	<code>show</code> Empty cells are treated the same as nonempty cells. <code>hide</code> Empty cells are not displayed. <code>inherit</code> The same as the parent element.
Initial Value	<code>show</code> .

float

Usage	Sets the spacing before the text element relative to the surrounding elements.
Values	<p><code>left</code> The content of the element floats to the left.</p> <p><code>right</code> The content of the element floats to the right.</p> <p><code>none</code> The content does not float.</p> <p><code>inherit</code> The same as the parent element.</p>
Initial Value	<code>none</code> .

Lists

These styles set styles (for example, the bullet type) for ordered and unordered lists. Combine these styles with margins and indents.

list-style-image

Usage	Sets an image to replace the list bullets.
Values	<p><code>url("...")</code> The URL for the image.</p> <p><code>none</code> No images.</p> <p><code>inherit</code> The same as the parent element.</p>
Initial Value	<code>none</code>

list-style-type

Usage	Sets the style of the unordered list bullets (or numbers in ordered lists).
Values	<p><code><shape></code> The bullets can be set to <code>disc</code> (closed circle), <code>circle</code> (open circle), or <code>square</code> (closed square).</p> <p><code><alignment></code> The bullets can be aligned around <code>decimal</code> or <code>decimal-leading-zero</code>.</p>

`<text>` Numbers in an ordered list can be Roman numerals (`lower-roman`, `upper-roman`), Latin numerals (`lower-latin`, `upper-latin`), or American alphabet characters (`lower-alpha`, `upper-alpha`).

`inherit` The same as the parent element.

Initial Value `disc`.

Layout

These styles can be used in combination with any of the previous style types to affect the entire page, or components of the page.

margin

Usage Sets the margins for the page.

Values `<margin-width>` The width of the margin in percentage or in fixed width.

`inherit` The same as the parent element.

Initial Value `0`.

Notes Set specific margins using the `margin-top`, `margin-right`, `margin-bottom`, and `margin-left` properties.

padding

Usage Sets the space that surrounds the element.

Values `<length>` Sets the table padding to a fixed length.

`<percentage>` Sets the table padding to a percentage.

Initial Value `0`.

Notes Set specific padding using the `padding-top`, `padding-right`, `padding-bottom`, and `padding-left` properties.

APPENDIX C

Special Characters



Symbols (such as & and ‘) are used in everyday writing. HTML uses a shorthand code to display certain characters correctly. The most frequently used characters are referenced in the following tables. As with any code, review your finished pages in the browser before sending your work to the Internet.



Caution The entity references in the tables that follow are case sensitive.

Symbol Entities

Character	Description	Entity Reference	Entity Number
“	quotation mark	"	"
‘	apostrophe	' (does not work in IE)	'
&	ampersand	&	&
<	less-than	<	<
>	greater-than	>	>
	non-breaking space	 	
¡	inverted exclamation mark	¡	¡

continues

Character	Description	Entity Reference	Entity Number
¤	currency	¤	¤
¢	cent	¢	¢
£	pound	£	£
¥	yen	¥	¥
¸	broken vertical bar	¦	¦
§	section	§	§
¨	spacing diaeresis	¨	¨
©	copyright	©	©
^a	feminine ordinal indicator	ª	ª
«	angle quotation mark (left)	«	«
¬	negation	¬	¬
–	soft hyphen	­	­
®	registered trademark	®	®
™	trademark	™	™
ˉ	spacing macron	¯	¯
°	degree	°	°
±	plus-or-minus	±	±
²	superscript 2	²	²
³	superscript 3	³	³
´	spacing acute	´	´
µ	micro	µ	µ
¶	paragraph	¶	¶
·	middle dot	·	·
¸	spacing cedilla	¸	¸

Character	Description	Entity Reference	Entity Number
¹	superscript 1	¹	¹
°	masculine ordinal indicator	º	º
»	angle quotation mark (right)	»	»
¼	fraction 1/4	¼	¼
½	fraction 1/2	½	½
¾	fraction 3/4	¾	¾
¿	inverted question mark	¿	¿
×	multiplication	×	×
÷	division	÷	÷

Character Entities

Character	Description	Entity Reference	Entity Number
À	capital a, grave accent	À	À
Á	capital a, acute accent	Á	Á
Â	capital a, circumflex accent	Â	Â
Ã	capital a, tilde	Ã	Ã
Ä	capital a, umlaut mark	Ä	Ä
Å	capital a, ring	Å	Å
Æ	capital ae	Æ	Æ
Ç	capital c, cedilla	Ç	Ç
È	capital e, grave accent	È	È
É	capital e, acute accent	É	É

continues

Character	Description	Entity Reference	Entity Number
Ê	capital e, circumflex accent	Ê	Ê
Ë	capital e, umlaut mark	Ë	Ë
Ì	capital i, grave accent	Ì	Ì
Í	capital i, acute accent	Í	Í
Î	capital i, circumflex accent	Î	Î
Ï	capital i, umlaut mark	Ï	Ï
Ð	capital eth, Icelandic	Ð	Ð
Ñ	capital n, tilde	Ñ	Ñ
Ò	capital o, grave accent	Ò	Ò
Ó	capital o, acute accent	Ó	Ó
Ô	capital o, circumflex accent	Ô	Ô
Õ	capital o, tilde	Õ	Õ
Ö	capital o, umlaut mark	Ö	Ö
Ø	capital o, slash	Ø	Ø
Ù	capital u, grave accent	Ù	Ù
Ú	capital u, acute accent	Ú	Ú
Û	capital u, circumflex accent	Û	Û
Ü	capital u, umlaut mark	Ü	Ü
Ý	capital y, acute accent	Ý	Ý
Þ	capital THORN, Icelandic	Þ	Þ
ß	small sharp s, German	ß	ß
à	small a, grave accent	à	à
á	small a, acute accent	á	á
â	small a, circumflex accent	â	â
ã	small a, tilde	ã	ã
ä	small a, umlaut mark	ä	ä

Character	Description	Entity Reference	Entity Number
å	small a, ring	å	å
æ	small ae	æ	æ
ç	small c, cedilla	ç	ç
è	small e, grave accent	è	è
é	small e, acute accent	é	é
ê	small e, circumflex accent	ê	ê
ë	small e, umlaut mark	ë	ë
ì	small i, grave accent	ì	ì
í	small i, acute accent	í	í
î	small i, circumflex accent	î	î
ï	small i, umlaut mark	ï	ï
ð	small eth, Icelandic	ð	ð
ñ	small n, tilde	ñ	ñ
ò	small o, grave accent	ò	ò
ó	small o, acute accent	ó	ó
ô	small o, circumflex accent	ô	ô
õ	small o, tilde	õ	õ
ö	small o, umlaut mark	ö	ö
ø	small o, slash	ø	ø
ù	small u, grave accent	ù	ù
ú	small u, acute accent	ú	ú
û	small u, circumflex accent	û	û
ü	small u, umlaut mark	ü	ü
ý	small y, acute accent	ý	ý
þ	small thorn, Icelandic	þ	þ
ÿ	small y, umlaut mark	ÿ	ÿ

Greek Entities

Character	Description	Entity Reference	Entity Number
A	Greek capital letter alpha	Α	Α
B	Greek capital letter beta	Β	Β
Γ	Greek capital letter gamma	Γ	Γ
Δ	Greek capital letter delta	Δ	Δ
E	Greek capital letter epsilon	Ε	Ε
Z	Greek capital letter zeta	Ζ	Ζ
H	Greek capital letter eta	Η	Η
Θ	Greek capital letter theta	Θ	Θ
I	Greek capital letter iota	Ι	Ι
K	Greek capital letter kappa	Κ	Κ
Λ	Greek capital letter lambda	Λ	Λ
M	Greek capital letter mu	Μ	Μ
N	Greek capital letter nu	Ν	Ν
Ξ	Greek capital letter xi	Ξ	Ξ
O	Greek capital letter omicron	Ο	Ο
Π	Greek capital letter pi	Π	Π
P	Greek capital letter rho	Ρ	Ρ
Σ	Greek capital letter sigma	Σ	Σ
T	Greek capital letter tau	Τ	Τ
Υ	Greek capital letter upsilon	Υ	Υ
Φ	Greek capital letter phi	Φ	Φ
X	Greek capital letter chi	Χ	Χ
Ψ	Greek capital letter psi	Ψ	Ψ

Character	Description	Entity Reference	Entity Number
Ω	Greek capital letter omega	Ω	Ω
α	Greek small letter alpha	α	α
β	Greek small letter beta	β	β
γ	Greek small letter gamma	γ	γ
δ	Greek small letter delta	δ	δ
ε	Greek small letter epsilon	ε	ε
ζ	Greek small letter zeta	ζ	ζ
η	Greek small letter eta	η	η
θ	Greek small letter theta	θ	θ
ι	Greek small letter iota	ι	ι
κ	Greek small letter kappa	κ	κ
λ	Greek small letter lambda	λ	λ
μ	Greek small letter mu	μ	μ
ν	Greek small letter nu	ν	ν
ξ	Greek small letter xi	ξ	ξ
ο	Greek small letter omicron	ο	ο
π	Greek small letter pi	π	π
ρ	Greek small letter rho	ρ	ρ
ς	Greek small letter final sigma	ς	ς
σ	Greek small letter sigma	σ	σ
τ	Greek small letter tau	τ	τ
υ	Greek small letter upsilon	υ	υ
φ	Greek small letter phi	φ	φ
χ	Greek small letter chi	χ	χ

continues

Character	Description	Entity Reference	Entity Number
ψ	Greek small letter psi	ψ	ψ
ω	Greek small letter omega	ω	ω
ϑ	Greek small letter theta symbol	ϑ (does not work in IE)	ϑ
γ	Greek upsilon with hook symbol	ϒ (does not work in IE)	ϒ
ϖ	Greek pi symbol	ϖ (does not work in IE)	ϖ

Other Entities

Character	Description	Entity Reference	Entity Number
Œ	capital ligature OE	Œ	Œ
œ	small ligature oe	œ	œ
Š	capital S with caron	Š	Š
š	small S with caron	š	š
Ÿ	capital Y with diaeresis	Ÿ	Ÿ
^	modifier letter circumflex accent	ˆ	ˆ
~	small tilde	˜	˜
	en space	 	 
	em space	 	 
	thin space	 	 

Character	Description	Entity Reference	Entity Number
—	en dash	–	–
—	em dash	—	—
‘	left single quotation mark	‘	‘
’	right single quotation mark	’	’
,	single low-9 quotation mark	‚	‚
“	left double quotation mark	“	“
”	right double quotation mark	”	”
„	double low-9 quotation mark	„	„
†	dagger	†	†
‡	double dagger	‡	‡
...	horizontal ellipsis	…	…
‰	per mille	‰	‰
◁	single left-pointing angle quotation	‹	‹
▷	single right-pointing angle quotation	›	›
€	euro	€	€

This page intentionally left blank

INDEX



SYMBOLS

<!--...--> tag, 182
^ (acute accent), 24
& (ampersand) symbol, 24
© (copyright) symbol, 24
<!DOCTYPE> tag, 171
/ (forward slash), 92
 HTML tag pairs, 13
 hyperlinks, 34
` (grave accent), 24
> (greater than) symbol, 24
< (less than) symbol, 24
- (minus) symbol, 24
 (non-breaking space) special character, 25
(number) symbol, 24
% (percent) symbol, 24, 92
+ (plus) symbol, 24
" " (quotes), 165
® (registered trademark) symbol, 24
_ (underscore), 17

A

<a href> tag, 36, 104
<a> tag, 32, 177-178
 href attribute, 33, 36
 id attribute, 35
 sound/video, 116
a:active selector, 51

a:hover selector, 51
a:link selector, 50
a:visited selector, 50
<absolute-size> tag, 194
action attribute, 105
active frames, 102
active Web pages, 134
 ActiveX, 137-138
 DHTML, 135-136
 Java, 137-138
ActiveX, 137-138
acute accent (´), 24
Add Me Web site, 156
adding
 images, 71
 sound, 116-119
 styles, 44-45
 text links for image maps, 87-88
 text pop-ups, 73-74
 video, 116-119
 Web sites to search engines, 155-156
<address> tag, 173
addresses (URLs), 32-33
Adobe GoLive, 149
Advanced Research Projects Agency (ARPA), 5
advertising, 157
align attribute
 tag, 76
 <table> tag, 66

- alignment
 - tables, 66
 - text, 76, 197
- alt attribute, 74
- alternate fonts, 127
- Amazon Web site, 82
- American Idol Web site, 69
- ampersand (&) symbol, 24
- analyzing data, 160
- anchor (<a>) tag, 32
- anchors, 35-37
- AnyBrowser Web site, 9
- appearance attribute, 119
- <area> tag, 86
- ARPA (Advanced Research Projects Agency), 5
- ARPAnet, 5
- attributes
 - action, 105
 - align, 66, 76
 - alt, 74
 - appearance, 119
 - autorewind, 119
 - autostart, 119
 - bgcolor, 66
 - border, 66
 - cellpadding, 66
 - cellspacing, 66
 - class, 41-43, 190
 - classid, 137
 - codebase, 137
 - cols, 91
 - colspan, 66-68
 - controls, 118
 - core, 190
 - dir, 190
 - height, 75-76
 - hidden, 119
 - href, 33, 36
 - <html> tag, 16
 - id, 35, 92, 190
 - tag, 75-76
 - internationalization, 190
 - lang, 26, 190
 - loop, 119
 - marginheight, 93
 - marginwidth, 93
 - method, 105
 - name, 92-93
 - noresize, 93
 - quotes, 165
 - refresh, 29
 - rows, 91
 - rowspan, 66-68
 - scrolling, 93
 - src, 71, 92
 - style, 190
 - target, 34, 99
 - title, 190
 - type, 108, 139
 - usemap, 87
 - valign, 66
 - value, 61
 - width, 66, 75-76, 118
- audio. *See* sound
- autorewind attribute, 119
- autostart attribute, 119

B

- tag, 175
- background property, 47, 52
- background-color property, 51, 199
- background-image property, 199
- background-position property, 199
- background-repeat property, 200
- backgrounds
 - color, 66, 199
 - images, 130-132, 199
 - repeating, 200
 - style sheet properties, 198-200
- BBEdit, 149
- bgcolor attribute, 66
- <big> tag, 175
- <blockquote> tag, 173
- </body> tag, 12
- <body> tag, 12, 171
- boldface text, 21
- border attribute, 66
- border-color property, 201
- border-style property, 201
- border-width property, 202
- borders
 - color, 201
 - frames, 93

- style sheet properties, 200-203
- styles, 201
- tables, 66, 69
- width, 202
- `
` tag, 20, 173
- browsers
 - compatibility, 8, 163
 - copying URLs from, 33
 - event handler support, 141
 - `<iframe>` tag support, 96
 - image compatibility, 74
 - limitations, 8
 - Lynx, 8
 - new windows, 34
 - paragraph text, reading, 19
 - source code, 14
 - style sheets
 - precedence*, 46
 - properties*, 50
 - text, 46
 - Web page formatting, 8
- bulleted lists, 57
 - formatting, 58-59
 - images, 203
- `<button>` tag, 184

C

- caption-side property, 202
- `<caption>` tag, 178
- captions (tables), 202
- Cascading Style Sheets. *See* CSS
- case sensitivity, 17
- cellpadding attribute, 66
- cells, 64, 202
- cellspacing attribute, 66
- Cerious Software Thumbs Plus, 79
- character entities, list of, 207-209
- check boxes, 109-110
- choosing
 - colors, 51
 - style sheets, 43
- class attribute, 41-43, 190
- classid attribute, 137
- client-side images, 85

- CNET
 - plug-ins Web site, 120
 - Web Hosting Buying Guide Web site, 152
- CNN Web site, 69
- code, testing, 163
- codebase attribute, 137
- color
 - backgrounds, 66, 199
 - borders, 201
 - charts online, 126
 - designs, 126-128
 - formatting, 51-52, 200
- color property, 47, 51-52, 200
- `<color>` tag, 200
- cols attribute, 91
- colspan attribute, 66-68
- commands, 14
- comments, 182
- complex tables, 68
- connecting to Internet, 9
- content (designs), 124
- controls attribute, 118
- coordinates (image maps), 83-85
- copying
 - images, 80
 - URLs, 33
- copyright symbol (©), 24
- core attributes, 190
- Corel® Paint Shop Pro Web site, 83
- crawlers, 153-154
- creating
 - expiration dates, 30
 - hyperlinks
 - anchors*, 35-37
 - email*, 34
 - files*, 33-34
 - Web pages*, 33
 - paragraphs, 19-20
 - splash pages, 29
- CSS (Cascading Style Sheets), 22, 39
 - backgrounds, 198-200
 - borders, 200-203
 - choosing, 43
 - class attribute, 41-43
 - color, 200

conflicts, 166
 declarations, 40
 defined, 39
 embedded, 39, 44
 fonts, 193-194
 horizontal lines, adding, 53-55
 inline, 40, 45
 layout, 204
 linked, 40, 44-45
 lists, 203-204
 margins, 55
 precedence, 46
 properties
 background-color, 199
 background-image, 199
 background-position, 199
 background-repeat, 200
 border-color, 201
 border-style, 201
 border-width, 202
 browser compatibility, 50
 caption-side, 202
 color, 52, 200
 empty-cells, 202
 float, 203
 font-family, 193
 font-size, 194
 font-style, 194
 font-variant, 195
 font-weight, 195
 letter-spacing, 196
 line-height, 197
 links, 51
 list-style-image, 203
 list-style-type, 203
 margin, 204
 padding, 204
 text, 47-48
 text-align, 197
 text-decoration, 195
 text-indent, 198
 text-transform, 196
 vertical-align, 197
 word-spacing, 196
 resources, 56
 rules, 40-41
 selectors, 40

<style> tag, 184
 text, formatting, 46-49
 color, 51
 links, 50-51
 properties, 195
 typography, 196-198
 XML templates, 161-163

D

data
 analyzing, 160
 forms, receiving, 114-115
 <dd> tag, 62, 176
 declarations (style sheets), 40
 defaults
 browser text, 46
 drop-down menus, 110
 definition lists, 62
 definition terms tag (<dt>), 62
 deleting underlines, 51
 deprecation of tags, 22
 designs
 color, 126-128
 content, 124
 fonts, 126-128
 images, 129-132
 layout, 124-125
 navigation, 125
 paper versus online, 122-124
 DHTML (Dynamic HTML), 135-136
 dir attribute, 190
 <div> tag, 173
 <dl> tag, 176
 <!DOCTYPE> tag, 14-15
 Document Type Definitions (DTDs), 159
 documents
 HTML
 commands, 14
 previewing, 13
 reading, 6
 required elements, 12
 saving, 13
 style sheets, 22
 XHTML, 14-16

Dreamweaver, 146-149
 drop-down menus, 110-111
 <dt> tag, 62, 177
 DTDs (Document Type Definitions), 159
 Dynamic HTML (DHTML), 135-136

E

editing text, 167
 elements
 active, 134
 ActiveX, 137-138
 DHTML, 135-136
 Java, 137-138
 embedded content, 182-184
 forms, 184-189
 frames, 180-182
 HTML documents, 12
 links, 177-178
 lists, 176-177
 margins, 55
 required, 170-172
 scripting, 189
 style sheets, 184
 tables, 178-180
 text, 173-176
 tag, 174
 email hyperlinks, creating, 34
 <embed> tag, 117-120
 embedding
 content, 182-184
 style sheets, 39, 44
 empty cells, 202
 empty-cells property, 202
 equations (math), 25
 etiquette (images), 79-80
 events, 140, 190
 expiration dates, 30
 eXtensible Hypertext Markup Language. *See* XHTML
 eXtensible Markup Language. *See* XML

F

<family name> tag, 193
 fields (forms)
 check boxes, 109-110
 drop-down menus, 110-111
 file browse boxes, 112
 radio buttons, 109-110
 Submit/Reset buttons, 113
 text areas/boxes, 108-109
 file browse boxes, 112
 finding plug-ins, 120
 float property, 203
 font-family property, 48, 51, 193
 font-size property, 48, 194
 font-style property, 48, 194
 font-variant property, 195
 font-weight property, 48, 195
 fonts
 alternate, 127
 appearances, 195
 designs, 126-128
 families, 193
 headings, 23
 size, 194
 style sheets, 193-194
 weight, 195
 foreign (non-English) text, 26-27
 <form> tag, 105, 185
 formatting
 bulleted lists, 58-59
 color, 51
 links, 50-51
 numbered lists, 60
 style sheets, 22
 tables, 65-67
 tags, 30
 text, 21-22, 46-49
 Web pages, 8
 forms
 data, receiving, 114-115
 fields
 check boxes, 109-110
 drop-down menus, 110-111
 file browse boxes, 112
 radio buttons, 109-110
 Submit/Reset buttons, 113
 text areas/boxes, 108-109

<form> tag, 105, 184-189
 forward slash (/), 92
 HTML tag pairs, 13
 hyperlinks, 34
 <frame> tag, 92-93, 104, 180
 frameborder attribute, 93
 frames, 14
 active, 102
 advantages, 103
 borders, 93
 disadvantages
 frameset URL, 100
 printing, 101-103
 Why Frames Suck (Most of the Time) Web site, 100
 <frame> tag, 92-93, 104, 180
 <frameset> tag, 91-92, 104, 181
 framesets, 89-91
 linking, 98-99
 names, 92
 nesting, 95-97
 <noframes> tag, 93-94, 104, 182
 orientation, 91
 overview, 89
 printing, 103
 resizing, 93
 scrolling, 93
 size, 91-92
 tags, list of, 180-182
 target, 99
 Frameset <!DOCTYPE> tag
 variation, 15
 <frameset> tag, 91-92, 104, 181
 framesets
 defined, 89
 example, 90-91
 URLs, 100
 FreeWebs Web site, 152
 FrontPage, 143-146
 Fusion, 149

G

<generic family> tag, 193
 GIFs (Graphics Interface Format), 71
 GoLive, 149
 Google Advertising Programs Web site, 155

graphics. *See* images
 grave accent (`), 24
 greater than (>) symbol, 24
 Greek entities, list of, 210-212

H

<h1> tag, 23, 174
 <h6> tag, 23, 174
 </head> tag, 12
 <head> tag, 12, 172
 heading tags, 23
 headings, 23, 64
 height attribute, 75-76
 hidden attribute, 119
 highlighting hyperlinks, 33
 HomeSite, 149
 horizontal lines, adding, 52-55
 hosting (Web), 151-152
 <hr> tag, 52, 56, 175
 href attribute, 33, 36
 .htm files, 13
 HTML (Hypertext Markup Language), 6
 defined, 6
 Dynamic (DHTML), 135-136
 new standards, 35
 overview, 6
 resources, 168
 XML, compared, 158-159
 </html> tag, 12
 <html> tag, 12, 16, 172
 HTTP (Hypertext Transfer Protocol), 6
 hyperlinks, 32
 <a> tag, 32
 anchors, 35-37
 email, 34
 files, 33-34
 formatting, 50-51
 highlighting, 33
 images as, 78-79
 opening new windows, 34
 redirecting, 29
 slashes, 34
 tags, 38, 177-178
 testing, 167

text, 87-88
 underlining, 51
 Web pages, 33

Hypertext Markup Language. *See*
 HTML

Hypertext Transfer Protocol
 (HTTP), 6

I

I Have A Dream speech Web site, 116

<i> tag, 175

IBM Web site, 69

id attribute, 190

<a> tag, 35

<frame> tag, 92

<iframe> tag, 96-97, 104, 181

image maps

client-side/server-side, 85-87

coordinates, 83-85

defined, 82

examples, 82

shapes, 84

text links, 87-88

images

adding, 71

aligning with text, 76

backgrounds, 130-132, 199

borders, 200-203

browser compatibility, 74

bulleted lists, 203

copying, 80

designs, 129

etiquette, 79-80

faster loading, 76

GIFs, 71

JPEGs, 71

as links, 78-79

loading speed, 129

PNGs, 71

size, 75-76

text pop-ups, 73-74

 tag, 71

 tag (attributes), 182

align, 76

alt, 74

height/width, 75-76

usemap, 87

inclusions, 182-184

indenting text, 198

indexes, 152-153

inline style sheets, 40, 45

<input> tag, 108, 186

Interactive Advertising Bureau Web
 site, 157

international text, 26-27

internationalization attributes, 190

Internet

ARPAnet, 5

connecting to, 9

history and emergence, 5

IP, 5

ISPs, 9

overview, 5

Web hosts, 10

WPPs, 9

intranets, 10

IP (Internet Protocol), 5

ISPs (Internet Service Providers), 9

italic text, 21

J-K

Java, 137-138

JavaScript, 138-140

Johnson's Baby Soft Web site, 83

JPEGs (Joint Photographic Experts
 Group), 71

Kerning text, 196

L

<label> tag, 187

lang attribute, 26, 190

languages

DHTML, 135-136

international, 26-27

Java, 137-138

JavaScript, 138-140

markup, 7

XHTML, 7

attribute quotes, 165

basic principles, 17-18

documents, 14-16

style sheets, 166

syntax, 164-165
tags. See XHTML, tags
 XML, 7
data, analyzing, 160
HTML, compared,
158-159
Resource Center Web
site, 168
resources, 168
standards, 160
style sheet templates,
161-163
tags, 166
 layout, 124-125
 style sheet properties, 204
 tables for, 68-69
 <legend> tag, 187
 <length> tag, 197
 less than (<) symbol, 24
 letter-spacing property, 196
 tag, 57, 177
 line-height property, 197
 lines
 adding, 52-55
 breaks, 20-21
 linking
 frames, 98-99
 style sheets, 40, 44-45
 links. *See* hyperlinks
 list item tag (), 57
 The List Web site, 152
 list-style-image property, 203
 list-style-type property, 203
 lists
 bulleted, 57
 formatting, 58-59
 images, 203
 definition, 62
 numbered, 59-61
 style sheet properties, 203-204
 tags, list of, 176-177
 types, 203
 loading images, 129
 loop attribute, 119
 looping sound/video, 119
 Lynx, 8

M

Macromedia Dreamweaver, 146-149
 <map> tag, 87, 183
 mapping images
 client-side/server-side, 85-87
 coordinates, 83-85
 defined, 82
 examples, 82
 shapes, 84
 text links, 87-88
 margin property, 204
 margin-left property, 55
 margin-right property, 55
 margin-top property, 55
 <margin-width> tag, 204
 marginheight attribute, 93
 margins, 55, 204
 marginwidth attribute, 93
 markup languages, 7
 mathematical notations, 25-26
 menus
 drop-down, 110-111
 navigational, 136
 meta information, 27
 expiration dates, 30
 refreshing pages, 29
 scripts, 139
 searches, 28-29
 spiders, 153
 splash pages, 29
 <meta> tag, 27-29, 153, 172
 method attribute, 105
 Microsoft
 FrontPage, 143-146
 Web sites, 69
 CSS tutorial, 56
 DHTML, 135
 FrontPage, 145, 152
 typography, 48
 minus (-) symbol, 24

N

name attribute, 92
 names
 anchors, 37
 frames, 92

navigation designs, 125, 136
 nesting
 frames, 95-97
 tags, 166
 NetMechanic Web site, 163
 NetObjects Fusion, 149
 Netscape
 DHTML Web site, 135
 plug-in archive Web site, 120
 networks
 ARPAnet, 5
 intranets, 10
 new standards (HTML), 35
 Nielsen, Jakob, 100
 <no-repeat> tag, 200
 <noembed> tag, 118-120
 <noframes> tag, 93-94, 104, 182
 non-breaking space (sp;) special
 character, 25
 noresize attribute, 93
 <noscript> tag, 189
 number (#) symbol, 24
 numbered lists, 59-61

O

<object> tag
 embedding applications, 137
 inclusions, 183
 sound/video, 119-120
 tag, 59-61, 177
 onclick events, 140, 190
 ondblclick events, 140, 190
 OnKeyDown events, 190
 OnKeyPress events, 190
 OnKeyUp events, 190
 online designs, 122-124
 onload events, 140
 OnMouseDown events, 190
 onmousemove events, 140, 190
 onmouseout events, 140, 190
 onmouseover events, 140, 190
 OnMouseUp events, 190
 onreset events, 140
 onsubmit events, 140
 onunload events, 140
 <option> tag, 110, 187

ordered list tag (), 59-61, 177
 ordered lists, 59-61
 organizing
 tables, 64
 Web pages
 bulleted lists, 57-59
 definition lists, 62
 numbered lists, 59-61

P

</p> tag, 19
 <p> tag, 19, 174
 padding property, 204
 padding Web pages, 204
 Paint Shop Pro Web site, 83
 paper designs, 122-124
 paragraphs
 creating, 19-20
 tags, 19, 173-175
 percent (%) symbol, 24, 92
 <percentage> tag, 198
 pixels, 67
 plug-ins, 120
 plus (+) symbol, 24
 PNGs (Portable Network
 Graphics), 71
 <pre> tag, 174
 precedence (style sheets), 46
 previewing HTML documents, 13
 printing frames, 101-103
 properties
 background, 47, 52
 background-color, 51, 199
 background-image, 199
 background-position, 199
 background-repeat, 200
 border-color, 201
 border-style, 201
 border-width, 202
 browser compatibility, 50
 caption-side, 202
 color, 47, 51-52, 200
 empty-cells, 202
 float, 203
 font-family, 48, 51, 193
 font-size, 48, 194

- font-style, 48, 194
- font-variant, 195
- font-weight, 48, 195
- letter-spacing, 196
- line-height, 197
- links, 51
- list-style-image, 203
- list-style-type, 203
- margin, 204
- margin-left, 55
- margin-right, 55
- margin-top, 55
- padding, 204
- text, 47-48
- text-align, 48, 197
- text-decoration, 48, 51, 195
- text-indent, 48, 198
- text-transform, 196
- vertical-align, 197
- word-spacing, 196

Properties inspector
(Dreamweaver), 147

protocols

- HTTP, 6
- IP, 5

Q-R

quotes (“ ”), 165

radio buttons, 109-110

reading HTML documents, 6

receiving form data, 114-115

redirecting hyperlinks, 29

refresh attribute, 29

refreshing Web pages, 29

Register FrontPage Hosts Web site, 152

registered trademark (®) symbol, 24

<relative-weight> tag, 195

<repeat> tag, 200

<repeat-x> tag, 200

<repeat-y> tag, 200

repeating backgrounds, 200

required elements (documents), 170

HTML

- <body> & </body> tags, 12, 171
- <!DOCTYPE> tag, 171

- <head> & </head> tags, 12, 172
- <html> & </html> tags, 12, 172
- <meta> tag, 172
- <title> & </title> tags, 12, 172

XHTML, 14-16

Reset buttons, 113

resizing

- frames, 93
- images, 76

rewinding sound/video, 119

robots, 153-154

rows (tables), 64

rows attribute, 91

rowspan attribute, 66-68

rules (style sheets), 40-41

S

saving HTML documents, 13

scientific notations, 25-26

<script> tag, 138, 189

scripting, 138-140, 189

scrolling attribute, 93

scrolling frames, 93

search engines

- defined, 152

- indexes, 153

- meta information, 28-29

- spiders, 153-154

- Web sites, adding, 155-156

<select> tag, 110, 187

selecting. *See* choosing

selectors, 40

- a:active, 51

- a:hover

- a:link, 50

- a:visited, 50

- links, formatting, 51

server-side images, 85

size

- fonts

- headings, 23*

- style sheets, 194*

- frames, 91-92

- images, 75-76
 - pixels, 67
- <small> tag, 176
- sound
 - adding, 116-119
 - looping, 119
 - plug-ins, 120
 - rewinding, 119
 - starting, 119
- source code, 14
- special characters, 24-25, 212-213
- spiders, 153-154
- splash pages, 29
- src attribute
 - <frame> tag, 92
 - tag, 71
- start values (numbered lists), 61
- starting sound/video, 119
- storing Web pages, 9
- Strict <!DOCTYPE> tag
 - variation, 15
- tag, 174
- style attribute, 66, 190
- style sheets. *See* CSS
- <style> tag, 48, 56, 184
- styles
 - borders, 201
 - fonts, 194
- <sub> tag, 25, 175
- Submit buttons, 113
- subscript tag (<sub>), 25, 175
- <sup> tag, 25, 175
- superscript numbers, 24
- superscript tag (<sup>), 25, 175
- symbols, 24-25
 - Greek entities, list of, 210-212
 - list of, 205-207, 212-213
- syntax, checking, 164-165

T

- <table> tag, 64, 178
- tables, 64
 - alignment, 66
 - background color, 66
 - borders, 66, 69, 200-203
 - captions, 202

- cell spacing, 66
- cells, 64
- complex, 68
- empty cells, 202
- formatting, 65-67
- headings, 64
- organizing, 64
- page layout, 68-69
- rows, 64
- style attributes, 66
- tags, list of, 69, 178-180
- width, 66
- tags, 6. *See also* names of specific tags
 - deprecation, 22
 - XHTML
 - case sensitivity, 17
 - <!DOCTYPE>, 14-15
 - <html>, 16
 - required, 18
- target attribute, 34, 99
- target frames, 99
- <td> tag, 64, 179
- testing
 - code, 163
 - colors, 52
 - links, 167
 - tag syntax, 164-165
- text
 - alignment, 76, 197
 - borders, 200-203
 - browser default, 46
 - editing, 167
 - formatting, 21-22, 175-176. *See also* CSS
 - forms, 108-109
 - headings, 23
 - image map links, 87-88
 - indenting, 198
 - international, 26-27
 - kerning, 196
 - line breaks, 20
 - lists
 - bulleted, 57-59
 - definition, 62
 - images as bullets, 203

numbered, 59-61
style sheet properties,
 203-204
types, 203
 mathematical notations, 25-26
 paragraphs, 19-20, 173-175
 phrases, 173-175
 pop-ups, 73-74
 scientific notations, 25-26
 special characters, 24-25
 typography, 196-198
 word spacing, 196
 wrapping, 20
 text-align property, 48, 197
 text-decoration property, 48, 51, 195
 text-indent property, 48, 198
 text-transform property, 196
 <textarea> tag, 109, 188
 <th> tag, 64, 179
 thumbnails, 78-79
 Thumbs Plus, 79
 title attribute, 190
 </title> tag, 12
 <title> tag, 12, 172
 tools (Web authoring)
 BBEdit, 149
 Dreamweaver, 146-149
 FrontPage, 143-146
 Fusion, 149
 GoLive, 149
 HomeSite, 149
 <tr> tag, 64, 180
 Transitional <!DOCTYPE> tag
 variation, 15
 Travel Alberta Web site, 82
 Tripod Web site, 152
 <tt> tag, 176
 type attribute
 <input> tag, 108
 <script> tag, 139
 typography, 196-198

U

 tag, 57, 177
 underlining links, 51
 underscore (_), 17

unordered list tag (), 57, 177
 unordered lists. *See* bulleted lists
 URIs (Uniform Resource
 Identifiers), 32
 URLs (Uniform Resource
 Locators), 32
 copying, 33
 framesets, 100
 usemap attribute, 87

V

valign attribute, 66
 value attribute, 61
 VBScript, 138-140
 vertical-align property, 197
 video
 adding, 116-119
 looping, 119
 plug-ins, 120
 rewinding, 119
 starting, 119
 viewing
 commands, 14
 documents, 13
 images, 74
 source code, 14
 Web pages, 8-9

W

W3C (World Wide Web
 Consortium), 6
 CSS properties, 56
 HTML and XHTML
 Specifications, 168
 math notation products, 26
 schools, 168
 style sheet recommendations, 192
 table attributes, 66
 XML 1.0 Recommendation, 168
 WANs (Wide Area Networks), 5
 Web
 authoring tools
 BBEdit, 149
 Dreamweaver, 146,
 148-149

- FrontPage*, 143, 145-146
- Fusion*, 149
- GoLive*, 149
- HomeSite*, 149
- browsers. *See* browsers
- hosts, 10, 151-152
- Web Developer
 - CSS tutorial Web site, 56
 - JavaScript samples, 139
 - Journal compatibility table Web site, 141
- Web Hosting Buying Guide Web site, 152
- Web pages
 - active, 134
 - ActiveX*, 137-138
 - DHTML*, 135-136
 - Java*, 137-138
 - addresses, 32-33, 100
 - color, testing, 52
 - creating, 11
 - expiration dates, 30
 - formatting, 8
 - frames, 14
 - layout
 - style sheet properties*, 204
 - tables for*, 68-69
 - margins, 204
 - organizing
 - bulleted lists*, 57-59
 - definition lists*, 62
 - numbered lists*, 59-61
 - padding, 204
 - refreshing, 29
 - storing, 9
 - viewing, 8-9
- Web Presence Providers (WPPs), 9
- Web sites
 - ActiveX downloads, 138
 - Add Me, 156
 - adding to search engines, 155-156
 - Adobe GoLive, 149
 - Amazon, 82
 - American Idol, 69
 - AnyBrowser, 9
 - BBEdit, 149
 - color charts, 126
 - CNET
 - plug-ins*, 120
 - Web Hosting Buying Guide*, 152
 - CNN, 69
 - FreeWebs, 152
 - Google Advertising Programs, 155
 - HomeSite, 149
 - I Have a Dream speech, 116
 - IBM, 69
 - Interactive Advertising Bureau, 157
 - Java, 138
 - Johnson's Baby Soft, 83
 - language support, 26
 - The List, 152
 - Microsoft, 69
 - CSS tutorial*, 56
 - DHTML*, 135
 - FrontPage*, 145
 - Register FrontPage Hosts*, 152
 - typography*, 48
 - NetMechanic, 163
 - NetObjects Fusion, 149
 - Netscape
 - DHTML*, 135
 - plug-in archive*, 120
 - Paint Shop Pro, 83
 - tables for page layout
 - examples, 69
 - Thumbs Plus, 79
 - Travel Alberta, 82
 - Tripod, 152
 - W3C
 - CSS properties*, 56
 - HTML and XHTML Specifications*, 168
 - math notation products*, 26
 - schools*, 168
 - style sheet recommendations*, 192
 - table attributes*, 66
 - XML 1.0 Recommendation*, 168

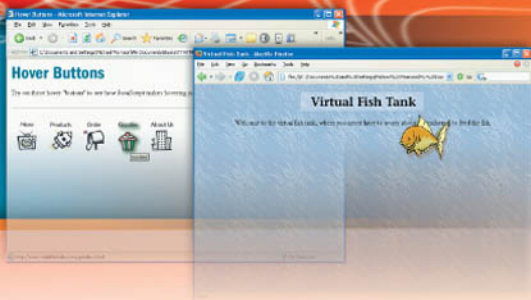
Web Developer
 CSS tutorial, 56
 JavaScript samples, 139
 Journal compatibility table, 141
 Webmaster Stop style sheet
 properties, 50
 WebSideStory StatMarket®, 98
 Why Frames Suck (Most of the Time), 100
 XML Resource Center, 168
 XML, Java, and the Future of the Web, 168
 Yahoo!
 Geocities, 152
 Suggest a Site, 155
 Webmaster Stop style sheet
 properties Web site, 50
 WebSideStory StatMarket® Web site, 98
 weight (fonts), 195
 <weight> tag, 195
 Weinmann, Lynda, 126
 What You See Is What You Get (WYSIWYG), 11
 whitespace, 124
 Why Frames Suck (Most of the Time) Web site, 100
 Wide Area Networks (WANs), 5
 width
 borders, 202
 tables, 66
 width attribute
 <embed> tag, 118
 tag, 75-76
 <table> tag, 66
 windows, opening, 34
 word processors, 11
 word-spacing property, 196
 World Wide Web Consortium.
 See W3C
 WPPs (Web Presence Providers), 9
 wrapping text, 20
 WYSIWYG (What You See Is What You Get), 11

X-Z

XHTML (eXtensible Hypertext Markup Language), 7
 attribute quotes, 165
 basic principles, 17-18
 documents, 14-16
 style sheets, 166
 syntax, 164-165
 tags
 case sensitivity, 17
 <!DOCTYPE>, 14-15
 <html>, 16
 nesting, 166
 XML (eXtensible Markup Language), 7, 158
 data, analyzing, 160
 HTML, compared, 158-159
 Resource Center Web site, 168
 resources, 168
 standards, 160
 style sheet templates, 161-163
 tags, 166
 XML, Java, and the Future of the Web site, 168
 Yahoo!
 Geocities Web site, 152
 Suggest a Site Web site, 155

SAMS
Teach
Yourself

FOURTH EDITION
Covers JavaScript 1.5
and Ajax



JavaScript™

Michael Moncur

in **24**
Hours

**SAMS
Teach
Yourself**

JavaScript

in **24**
Hours

Michael Moncur

SAMS

800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself JavaScript in 24 Hours

Copyright © 2007 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32879-8

Library of Congress Catalog Card Number: 2005909315

Printed in the United States of America

First Printing: July 2006

09 08 07 06 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Acquisitions Editor
Betsy Brown

Development Editor
Songlin Qiu

Managing Editor
Patrick Kanouse

Senior Project Editor
Matthew Purcell

Copy Editor
Jessica McCarty

Indexer
Tim Wright

Proofreader
Carla Lewis

Technical Editor
Jim O'Donnell

Publishing Coordinator
Vanessa Evans

Book Designer
Gary Adair

Page Layout
TnT Design, Inc.

Contents at a Glance

| | |
|--------------------|---|
| Introduction | 1 |
|--------------------|---|

Part I: Introducing the Concept of Web Scripting and the JavaScript Language

| | |
|--|----|
| HOURL 1 Understanding JavaScript | 7 |
| 2 Creating Simple Scripts | 23 |
| 3 Getting Started with JavaScript Programming | 37 |
| 4 Working with the Document Object Model (DOM) | 49 |

Part II: Learning JavaScript Basics

| | |
|---|-----|
| HOURL 5 Using Variables, Strings, and Arrays | 63 |
| 6 Using Functions and Objects | 85 |
| 7 Controlling Flow with Conditions and Loops | 101 |
| 8 Using Built-in Functions and Libraries | 121 |

Part III: Learning More About the DOM

| | |
|---|-----|
| HOURL 9 Responding to Events | 139 |
| 10 Using Windows and Frames | 157 |
| 11 Getting Data with Forms | 173 |
| 12 Working with Style Sheets | 191 |
| 13 Using the W3C DOM | 207 |
| 14 Using Advanced DOM Features | 219 |

Part IV: Working with Advanced JavaScript Features

| | |
|--|-----|
| HOURL 15 Unobtrusive Scripting | 235 |
| 16 Debugging JavaScript Applications | 255 |
| 17 AJAX: Remote Scripting | 273 |
| 18 Greasemonkey: Enhancing the Web with JavaScript | 293 |

Part V: Building Multimedia Applications with JavaScript

| | |
|--|-----|
| HOURL 19 Using Graphics and Animation | 313 |
| 20 Working with Sound and Plug-ins | 329 |

Part VI: Creating Complex Scripts

| | | |
|-----------------|---|-----|
| HOURL 21 | Building JavaScript Drop-down Menus | 345 |
| 22 | Creating a JavaScript Game | 359 |
| 23 | Creating JavaScript Applications | 377 |
| 24 | Your Future with JavaScript | 393 |

Part VII: Appendixes

| | | |
|----------|---------------------------------------|-----|
| A | Other JavaScript Resources | 409 |
| B | Tools for JavaScript Developers | 411 |
| C | Glossary | 415 |
| D | JavaScript Quick Reference | 419 |
| E | DOM Quick Reference | 427 |
| | Index | 433 |

Table of Contents

Part I: Introducing the Concept of Web Scripting and the JavaScript Language

| | |
|--|-----------|
| HOURL 1: Understanding JavaScript | 7 |
| Learning Web Scripting Basics | 7 |
| How JavaScript Fits into a Web Page | 9 |
| Browsers and JavaScript | 12 |
| Specifying JavaScript Versions | 15 |
| JavaScript Beyond the Browser | 16 |
| Exploring JavaScript's Capabilities | 16 |
| Alternatives to JavaScript | 17 |
| HOURL 2: Creating Simple Scripts | 23 |
| Tools for Scripting | 23 |
| Displaying Time with JavaScript | 25 |
| Beginning the Script | 26 |
| Adding JavaScript Statements | 26 |
| Creating Output | 27 |
| Adding the Script to a Web Page | 28 |
| Testing the Script | 29 |
| HOURL 3: Getting Started with JavaScript Programming | 37 |
| Basic Concepts | 37 |
| JavaScript Syntax Rules | 42 |
| Using Comments | 43 |
| Best Practices for JavaScript | 44 |
| HOURL 4: Working with the Document Object Model (DOM) | 49 |
| Understanding the Document Object Model (DOM) | 49 |
| Using Window Objects | 51 |
| Working with Web Documents | 52 |
| Accessing Browser History | 55 |
| Working with the Location Object | 55 |

Part II: Learning JavaScript Basics

| | |
|--|------------|
| HOURL 5: Using Variables, Strings, and Arrays | 63 |
| Using Variables | 63 |
| Understanding Expressions and Operators | 67 |
| Data Types in JavaScript | 68 |
| Converting Between Data Types | 69 |
| Using String Objects | 70 |
| Working with Substrings | 74 |
| Using Numeric Arrays | 76 |
| Using String Arrays | 77 |
| Sorting a Numeric Array | 79 |
| HOURL 6: Using Functions and Objects | 85 |
| Using Functions | 85 |
| Introducing Objects | 90 |
| Using Objects to Simplify Scripting | 91 |
| Extending Built-in Objects | 94 |
| HOURL 7: Controlling Flow with Conditions and Loops | 101 |
| The if Statement | 102 |
| Using Shorthand Conditional Expressions | 105 |
| Testing Multiple Conditions with If and Else | 105 |
| Using Multiple Conditions with switch | 107 |
| Using for Loops | 109 |
| Using While Loops | 111 |
| Using Do...While Loops | 112 |
| Working with Loops | 112 |
| Looping Through Object Properties | 114 |
| HOURL 8: Using Built-in Functions and Libraries | 121 |
| Using the Math Object | 121 |
| Working with Math Functions | 123 |
| Using the with Keyword | 125 |

| | |
|-----------------------------------|-----|
| Working with Dates | 126 |
| Using Third-Party Libraries | 128 |
| Other Libraries | 130 |

Part III: Learning More About the DOM

| | |
|---|------------|
| Hour 9: Responding to Events | 139 |
| Understanding Event Handlers | 139 |
| Using Mouse Events | 144 |
| Using Keyboard Events | 149 |
| Using the onLoad and onUnload Events | 151 |
| Hour 10: Using Windows and Frames | 157 |
| Controlling Windows with Objects | 157 |
| Moving and Resizing Windows | 160 |
| Using Timeouts | 162 |
| Displaying Dialog Boxes | 164 |
| Working with Frames | 166 |
| Hour 11: Getting Data with Forms | 173 |
| The Basics of HTML Forms | 173 |
| Using the Form Object with JavaScript | 174 |
| Scripting Form Elements | 176 |
| Displaying Data from a Form | 182 |
| Sending Form Results by Email | 184 |
| Hour 12: Working with Style Sheets | 191 |
| Style and Substance | 191 |
| Defining and Using CSS Styles | 192 |
| Using CSS Properties | 195 |
| Creating a Simple Style Sheet | 198 |
| Using External Style Sheets | 200 |
| Controlling Styles with JavaScript | 201 |

Sams Teach Yourself JavaScript in 24 Hours

| | |
|--|------------|
| HOURL 13: Using the W.3C DOM | 207 |
| The DOM and Dynamic HTML | 207 |
| Understanding DOM Structure | 208 |
| Creating Positionable Elements (Layers) | 210 |
| HOURL 14: Using Advanced DOM Features | 219 |
| Working with DOM Nodes | 219 |
| Hiding and Showing Objects | 222 |
| Modifying Text Within a Page | 223 |
| Adding Text to a Page | 225 |
|
Part IV: Working with Advanced JavaScript Features | |
| HOURL 15: Unobtrusive Scripting | 235 |
| Scripting Best Practices | 235 |
| Reading Browser Information | 242 |
| Cross-Browser Scripting | 245 |
| Supporting Non-JavaScript Browsers | 247 |
| HOURL 16: Debugging JavaScript Applications | 255 |
| Avoiding Bugs | 255 |
| Basic Debugging Tools | 258 |
| Creating Error Handlers | 260 |
| Advanced Debugging Tools | 263 |
| HOURL 17: AJAX: Remote Scripting | 273 |
| Introducing AJAX | 273 |
| Using XMLHttpRequest | 277 |
| Creating a Simple AJAX Library | 279 |
| Creating an AJAX Quiz Using the Library | 280 |
| Debugging AJAX Applications | 285 |
| HOURL 18: Greasemonkey: Enhancing the Web with JavaScript | 293 |
| Introducing Greasemonkey | 293 |
| Working with User Scripts | 296 |
| Creating Your Own User Scripts | 299 |

Part V: Building Multimedia Applications with JavaScript

| | |
|--|------------|
| HOURL 19: Using Graphics and Animation | 313 |
| Using Dynamic Images | 313 |
| Creating Rollovers | 315 |
| A Simple JavaScript Slideshow | 319 |
| HOURL 20: Working with Sound and Plug-Ins | 329 |
| Introducing Plug-Ins | 329 |
| JavaScript and Flash | 332 |
| Playing Sounds with JavaScript | 333 |
| Testing Sounds in JavaScript | 336 |

Part VI: Creating Complex Scripts

| | |
|--|------------|
| HOURL 21: Building JavaScript Drop-Down Menus | 345 |
| Designing Drop-Down Menus | 345 |
| Scripting Drop-Down Menu Behavior | 350 |
| HOURL 22: Creating a JavaScript Game | 359 |
| About the Game | 359 |
| Creating the HTML Document | 361 |
| Creating the Script | 363 |
| Adding Style with CSS | 368 |
| HOURL 23: Creating JavaScript Applications | 377 |
| Creating a Scrolling Window | 377 |
| Style Sheet Switching with JavaScript | 380 |
| HOURL 24: Your Future with JavaScript | 393 |
| Learning Advanced JavaScript Techniques | 393 |
| Future Web Technologies | 394 |
| Planning for the Future | 397 |
| Moving on to Other Languages | 398 |

Part VII: Appendixes

| | |
|--|------------|
| APPENDIX A: Other JavaScript Resources | 409 |
| Other Books | 409 |
| JavaScript Websites | 409 |
| Web Development Sites | 410 |
| This Book's Website | 410 |
| APPENDIX B: Tools for JavaScript Developers | 411 |
| HTML and Text Editors | 411 |
| HTML Validators | 413 |
| Debugging Tools | 413 |
| APPENDIX C: Glossary | 415 |
| APPENDIX D: JavaScript Quick Reference | 419 |
| Built-in Objects | 419 |
| Creating and Customizing Objects | 423 |
| JavaScript Statements | 424 |
| JavaScript Built-in Functions | 426 |
| APPENDIX E: DOM Quick Reference | 427 |
| DOM Level 0 | 427 |
| DOM Level 1 | 429 |
| Index | 433 |

About the Author

Michael Moncur is a freelance webmaster and author. He runs a network of websites, including the Web's oldest site about famous quotations, online since 1994. He wrote *Sams Teach Yourself DHTML in 24 Hours*, and has also written several bestselling books about networking, certification programs, and databases. He lives with his wife in Salt Lake City, Utah.

Dedication

To my family, and especially Laura. Thanks for all your love and support.

Acknowledgments

I'd like to thank everyone at Sams for their help with this book, and for the opportunity to write it. In particular, Betsy Brown got this edition started and kept it moving. Songlin Qiu managed the development of the book. Project editor Matt Purcell handled the editing process, and the copy editor, Jessica McCarty, saved me from many embarrassing errors. The technical reviewer, Jim O'Donnell, painstakingly tested the scripts and helped keep the writing grounded in reality.

I am grateful to everyone involved with previous editions of this book, including Scott Meyers, David Mayhew, Sean Medlock, Susan Hobbs, Michelle Wyner, Jeff Schultz, Amy Patton, George Nedeff, and Phil Karras. I'd also like to thank Neil Salkind and the rest of the team at Studio B for their help throughout this project.

Finally, personal thanks go to my wife, Laura; my parents, Gary and Susan Moncur; the rest of the family; and my friends, particularly Chuck Perkins, Matt Strebe, Cory Storm, Robert Parsons, Dylan Winslow, Ray Jones, Tyson Jensen, Curt Siffert, Richard Easlick, and Henry J. Tillman. I couldn't have done it without your support.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: `webdev@sampublishing.com`

Mail: Mark Taber
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.sampublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

The World Wide Web began as a simple repository for information, but it has grown into much more—it entertains, teaches, advertises, and communicates. As the Web has evolved, the tools have also evolved. Simple markup tools such as HTML have been joined by true programming languages—including JavaScript.

Now don't let the word "programming" scare you. For many, the term conjures up images of long nights staring at the screen, trying to remember which sequence of punctuation marks will produce the effect you need. (Don't get me wrong—some of us enjoy that sort of thing.)

Although JavaScript is programming, it's a very simple language. As a matter of fact, if you haven't programmed before, it makes a great introduction to programming. It requires very little knowledge to start programming with JavaScript—you'll write your first program in Hour 2, "Creating Simple Scripts."

If you can create a web page with HTML, you can easily use JavaScript to improve a page. JavaScript programs can range from a single line to a full-scale application. In this book, you'll start with simple scripts, and proceed to complex applications, such as a card game. You'll also explore some of the most recent uses of JavaScript, such as AJAX remote scripting.

If you've spent much time developing pages for the Web, you know that the Web is constantly changing, and it can be hard to keep up with the latest languages and tools. This book will help you add JavaScript to your web development toolbox, and I think you'll enjoy learning it.

JavaScript and Web Standards

When JavaScript first appeared in browsers, it had rather limited capabilities, and JavaScript programmers have always pushed the envelope to take maximum advantage of what the language was capable of. Unfortunately, this resulted in some bad practices, such as scripts that only worked in one browser, and JavaScript gained a bit of a bad reputation.

Now, thanks to wide browser support for standards established by the W3C (World Wide Web Consortium) and new technologies such as AJAX, JavaScript's future is looking brighter than ever, and a new, more responsible style of scripting is gaining favor. Unobtrusive scripting focuses on adding interactive features while keeping the HTML simple and standards-compliant.

Throughout this book, you'll learn the best practices for using JavaScript responsibly and following web standards. All of the examples in this book avoid browser-specific techniques in favor of standard techniques, and all of the examples will work in most modern browsers.

How to Use This Book

This book is divided into 24 lessons. Each covers a single JavaScript topic, and should take about an hour to complete. The lessons start with the basics of JavaScript, and continue with more advanced topics. You can study an hour a day, or whatever pace suits you. (If you choose to forego sleep and do your studying in a single 24-hour period, you might have what it takes to be a computer book author.)

Organization of This Book

This book is divided into six parts, each focusing on one area of JavaScript:

- ▶ Part I, “Introducing the Concept of Web Scripting and the JavaScript Language,” introduces JavaScript, describes how it fits in with other languages, and explains the basic language features of JavaScript. It also introduces the DOM (Document Object Model), which connects JavaScript to web documents.
- ▶ Part II, “Learning JavaScript Basics,” covers the fundamentals of the JavaScript language: variables, functions, objects, loops and conditions, and built-in functions. You'll also learn about third-party libraries that add functionality to JavaScript.
- ▶ Part III, “Learning More About the DOM,” digs deeper into the DOM objects you'll use in nearly every JavaScript program. It covers events, windows, and web forms. You'll also learn about CSS style sheets, and the DOM features that enable you to change styles. Finally, you'll learn about the W3C DOM, which enables you to modify any part of a page using JavaScript.
- ▶ Part IV, “Working with Advanced JavaScript Features,” begins with a look at unobtrusive scripting techniques to keep JavaScript from intruding on the functionality and validity of HTML documents. You'll also learn how to debug JavaScript applications, and finally take a look at two cutting-edge JavaScript features: AJAX and Greasemonkey.
- ▶ Part V, “Building Multimedia Applications with JavaScript,” describes JavaScript's features for working with graphics, animation, sound, and browser plug-ins.
- ▶ Part VI, “Creating Complex Scripts,” focuses on helping you create complete JavaScript applications. You'll learn how to create drop-down menus, a card game written in JavaScript, and other examples. In the last hour, you'll learn about what's in store for JavaScript and what other languages you might want to learn next.

Conventions Used in This Book

This book contains special elements as described by the following:

These boxes highlight information that can make your JavaScript programming more efficient and effective.

***Did you
Know?***

These boxes provide additional information related to material you just read.

***By the
Way***

These boxes focus your attention on problems or side effects that can occur in specific situations.

***Watch
Out!***

A special monospace font is used on programming-related terms and language.

Try It Yourself



The Try It Yourself section at the end of each chapter guides you through the process of creating your own script or applying the techniques learned throughout the hour. This will help you create practical applications of JavaScript based on what you've learned. ▲

Q&A, Quiz, and Exercises

At the end of each hour's lesson, you'll find three final sections. Q&A answers a few of the most common questions about the hour's topic. The Quiz tests your knowledge of the skills you learned in that hour, and the Exercises offer ways for you to gain more experience with the techniques the hour covers.

This Book's Website

Because JavaScript and the Web are constantly changing, you'll need to stay up-to-date after reading this book. This book's website includes the latest updates as well as downloadable versions of the listings and graphics for the examples used in this book. To access the book's website, register your book at <http://www.sampublishing.com/register>.

The Author's Website

The author of this book, Michael Moncur, maintains a website about JavaScript at <http://www.jsworkshop.com/>. There you'll find regular updates on the JavaScript language and the DOM, links to script examples, and detailed tutorial articles.

If you have questions or comments about this book, have noticed an error, or have trouble getting one of the scripts to work, you can also reach the author by email at js4@starlingtech.com. (Please check the website first to see if your question has been answered.)

PART I:

Introducing the Concept of Web scripting and the JavaScript Language

| | | |
|----------------|--|-----------|
| HOURL 1 | Understanding JavaScript | 7 |
| HOURL 2 | Creating Simple Scripts | 23 |
| HOURL 3 | Getting Started with JavaScript Programming | 37 |
| HOURL 4 | Working with the Document Object Model (DOM) | 49 |

This page intentionally left blank

HOUR 1

Understanding JavaScript

What You'll Learn in This Hour:

- ▶ What web scripting is and what it's good for
- ▶ How scripting and programming are different (and similar)
- ▶ What JavaScript is and where it came from
- ▶ How to include JavaScript commands in a web page
- ▶ How different browsers handle JavaScript
- ▶ What JavaScript can do for your web pages
- ▶ How to choose between JavaScript and alternative languages

The World Wide Web (WWW) began as a text-only medium—the first browsers didn't even support images within web pages. Although it's still not quite ready to give television a run for its money, the Web has come a long way since then.

Today's websites can include a wealth of features: graphics, sounds, animation, video, and occasionally useful content. Web scripting languages, such as JavaScript, are one of the easiest ways to spice up a web page and to interact with users in new ways.

The first hour of this book introduces the concept of web scripting and the JavaScript language. It also describes how JavaScript fits in with other web languages.

Learning Web Scripting Basics

In the world of science fiction movies (and many other movies that have no excuse), computers are often seen obeying commands in English. Although this might indeed happen in the near future, computers currently find it easier to understand languages such as BASIC, C, and Java.

If you know how to use HTML (Hypertext Markup Language) to create a web document, you've already worked with one computer language. You use HTML tags to describe how

you want your document formatted, and the browser obeys your commands and shows the formatted document to the user.

Because HTML is a simple text markup language, it can't respond to the user, make decisions, or automate repetitive tasks. Interactive tasks such as these require a more sophisticated language: a programming language, or a *scripting* language.

Although many programming languages are complex, scripting languages are generally simple. They have a simple syntax, can perform tasks with a minimum of commands, and are easy to learn. Web scripting languages enable you to combine scripting with HTML to create interactive web pages.

Scripts and Programs

A movie or a play follows a script—a list of actions (or lines) for the actors to perform. A web script provides the same type of instructions for the web browser. A script in JavaScript can range from a single line to a full-scale application. (In either case, JavaScript scripts usually run within a browser.)

By the Way

Is JavaScript a scripting language or a programming language? It depends on who you ask. We'll refer to scripting throughout this book, but feel free to include JavaScript programming on your résumé after you've finished this book.

Some programming languages must be *compiled*, or translated, into machine code before they can be executed. JavaScript, on the other hand, is an *interpreted* language: The browser executes each line of script as it comes to it.

There is one main advantage to interpreted languages: Writing or changing a script is very simple. Changing a JavaScript script is as easy as changing a typical HTML document, and the change is enacted as soon as you reload the document in the browser.

By the Way

Interpreted languages have their disadvantages—they can't execute really quickly, so they're not ideally suited for complicated work, such as graphics. Also, they require the interpreter (in JavaScript's case, usually a browser) in order to work.

Introducing JavaScript

JavaScript was developed by Netscape Communications Corporation, the maker of the Netscape web browser. JavaScript was the first web scripting language to be supported by browsers, and it is still by far the most popular.

A bit of history: JavaScript was originally called LiveScript and was first introduced in Netscape Navigator 2.0 in 1995. It was soon renamed JavaScript to indicate a marketing relationship with Sun's Java language.

**By the
Way**

JavaScript is almost as easy to learn as HTML, and it can be included directly in HTML documents. Here are a few of the things you can do with JavaScript:

- ▶ Display messages to the user as part of a web page, in the browser's status line, or in alert boxes
- ▶ Validate the contents of a form and make calculations (for example, an order form can automatically display a running total as you enter item quantities)
- ▶ Animate images or create images that change when you move the mouse over them
- ▶ Create ad banners that interact with the user, rather than simply displaying a graphic
- ▶ Detect the browser in use or its features and perform advanced functions only on browsers that support them
- ▶ Detect installed plug-ins and notify the user if a plug-in is required
- ▶ Modify all or part of a web page without requiring the user to reload it
- ▶ Display or interact with data retrieved from a remote server

You can do all this and more with JavaScript, including creating entire applications. We'll explore the uses of JavaScript throughout this book.

How JavaScript Fits into a Web Page

As you hopefully already know, HTML is the language you use to create web documents. To refresh your memory, Listing 1.1 shows a short but sadly typical web document.

LISTING 1.1 A Simple HTML Document

```
<html>
<head>
<title>Our Home Page</title>
</head>
<body>
<h1>The American Eggplant Society</h1>
<p>Welcome to our Web page. Unfortunately,
it's still under construction.</p>
</body>
</html>
```

This document consists of a header within the `<head>` tags and the body of the page within the `<body>` tags. To add JavaScript to a page, you'll use a similar tag:

`<script>`.

The `<script>` tag tells the browser to start treating the text as a script, and the closing `</script>` tag tells the browser to return to HTML mode. In most cases, you can't use JavaScript statements in an HTML document except within `<script>` tags. The exception is event handlers, described later in this hour.

JavaScript and HTML

Using the `<script>` tag, you can add a short script (in this case, just one line) to a web document, as shown in Listing 1.2.

Did you Know?

If you want to try this example in a browser but don't want to type it, the HTML document is available on this book's website (as are all of the other listings).

LISTING 1.2 A Simple HTML Document with a Simple Script

```
<html>
<head>
<title>Our Home Page</title>
</head>
<body>
<h1>The American Eggplant Society</h1>
<p>Welcome to our Web page. Unfortunately,
it's still under construction.
We last worked on it on this date:
<script language="JavaScript" type="text/javascript">
document.write(document.lastModified);
</script>
</p>
</body>
</html>
```

JavaScript's `document.write` statement, which you'll learn more about later, sends output as part of the web document. In this case, it displays the modification date of the document.

By the Way

Notice that the `<script>` tag in Listing 1.2 includes the parameter `type="text/javascript"`. This specifies the scripting language to the browser. You can also specify a JavaScript version, as you'll learn later in this hour.

In this example, we placed the script within the body of the HTML document. There are actually four different places where you might use scripts:

- ▶ **In the body of the page**—In this case, the script's output is displayed as part of the HTML document when the browser loads the page.
- ▶ **In the header of the page between the `<head>` tags**—Scripts in the header don't immediately affect the HTML document, but can be referred to by other scripts. The header is often used for functions—groups of JavaScript statements that can be used as a single unit. You will learn more about functions in Hour 3, "Getting Started with JavaScript Programming."
- ▶ **Within an HTML tag, such as `<body>` or `<form>`**—This is called an *event handler* and enables the script to work with HTML elements. When using JavaScript in event handlers, you don't need to use the `<script>` tag. You'll learn more about event handlers in Hour 3.
- ▶ **In a separate file entirely**—JavaScript supports the use of files with the `.js` extension containing scripts; these can be included by specifying a file in the `<script>` tag.

Using Separate JavaScript Files

When you create more complicated scripts, you'll quickly find your HTML documents become large and confusing. To avoid this, you can use one or more external JavaScript files. These are files with the `.js` extension that contain JavaScript statements.

External scripts are supported by all modern browsers. To use an external script, you specify its filename in the `<script>` tag:

```
<script language="JavaScript" type="text/javascript" src="filename.js">
</script>
```

Because you'll be placing the JavaScript statements in a separate file, you don't need anything between the opening and closing `<script>` tags—in fact, anything between them will be ignored by the browser.

You can create the `.js` file using a text editor. It should contain one or more JavaScript commands, and only JavaScript—don't include `<script>` tags, other HTML tags, or HTML comments. Save the `.js` file in the same directory as the HTML documents that refer to it. See the Try It Yourself section of Hour 2 for an example of separate HTML and script files.

External JavaScript files have a distinct advantage: You can link to the same `.js` file from two or more HTML documents. Because the browser stores this file in its cache, this can reduce the time it takes your web pages to display.

***Did you
Know?***

Events

Many of the useful things you can do with JavaScript involve interacting with the user, and that means responding to *events*—for example, a link or a button being clicked. You can define event handlers within HTML tags to tell the browser how to respond to an event. For example, Listing 1.3 defines a button that displays a message when clicked.

LISTING 1.3 A Simple Event Handler

```
<html>
<head>
<title>Event Test</title>
</head>
<body>
<h1>Event Test</h1>
<button onclick="alert('You clicked the button.')">
</body>
</html>
```

In Hour 9, “Responding to Events,” you’ll learn more about JavaScript’s event model and creating simple and complex event handlers.

By the Way

You can also use an external script to define event handlers. This is a good practice because it lets you keep all of your JavaScript in one place, rather than scattered across the HTML document. See Hour 9 for details.

Browsers and JavaScript

Like HTML, JavaScript requires a web browser to be displayed, and different browsers may display it differently. Unlike HTML, the results of a browser incompatibility with JavaScript are more drastic: Rather than simply displaying your text incorrectly, the script may not execute at all, may display an error message, or may even crash the browser.

We’ll take a quick look at the way different browsers—and different versions of the same browser—treat JavaScript in the following sections.

The DOM (Document Object Model)

Let’s start with one reason you shouldn’t have to think too much about different browsers. Almost everything you do with JavaScript involves working with the Document Object Model (DOM)—a standardized set of objects that represent a web document.

The DOM includes objects that enable you to work with all aspects of the current document. For example, you can read the value the user types in a form field, or the filename of the current page.

The DOM is defined by the W3C (World Wide Web Consortium) and the latest browsers support DOM levels 1 and 2, which enable you to control all parts of a web page with JavaScript.

Early versions of the DOM only allowed JavaScript to manipulate certain parts of a page—such as form elements and links. The new DOM enables you to work with every element defined in HTML.

***Did you
Know?***

Internet Explorer

Microsoft's Internet Explorer (IE) browser was a latecomer to the Internet, but has now become the most popular browser. The latest versions of IE support most of JavaScript 1.5 and the W3C DOM.

At this writing, IE 6.0 is the latest released version, and IE 7.0 is in beta. Although most of the examples in this book will work in IE 5.0 and later, I recommend testing your scripts with the latest browsers.

Netscape and Firefox

Netscape, which for a time made the Web's most popular browser, established the Mozilla Foundation to maintain an open-source version of the browser. This led to the Mozilla browser and more recently, Firefox, a streamlined browser based on the Mozilla engine.

Firefox has recently begun to challenge Microsoft's browser dominance, with an estimated 10% of web users. That might not sound like many, but ignoring Firefox means ignoring at least 10% of your audience, and on many sites the percentage is much higher.

Firefox is available for Windows, Macintosh, and Linux platforms and is free, open-source software. You can download Firefox from the Mozilla website at <http://www.mozilla.org/>.

At this writing, the current version of Firefox is 1.5. Most of the scripts in this book will work with Firefox 1.0 or later, as well as versions 6 and 7 of the Netscape browser.

**By the
Way**

Netscape 4.0 and Internet Explorer 4.0 supported incompatible versions of Dynamic HTML (DHTML)—an attempt to overcome the limits of the current DOM. The new W3C DOM eliminates the need for these proprietary models, and you can now write standard code that will work on most modern browsers.

Other Browsers

Although Internet Explorer and Firefox are the most popular browsers, there are many other browsers. Here are two less-common browsers you'll probably hear about:

- ▶ Safari, Apple's browser, is included with MacOS and is the default browser on most Macintosh computers.
- ▶ Opera, from Opera Software, is an alternative browser notable for its support of many platforms, including mobile phones. The latest version of Opera, 8.0, supports the W3C DOM and JavaScript 1.5, and should work with most scripts in this book.

**Did you
Know?**

There are many other browsers out there, but you don't need to know all of them to create working scripts—as long as you follow the standards, your scripts will work on browsers that support JavaScript almost every time. This book will focus on teaching standards-based scripting that will work in all modern browsers.

Versions of JavaScript

The JavaScript language has evolved since its original release in Netscape 2.0. There have been several versions of JavaScript:

- ▶ JavaScript 1.0, the original version, is supported by Netscape 2.0 and Internet Explorer 3.0.
- ▶ JavaScript 1.1 is supported by Netscape 3.0 and mostly supported by Internet Explorer 4.0.
- ▶ JavaScript 1.2 is supported by Netscape 4.0 and partially supported by Internet Explorer 4.0.
- ▶ JavaScript 1.3 is supported by Netscape 4.5 and Internet Explorer 5.0 and 6.0.
- ▶ JavaScript 1.5 is partially supported by Internet Explorer 6.0, and supported by Netscape 6.0 and Firefox 1.0.
- ▶ JavaScript 1.6 is currently supported by Firefox 1.5.

Each of these versions is an improvement over the previous version and includes a number of new features. With rare exception, browsers that support the new version will also support scripts written for earlier versions.

The European Computer Manufacturing Association (ECMA) has finalized the ECMA-262 specification for ECMAScript, a standardized version of JavaScript. JavaScript 1.3 follows the ECMA-262 standard, and JavaScript 1.5 follows ECMA-262 revision 3.

Another language you might hear of is JScript. This is how Microsoft refers to its implementation of JavaScript, which is generally compatible with the standard version.

***By the
Way***

The Mozilla Foundation, the open-source offshoot of Netscape that develops the Firefox browser, is also working with ECMA on JavaScript 2.0, a future version that will correspond with the fourth edition of the ECMAScript standard. JavaScript 2.0 will improve upon earlier versions with a more modular approach, better object support, and features to make JavaScript useful as a general-purpose scripting language as well as a web language.

Specifying JavaScript Versions

As mentioned earlier in this hour, you can specify a version of JavaScript in the `<script>` tag. For example, this tag specifies JavaScript version 1.3:

```
<script language="JavaScript1.3" type="text/javascript">
```

There are two ways of specifying the JavaScript language in the `<script>` tag. The old method uses the `language` attribute, and the new method recommended by the HTML 4.0 specification uses the `type` attribute. To maintain compatibility with older browsers, you can use both attributes.

When you specify a version number in the `language` attribute, this allows your script to execute only if the browser supports the version you specified or a later version.

When the `<script>` tag doesn't specify a version number, all browsers that support JavaScript will run the script. Because most of the JavaScript language has remained the same since version 1.0, you will rarely need to worry about JavaScript versions.

In most cases, you shouldn't specify a JavaScript version at all. This allows your script to run on all of the browsers that support JavaScript. You should only specify a particular version when your script uses features unique to a specific version.

***Did you
Know?***

JavaScript Beyond the Browser

Although JavaScript programs traditionally run within a web browser, and web-based JavaScript is the focus of this book, JavaScript is becoming increasingly popular in other applications. Here are a few examples:

- ▶ Adobe Dreamweaver and Flash, used for web applications and multimedia, can be extended with JavaScript.
- ▶ Several server-side versions of JavaScript are available. These run within a web server rather than a browser.
- ▶ Microsoft's Windows Scripting Host (WSH) supports JScript, Microsoft's implementation of JavaScript, as a general-purpose scripting language for Windows. Unfortunately, the most popular applications developed for WSH so far have been email viruses.
- ▶ Microsoft's Common Language Runtime (CLR), part of the .NET framework, supports JavaScript.

Along with these examples, many of the changes in the upcoming JavaScript 2.0 are designed to make it more suitable as a general-purpose scripting language.

Exploring JavaScript's Capabilities

If you've spent any time browsing the Web, you've undoubtedly seen lots of examples of JavaScript in action. Here are some brief descriptions of typical applications for JavaScript, all of which you'll explore further, later in this book.

Improving Navigation

Some of the most common uses of JavaScript are in navigation systems for websites. You can use JavaScript to create a navigation tool—for example, a drop-down menu to select the next page to read, or a submenu that pops up when you hover over a navigation link.

When it's done right, this kind of JavaScript interactivity can make a site easier to use, while remaining usable for browsers that don't support JavaScript.

Validating Forms

Form validation is another common use of JavaScript. A simple script can read values the user types into a form and make sure they're in the right format, such as with ZIP Codes or phone numbers. This allows users to notice common errors and

fix them without waiting for a response from the web server. You'll learn how to write form validation scripts in Hour 11, "Getting Data with Forms."

Special Effects

One of the earliest and most annoying uses of JavaScript was to create attention-getting special effects—for example, scrolling a message in the browser's status line or flashing the background color of a page.

These techniques have fortunately fallen out of style, but thanks to the W3C DOM and the latest browsers, some more impressive effects are possible with JavaScript—for example, creating objects that can be dragged and dropped on a page, or creating fading transitions between images in a slideshow.

Remote Scripting (AJAX)

For a long time, the biggest limitation of JavaScript was that there was no way for it to communicate with a web server. For example, you could use it to verify that a phone number had the right number of digits, but not to look up the user's location in a database based on the number.

Now that some of JavaScript's advanced features are supported by most browsers, this is no longer the case. Your scripts can get data from a server without loading a page, or send data back to be saved. These features are collectively known as AJAX (Asynchronous JavaScript And XML), or *remote scripting*. You'll learn how to develop AJAX scripts in Hour 17, "AJAX: Remote Scripting."

You've seen AJAX in action if you've used Google's Gmail mail application, or recent versions of Yahoo! Mail or Microsoft Hotmail. All of these use remote scripting to present you with a responsive user interface that works with a server in the background.

Alternatives to JavaScript

JavaScript is not the only language used on the Web, and in some cases, it may not be the right tool for the job. Other languages, such as Java, can do some things better than JavaScript. In the following sections, we'll look at a few other commonly used web languages and their advantages.

Java

Java is a programming language developed by Sun Microsystems that can be used to create *applets*, or programs that execute within a web page.

Java is a compiled language, but the compiler produces code for a *virtual machine* rather than a real computer. The virtual machine is a set of rules for bytecodes and their meanings, with capabilities that fit well into the scope of a web browser.

The virtual machine code is then interpreted by a web browser. This allows the same Java applet to execute the same way on PCs, Macintoshes, and UNIX machines, and on different browsers.

By the Way

Java is also a densely populated island in Indonesia and a slang term for coffee. This has resulted in a widespread invasion of coffee-related terms in computer literature.

At this point, we need to make one thing clear: Java is a fine language, but you won't be learning it in this book. Although their names and some of their commands are similar, JavaScript and Java are entirely different languages.

ActiveX

ActiveX is a specification developed by Microsoft that enables ordinary Windows programs to be run within a web page. ActiveX programs can be written in languages such as Visual C++ and Visual Basic, and they are compiled before being placed on the web server.

ActiveX applications, called *controls*, are downloaded and executed by the web browser, like Java applets. Unlike Java applets, controls can be installed permanently when they are downloaded, eliminating the need to download them again.

ActiveX's main advantage is that it can do just about anything. This can also be a disadvantage: Several enterprising programmers have already used ActiveX to bring exciting new capabilities to web pages, such as "the web page that turns off your computer" and "the web page that formats your disk drive."

Fortunately, ActiveX includes a signature feature that identifies the source of the control and prevents controls from being modified. Although this won't prevent a control from damaging your system, you can specify which sources of controls you trust.

ActiveX has two main disadvantages: First, it isn't as easy to program as a scripting language or Java. Second, ActiveX is proprietary—it works only in Microsoft Internet Explorer, and only under Windows platforms.

VBScript

VBScript, sometimes known as Visual Basic Scripting Edition, is Microsoft's answer to JavaScript. Just as JavaScript's syntax is loosely based on Java, VBScript's syntax is

loosely based on Microsoft Visual Basic, a popular programming language for Windows machines.

Like JavaScript, VBScript is a simple scripting language, and you can include VBScript statements within an HTML document. VBScript can work with the DOM in the same way as JavaScript. To begin a VBScript script, you use the `<script LANGUAGE="VBScript">` tag.

VBScript can do many of the same things as JavaScript, and it even looks similar in some cases. It has two main advantages:

- ▶ For those who already know Visual Basic, it may be easier to learn than JavaScript.
- ▶ It is closely integrated with ActiveX, Microsoft's standard for web-embedded applications.

VBScript's main disadvantage is that it is supported only by Microsoft Internet Explorer. JavaScript, on the other hand, is supported by Netscape, Internet Explorer, and several other browsers. JavaScript is a much more popular language, and you can see it in use all over the Web.

CGI and Server-Side Scripting

CGI (Common Gateway Interface) is not really a language, but a specification that enables programs to run on web servers. CGI programs can be written in any number of languages, including Perl, C, and Visual Basic.

Along with traditional CGI, scripting languages such as Microsoft's Active Server Pages, Java Server Pages, Cold Fusion, and PHP are often used on web servers. A server-side implementation of JavaScript is also available.

Server-side programs are heavily used on the Web. Almost every time you type information into a form and press a button to send it to a website, the data is processed by a server-side application.

The main difference between JavaScript and server-side languages is that JavaScript applications execute on the client (the web browser) and server-side applications execute on the web server. The main disadvantage of this approach is that, because the data must be sent to the web server and back, response time might be slow.

On the other hand, CGI can do things JavaScript can't do. In particular, it can read and write files on the server and interact with other server components, such as databases. Although a client-side JavaScript program can read information from a form and then manipulate it, it can't store the data on the web server.

JavaScript is often used in conjunction with server-side languages. In its simplest form, this means JavaScript handles client-side chores such as form validation, whereas a server-side language receives data and stores it in a database. Using AJAX, this interaction can be instantaneous and does not even require loading a new page.

***Did you
Know?***

CGI and server-side programming are outside the focus of this book. You can learn more about these technologies with other Sams books, including *Teach Yourself CGI Programming in 24 Hours*, *Teach Yourself Perl in 24 Hours*, and *Teach Yourself PHP in 24 Hours*. See Appendix A, “Other JavaScript Resources,” for more sources of information.

Summary

During this hour, you’ve learned what web scripting is and what JavaScript is. You’ve also learned how to insert a script into an HTML document or refer to an external JavaScript file, what sorts of things JavaScript can do, and how JavaScript differs from other web languages.

If you’re waiting for some real JavaScript code, look no further. The next hour, “Creating Simple Scripts,” guides you through the process of creating several working JavaScript examples. You’ll also learn about the tools you’ll need to work with JavaScript.

Q&A

Q. *Do I need to test my JavaScript on more than one browser?*

A. In an ideal world, any script you write that follows the standards for JavaScript will work in all browsers, and 90% of the time that’s true in the real world. But browsers do have their quirks, and you should test your scripts on Internet Explorer and Firefox at a minimum.

Q. *If I plan to learn Java or CGI anyway, will I have any use for JavaScript?*

A. Certainly. JavaScript is the ideal tool for many applications, such as form validation. Although Java and CGI have their uses, they can’t do all that JavaScript can do.

Q. *Are there browsers out there that don’t support JavaScript?*

A. Yes. A few niche browsers, such as text-based browsers and tools for blind users, have partial JavaScript support or no support. Mobile phone browsers

often support little or no JavaScript. Finally, many users of Internet Explorer or Firefox have JavaScript support turned off, and some corporate firewalls and ad-blocking software block JavaScript. Hour 2 describes how to account for browsers that don't support JavaScript.

Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

1. Why do JavaScript and Java have similar names?
 - a. JavaScript is a stripped-down version of Java.
 - b. Netscape's marketing department wanted them to sound related.
 - c. They both originated on the island of Java.
2. When a user views a page containing a JavaScript program, which machine actually executes the script?
 - a. The user's machine running a web browser
 - b. The web server
 - c. A central machine deep within Netscape's corporate offices
3. Which of the following languages is supported by both Microsoft Internet Explorer and Netscape?
 - a. VBScript
 - b. ActiveX
 - c. JavaScript

Quiz Answers

1. b. Although some of the syntax is similar, JavaScript got its Java-based name mostly because of a marketing relationship.
2. a. JavaScript programs execute on the web browser. (There is actually a server-side version of JavaScript, but that's another story.)
3. c. JavaScript is supported by both Netscape and Internet Explorer, although the implementations are not identical.

Exercises

If you want to learn a bit about JavaScript or check out the latest developments before you proceed with the next hour, perform these activities:

- ▶ Visit this book's website to check for news about JavaScript and updates to the scripts in this book.
- ▶ View some of the examples on this book's website to see JavaScript in action.

HOOR 2

Creating Simple Scripts

What You'll Learn in This Hour:

- ▶ The software tools you will need to create and test scripts
- ▶ Beginning and ending scripts
- ▶ Formatting JavaScript statements
- ▶ How a script can display a result
- ▶ Including a script within a web document
- ▶ Testing a script using browsers
- ▶ Modifying a script
- ▶ Dealing with errors in scripts
- ▶ Moving scripts into separate files

As you learned in Hour 1, “Understanding JavaScript,” JavaScript is a scripting language for web pages. You can include JavaScript commands directly in the HTML document, and the script will be executed when the page is viewed in a browser.

During this hour, you will create a simple script, edit it, and test it using a web browser. Along the way you'll learn the basic tasks involved in creating and using scripts.

Tools for Scripting

Unlike many programming languages, you won't need any special software to create JavaScript scripts. In fact, you probably already have everything you need.

Text Editors

The first tool you'll need to work with JavaScript is a *text editor*. JavaScript scripts are stored in simple text files, usually as part of HTML documents. Any editor that can store ASCII text files will work.

You can choose from a wide range of editors, from simple text editors to word processors. If you don't have a favorite editor already, a simple editor is most likely included with your computer. For Windows computers, the Notepad accessory will work just fine.

Watch Out!

If you use a word processor to create JavaScript programs, be sure you save the files as ASCII text rather than as word processing documents. Otherwise, the browser might not recognize them.

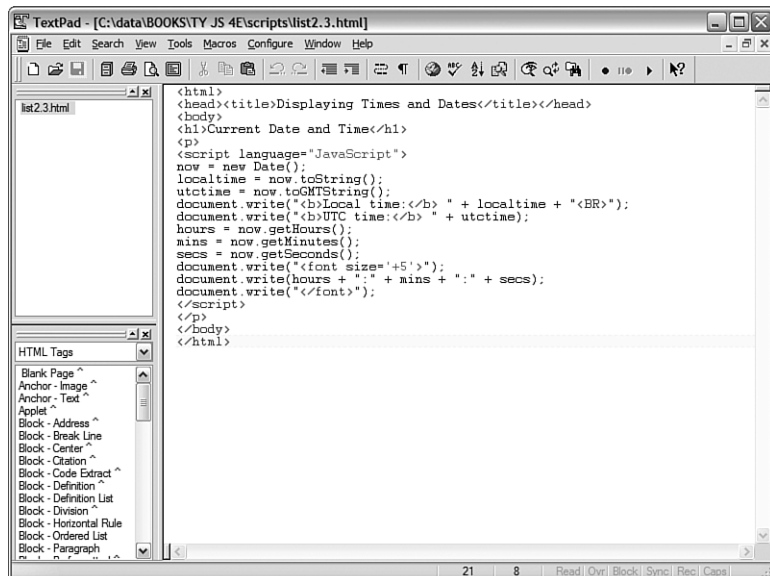
A variety of dedicated HTML editors is also available and will work with JavaScript. In fact, many include features specifically for JavaScript—for example, color-coding the various JavaScript statements to indicate their purposes, or even creating simple scripts automatically.

For Windows computers, here are a few recommended editors:

- ▶ **HomeSite**—An excellent HTML editor that includes JavaScript support. HomeSite is included as part of Adobe Dreamweaver and is also available separately.
- ▶ **Microsoft FrontPage 2003**—Microsoft's visual HTML editor. The Script Builder component enables you to easily create simple scripts.
- ▶ **TextPad**—A powerful text editor that includes a number of features missing from Notepad. TextPad's view of a JavaScript document is shown in Figure 2.1.

FIGURE 2.1

A text editor (TextPad) with a JavaScript document.



The following editors are available for both Windows and Macintosh:

- ▶ **Adobe Dreamweaver**—A visually oriented editor that works with HTML, JavaScript, and Macromedia’s Flash plug-in.
- ▶ **Adobe GoLive**—A visual and HTML editor that also includes features for designing and organizing the structure of large sites.

Additionally for the Macintosh, BBEdit, TextWrangler, and Alpha are good HTML editors that you can use to create web pages and scripts.

Appendix B, “Tools for JavaScript Developers,” includes web addresses to download these and other HTML and JavaScript editors.

**By the
Way**

Browsers

You’ll need two other things to work with JavaScript: a web browser and a computer to run it on. Because this book covers new features introduced up to JavaScript 1.5 and the latest W3C DOM, I recommend that you use the latest version of Mozilla Firefox or Microsoft Internet Explorer. See the Mozilla (<http://www.mozilla.com>) or Microsoft (<http://www.microsoft.com>) website to download a copy.

At a minimum, you should have Firefox 1.0, Netscape 7.0, or Internet Explorer 6.0 or later. Although Netscape 4.x and Internet Explorer 4 will run many of the scripts in this book, they don’t support a lot of the latest features you’ll learn about.

You can choose whichever browser you like for your web browsing, but for developing JavaScript you should have more than one browser—at a minimum, Firefox and Internet Explorer. This will allow you to test your scripts in the common browsers users will employ on your site.

If you plan on making your scripts available over the Internet, you’ll also need a web server, or access to one. However, you can use most of the JavaScript examples in this book directly from your computer’s hard disk.

**By the
Way**

Displaying Time with JavaScript

One common and easy use for JavaScript is to display dates and times. Because JavaScript runs on the browser, the times it displays will be in the user’s current time zone. However, you can also use JavaScript to calculate “universal” (UTC) time.

**By the
Way**

UTC stands for Universal Time (Coordinated), and is the atomic time standard based on the old GMT (Greenwich Mean Time) standard. This is the time at the Prime Meridian, which runs through Greenwich, London, England.

As a basic introduction to JavaScript, you will now create a simple script that displays the current time and the UTC time within a web page.

Beginning the Script

Your script, like most JavaScript programs, begins with the HTML `<script>` tag. As you learned in Hour 1, you use the `<script>` and `</script>` tags to enclose a script within the HTML document.

**Watch
Out!**

Remember to include only valid JavaScript statements between the starting and ending `<script>` tags. If the browser finds anything but valid JavaScript statements within the `<script>` tags, it will display a JavaScript error message.

To begin creating the script, open your favorite text editor and type the beginning and ending `<script>` tags as shown.

```
<script LANGUAGE="JavaScript" type="text/javascript">
</script>
```

Because this script does not use any of the new features of JavaScript 1.1 or later, you won't need to specify a version number in the `<script>` tag. This script should work with all browsers going back to Netscape 2.0 or Internet Explorer 3.0.

Adding JavaScript Statements

Your script now needs to determine the local and UTC times, and then display them to the browser. Fortunately, all of the hard parts, such as converting between date formats, are built in to the JavaScript interpreter.

Storing Data in Variables

To begin the script, you will use a *variable* to store the current date. You will learn more about variables in Hour 5, "Using Variables, Strings, and Arrays." A variable is a container that can hold a value—a number, some text, or in this case, a date.

To start writing the script, add the following line after the first `<script>` tag. Be sure to use the same combination of capital and lowercase letters in your version because JavaScript commands and variable names are case sensitive.

```
now = new Date();
```

This statement creates a variable called `now` and stores the current date and time in it. This statement and the others you will use in this script use JavaScript's built-in `Date` object, which enables you to conveniently handle dates and times. You'll learn more about working with dates in Hour 8, "Using Built-in Functions and Libraries."

Notice the semicolon at the end of the previous statement. This tells the browser that it has reached the end of a statement. Semicolons are optional, but using them helps you avoid some common errors. We'll use them throughout this book for clarity.

**By the
Way**

Calculating the Results

Internally, JavaScript stores dates as the number of milliseconds since January 1, 1970. Fortunately, JavaScript includes a number of functions to convert dates and times in various ways, so you don't have to figure out how to convert milliseconds to day, date, and time.

To continue your script, add the following two statements before the final `</script>` tag:

```
localtime = now.toString();  
utctime = now.toGMTString();
```

These statements create two new variables: `localtime`, containing the current time and date in a nice readable format, and `utctime`, containing the UTC equivalent.

The `localtime` and `utctime` variables store a piece of text, such as January 1, 2001 12:00 PM. In programming parlance, a piece of text is called a *string*. You will learn more about strings in Hour 5.

**By the
Way**

Creating Output

You now have two variables—`localtime` and `utctime`—which contain the results we want from our script. Of course, these variables don't do us much good unless we can see them. JavaScript includes a number of ways to display information, and one of the simplest is the `document.write` statement.

The `document.write` statement displays a text string, a number, or anything else you throw at it. Because your JavaScript program will be used within a web page, the output will be displayed as part of the page. To display the result, add these statements before the final `</script>` tag:

```
document.write("<b>Local time:</b> " + localtime + "<br>");  
document.write("<b>UTC time:</b> " + utctime);
```

These statements tell the browser to add some text to the web page containing your script. The output will include some brief strings introducing the results, and the contents of the `localtime` and `utctime` variables.

Notice the HTML tags, such as ``, within the quotation marks—because JavaScript's output appears within a web page, it needs to be formatted using HTML. The `
` tag in the first line ensures that the two times will be displayed on separate lines.

By the Way

Notice the plus signs (+) used between the text and variables in the previous statements. In this case, it tells the browser to combine the values into one string of text. If you use the plus sign between two numbers, they are added together.

Adding the Script to a Web Page

You should now have a complete script that calculates a result and displays it. Your listing should match Listing 2.1.

LISTING 2.1 The Complete Date and Time Script

```
<script language="JavaScript" type="text/javascript">
now = new Date();
localtime = now.toString();
utctime = now.toGMTString();
document.write("<b>Local time:</b> " + localtime + "<BR>");
document.write("<b>UTC time:</b> " + utctime);
</script>
```

To use your script, you'll need to add it to an HTML document. In its most basic form, the HTML document should include opening and closing `<html>` tags, `<head>` tags, and `<body>` tags.

If you add these tags to the document containing your script along with a descriptive heading, you should end up with something like Listing 2.2.

LISTING 2.2 The Date and Time Script in an HTML Document

```
<html>
<head><title>Displaying Times and Dates</title></head>
<body>
<h1>Current Date and Time</h1>
<p>
<script language="JavaScript" type="text/javascript">
now = new Date();
localtime = now.toString();
utctime = now.toGMTString();
document.write("<b>Local time:</b> " + localtime + "<BR>");
```

```
document.write("<b>UTC time:</b> " + utctime);  
</script>  
</p>  
</body>  
</html>
```

Now that you have a complete HTML document, save it with the .htm or .html extension.

Notepad and other Windows text editors might try to be helpful and add the .txt extension to your script. Be sure your saved file has the correct extension.

**By the
Way**

Testing the Script

To test your script, you simply need to load the HTML document you created in a web browser. Start Netscape or Internet Explorer and select Open from the File menu. Click the Choose File or Browse button, and then find your HTML file. After you've selected it, click the Open button to view the page.

If you typed the script correctly, your browser should display the result of the script, as shown in Figure 2.2. (Of course, your result won't be the same as mine, but it should be the same as the setting of your computer's clock.)

A note about Internet Explorer 6.0 and above: Depending on your security settings, the script might not execute, and a yellow highlighted bar on the top of the browser might display a security warning. In this case, click the yellow bar and select Allow Blocked Content to allow your script to run. (This happens because the default security settings allow JavaScript in online documents, but not in local files.)

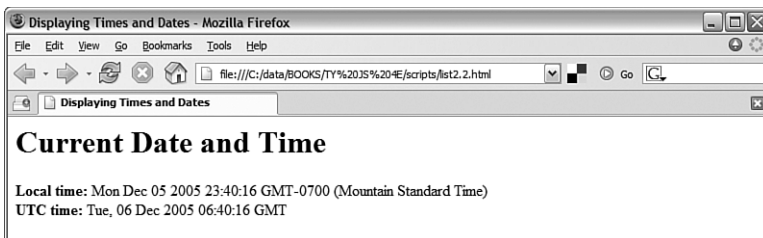


FIGURE 2.2
Firefox displays the results of the Date and Time script.

You can download the HTML document for this hour from this book's website. If the version you type doesn't work, try downloading the online version.

**Did you
Know?**

Modifying the Script

Although the current script does indeed display the current date and time, its display isn't nearly as attractive as the clock on your wall or desk. To remedy that, you can use some additional JavaScript features and a bit of HTML to display a large clock.

To display a large clock, we need the hours, minutes, and seconds in separate variables. Once again, JavaScript has built-in functions to do most of the work:

```
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
```

These statements load the hours, mins, and secs variables with the components of the time using JavaScript's built-in date functions.

After the hours, minutes, and seconds are in separate variables, you can create `document.write` statements to display them:

```
document.write("<h1>");
document.write(hours + ":" + mins + ":" + secs);
document.write("</h1>");
```

The first statement displays an HTML `<h1>` header tag to display the clock in a large typeface. The second statement displays the hours, mins, and secs variables, separated by colons, and the third adds the closing `</h1>` tag.

You can add the preceding statements to the original date and time script to add the large clock display. Listing 2.3 shows the complete modified version of the script.

LISTING 2.3 The Date and Time Script with Large Clock Display

```
<html>
<head><title>Displaying Times and Dates</title></head>
<body>
<h1>Current Date and Time</h1>
<p>
<script language="JavaScript">
now = new Date();
localtime = now.toString();
utctime = now.toGMTString();
document.write("<b>Local time:</b> " + localtime + "<BR>");
document.write("<b>UTC time:</b> " + utctime);
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
document.write("<h1>");
document.write(hours + ":" + mins + ":" + secs);
document.write("</h1>");
</script>
</p>
</body>
</html>
```

Now that you have modified the script, save the HTML file and open the modified file in your browser. If you left the browser running, you can simply use the Reload button to load the new version of the script. Try it and verify that the same time is displayed in both the upper portion of the window and the new large clock. Figure 2.3 shows the results.

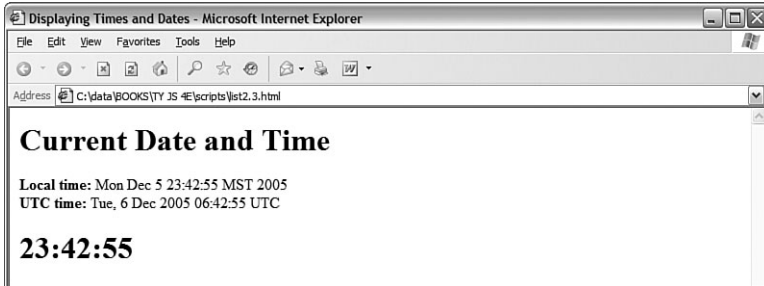


FIGURE 2.3
Internet Explorer displays the modified Date and Time script.

The time formatting produced by this script isn't perfect: Hours after noon are in 24-hour time, and there are no leading zeroes, so 12:04 is displayed as 12:4. See Hour 8, "Using Built-in Functions and Libraries," for solutions to these issues.

**By the
Way**

Dealing with JavaScript Errors

As you develop more complex JavaScript applications, you're going to run into errors from time to time. JavaScript errors are usually caused by mistyped JavaScript statements.

To see an example of a JavaScript error message, modify the statement you added in the previous section. We'll use a common error: omitting one of the parentheses. Change the last `document.write` statement in Listing 2.3 to read

```
document.write("</h1>");
```

Save your HTML document again and load the document into the browser. Depending on the browser version you're using, one of two things will happen: Either an error message will be displayed, or the script will simply fail to execute.

If an error message is displayed, you're halfway to fixing the problem by adding the missing parenthesis. If no error was displayed, you should configure your browser to display error messages so that you can diagnose future problems:

- In Netscape or Firefox, type **javascript:** into the browser's Location field to display the JavaScript Console. In Firefox, you can also select Tools, JavaScript Console from the menu. The console is shown in Figure 2.4, displaying the error message you created in this example.

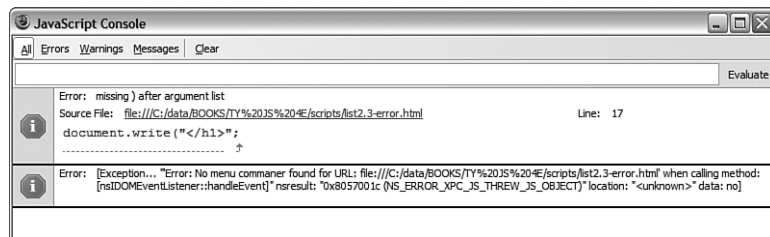
- In Internet Explorer, select Tools, Internet Options. On the Advanced page, uncheck the Disable Script Debugging box and check the Display a Notification About Every Script Error box. (If this is disabled, a yellow icon in the status bar will still notify you of errors.)

By the Way

Notice the field at the top of the JavaScript Console. This enables you to type a JavaScript statement, which will be executed immediately. This is a handy way to test JavaScript's features.

FIGURE 2.4

Firefox's JavaScript Console displays an error message.



The error we get in this case is missing `)` after argument list (Firefox) or Expected `') '` (Internet Explorer), which turns out to be exactly the problem. Be warned, however, that error messages aren't always this enlightening.

While Internet Explorer displays error dialog boxes for each error, Firefox's JavaScript Console displays a single list of errors and allows you to test commands. For this reason, you might find it useful to install Firefox for debugging and testing JavaScript, even if Internet Explorer is your primary browser.

Did you Know?

As you develop larger JavaScript applications, finding and fixing errors becomes more important. You'll learn more about dealing with JavaScript errors in Hour 16, "Debugging JavaScript Applications."



Try It Yourself

Using a Separate JavaScript File

Although simple scripts like this one can be embedded in an HTML file, as in the previous example, it's good practice to separate the HTML and JavaScript by using a separate JavaScript file. This has a few advantages:

- Browsers with JavaScript disabled, or older browsers that don't support it, will ignore the script.

- ▶ When multiple pages on your site use the same script, the browser only has to load the JavaScript file once, and use a cached copy on other pages.
- ▶ It's easier to maintain the HTML and JavaScript code when they're separated, especially if different people are working on the design and the scripting.

We'll also be using separate JavaScript files for most of the examples in this book, so you should be familiar with this technique.

To use a separate JavaScript file with the date and time example, you will need two files. A quick way to create them is to save the combined HTML/JavaScript file in Listing 2.3 to two files, and then edit them.

The first file, `datetime.html`, will be the HTML file. Remove everything between the `<script>` tags, and add the `src="datetime.js"` attribute to the opening `<script>` tag. The resulting file is shown in Listing 2.4.

LISTING 2.4 HTML File for the Date and Time Script (`datetime.html`)

```
<html>
<head><title>Displaying Times and Dates</title></head>
<body>
<h1>Current Date and Time</h1>
<p>
<script language="JavaScript" type="text/javascript"
  src = "datetime.js">
</script>
</p>
</body>
</html>
```

The second file, `datetime.js`, will contain only JavaScript commands—the same ones you removed from the HTML file. This file should *not* include `<script>` tags, or any HTML tags. The JavaScript file is shown in Listing 2.5.

LISTING 2.5 The Date and Time Script (`datetime.js`)

```
now = new Date();
localtime = now.toString();
utctime = now.toGMTString();
document.write("<b>Local time:</b> " + localtime + "<BR>");
document.write("<b>UTC time:</b> " + utctime);
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
document.write("<h1>");
document.write(hours + ":" + mins + ":" + secs);
document.write("</h1>");
```

**By the
Way**

If Internet Explorer displays a warning message in a yellow bar at the top of the browser window instead of executing your script, simply click the bar and select Allow Blocked Content.

As you create larger scripts, you'll find it far less confusing to keep the HTML and JavaScript in separate files. The next hour discusses this and other best practices for JavaScript.

Summary

During this hour, you wrote a simple JavaScript program and tested it using a browser. You learned about the tools you need to work with JavaScript—basically, an editor and a browser. You also learned how to modify and test scripts, and what happens when a JavaScript program runs into an error. Finally, you learned how to use scripts in separate JavaScript files.

In the process of writing this script, you have used some of JavaScript's basic features: variables, the `document.write` statement, and functions for working with dates and times.

Now that you've learned a bit of JavaScript syntax, you're ready to learn more of the details. You'll do that in Hour 3, "Getting Started with JavaScript Programming."

Q&A

- Q.** *Why do I need more than one browser to test scripts? Won't JavaScript behave the same way on both browsers?*
- A.** Although JavaScript is standardized, the browsers don't interpret it in exactly the same way. Your script might have minor flaws that have no effect in one browser but cause an error in another. Also, as you move on to more advanced features of JavaScript, you'll need to deal with browsers in different ways, as described in Hour 15, "Unobtrusive Scripting," and you'll need to test each one.
- Q.** *When I try to run my script, the browser displays the actual script in the browser window instead of executing it. What did I do wrong?*

- A.** This is most likely caused by one of three errors. First, you might be missing the beginning or ending `<script>` tags. Check them, and verify that the first reads `<script LANGUAGE="JavaScript" type="text/javascript">`. Second, your file might have been saved with a `.txt` extension, causing the browser to treat it as a text file. Rename it to `.htm` or `.html` to fix the problem. Third, make sure your browser supports JavaScript, and that it is not disabled in the Preferences dialog.
- Q.** *Why are the `` and `
` tags allowed in the statements to print the time? I thought HTML tags weren't allowed within the `<script>` tags.*
- A.** Because this particular tag is inside quotation marks, it's considered a valid part of the script. The script's output, including any HTML tags, is interpreted and displayed by the browser. You can use other HTML tags within quotation marks to add formatting, such as the `<h1>` tags we added for the large clock display.

Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

- 1.** What software do you use to create and edit JavaScript programs?
 - a.** A browser
 - b.** A text editor
 - c.** A pencil and a piece of paper
- 2.** What are variables used for in JavaScript programs?
 - a.** Storing numbers, dates, or other values
 - b.** Varying randomly
 - c.** Causing high school algebra flashbacks
- 3.** What should appear at the very end of a JavaScript script embedded in an HTML file?
 - a.** The `<script LANGUAGE="JavaScript">` tag
 - b.** The `</script>` tag
 - c.** The END statement

Quiz Answers

1. b. Any text editor can be used to create scripts. You can also use a word processor if you're careful to save the document as a text file with the .html or .htm extension.
2. a. Variables are used to store numbers, dates, or other values.
3. b. Your script should end with the `</script>` tag.

Exercises

To further your knowledge of JavaScript, perform the following exercises:

- ▶ Add a millisecond field to the large clock. You can use the `getMilliseconds` function, which works just like `getSeconds` but returns milliseconds.
- ▶ Modify the script to display the time, including milliseconds, twice. Notice whether any time passes between the two time displays when you load the page.

Hour 3

Getting Started with JavaScript Programming

What You'll Learn in This Hour:

- ▶ Organizing scripts using functions
- ▶ What objects are and how JavaScript uses them
- ▶ How JavaScript can respond to events
- ▶ An introduction to conditional statements and loops
- ▶ How browsers execute scripts in the proper order
- ▶ Syntax rules for avoiding JavaScript errors
- ▶ Adding comments to document your JavaScript code

You've reached the halfway point of Part I of this book. In the first couple of hours, you've learned what JavaScript is, learned the variety of things JavaScript can do, and created a simple script.

In this hour, you'll learn a few basic concepts and script components that you'll use in just about every script you write. This will prepare you for the remaining hours of this book, in which you'll explore specific JavaScript functions and features.

Basic Concepts

There are a few basic concepts and terms you'll run into throughout this book. In the following sections, you'll learn about the basic building blocks of JavaScript.

Statements

Statements are the basic units of a JavaScript program. A statement is a section of code that performs a single action. For example, the following three statements are from the date and time example in Hour 2, “Creating Simple Scripts”:

```
hours = now.getHours();  
mins = now.getMinutes();  
secs = now.getSeconds();
```

Although a statement is typically a single line of JavaScript, this is not a rule—it’s possible to break a statement across multiple lines, or to include more than one statement in a single line.

A semicolon marks the end of a statement. You can also omit the semicolon if you start a new line after the statement. If you combine statements into a single line, you must use semicolons to separate them.

Combining Tasks with Functions

In the basic scripts you’ve examined so far, you’ve seen some JavaScript statements that have a section in parentheses, like this:

```
document.write("Testing.");
```

This is an example of a *function*. Functions provide a simple way to handle a task, such as adding output to a web page. JavaScript includes a wide variety of built-in functions, which you will learn about throughout this book. A statement that uses a function, as in the preceding example, is referred to as a *function call*.

Functions take parameters (the expression inside the parentheses) to tell them what to do. Additionally, a function can return a value to a waiting variable. For example, the following function call prompts the user for a response and stores it in the text variable:

```
text = prompt("Enter some text.")
```

You can also create your own functions. This is useful for two main reasons: First, you can separate logical portions of your script to make it easier to understand. Second, and more importantly, you can use the function several times or with different data to avoid repeating script statements.

Variables

In Hour 2, you learned that variables are containers that can store a number, a string of text, or another value. For example, the following statement creates a variable called `fred` and assigns it the value 27:

```
var fred = 27;
```

JavaScript variables can contain numbers, text strings, and other values. You'll learn more about them in Hour 5, "Using Variables, Strings, and Arrays."

Understanding Objects

JavaScript also supports *objects*. Like variables, objects can store data—but they can store two or more pieces of data at once.

The items of data stored in an object are called the *properties* of the object. For example, you could use objects to store information about people such as in an address book. The properties of each person object might include a name, an address, and a telephone number.

JavaScript uses periods to separate object names and property names. For example, for a person object called `Bob`, the properties might include `Bob.address` and `Bob.phone`.

Objects can also include *methods*. These are functions that work with the object's data. For example, our person object for the address book might include a `display()` method to display the person's information. In JavaScript terminology, the statement `Bob.display()` would display Bob's details.

The `document.write` function we discussed earlier this hour is actually the `write` method of the `document` object. You will learn more about this object in Hour 4, "Working with the Document Object Model (DOM)."

**By the
Way**

Don't worry if this sounds confusing—you'll be exploring objects in much more detail later in this book. For now, you just need to know the basics. JavaScript supports three kinds of objects:

- ▶ *Built-in objects* are built in to the JavaScript language. You've already encountered one of these, `Date`, in Hour 2. Other built-in objects include `Array` and `String`, which you'll explore in Hour 5, and `Math`, which is explained in Hour 8, "Using Built-in Functions and Libraries."
- ▶ *DOM (Document Object Model) objects* represent various components of the browser and the current HTML document. For example, the `alert()` function

you used earlier in this hour is actually a method of the window object. You'll explore these in more detail in Hour 4.

- *Custom objects* are objects you create yourself. For example, you could create a person object, as in the examples in this section. You'll learn to use custom objects in Hour 6.

Conditionals

Although event handlers notify your script when something happens, you might want to check certain conditions yourself. For example, did the user enter a valid email address?

JavaScript supports *conditional statements*, which enable you to answer questions like this. A typical conditional uses the `if` statement, as in this example:

```
if (count==1) alert("The countdown has reached 1.");
```

This compares the variable `count` with the constant `1`, and displays an alert message to the user if they are the same. You will use conditional statements like this in most of your scripts.

By the Way

You'll learn more about conditionals in Hour 7, "Controlling Flow with Conditions and Loops."

Loops

Another useful feature of JavaScript—and most other programming languages—is the capability to create *loops*, or groups of statements that repeat a certain number of times. For example, these statements display the same alert 10 times, greatly annoying the user:

```
for (i=1; i<=10; i++) {  
    Alert("Yes, it's yet another alert!");  
}
```

The `for` statement is one of several statements JavaScript uses for loops. This is the sort of thing computers are supposed to be good at: performing repetitive tasks. You will use loops in many of your scripts, in much more useful ways than this example.

By the Way

Loops are covered in detail in Hour 7.

Event Handlers

As mentioned in Hour 1, “Understanding JavaScript,” not all scripts are located within `<script>` tags. You can also use scripts as *event handlers*. Although this might sound like a complex programming term, it actually means exactly what it says: Event handlers are scripts that handle events.

In real life, an event is something that happens to you. For example, the things you write on your calendar are events: “Dentist appointment” or “Fred’s birthday.” You also encounter unscheduled events in your life: for example, a traffic ticket, an IRS audit, or an unexpected visit from relatives.

Whether events are scheduled or unscheduled, you probably have normal ways of handling them. Your event handlers might include things such as *When Fred’s birthday arrives, send him a present* or *When relatives visit unexpectedly, turn out the lights and pretend nobody is home*.

Event handlers in JavaScript are similar: They tell the browser what to do when a certain event occurs. The events JavaScript deals with aren’t as exciting as the ones you deal with—they include such events as *When the mouse button clicks* and *When this page is finished loading*. Nevertheless, they’re a very useful part of JavaScript.

Many JavaScript events (such as mouse clicks) are caused by the user. Rather than doing things in a set order, your script can respond to the user’s actions. Other events don’t involve the user directly—for example, an event is triggered when an HTML document finishes loading.

Each event handler is associated with a particular browser object, and you can specify the event handler in the tag that defines the object. For example, images and text links have an event, `onMouseOver`, that happens when the mouse pointer moves over the object. Here is a typical HTML image tag with an event handler:

```

```

You specify the event handler as an attribute to the HTML tag and include the JavaScript statement to handle the event within the quotation marks. This is an ideal use for functions because function names are short and to the point and can refer to a whole series of statements.

See the Try It Yourself section at the end of this hour for a complete example of an event handler within an HTML document.

You can also define event handlers within JavaScript without using HTML attributes. You’ll learn this technique, and more about event handlers, in Hour 9, “Responding to Events.”

**By the
Way**

Which Script Runs First?

You can actually have several scripts within a web document: one or more sets of `<script>` tags, external JavaScript files, and any number of event handlers. With all of these scripts, you might wonder how the browser knows which to execute first. Fortunately, this is done in a logical fashion:

- ▶ Sets of `<script>` tags within the `<head>` section of an HTML document are handled first, whether they include embedded code or refer to a JavaScript file. Because these scripts cannot create output in the web page, it's a good place to define functions for use later.
- ▶ Sets of `<script>` tags within the `<body>` section of the HTML document are executed after those in the `<head>` section, while the web page loads and displays. If there is more than one script in the body, they are executed in order.
- ▶ Event handlers are executed when their events happen. For example, the `onLoad` event handler is executed when the body of a web page loads. Because the `<head>` section is loaded before any events, you can define functions there and use them in event handlers.

JavaScript Syntax Rules

JavaScript is a simple language, but you do need to be careful to use its *syntax*—the rules that define how you use the language—correctly. The rest of this book covers many aspects of JavaScript syntax, but there are a few basic rules you should understand to avoid errors.

Case Sensitivity

Almost everything in JavaScript is *case sensitive*: you cannot use lowercase and capital letters interchangeably. Here are a few general rules:

- ▶ JavaScript keywords, such as `for` and `if`, are always lowercase.
- ▶ Built-in objects such as `Math` and `Date` are capitalized.
- ▶ DOM object names are usually lowercase, but their methods are often a combination of capitals and lowercase. Usually capitals are used for all but the first word, as in `toLowerCase` and `getElementById`.

When in doubt, follow the exact case used in this book or another JavaScript reference. If you use the wrong case, the browser will usually display an error message.

Variable, Object, and Function Names

When you define your own variables, objects, or functions, you can choose their names. Names can include uppercase letters, lowercase letters, numbers, and the underscore (`_`) character. Names must begin with a letter or underscore.

You can choose whether to use capitals or lowercase in your variable names, but remember that JavaScript is case sensitive: `score`, `Score`, and `SCORE` would be considered three different variables. Be sure to use the same name each time you refer to a variable.

Reserved Words

One more rule for variable names—they must not be *reserved words*. These include the words that make up the JavaScript language, such as `if` and `for`, DOM object names such as `window` and `document`, and built-in object names such as `Math` and `Date`. A complete list of reserved words is included in Appendix D, “JavaScript Quick Reference.”

Spacing

Blank space (known as *whitespace* by programmers) is ignored by JavaScript. You can include spaces and tabs within a line, or blank lines, without causing an error. Blank space often makes the script more readable.

Using Comments

JavaScript *comments* enable you to include documentation within your script. This will be useful if someone else tries to understand the script, or even if you try to understand it after a long break. To include comments in a JavaScript program, begin a line with two slashes, as in this example:

```
//this is a comment.
```

You can also begin a comment with two slashes in the middle of a line, which is useful for documenting a script. In this case, everything on the line after the slashes is treated as a comment and ignored by the browser. For example,

```
a = a + 1; // add one to the value of a
```

JavaScript also supports C-style comments, which begin with `/*` and end with `*/`. These comments can extend across more than one line, as the following example demonstrates:

```
/*This script includes a variety  
of features, including this comment.*/
```

Because JavaScript statements within a comment are ignored, C-style comments are often used for *commenting out* sections of code. If you have some lines of JavaScript that you want to temporarily take out of the picture while you debug a script, you can add `/*` at the beginning of the section and `*/` at the end.

By the Way

Because these comments are part of JavaScript syntax, they are only valid inside `<script>` tags or within an external JavaScript file.

Best Practices for JavaScript

You should now be familiar with the basic rules for writing valid JavaScript. Along with following the rules, it's also a good idea to follow *best practices*. The following practices may not be required, but you'll save yourself and others some headaches if you follow them.

- ▶ **Use comments liberally**—These make your code easier for others to understand, and also easier for you to understand when you edit them later. They are also useful for marking the major divisions of a script.
- ▶ **Use a semicolon at the end of each statement, and only use one statement per line**—This will make your scripts easier to debug.
- ▶ **Use separate JavaScript files whenever possible**—This separates JavaScript from HTML and makes debugging easier, and also encourages you to write modular scripts that can be reused.
- ▶ **Avoid being browser-specific**—As you learn more about JavaScript, you'll learn some features that only work in one browser. Avoid them unless absolutely necessary, and always test your code in more than one browser.
- ▶ **Keep JavaScript optional**—Don't use JavaScript to perform an essential function on your site—for example, the primary navigation links. Whenever possible, users without JavaScript should be able to use your site, although it may not be quite as attractive or convenient. This strategy is known as *progressive enhancement*.

There are many more best practices involving more advanced aspects of JavaScript. These are covered in detail in Hour 15, "Unobtrusive Scripting."

Try It Yourself



Using an Event Handler

To conclude this hour, here's a simple example of an event handler. This will demonstrate how you set up an event, which you'll use throughout this book, and how JavaScript works without `<script>` tags. Listing 3.1 shows an HTML document that includes a simple event handler.

LISTING 3.1 An HTML Document with a Simple Event Handler

```
<html>
<head>
<title>Event Handler Example</title>
</head>
<body>
<h1>Event Handler Example</h1>
<p>
<a href="http://www.jsworkshop.com/"
onClick="alert('Aha! An Event!');">Click this link</a>
to test an event handler.
</p>
</body>
</html>
```

The event handler is defined with the following `onClick` attribute within the `<a>` tag that defines a link:

```
onClick="alert('Aha! An Event!');"
```

This event handler uses the built-in `alert()` function to display a message when you click on the link. In more complex scripts, you will usually define your own function to act as an event handler. Figure 3.1 shows this example in action.

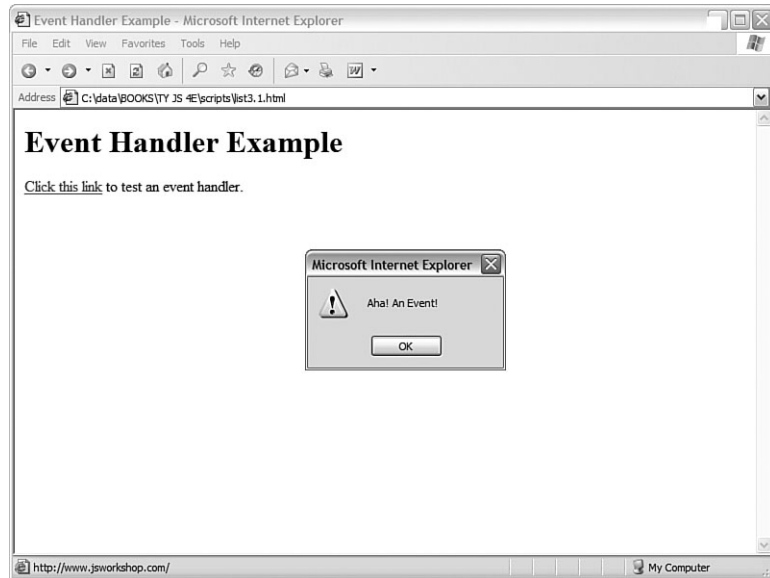
You'll use other event handlers similar to this in the next hour, and events will be covered in more detail in Hour 9.

Notice that after you click the OK button on the alert, the browser follows the link defined in the `<a>` tag. Your event handler could also stop the browser from following the link, as described in Hour 9.

***Did you
Know?***

FIGURE 3.1

The browser displays an alert when you click the link.



Summary

During this hour, you've been introduced to several components of JavaScript programming and syntax: functions, objects, event handlers, conditions, and loops. You also learned how to use JavaScript comments to make your script easier to read, and looked at a simple example of an event handler.

In the next hour, you'll look at the Document Object Model (DOM) and learn how you can use the objects within the DOM to work with web pages and interact with users.

Q&A

- Q.** *I've heard the term object-oriented applied to languages such as C++ and Java. If JavaScript supports objects, is it an object-oriented language?*
- A.** Yes, although it might not fit some people's strict definitions. JavaScript objects do not support all of the features that languages such as C++ and Java support, although the latest versions of JavaScript have added more object-oriented features.
- Q.** *Having several scripts that execute at different times seems confusing. Why would I want to use event handlers?*

- A.** Event handlers are the ideal way (and in JavaScript, the only way) to handle gadgets within the web page, such as buttons, check boxes, and text fields. It's actually more convenient to handle them this way. Rather than writing a script that sits and waits for a button to be pushed, you can simply create an event handler and let the browser do the waiting for you.
- Q.** *Some examples in other books suggest enclosing scripts in HTML comments (<!-- and -->) to hide the script from older browsers. Is this necessary?*
- A.** This technique was only necessary for supporting very old browsers, such as Netscape 2.0. I no longer recommend this because all modern browsers handle JavaScript correctly. If you are still concerned about non-JavaScript browsers, the best way to hide your script is to use an external JavaScript file, as described in Hour 2.

Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

- 1.** A script that executes when the user clicks the mouse button is an example of what?
 - a.** An object
 - b.** An event handler
 - c.** An impossibility
- 2.** Which of the following are capabilities of functions in JavaScript?
 - a.** Accept parameters
 - b.** Return a value
 - c.** Both of the above
- 3.** Which of the following is executed first by a browser?
 - a.** A script in the <head> section
 - b.** A script in the <body> section
 - c.** An event handler for a button

Quiz Answers

1. b. A script that executes when the user clicks the mouse button is an event handler.
2. c. Functions can accept both parameters and return values.
3. a. Scripts defined in the <head> section of an HTML document are executed first by the browser.

Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Examine the Date and Time script you created in Hour 2 and find any examples of functions and objects being used.
- ▶ Add JavaScript comments to the Date and Time script to make it more clear what each line does. Verify that the script still runs properly.

HOURL 4

Working with the Document Object Model (DOM)

What You'll Learn in This Hour:

- ▶ How to access the various objects in the DOM
- ▶ Working with windows using the `window` object
- ▶ Working with web documents using the `document` object
- ▶ Using objects for links and anchors
- ▶ Using the `location` object to work with URLs
- ▶ Creating JavaScript-based Back and Forward buttons

You've reached the end of Part I. In this hour, you'll be introduced to one of the most important tools you'll use with JavaScript: the Document Object Model (DOM), which lets your scripts manipulate web pages, windows, and documents.

Without the DOM, JavaScript would be just another scripting language—with the DOM, it becomes a powerful tool for making pages dynamic. This hour will introduce the idea of the DOM and some of the objects you'll use most often.

Understanding the Document Object Model (DOM)

One advantage that JavaScript has over basic HTML is that scripts can manipulate the web document and its contents. Your script can load a new page into the browser, work with parts of the browser window and document, open new windows, and even modify text within the page dynamically.

To work with the browser and documents, JavaScript uses a hierarchy of parent and child objects called the Document Object Model (DOM). These objects are organized into a tree-like structure, and represent all of the content and components of a web document.

By the Way

The DOM is not part of the JavaScript language—rather, it’s an API (application programming interface) built in to the browser. While the DOM is most often used with JavaScript, it can also be used by other languages, such as VBScript and Java.

The objects in the DOM have *properties*—variables that describe the web page or document, and *methods*—functions that enable you to work with parts of the web page.

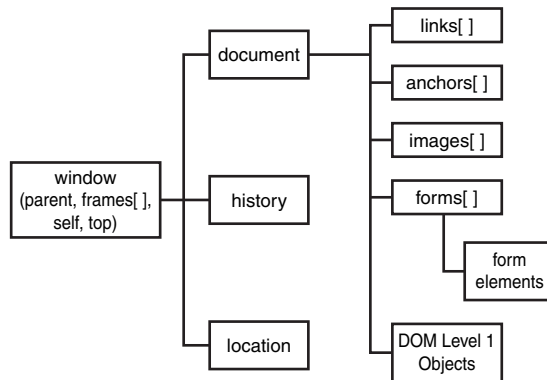
When you refer to an object, you use the parent object name followed by the child object name or names, separated by periods. For example, JavaScript stores objects to represent images in a document as children of the document object. The following refers to the `image9` object, a child of the document object, which is a child of the window object:

```
window.document.image9
```

The window object is the parent object for all of the objects we will be looking at in this hour. Figure 4.1 shows this section of the DOM object hierarchy and a variety of its objects.

FIGURE 4.1

The DOM object hierarchy.

**By the Way**

This diagram only includes the basic browser objects that will be covered in this hour. These are actually a small part of the DOM, which you’ll learn more about in Part III, “Learning More About the DOM.”

History of the DOM

Starting with the introduction of JavaScript 1.0 in Netscape 2.0, browsers have included objects that represent parts of a web document and other browser features. However, there was never a true standard. While both Netscape and Microsoft Internet

Explorer included many of the same objects, there was no guarantee that the same objects would work the same way in both browsers, let alone in less common browsers.

The bad news is that there are still differences between the browsers—but here’s the good news. Since the release of Netscape 3.0 and Internet Explorer 4.0, all of the basic objects (those covered in this hour) are supported in much the same way in both browsers. With more recent browser releases, a much more advanced DOM is supported.

DOM Levels

The W3C (World Wide Web Consortium) developed the DOM level 1 recommendation. This is a standard that defines not only basic objects, but an entire set of objects that encompass all parts of an HTML document. A level 2 DOM standard has also been released, and level 3 is under development.

Netscape 4 and Internet Explorer 4 supported their own DOMs that allowed more control over documents, but weren’t standardized. Fortunately, starting with Internet Explorer 5 and Netscape 6, both support the W3C DOM, so you can support both browsers with simple, standards-compliant code. All of today’s current browsers support the W3C DOM.

The basic object hierarchy described in this hour is informally referred to as DOM level 0, and the objects are included in the DOM level 1 standard. You’ll learn how to use the W3C DOM to work with any part of a web document later in this book.

The W3C DOM allows you to modify a web page in real time after it has loaded. You’ll learn how to do this in Part III.

***Did you
Know?***

Using window Objects

At the top of the browser object hierarchy is the window object, which represents a browser window. You’ve already used at least one method of the window object: the `window.alert()` method, or simply `alert()`, displays a message in an alert box.

There can be several window objects at a time, each representing an open browser window. Frames are also represented by window objects. You’ll learn more about windows and frames in Hour 10, “Using Windows and Frames.”

Layers, which enable you to include, modify, and position dynamic content within a web document, are also similar to window objects. These are explained in Hour 13, “Using the W3C DOM.”

***By the
Way***

Working with Web Documents

The document object represents a web document, or page. Web documents are displayed within browser windows, so it shouldn't surprise you to learn that the document object is a child of the window object.

Because the window object always represents the current window (the one containing the script), you can use `window.document` to refer to the current document. You can also simply refer to `document`, which automatically refers to the current window.

By the Way

You've already used the `document.write` method to display text within a web document. The examples in earlier hours only used a single window and document, so it was unnecessary to use `window.document.write`—but this longer syntax would have worked equally well.

If multiple windows or frames are in use, there might be several window objects, each with its own document object. To use one of these document objects, you use the name of the window and the name of the document.

In the following sections, you will look at some of the properties and methods of the document object that will be useful in your scripting.

Getting Information About the Document

Several properties of the document object include information about the current document in general:

- ▶ `document.URL` specifies the document's URL. This is a simple text field. You can't change this property. If you need to send the user to a different location, use the `window.location` object, described later in this hour.
- ▶ `document.title` lists the title of the current page, defined by the HTML `<title>` tag.
- ▶ `document.referrer` is the URL of the page the user was viewing prior to the current page—usually, the page with a link to the current page.
- ▶ `document.lastModified` is the date the document was last modified. This date is sent from the server along with the page.
- ▶ `document.bgColor` and `document.fgColor` are the background and foreground (text) colors for the document, corresponding to the `BGCOLOR` and `TEXT` attributes of the `<body>` tag.

- ▶ `document.linkColor`, `document.alinkColor`, and `document.vlinkColor` are the colors for links within the document. These correspond to the LINK, ALINK, and VLINK attributes of the `<body>` tag.
- ▶ `document.cookie` enables you to read or set a cookie for the document. See <http://www.jsworkshop.com/cookies.html> for information about cookies.

As an example of a document property, Listing 4.1 shows a short HTML document that displays its last modified date using JavaScript.

LISTING 4.1 Displaying the Last Modified Date

```
<html><head><title>Test Document</title></head>
<body>
<p>This page was last modified on:
<script language="JavaScript" type="text/javascript">
document.write(document.lastModified);
</script>
</p>
</body>
</html>
```

This can tell the user when the page was last changed. If you use JavaScript, you don't have to remember to update the date each time you modify the page. (You could also use the script to always print the current date instead of the last modified date, but that would be cheating.)

You might find that the `document.lastModified` property doesn't work on your web pages, or returns the wrong value. The date is received from the web server, and some servers do not maintain modification dates correctly.

**By the
Way**

Writing Text in a Document

The simplest document object methods are also the ones you will use most often. In fact, you've used one of them already. The `document.write` method prints text as part of the HTML page in a document window. This statement is used whenever you need to include output in a web page.

An alternative statement, `document.writeln`, also prints text, but it also includes a newline (`\n`) character at the end. This is handy when you want your text to be the last thing on the line.

Bear in mind that the newline character is displayed as a space by the browser, except inside a `<pre>` container. You will need to use the `
` tag if you want an actual line break.

**Watch
Out!**

You can use these methods only within the body of the web page, so they will be executed when the page loads. You can't use these methods to add to a page that has already loaded without reloading it.

By the Way

You can also directly modify the text of a web page on newer browsers using the features of the new DOM. You'll learn these techniques in Hour 14.

The `document.write` method can be used within a `<script>` tag in the body of an HTML document. You can also use it in a function, provided you include a call to the function within the body of the document.

Using Links and Anchors

Another child of the `document` object is the `link` object. Actually, there can be multiple `link` objects in a document. Each one includes information about a link to another location or an anchor.

Did you Know?

Anchors are named places in an HTML document that can be jumped to directly. You define them with a tag like this: ``. You can then link to them: ``.

You can access `link` objects with the `links` array. Each member of the array is one of the `link` objects in the current page. A property of the array, `document.links.length`, indicates the number of links in the page.

Each `link` object (or member of the `links` array) has a list of properties defining the URL. The `href` property contains the entire URL, and other properties define portions of it. These are the same properties as the `location` object, defined later in this hour.

You can refer to a property by indicating the link number and property name. For example, the following statement assigns the entire URL of the first link to the variable `link1`:

```
link1 = links[0].href;
```

The anchor objects are also children of the `document` object. Each anchor object represents an anchor in the current document—a particular location that can be jumped to directly.

Like links, you can access anchors with an array: `anchors`. Each element of this array is an anchor object. The `document.anchors.length` property gives you the number of elements in the `anchors` array.

Accessing Browser History

The history object is another child (property) of the window object. This object holds information about the URLs that have been visited before and after the current one, and it includes methods to go to previous or next locations.

The history object has one property you can access:

- ▶ `history.length` keeps track of the length of the history list—in other words, the number of different locations that the user has visited.

The history object has `current`, `previous`, and `next` properties that store URLs of documents in the history list. However, for security and privacy reasons, these objects are not normally accessible in today's browsers.

**By the
Way**

The history object has three methods you can use to move through the history list:

- ▶ `history.go()` opens a URL from the history list. To use this method, specify a positive or negative number in parentheses. For example, `history.go(-2)` is equivalent to pressing the Back button twice.
- ▶ `history.back()` loads the previous URL in the history list—equivalent to pressing the Back button.
- ▶ `history.forward()` loads the next URL in the history list, if available. This is equivalent to pressing the Forward button.

You'll use these methods in the Try It Yourself section at the end of this hour.

Working with the location Object

A third child of the window object is the location object. This object stores information about the current URL stored in the window. For example, the following statement loads a URL into the current window:

```
window.location.href="http://www.starlingtech.com";
```

The `href` property used in this statement contains the entire URL of the window's current location. You can also access portions of the URL with various properties of the location object. To explain these properties, consider the following URL:

```
http://www.jsworkshop.com:80/test.cgi?lines=1#anchor
```

The following properties represent parts of the URL:

- ▶ `location.protocol` is the protocol part of the URL (`http:` in the example).
- ▶ `location.hostname` is the host name of the URL (`www.jsworkshop.com` in the example).
- ▶ `location.port` is the port number of the URL (`80` in the example).
- ▶ `location.pathname` is the filename part of the URL (`test.cgi` in the example).
- ▶ `location.search` is the query portion of the URL, if any (`lines=1` in the example). Queries are used mostly by CGI scripts.
- ▶ `location.hash` is the anchor name used in the URL, if any (`#anchor` in the example).

The `link` object, introduced earlier this hour, also includes this list of properties for accessing portions of the URL.

By the Way

Although the `location.href` property usually contains the same URL as the `document.URL` property described earlier in this hour, you can't change the `document.URL` property. Always use `location.href` to load a new page.

The `location` object has two methods:

- ▶ `location.reload()` reloads the current document. This is the same as the Reload button on the browser's toolbar. If you optionally include the `true` parameter, it will ignore the browser's cache and force a reload whether the document has changed or not.
- ▶ `location.replace()` replaces the current location with a new one. This is similar to setting the `location` object's properties yourself. The difference is that the `replace` method does not affect the browser's history. In other words, the Back button can't be used to go to the previous location. This is useful for splash screens or temporary pages that it would be useless to return to.



Try It Yourself

Creating Back and Forward Buttons

You can use the `back` and `forward` methods of the `history` object to add your own Back and Forward buttons to a web document. The browser already has Back and Forward buttons, of course, but it's occasionally useful to include your own links that serve the same purpose.

You will now create a script that displays Back and Forward buttons and use these methods to navigate the browser. Here's the code that will create the Back button:

```
<input type="button"
  onClick="history.back();" value="<- Back">
```

The `<input>` tag defines a button labeled Back. The `onClick` event handler uses the `history.back()` method to go to the previous page in history. The code for the Forward button is similar:

```
<input type="button"
  onClick="history.forward();" value="Forward -->">
```

With these out of the way, you just need to build the rest of the HTML document. Listing 4.2 shows the complete HTML document, and Figure 4.2 shows a browser's display of the document. After you load this document into a browser, visit other URLs and make sure the Back and Forward buttons work.

LISTING 4.2 A Web Page That Uses JavaScript to Include Back and Forward Buttons

```
<html>
<head><title>Back and Forward Buttons</title>
</head>
<body>
<h1>Back and Forward Buttons</h1>
<p>This page allows you to go back or forward to pages in the history list.
These should be equivalent to the back and forward arrow buttons in the
browser's toolbar.</p>
<p>
<input type="button"
  onClick="history.back();" value="<- Back">
<input type="button"
  onClick="history.forward();" value="Forward -->">
</p>
</body>
</html>
```

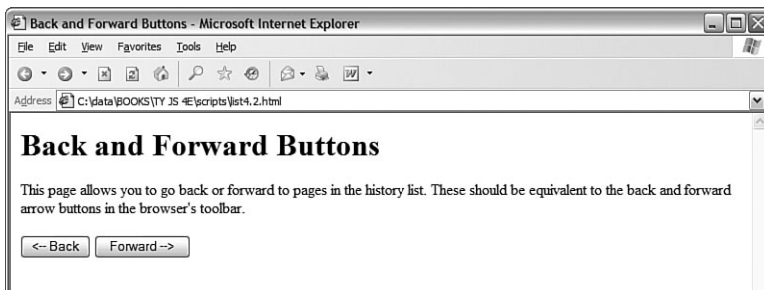


FIGURE 4.2
The Back and Forward buttons in Internet Explorer.

Summary

In this hour, you've learned about the Document Object Model (DOM), JavaScript's hierarchy of web page objects. You've learned how you can use the document object to work with documents, and used the history and location objects to control the current URL displayed in the browser.

You should now have a basic understanding of the DOM and some of its objects—you'll learn about more of the objects throughout this book.

Congratulations! You've reached the end of Part I of this book. In Part II, you'll get back to learning the JavaScript language, starting with Hour 5, "Using Variables, Strings, and Arrays."

Q&A

- Q.** *I can use history and document instead of window.history and window.document. Can I leave out the window object in other cases?*
- A.** Yes. For example, you can use alert instead of window.alert to display a message. The window object contains the current script, so it's treated as a default object. However, be warned that you shouldn't omit the window object's name when you're using frames, layers, or multiple windows, or in an event handler.
- Q.** *I used the document.lastModified method to display a modification date for my page, but it displays a date in 1970, or a date that I know is incorrect. What's wrong?*
- A.** This function depends on the server sending the last modified date of the document to the browser. Some web servers don't do this properly, or require specific file attributes in order for this to work.
- Q.** *Can I change history entries, or prevent the user from using the Back and Forward buttons?*
- A.** You can't change the history entries. You can't prevent the use of the Back and Forward buttons, but you can use the location.replace() method to load a series of pages that don't appear in the history. There are a few tricks for preventing the Back button from working properly, but I don't recommend them—that's the sort of thing that gives JavaScript a bad name.

Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

1. Which of the following objects can be used to load a new URL into the browser window?
 - a. `document.url`
 - b. `window.location`
 - c. `window.url`
2. Which object contains the `alert()` method?
 - a. `window`
 - b. `document`
 - c. `location`
3. Which of the following DOM levels describes the objects described in this hour?
 - a. DOM level 0
 - b. DOM level 1
 - c. DOM level 2

Quiz Answers

1. b. The `window.location` object can be used to send the browser to a new URL.
2. a. The `window` object contains the `alert()` method.
3. a. The objects described in this hour fall under the informal DOM level 0 specification.

Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Modify the Back and Forward example in Listing 4.2 to include a Reload button along with the Back and Forward buttons. (This button would trigger the `location.reload()` method.)
- ▶ Modify the Back and Forward example to display the current number of history entries.

This page intentionally left blank

PART II:

Learning JavaScript Basics

HOURL 5	Using Variables, Strings, and Arrays	63
HOURL 6	Using Functions and Objects	85
HOURL 7	Controlling Flow with Conditions and Loops	101
HOURL 8	Using Built-in Functions and Libraries	121

This page intentionally left blank

HOURL 5

Using Variables, Strings, and Arrays

What You'll Learn in This Hour:

- ▶ Naming and declaring variables
- ▶ Choosing whether to use local or global variables
- ▶ Assigning values to variables
- ▶ How to convert between different data types
- ▶ Using variables and literals in expressions
- ▶ How strings are stored in String objects
- ▶ Creating and using String objects
- ▶ Creating and using arrays of numbers and strings

Welcome to the beginning of Part II of this book. Now that you have learned some of the fundamentals of JavaScript and the DOM, it's time to dig into more details of the JavaScript language.

In this hour, you'll learn three tools for storing data in JavaScript: variables, which store numbers or text; strings, which are special variables for working with text; and arrays, which are multiple variables you can refer to by number.

Using Variables

Unless you skipped the first few hours of this book, you've already used a few variables. You probably can also figure out how to use a few more without any help. Nevertheless, there are some aspects of variables you haven't learned yet. We will now look at some of the details.

Choosing Variable Names

Variables are named containers that can store data (for example, a number, a text string, or an object). As you learned earlier in this book, each variable has a name. There are specific rules you must follow when choosing a variable name:

- ▶ Variable names can include letters of the alphabet, both upper- and lowercase. They can also include the digits 0–9 and the underscore (_) character.
- ▶ Variable names cannot include spaces or any other punctuation characters.
- ▶ The first character of the variable name must be either a letter or an underscore.
- ▶ Variable names are case sensitive—`totalnum`, `Totalnum`, and `TotalNum` are separate variable names.
- ▶ There is no official limit on the length of variable names, but they must fit within one line.

Using these rules, the following are examples of valid variable names:

```
total_number_of_fish
LastInvoiceNumber
temp1
a
_var39
```

By the Way

You can choose to use either friendly, easy-to-read names or completely cryptic ones. Do yourself a favor: use longer, friendly names whenever possible. Although you might remember the difference between `a`, `b`, `x`, and `x1` right now, you might not after a good night's sleep.

Using Local and Global Variables

Some computer languages require you to declare a variable before you use it. JavaScript includes the `var` keyword, which can be used to declare a variable. You can omit `var` in many cases; the variable is still declared the first time you assign a value to it.

To understand where to declare a variable, you will need to understand the concept of *scope*. A variable's scope is the area of the script in which that variable can be used. There are two types of variables:

- ▶ *Global variables* have the entire script (and other scripts in the same HTML document) as their scope. They can be used anywhere, even within functions.
- ▶ *Local variables* have a single function as their scope. They can be used only within the function they are created in.

To create a global variable, you declare it in the main script, outside any functions. You can use the `var` keyword to declare the variable, as in this example:

```
var students = 25;
```

This statement declares a variable called `students` and assigns it a value of 25. If this statement is used outside functions, it creates a global variable. The `var` keyword is optional in this case, so this statement is equivalent to the previous one:

```
students = 25;
```

Before you get in the habit of omitting the `var` keyword, be sure you understand exactly when it's required. It's actually a good idea to always use the `var` keyword—you'll avoid errors and make your script easier to read, and it won't usually cause any trouble.

For the most part, the variables you've used in earlier hours of this book have been global.

**By the
Way**

A local variable belongs to a particular function. Any variable you declare with the `var` keyword in a function is a local variable. Additionally, the variables in the function's parameter list are always local variables.

To create a local variable within a function, you must use the `var` keyword. This forces JavaScript to create a local variable, even if there is a global variable with the same name.

You should now understand the difference between local and global variables. If you're still a bit confused, don't worry—if you use the `var` keyword every time, you'll usually end up with the right type of variable.

Assigning Values to Variables

As you learned in Hour 2, "Creating a Simple Script," you can use the equal sign to assign a value to a variable. For example, this statement assigns the value 40 to the variable `lines`:

```
lines = 40;
```

You can use any expression to the right of the equal sign, including other variables. You have used this syntax earlier to add one to a variable:

```
lines = lines + 1;
```

Because incrementing or decrementing variables is quite common, JavaScript includes two types of shorthand for this syntax. The first is the `+=` operator, which enables you to create the following shorter version of the preceding example:

```
lines += 1;
```

Similarly, you can subtract a number from a variable using the `-=` operator:

```
lines -= 1;
```

If you still think that's too much to type, JavaScript also includes the increment and decrement operators, `++` and `--`. This statement adds one to the value of `lines`:

```
lines++;
```

Similarly, this statement subtracts one from the value of `lines`:

```
lines--;
```

You can alternately use the `++` or `--` operators before a variable name, as in `++lines`. However, these are not identical. The difference is when the increment or decrement happens:

- ▶ If the operator is after the variable name, the increment or decrement happens *after* the current expression is evaluated.
- ▶ If the operator is before the variable name, the increment or decrement happens *before* the current expression is evaluated.

This difference is only an issue when you use the variable in an expression and increment or decrement it in the same statement. As an example, suppose you have assigned the `lines` variable the value 40. The following two statements have different effects:

```
alert(lines++);  
alert(++lines);
```

The first statement displays an alert with the value 40, and then increments `lines` to 41. The second statement first increments `lines` to 41, then displays an alert with the value 41.

Understanding Expressions and Operators

An *expression* is a combination of variables and values that the JavaScript interpreter can evaluate to a single value. The characters that are used to combine these values, such as + and /, are called *operators*.

Along with variables and constant values, you can also use calls to functions that return results within an expression.

***Did you
Know?***

Using JavaScript Operators

You've already used some operators, such as the + sign (addition) and the increment and decrement operators. Table 5.1 lists some of the most important operators you can use in JavaScript expressions.

TABLE 5.1 Common JavaScript Operators

Operator	Description	Example
+	Concatenate (combine) strings	message="this is" + " a test";
+	Add	result = 5 + 7;
-	Subtract	score = score - 1;
*	Multiply	total = quantity * price;
/	Divide	average = sum / 4;
%	Modulo (remainder)	remainder = sum % 4;
++	Increment	tries++;
--	Decrement	total--;

Along with these, there are also many other operators used in conditional statements—you'll learn about these in Hour 7, "Controlling Flow with Conditions and Loops."

Operator Precedence

When you use more than one operator in an expression, JavaScript uses rules of *operator precedence* to decide how to calculate the value. Table 5.1 lists the operators from lowest to highest precedence, and operators with highest precedence are evaluated first. For example, consider this statement:

```
result = 4 + 5 * 3;
```

If you try to calculate this result, there are two ways to do it. You could multiply $5 * 3$ first and then add 4 (result: 19) or add $4 + 5$ first and then multiply by 3 (result: 27). JavaScript solves this dilemma by following the precedence rules: Because multiplication has a higher precedence than addition, it first multiplies $5 * 3$ and then adds 4, producing a result of 19.

By the Way

If you're familiar with any other programming languages, you'll find that the operators and precedence in JavaScript work, for the most part, the same way as those in C, C++, and Java.

Sometimes operator precedence doesn't produce the result you want. For example, consider this statement:

```
result = a + b + c + d / 4;
```

This is an attempt to average four numbers by adding them all together and then dividing by four. However, because JavaScript gives division a higher precedence than addition, it will divide the *d* variable by 4 before adding the other numbers, producing an incorrect result.

You can control precedence by using parentheses. Here's the working statement to calculate an average:

```
result = (a + b + c + d) / 4;
```

The parentheses ensure that the four variables are added first, and then the sum is divided by four.

Did you Know?

If you're unsure about operator precedence, you can use parentheses to make sure things work the way you expect and to make your script more readable.

Data Types in JavaScript

In some computer languages, you have to specify the type of data a variable will store: for example, a number or a string. In JavaScript, you don't need to specify a data type in most cases. However, you should know the types of data JavaScript can deal with.

These are the basic JavaScript data types:

- *Numbers*, such as 3, 25, or 1.4142138. JavaScript supports both integers and floating-point numbers.

- ▶ *Boolean*, or logical values. These can have one of two values: true or false. These are useful for indicating whether a certain condition is true.

You'll learn more about Boolean values, and about using conditions in JavaScript, in Hour 7.

**By the
Way**

- ▶ *Strings*, such as "I am a jelly doughnut". These consist of one or more characters of text. (Strictly speaking, these are String objects, which you'll learn about later in this hour.)
- ▶ *The null value*, represented by the keyword null. This is the value of an undefined variable. For example, the statement `document.write(fig)` will result in this value (and an error message) if the variable `fig` has not been previously used or defined.

Although JavaScript keeps track of the data type currently stored in each variable, it doesn't restrict you from changing types midstream. For example, suppose you declared a variable by assigning it a value:

```
total = 31;
```

This statement declares a variable called `total` and assigns it the value of 31. This is a numeric variable. Now suppose you changed the value of `total`:

```
total = "albatross";
```

This assigns a string value to `total`, replacing the numeric value. JavaScript will not display an error when this statement executes; it's perfectly valid, although it's probably not a very useful `total`.

Although this feature of JavaScript is convenient and powerful, it can also make it easy to make a mistake. For example, if the `total` variable was later used in a mathematical calculation, the result would be invalid—but JavaScript does not warn you that you've made this mistake.

**By the
Way**

Converting Between Data Types

JavaScript handles conversions between data types for you whenever it can. For example, you've already used statements like this:

```
document.write("The total is " + total);
```

This statement prints out a message such as "The total is 40". Because the `document.write` function works with strings, the JavaScript interpreter automatically converts any nonstrings in the expression (in this case, the value of `total`) to strings before performing the function.

This works equally well with floating-point and Boolean values. However, there are some situations where it won't work. For example, the following statement will work fine if the value of `total` is 40:

```
average = total / 3;
```

However, the `total` variable could also contain a string; in this case, the preceding statement would result in an error.

In some situations, you might end up with a string containing a number, and need to convert it to a regular numeric variable. JavaScript includes two functions for this purpose:

- ▶ `parseInt()` — Converts a string to an integer number.
- ▶ `parseFloat()` — Converts a string to a floating-point number.

Both of these functions will read a number from the beginning of the string and return a numeric version. For example, these statements convert the string "30 angry polar bears" to a number:

```
stringvar = "30 angry polar bears";  
numvar = parseInt(stringvar);
```

After these statements execute, the `numvar` variable contains the number 30. The nonnumeric portion of the string is ignored.

By the Way

These functions look for a number of the appropriate type at the beginning of the string. If a valid number is not found, the function will return the special value NaN, meaning *not a number*.

Using String Objects

You've already used several strings during the first few hours of this book. Strings store a group of text characters, and are named similarly to other variables. As a simple example, this statement assigns the string `This is a test` to a string variable called `test`:

```
test = "This is a test";
```

Creating a String Object

JavaScript stores strings as String objects. You usually don't need to worry about this, but it will explain some of the techniques for working with strings, which use methods (built-in functions) of the String object.

There are two ways to create a new String object. The first is the one you've already used, whereas the second uses object-oriented syntax. The following two statements create the same string:

```
test = "This is a test";  
test = new String("This is a test");
```

The second statement uses the new keyword, which you use to create objects. This tells the browser to create a new String object containing the text `This is a test`, and assigns it to the variable `test`.

Although you can create a string using object-oriented syntax, the standard JavaScript syntax is simpler, and there is no difference in the strings created by these two methods.

**By the
Way**

Assigning a Value

You can assign a value to a string in the same way as any other variable. Both of the examples in the previous section assigned an initial value to the string. You can also assign a value after the string has already been created. For example, the following statement replaces the contents of the `test` variable with a new string:

```
test = "This is only a test.";
```

You can also use the concatenation operator (+) to combine the values of two strings. Listing 5.1 shows a simple example of assigning and combining the values of strings.

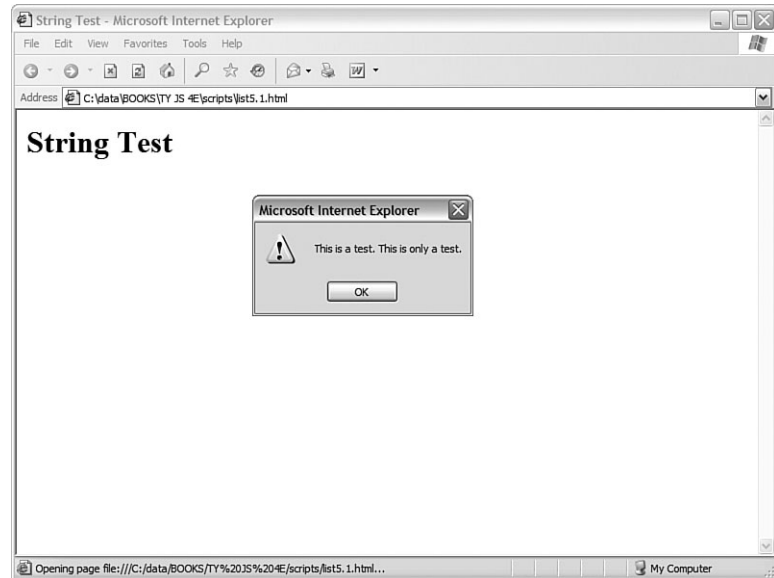
LISTING 5.1 Assigning Values to Strings and Combining Them

```
<html>  
<head>  
<title>String Test</title>  
</head>  
<body>  
<h1>String Test</h1>  
<script language="JavaScript" type="text/javascript">  
test1 = "This is a test. ";  
test2 = "This is only a test.";  
both = test1 + test2;  
alert(both);  
</script>  
</body>  
</html>
```

This script assigns values to two string variables, `test1` and `test2`, and then displays an alert with their combined value. If you load this HTML document in a browser, your output should resemble Figure 5.1.

FIGURE 5.1

The output of the string example script.



In addition to using the `+` operator to concatenate two strings, you can use the `+=` operator to add text to a string. For example, this statement adds a period to the current contents of the string `sentence`:

```
sentence += ".";
```

By the Way

The plus sign (`+`) is also used to add numbers in JavaScript. The browser knows whether to use addition or concatenation based on the types of data you use with the plus sign. If you use it between a number and a string, the number is converted to a string and concatenated.

Calculating the String's Length

From time to time, you might find it useful to know how many characters a string variable contains. You can do this with the `length` property of `String` objects, which you can use with any string. To use this property, type the string's name followed by `.length`.

For example, `test.length` refers to the length of the `test` string. Here is an example of this property:

```
test = "This is a test.";
document.write(test.length);
```

The first statement assigns the string `This is a test` to the `test` variable. The second statement displays the length of the string—in this case, 15 characters. The `length` property is a read-only property, so you cannot assign a value to it to change a string's length.

Remember that although `test` refers to a string variable, the value of `test.length` is a number and can be used in any numeric expression.

**By the
Way**

Converting the String's Case

Two methods of the `String` object enable you to convert the contents of a string to all uppercase or all lowercase:

- ▶ `toUpperCase()`—Converts all characters in the string to uppercase.
- ▶ `toLowerCase()`—Converts all characters in the string to lowercase.

For example, the following statement displays the value of the `test` string variable in lowercase:

```
document.write(test.toLowerCase());
```

Assuming that this variable contained the text `This Is A Test`, the result would be the following string:

```
this is a test
```

Note that the statement doesn't change the value of the `test` variable. These methods return the upper- or lowercase version of the string, but they don't change the string itself. If you want to change the string's value, you can use a statement like this:

```
test = test.toLowerCase();
```

Note that the syntax for these methods is similar to the `length` property introduced earlier. The difference is that methods always use parentheses, whereas properties don't. The `toUpperCase` and `toLowerCase` methods do not take any parameters, but you still need to use the parentheses.

**By the
Way**

Working with Substrings

So far, you've worked with entire strings. JavaScript also enables you to work with *substrings*, or portions of a string. You can use the `substring` method to retrieve a portion of a string, or the `charAt` method to get a single character. These are explained in the following sections.

Using Part of a String

The `substring` method returns a string consisting of a portion of the original string between two index values, which you must specify in parentheses. For example, the following statement displays the fourth through sixth characters of the text string:

```
document.write(text.substring(3,6));
```

At this point, you're probably wondering where the 3 and the 6 come from. There are three things you need to understand about the index parameters:

- ▶ Indexing starts with 0 for the first character of the string, so the fourth character is actually index 3.
- ▶ The second index is noninclusive. A second index of 6 includes up to index 5 (the sixth character).
- ▶ You can specify the two indexes in either order. The smaller one will be assumed to be the first index. In the previous example, (6,3) would have produced the same result. Of course, there is rarely a reason to use the reverse order.

As another example, suppose you defined a string called `alpha` to hold the alphabet:

```
alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

The following are examples of the `substring()` method using this string:

- ▶ `alpha.substring(0,4)` returns ABCD.
- ▶ `alpha.substring(10,12)` returns KL.
- ▶ `alpha.substring(12,10)` also returns KL. Because it's smaller, 10 is used as the first index.
- ▶ `alpha.substring(6,7)` returns G.
- ▶ `alpha.substring(24,26)` returns YZ.
- ▶ `alpha.substring(0,26)` returns the entire alphabet.
- ▶ `alpha.substring(6,6)` returns the null value, an empty string. This is true whenever the two index values are the same.

Getting a Single Character

The `charAt` method is a simple way to grab a single character from a string. You specify the character's index, or position, in parentheses. The indexes begin at 0 for the first character. Here are a few examples using the `alpha` string:

- ▶ `alpha.charAt(0)` returns A.
- ▶ `alpha.charAt(12)` returns M.
- ▶ `alpha.charAt(25)` returns Z.
- ▶ `alpha.charAt(27)` returns an empty string because there is no character at that position.

Finding a Substring

Another use for substrings is to find a string within another string. One way to do this is with the `indexOf` method. To use this method, add `indexOf` to the string you want to search, and specify the string to search for in the parentheses. This example searches for "this" in the `test` string:

```
loc = test.indexOf("this");
```

As with most JavaScript methods and property names, `indexOf` is case sensitive. Make sure you type it exactly as shown here when you use it in scripts.

**By the
Way**

The value returned in the `loc` variable is an index into the string, similar to the first index in the substringing method. The first character of the string is index 0.

You can specify an optional second parameter to indicate the index value to begin the search. For example, this statement searches for the word `fish` in the `temp` string, starting with the 20th character:

```
location = temp.indexOf("fish",19);
```

One use for the second parameter is to search for multiple occurrences of a string. After finding the first occurrence, you search starting with that location for the second one, and so on.

**By the
Way**

A second method, `lastIndexOf()`, works the same way, but finds the *last* occurrence of the string. It searches the string backwards, starting with the last character. For example, this statement finds the last occurrence of `Fred` in the `names` string:

```
location = names.lastIndexOf("Fred");
```

As with `indexOf()`, you can specify a location to search from as the second parameter. In this case, the string will be searched backward starting at that location.

Using Numeric Arrays

An array is a numbered group of data items that you can treat as a single unit. For example, you might use an array called `scores` to store several scores for a game. Arrays can contain strings, numbers, objects, or other types of data. Each item in an array is called an *element* of the array.

Creating a Numeric Array

Unlike most other types of JavaScript variables, you typically need to declare an array before you use it. The following example creates an array with four elements:

```
scores = new Array(4);
```

To assign a value to the array, you use an *index* in brackets. Indexes begin with 0, so the elements of the array in this example would be numbered 0 to 3. These statements assign values to the four elements of the array:

```
scores[0] = 39;  
scores[1] = 40;  
scores[2] = 100;  
scores[3] = 49;
```

You can also declare an array and specify values for elements at the same time. This statement creates the same `scores` array in a single line:

```
scores = new Array(39,40,100,49);
```

In JavaScript 1.2 and later, you can also use a shorthand syntax to declare an array and specify its contents. The following statement is an alternative way to create the `scores` array:

```
scores = [39,40,100,49];
```

Did you Know?

Remember to use parentheses when declaring an array with the `new` keyword, as in `a=new Array(3,4,5)`, and use brackets when declaring an array without `new`, as in `a=[3,4,5]`. Otherwise, you'll run into JavaScript errors.

Understanding Array Length

Like strings, arrays have a `length` property. This tells you the number of elements in the array. If you specified the length when creating the array, this value becomes the `length` property's value. For example, these statements would print the number 30:

```
scores = new Array(30);
document.write(scores.length);
```

You can declare an array without a specific length, and change the length later by assigning values to elements or changing the `length` property. For example, these statements create a new array and assign values to two of its elements:

```
test = new Array();
test[0]=21;
test[5]=22;
```

In this example, because the largest index number assigned so far is 5, the array has a `length` property of 6—remember, elements are numbered starting at 0.

Accessing Array Elements

You can read the contents of an array using the same notation you used when assigning values. For example, the following statements would display the values of the first three elements of the `scores` array:

```
scoredisp = "Scores: " + scores[0] + ", " + scores[1] + ", " + scores[2];
document.write(scoredisp);
```

Looking at this example, you might imagine it would be inconvenient to display all the elements of a large array. This is an ideal job for loops, which enable you to perform the same statements several times with different values. You'll learn all about loops in Hour 7.

***Did you
Know?***

Using String Arrays

So far, you've used arrays of numbers. JavaScript also allows you to use *string arrays*, or arrays of strings. This is a powerful feature that enables you to work with a large number of strings at the same time.

Creating a String Array

You declare a string array in the same way as a numeric array—in fact, JavaScript does not make a distinction between them:

```
names = new Array(30);
```

You can then assign string values to the array elements:

```
names[0] = "Henry J. Tillman";  
names[1] = "Sherlock Holmes";
```

As with numeric arrays, you can also specify a string array's contents when you create it. Either of the following statements would create the same string array as the preceding example:

```
names = new Array("Henry J. Tillman", "Sherlock Holmes");  
names = ["Henry J. Tillman", "Sherlock Holmes"];
```

You can use string array elements anywhere you would use a string. You can even use the string methods introduced earlier. For example, the following statement prints the first five characters of the first element of the `names` array, resulting in `Henry`:

```
document.write(names[0].substring(0,5));
```

Splitting a String

JavaScript includes a string method called `split`, which splits a string into its component parts. To use this method, specify the string to split and a character to divide the parts:

```
test = "John Q. Public";  
parts = test.split(" ");
```

In this example, the `test` string contains the name `John Q. Public`. The `split` method in the second statement splits the name string at each space, resulting in three strings. These are stored in a string array called `parts`. After the example statements execute, the elements of `parts` contain the following:

- ▶ `parts[0]` = `"John"`
- ▶ `parts[1]` = `"Q."`
- ▶ `parts[2]` = `"Public"`

JavaScript also includes an array method, `join`, which performs the opposite function. This statement reassembles the `parts` array into a string:

```
fullname = parts.join(" ");
```

The value in the parentheses specifies a character to separate the parts of the array. In this case, a space is used, resulting in the final string John Q. Public. If you do not specify a character, commas are used.

Sorting a String Array

JavaScript also includes a sort method for arrays, which returns an alphabetically sorted version of the array. For example, the following statements initialize an array of four names and sort it:

```
names[0] = "Public, John Q.";
names[1] = "Tillman, Henry J.";
names[2] = "Bush, George W.";
names[3] = "Mouse, Mickey";
sortednames = names.sort();
```

The last statement sorts the names array and stores the result in a new array, sortednames.

Sorting a Numeric Array

Because the sort method sorts alphabetically, it won't work with a numeric array—at least not the way you'd expect. If an array contains the numbers 4, 10, 30, and 200, for example, it would sort them as 10, 200, 30, 4—not even close. Fortunately, there's a solution: You can specify a function in the sort method's parameters, and that function will be used to compare the numbers. The following code sorts a numeric array correctly:

```
function numcompare(a,b) {
    return a-b;
}
nums = new Array(30, 10, 200, 4);
sortednums = nums.sort(numcompare);
```

This example defines a simple function, numcompare, which subtracts the two numbers. After you specify this function in the sort method, the array is sorted in the correct numeric order: 4, 10, 30, 200.

JavaScript expects the comparison function to return a negative number if a belongs before b, 0 if they are the same, or a positive number if a belongs after b. This is why a - b is all you need for the function to sort numerically.



Try It Yourself

Sorting and Displaying Names

To gain more experience working with JavaScript's string and array features, you can create a script that enables the user to enter a list of names, and displays the list in sorted form.

Because this will be a larger script, you will create separate HTML and JavaScript files, as described in Hour 3, "Getting Started with JavaScript Programming." First, the `sort.html` file will contain the HTML structure and form fields for the script to work with. Listing 5.2 shows the HTML document.

LISTING 5.2 The HTML Document for the Sorting Example

```
<html>
<head>
<title>Array Sorting Example</title>
<script type="text/javascript" language="javascript" src="sort.js">
</script>
</head>
<body>
<h1>Sorting String Arrays</h1>
<p>Enter two or more names in the field below,
and the sorted list of names will appear in the
text area.</p>
<form name="theform">
Name:
<input type="text" name="newname" size="20">
<input type="button" name="addname" value="Add"
onclick="SortNames();">
<br>
<h2>Sorted Names</h2>
<textarea cols="60" rows="10" name="sorted">
The sorted names will appear here.
</textarea>
</form>
</body>
</html>
```

Because the script will be in a separate document, the `<script>` tag in the header of this document uses the `src` attribute to include a JavaScript file called `sort.js`. You will create this file next.

This document defines a form named `theform`, a text field named `newname`, an `addname` button, and a textarea named `sorted`. Your script will use these form fields as its user interface. Listing 5.3 shows the JavaScript file.

LISTING 5.3 The JavaScript File for the Sorting Example

```
// initialize the counter and the array
var numnames=0;
var names = new Array();
function SortNames() {
```


LISTING 5.3 Continued

```
// Get the name from the text field
thename=document.theform.newname.value;
// Add the name to the array
names[numnames]=thename;
// Increment the counter
numnames++;
// Sort the array
names.sort();
document.theform.sorted.value=names.join("\n");
}
```

The script begins by defining two variables with the `var` keyword: `numnames` will be a counter that increments as each name is added, and the `names` array will store the names.

When you type a name into the text field and click the button, the `onclick` event handler calls the `SortNames` function. This function stores the text field value in a variable, `thename`, and then adds the name to the `names` array using `numnames` as the index. It then increments `numnames` to prepare for the next name.

The final section of the script sorts the names and displays them. First, the `sort()` method is used to sort the `names` array. Next, the `join()` method is used to combine the names, separating them with line breaks, and display them in the text area.

To test the script, save it as `sort.js`, and then load the `sort.html` file you created previously into a browser. You can then add some names and test the script. Figure 5.2 shows the result after sorting several names.

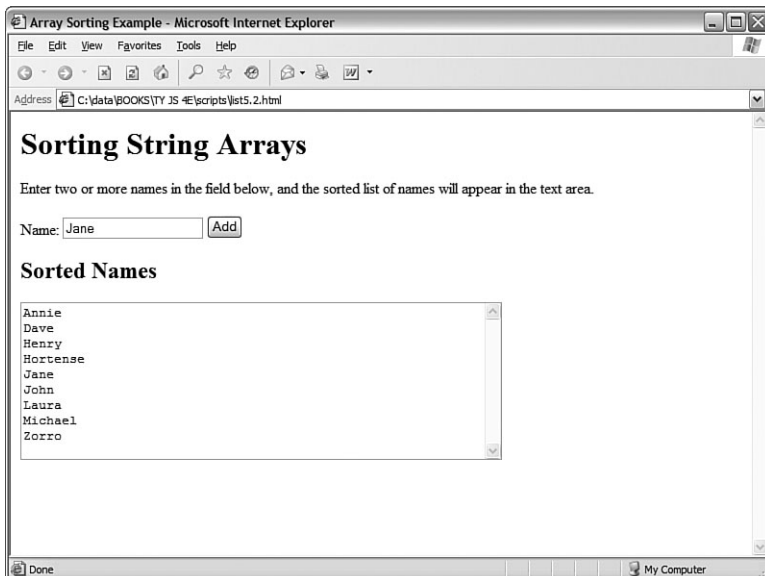


FIGURE 5.2
The output of
the name-sort-
ing example.

Summary

During this hour, you've focused on variables and how JavaScript handles them. You've learned how to name variables, how to declare them, and the differences between local and global variables. You also explored the data types supported by JavaScript and how to convert between them.

You also learned about JavaScript's more complex variables, strings and arrays, and looked at the features that enable you to perform operations on them, such as converting strings to uppercase or sorting arrays.

In the next hour, you'll continue your JavaScript education by learning more about two additional key features: functions and objects.

Q&A

Q. *What is the importance of the `var` keyword? Should I always use it to declare variables?*

A. You only need to use `var` to define a local variable in a function. However, if you're unsure at all, it's always safe to use `var`. Using it consistently will help you keep your scripts organized and error free.

Q. *Is there any reason I would want to use the `var` keyword to create a local variable with the same name as a global one?*

A. Not on purpose. The main reason to use `var` is to avoid conflicts with global variables you might not know about. For example, you might add a global variable in the future, or you might add another script to the page that uses a similar variable name. This is more of an issue with large, complex scripts.

Q. *What good are Boolean variables?*

A. Often in scripts you'll need a variable to indicate whether something has happened—for example, whether a phone number the user has entered is in the right format. Boolean variables are ideal for this; they're also useful in working with conditions, as you'll see in Hour 7.

Q. *Can I store other types of data in an array? For example, can I have an array of dates?*

A. Absolutely. JavaScript allows you to store any data type in an array.

Q. *What about two-dimensional arrays?*

A. These are arrays with two indexes (such as columns and rows). JavaScript does not directly support this type of array, but you can use objects to achieve the same effect. You will learn more about objects in the next hour.

Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

1. Which of the following is *not* a valid JavaScript variable name?
 - a. 2names
 - b. first_and_last_names
 - c. FirstAndLast
2. If the statement `var fig=2` appears in a function, which type of variable does it declare?
 - a. A global variable
 - b. A local variable
 - c. A constant variable
3. If the string `test` contains the value `The eagle has landed.`, what would be the value of `test.length`?
 - a. 4
 - b. 21
 - c. The
4. Using the same example string, which of these statements would return the word `eagle`?
 - a. `test.substring(4,9)`
 - b. `test.substring(5,9)`
 - c. `test.substring("eagle")`
5. What will be the result of the JavaScript expression `31 + "angry polar bears"`?
 - a. An error message
 - b. 32
 - c. "31 angry polar bears"

Quiz Answers

1. a. 2names is an invalid JavaScript variable name because it begins with a number. The others are valid, although they're probably not ideal choices for names.
2. b. Because the variable is declared in a function, it is a local variable. The var keyword ensures that a local variable is created.
3. b. The length of the string is 21 characters.
4. a. The correct statement is `test.substring(4,9)`. Remember that the indexes start with 0, and that the second index is noninclusive.
5. c. JavaScript converts the whole expression to the string "31 angry polar bears". (No offense to polar bears, who are seldom angry and rarely seen in groups this large.)

Exercises

To further explore JavaScript variables, strings, and arrays, you can perform the following exercises:

- ▶ Modify the sorting example in Listing 5.3 to convert the names to all uppercase before sorting and displaying them.
- ▶ Modify Listing 5.3 to display a numbered list of names in the textarea.

HOURL 6

Using Functions and Objects

What You'll Learn in This Hour:

- ▶ Defining and calling functions
- ▶ Returning values from functions
- ▶ Understanding JavaScript objects
- ▶ Defining custom objects
- ▶ Working with object properties and values
- ▶ Defining and using object methods
- ▶ Using objects to store data and related functions

In this hour, you'll learn about two more key JavaScript concepts that you'll use throughout the rest of this book. First, you'll learn the details of using functions, which enable you to group any number of statements into a block. This is useful for repeating sections of code, and you can also create functions that accept parameters and return values for later use.

Whereas functions enable you to group sections of code, objects enable you to group data—you can use them to combine related data items and functions for working with the data.

Using Functions

The scripts you've seen so far are simple lists of instructions. The browser begins with the first statement after the `<script>` tag and follows each instruction in order until it reaches the closing `</script>` tag (or encounters an error).

Although this is a straightforward approach for short scripts, it can be confusing to read a longer script written in this fashion. To make it easier for you to organize your scripts, JavaScript supports functions, which you learned about in Hour 3, "Getting Started with JavaScript Programming." In this section, you will learn how to define and use functions.

Defining a Function

Functions are groups of JavaScript statements that can be treated as a single unit. To use a function, you must first define it. Here is a simple example of a function definition:

```
function Greet() {  
    alert("Greetings.");  
}
```

This defines a function that displays an alert message to the user. This begins with the `function` keyword. The function's name is `Greet`. Notice the parentheses after the function's name. As you'll learn next, the space between them is not always empty.

The first and last lines of the function definition include braces (`{` and `}`). You use these to enclose all of the statements in the function. The browser uses the braces to determine where the function begins and ends.

Between the braces, this particular function contains a single line. This uses the built-in `alert()` function, which displays an alert message. The message will contain the text "Greetings."

By the Way

Function names are case sensitive. If you define a function such as `Greet()` with a capital letter, be sure you use the identical name when you call the function.

Now, about those parentheses. The current `Greet()` function always does the same thing: Each time you use it, it displays the same message. Although this avoids a bit of typing, it doesn't really provide much of an advantage.

To make your function more flexible, you can add *parameters*, also known as *arguments*. These are variables that are received by the function each time it is called. For example, you can add a parameter called `who` that tells the function the name of the person to greet. Here is the modified `Greet()` function:

```
function Greet(who) {  
    alert("Greetings, " + who);  
}
```

Of course, to use this function, you should include it in an HTML document. Traditionally, the best place for a function definition is within the `<head>` section of the document. Because the statements in the `<head>` section are executed first, this ensures that the function is defined before it is used.

Listing 6.1 shows the `Greet()` function embedded in the header section of an HTML document.

LISTING 6.1 The Greet() Function in an HTML Document

```
<html>
<head>
<title>Functions</title>
<script language="JavaScript" type="text/javascript">
function Greet(who) {
    alert("Greetings, " + who);
}
</script>
</head>
<body>
This is the body of the page.
</body>
</html>
```

As usual, you can download the listings for this hour or view them online at this book's website.

**By the
Way**

Calling the Function

You have now defined a function and placed it in an HTML document. However, if you load Listing 6.1 into a browser, you'll notice that it does absolutely nothing. This is because the function is defined—ready to be used—but we haven't used it yet.

Making use of a function is referred to as *calling* the function. To call a function, use the function's name as a statement in a script. You will need to include the parentheses and the values for the function's parameters. For example, here's a statement that calls the Greet function:

```
Greet("Fred");
```

This tells the JavaScript interpreter to transfer control to the first statement in the Greet function. It also passes the parameter "Fred" to the function. This value will be assigned to the who variable inside the function.

Functions can have more than one parameter. To define a function with multiple parameters, list a variable name for each parameter, separated by commas. To call the function, specify values for each parameter separated by commas.

**By the
Way**

Listing 6.2 shows a complete HTML document that includes the function definition and a second script in the body of the page that actually calls the function. To demonstrate the usefulness of functions, we'll call it twice to greet two different people.

LISTING 6.2 The Complete Function Example

```

<html>
<head>
<title>Functions</title>
<script language="JavaScript" type="text/javascript">
function Greet(who) {
    alert("Greetings, " + who);
}
</script>
</head>
<body>
<h1>Function Example</h1>
<p>Prepare to be greeted twice.</p>
<script language="JavaScript" type="text/javascript">
Greet("Fred");
Greet("Ethel");
</script>
</body>
</html>>

```

This listing includes a second set of `<script>` tags in the body of the page. The second script includes two function calls to the `Greet` function, each with a different name.

Now that you have a script that actually does something, try loading it into a browser. You should see something like Figure 6.1, which shows the Greeting script running in Firefox.

FIGURE 6.1
The output of
the Greeting
example.



By the Way

Notice that the second alert message isn't displayed until you press the OK button on the first alert. This is because JavaScript processing is halted while alerts are displayed.

Returning a Value

The function you just created displays a message to the user, but functions can also return a value to the script that called them. This allows you to use functions to calculate values. As an example, you can create a function that averages four numbers.

Your function should begin with the function keyword, the function's name, and the parameters it accepts. We will use the variable names *a*, *b*, *c*, and *d* for the four numbers to average. Here is the first line of the function:

```
function Average(a,b,c,d) {
```

I've also included the opening brace (*{*) on the first line of the function. This is a common style, but you can also place the brace on the next line, or on a line by itself.

**By the
Way**

Next, the function needs to calculate the average of the four parameters. You can calculate this by adding them, and then dividing by the number of parameters (in this case, 4). Thus, here is the next line of the function:

```
result = (a + b + c + d) / 4;
```

This statement creates a variable called *result* and calculates the result by adding the four numbers, and then dividing by 4. (The parentheses are necessary to tell JavaScript to perform the addition before the division.)

To send this result back to the script that called the function, you use the *return* keyword. Here is the last part of the function:

```
return result;  
}
```

Listing 6.3 shows the complete *Average()* function in an HTML document. This HTML document also includes a small script in the *<body>* section that calls the *Average()* function and displays the result.

LISTING 6.3 The *Average()* Function in an HTML Document

```
<html>_  
<head>  
<title>Function Example</title>  
<script language="JavaScript" type="text/javascript">  
function Average(a,b,c,d) {  
result = (a + b + c + d) / 4;  
return result;  
}  
</script>  
</head>  
<body>  
<p>The following is the result of the function call.</p>  
<script LANGUAGE="JavaScript" type="text/javascript">  
score = Average(3,4,5,6);  
document.write("The average is: " + score);  
</script>_  
</body>  
</html>
```

You can use a variable with the function call, as shown in this listing. This statement averages the numbers 3, 4, 5, and 6 and stores the result in a variable called `score`:

```
score = Average(3,4,5,6);
```

Did you Know?

You can also use the function call directly in an expression. For example, you could use the `alert` statement to display the result of the function:
`alert(Average(1,2,3,4))` .

Introducing Objects

In the previous hour, you learned how to use variables to represent different kinds of data in JavaScript. JavaScript also supports *objects*, a more complex kind of variable that can store multiple data items and functions.

Although a variable can have only one value at a time, an object can contain multiple values, as well as functions for working with the values. This allows you to group related data items and the functions that deal with them into a single object.

In this hour, you'll learn how to define and use your own objects. You've already worked with some objects:

- ▶ **DOM objects**—Allow your scripts to interact with web pages. You learned about these in Hour 4, “Working with the Document Object Model (DOM).”
- ▶ **Built-in objects**—Include strings and arrays, which you learned about in Hour 5, “Using Variables, Strings, and Arrays.”

The syntax for working with all three types of objects—DOM objects, built-in objects, and custom objects—is the same, so even if you don't end up creating your own objects, you should have a good understanding of JavaScript's object terminology and syntax.

Creating Objects

When you created an array in the previous hour, you used the following JavaScript statement:

```
scores = new Array(4);
```

The `new` keyword tells the JavaScript interpreter to use a function—in this case, the built-in `Array` function—to create an object. You'll create a function for a custom object later in this hour.

Object Properties and Values

Each object has one or more *properties*—essentially, variables that will be stored within the object. For example, in Hour 4, you learned that the `location.href` property gives you the URL of the current document. The `href` property is one of the properties of the `location` object in the DOM.

You’ve also used the `length` property of `String` objects, as in the following example from the previous hour:

```
test = "This is a test.";
document.write(test.length);
```

Like variables, each object property has a *value*. To read a property’s value, you simply include the object name and property name, separated by a period, in any expression, as in `test.length` previously. You can change a property’s value using the `=` operator, just like a variable. The following example sends the browser to a new URL by changing the `location.href` property:

```
location.href="http://www.jsworkshop.com/";
```

An object can also be a property of another object. This is referred to as a *child object*.

**By the
Way**

Understanding Methods

Along with properties, each object can have one or more *methods*. These are functions that work with the object’s data. For example, the following JavaScript statement reloads the current document, as you learned in Hour 4:

```
location.reload();
```

When you use `reload()`, you’re using a method of the `location` object. Like normal functions, methods can accept arguments in parentheses, and can return values.

Using Objects to Simplify Scripting

Although JavaScript’s variables and arrays are versatile ways to store data, sometimes you need a more complicated structure. For example, suppose you are creating a script to work with a business card database that contains names, addresses, and phone numbers for a variety of people.

If you were using regular variables, you would need several separate variables for each person in the database: a name variable, an address variable, and so on. This would be very confusing.

Arrays would improve things slightly. You could have a names array, an addresses array, and a phone number array. Each person in the database would have an entry in each array. This would be more convenient, but still not perfect.

With objects, you can make the variables that store the database as logical as business cards. Each person is represented by a Card object, which has properties for name, address, and phone number. You can even add methods to the object to display or work with the information.

In the following sections, you'll use JavaScript to actually create the Card object and its properties and methods. Later in this hour, you'll use the Card object in a script to display information for several members of the database.

Defining an Object

The first step in creating an object is to name it and its properties. We've already decided to call the object a Card object. Each object will have the following properties:

- ▶ name
- ▶ address
- ▶ workphone
- ▶ homephone

The first step in using this object in a JavaScript program is to create a function to make new Card objects. This function is called the *constructor* for an object. Here is the constructor function for the Card object:

```
function Card(name,address,work,home) {  
    this.name = name;  
    this.address = address;  
    this.workphone = work;  
    this.homephone = home;  
}
```

The constructor is a simple function that accepts parameters to initialize a new object and assigns them to the corresponding properties. This function accepts several parameters from the statement that calls the function, and then assigns them as properties of an object. Because the function is called Card, the object is the Card object.

Notice the `this` keyword. You'll use it anytime you create an object definition. Use `this` to refer to the current object—the one that is being created by the function.

Defining an Object Method

Next, you will create a method to work with the Card object. Because all Card objects will have the same properties, it might be handy to have a function that prints out the properties in a neat format. Let's call this function `PrintCard()`.

Your `PrintCard()` function will be used as a method for Card objects, so you don't need to ask for parameters. Instead, you can use the `this` keyword again to refer to the current object's properties. Here is a function definition for the `PrintCard()` function:

```
function PrintCard() {  
    line1 = "Name: " + this.name + "<br>\n";  
    line2 = "Address: " + this.address + "<br>\n";  
    line3 = "Work Phone: " + this.workphone + "<br>\n";  
    line4 = "Home Phone: " + this.homephone + "<hr>\n";  
    document.write(line1, line2, line3, line4);  
}
```

This function simply reads the properties from the current object (`this`), prints each one with a caption, and skips to a new line.

You now have a function that prints a card, but it isn't officially a method of the Card object. The last thing you need to do is make `PrintCard()` part of the function definition for Card objects. Here is the modified function definition:

```
function Card(name,address,work,home) {  
    this.name = name;  
    this.address = address;  
    this.workphone = work;  
    this.homephone = home;  
    this.PrintCard = PrintCard;  
}
```

The added statement looks just like another property definition, but it refers to the `PrintCard()` function. This will work so long as the `PrintCard()` function is defined with its own function definition. Methods are essentially properties that define a function rather than a simple value.

The previous example uses lowercase names such as `workphone` for properties, and an uppercase name (`PrintCard`) for the method. You can use any case for property and method names, but this is one way to make it clear that `PrintCard` is a method rather than an ordinary property.

***Did you
Know?***

Creating an Object Instance

Now let's use the object definition and method you just created. To use an object definition, you create a new object. This is done with the `new` keyword. This is the same keyword you've already used to create `Date` and `Array` objects.

The following statement creates a new Card object called tom:

```
tom = new Card("Tom Jones", "123 Elm Street", "555-1234", "555-9876");
```

As you can see, creating an object is easy. All you do is call the Card() function (the object definition) and give it the required attributes, in the same order as the definition.

After this statement executes, a new object is created to hold Tom's information. This is called an *instance* of the Card object. Just as there can be several string variables in a program, there can be several instances of an object you define.

Rather than specify all the information for a card with the new keyword, you can assign them after the fact. For example, the following script creates an empty Card object called holmes, and then assigns its properties:

```
holmes = new Card();  
holmes.name = "Sherlock Holmes";  
holmes.address = "221B Baker Street";  
holmes.workphone = "555-2345";  
holmes.homephone = "555-3456";
```

After you've created an instance of the Card object using either of these methods, you can use the PrintCard() method to display its information. For example, this statement displays the properties of the tom card:

```
tom.PrintCard();
```

Extending Built-in Objects

JavaScript includes a feature that enables you to extend the definitions of built-in objects. For example, if you think the String object doesn't quite fit your needs, you can extend it, adding a new property or method. This might be very useful if you were creating a large script that used many strings.

You can add both properties and methods to an existing object by using the prototype keyword. (A *prototype* is another name for an object's definition, or constructor function.) The prototype keyword enables you to change the definition of an object outside its constructor function.

As an example, let's add a method to the String object definition. You will create a method called heading, which converts a string into an HTML heading. The following statement defines a string called title:

```
title = "Fred's Home Page";
```

This statement would output the contents of the title string as an HTML level 1 heading:

```
document.write(title.heading(1));
```

Listing 6.4 adds a heading method to the String object definition that will display the string as a heading, and then displays three headings using the method.

LISTING 6.4 Adding a Method to the String Object

```
<html>
<head><title>Test of heading method</title>
</head>
<body>
<script LANGUAGE="JavaScript" type="text/javascript">
function addhead (level) {
    html = "H" + level;
    text = this.toString();
    start = "<" + html + ">";
    stop = "</" + html + ">";
    return start + text + stop;
}
String.prototype.heading = addhead;
document.write ("This is a heading 1".heading(1));
document.write ("This is a heading 2".heading(2));
document.write ("This is a heading 3".heading(3));
</script>
</body>
</html>
```

First, you define the `addhead()` function, which will serve as the new string method. It accepts a number to specify the heading level. The `start` and `stop` variables are used to store the HTML “begin header” and “end header” tags, such as `<h1>` and `</h1>`.

After the function is defined, use the `prototype` keyword to add it as a method of the String object. You can then use this method on any String object or, in fact, any JavaScript string. This is demonstrated by the last three statements, which display quoted text strings as level 1, 2, and 3 headers.

Try It Yourself



Storing Data in Objects

Now you’ve created a new object to store business cards and a method to print them out. As a final demonstration of objects, properties, functions, and methods, you will now use this object in a web page to display data for several cards.

Your script will need to include the function definition for `PrintCard()`, along with the function definition for the Card object. You will then create three cards and print them out in the body of the document. We will use separate HTML and JavaScript files for this example. Listing 6.5 shows the complete script.

LISTING 6.5 An Example Script That Uses the Card Object

```
// define the functions
function PrintCard() {
  line1 = "<b>Name: </b>" + this.name + "<br>\n";
  line2 = "<b>Address: </b>" + this.address + "<br>\n";
  line3 = "<b>Work Phone: </b>" + this.workphone + "<br>\n";
  line4 = "<b>Home Phone: </b>" + this.homephone + "<hr>\n";
  document.write(line1, line2, line3, line4);
}
function Card(name,address,work,home) {
  this.name = name;
  this.address = address;
  this.workphone = work;
  this.homephone = home;
  this.PrintCard = PrintCard;
}
// Create the objects
sue = new Card("Sue Suthers", "123 Elm Street", "555-1234", "555-9876");
phred = new Card("Phred Madsen", "233 Oak Lane", "555-2222", "555-4444");
henry = new Card("Henry Tillman", "233 Walnut Circle", "555-1299", "555-1344");
// And print them
sue.PrintCard();
phred.PrintCard();
henry.PrintCard();
```

Notice that the `PrintCard()` function has been modified slightly to make things look good with the captions in boldface. To use this script, save it as `cardtest.js`. Next, you'll need to include the script in a simple HTML document. Listing 6.6 shows the HTML document for this example.

LISTING 6.6 The HTML File for the Card Object Example

```
<html>
<head>
<title>JavaScript Business Cards</title>
</head>
<body>
<h1>JavaScript Business Card Test</h1>
<p>Script begins here.</p><hr>
<script language="JavaScript" type="text/javascript"
  src="cardtest.js">
</script>
<p>End of script.</p>
</body>
</html>
```

To test the script, save the HTML document in the same directory as the `cardtest.js` file you created earlier, and then load the HTML document into a browser. The browser's display of this example is shown in Figure 6.2.

This example isn't a very sophisticated database because you have to include the data for each person in the script. However, an object like this could be used to store a database record retrieved from a database server with thousands of records.

**By the
Way**

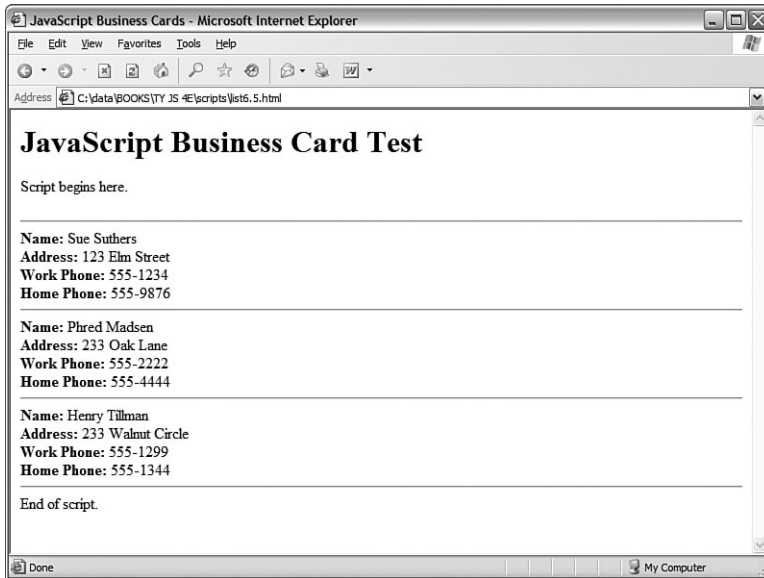


FIGURE 6.2
Internet Explorer displays the output of the business card example.

Summary

In this hour, you've looked at two important features of JavaScript. First, you learned how to use functions to group JavaScript statements, and how to call functions and use the values they return.

You also learned about JavaScript's object-oriented features—defining objects with constructor functions, creating object instances, and working with properties, property values, and methods.

In the next hour, you'll look at two more features you'll use in almost every script—conditions to let your scripts evaluate data, and loops to repeat sections of code.

Q&A

- Q.** *Many objects in JavaScript, such as DOM objects, include parent and child objects. Can I include child objects in my custom object definitions?*
- A.** Yes. Just create a constructor function for the child object, and then add a property to the parent object that corresponds to it. For example, if you created a Nicknames object to store several nicknames for a person in the card file example, you could add it as a child object in the Card object's constructor:
- ```
this.nick = new Nicknames();
```
- Q.** *Can I create an array of custom objects?*
- A.** Yes. First, create the object definition as usual and define an array with the required number of elements. Then assign a new object to each array element (for example, `cardarray[1] = new Card();`). You can use a loop, described in the next hour, to assign objects to an entire array at once.
- Q.** *Can I modify all properties of objects?*
- A.** With custom objects, yes—but this varies with built-in objects and DOM objects. For example, you can use the `length` property to find the length of a string, but it is a *read-only property* and cannot be modified.

## Quiz Questions

Test your knowledge of JavaScript by answering the following questions:

1. What JavaScript keyword is used to create an instance of an object?
  - a. `object`
  - b. `new`
  - c. `instance`
2. What is the meaning of the `this` keyword in JavaScript?
  - a. The current object.
  - b. The current script.
  - c. It has no meaning.

3. What does the prototype keyword allow you to do in a script?
  - a. Change the syntax of JavaScript commands.
  - b. Modify the definitions of built-in objects.
  - c. Modify the user's browser so only your scripts will work.

## Quiz Answers

1. b. The new keyword creates an object instance.
2. a. The this keyword refers to the current object.
3. b. The prototype keyword allows you to modify the definitions of built-in objects.

## Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Modify the Greet () function to accept two parameters, who1 and who2, and to include both names in a single greeting dialog. Modify Listing 6.2 to use a single function call to the new function.
- ▶ Modify the definition of the Card object to include a property called email for the person's email address. Modify the PrintCard () function in Listing 6.5 to include this property.

*This page intentionally left blank*

## HOURL 7

# Controlling Flow with Conditions and Loops

---

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

- ▶ Testing variables with the `if` statement
- ▶ Using various operators to compare values
- ▶ Using logical operators to combine conditions
- ▶ Using alternative conditions with `else`
- ▶ Creating expressions with conditional operators
- ▶ Testing for multiple conditions
- ▶ Performing repeated statements with the `for` loop
- ▶ Using `while` for a different type of loop
- ▶ Using `do...while` loops
- ▶ Creating infinite loops (and why you shouldn't)
- ▶ Escaping from loops and continuing loops
- ▶ Looping through an array's properties

Statements in a JavaScript program generally execute in the order in which they appear, one after the other. Because this isn't always practical, most programming languages provide *flow control* statements that let you control the order in which code is executed. Functions, which you learned about in the previous hour, are one type of flow control—although a function might be defined first thing in your code, its statements can be executed anywhere in the script.

In this hour, you'll look at two other types of flow control in JavaScript: conditions, which allow a choice of different options depending on a value, and loops, which allow repetitive statements.

## The `if` Statement

One of the most important features of a computer language is the capability to test and compare values. This allows your scripts to behave differently based on the values of variables, or based on input from the user.

The `if` statement is the main conditional statement in JavaScript. This statement means much the same in JavaScript as it does in English—for example, here is a typical conditional statement in English:

*If the phone rings, answer it.*

This statement consists of two parts: a condition (*If the phone rings*) and an action (*answer it*). The `if` statement in JavaScript works much the same way. Here is an example of a basic `if` statement:

```
if (a == 1) window.alert("Found a 1!");
```

This statement includes a condition (if `a` equals 1) and an action (display a message). This statement checks the variable `a` and, if it has a value of 1, displays an alert message. Otherwise, it does nothing.

If you use an `if` statement like the preceding example, you can use a single statement as the action. You can also use multiple statements for the action by enclosing them in braces (`{}`), as shown here:

```
if (a == 1) {
 window.alert("Found a 1!");
 a = 0;
}
```

This block of statements checks the variable `a` once again. If it finds a value of 1, it displays a message and sets `a` back to 0.

## Conditional Operators

The action part of an `if` statement can include any of the JavaScript statements you've already learned (and any others, for that matter), but the condition part of the statement uses its own syntax. This is called a *conditional expression*.

A conditional expression usually includes two values to be compared (in the preceding example, the values were `a` and 1). These values can be variables, constants, or even expressions in themselves.

Between the two values to be compared is a *conditional operator*. This operator tells JavaScript how to compare the two values. For instance, the `==` operator is used to test whether the two values are equal. A variety of conditional operators is available:

- ▶ `==` — Is equal to
- ▶ `!=` — Is not equal to
- ▶ `<` — Is less than
- ▶ `>` — Is greater than
- ▶ `>=` — Is greater than or equal to
- ▶ `<=` — Is less than or equal to

Be sure not to confuse the equality operator (`==`) with the assignment operator (`=`), even though they both might be read as “equals.” Remember to use `=` when *assigning* a value to a variable, and `==` when *comparing* values. Confusing these two is one of the most common mistakes in JavaScript programming.

**By the  
Way**

## Combining Conditions with Logical Operators

Often, you’ll want to check a variable for more than one possible value, or check more than one variable at once. JavaScript includes *logical operators*, also known as Boolean operators, for this purpose. For example, the following two statements check different conditions and use the same action:

```
if (phone == "") window.alert("error!");
if (email == "") window.alert("error!");
```

Using a logical operator, you can combine them into a single statement:

```
if (phone == "" || email == "") window.alert("Something's Missing!");
```

This statement uses the logical Or operator (`||`) to combine the conditions. Translated to English, this would be, “If the phone number is blank or the email address is blank, display an error message.”

An additional logical operator is the And operator, `&&`. Consider this statement:

```
if (phone == "" && email == "") window.alert("Both are Missing!");
```

This statement uses `&&` (And) instead of `||` (Or), so the error message will only be displayed if *both* the email address and phone number variables are blank. (In this particular case, Or is a better choice.)

***Did you  
Know?***

If the JavaScript interpreter discovers the answer to a conditional expression before reaching the end, it does not evaluate the rest of the condition. For example, if the first of two conditions separated by the `&&` operator is false, the second is not evaluated. You can take advantage of this to improve the speed of your scripts.

The third logical operator is the exclamation mark (`!`), which means Not. It can be used to invert an expression—in other words, a true expression would become false, and a false one would become true. For example, here's a statement that uses the Not operator:

```
if (!$phone == "") alert("phone is OK");
```

In this statement, the `!` (Not) operator inverts the condition, so the action of the `if` statement is executed only if the phone number variable is *not* blank. The extra parentheses are necessary because all JavaScript conditions must be in parentheses. You could also use the `!=` (Not equal) operator to simplify this statement:

```
if ($phone != "") alert("phone is OK");
```

As with the previous statement, this alerts you if the phone number field is not blank.

***Did you  
Know?***

The logical operators are powerful, but it's easy to accidentally create an impossible condition with them. For example, the condition `(a < 10 && a > 20)` might look correct at first glance. However, if you read it out loud, you get "If a is less than 10 and a is greater than 20"—an impossibility in our universe. In this case, Or (`||`) should have been used.

## The else Keyword

An additional feature of the `if` statement is the `else` keyword. Much like its English equivalent, `else` tells the JavaScript interpreter what to do if the condition isn't true. The following is a simple example of the `else` keyword in action:

```
if (a == 1) {
 alert("Found a 1!");
 a = 0;
}
else {
 alert("Incorrect value: " + a);
}
```

This is a modified version of the previous example. This displays a message and resets the variable `a` if the condition is met. If the condition is not met (if `a` is not 1), a different message is displayed.



Like the `if` statement, `else` can be followed either by a single action statement or by a number of statements enclosed in braces.

**By the  
Way**

## Using Shorthand Conditional Expressions

In addition to the `if` statement, JavaScript provides a shorthand type of conditional expression that you can use to make quick decisions. This uses a peculiar syntax that is also found in other languages, such as C. A conditional expression looks like this:

```
variable = (condition) ? (true action) : (false action);
```

This assigns one of two values to the variable: one if the condition is true, and another if it is false. Here is an example of a conditional expression:

```
value = (a == 1) ? 1 : 0;
```

This statement might look confusing, but it is equivalent to the following `if` statement:

```
if (a == 1)
 value = 1;
else
 value = 0;
```

In other words, the value after the question mark (?) will be used if the condition is true, and the value after the colon (:) will be used if the condition is false. The colon represents the `else` portion of this statement and, like the `else` portion of the `if` statement, is optional.

These shorthand expressions can be used anywhere JavaScript expects a value. They provide an easy way to make simple decisions about values. As an example, here's an easy way to display a grammatically correct message about a variable:

```
document.write("Found " + counter + ((counter == 1) ? " word." : " words."));
```

This will print the message `Found 1 word` if the `counter` variable has a value of 1, and `Found 2 words` if its value is 2 or greater. This is one of the most common uses for a conditional expression.

## Testing Multiple Conditions with `if` and `else`

You can now create an example script using `if` and `else`. In Hour 2, “Creating Simple Scripts,” you created a script that displays the current date and time.

This example will use conditions to display a greeting that depends on the time:

“Good morning,” “Good Afternoon,” “Good Evening,” or “Good Day”. To accomplish this, you can use a combination of several if statements:

```
if (hours < 10) document.write("Good morning.");
else if (hours >= 14 && hours <= 17) document.write("Good afternoon.");
else if (hours >= 17) document.write("Good evening.");
else document.write("Good day.");
```

The first statement checks the hours variable for a value less than 10—in other words, it checks whether the current time is before 10:00 a.m. If so, it displays the greeting “Good morning.”

The second statement checks whether the time is between 2:00 p.m. and 5:00 p.m. and, if so, displays “Good afternoon.” This statement uses `else if` to indicate that this condition will only be tested if the previous one failed—if it’s morning, there’s no need to check whether it’s afternoon. Similarly, the third statement checks for times after 5:00 p.m. and displays “Good evening.”

The final statement uses a simple `else`, meaning it will be executed if none of the previous conditions matched. This covers the times between 10:00 a.m. and 2:00 p.m. (neglected by the other statements) and displays “Good day.”

## The HTML File

To try this example in a browser, you’ll need an HTML file. We will keep the JavaScript code separate, so Listing 7.1 is the complete HTML file. Save it as `timegreet.html` but don’t load it into the browser until you’ve prepared the JavaScript file in the next section.

---

### LISTING 7.1 The HTML File for the Time and Greeting Example

```
<html>
<head><title>if statement example</title></head>
<body>
<h1>Current Date and Time</h1>
<p>
<script language="JavaScript" type="text/javascript"
 src = "timegreet.js">
</script>
</p>
</body>
</html>
```

---

## The JavaScript File

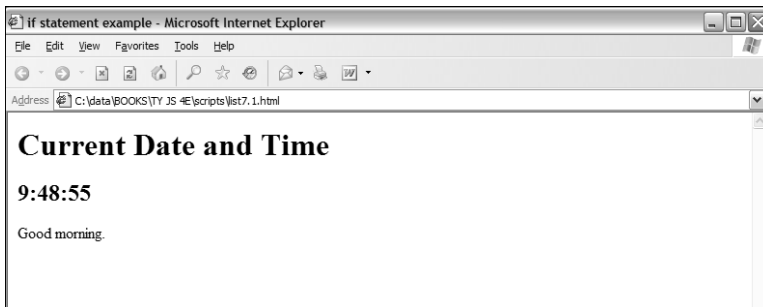
Listing 7.2 shows the complete JavaScript file for the time greeting example. This uses the built-in `Date` object functions to find the current date and store it in hours, mins,

and secs variables. Next, document.write statements display the current time, and the if and else statements introduced earlier display an appropriate greeting.

### LISTING 7.2 A Script to Display the Current Time and a Greeting

```
// Get the current date
now = new Date();
// Split into hours, minutes, seconds
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
// Display the time
document.write("<h2>");
document.write(hours + ":" + mins + ":" + secs);
document.write("</h2>");
// Display a greeting
document.write("<p>");
if (hours < 10) document.write("Good morning.");
else if (hours >= 14 && hours <= 17) document.write("Good afternoon.");
else if (hours > 17) document.write("Good evening.");
else document.write("Good day.");
document.write("</p>");
```

To try this example, save this file as timegreet.js (or download it from this book's website) and then load the timegreet.html file into your browser. Figure 7.1 shows the results of this script.



**FIGURE 7.1**  
The output of the time greet-  
ing example, as  
shown by  
Internet  
Explorer.

## Using Multiple Conditions with switch

In the previous example, you used several if statements in a row to test for different conditions. Here is another example of this technique:

```
if (button=="next") window.location="next.html";
else if (button=="previous") window.location="prev.html";
else if (button=="home") window.location="home.html";
else if (button=="back") window.location="menu.html";
```

Although this is a compact way of doing things, this method can get messy if each if statement has its own block of code with several statements. As an alternative, JavaScript includes the switch statement, which enables you to combine several tests of the same variable or expression into a single block of statements. The following shows the same example converted to use switch:

```
switch(button) {
 case "next":
 window.location="next.html";
 break;
 case "previous":
 window.location="prev.html";
 break;
 case "home":
 window.location="home.html";
 break;
 case "back":
 window.location="menu.html";
 break;
 default:
 window.alert("Wrong button.");
}
```

The switch statement has several components:

- ▶ The initial switch statement. This statement includes the value to test (in this case, button) in parentheses.
- ▶ Braces ({ and }) enclose the contents of the switch statement, similar to a function or an if statement.
- ▶ One or more case statements. Each of these statements specifies a value to compare with the value specified in the switch statement. If the values match, the statements after the case statement are executed. Otherwise, the next case is tried.
- ▶ The break statement is used to end each case. This skips to the end of the switch. If break is not included, statements in multiple cases might be executed whether they match or not.
- ▶ Optionally, the default case can be included and followed by one or more statements that are executed if none of the other cases were matched.

## Using for Loops

Loops are useful any time you need a section of code to execute more than once. The `for` keyword is the first tool to consider for creating loops. A `for` loop typically uses a variable (called a *counter* or an *index*) to keep track of how many times the loop has executed, and it stops when the counter reaches a certain number. A basic `for` statement looks like this:

```
for (var = 1; var < 10; var++) {
```

There are three parameters to the `for` loop, separated by semicolons:

- ▶ The first parameter (`var = 1` in the example) specifies a variable and assigns an initial value to it. This is called the *initial expression* because it sets up the initial state for the loop.
- ▶ The second parameter (`var < 10` in the example) is a condition that must remain true to keep the loop running. This is called the *condition* of the loop.
- ▶ The third parameter (`var++` in the example) is a statement that executes with each iteration of the loop. This is called the *increment expression* because it is typically used to increment the counter. The increment expression executes at the end of each loop iteration.

After the three parameters are specified, a left brace (`{`) is used to signal the beginning of a block. A right brace (`}`) is used at the end of the block. All the statements between the braces will be executed with each iteration of the loop.

The parameters for a `for` loop may sound a bit confusing, but once you're used to it, you'll use `for` loops frequently. Here is a simple example of this type of loop:

```
for (i=0; i<10; i++) {
 document.write("This is line " + i + "
");
}
```

These statements define a loop that uses the variable `i`, initializes it with the value of zero, and loops as long as the value of `i` is less than 10. The increment expression, `i++`, adds one to the value of `i` with each iteration of the loop. Because this happens at the end of the loop, the output will list the numbers zero through nine.

When a loop includes only a single statement between the braces, as in this example, you can omit the braces if you want. The following statement defines the same loop without braces:

```
for (i=0; i<10; i++)
 document.write("This is line " + i + "
");
```

**Did you  
Know?**

It's a good style convention to use braces with all loops whether they contain one statement or many. This makes it easy to add statements to the loop later without causing syntax errors.

The loop in this example contains a `document.write` statement that will be repeatedly executed. To see just what this loop does, you can add it to a `<script>` section of an HTML document as shown in Listing 7.3.

**LISTING 7.3** A Loop Using the `for` Keyword

```
<html>
<head>
<title>Using a for Loop</title>
</head>
<body>
<h1>"for" Loop Example</h1>
<p>The following is the output of the
for loop:</p>
<script language="JavaScript" type="text/javascript">
for (i=1;i<10;i++) {
 document.write("This is line " + i + "
");
}
</script>
</body>
</html>
```

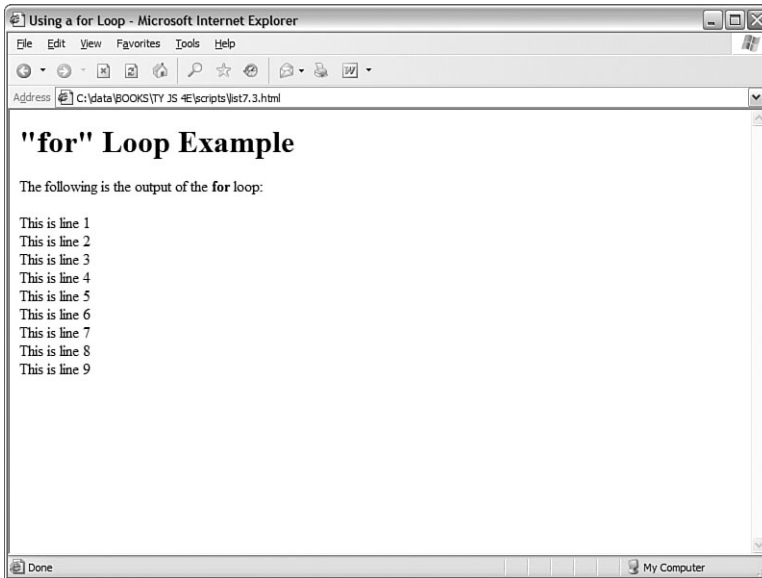
This example displays a message with the loop's counter during each iteration. The output of Listing 7.3 is shown in Figure 7.2.

Notice that the loop was only executed nine times. This is because the conditional is `i<10`. When the counter (`i`) is incremented to 10, the expression is no longer true. If you need the loop to count to 10, you can change the conditional; either `i<=10` or `i<11` will work fine.

**By the  
Way**

You might notice that the variable name `i` is often used as the counter in loops. This is a programming tradition that began with an ancient language called Forth. There's no need for you to follow this tradition, but it is a good idea to use one consistent variable for counters. (To learn more about Forth, see the Forth Interest Group's website at [www.forth.org](http://www.forth.org).)

The structure of the `for` loop in JavaScript is based on Java, which in turn is based on C. Although it is traditionally used to count from one number to another, you can use just about any statement for the initialization, condition, and increment. However, there's usually a better way to do other types of loops with the `while` keyword, described in the next section.



**FIGURE 7.2**  
The results of  
the for loop  
example.

## Using while Loops

Another keyword for loops in JavaScript is `while`. Unlike `for` loops, `while` loops don't necessarily use a variable to count. Instead, they execute as long as a condition is true. In fact, if the condition starts out as false, the statements won't execute at all.

The `while` statement includes the condition in parentheses, and it is followed by a block of statements within braces, just like a `for` loop. Here is a simple `while` loop:

```
while (total < 10) {
 n++;
 total += values[n];
}
```

This loop uses a counter, `n`, to iterate through the `values` array. Rather than stopping at a certain count, however, it stops when the total of the values reaches 10.

You might have noticed that you could have done the same thing with a `for` loop:

```
for (n=0; total < 10; n++) {
 total += values[n];
}
```

As a matter of fact, the `for` loop is nothing more than a special kind of `while` loop that handles an initialization and an increment for you. You can generally use `while` for any loop. However, it's best to choose whichever type of loop makes the most sense for the job, or that takes the least amount of typing.

## Using do...while Loops

JavaScript 1.2 introduced a third type of loop: the do...while loop. This type of loop is similar to an ordinary while loop, with one difference: The condition is tested at the *end* of the loop rather than the beginning. Here is a typical do...while loop:

```
do {
 n++;
 total += values[n];
}
while (total < 10);
```

As you've probably noticed, this is basically an upside-down version of the previous while example. There is one difference: With the do loop, the condition is tested at the end of the loop. This means that the statements in the loop will always be executed at least once, even if the condition is never true.

### By the Way

As with the for and while loops, the do loop can include a single statement without braces, or a number of statements enclosed in braces.

## Working with Loops

Although you can use simple for and while loops for straightforward tasks, there are some considerations you should make when using more complicated loops. In the next sections, we'll look at infinite loops and the break and continue statements, which give you more control over your loops.

## Creating an Infinite Loop

The for and while loops give you quite a bit of control over the loop. In some cases, this can cause problems if you're not careful. For example, look at the following loop code:

```
while (i < 10) {
 n++;
 values[n] = 0;
}
```

There's a mistake in this example. The condition of the while loop refers to the *i* variable, but that variable doesn't actually change during the loop. This creates an *infinite loop*. The loop will continue executing until the user stops it, or until it generates an error of some kind.

Infinite loops can't always be stopped by the user, except by quitting the browser—and some loops can even prevent the browser from quitting, or cause a crash.



Obviously, infinite loops are something to avoid. They can also be difficult to spot because JavaScript won't give you an error that actually tells you there is an infinite loop. Thus, each time you create a loop in a script, you should be careful to make sure there's a way out.

Depending on the browser version in use, an infinite loop might even make the browser stop responding to the user. Be sure you provide an escape route from infinite loops, and save your script before you test it just in case.

***By the  
Way***

Occasionally, you might want to create an infinite loop deliberately. This might include situations when you want your program to execute until the user stops it, or if you are providing an escape route with the `break` statement, which is introduced in the next section. Here's an easy way to create an infinite loop:

```
while (true) {
```

Because the value `true` is the conditional, this loop will always find its condition to be true.

## Escaping from a Loop

There is one way out of an infinite loop. You can use the `break` statement during a loop to exit it immediately and continue with the first statement after the loop. Here is a simple example of the use of `break`:

```
while (true) {
 n++;
 if (values[n] == 1) break;
}
```

Although the `while` statement is set up as an infinite loop, the `if` statement checks the corresponding value of an array. If it finds a value of 1, it exits the loop.

When the JavaScript interpreter encounters a `break` statement, it skips the rest of the loop and continues the script with the first statement after the right brace at the loop's end. You can use the `break` statement in any type of loop, whether infinite or not. This provides an easy way to exit if an error occurs, or if another condition is met.

## Continuing a Loop

One more statement is available to help you control the execution of statements in a loop. The `continue` statement skips the rest of the loop but, unlike `break`, it continues with the next iteration of the loop. Here is a simple example:

```
for (i=1; i<21; i++) {
 if (score[i]==0) continue;
 document.write("Student number ",i, " Score: ", score[i], "\n");
}
```

This script uses a for loop to print out scores for 20 students, stored in the score array. The if statement is used to check for scores with a value of 0. The script assumes that a score of 0 means that the student didn't take the test, so it continues the loop without printing that score.

## Looping Through Object Properties

A third type of loop is available in JavaScript. The for...in loop is not as flexible as an ordinary for or while loop. Instead, it is specifically designed to perform an operation on each property of an object.

For example, the navigator object contains properties that describe the user's browser, as you'll learn in Hour 15, "Unobtrusive Scripting." You can use for...in to display this object's properties:

```
for (i in navigator) {
 document.write("property: " + i);
 document.write(" value: " + navigator[i] + "
");
}
```

Like an ordinary for loop, this type of loop uses an index variable (i in the example). For each iteration of the loop, the variable is set to the next property of the object. This makes it easy when you need to check or modify each of an object's properties.



### Try It Yourself

#### Working with Arrays and Loops

To apply your knowledge of loops, you will now create a script that deals with arrays using loops. As you progress through this script, try to imagine how difficult it would be without JavaScript's looping features.

This simple script will prompt the user for a series of names. After all of the names have been entered, it will display the list of names in a numbered list. To begin the script, initialize some variables:

```
names = new Array();
i = 0;
```

The names array will store the names the user enters. You don't know how many names will be entered, so you don't need to specify a dimension for the array. The `i` variable will be used as a counter in the loops.

Next, use the prompt statement to prompt the user for a series of names. Use a loop to repeat the prompt for each name. You want the user to enter at least one name, so a do loop is ideal:

```
do {
 next = prompt("Enter the Next Name", "");
 if (next > " ") names[i] = next;
 i = i + 1;
}
while (next > " ");
```

If you're interested in making your scripts as short as possible, remember that you could use the increment (`++`) operator to combine the `i = i + 1` statement with the previous statement: `names[i++] = next`.

***Did you  
Know?***

This loop prompts for a string called `next`. If a name was entered and isn't blank, it's stored as the next entry in the names array. The `i` counter is then incremented. The loop repeats until the user doesn't enter a name or clicks Cancel in the prompt dialog.

Next, your script can display the number of names that was entered:

```
document.write("<h2>" + (names.length) + " names entered.</h2>");
```

This statement displays the `length` property of the names array, surrounded by level 2 heading tags for emphasis.

Next, the script should display all the names in the order they were entered. Because the names are in an array, the `for...in` loop is a good choice:

```
document.write("");
for (i in names) {
 document.write("" + names[i] + "
");
}
document.write("");
```

Here you have a `for...in` loop that loops through the names array, assigning the counter `i` to each index in turn. The script then prints the name with a `<li>` tag as an item in an ordered list. Before and after the loop, the script prints beginning and ending `<ol>` tags.

You now have everything you need for a working script. Listing 7.4 shows the HTML file for this example, and Listing 7.5 shows the JavaScript file.

**LISTING 7.4 A Script to Prompt for Names and Display Them (HTML)**

---

```
<html>
<head>
<title>Loops Example</title>
</head>
<body>
<h1>Loop Example</h1>
<p>Enter a series of names. I will then
display them in a nifty numbered list.</p>
<script language="JavaScript" type="text/javascript"
src="loops.js">
</script>
</body>
</html>
```

---

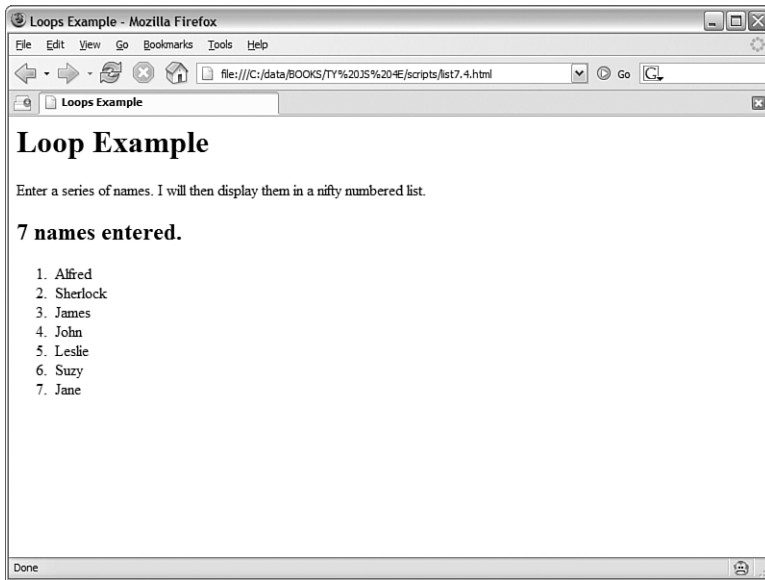
**LISTING 7.5 A Script to Prompt for Names and Display Them (JavaScript)**

---

```
// create the array
names = new Array();
i = 0;
// loop and prompt for names
do {
 next = window.prompt("Enter the Next Name", "");
 if (next > " ") names[i] = next;
 i = i + 1;
} while (next > " ");
document.write("<h2>" + (names.length) + " names entered.</h2>");
// display all of the names
document.write("");
for (i in names) {
 document.write("" + names[i] + "
");
}
document.write("");
```

---

To try this example, save the JavaScript file as `loops.js` and then load the HTML document into a browser. You'll be prompted for one name at a time. Enter several names, and then click Cancel to indicate that you're finished. Figure 7.3 shows what the final results should look like in a browser.



**FIGURE 7.3**  
The output of  
the names  
example, as  
shown by  
Firefox.

---

## Summary

In this hour, you've learned two ways to control the flow of your scripts. First, you learned how to use the `if` statement to evaluate conditional expressions and react to them. You also learned a shorthand form of conditional expression using the `?` operator, and the `switch` statement for working with multiple conditions.

You also learned about JavaScript's looping capabilities using `for`, `while`, and other loops, and how to control loops further using the `break` and `continue` statements. Lastly, you looked at the `for...in` loop for working with each property of an object.

In the next hour, you'll look at JavaScript's built-in functions, another essential tool for creating your own scripts. You'll also learn about third-party libraries that enable you to create complex effects with simple scripts.

## Q&A

- Q.** *What happens if I compare two items of different data types (for example, a number and a string) in a conditional expression?*
- A.** The JavaScript interpreter does its best to make the values a common format and compare them. In this case, it would convert them both to strings before comparing. In JavaScript 1.3 and later, you can use the special equality operator `===` to compare two values and their types—using this operator, the expression will be true only if the expressions have the same value *and* the same data type.
- Q.** *Why would I use `switch` if using `if` and `else` is just as simple?*
- A.** Either one works, so it's your choice. Personally, I find `switch` statements confusing and prefer to use `if`. Your choice might also depend on what other programming languages you're familiar with because some support `switch` and others don't.
- Q.** *Why don't I get a friendly error message if I accidentally use `=` instead of `==`?*
- A.** In some cases, this will result in an error. However, the incorrect version often appears to be a correct statement. For example, in the statement `if (a=1)`, the variable `a` will be assigned the value 1. The `if` statement is considered true, and the value of `a` is lost.
- Q.** *It seems like I could use a `for` loop to replace any of the other loop methods (`while`, `do`, and `so on`). Why so many choices?*
- A.** You're right. In most cases, a `for` loop will work, and you can do all your loops that way if you want. For that matter, you can use `while` to replace a `for` loop. You can use whichever looping method makes the most sense for your application.

## Quiz Questions

Test your knowledge of JavaScript conditions and loops by answering the following questions.

1. Which of the following operators means “is not equal to” in JavaScript?
  - a. `!`
  - b. `!=`
  - c. `<>`

2. What does the switch statement do?
  - a. Tests a variable for a number of different values
  - b. Turns a variable on or off
  - c. Makes ordinary if statements longer and more confusing
3. Which type of JavaScript loop checks the condition at the *end* of the loop?
  - a. for
  - b. while
  - c. do...while
4. Within a loop, what does the break statement do?
  - a. Crashes the browser
  - b. Starts the loop over
  - c. Escapes the loop entirely
5. The statement while (3==3) is an example of
  - a. A typographical error
  - b. An infinite loop
  - c. An illegal JavaScript statement

## Quiz Answers

1. b. The != operator means *is not equal to*.
2. a. The switch statement can test the same variable or expression for a number of different values.
3. c. The do...while loop uses a condition at the end of the loop.
4. c. The break statement escapes the loop.
5. b. Because the condition (3==3) will always be true, this statement creates an infinite loop.

## Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Modify Listing 7.4 to sort the names in alphabetical order before displaying them. You can use the `sort` method of the `Array` object, described in Hour 5, “Using Variables, Strings, and Arrays.”
- ▶ Modify Listing 7.4 to prompt for exactly 10 names. What happens if you click the Cancel button instead of entering a name?



## Hour 8

# Using Built-in Functions and Libraries

---

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

- ▶ Using the Math object's methods
- ▶ Using the Date object to work with dates
- ▶ Creating an application using JavaScript math functions
- ▶ Using with to work with objects
- ▶ How third-party libraries make scripting easier
- ▶ Using third-party libraries in your scripts

You've nearly reached the end of Part II! In this hour, you'll learn the basics of objects in JavaScript and the details of using the Math and Date objects. You'll also look at some third-party libraries, which enable you to achieve amazing JavaScript effects with a few lines of code.

## **Using the Math Object**

The Math object is a built-in JavaScript object that includes math constants and functions. You don't need to create a Math object; it exists automatically in any JavaScript program. The Math object's properties represent mathematical constants, and its methods are mathematical functions.

### **Rounding and Truncating**

Three of the most useful methods of the Math object enable you to round decimal values up and down:

- ▶ `Math.ceil()` rounds a number up to the next integer.
- ▶ `Math.floor()` rounds a number down to the next integer.
- ▶ `Math.round()` rounds a number to the nearest integer.

All of these take the number to be rounded as their single parameter. You might notice one thing missing: the capability to round to a decimal place, such as for dollar amounts. Fortunately, you can easily simulate this. Here is a simple function that rounds numbers to two decimal places:

```
function round(num) {
 return Math.round(num * 100) / 100;
}
```

This function multiplies the value by 100 to move the decimal, and then rounds the number to the nearest integer. Finally, the value is divided by 100 to restore the decimal to its original position.

## Generating Random Numbers

One of the most commonly used methods of the `Math` object is the `Math.random()` method, which generates a random number. This method doesn't require any parameters. The number it returns is a random decimal number between zero and one.

You'll usually want a random number between one and a value. You can do this with a general-purpose random number function. The following is a function that generates random numbers between one and the parameter you send it:

```
function rand(num) {
 return Math.floor(Math.random() * num) + 1;
}
```

This function multiplies a random number by the value specified in the `num` parameter, and then converts it to an integer between one and the number by using the `Math.floor()` method.

## Other Math Functions

The `Math` object includes many functions beyond those you've looked at here. For example, `Math.sin()` and `Math.cos()` calculate sines and cosines. The `Math` object also includes properties for various mathematical constants, such as `Math.PI`. See Appendix D, "JavaScript Quick Reference," for a complete list of math functions and constants.

## Working with Math Functions

The `Math.random()` method generates a random number between 0 and 1. However, it's very difficult for a computer to generate a truly random number. (It's also hard for a human being to do so—that's why dice were invented.)

Today's computers do reasonably well at generating random numbers, but just how good is JavaScript's `Math.random` function? One way to test it is to generate many random numbers and calculate the average of all of them.

In theory, the average should be somewhere near .5, halfway between 0 and 1. The more random values you generate, the closer the average should get to this middle ground.

As an example of the use of the `Math` object's methods, you can create a script that tests JavaScript's random number function. To do this, you'll generate 5,000 random numbers and calculate their average.

Rather than typing it in, you can download and try this hour's example at this book's website.

***Did you  
Know?***

In case you skipped Hour 7, "Controlling Flow with Conditions and Loops," and are getting out your calculator, don't worry—you'll use a loop to generate the random numbers. You'll be surprised how fast JavaScript can do this.

To begin your script, you will initialize a variable called `total`. This variable will store a running total of all of the random values, so it's important that it starts at 0:

```
total = 0;
```

Next, begin a loop that will execute 5,000 times. Use a `for` loop because you want it to execute a fixed number of times:

```
for (i=1; i<=5000; i++) {
```

Within the loop, you will need to create a random number and add its value to `total`. Here are the statements that do this and continue with the next iteration of the loop:

```
 num = Math.random();
 total += num;
}
```

Depending on the speed of your computer, it might take a few seconds to generate those 5,000 random numbers. Just to be sure something is happening, the script will display a status message after each 1,000 numbers:

```
if (i % 1000 == 0)
 document.write("Generated " + i + " numbers...
");
```

### By the Way

The % symbol in the previous code is the *modulo operator*, which gives you the remainder after dividing one number by another. Here it is used to find even multiples of 1,000.

The final part of your script will calculate the average by dividing `total` by 5,000. Your script can also round the average to three decimal places, using the trick you learned earlier in this hour:

```
average = total / 5000;
average = Math.round(average * 1000) / 1000;
document.write("<H2>Average of 5000 numbers: " + average + "</H2>");
```

To test this script and see just how random those numbers are, combine the complete script with an HTML document and `<script>` tags. Listing 8.1 shows the complete random number testing script.

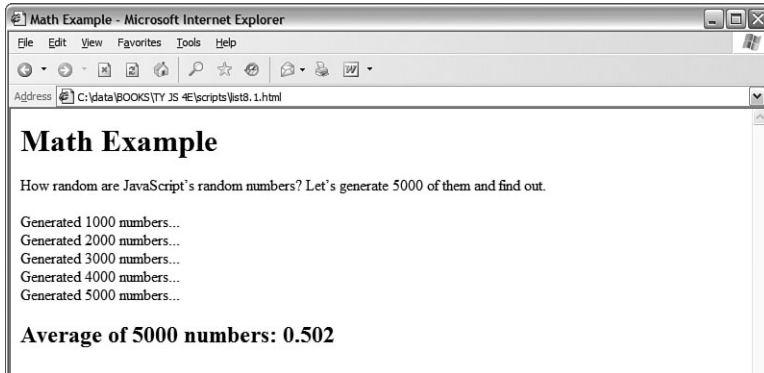
#### LISTING 8.1 A Script to Test JavaScript's Random Number Function

```
<html>
<head>
<title>Math Example</title>
</head>
<body>
<h1>Math Example</h1>
<p>How random are JavaScript's random numbers?
Let's generate 5000 of them and find out.</p>
<script language="JavaScript" type="text/javascript">
total = 0;
for (i=1; i<=5000; i++) {
 num = Math.random();
 total += num;
 if (i % 1000 == 0)
 document.write("Generated " + i + " numbers...
");
}
average = total / 5000;
average = Math.round(average * 1000) / 1000;
document.write("<H2>Average of 5000 numbers: " + average + "</H2>");
</script>
</body>
</html>
```

To test the script, load the HTML document into a browser. After a short delay, you should see a result. If it's close to .5, the numbers are reasonably random. My result was .502, as shown in Figure 8.1.

The average you've used here is called an *arithmetic mean*. This type of average isn't a perfect way to test randomness. Actually, all it tests is the distribution of the numbers above and below .5. For example, if the numbers turned out to be 2,500 .4s and 2,500 .6s, the average would be a perfect .5—but they wouldn't be very random numbers. (Thankfully, JavaScript's random numbers don't have this problem.)

**By the  
Way**



**FIGURE 8.1**  
The random  
number testing  
script in action.

## Using the with Keyword

The with keyword is one you haven't seen before. You can use it to make JavaScript programming easier—or at least easier to type.

The with keyword specifies an object, and it is followed by a block of statements enclosed in braces. For each statement in the block, any properties you mention without specifying an object are assumed to be for that object.

As an example, suppose you have a string called lastname. You can use with to perform string operations on it without specifying the name of the string every time:

```
with (lastname) {
 window.alert("length of last name: " + length);
 capname = toUpperCase();
}
```

In this example, the length property and the toUpperCase method refer to the lastname string, although it is only specified once with the with keyword.

Obviously, the with keyword only saves a bit of typing in situations like this. However, you might find it more useful when you're dealing with a DOM object throughout a large procedure, or when you are using a built-in object, such as the Math object, repeatedly.

## Working with Dates

The `Date` object is a built-in JavaScript object that enables you to conveniently work with dates and times. You can create a `Date` object anytime you need to store a date, and use the `Date` object's methods to work with the date.

You encountered one example of a `Date` object in Hour 2, "Creating Simple Scripts," with the time/date script. The `Date` object has no properties. To set or obtain values from a `Date` object, you must use the methods described in the next section.

### By the Way

JavaScript dates are stored as the number of milliseconds since midnight, January 1, 1970. This date is called the *epoch*. Dates before 1970 weren't allowed in early versions, but are now represented by negative numbers.

## Creating a Date Object

You can create a `Date` object using the `new` keyword. You can also optionally specify the date to store in the object when you create it. You can use any of the following formats:

```
birthday = new Date();
birthday = new Date("June 20, 2003 08:00:00");
birthday = new Date(6, 20, 2003);
birthday = new Date(6, 20, 2003, 8, 0, 0);
```

You can choose any of these formats, depending on which values you wish to set. If you use no parameters, as in the first example, the current date is stored in the object. You can then set the values using the `set` methods, described in the next section.

## Setting Date Values

A variety of `set` methods enable you to set components of a `Date` object to values:

- ▶ `setDate()` sets the day of the month.
- ▶ `setMonth()` sets the month. JavaScript numbers the months from 0 to 11, starting with January (0).
- ▶ `setFullYear()` sets the year.
- ▶ `setTime()` sets the time (and the date) by specifying the number of milliseconds since January 1, 1970.
- ▶ `setHours()`, `setMinutes()`, and `setSeconds()` set the time.

As an example, the following statement sets the year of a `Date` object called `holiday` to 2003:

```
holiday.setFullYear(2003);
```

## Reading Date Values

You can use the `get` methods to get values from a `Date` object. This is the only way to obtain these values, because they are not available as properties. Here are the available `get` methods for dates:

- ▶ `getDate()` gets the day of the month.
- ▶ `getMonth()` gets the month.
- ▶ `getFullYear()` gets the year.
- ▶ `getTime()` gets the time (and the date) as the number of milliseconds since January 1, 1970.
- ▶ `getHours()`, `getMinutes()`, `getSeconds()`, and `getMilliseconds()` get the components of the time.

Along with `setFullYear` and `getFullYear`, which require four-digit years, JavaScript includes `setYear` and `getYear` methods, which use two-digit year values. You should always use the four-digit version to avoid Year 2000 issues.

***By the  
Way***

## Working with Time Zones

Finally, a few functions are available to help your `Date` objects work with local time values and time zones:

- ▶ `getTimezoneOffset()` gives you the local time zone's offset from UTC (Coordinated Universal Time, based on the old Greenwich Mean Time standard). In this case, *local* refers to the location of the browser. (Of course, this only works if the user has set his or her system clock accurately.)
- ▶ `toUTCString()` converts the date object's time value to text, using UTC. This method was introduced in JavaScript 1.2 to replace the `toGMTString` method, which still works but should be avoided.
- ▶ `toLocaleString()` converts the date object's time value to text, using local time.

Along with these basic functions, JavaScript 1.2 and later include UTC versions of several of the functions described previously. These are identical to the regular commands, but work with UTC instead of local time:

- ▶ `getUTCDate()` gets the day of the month in UTC time.
- ▶ `getUTCDay()` gets the day of the week in UTC time.
- ▶ `getUTCFullYear()` gets the four-digit year in UTC time.
- ▶ `getUTCMonth()` returns the month of the year in UTC time.
- ▶ `getUTCHours()`, `getUTCMinutes()`, `getUTCSeconds()`, and `getUTCMilliseconds()` return the components of the time in UTC.
- ▶ `setUTCDate()`, `setUTCFullYear()`, `setUTCMonth()`, `setUTCHours()`, `setUTCMinutes()`, `setUTCSeconds()`, and `setUTCMilliseconds()` set the time in UTC.

## Converting Between Date Formats

Two special methods of the `Date` object allow you to convert between date formats. Instead of using these methods with a `Date` object you created, you use them with the built-in object `Date` itself. These include the following:

- ▶ `Date.parse()` converts a date string, such as `June 20, 1996`, to a `Date` object (number of milliseconds since 1/1/1970).
- ▶ `Date.UTC()` does the opposite. It converts a `Date` object value (number of milliseconds) to a UTC (GMT) time.

## Using Third-Party Libraries

When you use JavaScript's built-in `Math` and `Date` functions, JavaScript does most of the work—you don't have to figure out how to convert dates between formats or calculate a cosine. Third-party libraries are not included with JavaScript, but they serve a similar purpose—enabling you to do complicated things with only a small amount of code.

Using one of these libraries is usually as simple as copying one or more files to your site and including a `<script>` tag in your document to load the library. Several popular JavaScript libraries are discussed in the following sections.



JavaScript libraries are a relatively new phenomenon, and new libraries are appearing regularly. See this book's website for an updated list of libraries.

***Did you  
Know?***

## Prototype

Prototype, created by Sam Stephenson, is a JavaScript library that simplifies tasks such as working with DOM objects, dealing with data in forms, and remote scripting (AJAX). By including a single `prototype.js` file in your document, you have access to many improvements to basic JavaScript.

For example, you've used the `document.getElementById` method to obtain the DOM object for an element within a web page. Prototype includes an improved version of this in the `$( )` function. Not only is it easier to type, but it is also more sophisticated than the built-in function and supports multiple objects.

Adding Prototype to your pages requires only one file, `prototype.js`, and one `<script>` tag:

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

Prototype is free, open-source software. You can download it from its official website at <http://prototype.conio.net>. Prototype is also built into the Ruby on Rails framework for the server-side language Ruby—see <http://www.rubyonrails.com/> for more information.

***By the  
Way***

## Script.aculo.us

By the end of this book, you'll learn to do some impressive things with JavaScript—for example, animating an object within a page. The code for a task like this is complex, but you can also include effects in your pages using a prebuilt library. This enables you to use impressive effects with only a few lines of code.

Script.aculo.us by Thomas Fuchs is one such library. It includes functions to simplify drag-and-drop tasks, such as rearranging lists of items. It also includes a number of Combination Effects, which enable you to use highlighting and animated transitions within your pages. For example, a new section of the page can be briefly highlighted in yellow to get the user's attention, or a portion of the page can fade out or slide off the screen.

After you've included the appropriate files, using effects is as easy as using any of JavaScript's built-in methods. For example, the following statements use Script.aculo.us to fade out an element of the page with the `id` value `test`:

```
obj = document.getElementById("test");
new Effect.Fade(obj);
```

Script.aculo.us is built on the Prototype framework described in the previous section, and includes all of the functions of Prototype, so you could also simplify this further by using the `$` function:

```
new Effect.Fade$("#test");
```

### ***Did you Know?***

You will create a script that demonstrates several Script.aculo.us effects in the Try It Yourself section later this hour.

## **AJAX Frameworks**

AJAX (Asynchronous JavaScript and XML), also known as *remote scripting*, enables JavaScript to communicate with a program running on the web server. This enables JavaScript to do things that were traditionally not possible, such as dynamically loading information from a database or storing data on a server without refreshing a page.

Unfortunately, AJAX requires some complex scripting, particularly because the methods you use to communicate with the server vary depending on the browser in use. Fortunately, many libraries have been created to fill the need for a simple way to use AJAX.

The Prototype library, described previously, includes AJAX features. There are also many dedicated AJAX libraries. One of the most popular is SAJAX (Simple AJAX), an open-source toolkit that makes it easy to use AJAX to communicate with PHP, Perl, and other languages from JavaScript. Visit the SAJAX website for details at <http://www.modernmethod.com/sajax>.

### ***By the Way***

See Hour 17, “AJAX: Remote Scripting,” for examples of remote scripting, with and without using third-party libraries.

## **Other Libraries**

There are many more JavaScript libraries out there, and more are appearing all of the time as JavaScript is taken more seriously as an application language. Here are some more libraries you might want to explore:

- Dojo (<http://www.dojotoolkit.org/>) is an open-source toolkit that adds power to JavaScript to simplify building applications and user interfaces. It adds features ranging from extra string and math functions to animation and AJAX.

- ▶ The Yahoo! UI Library (<http://developer.yahoo.net/yui/>) was developed by Yahoo! and made available to everyone under an open-source license. It includes features for animation, DOM features, event management, and easy-to-use user interface elements such as calendars and sliders.
- ▶ MochiKit (<http://mochikit.com/>) is a lightweight library that adds features for working with the DOM, CSS colors, string formatting, and AJAX. It also supports a nice logging mechanism for debugging your scripts.

## Try It Yourself



### Adding Effects with a Library

To see how simple it is to use an external library, you will now create an example script that includes the Script.aculo.us library and use event handlers to demonstrate several of the available effects.

This example was created using version 1.5.1 of the Script.aculo.us library. It should work with later versions, but the library might have changed since this was written. If you have trouble, you might need to use this specific version.

**Watch  
Out!**

## Downloading the Library

To use the library, you will need to download it and copy the files you need to the same folder where you will store your script. You can download the library from the Script.aculo.us website at <http://script.aculo.us/downloads>.

The download is available as a Zip file. Inside the Zip file you will find a folder called `scriptaculous-js-x.x.x`. You will need the following files from the folders under this folder:

- ▶ `prototype.js` (the Prototype library) from the `lib` folder
- ▶ `effects.js` (the effects functions) from the `src` folder

Copy both of these files to a folder on your computer, and be sure to create your demonstration script in the same folder.

The Script.aculo.us download includes many other files, and you can include the entire library if you intend to use all of its features. For this example, you only need the two files described here.

**By the  
Way**

## Including the Files

To add the library to your HTML document, simply use `<script>` tags to include the two JavaScript files you copied from the download:

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

If you include these statements as the first things in the `<head>` section of your document, the library functions will be available to other scripts or event handlers anywhere in the page.

## Using Effects

After you have included the library, you simply need to include a bit of JavaScript to trigger the effects. We will use a section of the page wrapped in a `<div>` tag with the `id` value `test` to demonstrate the effects. Each effect is triggered by a simple event handler on a button. For example, this code defines the Fade Out button:

```
<input type="button" value="Fade Out"
 onClick="new Effect.Fade($('test'))">
```

This uses the `$` function built into Prototype to obtain the object for the element with the `id` value `test`, and then passes it to the `Effect.Fade()` function built into Script.aculo.us.

### ***Did you Know?***

This example will demonstrate six effects: Fade, Appear, SlideUp, SlideDown, Highlight, and Shake. There are more than 16 effects in the library, plus methods for supporting Drag and Drop and other features. See <http://script.aculo.us> for details.

## Building the Script

After you have included the libraries, you can combine them with event handlers and some example text to create a complete demonstration of Script.aculo.us effects. The complete HTML document for this example is shown in Listing 8.2.

### **LISTING 8.2 The Complete Library Effects Example**

```
<html>
<head>
<title>Testing script.aculo.us effects</title>
<script type="text/javascript" src="prototype.js"> </script>
<script type="text/javascript" src="effects.js"> </script>
</head>
<body">
<h1>Testing script.aculo.us Effects</h1>
<form name="form1">
```

**LISTING 8.2 Continued**

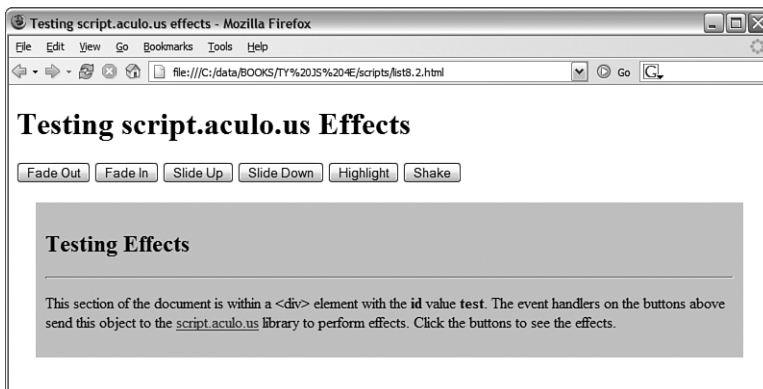
```

<input type="button" value="Fade Out"
 onClick="new Effect.Fade($('test'))">
<input type="button" value="Fade In"
 onClick="new Effect.Appear($('test'))">
<input type="button" value="Slide Up"
 onClick="new Effect.SlideUp($('test'))">
<input type="button" value="Slide Down"
 onClick="new Effect.SlideDown($('test'))">
<input type="button" value="Highlight"
 onClick="new Effect.Highlight($('test'))">
<input type="button" value="Shake"
 onClick="new Effect.Shake($('test'))">
</form>
<div id="test"
 style="background-color:#CCC; margin:20px; padding:10px;">
<h2>Testing Effects</h2>
<hr>
<p>This section of the document is within a <div> element
with the id value test. The event handlers on the
buttons above send this object to the
script.aculo.us library
to perform effects. Click the buttons to see the effects.
</p>
</div>
</body>
</html>

```

This document starts with two `<script>` tags to include the library's files. The effects are triggered by the event handlers defined for each of the six buttons. The `<div>` section at the end defines the test element that will be used to demonstrate the effects.

To try this example, make sure the `prototype.js` and `effects.js` files from Script.aculo.us are stored in the same folder as your script, and then load the HTML file into a browser. The display should look like Figure 8.2, and you can use the six buttons at the top of the page to trigger effects.



**FIGURE 8.2**  
The library effects example as displayed by Firefox.

## Summary

In this hour, you learned some specifics about the `Math` and `Date` objects built into JavaScript, and learned more than you ever wanted to know about random numbers. You also learned how third-party libraries can simplify your scripting, and you used a library to create special effects in a web page.

You've reached the end of Part II, which covered some basic building blocks of JavaScript programs. In Part III, you'll learn more about the Document Object Model, which contains objects that refer to various parts of the browser window and HTML document. This begins in Hour 9, "Responding to Events."

## Q&A

- Q.** *The random numbers are generated so quickly I can't be sure it's happening at all. Is there a way to slow this process down?*
- A.** Yes. If you add one or more form fields to the example and use them to display the data as it is generated, you'll see a much slower result. It will still be done within a couple of seconds on a fast computer, though.
- Q.** *Can I use more than one third-party library in the same script?*
- A.** Yes, in theory: If the libraries are well written and designed not to interfere with each other, there should be no problem combining them. In practice, this will depend on the libraries you need and how they were written.
- Q.** *Can I build my own library to simplify scripting?*
- A.** Yes, as you deal with more complicated scripts, you'll find yourself using the same functions over and over. You can combine them into a library for your own use. This is as simple as creating a `.js` file.

## Quiz Questions

Test your knowledge of JavaScript libraries and built-in functions by answering the following questions.

1. Which of the following objects *cannot* be used with the `new` keyword?
  - a. `Date`
  - b. `Math`
  - c. `String`

2. How does JavaScript store dates in a `Date` object?
  - a. The number of milliseconds since January 1, 1970
  - b. The number of days since January 1, 1900
  - c. The number of seconds since Netscape's public stock offering
3. What is the range of random numbers generated by the `Math.random` function?
  - a. Between 1 and 100
  - b. Between 1 and the number of milliseconds since January 1, 1970
  - c. Between 0 and 1

## Quiz Answers

1. b. The `Math` object is static; you can't create a `Math` object.
2. a. Dates are stored as the number of milliseconds since January 1, 1970.
3. c. JavaScript's random numbers are between 0 and 1.

## Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Modify the random number script in Listing 8.1 to run three times, calculating a total of 15,000 random numbers, and display separate totals for each set of 5,000. (You'll need to use another `for` loop that encloses most of the script.)
- ▶ Visit the Script.aculo.us page at <http://script.aculo.us/> to find the complete list of effects. Modify Listing 8.2 to add buttons for one or more additional effects.

*This page intentionally left blank*



## **PART III:**

# **Learning More About the DOM**

<b>HOURL 9</b>	<b>Responding to Events</b>	<b>139</b>
<b>HOURL 10</b>	<b>Using Windows and Frames</b>	<b>157</b>
<b>HOURL 11</b>	<b>Getting Data with Forms</b>	<b>173</b>
<b>HOURL 12</b>	<b>Working with Style Sheets</b>	<b>191</b>
<b>HOURL 13</b>	<b>Using the W3C DOM</b>	<b>207</b>
<b>HOURL 14</b>	<b>Using Advanced DOM Features</b>	<b>219</b>

*This page intentionally left blank*

## HOURL 9

# Responding to Events

---

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

- ▶ How event handlers work
- ▶ How event handlers relate to objects
- ▶ Creating an event handler
- ▶ Testing an event handler
- ▶ Detecting mouse actions
- ▶ Detecting keyboard actions
- ▶ Intercepting events with a special handler
- ▶ Adding friendly link descriptions to a web page

In your experience with JavaScript so far, most of the scripts you've written have executed in a calm, orderly fashion, moving from the first statement to the last.

In this hour, you'll learn to use the wide variety of event handlers supported by JavaScript. Rather than executing in order, scripts using event handlers can interact directly with the user. You'll use event handlers in just about every script you write in the rest of this book.

## **Understanding Event Handlers**

As you learned in Hour 3, "Getting Started with JavaScript Programming," JavaScript programs don't have to execute in order. You also learned they can detect *events* and react to them. Events are things that happen to the browser—the user clicking a button, the mouse pointer moving, or a web page or image loading from the server.

A wide variety of events enable your scripts to respond to the mouse, the keyboard, and other circumstances. Events are the key method JavaScript uses to make web documents interactive.

The script that you use to detect and respond to an event is called an *event handler*. Event handlers are among the most powerful features of JavaScript. Luckily, they're also among the easiest features to learn and use—often, a useful event handler requires only a single statement.

## Objects and Events

As you learned in Hour 4, “Working with the Document Object Model (DOM),” JavaScript uses a set of objects to store information about the various parts of a web page—buttons, links, images, windows, and so on. An event can often happen in more than one place (for example, the user could click any one of the links on the page), so each event is associated with an object.

Each event has a name. For example, the `onMouseOver` event occurs when the mouse pointer moves over an object on the page. When the pointer moves over a particular link, the `onMouseOver` event is sent to that link's event handler, if it has one.

### **By the Way**

Notice the strange capitalization on the `onMouseOver` keyword. This is the standard notation for event handlers. The `on` is always lowercase, and each word in the event name is capitalized.

## Creating an Event Handler

You don't need the `<script>` tag to define an event handler. Instead, you can add an event handler attribute to an individual HTML tag. For example, here is a link that includes an `onMouseOver` event handler:

```
<a href="http://www.jsworkshop.com/"
 onMouseOver="window.alert('You moved over the link.');">
Click here
```

Note that this is all one `<a>` tag, although it's split into multiple lines. This specifies a statement to be used as the `onMouseOver` event handler for the link. This statement displays an alert message when the mouse moves over the link.

### **By the Way**

The previous example uses single quotation marks to surround the text. This is necessary in an event handler because double quotation marks are used to surround the event handler itself. (You can also use single quotation marks to surround the event handler and double quotes within the script statements.)

You can use JavaScript statements like the previous one in an event handler, but if you need more than one statement, it's a good idea to use a function instead. Just define the function in the header of the document, and then call the function as the event handler like this:

```
Move the mouse over this link.
```

This example calls a function called `DoIt()` when the user moves the mouse over the link. Using a function is convenient because you can use longer, more readable JavaScript routines as event handlers. You'll use a longer function to handle events in the "Try It Yourself: Adding Link Descriptions to a Web Page" section of this hour.

For simple event handlers, you can use two statements if you separate them with a semicolon. However, in most cases it's easier to use a function to perform the statements.

***Did you  
Know?***

## Defining Event Handlers with JavaScript

Rather than specifying an event handler in an HTML document, you can use JavaScript to assign a function as an event handler. This allows you to set event handlers conditionally, turn them on and off, and change the function that handles an event dynamically.

Setting up event handlers this way is also a good practice in general: It allows you to use an external JavaScript file to define the function and set up the event, keeping the JavaScript code completely separate from the HTML file.

***Did you  
Know?***

To define an event handler in this way, you first define a function, and then assign the function as an event handler. Event handlers are stored as properties of the document object or another object that can receive an event. For example, these statements define a function called `mousealert()`, and then assign it as the `onMouseDown` event handler for the document:

```
function mousealert() {
 alert ("You clicked the mouse!");
}
document.onmousedown = mousealert;
```

You can use this technique to set up an event handler for any HTML element, but an additional step is required: You must first find the object corresponding to the element. To do this, use the `document.getElementById()` function. First, define an element in the HTML document and specify an `id` attribute:

```

```

Next, in the JavaScript code, find the object and apply the event handler:

```
obj = document.getElementById("link1");
obj.onclick = MyFunction;
```

You can do this for any object as long as you've defined it with a unique `id` attribute in the HTML file. Using this technique, you can easily assign the same function to handle events for multiple objects without adding clutter to your HTML code. See the "Try It Yourself" section in this hour for an example of this technique.

## Supporting Multiple Event Handlers

What if you want more than one thing to happen when you click on an element? For example, suppose you want two functions called `update` and `display` to both execute when a button is clicked. You can't assign two functions to the `onclick` property. One solution is to define a function that calls both functions:

```
function UpdateDisplay() {
 update();
 display();
}
```

This isn't always the ideal way to do things. For example, if you're using two third-party scripts and both of them want to add an `onLoad` event to the page, there should be a way to add both. The W3C DOM standard defines a function, `addEventListener`, for this purpose. This function defines a *listener* for a particular event and object, and you can add as many listener functions as you need.

Unfortunately, `addEventListener` is not supported by Internet Explorer (as of versions 6 and 7), so you have to use a different function, `attachEvent`, in that browser. See Hour 15, "Unobtrusive Scripting," for a function that combines these two for a cross-browser event-adding script.

## Using the event Object

When an event occurs, you might need to know more about the event—for example, for a keyboard event, you need to know which key was pressed. The DOM includes an event object that provides this information.

To use the event object, you can pass it on to your event handler function. For example, this statement defines an `onKeyPress` event that passes the event object to a function:

```
<body onKeyPress="getkey(event)";">
```

You can then define your function to accept the event as a parameter:

```
function getkey(e) {
 ...
}
```

In Mozilla-based browsers (Firefox and Netscape), an event object is automatically passed to the event handler function, so this will work even if you use JavaScript rather than HTML to define an event handler. In Internet Explorer, the most recent event is stored in the `window.event` object. The previous HTML example passes this object to the event handler function. If you define the event handler with JavaScript, this is not possible, so you need to use some code to find the correct object:

```
Function getkey(e) {
 if (!e) e=window.event;
 ...
}
```

This checks whether the `e` variable is already defined. If not, it gets the `window.event` object and stores it in `e`. This ensures that you have a valid event object in any browser.

Unfortunately, while both Internet Explorer and Mozilla-based browsers support event objects, they support different properties. One property that is the same in both browsers is `event.type`, the type of event. This is simply the name of the event, such as `mouseover` for an `onMouseOver` event, and `keypress` for an `onKeyPress` event. The following sections list some additional useful properties for each browser.

## Internet Explorer event Properties

The following are some of the commonly used properties of the event object for Internet Explorer 4.0 and later:

- ▶ **event.button**—The mouse button that was pressed. This value is 1 for the left button and usually 2 for the right button.
- ▶ **event.clientX**—The x-coordinate (column, in pixels) where the event occurred.
- ▶ **event.clientY**—The y-coordinate (row, in pixels) where the event occurred.
- ▶ **event.altkey**—A flag that indicates whether the Alt key was pressed during the event.
- ▶ **event.ctrlkey**—Indicates whether the Ctrl key was pressed.
- ▶ **event.shiftkey**—Indicates whether the Shift key was pressed.

- ▶ **event.keyCode**—The key code (in Unicode) for the key that was pressed.
- ▶ **event.srcElement**—The object where the element occurred.

### By the Way

See the Try it Yourself section of this hour for an example that uses the `srcElement` property and Mozilla's `target` property for a cross-browser method of determining the object for an event.

## Netscape and Firefox event Properties

The following are some of the commonly used properties of the event object for Netscape 4.0 and later:

- ▶ **event.modifiers**—Indicates which modifier keys (Shift, Ctrl, Alt, and so on) were held down during the event. This value is an integer that combines binary values representing the different keys.
- ▶ **event.pageX**—The x-coordinate of the event within the web page.
- ▶ **event.pageY**—The y-coordinate of the event within the web page.
- ▶ **event.which**—The keycode for keyboard events (in Unicode), or the button that was pressed for mouse events (It's best to use the cross-browser `button` property instead.)
- ▶ **event.button**—The mouse button that was pressed. This works just like Internet Explorer except that the left button's value is 0 and the right button's value is 2.
- ▶ **event.target**—The object where the element occurred.

### By the Way

The `event.pageX` and `event.pageY` properties are based on the top-left corner of the element where the event occurred, not always the exact position of the mouse pointer.

## Using Mouse Events

The DOM includes a number of event handlers for detecting mouse actions. Your script can detect the movement of the mouse pointer and when a button is clicked, released, or both.



## Over and Out

You’ve already seen the first and most common event handler, `onMouseOver`. This handler is called when the mouse pointer moves over a link or other object.

The `onMouseOut` handler is the opposite—it is called when the mouse pointer moves out of the object’s border. Unless something strange happens, this always happens sometime after the `onMouseOver` event is called.

This handler is particularly useful if your script has made a change when the pointer moved over the object—for example, displaying a message in the status line or changing an image. You can use an `onMouseOut` handler to undo the action when the pointer moves away.

You’ll use both `onMouseOver` and `onMouseOut` handlers in the “Try it Yourself: Adding Link Descriptions to a Web Page” section at the end of this hour.

One of the most common uses for the `onMouseOver` and `onMouseOut` event handlers is to create *rollovers*—images that change when the mouse moves over them. You’ll learn how to create these in Hour 19, “Using Graphics and Animation.”

***Did you  
Know?***

## Using the `onMouseMove` Event

The `onMouseMove` event occurs any time the mouse pointer moves. As you might imagine, this happens quite often—the event can trigger hundreds of times as the mouse pointer moves across a page.

Because of the large number of generated events, browsers don’t support the `onMouseMove` event by default. To enable it for a page, you need to use *event capturing*. This is similar to the dynamic events technique you learned earlier in this hour, but requires an extra step for some older browsers.

The basic syntax to support this event, for both browsers, is to set a function as the `onMouseMove` handler for the document or another object. For example, this statement sets the `onMouseMove` handler for the document to a function called `MoveHere`, which must be defined in the same page:

```
document.onMouseMove=MoveHere;
```

Additionally, older versions of Netscape require that you specifically enable the event using the `document.captureEvents` method:

```
document.captureEvents(Event.MOUSEMOVE);
```

## Ups and Downs (and Clicks)

You can also use events to detect when the mouse button is clicked. The basic event handler for this is `onClick`. This event handler is called when the mouse button is clicked while positioned over the appropriate object.

### By the Way

The object in this case can be a link. It can also be a form element. You'll learn more about forms in Hour 11, "Getting Data with Forms."

For example, you can use the following event handler to display an alert when a link is clicked:

```
<a href="http://www.jsworkshop.com/"
onClick="alert('You are about to leave this site.');">Click Here
```

In this case, the `onClick` event handler runs before the linked page is loaded into the browser. This is useful for making links conditional or displaying a disclaimer before launching the linked page.

If your `onClick` event handler returns the `false` value, the link will not be followed. For example, the following is a link that displays a confirmation dialog. If you click Cancel, the link is not followed; if you click OK, the new page is loaded:

```
<a href="http://www.jsworkshop.com/"
onClick="return(window.confirm('Are you sure?'));">
Click Here
```

This example uses the `return` statement to enclose the event handler. This ensures that the `false` value that is returned when the user clicks Cancel is returned from the event handler, which prevents the link from being followed.

The `ondblclick` event handler is similar, but is only used if the user double-clicks on an object. Because links usually require only a single click, you could use this to make a link do two different things depending on the number of clicks. (Needless to say, this could be confusing.) You can also detect double-clicks on images and other objects.

To give you even more control of what happens when the mouse button is pressed, two more events are included:

- ▶ `onMouseDown` is used when the user presses the mouse button.
- ▶ `onMouseUp` is used when the user releases the mouse button.

These two events are the two halves of a mouse click. If you want to detect an entire click, use `onClick`. Use `onMouseUp` and `onMouseDown` to detect just one or the other.

To detect which mouse button is pressed, you can use the `button` property of the event object. This property is assigned the value 0 or 1 for the left button, and 2 for the right button. This property is assigned for `onClick`, `onDbClick`, `onMouseUp`, and `onMouseDown` events.

Browsers don't normally detect `onClick` or `onDbClick` events for the right mouse button. If you want to detect the right button, `onMouseDown` is the most reliable way.

**Watch  
Out!**

As an example of these event handlers, you can create a script that displays information about mouse button events and determines which button is pressed. Listing 9.1 shows the mouse event script.

---

**LISTING 9.1** The JavaScript file for the mouse click example.

```
function mousestatus(e) {
 if (!e) e = window.event;
 btn = e.button;
 whichone = (btn < 2) ? "Left" : "Right";
 message=e.type + " : " + whichone + "\n";
 document.form1.info.value += message;
}
obj=document.getElementById("testlink");
obj.onmousedown = mousestatus;
obj.onmouseup = mousestatus;
obj.onclick = mousestatus;
obj.ondbclick = mousestatus;
```

---

This script includes a function, `mousestatus()`, that detects mouse events. This function uses the `button` property of the event object to determine which button was pressed. It also uses the `type` property to display the type of event, since the function will be used to handle multiple event types.

After the function, the script finds the object for a link with the `id` attribute `testlink` and assigns its `onmousedown`, `onmouseup`, `onclick`, and `ondbclick` events to the `mousestatus` function.

Save this script as `click.js`. Next, you will need an HTML document to work with the script, shown in Listing 9.2.

---

**LISTING 9.2** The HTML file for the mouse click example.

```
<html>
<head>
<title>Mouse click test</title>
</head>
<body>
<h1>Mouse Click Test</h1>
```

**LISTING 9.2 Continued**

```

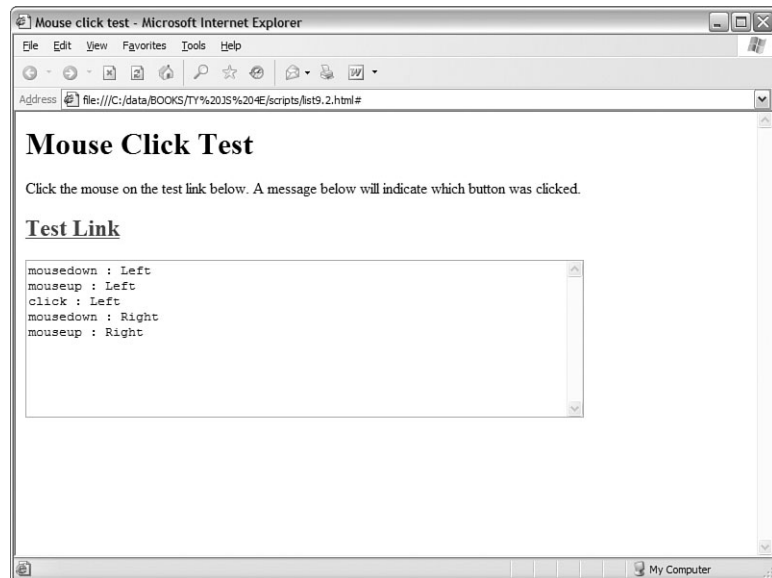
<p>Click the mouse on the test link below. A message below
will indicate which button was clicked.</p>
<h2>Test Link</h2>
<form name="form1">
<textarea rows="10" cols="70" name="info"></textarea>
</form>
<script language="javascript" type="text/javascript"
 src="click.js">
</script>
</body>
</html>

```

This file defines a test link with the id property testlink, which is used in the script to assign event handlers. It also defines a form and a textarea used by the script to display the events. To test this document, save it in the same folder as the JavaScript file you created previously and load the HTML document into a browser. The results are shown in Figure 9.1.

**FIGURE 9.1**

The mouse click example in action.



**By the  
Way**

Notice that a single click of the left mouse button triggers three events: onMouseDown, onMouseUp, and then onClick.

## Using Keyboard Events

JavaScript can also detect keyboard actions. The main event handler for this purpose is `onKeyPress`, which occurs when a key is pressed and released, or held down. As with mouse buttons, you can detect the down and up parts of the keypress with the `onKeyDown` and `onKeyUp` event handlers.

Of course, you might find it useful to know which key the user pressed. You can find this out with the event object, which is sent to your event handler when the event occurs. In Netscape and Firefox, the `event.which` property stores the ASCII character code for the key that was pressed. In Internet Explorer, `event.keyCode` serves the same purpose.

ASCII (American Standard Code for Information Interchange) is the standard numeric code used by most computers to represent characters. It assigns the numbers 0–128 to various characters—for example, the capital letters A through Z are ASCII values 65 to 90.

***By the  
Way***

## Displaying Typed Characters

If you'd rather deal with actual characters than key codes, you can use the `fromCharCode` string method to convert them. This method converts a numeric ASCII code to its corresponding string character. For example, the following statement converts the `event.which` property to a character and stores it in the `key` variable:

```
key = String.fromCharCode(event.which);
```

Because different browsers have different ways of returning the key code, displaying keys browser independently is a bit harder. However, you can create a script that displays keys for either browser. The following function will display each key as it is typed:

```
function DisplayKey(e) {
 // which key was pressed?
 if (e.keyCode) keycode=e.keyCode;
 else keycode=e.which;
 character=String.fromCharCode(keycode);
 // find the object for the destination paragraph
 k = document.getElementById("keys");
 // add the character to the paragraph
 k.innerHTML += character;
}
```

The `DisplayKey()` function receives the event object from the event handler and stores it in the variable `e`. It checks whether the `e.keyCode` property exists, and stores it in the `keycode` variable if present. Otherwise, it assumes the browser is Netscape or Firefox and assigns `keycode` to the `e.which` property.

The remaining lines of the function convert the key code to a character and add it to the paragraph in the document with the `id` attribute `keys`. Listing 9.3 shows a complete example using this function.

### **By the Way**

The final lines in the `DisplayKey()` function use the `getElementById()` function and the `innerHTML` attribute to display the keys you type within a paragraph on the page. This technique is explained in Hour 13, "Using the W3C DOM."

---

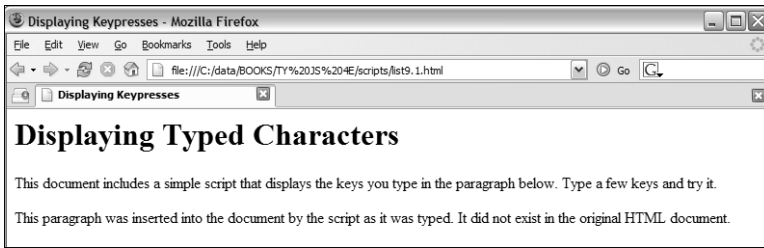
#### **LISTING 9.3**    Displaying Typed Characters

---

```
<html>
<head>
<title>Displaying Keypresses</title>
<script language="javascript" type="text/javascript">
 function DisplayKey(e) {
 // which key was pressed?
 if (e.keyCode) keycode=e.keyCode;
 else keycode=e.which;
 character=String.fromCharCode(keycode);
 // find the object for the destination paragraph
 k = document.getElementById("keys");
 // add the character to the paragraph
 k.innerHTML += character;
 }
</script>
</head>
<body onKeyPress="DisplayKey(event);">
<h1>Displaying Typed Characters</h1>
<p>This document includes a simple script that displays the keys
you type in the paragraph below. Type a few keys and try it. </p>
<p id="keys">
</p>
</body>
</html>
```

---

When you load this example into either Netscape or Internet Explorer, you can type and see the characters you've typed appear in a paragraph of the document. Figure 9.2 shows this example in action in Firefox.



**FIGURE 9.2**  
Firefox displays  
the keypress  
example.

## Using the onLoad and onUnload Events

Another event you'll use frequently is onLoad. This event occurs when the current page (including all of its images) finishes loading from the server.

The onLoad event is related to the window object, and to define it you use an event handler in the <body> tag. For example, the following is a <body> tag that uses a simple event handler to display an alert when the page finishes loading:

```
<body onLoad="alert('Loading complete.');">
```

Because the onLoad event occurs after the HTML document has finished loading and displaying, you cannot use the document.write or document.open statements within an onLoad event handler. This would overwrite the current document.

**Watch  
Out!**

In JavaScript 1.1 and later, images can also have an onLoad event handler. When you define an onLoad event handler for an <img> tag, it is triggered as soon as the specified image has completely loaded.

To set an onLoad event using JavaScript, you assign a function to the onload property of the window object:

```
window.onload = MyFunction;
```

You can also specify an onUnload event for the <body> tag. This event will be triggered whenever the browser unloads the current document—this occurs when another page is loaded or when the browser window is closed.

### Try It Yourself



#### Adding Link Descriptions to a Web Page

One of the most common uses for an event handler is to display descriptions of links when the user moves the mouse over them. For example, moving the mouse over the Order Form link might display a message such as “Order a product or check an order’s status”.

Link descriptions like these are typically displayed with the `onMouseOver` event handler. You will now create a script that displays messages in this manner and clears the message using the `onMouseOut` event handler. You'll use functions to simplify the process.

### **By the Way**

This example uses the `innerHTML` property to display the descriptions within a heading on the page. See Hour 13 for a complete description of this property.

This will also be an example of defining event handlers entirely with JavaScript. The HTML document, shown in Listing 9.4, does not include any `<script>` tags or event handlers—the only thing it requires is some `id` attributes on the objects we will be using in the script.

#### **LISTING 9.4** The HTML Document for the Descriptive Links Example

```
<html>
<head>
<title>Descriptive Links</title>
</head>
<body>
<h1>Descriptive Links</h1>
<p>Move the mouse pointer over one of
these links to view a description:</p>

Order Form
Email
Complaint Department

<h2 id="description"></h2>
<script language="JavaScript" type="text/javascript" src="linkdesc.js">
</script>
</body>
</html>
```

This document defines three links in a bulleted list. Each `<a>` tag is defined with an `id` attribute for the script to use to attach an event handler. The `<h2>` tag with the `id` value `description`, currently blank, will be used to display a description of each link.

### **By the Way**

Notice that the `<script>` tag is below the content of the HTML document. It would not work at the top of the document because the objects the script uses are not yet defined. You can also deal with this issue by using an `onLoad` event handler instead of a simple script to set up the event handlers.



The script will begin with a function to serve as the `onMouseOver` event handler for the links:

```
function hover(e) {
 if (!e) var e = window.event;
 // which link was the mouse over?
 whichlink = (e.target) ? e.target.id : e.srcElement.id;
 // choose the appropriate description
 if (whichlink=="order") desc = "Order a product";
 else if (whichlink=="email") desc = "Send us a message";
 else if (whichlink=="complain") desc = "Insult us, our products, or our
families";
 // display the description in the H2
 d = document.getElementById("description");
 d.innerHTML = desc;
}
```

The `hover` function uses the `target` or `srcElement` properties to find the target object for the link, and then finds its `id` attribute. Three `if` statements evaluate the `id` and choose an appropriate description. Finally, the script uses the `getElementById()` method to find the `<h2>` tag that will display the descriptions, and displays the description using the `innerHTML` property.

The conditional statement on the third line of the `hover` function checks whether the `target` property exists, and if not, it uses the `srcElement` property. This is called *feature sensing*—detecting whether the browser supports a feature—and is explained further in Hour 15, “Unobtrusive Scripting.”

***Did you  
Know?***

One more function will be required. The `cleardesc()` function will serve as the `onMouseOut` event handler and clear the description when the mouse is no longer over one of the links.

```
function cleardesc() {
 d = document.getElementById("description");
 d.innerHTML = "";
}
```

Now that the functions are defined, you need to set them as the event handlers for the links. Each link requires the following three lines of code:

```
orderlink = document.getElementById("order");
orderlink.onmouseover=hover;
orderlink.onmouseout=cleardesc;
```

After using `getElementById()` to find the object with the `id` attribute "order", this sets up the `hover()` and `cleardesc()` functions as its `onMouseOver` and `onMouseOut` event handlers. This will need to be repeated for the other two links. Putting all of this together, the complete JavaScript file for this example is shown in Listing 9.5.

**LISTING 9.5 The JavaScript File for the Link Descriptions Example**

```
function cleardesc() {
 d = document.getElementById("description");
 d.innerHTML = "";
}
function hover(e) {
 if (!e) var e = window.event;
 // which link was the mouse over?
 whichlink = (e.target) ? e.target.id : e.srcElement.id;
 // choose the appropriate description
 if (whichlink=="order") desc = "Order a product";
 else if (whichlink=="email") desc = "Send us a message";
 else if (whichlink=="complain") desc = "Insult us, our products, or our
families";
 // display the description in the H2
 d = document.getElementById("description");
 d.innerHTML = desc;
}
// Set up the event handlers
orderlink = document.getElementById("order");
orderlink.onmouseover=hover;
orderlink.onmouseout=cleardesc;
emailink = document.getElementById("email");
emailink.onmouseover=hover;
emailink.onmouseout=cleardesc;
complainlink = document.getElementById("complain");
complainlink.onmouseover=hover;
complainlink.onmouseout=cleardesc;
```

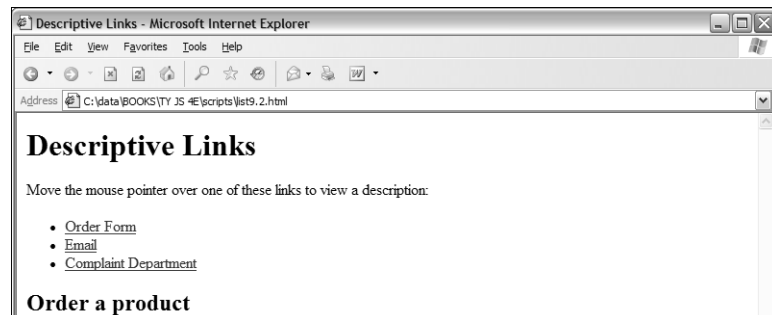
To test the script, store it as `linkdesc.js` in the same folder as the HTML document, and load the HTML file into a browser; this script should work on any JavaScript-capable browser. Internet Explorer's display of the example is shown in Figure 9.3.

**Did you  
Know?**

As usual, you can download the listings for this hour from this book's website.

**FIGURE 9.3**

Internet Explorer displays the descriptive links example.



## Summary

In this hour, you've learned to use events to detect mouse actions, keyboard actions, and other events, such as the loading of the page. You can use event handlers to perform a simple JavaScript statement when an event occurs, or to call a more complicated function.

JavaScript includes a variety of other events. Many of these are related to forms, which you'll learn more about in Hour 11. Another useful event is `onError`, which you can use to prevent error messages from displaying. This event is described in Hour 16, "Debugging JavaScript Applications."

In the next hour, you'll continue learning about the objects in the DOM. Specifically, Hour 10, "Using Windows and Frames," looks at the objects associated with windows, frames, and layers, and how they work with JavaScript.

## Q&A

- Q.** *I noticed that the `<img>` tag in HTML can't have `onMouseOver` or `onClick` event handlers in some browsers. How can my scripts respond when the mouse moves over an image?*
- A.** The easiest way to do this is to make the image a link by surrounding it with an `<a>` tag. You can include the `BORDER=0` attribute to prevent the blue link border from being displayed around the image.
- Q.** *My image rollovers using `onMouseOver` work perfectly in Internet Explorer, but not in Netscape. Why?*
- A.** Re-read the previous answer, and check whether you've used an `onMouseOver` event for an `<img>` tag. This is supported by Internet Explorer and Netscape 6, but not by earlier versions of Netscape.
- Q.** *What happens if I define both `onKeyDown` and `onKeyPress` event handlers? Will they both be called when a key is pressed?*
- A.** The `onKeyDown` event handler is called first. If it returns `true`, the `onKeyPress` event is called. Otherwise, no `keypress` event is generated.
- Q.** *When I use the `onLoad` event, my event handler sometimes executes before the page is done loading, or before some of the graphics. Is there a better way?*
- A.** This is a bug in some older browsers. One solution is to add a slight delay to your script using the `setTimeout` method. You'll learn how to use this method in Hour 10.

## Quiz Questions

Test your knowledge of JavaScript events by answering the following questions.

1. Which of the following is the correct event handler to detect a mouse click on a link?
  - a. `onMouseUp`
  - b. `onLink`
  - c. `onClick`
2. When does the `onLoad` event handler for the `<body>` tag execute?
  - a. When an image is finished loading
  - b. When the entire page is finished loading
  - c. When the user attempts to load another page
3. Which of the following event object properties indicates which key was pressed for an `onKeyPress` event in Internet Explorer?
  - a. `event.which`
  - b. `event.keyCode`
  - c. `event.onKeyPress`

## Quiz Answers

1. c. The event handler for a mouse click is `onClick`.
2. b. The `<body>` tag's `onLoad` handler executes when the page and all its images are finished loading.
3. b. In Internet Explorer, the `event.keyCode` property stores the character code for each keypress.

## Exercises

To gain more experience using event handlers in JavaScript, try the following exercises:

- ▶ Add one or more additional links to the document in Listing 9.4. Add event handlers to the script in Listing 9.5 to display a unique description for each link.
- ▶ Modify Listing 9.5 to display a default welcome message whenever a description isn't being displayed. (Hint: You'll need to include a statement to display the welcome message when the page loads. You'll also need to change the `cleardesc` function to restore the welcome message.)

## HOURL 10

# Using Windows and Frames

---

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

- ▶ The window object hierarchy
- ▶ Creating new windows with JavaScript
- ▶ Delaying your script's actions with timeouts
- ▶ Displaying alerts, confirmations, and prompts
- ▶ Using JavaScript to work with frames
- ▶ Creating a JavaScript-based navigation frame

You should now have a basic understanding of the objects in the level 0 DOM, and the events that can be used with each object.

In this hour, you'll learn more about some of the most useful objects in the level 0 DOM—browser windows and frames—and how JavaScript can work with them.

## **Controlling Windows with Objects**

In Hour 4, “Working with the Document Object Model (DOM),” you learned that you can use DOM objects to represent various parts of the browser window and the current HTML document. You also learned that the history, document, and location objects are all children of the window object.

In this hour, you'll take a closer look at the window object itself. As you've probably guessed by now, this means you'll be dealing with browser windows. A variation of the window object also enables you to work with frames, as you'll see later in this hour.

The window object always refers to the current window (the one containing the script). The `self` keyword is also a synonym for the current window. As you'll learn in the next sections, you can have more than one window on the screen at the same time, and can refer to them with different names.

## Properties of the window Object

Although there is normally a single window object, there might be more than one if you are using pop-up windows or frames. As you learned in Hour 4, the document, history, and location objects are properties (or children) of the window object. In addition to these, each window object has the following properties:

- ▶ **window.closed**—Indicates whether the window has been closed. This only makes sense when working with multiple windows because the current window contains the script and cannot be closed without ending the script.
- ▶ **window.defaultstatus** and **window.status**—The default message for the status line, and a temporary message to display on the status line. Some recent browsers disable status line changes by default, so you might not be able to use these.
- ▶ **window.frames[]**—An array of objects for frames, if the window contains them.
- ▶ **window.name**—The name specified for a frame, or for a window opened by a script.
- ▶ **window.opener**—In a window opened by a script, this is a reference to the window containing the script that opened it.
- ▶ **window.parent**—For a frame, a reference to the parent window containing the frame.
- ▶ **window.screen**—A child object that stores information about the screen the window is in—its resolution, color depth, and so on.
- ▶ **window.self**—A synonym for the current window object.
- ▶ **window.top**—A reference to the top-level window when frames are in use.

### By the Way

The properties of the window.screen object include height, width, availHeight, and availWidth (the available height and width rather than total), and colorDepth, which indicates the color support of the monitor: 8 for 8-bit color, 32 for 32-bit color, and so on.

## Creating a New Window

One of the most convenient uses for the window object is to create a new window. You can do this to display a document—for example, a pop-up advertisement or the instructions for a game—without clearing the current window. You can also create windows for specific purposes, such as navigation windows.

You can create a new browser window with the `window.open()` method. A typical statement to open a new window looks like this:

```
WinObj=window.open("URL", "WindowName", "Feature List");
```

The following are the components of the `window.open()` statement:

- ▶ The `WinObj` variable is used to store the new window object. You can access methods and properties of the new object by using this name.
- ▶ The first parameter of the `window.open()` method is a URL, which will be loaded into the new window. If it's left blank, no web page will be loaded. In this case, you could use JavaScript to fill the window with content.
- ▶ The second parameter specifies a window name (here, `WindowName`). This is assigned to the window object's name property and is used to refer to the window.
- ▶ The third parameter is a list of optional features, separated by commas. You can customize the new window by choosing whether to include the toolbar, status line, and other features. This enables you to create a variety of "floating" windows, which might look nothing like a typical browser window.

The features available in the third parameter of the `window.open()` method include width and height, to set the size of the window in pixels, and several features that can be set to either yes (1) or no (0): toolbar, location, directories, status, menubar, scrollbars, and resizable. You can list only the features you want to change from the default. This example creates a small window with no toolbar or status line:

```
SmallWin = window.open("", "small", "width=100,height=120,toolbar=0,status=0");
```

## Opening and Closing Windows

Of course, you can close windows as well. The `window.close()` method closes a window. Browsers don't normally allow you to close the main browser window without the user's permission; this method's main purpose is for closing windows you have created. For example, this statement closes a window called `updatewindow`:

```
updatewindow.close();
```

As another example, Listing 10.1 shows an HTML document that enables you to open a small new window by pressing a button. You can then press another button to close the new window. The third button attempts to close the current window. Depending on your browser and its settings, this might or might not work. If it does close the window, most browsers will ask for confirmation first.

**LISTING 10.1** An HTML Document That Uses JavaScript to Enable You to Create and Close Windows

```

<html>
<head><title>Create a New Window</title>
</head>
<body>
<h1>Create a New Window</h1>
<hr>
<p>Use the buttons below to test opening and closing windows in JavaScript.</p>
<hr>
<form NAME="winform">
<input TYPE="button" VALUE="Open New Window"
onClick="NewWin=window.open('','NewWin',
'toolbar=no,status=no,width=200,height=100'); ">
<p><input TYPE="button" VALUE="Close New Window"
onClick="NewWin.close();" ></p>
<p><input TYPE="button" VALUE="Close Main Window"
onClick="window.close();" ></p>
</form>

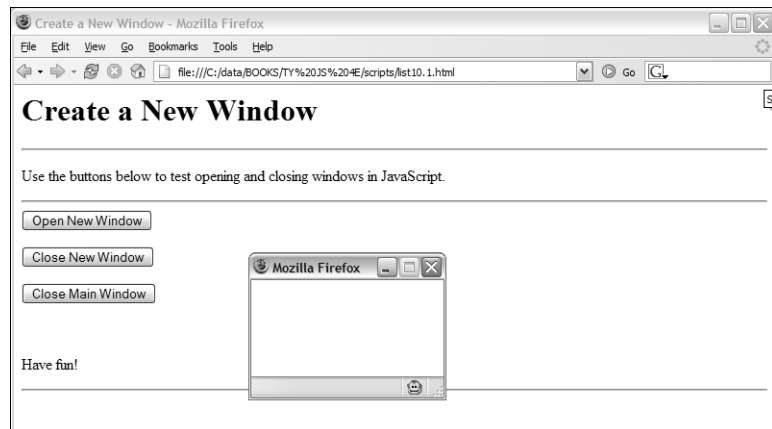
<p>Have fun!</p>
<hr>
</body>
</html>

```

This example uses simple event handlers to do its work, one for each of the buttons. Figure 10.1 shows Firefox's display of this page, with the small new window on top.

**FIGURE 10.1**

A new browser window opened with JavaScript.



## Moving and Resizing Windows

The DOM also enables you to move or resize windows. Although earlier browsers placed some restrictions on this, most modern browsers allow you to move and resize any window freely. You can do this using the following methods for any window object:



- ▶ `window.moveTo()` moves the window to a new position. The parameters specify the x (column) and y (row) position.
- ▶ `window.moveBy()` moves the window relative to its current position. The x and y parameters can be positive or negative, and are added to the current values to reach the new position.
- ▶ `window.resizeTo()` resizes the window to the width and height specified as parameters.
- ▶ `window.resizeBy()` resizes the window relative to its current size. The parameters are used to modify the current width and height.

As an example, Listing 10.2 shows an HTML document with a simple script that enables you to resize or move the main window.

---

**LISTING 10.2** Moving and Resizing the Current Window

---

```
<html>
<head>
<title>Moving and resizing windows</title>
<script language="javascript" type="text/javascript">
 function DoIt() {
 if (document.form1.w.value && document.form1.h.value)
 self.resizeTo(document.form1.w.value, document.form1.h.value);
 if (document.form1.x.value && document.form1.y.value)
 self.moveTo(document.form1.x.value, document.form1.y.value);
 }
</script>
</head>
<body>
<h1>Moving and Resizing Windows</h1>
<form name="form1">
 Width: <input type="text" name="w">

 Height: <input type="text" name="h">

 X-position: <input type="text" name="x">

 Y-position: <input type="text" name="y">

 <input type="button" value="Change Window" onClick="DoIt();">
</form>
</body>
</html>
```

---

In this example, the `DoIt()` function is called as an event handler when you click the Change Window button. This function checks whether you have specified width and height values. If you have, it uses the `self.resizeTo()` method to resize the current window. Similarly, if you have specified x and y values, it uses `self.moveTo()` to move the window.

Depending on their settings, some browsers might not allow your script to resize or move the main window. In particular, Firefox can be configured to disallow it. You

can enable it by selecting Tools, Options from the menu. Select the Content tab, click the Advanced button next to the Enable JavaScript option, and enable the Move or Resize Existing Windows option.

### **Watch Out!**

This is one of those JavaScript features you should think twice about before using. These methods are best used for resizing or moving pop-up windows your script has generated—not as a way to force the user to use your preferred window size, which most users will find very annoying. You should also be aware that browser settings may be configured to prevent resizing or moving windows, so make sure your script still works even without resizing.

## Using Timeouts

Sometimes the hardest thing to get a script to do is to do nothing at all—for a specific amount of time. Fortunately, JavaScript includes a built-in function to do this. The `window.setTimeout` method enables you to specify a time delay and a command that will execute after the delay passes.

### **By the Way**

Timeouts don't actually make the browser stop what it's doing. Although the statement you specify in the `setTimeout` method won't be executed until the delay passes, the browser will continue to do other things while it waits (for example, acting on event handlers).

You begin a timeout with a call to the `setTimeout()` method, which has two parameters. The first is a JavaScript statement, or group of statements, enclosed in quotes. The second parameter is the time to wait in milliseconds (thousandths of seconds). For example, the following statement displays an alert dialog box after 10 seconds:

```
ident=window.setTimeout("alert('Time's up!')",10000);
```

### **Watch Out!**

Like event handlers, timeouts use a JavaScript statement within quotation marks. Make sure that you use a single quote (apostrophe) on each side of each string within the statement, as shown in the preceding example.

A variable (`ident` in this example) stores an identifier for the timeout. This enables you to set multiple timeouts, each with its own identifier. Before a timeout has elapsed, you can stop it with the `clearTimeout()` method, specifying the identifier of the timeout to stop:

```
window.clearTimeout(ident);
```

## Updating a Page with Timeouts

Normally, a timeout only happens once because the statement you specify in the `setTimeout()` method statement is only executed once. But often, you'll want your statement to execute over and over. For example, your script might be updating a clock or a countdown and need to execute once per second.

You can make a timeout repeat by issuing the `setTimeout()` method call again in the function called by the timeout. Listing 10.3 shows an HTML document that demonstrates a repeating timeout.

### LISTING 10.3 Using Timeouts to Update a Page Every Two Seconds

```
<html>
<head><title>Timeout Example</title>
<script language="javascript" type="text/javascript">
var counter = 0;
// call Update function in 2 seconds after first load
ID=window.setTimeout("Update();",2000);
function Update() {
 counter++;
 document.form1.input1.value="The counter is now at " + counter;
// set another timeout for the next count
 ID=window.setTimeout("Update();",2000);
}
</script>
</head>
<body>
<h1>Timeout Example</h1>
<hr><p>
The text value below is being updated every two seconds.
Press the RESET button to restart the count, or the STOP button to stop it.
</p><hr>
<form NAME="form1">
<input TYPE="text" NAME="input1" SIZE="40">

<input TYPE="button" VALUE="RESET" onClick="counter = 0;">

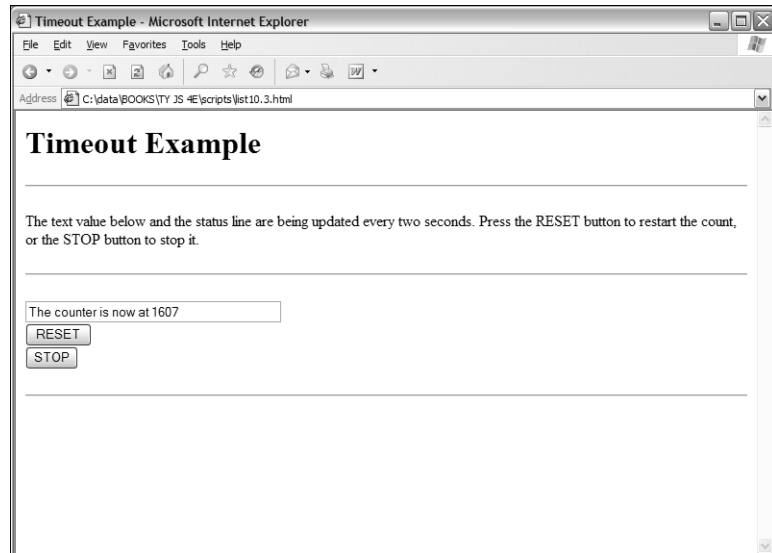
<input TYPE="button" VALUE="STOP" onClick="window.clearTimeout(ID);">
</form>
<hr>
</body>
</html>
```

This program displays a message in a text field every two seconds, including a counter that increments each time. You can use the Reset button to start the count over and the Stop button to stop the counting.

This script calls the `setTimeout()` method when the page loads, and again at each update. The `Update()` function performs the update, adding one to the counter and setting the next timeout. The Reset button sets the counter to zero, and the Stop button demonstrates the `clearTimeout()` method. Figure 10.2 shows Internet Explorer's display of the timeout example after the counter has been running for a while.

**FIGURE 10.2**

The output of the timeout example.



### By the Way

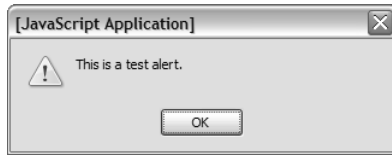
This example and the next one use buttons, which are a simple example of what you can do with HTML forms and JavaScript. You'll learn much more about forms in Hour 11, "Getting Data with Forms."

## Displaying Dialog Boxes

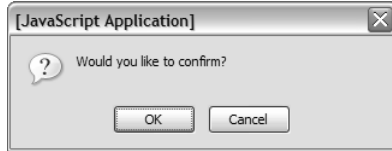
The window object includes three methods that are useful for displaying messages and interacting with the user. You've already used these in some of your scripts.

Here's a summary:

- ▶ `window.alert(message)` displays an alert dialog box, shown in Figure 10.3. This dialog box simply gives the user a message.
- ▶ `window.confirm(message)` displays a confirmation dialog box. This displays a message and includes OK and Cancel buttons. This method returns `true` if OK is pressed and `false` if Cancel is pressed. A confirmation is displayed in Figure 10.4.
- ▶ `window.prompt(message,default)` displays a message and prompts the user for input. It returns the text entered by the user. If the user does not enter anything, the default value is used.

**FIGURE 10.3**

A JavaScript alert dialog box displays a message.

**FIGURE 10.4**

A JavaScript confirm dialog box asks for confirmation.

To use the `confirm()` and `prompt()` methods, use a variable to receive the user's response. For example, this statement displays a prompt and stores the text the user enters in the text variable:

```
text = window.prompt("Enter some text", "Default value");
```

You can usually omit the window object when referring to these methods because it is the default context of a script (for example, `alert("text")`).

***Did you  
Know?***

## Creating a Script to Display Dialog Boxes

As a further illustration of these types of dialog boxes, Listing 10.4 shows an HTML document that uses buttons and event handlers to enable you to test dialog boxes.

### LISTING 10.4 An HTML Document That Uses JavaScript to Display Alerts, Confirmations, and Prompts

```
<html>
<head><title>Alerts, Confirmations, and Prompts</title>
</head>
<body>
<h1>Alerts, Confirmations, and Prompts</h1>
<hr>
Use the buttons below to test dialogs in JavaScript.
<hr>
<form NAME="winform">
<p><input TYPE="button" VALUE="Display an Alert"
onClick="window.alert('This is a test alert.');" "></p>
<p><input TYPE="button" VALUE="Display a Confirmation"
onClick="window.confirm('Would you like to confirm?');" "></p>
<p><input TYPE="button" VALUE="Display a Prompt"
onClick="window.prompt('Enter some Text:', 'This is the default value');" ">
</p>
</form>

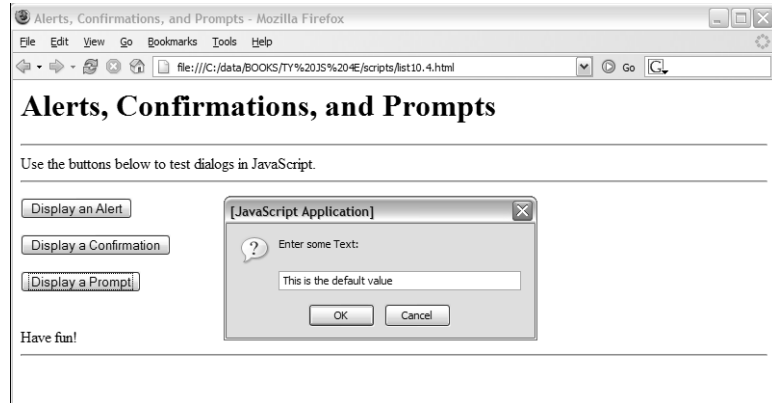
Have fun!
<hr>
</body>
</html>
```

This document displays three buttons, and each one uses an event handler to display one of the dialog boxes.

Figure 10.5 shows the script in Listing 10.4 in action. The prompt dialog box is currently displayed and shows the default value.

**FIGURE 10.5**

The dialog box example's output, including a prompt dialog box.



## Working with Frames

Browsers also support *frames*, which enable you to divide the browser window into multiple panes. Each frame can contain a separate URL or the output of a script.

### Using JavaScript Objects for Frames

When a window contains multiple frames, each frame is represented in JavaScript by a frame object. This object is equivalent to a window object, but it is used for dealing specifically with that frame. The frame object's name is the same as the NAME attribute you give it in the <frame> tag.

Remember the window and self keywords, which refer to the current window? When you are using frames, these keywords refer to the current frame instead. Another keyword, parent, enables you to refer to the main window.

Each frame object in a window is a child of the parent window object. Suppose you define a set of frames using the following HTML:

```
<frameset ROWS="*,*" COLS="*,*">
<frame NAME="topleft" SRC="topleft.htm">
<frame NAME="topright" SRC="topright.htm">
<frame NAME="bottomleft" SRC="botleft.htm">
<frame NAME="bottomright" SRC="botright.htm">
</frameset>
```

This simply divides the window into quarters. If you have a JavaScript program in the `topleft.htm` file, it would refer to the other windows as `parent.topright`, `parent.bottomleft`, and so on. The keywords `window` and `self` would refer to the `topleft` frame.

If you use nested framesets, things are a bit more complicated. `window` still represents the current frame, `parent` represents the frameset containing the current frame, and `top` represents the main frameset that contains all the others.

***By the  
Way***

## The frames Array

Rather than referring to frames in a document by name, you can use the `frames` array. This array stores information about each of the frames in the document. The frames are indexed starting with zero and beginning with the first `<frame>` tag in the frameset document.

For example, you could refer to the frames defined in the previous example using array references:

- ▶ `parent.frames[0]` is equivalent to the `topleft` frame.
- ▶ `parent.frames[1]` is equivalent to the `topright` frame.
- ▶ `parent.frames[2]` is equivalent to the `bottomleft` frame.
- ▶ `parent.frames[3]` is equivalent to the `bottomright` frame.

You can refer to a frame using either method interchangeably, and depending on your application, you should use the most convenient method. For example, a document with 10 frames would probably be easier to use by number, but a simple two-frame document is easier to use if the frames have meaningful names.

## Try it Yourself



### Using Frames with JavaScript

As a simple example of addressing frames using JavaScript, you will now create an HTML document that divides the window into four frames, and a document with a script for the top-left corner frame. Buttons in the top-left frame will trigger JavaScript event handlers that display text in the other frames.

To begin, you will need a frameset document. Listing 10.5 shows a simple HTML document to divide the window into four frames.

**LISTING 10.5** An HTML Document That Divides the Window into Four Frames

---

```
<frameset ROWS="*,*" COLS="*,*">
<frame NAME="top_left" SRC="topleft.html">
<frame NAME="top_right" SRC="">
<frame NAME="bottom_left" SRC="">
<frame NAME="bottom_right" SRC="">
</frameset>
```

---

The first frame defined here, `top_left`, will contain an HTML document and a simple script. Listing 10.6 shows the HTML and JavaScript code for the top-left frame.

**LISTING 10.6** The HTML and JavaScript for the Frame Example

---

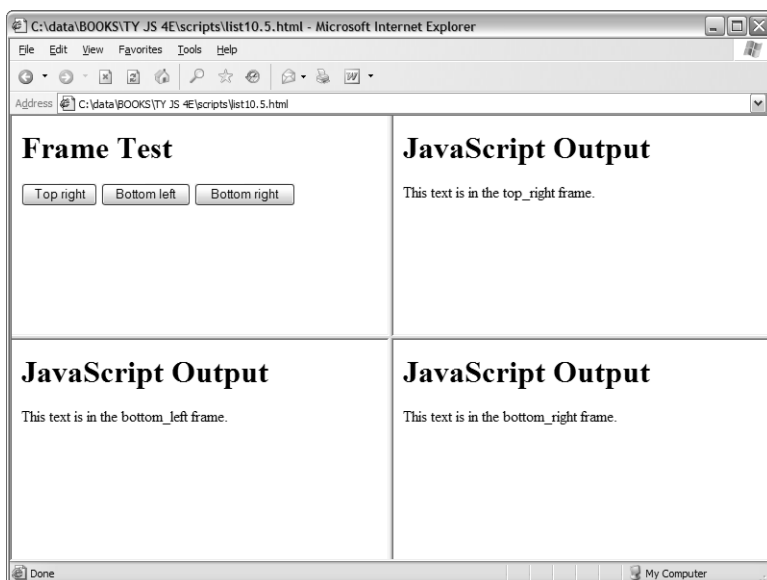
```
<html>
<head>
<title>Frame Test</title>
<script language="javascript" type="text/javascript">
function FillFrame(framename) {
 // Find the object for the frame
 theframe=parent[framename];
 // Open and clear the frame's document
 theframe.document.open();
 // Create some output
 theframe.document.write("<h1>JavaScript Output</h1>");
 theframe.document.write("<p>This text is in the ");
 theframe.document.write(framename + " frame.</p>");
}
</script>
</head>
<body>
<h1>Frame Test</h1>
<form name="form1">
<input type="button" value="Top right"
onClick="FillFrame('top_right');">
<input type="button" value="Bottom left"
onClick="FillFrame('bottom_left');">
<input type="button" id="js" value="Bottom right"
onClick="FillFrame('bottom_right');">
</form>
</body>
</html>
```

---

This document defines three buttons with event handlers that call the `FillFrame()` function with a parameter for the frame name. The function finds the correct child of the parent window object for the specified frame, uses `document.open` to create a new document in the frame, and uses `document.write` to display text in the frame.

To try this example, save Listing 10.6 as `topleft.html` in the same folder as the `frameset` document from Listing 10.5, and load Listing 10.5 into a browser. Figure 10.6 shows the result of this example after all three buttons have been clicked.



**FIGURE 10.6**

The frame example as displayed by Internet Explorer.

## Summary

In this hour, you've learned how to use the window object to work with browser windows, and used its properties and methods to set timeouts and display dialog boxes. You've also learned how JavaScript can work with framed documents.

In the next hour, you'll move on to another unexplored area of the JavaScript object hierarchy—the form object. You'll learn how to use forms to create some of the most useful applications of JavaScript.

## Q&A

- Q.** *When a script is running in a window created by another script, how can it refer back to the original window?*
- A.** JavaScript 1.1 and later include the `window.opener` property, which lets you refer to the window that opened the current window.
- Q.** *I've heard about layers, which are similar to frames, but more versatile, and are supported in the latest browsers. Can I use them with JavaScript?*
- A.** Yes. You'll learn how to use layers with JavaScript in Hour 13, "Using the W3C DOM."

- Q.** *How can I update two frames at once when the user clicks on a single link?*
- A.** You can do this by using an event handler, as in Listing 10.6, and including two statements to load URLs into different frames.

## Quiz Questions

Test your knowledge of the DOM's window features by answering the following questions.

- 1.** Which of the following methods displays a dialog box with OK and Cancel buttons, and waits for a response?
  - a.** `window.alert`
  - b.** `window.confirm`
  - c.** `window.prompt`
- 2.** What does the `window.setTimeout` method do?
  - a.** Executes a JavaScript statement after a delay
  - b.** Locks up the browser for the specified amount of time
  - c.** Sets the amount of time before the browser exits automatically
- 3.** You're working with a document that contains three frames with the names `first`, `second`, and `third`. If a script in the second frame needs to refer to the first frame, what is the correct syntax?
  - a.** `window.first`
  - b.** `parent.first`
  - c.** `frames.first`

## Quiz Answers

- 1.** **b.** The `window.confirm` method displays a dialog box with OK and Cancel buttons.
- 2.** **a.** The `window.setTimeout` method executes a JavaScript statement after a delay.
- 3.** **b.** The script in the second frame would use `parent.first` to refer to the first frame.

## Exercises

If you want to study the window object and its properties and methods further, perform these exercises:

- ▶ Return to the date/time script you created in Hour 2, “Creating Simple Scripts.” This script only displays the date and time once when the page is loaded. Using timeouts, you can modify the script to reload automatically every second or two and display a “live” clock. (Use the `location.reload()` method, described in Hour 4.)
- ▶ Modify the examples in Listings 10.5 and 10.6 to use three horizontal frames instead of four frames in a grid. Change the buttons to make it clear which frame they will affect.

*This page intentionally left blank*

## HOURL 11

# Getting Data with Forms

---

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

- ▶ Understanding HTML forms
- ▶ Creating a form
- ▶ Using the `form` object to work with forms
- ▶ How form elements are represented by JavaScript
- ▶ Getting data from a form
- ▶ Sending form results by email
- ▶ Validating a form with JavaScript

In this hour, you'll explore one of the most powerful uses for JavaScript: working with HTML forms. You can use JavaScript to make a form more interactive, validate data the user enters, and enter data based on other data.

## **The Basics of HTML Forms**

Forms are among the most useful features of the HTML language. As you'll learn during this hour, adding JavaScript to forms can make them more interactive and provide a number of useful features. The first step in creating an interactive form is to create the HTML form itself.

### **Defining a Form**

An HTML form begins with the `<form>` tag. This tag indicates that a form is beginning, and it enables form elements to be used. The `<form>` tag includes several attributes:

- ▶ `name` is simply a name for the form. You can use forms without giving them names, but you'll need to assign a name to a form in order to easily use it with JavaScript.

- ▶ method is either GET or POST; these are the two ways the data can be sent to the server.
- ▶ action is the CGI script that the form data will be sent to when submitted. You can also use the `mailto:` action to send the form's results to an email address, as described later in this hour.
- ▶ enctype is the MIME type the form's data will be encoded with. This is usually not necessary; see the "Sending Form Results by Email" section of this hour for an example that requires it.

For example, here is a `<form>` tag for a form named `Order`. This form uses the GET method and sends its data to a CGI script called `order.cgi` in the same directory as the web page itself:

```
<form name="Order" method="GET" action="order.cgi">
```

For a form that will be processed entirely by JavaScript (such as a calculator or an interactive game), the `method` and `action` attributes are not needed. You can use a simple `<form>` tag that names the form:

```
<form name="calcform">
```

The `<form>` tag is followed by one or more form elements. These are the data fields in the form, such as text fields, buttons, and check boxes. In the next section, you'll learn how JavaScript assigns objects to each of the form elements.

## Using the form Object with JavaScript

Each form in your HTML page is represented in JavaScript by a form object, which has the same name as the `NAME` attribute in the `<form>` tag you used to define it.

Alternatively, you can use the `forms` array to refer to forms. This array includes an item for each form element, indexed starting with 0. For example, if the first form in a document has the name `form1`, you can refer to it in one of two ways:

```
document.form1
document.forms[0]
```

## The form Object's Properties

Along with the elements, each form object also has a list of properties, most of which are defined by the corresponding `<form>` tag. You can also set these from within JavaScript. They include the following:

- ▶ `action` is the form's `action` attribute, or the program to which the form data will be submitted.
- ▶ `encoding` is the MIME type of the form, specified with the `enctype` attribute. In most cases, this is not needed. See the "Sending Form Results by Email" section of this hour for an example of its use.
- ▶ `length` is the number of elements in the form. You cannot change this property.
- ▶ `method` is the method used to submit the form, either GET or POST. This determines the data format used to send the form result to a CGI script, and does not affect JavaScript.
- ▶ `target` specifies the window in which the result of the form (from the CGI script) will be displayed. Normally, this is done in the main window, replacing the form itself, but you can use this attribute to work with pop-up windows or frames.

## Submitting and Resetting Forms

The form object has two methods, `submit()` and `reset()`. You can use these methods to submit the data or reset the form yourself, without requiring the user to press a button. One reason for this is to submit the form when the user clicks an image or performs another action that would not usually submit the form.

If you use the `submit()` method to send data to a server or by email, most browsers will prompt the user to verify that he or she wants to submit the information. There's no way to do this behind the user's back.

**Watch  
Out!**

## Detecting Form Events

The form object has two event handlers, `onSubmit` and `onReset`. You can specify a group of JavaScript statements or a function call for these events within the `<form>` tag that defines the form.

If you specify a statement or a function for the `onSubmit` event, the statement is called before the data is submitted to the CGI script. You can prevent the submission from happening by returning a value of `false` from the `onSubmit` event handler. If the statement returns `true`, the data will be submitted. In the same fashion, you can prevent a Reset button from working with an `onReset` event handler.

## Scripting Form Elements

The most important property of the form object is the `elements` array, which contains an object for each of the form elements. You can refer to an element by its own name or by its index in the array. For example, the following two expressions both refer to the first element in the order form, the `name1` text field:

```
document.order.elements[0]
document.order.name1
```

### By the Way

Both forms and elements can be referred to by their own names or as indices in the `forms` and `elements` arrays. For clarity, the examples in this hour use individual form and element names rather than array references. You'll also find it easier to use names in your own scripts.

If you do refer to forms and elements as arrays, you can use the `length` property to determine the number of objects in the array: `document.forms.length` is the number of forms in a document, and `document.form1.elements.length` is the number of elements in the `form1` form.

You can also access form elements using the W3C DOM. In this case, you use an `id` attribute on the form element in the HTML document, and use the `document.getElementById()` method to find the object for the form. For example, this statement finds the object for the text field called `firstname` and stores it in the `fn` variable:

```
fn = document.getElementById("firstname");
```

This allows you to quickly access a form element without first finding the form object. You can assign an `id` to the `<form>` tag and find the corresponding object if you need to work with the form's properties and methods.

### Did you Know?

See Hour 13, "Using the W3C DOM," for details on the `document.getElementById()` method.

## Text Fields

Probably the most commonly used form elements are text fields. You can use them to prompt for a name, an address, or any information. With JavaScript, you can display text in the field automatically. The following is an example of a simple text field:

```
<input type="TEXT" name="text1" value="hello" SIZE="30">
```



This defines a text field called `text1`. The field is given a default value of "hello" and allows up to 30 characters to be entered. JavaScript treats this field as a text object with the name `text1`.

Text fields are the simplest to work with in JavaScript. Each text object has the following properties:

- ▶ `name` is the name given to the field. This is also used as the object name.
- ▶ `defaultValue` is the default value and corresponds to the `VALUE` attribute. This is a read-only property.
- ▶ `value` is the current value. This starts out the same as the default value, but can be changed, either by the user or by JavaScript functions.

When you work with text fields, most of the time you will use the `value` attribute to read the value the user has entered or to change the value. For example, the following statement changes the value of a text field called `username` in the `order` form to "John Q. User":

```
document.order.username.value = "John Q. User"
```

## Text Areas

Text areas are defined with their own tag, `<textarea>`, and are represented by the `textarea` object. There is one major difference between a text area and a text field: Text areas enable the user to enter more than just one line of information. Here is an example of a text area definition:

```
<textarea name="text1" rows="2" cols="70">
This is the content of the TEXTAREA tag.
</textarea>
```

This HTML defines a text area called `text1`, with two rows and 70 columns available for text. In JavaScript, this would be represented by a text area object called `text1` under the form object.

The text between the opening and closing `<textarea>` tags is used as the initial value for the text area. You can include line breaks within the default value with the special character `\n`.

## Working with Text in Forms

The text and `textarea` objects also have a few methods you can use:

- ▶ `focus()` sets the focus to the field. This positions the cursor in the field and makes it the current field.

- ▶ `blur()` is the opposite; it removes the focus from the field.
- ▶ `select()` selects the text in the field, just as a user can do with the mouse. All of the text is selected; there is no way to select part of the text.

You can also use event handlers to detect when the value of a text field changes. The `text` and `textarea` objects support the following event handlers:

- ▶ The `onFocus` event happens when the text field gains focus.
- ▶ The `onBlur` event happens when the text field loses focus.
- ▶ The `onChange` event happens when the user changes the text in the field and then moves out of it.
- ▶ The `onSelect` event happens when the user selects some or all of the text in the field. Unfortunately, there's no way to tell exactly which part of the text was selected. (If the text is selected with the `select()` method described previously, this event is not triggered.)

If used, these event handlers should be included in the `<input>` tag declaration. For example, the following is a text field including an `onChange` event that displays an alert:

```
<input type="TEXT" name="text1" onChange="window.alert('Changed.');">
```

## Buttons

One of the most useful types of form element is a button. Buttons use the `<input>` tag and can use one of three different types:

- ▶ `type=SUBMIT` is a Submit button. This button causes the data in the form fields to be sent to the CGI script.
- ▶ `type=RESET` is a Reset button. This button sets all the form fields back to their default value, or blank.
- ▶ `type=BUTTON` is a generic button. This button performs no action on its own, but you can assign it one using a JavaScript event handler.

All three types of buttons include a `name` attribute to identify the button and a `value` attribute that indicates the text to display on the button's face. A few buttons were used in the examples in Hour 10, "Using Windows and Frames." As another example, the following defines a Submit button with the name `sub1` and the value "Click Here":

```
<input type="SUBMIT" name="sub1" value="Click Here">
```

If the user presses a Submit or a Reset button, you can detect it with the `onSubmit` or `onReset` event handlers, described earlier in this hour. For generic buttons, you can use an `onClick` event handler.

## Check Boxes

A check box is a form element that looks like a small box. Clicking on the check box switches between the checked and unchecked states, which is useful for indicating Yes or No choices in your forms. You can use the `<input>` tag to define a check box. Here is a simple example:

```
<input type="CHECKBOX" name="check1" value="Yes" checked>
```

Again, this gives a name to the form element. The `value` attribute assigns a meaning to the check box; this is a value that is returned to the server if the box is checked. The default value is “on.” The `checked` attribute can be included to make the box checked by default.

A check box is simple: It has only two states. Nevertheless, the `checkbox` object in JavaScript has four different properties:

- ▶ `name` is the name of the check box, and also the object name.
- ▶ `value` is the “true” value for the check box—usually `on`. This value is used by server-side programs to indicate whether the check box was checked. In JavaScript, you should use the `checked` property instead.
- ▶ `defaultChecked` is the default status of the check box, assigned by the `checked` attribute in HTML.
- ▶ `checked` is the current value. This is a Boolean value: `true` for checked and `false` for unchecked.

To manipulate the check box or use its value, you use the `checked` property. For example, this statement turns on a check box called `same` in the order form:

```
document.order.same.checked = true;
```

The check box has a single method, `click()`. This method simulates a click on the box. It also has a single event, `onClick`, which occurs whenever the check box is clicked. This happens whether the box was turned on or off, so you’ll need to examine the `checked` property to see what happened.

## Radio Buttons

Another element for decisions is the radio button, using the `<input>` tag's `RADIO` type. Radio buttons are also known as option buttons. These are similar to check boxes, but they exist in groups and only one button can be checked in each group. They are used for a multiple-choice or “one of many” input. Here's an example of a group of radio buttons:

```
<input type="RADIO" name="radio1" value="Option1" checked> Option 1
<input type="RADIO" name="radio1" value="Option2"> Option 2
<input type="RADIO" name="radio1" value="Option3"> Option 3
```

These statements define a group of three radio buttons. The `name` attribute is the same for all three (which is what makes them a group). The `value` attribute is the value passed to a script or a CGI program to indicate which button is selected—be sure you assign a different value to each button.

### By the Way

Radio buttons are named for their similarity to the buttons on old pushbutton radios. Those buttons used a mechanical arrangement so that when you pushed one button in, the others popped out.

As for scripting, radio buttons are similar to check boxes, except that an entire group of them shares a single name and a single object. You can refer to the following properties of the radio object:

- ▶ `name` is the name common to the radio buttons.
- ▶ `length` is the number of radio buttons in the group.

To access the individual buttons, you treat the radio object as an array. The buttons are indexed, starting with `0`. Each individual button has the following properties:

- ▶ `value` is the value assigned to the button. (This is used by the server.)
- ▶ `defaultChecked` indicates the value of the checked attribute and the default state of the button.
- ▶ `checked` is the current state.

For example, you can check the first radio button in the `radio1` group on the `form1` form with this statement:

```
document.form1.radio1[0].checked = true;
```

However, if you do this, be sure you set the other values to `false` as needed. This is not done automatically. You can use the `click()` method to do both of these in one step.

Like a check box, radio buttons have a `click()` method and an `onClick` event handler. Each radio button can have a separate statement for this event.

You can have more than one group of radio buttons on a page, and they will act independently. Assign a separate name attribute value to each group.

***Did you  
Know?***

## Drop-Down Lists

A final form element is also useful for multiple-choice selections. The `<select>` HTML tag is used to define a *selection list*, or a drop-down list of text items. The following is an example of a selection list:

```
<select name="select1" SIZE=40>
<option value="choice1" SELECTED>This is the first choice.
<option value="choice2">This is the second choice.
<option value="choice3">This is the third choice.
</select>
```

Each of the `<option>` tags defines one of the possible choices. The `value` attribute is the name that is returned to the program, and the text outside the `<option>` tag is displayed as the text of the option.

An optional attribute to the `<select>` tag, `multiple`, can be specified to allow multiple items to be selected. Browsers usually display a single-selection `<select>` as a drop-down list and a multiple-selection list as a scrollable list.

The object for selection lists is the `select` object. The object itself has the following properties:

- ▶ `name` is the name of the selection list.
- ▶ `length` is the number of options in the list.
- ▶ `options` is the array of options. Each selectable option has an entry in this array.
- ▶ `selectedIndex` returns the index value of the currently selected item. You can use this to check the value easily. In a multiple-selection list, this indicates the first selected item.

The options array has a single property of its own, `length`, which indicates the number of selections. In addition, each item in the options array has the following properties:

- ▶ `index` is the index into the array.
- ▶ `defaultSelected` indicates the state of the selected attribute.

- ▶ `selected` is the current state of the option. Setting this property to `true` selects the option. The user can select multiple options if the `multiple` attribute is included in the `<select>` tag.
- ▶ `name` is the value of the `name` attribute. This is used by the server.
- ▶ `text` is the text that is displayed in the option.

The `select` object has two methods—`blur()` and `focus()`—which perform the same purposes as the corresponding methods for text objects. The event handlers are `onBlur`, `onFocus`, and `onChange`, also similar to other objects.

### **By the Way**

You can change selection lists dynamically—for example, choosing a product in one list could control which options are available in another list. You can also add and delete options from the list.

Reading the value of a selected item is a two-step process. You first use the `selectedIndex` property, and then use the `value` property to find the value of the selected choice. Here's an example:

```
ind = document.navform.choice.selectedIndex;
val = document.navform.choice.options[ind].value;
```

This uses the `ind` variable to store the selected index, and then assigns the `val` variable to the value of the selected choice. Things are a bit more complicated with a multiple selection: You have to test each option's `selected` attribute separately.

## **Displaying Data from a Form**

As a simple example of using forms, Listing 11.1 shows a form with name, address, and phone number fields, as well as a JavaScript function that displays the data from the form in a pop-up window.

### **LISTING 11.1 A Form That Displays Data in a Pop-up Window**

```
<html>
<head>
<title>Form Example</title>
<script language="JavaScript" type="text/javascript">
function display() {
 DispWin = window.open('', 'NewWin',
```

**LISTING 11.1** Continued

```
'toolbar=no,status=no,width=300,height=200')
 message = "NAME: " + document.form1.yourname.value;
 message += "ADDRESS: " + document.form1.address.value;
 message += "PHONE: " + document.form1.phone.value + "";
 DispWin.document.write(message);
}
</script>
</head>
<body>
<h1>Form Example</h1>
Enter the following information. When you press the Display button,
the data you entered will be displayed in a pop-up window.
<form name="form1">
<p>Name: <input type="TEXT" size="20" name="yourname">
</p>
<p>Address: <input type="TEXT" size="30" name="address">
</p>
<p>Phone: <input type="TEXT" size="15" name="phone">
</p>
<p><input type="BUTTON" value="Display" onClick="display();"></p>
</form>
</body>
</html>
```

Here is a breakdown of how this HTML document and script work:

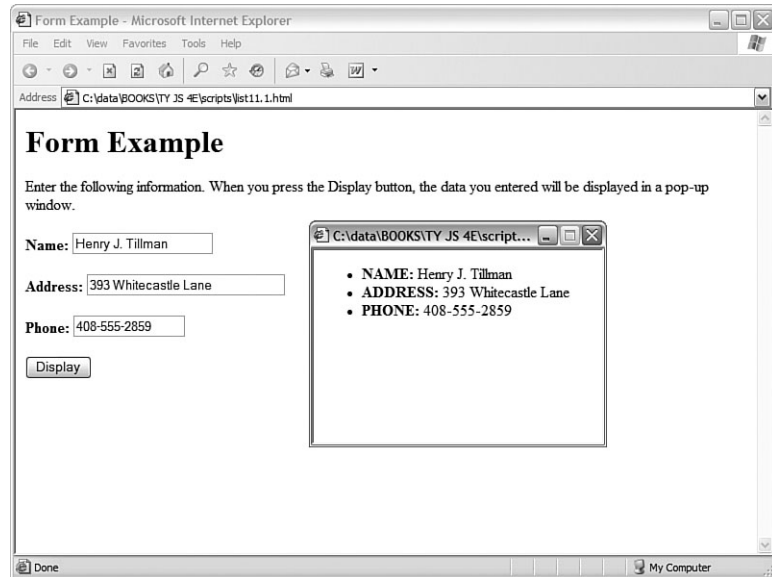
- ▶ The <script> section in the document's header defines a function called `display()` that opens a new window (as described in Hour 10) and displays the information from the form.
- ▶ The <form> tag begins the form. Because this form is handled entirely by JavaScript, no form action or method is needed.
- ▶ The <input> tags define the form's three fields: `yourname`, `address`, and `phone`. The last <input> tag defines the Display button, which is set to run the `display()` function.

As usual, you can download the listings for this hour from this book's website.

***Did you  
Know?***

Figure 11.1 shows this form in action. The Display button has been pressed, and the pop-up window shows the results.

**FIGURE 11.1**  
Displaying data  
from a form in a  
pop-up window.



## Sending Form Results by Email

One easy way to use a form is to send the results by email. You can do this without using any JavaScript, although you could use JavaScript to validate the information entered (as you'll learn later in this hour).

To send a form's results by email, you use the `mailto:` action in the form's action attribute. Listing 11.2 is a modified version of the name and address form from Listing 11.1 that sends the results by email.

### LISTING 11.2 Sending a Form's Results by Email

```
<html>
<head>
<title>Email Form Example</title>
</head>
<body>
<h1>Email Form Example</h1>
Enter the following information. When you press the Submit button,
the data you entered will be sent by email.
<form name="form1" action="mailto:user@host.com"
 enctype="text/plain" method="POST">
 <p>Name: <input type="TEXT" size="20" name="yourname">
</p>
 <p>Address: <input type="TEXT" size="30" name="address">
</p>
 <p>Phone: <input type="TEXT" size="15" name="phone">
</p>
 <p><input type="submit" value="Submit"></p>
</form>
</body>
</html>
```



To use this form, change `user@host.com` in the `action` attribute of the `<form>` tag to your email address. Notice the `enctype=text/plain` attribute in the `<form>` tag. This ensures that the information in the email message will be in a readable plain-text format rather than encoded.

Although this provides a quick and dirty way of retrieving data from a form, the disadvantage of this technique is that it is highly browser dependent. Whether it will work for each user of your page depends on the configuration of his or her browser and email client.

Because this technique does not consistently work on all browsers, I don't recommend you use it. For a more reliable way of sending form results, you can use a CGI form-to-email gateway. Several free CGI scripts and services are available. You'll find links to them on this book's website.

**Watch  
Out!**

## Try It Yourself



### Validating a Form

One of JavaScript's most useful purposes is validating forms. This means using a script to verify that the information entered is valid—for example, that no fields are blank and that the data is in the right format.

You can use JavaScript to validate a form whether it's submitted by email or to a CGI script, or is simply used by a script. Listing 11.3 is a version of the name and address form that includes validation.

#### LISTING 11.3 A Form with a Validation Script

```
<html>
<head>
<title>Form Example</title>
<script language="JavaScript" type="text/javascript">
function validate() {
 if (document.form1.yourname.value.length < 1) {
 alert("Please enter your full name.");
 return false;
 }
 if (document.form1.address.value.length < 3) {
 alert("Please enter your address.");
 return false;
 }
 if (document.form1.phone.value.length < 3) {
 alert("Please enter your phone number.");
 return false;
 }
 return true;
}
```

**LISTING 11.3 Continued**


---

```

</script>
</head>
<body>
<h1>Form Example</h1>
<p>Enter the following information. When you press the Submit button,
the data you entered will be validated, then sent by email.</p>
<form name="form1" action="mailto:user@host.com" enctype="text/plain"
method="POST" onSubmit="return validate();">
<p>Name: <input type="TEXT" size="20" name="yourname">
</p>
<p>Address: <input type="TEXT" size="30" name="address">
</p>
<p>Phone: <input type="TEXT" size="15" name="phone">
</p>
<p><input type="SUBMIT" value="Submit"></p>
</form>
</body>
</html>

```

---

This form uses a function called `validate()` to check the data in each of the form fields. Each `if` statement in this function checks a field's length. If the field is long enough to be valid, the form can be submitted; otherwise, the submission is stopped and an alert message is displayed.

**By the  
Way**

The validation in this script is basic—you could go further and ensure that the phone field contains only numbers, and the right amount of digits, by using JavaScript's string features described in Hour 5, "Using Variables, Strings, and Arrays."

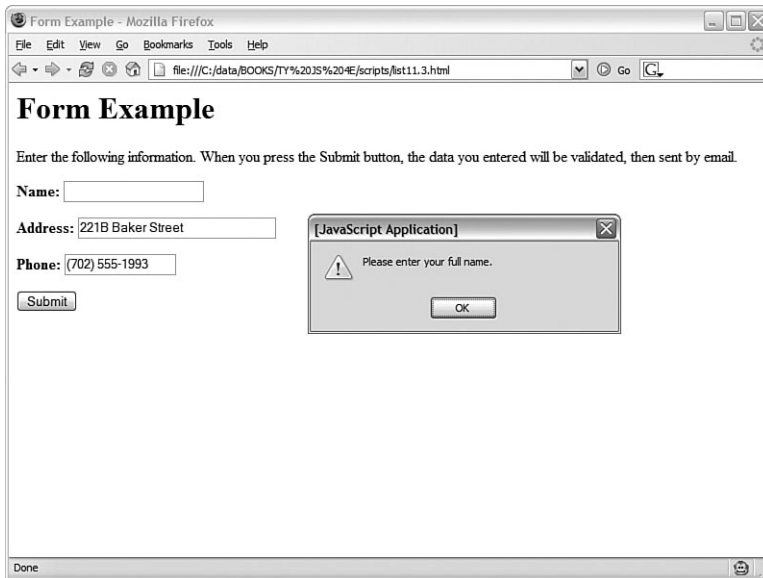
This form is set up to send its results by email, as in Listing 11.2. If you wish to use this feature, be sure to read the information about email forms earlier in this hour and change `user@host.com` to your desired email address.

The `<form>` tag uses an `onSubmit` event handler to call the `validate()` function. The `return` keyword ensures that the value returned by `validate()` will determine whether the form is submitted.

**Did you  
Know?**

You can also use the `onChange` event handler in each form field to call a validation routine. This allows the field to be validated before the Submit button is pressed.

Figure 11.2 shows this script in action, as displayed by Firefox. The form has been filled out except for the name, and a dialog box indicates that the name needs to be entered.



**FIGURE 11.2**  
The form validation example in action.

## Summary

During this hour, you've learned all about HTML forms and how they can be used with JavaScript. You learned about the `form` object and the objects for the various form elements, and used them in several example scripts.

You also learned how to submit a form by email, and how to use JavaScript to validate a form before it is submitted.

In the next hour, you'll look at CSS (Cascading Style Sheets)—a standards-compliant way to achieve just about any visual effect on a page, and the foundation for using JavaScript to change a page's appearance.

## Q&A

- Q.** *If I use JavaScript to add validation and other features to my form, can users with non-JavaScript browsers still use the form?*
- A.** Yes, if you're careful. Be sure to use a Submit button rather than the submit action. Also, the CGI script might receive nonvalidated data, so be sure to include the same validation in the CGI script. Non-JavaScript users will be able to use the form, but won't receive instant feedback about their errors.

- Q.** *Can I add new form elements on the fly or change them—for example, change a text box into a password field?*
- A.** Not in the traditional way described in this hour. However, you can change any aspect of a page, including adding, removing, or changing form elements, using the W3C DOM. See Hour 13 for details.
- Q.** *Is there any way to create a large number of text fields without dealing with different names for all of them?*
- A.** Yes. If you use the same name for several elements in the form, their objects will form an array. For example, if you defined 20 text fields with the name member, you could refer to them as member[0] through member[19]. This also works with other types of form elements.
- Q.** *Is there a way to place the cursor on a particular field when the form is loaded, or after my validation routine displays an error message?*
- A.** Yes. You can use the field's focus() method to send the cursor there. To do this when the page loads, you can use the onLoad method in the <body> tag. However, there is no way to place the cursor in a particular position within the field.

## Quiz Questions

Test your knowledge of JavaScript and forms by answering the following questions.

- 1.** Which of these attributes of a <form> tag determines where the data will be sent?
  - a.** action
  - b.** method
  - c.** name
- 2.** Where do you place the onSubmit event handler to validate a form?
  - a.** In the <body> tag
  - b.** In the <form> tag
  - c.** In the <input> tag for the Submit button

3. What can JavaScript do with forms that a CGI script can't?
  - a. Cause all sorts of problems
  - b. Give the user instant feedback about errors
  - c. Submit the data to a server

## Quiz Answers

1. a. The action attribute determines where the data is sent.
2. b. You place the onSubmit event handler in the <form> tag.
3. b. JavaScript can validate a form and let the user know about errors immediately, without waiting for a response from a server.

## Exercises

To further explore the JavaScript features you learned about in this hour, you can perform the following exercises:

- ▶ Change the validate function in Listing 11.3 so that after a message is displayed indicating that a field is wrong, the cursor is moved to that field. (Use the `focus()` method for the appropriate form element.)
- ▶ Add a text field to the form in Listing 11.3 for an email address. Add a feature to the validate function that verifies that the email address is at least five characters and that it contains the @ symbol.

*This page intentionally left blank*

## HOURL 12

# Working with Style Sheets

---

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

- ▶ Why style sheets are needed
- ▶ How to define Cascading Style Sheets (CSS)
- ▶ How to use a style sheet in a document
- ▶ Using an external style sheet file
- ▶ Using JavaScript to change styles dynamically

This hour begins with an introduction to style sheets, which you can use to take more control over how the browser displays your document. You can also use JavaScript with style sheets to change the appearance of a page dynamically.

## **Style and Substance**

If you've ever tried to make a really good-looking web page, you've probably encountered some problems. First of all, HTML doesn't give you very much control over a page's appearance. For example, you can't change the amount of space between words—in fact, you can't even use two spaces between words because they'll be converted to a single space.

Second, even when you do your best to make a perfect-looking document using HTML, you will find that it doesn't necessarily display the same way on all browsers—or even on different computers running the same browser.

The reason for these problems is simple: HTML was never meant to handle such things as layout, justification, and spacing. HTML deals with a document's *structure*—in other words, how the document is divided into paragraphs, headings, lists, and other elements.

This isn't a bad thing. In fact, it's one of the most powerful features of HTML. You only define the structure of the document, so it can be displayed in all sorts of different ways

without changing its meaning. For example, a well-written HTML document can be displayed in Netscape, Firefox, or Internet Explorer, which generally treat elements the same way—there is a space between paragraphs, headings are in big, bold text, and so on.

Because HTML only defines the structure, the same document can be displayed in a text-based browser, such as Lynx. In this case, the different elements will be displayed differently, but you can still tell which text is a heading, which is a list, and so on.

### By the Way

Text-based browsers aren't the only alternative way of displaying HTML. Browsers designed for the blind can read a web page using a speech synthesizer, with different voices or sounds that indicate the different elements.

As you should now understand, HTML is very good at its job—defining a document's structure. Not surprisingly, using this language to try to control the document's *presentation* will only drive you crazy.

Fortunately, the World Wide Web Consortium (W3C) realized that web authors need to control the layout and presentation of documents. This resulted in the *Cascading Style Sheets (CSS)* standard.

CSS adds a number of features to standard HTML to control style and appearance. More importantly, it does this without affecting HTML's capability to describe document structures. Although style sheets still won't make your document look 100% identical on all browsers and all platforms, it is certainly a step in the right direction.

Let's look at a real-world example. If you're browsing the Web with a CSS-supported browser and come across a page that uses CSS, you'll see the document exactly as it was intended. You can also turn off your browser's support for style sheets if you'd rather view all the pages in the same consistent way.

### Did you Know?

Using CSS and simplifying HTML markup is also helpful in making pages compatible with the various tiny browsers used on mobile phones.

## Defining and Using CSS Styles

You can define a CSS style sheet within an HTML document using the `<style>` tag. The opening `<style>` tag specifies the type of style sheet—CSS is currently the only valid type—and begins a list of styles to apply to the document. The `</style>` tag ends the style sheet. Here's a simple example:





**By the  
Way**

If you make a rule that sets the style of the `<body>` tag, it will affect the entire document. This becomes the default rule for the document, but you can override it with the styles of elements within the body of the page.

## Setting Styles for Specific Elements

Rather than setting the style for all elements of a certain type, you can specify a style for an individual element only. For example, the following HTML tag represents a Level 1 heading colored red:

```
<h1 style="color: red; text-align: center;">This is a red heading.</h1>
```

This is called an *inline style* because it's specified in the HTML tag itself. You don't need to use `<style>` tags with this type of style. If you have used both, inline style rules override rules in a style sheet—for example, if the preceding tag appeared in a document that sets H1 headings to be blue in a style sheet, the heading would still be red.

## Using id Attributes

You can also create a rule within a style sheet that will only apply to a certain element. The `id` attribute of an HTML tag enables you to assign a unique identifier to that element. For example, this tag defines a paragraph with the `id` attribute `intro`:

```
<p id="intro">This is a paragraph</p>
```

After you've assigned this attribute to the tag, you can include rules for it as part of a style sheet. CSS uses the pound sign (`#`) to indicate that a rule applies to a specific `id`. For example, the following style sheet sets the `intro` paragraph to be red in color:

```
<style type="text/css">
 #intro {color: red;}
</style>
```

**Watch  
Out!**

An `id` value should define a single element in a page. Most browsers will enable you to define more than one element with the same `id` value, but this is not valid and will not work consistently. It's best to use classes, as described in the next section, when you need to apply the same styles to multiple elements.

## Using Classes

Although the `id` attribute is useful, you can only use each unique `id` value with a single HTML tag. If you need to apply the same style to several tags, you can use the `class` attribute instead. For example, this HTML tag defines a paragraph in a class called `smallprint`:

```
<p class="smallprint">This is the small print</p>
```

To refer to a class within a style sheet, you use a period followed by the class name. Here is a style sheet that defines styles for the `smallprint` class:

```
<style type="text/css">
 .smallprint {color: black;
 font-size: 10px; }
</style>
```

You can use a class on any number of elements within a page. You can also define multiple classes for an element, separated by spaces: `class="smallprint bold"`. When you do this, the CSS rules for all of the classes will be applied to the element.

***By the  
Way***

## Using CSS Properties

CSS supports a wide variety of properties, such as `color` and `text-align`, in the previous example. The following sections list some of the most useful CSS properties for aligning text, changing colors, working with fonts, and setting margins and borders.

This is only an introduction to CSS, and there are many properties beyond those listed here. For more details about CSS, consult one of the web resources or books listed in Appendix A, “Other JavaScript Resources.”

***Did you  
Know?***

## Aligning Text

One of the most useful features of style sheets is the capability to change the spacing and alignment of text. Most of these features aren’t available using standard HTML. You can use the following properties to change the alignment and spacing of text:

- ▶ **letter-spacing**—Specifies the spacing between letters.
- ▶ **text-decoration**—Enables you to create lines over, under, or through the text, or to choose blinking text. The value can be `none` (default), `underline`, `overline`, `line-through`, or `blink`. Blinking text is, thankfully, unsupported by most browsers.
- ▶ **vertical-align**—Enables you to move the element up or down to align with other elements on the same line. The value can be `baseline`, `sub`, `super`, `top`, `text-top`, `middle`, `text-bottom`, and `bottom`.

- ▶ **text-align**—Specifies the justification of text. This can be left, right, center, or justify.
- ▶ **text-transform**—Changes the capitalization of text. `capitalize` makes the first letter of each word uppercase; `uppercase` makes *all* letters uppercase; and `lowercase` makes all letters lowercase.
- ▶ **text-indent**—Enables you to specify the amount of indentation for paragraphs and other elements.
- ▶ **line-height**—Enables you to specify the distance between the top of one line of text and the top of the next.

## Changing Colors and Background Images

You can also use style sheets to gain more control over the colors and background images used on your web page. CSS includes the following properties for this purpose:

- ▶ **color**—Specifies the color of the text within an element. This is useful for emphasizing text or for using a specific color scheme for the document. You can specify a named color (for example, `red`) or red, green, and blue values to define a specific color (for example, `#0522A5`).
- ▶ **background-color**—Specifies the background color of an element. By setting this value, you can make paragraphs, table cells, and other elements with unique background colors. As with `color`, you can specify a color name or numeric color.
- ▶ **background-image**—Specifies the URL for an image to be used as the background for the element. This is specified with the keyword `url` and a URL in parentheses, as in `url(/back.gif)`.
- ▶ **background-repeat**—Specifies whether the background image is repeated (tiled). The image can be repeated horizontally, vertically, or both.
- ▶ **background-attachment**—Controls whether the background image scrolls when you scroll through the document. `fixed` means that the background image stays still while the document scrolls; `scroll` means the image scrolls with the document (like background images on normal web documents).
- ▶ **background-position**—Enables you to offset the position of the background image.
- ▶ **background**—Provides a quick way to set all of the background elements in this list. You can specify all of the attributes in a single background rule.

The basic list of colors supported by most browsers for the color and background-color properties includes aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow.

***Did you  
Know?***

## Working with Fonts

Style sheets also enable you to control the fonts used on the web document and how they are displayed. You can use the following properties to control fonts:

- ▶ **font-family**—Specifies the name of a font, such as arial or helvetica, to use with the element. Because not all users have the same fonts installed, you can list several fonts. The CSS specification also supports several generic font families that are guaranteed to be available: serif, sans-serif, cursive, fantasy, and monospace.
- ▶ **font-style**—Specifies the style of a font, such as normal, italic, or oblique.
- ▶ **font-variant**—This value is normal for normal text, and small-caps to display lowercase letters as small capitals.
- ▶ **font-weight**—Enables you to specify the weight of text: normal or bold. You can also specify a numeric font weight for a specific amount of boldness.
- ▶ **font-size**—The point size of the font.
- ▶ **font**—This is a quick way to set all the font properties in this list. You can list all the values in a single font rule.

## Margins and Borders

Last but not least, you can use style sheets to control the general layout of the page. The following properties affect margins, borders, and the width and height of elements on the web page:

- ▶ **margin-top**, **margin-bottom**, **margin-left**, **margin-right**—Specify the margins of the element. You can specify the margins as an exact number or as a percentage of the page's width.
- ▶ **margin**—Allows you to specify a single value for all four of the margins.
- ▶ **width**—Specifies the width of an element, such as an image.
- ▶ **height**—Specifies the height of an element.
- ▶ **float**—Enables the text to flow around an element. This is particularly useful with images or tables.
- ▶ **clear**—Specifies that the text should stop flowing around a floating image.

***Did you know?***

Along with these features, CSS style sheets enable you to create sections of the document that can be positioned independently. This feature is described in Hour 13, “Using the W3C DOM.”

## Units for Style Sheets

Style sheet properties support a wide variety of *units*, or types of values you can specify. Most properties that accept a numeric value support the following types of units:

- ▶ px—Pixels (for example, 15px). Pixels are the smallest addressable units on a computer screen or other device. In some devices with non-typical resolutions (for example, handheld computers) the browser might rescale this value to fit the device.
- ▶ pt—Points (for example, 10pt). Points are a standard unit for font size. The size of text of a specified point size varies depending on the monitor resolution. Points are equal to 1/72 of an inch.
- ▶ ex— Approximate height of the letter x in the current font (for example, 1.2ex).
- ▶ em—Approximate width of the letter m in the current font (for example, 1.5em). This is usually equal to the font-size property for the current element.
- ▶ %—Percentage of the containing object’s value (for example, 150%).

Which unit you choose to use is generally a matter of convenience. Point sizes are commonly used for fonts, pixel units for the size and position of layers or other objects, and so on.

## Creating a Simple Style Sheet

As an example of CSS, you can now create a web page that uses a wide variety of styles:

- ▶ For the entire body, the text is blue.
- ▶ Paragraphs are centered and have a wide margin on either side.
- ▶ Level 1, 2, and 3 headings are red.
- ▶ Bullet lists are boldface and green by default.

The following is the CSS style sheet to define these properties, using the <style> tags:

```
<style type="text/css">
BODY {color: blue}
P {text-align: center;
 margin-left:20%;
 margin-right:20%}
H1, H2, H3 {color: red}
UL {color: green;
 font-weight: bold}
</style>
```

Here's a rundown of how this style sheet works:

- ▶ The <style> tags enclose the style sheet.
- ▶ The BODY section sets the page body's default text color to blue.
- ▶ The P section defines the style for paragraphs.
- ▶ The H1, H2, H3 section defines the style for heading tags.
- ▶ The UL section defines a style for bullet lists.

To show how this style sheet works, Listing 12.1 shows a document that includes this style sheet and a few examples of overriding styles for particular elements. Figure 12.1 shows Internet Explorer's display of this example.

---

**LISTING 12.1** An Example of a Document Using CSS Style Sheets

---

```
<html>
<head><title>Style Sheet Example</title>
<style type="text/css">
BODY {color: blue}
P {text-align: center;
 margin-left:20%;
 margin-right:20%}
H1, H2, H3 {color: red}
UL {color: green;
 font-weight: bold}
</style>
</head>
<body>
<h1>Welcome to this page</h1>
<p>The above heading is red, since we specified that H1-H3 headings
are red. This paragraph is blue, which is the default color for
the entire body. It's also centered and has 20% margins, which we
specified as the default for paragraphs.
</p>
<p style="color:black">This paragraph has black text, because it overrides
the default color in the paragraph tag. We didn't override the centering,
so this paragraph is also centered.</p>

```

**LISTING 12.1 Continued**

```
This is a bullet list.
It's green and bold, because we specified those defaults for bullet lists.
<li style="color:red">This item is red, overriding the default.
This item is back to normal.

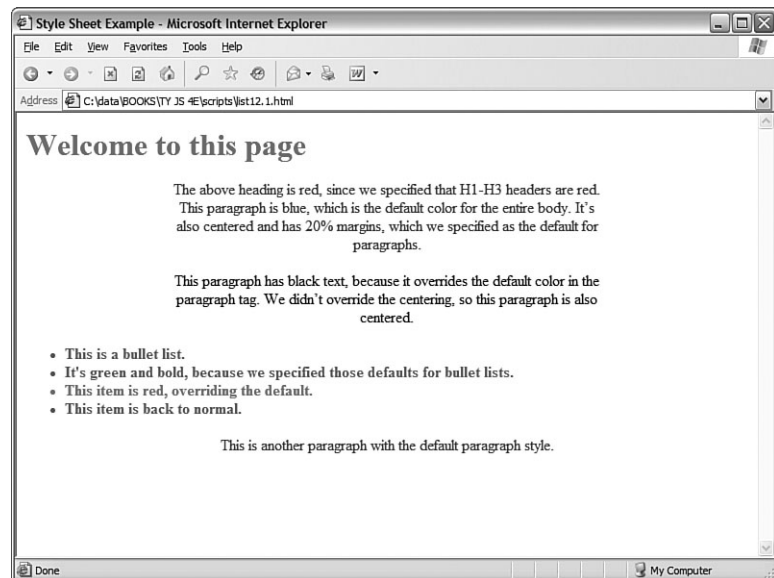
<p>This is another paragraph with the default paragraph style.</p>
</body>
</html>
```

***Did you know?***

Remember that you can download the code for this listing from this book's web-site.

**FIGURE 12.1**

The style sheet example as displayed by Internet Explorer.



## Using External Style Sheets

The preceding example only changes a few aspects of the HTML document's appearance, but it adds about 10 lines to its length. If you were trying to make a very stylish page and had defined new styles for all of the attributes, you would end up with a very long and complicated document.

For this reason, you can use a CSS style sheet from a separate file in your document. This makes your document short and to the point. More importantly, it enables you to define a single style sheet and use it to control the appearance of all of the pages on your site.



## Linking to External Style Sheets

You can refer to an external CSS file by using the `<link>` tag in the `<head>` section of one or more HTML documents:

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

This tag refers to an external CSS style sheet stored in the `style.css` file.

Using external style sheets is a good practice because it separates content (HTML), presentation (CSS), and behavior (JavaScript). See Hour 15, “Unobtrusive Scripting,” for more information on best practices.

**By the  
Way**

## Creating External .css Files

After you’ve linked to an external `.css` file, you need to create the file itself. The external style sheet is a simple text file that you can create with the same editor you use for HTML documents.

The `.css` file should contain a list of CSS rules, in the same format you would use between `<style>` tags. However, the file should not include `<style>` tags or any other HTML tags. Here is what the styles from the previous example would look like as an external style sheet:

```
BODY {color: blue}
P {text-align: center;
 margin-left:20%;
 margin-right:20%}
H1, H2, H3 {color: red}
UL {color: green;
 font-weight: bold}
```

## Controlling Styles with JavaScript

The new W3C DOM (Document Object Model) makes it easy for JavaScript applications to control the styles on a page. Whether or not you use style sheets, you can use JavaScript to modify the style of any element on a page.

As you learned in Hour 4, “Working with the Document Object Model (DOM),” the DOM enables you to access the entire HTML document and all of its elements as scriptable objects. You can change any object’s style by modifying its `style` object properties.

The names and values of objects under the `style` object are the same as you’ve learned in this hour. For example, you can change an element’s color by modifying its `style.color` attribute:

```
element.style.color="blue";
```

Here, `element` represents the object for an element. There are many ways of finding an element's corresponding object, which you will learn about in detail in Hour 13.

In the meantime, an easy way to find an element's object is to assign an identifier to it with the `id` attribute. The following statement creates an `<h1>` element with the identifier "head1":

```
<h1 id = "head1">This is a heading</h1>
```

Now that you've assigned an identifier, you can use the `getElementById()` method to find the DOM object for the element:

```
element = document.getElementById("head1");
```

You can also use a shortcut to set styles and avoid the use of a variable by directly working with the `getElementById()` method:

```
document.getElementById("head1").style.color="blue";
```

This statement combines the preceding examples by directly assigning the blue color style to the `head1` element of the page. You'll use this technique to create a dynamic page in the following Try It Yourself section.



## Try It Yourself

### Creating Dynamic Styles

Using the DOM style objects, you can create a page that enables you to directly control the colors used in the page's text. To begin with, you will need a form with which to select colors. You can use `<select>` tags to allow a color choice:

```
<select name="heading" onChange="changehead();">
 <option value="black">Black</option>
 <option value="red">Red</option>
 <option value="blue">Blue</option>
 <option value="green">Green</option>
 <option value="yellow">Yellow</option>
</select>
```

### By the Way

If you are unsure of the syntax used in forms, you might want to review Hour 11, "Getting Data with Forms."

Notice that this `<select>` definition uses `onChange` attributes in the `<select>` tags to call two functions, `changehead()` and `changebody()`, when their respective selection changes.

Combining two of these selections with some basic HTML results in the complete HTML document shown in Listing 12.2.

---

**LISTING 12.2** The HTML File for the Dynamic Styles Example

---

```
<html>
<head>
<title>Controlling Styles with JavaScript</title>
<script language="Javascript" type="text/javascript"
 src="styles.js">
</script>
</head>
<body>
<h1 id="head1">
Controlling Styles with JavaScript</h1>
<hr>
<p id="p1">
Select the color for paragraphs and headings using the form below.
The colors you specified will be dynamically changed in this document.
The change occurs as soon as you change the value of either of the
drop-down lists in the form.
</p>
<form name="form1">
Heading color:
<select name="heading" onChange="changehead();">
 <option value="black">Black</option>
 <option value="red">Red</option>
 <option value="blue">Blue</option>
 <option value="green">Green</option>
 <option value="yellow">Yellow</option>
</select>

Body text color:
<select name="body" onChange="changebody();">
 <option value="black">Black</option>
 <option value="red">Red</option>
 <option value="blue">Blue</option>
 <option value="green">Green</option>
 <option value="yellow">Yellow</option>
</select>
</form>
</body>
</html>
```

---

Notice that the `<h1>` tag has an `id` attribute of "head1", and the `<p>` tag has an `id` of "p1". These are the values the script will use in the `getElementById()` function. The `<script>` tag in the `<head>` section links the document to the `styles.js` script, which you will create next.

Save your HTML file as `styles.html`. You can test it in a browser now, but the dynamic features will not work until you create the JavaScript file containing the script functions. Listing 12.3 shows the JavaScript code for this example.

**LISTING 12.3** The JavaScript File for the Dynamic Styles Example

```
function changehead() {
 i = document.form1.heading.selectedIndex;
 headcolor = document.form1.heading.options[i].value;
 document.getElementById("head1").style.color = headcolor;
}
function changebody() {
 i = document.form1.body.selectedIndex;
 doccolor = document.form1.body.options[i].value;
 document.getElementById("p1").style.color = doccolor;
}
```

This script first defines the `changehead()` function. This reads the index for the currently selected heading color, and then reads the color value for the index. This function uses the `getElementById()` method described in the previous section to change the color. The `changebody()` function uses the same syntax to change the body color.

Store your JavaScript file as `styles.js`, and be sure it is in the same folder as the HTML document you saved from Listing 12.2.

To test the dynamic styles script, load Listing 12.2 (`styles.html`) into the browser. Select the colors, and notice the immediate change in the heading or body of the page. Figure 12.2 shows a typical display of this document after the colors have been changed.

**FIGURE 12.2**  
The dynamic  
styles example  
in action.



## Summary

In this hour, you've used style sheets to control the appearance of web documents. You've learned the CSS syntax for creating style sheets, and used JavaScript to control the styles of a document.

In the next hour, you will move on to Dynamic HTML (DHTML) using layers and other features of the W3C DOM.

## Q&A

- Q. *What's the difference between changing the appearance of text with traditional tags, such as `<b>` and `<i>`, and using a style sheet?***
- A.** Functionally, there is no difference. In principle, though, the HTML should define the structure of the document, and CSS should define the presentation.
- Q. *What happens if two style sheets affect the same text?***
- A.** The CSS specification is designed to allow style sheets to overlap, or cascade. Thus, you can specify a style for the body of the document and override it for specific elements, such as headings and paragraphs. You can even go one step further and override the style for one particular instance of an element. CSS has a set of rules governing which styles have precedence over others, although you might find that different browsers interpret CSS differently when you have many overlapping styles.
- Q. *With CSS in one separate file and JavaScript in another, doesn't web development get confusing?***
- A.** Yes, this can make a simple page unnecessarily complex. However, as you build more complex pages, you'll find it very helpful to have three separate files. This lets you deal with the content and structure (HTML), presentation (CSS), and behavior (JavaScript) separately.
- Q. *What if users don't like the styles I use in my pages?***
- A.** This is another distinct advantage style sheets have over browser-specific tags. With the latest browsers, users can choose a default style sheet of their own and override any properties they want.

## Quiz Questions

Test your knowledge of style sheets and JavaScript by answering the following questions.

- 1.** Which of the following tags is the correct way to begin a CSS style sheet?
  - a.** `<style>`
  - b.** `<style type="text/css">`
  - c.** `<style rel="css">`

2. Why isn't the normal HTML language very good at defining layout and presentation?
  - a. Because it was designed by programmers.
  - b. Because magazines feared the competition.
  - c. Because its main purpose is to define document structure.
3. Which feature of new browsers allows you to use JavaScript statements to change styles?
  - a. HTML 4.0
  - b. The DOM
  - c. CSS 2.0

## Quiz Answers

1. b. You begin a CSS style sheet with the tag `<style type="text/css">`.
2. c. HTML is primarily intended to describe the structure of documents.
3. b. The DOM (Document Object Model) enables you to change styles using JavaScript.

## Exercises

If you want to gain more experience using CSS style sheets, try the following exercise:

- ▶ Modify Listing 12.2 to include an `<h2>` tag with a subheading. Add a form element to select this tag's color, and a corresponding `changeh2` function in the script.
- ▶ Now that Listing 12.2 has three different changeable elements, there is quite a bit of repetition in the script. Create a single `ChangeColor` function that takes a parameter for the element to change, and modify the `onChange` event handlers to send the appropriate element `id` value as a parameter to this function.

## Hour 13

# Using the W3C DOM

---

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

- ▶ How the W3C DOM standard makes dynamic pages easier
- ▶ How the DOM's objects are structured
- ▶ Understanding nodes, parents, children, and siblings
- ▶ Creating positionable layers
- ▶ Using CSS's positioning properties
- ▶ Controlling positioning with JavaScript

Throughout this book, you've learned about the DOM (Document Object Model), JavaScript's way of referencing objects within web documents. In the last hour, you learned to modify style sheet properties on the fly using JavaScript.

During this hour, you'll learn more about how the DOM represents the objects that make up a web document, and how to use DOM objects to move objects within a page.

## **The DOM and Dynamic HTML**

Due to the basic DOM of older browsers, JavaScript could only have a limited effect on a page. There were certain elements, such as forms and images, that you could control with JavaScript, but if you wanted to do something more complex—such as adding or removing several paragraphs, making a form appear out of nowhere, or displaying data dynamically within text—you were out of luck.

To escape this limitation, browser manufacturers created *Dynamic HTML*, or DHTML—an extended DOM that allowed JavaScript to manipulate more of a page. Unfortunately, this was still limiting—you had to work with certain defined parts of the page called *layers* rather than having complete control over the page.

Worse yet, Microsoft and Netscape created completely different and incompatible versions of DHTML, which led to some complicated and unreliable scripting.

Fortunately, you won't have to learn about those incompatible versions of DHTML because the W3C DOM has made them unnecessary. Although browsers still aren't perfectly interchangeable, today's browsers support enough of the standard DOM to enable you to fully control the content of pages without much concern over browser issues. In this hour and the next hour, you'll create several examples of DOM scripts that will work fine in all modern browsers.

### By the Way

There are still browser issues, of course. Hour 15, "Unobtrusive Scripting," will show you how to deal with browser differences and how to minimize your chances of running into problems with new browsers.

## Understanding DOM Structure

In Hour 4, "Working with the Document Object Model (DOM)," you learned about how some of the most important DOM objects are organized: The window object contains the document object, and so on. Although these objects were the only ones available in older browsers, the new DOM adds objects under the document object for every element of a page.

To better understand this concept, let's look at the simple HTML document in Listing 13.1. This document has the usual <head> and <body> sections, a heading, and a single paragraph of text.

### LISTING 13.1 A Simple HTML Document

```
<html>
<head>
<title>A simple HTML Document</title>
</head>
<body>
<h1>This is a Heading</h1>
<p>This is a paragraph</p>
</body>
</html>
```

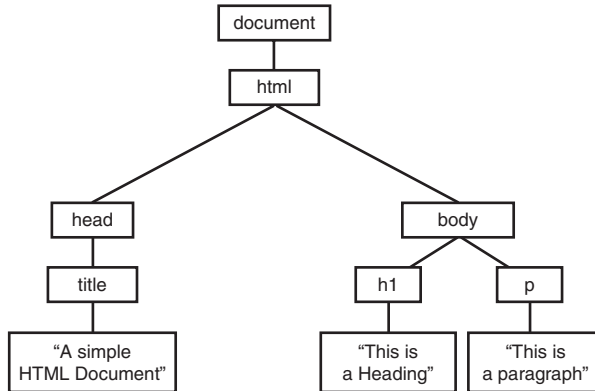
Like all HTML documents, this one is composed of various *containers* and their contents. The <html> tags form a container that includes the entire document, the <body> tags contain the body of the page, and so on.

In the DOM, each container within the page and its contents are represented by an object. The objects are organized into a tree-like structure, with the document object



itself at the root of the tree, and individual elements such as the heading and paragraph of text at the leaves of the tree. Figure 13.1 shows a diagram of these relationships.

In the following sections, you will examine the structure of the DOM more closely.



**FIGURE 13.1**  
How the DOM represents an HTML document.

Don't worry if this tree structure confuses you; you can do almost anything by simply assigning IDs to elements and referring to them. This is the method used in earlier hours of this book, as well as in the Try It Yourself section of this hour. In Hour 14, "Using Advanced DOM Features," you will look at more complicated examples that require you to understand the way objects are organized in the DOM.

***By the Way***

## Nodes

Each container or element in the document is called a *node* in the DOM. In the example in Figure 13.1, each of the objects in boxes is a node, and the lines represent the relationships between the nodes.

You will often need to refer to individual nodes in scripts. You can do this by assigning an ID, or by navigating the tree using the relationships between the nodes.

## Parents and Children

As you learned earlier in this book, each JavaScript object can have a *parent*—an object that contains it—and can also have *children*—objects that it contains. The DOM uses the same terminology.

In Figure 13.1, the document object is the parent object for the remaining objects, and does not have a parent itself. The html object is the parent of the head and body objects, and the h1 and p objects are children of the body object.

Text nodes work a bit differently. The actual text in the paragraph is a node in itself, and is a child of the `p` object. Similarly, the text within the `<h1>` tags is a child of the `h1` object.

### By the Way

In Hour 14, you will learn methods of referring to objects by their parent and child relationships, as well as ways of adding and removing nodes from the document.

## Siblings

The DOM also uses another term for organization of objects: *siblings*. As you might expect, this refers to objects that have the same parent—in other words, objects at the same level in the DOM object tree.

In Figure 13.1, the `h1` and `p` objects are siblings: Both are children of the `body` object. Similarly, the `head` and `body` objects are siblings under the `html` object.

## Creating Positionable Elements (Layers)

Using the W3C DOM, you can control any element in a web page, such as a paragraph or an image. You can change an element's style, as you learned in the previous hour. You can also use the DOM to change the position, visibility, and other attributes of the element.

Before the W3C DOM and CSS2 standards, you could only reposition *layers*, special groups of elements defined with a proprietary tag. Although you can now position any element, it's still useful to work with groups of elements in many cases.

You can effectively create a layer, or a group of HTML objects that can be controlled as a group, using the `<div>` or `<span>` tags.

### By the Way

The `<div>` and `<span>` tags are part of the HTML 3.0 standard. `<span>` defines an arbitrary section of the HTML document, and does not specify any formatting for the text it contains. `<div>` is similar, but includes a line break before and after its contents.

To create a layer with `<div>`, enclose the content of the layer between the two division tags and specify the layer's properties in the `style` attribute of the `<div>` tag. Here's a simple example:

```
<div id="layer1" style="position:absolute; left:100; top:100">
<p>This is the content of the layer.</p>
</div>
```

This code defines a layer with the name `layer1`. This is a moveable layer positioned 100 pixels down and 100 pixels to the right of the upper-left corner of the browser window. You'll learn more details about the positioning properties in the next section.

As with all CSS properties, you can specify the `position` property and other layer properties in a `<style>` block, in an external style sheet, or in the `style` attribute of an HTML tag. You can also control these properties using JavaScript, as described later in this hour.

***Did you  
Know?***

## Setting Object Position and Size

You can use various properties in the `style` attribute of the `<div>` tag when you define a layer to set its position, visibility, and other features. The following properties control the object's position and size:

- ▶ `position` is the main positioning attribute and can affect the following properties. The `position` property can have one of three values:
  - ▶ `static` defines items that are laid out in normal HTML fashion, and cannot be moved. This is the default.
  - ▶ `absolute` specifies that an item will be positioned using coordinates you specify.
  - ▶ `relative` defines an item that is offset a certain amount from the static position, where the element would normally have been laid out within the HTML page.
- ▶ `left` and `top` specify offsets for the position of the item. For absolute positioning, this is relative to the main browser window or a containing item. For relative positioning, it's relative to the usual static position.
- ▶ `right` and `bottom` are an alternative way to specify the position of the item. You can use these when you need to align the object's right or bottom edge.
- ▶ `width` and `height` are similar to the standard HTML width and height attributes and specify a width and height for the item.
- ▶ `z-index` specifies how items overlap. Normally indexes start with 1 and go up with each layer added "on top" of the page. By changing this value, you can specify which item is on top.

**By the  
Way**

Properties such as `left` and `top` work in pixels by default. You can also use any of the units described in the previous hour: `px`, `pt`, `ex`, `em`, or percentages.

## Setting Overflow Properties

Sometimes the content inside a layer is larger than the size the layer can display. Two properties affect how the layer is displayed in this case:

- ▶ `overflow` indicates whether the content of an element is cut off at the edges of the element, or whether a scroll bar allows viewing the rest of the item. Values include `visible` to display content outside the element; `hidden` to hide the clipped content; `scroll` to display scroll bars; `auto` to let the browser decide whether to display scroll bars; or `inherit` to use a parent object's setting.
- ▶ `clip` specifies the clipping rectangle for an item. Only the portion of the item inside this rectangle is displayed. Normally this is the same as the element's dimensions, but you can define an offset inside the element here.

## Using Visibility Properties

Along with positioning objects, you can use CSS positioning to control whether the objects are visible at all, and how the document is formatted around them. These properties control how objects are displayed:

- ▶ `display` specifies whether an item is displayed in the browser. A value of `none` hides the object. Other values include `block` to display the object preceded and followed by line breaks, `inline` to display it without line breaks, and `list-item` to display it as part of a list.
- ▶ `visibility` specifies whether an item is visible. Values include `visible` (default), `hidden`, and `inherit`. A value of `inherit` means the item inherits the visibility of any item it appears within (such as a table or a paragraph).

The difference between `display` and `visibility` is that objects set to `display: none` will not be displayed at all, and the page will be laid out as if the element wasn't there. Objects set to `visibility: hidden` will still be included in the layout of the page, but as empty space.

## Setting Background and Border Properties

You can use the following properties to set the color and background image for a layer or other object and control whether borders are displayed:

- ▶ `background-color` specifies the color for the background of any text in the layer.
- ▶ `background-image` specifies a background image for any text in the layer.
- ▶ `border-width` sets the width of the border for all four sides. This can be a numeric value or the keywords `thin`, `medium`, or `thick`.
- ▶ `border-style` sets the style of border. Values include `none` (default), `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, or `outset`.
- ▶ `border-color` sets the color of the border. As with other color properties, this can be a named color such as `blue` or an RGB color such as `#FF03A5`.

## Controlling Positioning with JavaScript

As you learned in the previous hour, you can control the style attributes for an object with the attributes of the object's `style` property. You can control the positioning attributes listed in the previous section the same way.

Suppose you have created a layer with the following `<div>` tags:

```
<div id="layer1" style="position:absolute; left:100; top:100">
<p>This is the content of the layer.</p>
</div>
```

To move this layer up or down within the page using JavaScript, you can change its `style.top` attribute. For example, the following statements move the layer 100 pixels down from its original position:

```
var obj = document.getElementById("layer1");
obj.style.top=200;
```

The `document.getElementById()` method returns the object corresponding to the layer's `<div>` tag, and the second statement sets the object's `top` positioning property to 200. As you learned in the previous hour, you can also combine these two statements:

```
document.getElementById("layer1").style.top = 200;
```

This simply sets the `style.top` property for the layer without assigning a variable to the layer's object. You will use this technique in this hour's Try It Yourself section.

Some CSS properties, such as `text-indent` and `border-color`, have hyphens in their names. When you use these properties in JavaScript, you combine the hyphenated sections and use a capital letter: `textIndent` and `borderColor`.

**By the  
Way**



## Try It Yourself

### Creating a Movable Layer

As an example of positioning an element with JavaScript, you can now create an HTML document that defines a layer, and combine it with a script to allow the layer to be moved, hidden, or shown using buttons. Listing 13.2 shows the HTML document that defines the buttons and the layer.

#### LISTING 13.2 The HTML Document for the Movable Layer Example

```
<html>
<head>
<title>Positioning Elements with JavaScript</title>
<script language="javascript" type="text/javascript"
 src="position.js">
</script>
<style>
#square {
 position:absolute;
 top: 150;
 left: 100;
 width: 200;
 height: 200;
 border: 2px solid black;
 padding: 10px;
 background-color: #E0E0E0;
}
</style>
</head>
<body>
<h1>Positioning Elements</h1>
<hr>
<form name="form1">
<input type="button" name="left" value="< Left"
 onClick="pos(-1,0);">
<input type="button" name="right" value="Right >"
 onClick="pos(1,0);">
<input type="button" name="up" value="Up"
 onClick="pos(0,-1);">
<input type="button" name="down" value="Down"
 onClick="pos(0,1);">
<input type="button" name="hide" value="Hide"
 onClick="hideSquare();">
<input type="button" name="show" value="Show"
 onClick="showSquare();">
</form>
<hr>
<div id="square">
<p>This square is an absolutely positioned
layer that you can move using the buttons above.</p>
</div>
</body>
</html>
```

In addition to some basic HTML, this document consists of the following:

- ▶ The `<script>` tag in the header reads a script called `position.js`, which you will create later in this section.
- ▶ The `<style>` section is a brief style sheet that defines the properties for the movable layer. It sets the `position` property to `absolute` to indicate that it can be positioned at an exact location, sets the initial position in the `top` and `left` properties, and sets `border` and `background-color` properties to make the layer clearly visible.
- ▶ The `<input>` tags within the `<form>` section define six buttons: four to move the layer left, right, up, or down, and two to control whether it is visible or hidden.
- ▶ The `<div>` section defines the layer itself. The `id` attribute is set to the value `"square"`. This `id` is used in the style sheet to refer to the layer, and will also be used in your script.

Type this document (or download it from this book's website) and save it. If you load it into a browser, you should see the buttons and the `"square"` layer, but the buttons won't do anything yet. The script in Listing 13.3 adds the action to the HTML.

---

**LISTING 13.3** The Script for the Movable Layer Example

---

```
var x=100,y=150;
function pos(dx,dy) {
 if (!document.getElementById) return;
 x += 10*dx;
 y += 10*dy;
 obj = document.getElementById("square");
 obj.style.top=y;
 obj.style.left=x;
}
function hideSquare() {
 if (!document.getElementById) return;
 obj = document.getElementById("square");
 obj.style.display="none";
}
function showSquare() {
 if (!document.getElementById) return;
 obj = document.getElementById("square");
 obj.style.display="block";
}
```

---

The `var` statement at the beginning of the script defines two variables, `x` and `y`, that will store the current position of the layer. The `pos` function is called by the event handlers for all four of the movement buttons.

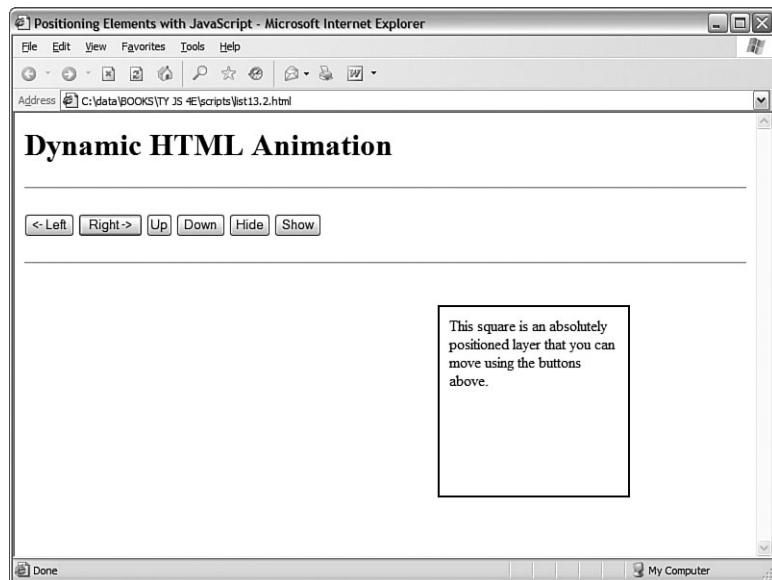
The parameters of the `pos()` function, `dx` and `dy`, tell the script how the layer should move: If `dx` is negative, a number will be subtracted from `x`, moving the layer to the left. If `dx` is positive, a number will be added to `x`, moving the layer to the right. Similarly, `dy` indicates whether to move up or down.

The `pos()` function begins by making sure the `getElementById()` function is supported, so it won't attempt to run in older browsers. It then multiplies `dx` and `dy` by 10 (to make the movement more obvious) and applies them to `x` and `y`. Finally, it sets the `top` and `left` properties to the new position, moving the layer.

Two more functions, `hideSquare()` and `showsquare()`, hide or show the layer by setting its `display` property to "none" (hidden) or "block" (shown).

To use this script, save it as `position.js` and then load the HTML document, Listing 13.2, into your browser. Figure 13.2 shows this script in action.

**FIGURE 13.2**  
The movable  
layer example in  
Internet  
Explorer.



**By the  
Way**



By assigning values to the layer's positioning properties repeatedly rather than at each click of a button, you can produce an animation effect. See Hour 19, "Using Graphics and Animation," for an example of this technique.



## Summary

In this hour, you've learned a bit more about the structure of DOM objects that make up a page, how to use HTML and CSS to define a positionable layer, and how you can use positioning properties dynamically with JavaScript.

Layers are only a simple aspect of what you can do to a page with the W3C DOM. In the next hour, you'll learn how to manipulate the DOM tree to add elements, remove elements, and dynamically change the text within a page.

## Q&A

**Q.** *What happens when my web page includes multiple HTML documents, such as when frames are used?*

**A.** In this case, each window or frame has its own document object that stores the elements of the HTML document it contains.

**Q.** *If the DOM allows any object to be dynamically changed, why does the positioning example need to use <div> tags to define a layer?*

**A.** The example could just as easily move a heading, or a paragraph. The layer is just a convenient way to group objects and to create a square object with a border.

**Q.** *Exactly which browsers support positioning elements with the DOM?*

**A.** Support for the W3C DOM first appeared in Internet Explorer 5.0 and Netscape 5.0, although it was buggy. Current browsers, such as Internet Explorer 6 and 7, Firefox 1.x, and Opera 7 and 8, have solid and consistent DOM support.

## Quiz Questions

Test your knowledge of the W3C DOM by answering the following questions.

**1.** Which of the following tags is used to create a layer?

- a.** <layer>
- b.** <div>
- c.** <style>

2. Which property controls an element's left-to-right position?
  - a. `left`
  - b. `width`
  - c. `lrpos`
3. Which of the following CSS rules would create a heading that is not currently visible in the page?
  - a. `h1 {visibility: invisible;}`
  - b. `h1 {display: none;}`
  - c. `h1 {style: invisible;}`

## Quiz Answers

1. b. The `<div>` tag can be used to create positionable layers.
2. a. The `left` property controls an element's left-to-right position.
3. b. The `none` value for the `display` property makes it invisible. The `visibility` property could also be used, but its possible values are `visible` or `hidden`.

## Exercises

If you want to gain more experience using the W3C DOM, try the following exercises:

- ▶ Modify the positioning example in Listings 13.2 and 13.3 to move the square one pixel at a time rather than ten at a time.
- ▶ Modify the positioning example to eliminate the `<div>` layer and move a paragraph element instead. You will need to move the `id` attribute to the paragraph.

## HOUR 14

# Using Advanced DOM Features

---

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

- ▶ Using the properties of DOM nodes
- ▶ Understanding DOM node methods
- ▶ Hiding and showing objects within a page
- ▶ Modifying text within a page
- ▶ Adding text to a page
- ▶ Creating a dynamic navigation tree

During this hour, you will take a closer look at the objects in the DOM, and the properties and methods you can use to control them. You will also explore several examples of dynamic HTML pages using these DOM features.

## **Working with DOM Nodes**

As you learned in Hour 13, “Using the W3C DOM,” the DOM organizes objects within a web page into a tree-like structure. Each node (object) in this tree can be accessed in JavaScript. In the next sections you will learn how you can use the properties and methods of nodes to manage them.

The following sections only describe the most important properties and methods of nodes, and those that are supported by current browsers. For a complete list of available properties, see the W3C's DOM specification at <http://www.w3.org/TR/DOM-Level-2/>.

***By the  
Way***

---

## Basic Node Properties

You have already used the `style` property of nodes to change their style sheet values. Each node also has a number of basic properties that you can examine or set. These include the following:

- ▶ `nodeName` is the name of the node (not the ID). For nodes based on HTML tags, such as `<p>` or `<body>`, the name is the tag name: `P` or `BODY`. For the document node, the name is a special code: `#document`. Similarly, text nodes have the name `#text`.
- ▶ `nodeType` is an integer describing the node's type: 1 for normal HTML tags, 3 for text nodes, and 9 for the document node.
- ▶ `nodeValue` is the actual text contained within a text node. This property is not valid for other types of nodes.
- ▶ `innerHTML` is the HTML content of any node. You can assign a value including HTML tags to this property and change the DOM child objects for a node dynamically.

### By the Way

The `innerHTML` property is not a part of the W3C DOM specification. However, it is supported by the major browsers, and is often the easiest way to change content in a page. You can also accomplish this in a more standard way by deleting and creating nodes, as described later in this hour.

## Node Relationship Properties

In addition to the basic properties described previously, each node has a number of properties that describe its relation to other nodes. These include the following:

- ▶ `firstChild` is the first child object for a node. For nodes that contain text, such as `h1` or `p`, the text node containing the actual text is the first child.
- ▶ `lastChild` is the node's last child object.
- ▶ `childNodes` is an array that includes all of a node's child nodes. You can use a loop with this array to work with all the nodes under a given node.
- ▶ `previousSibling` is the sibling (node at the same level) previous to the current node.
- ▶ `nextSibling` is the sibling after the current node.

Remember that, like all JavaScript objects and properties, the node properties and functions described here are case sensitive. Be sure you type them exactly as shown.

**Watch  
Out!**

## Document Methods

The document node itself has several methods you might find useful. You have already used one of these, `getElementById()`, to refer to DOM objects by their ID properties. The document node's methods include the following:

- ▶ `getElementById(id)` returns the element with the specified `id` attribute.
- ▶ `getElementsByName(tag)` returns an array of all of the elements with a specified tag name. You can use the wildcard `*` to return an array containing all the nodes in the document.
- ▶ `createTextNode(text)` creates a new text node containing the specified text, which you can then add to the document.
- ▶ `createElement(tag)` creates a new HTML element for the specified tag. As with `createTextNode`, you need to add the element to the document after creating it. You can assign content within the element by changing its child objects or the `innerHTML` property.

## Node Methods

Each node within a page has a number of methods available. Which of these are valid depends on the node's position in the page, and whether it has parent or child nodes. These include the following:

- ▶ `appendChild(new)` appends the specified new node after all of the object's existing nodes.
- ▶ `insertBefore(new, old)` inserts the specified new child node before the specified old child node, which must already exist.
- ▶ `replaceChild(new, old)` replaces the specified old child node with a new node.
- ▶ `removeChild(node)` removes a child node from the object's set of children.
- ▶ `hasChildNodes()` returns a Boolean value of `true` if the object has one or more child nodes, or `false` if it has none.
- ▶ `cloneNode()` creates a copy of an existing node. If a parameter of `true` is supplied, the copy will also include any child nodes of the original node.

## Hiding and Showing Objects

We will now move on to a number of real-world examples using the DOM objects to manipulate web pages. As a simple example, you can create a script that hides or shows objects within a page.

As you learned in Hour 13, objects have a `visibility` style property that specifies whether they are currently visible within the page:

```
Object.style.visibility="hidden"; // hides an object
Object.style.visibility="visible"; // shows an object
```

Using this property, you can create a script that hides or shows objects in either browser. Listing 14.1 shows the HTML document for a script that allows two headings to be shown or hidden.

---

### LISTING 14.1 Hiding and Showing Objects

---

```
<html>
<head>
<title>Hiding and Showing Objects</title>
<script language="Javascript" type="text/javascript">
function ShowHide() {
 if (!document.getElementById) return;
 var head1 = document.getElementById("head1");
 var head2 = document.getElementById("head2");
 var showhead1 = document.form1.head1.checked;
 var showhead2 = document.form1.head2.checked;
 head1.style.visibility=(showhead1) ? "visible" : "hidden";
 head2.style.visibility=(showhead2) ? "visible" : "hidden";
}
</script>
</head>
<body>
<h1 ID="head1">This is the first heading</h1>
<h1 ID="head2">This is the second heading</h1>
<p>Using the W3C DOM, you can choose
whether to show or hide the headings on
this page using the checkboxes below.</p>
<form name="form1">
<input type="checkbox" name="head1"
checked onClick="ShowHide();">
Show first heading

<input type="checkbox" name="head2"
checked onClick="ShowHide();">
Show second heading

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

---

The `<h1>` tags in this document define headings with the identifiers `head1` and `head2`. The `<form>` section defines a form with two check boxes, one for each of the

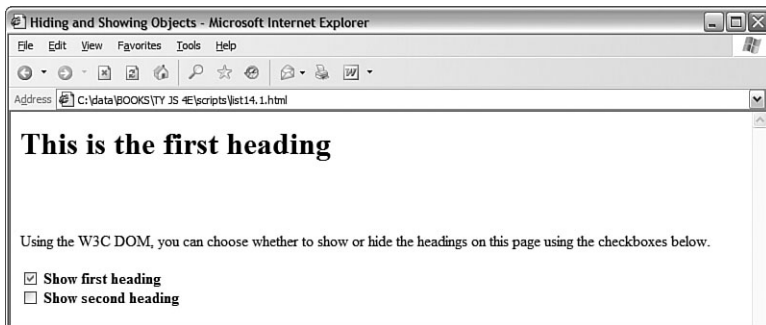
headings. When a check box is modified, the `onClick` method is used to call the `ShowHide()` function.

This function is defined within the `<script>` statements in the header. The function assigns the `head1` and `head2` variables to the objects for the headings, using the `getElementById()` method. Next, it assigns the `showhead1` and `showhead2` variables to the contents of the check boxes. Finally, the function uses the `style.visibility` attributes to set the visibility of the headings.

The lines that set the `visibility` property might look a bit strange. The `?` and `:` characters create *conditional expressions*, a shorthand way of handling `if` statements. To review conditional expressions, see Hour 7, “Controlling Flow with Conditions and Loops.”

**Did you  
Know?**

Figure 14.1 shows this example in action in Internet Explorer. In the figure, the second heading’s check box has been unchecked, so only the first heading is visible.



**FIGURE 14.1**  
The text hiding/showing example in Internet Explorer.

## Modifying Text Within a Page

Next, you can create a simple script to modify the contents of a heading within a web page. As you learned earlier this hour, the `nodeValue` property of a text node contains its actual text, and the text node for a heading is a child of that heading. Thus, the syntax to change the text of a heading with the identifier `head1` would be

```
Var head1 = document.getElementById("head1");
Head1.firstChild.nodeValue = "New Text Here";
```

This assigns the variable `head1` to the heading’s object. The `firstChild` property returns the text node that is the only child of the heading, and its `nodeValue` property contains the heading text.

Using this technique, it’s easy to create a page that allows the heading to be changed dynamically. Listing 14.2 shows the complete HTML document for this script.

**LISTING 14.2** The Complete Text-Modifying Example

```

<html>
<head>
<title>Dynamic Text in JavaScript</title>
<script language="JavaScript" type="text/javascript">
function ChangeTitle() {
 if (!document.getElementById) return;
 var newtitle = document.form1.newtitle.value;
 var head1 = document.getElementById("head1");
 head1.firstChild.nodeValue=newtitle;
}
</script>
</head>
<body>
<h1 ID="head1">Dynamic Text in JavaScript</h1>
<p>Using the W3C DOM, you can dynamically
change the heading at the top of this
page. Enter a new title and click the
Change button.</p>
<form name="form1">
<input type="text" name="newtitle" size="25">
<input type="button" value="Change!"
 onClick="ChangeTitle();">
</form>
</body>
</html>

```

This example defines a form that allows the user to enter a new heading for the page. Pressing the button calls the `ChangeTitle()` function, defined in the header. This function gets the value the user entered in the form, and changes the heading's value to the new text.

Figure 14.2 shows this page in action in Internet Explorer after a new title has been entered and the Change button has been clicked.

**FIGURE 14.2**  
The heading-  
changing exam-  
ple in action.





## Adding Text to a Page

Next, you can create a script that actually adds text to a page. To do this, you must first create a new text node. This statement creates a new text node with the text “this is a test”:

```
var node=document.createTextNode("this is a test");
```

Next, you can add this node to the document. To do this, you use the `appendChild` method. The text can be added to any element that can contain text, but we will use a paragraph. The following statement adds the text node defined previously to the paragraph with the identifier `p1`:

```
document.getElementById("p1").appendChild(node);
```

Listing 14.3 shows the HTML document for a complete example that uses this technique, using a form to allow the user to specify text to add to the page.

---

### LISTING 14.3 Adding Text to a Page

---

```
<html>
<head>
<title>Adding to a page</title>
<script language="Javascript" type="text/javascript">
function AddText() {
 if (!document.getElementById) return;
 var sentence=document.form1.sentence.value;
 var node=document.createTextNode(" " + sentence);
 document.getElementById("p1").appendChild(node);
 document.form1.sentence.value="";
}
</script>
</head>
<body>
<h1>Create Your Own Content</h1>
<p id="p1">Using the W3C DOM, you can dynamically
add sentences to this paragraph. Type a sentence
and click the Add button.</p>
<form name="form1">
<input type="text" name="sentence" size="65">
<input type="button" value="Add" onClick="AddText();">
</form>
</body>
</html>
```

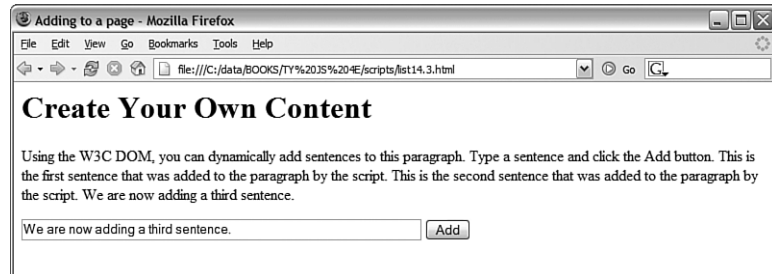
---

In this example, the `<p>` section defines the paragraph that will hold the added text. The `<form>` section defines a form with a text field called `sentence`, and an `Add` button, which calls the `AddText()` function. This function is defined in the header.

The `AddText()` function first assigns the sentence variable to the text typed in the text field. Next, it creates a new text node containing the sentence, and appends the new text node to the paragraph.

Load this document into a browser to test it, and try adding several sentences by typing them and clicking the Add button. Figure 14.3 shows Firefox's display of this document after several sentences have been added to the paragraph.

**FIGURE 14.3**  
Firefox shows  
the text-adding  
example.



## Try It Yourself

### Creating a Navigation Tree

One common use of JavaScript and the DOM is to create a dynamic tree-like navigation system for a site, with sections that can be expanded and collapsed.

Although this is unnecessary for small sites, it's a good way to organize what may be hundreds of links for a larger site. To further experiment with the techniques you learned about in this hour, you can create a simple navigation tree using the DOM.

To begin, you will need an HTML document that defines the content of the navigation tree, shown in Listing 14.4.

#### **LISTING 14.4** The HTML for the Navigation Tree Example

```
<html>
<head><title>Creating a Navigation Tree</title>
<style>
 A {text-decoration: none;}
 #productsmenu,#supportmenu,#contactmenu {
 display: none;
 margin-left: 2em;
 }
</style>
</head>
<body>
<h1>Navigation Tree Example</h1>
<p>The navigation tree below allows you to expand and
collapse items.</p>

 [+] Products
```

**LISTING 14.4 Continued**


---

```

<ul ID="productsmenu">
 Product List
 Order Form
 Price List

[+] Support
 <ul id="supportmenu">
 Support Forum
 Contact Support

[+] Contact Us
 <ul id="contactmenu">
 Service Department
 Sales Department

<script language="javascript" type="text/javascript"
 src="tree.js">
</script>
</body>
</html>

```

---

In this document, the links are laid out as a nested list using `<ul>` and `<li>` tags. Using a standard list like this rather than `<div>` tags has two benefits: First, the browser formats the tree as a list with bullets automatically. Second, it supports older browsers—even a browser that does not support CSS or JavaScript will load and display the list correctly. It won't have the dynamic features, but the links will still work.

The tree has three main nodes: Products, Support, and Contact Us. Each one has a link you can click to display or hide the links in that section. The `id` attribute has been used on each `<a>` tag so the script can attach an event handler to it. Each node also has a submenu defined with `<ul>` and `<li>` tags. An `id` attribute is also used on the `<ul>` tag so the script can hide or display the list.

The `<script>` tag at the end of the document includes the script you will create next. The tag is placed after the body of the page so that the script can add event handlers to elements in the page.

The `<style>` block at the beginning of the document adds some formatting to the links, and uses the `display: none` attribute to initially hide the submenus. They will be revealed by the script when the link is clicked.

The script for this example is shown in Listing 14.5.

**LISTING 14.5 The JavaScript File for the Navigation Tree Example**


---

```
function Toggle(e) {
 // Don't try this in old browsers
 if (!document.getElementById) return;
 // Get the event object
 if (!e) var e = window.event;
 // Which link was clicked?
 whichlink = (e.target) ? e.target.id : e.srcElement.id;
 // get the menu object
 obj=document.getElementById(whichlink+"menu");
 // Is the menu visible?
 visible=(obj.style.display=="block")
 // Get the key object (the link itself)
 key=document.getElementById(whichlink);
 // Get the name (Products, Contact, etc.)
 keyname = key.firstChild.nodeValue.substring(3);
 if (visible) {
 // hide the menu
 obj.style.display="none";
 key.firstChild.nodeValue = "[+]" + keyname;
 } else {
 // show the menu
 obj.style.display="block";
 key.firstChild.nodeValue = "[-]" + keyname;
 }
}
document.getElementById("products").onclick=Toggle;
document.getElementById("support").onclick=Toggle;
document.getElementById("contact").onclick=Toggle;
```

---

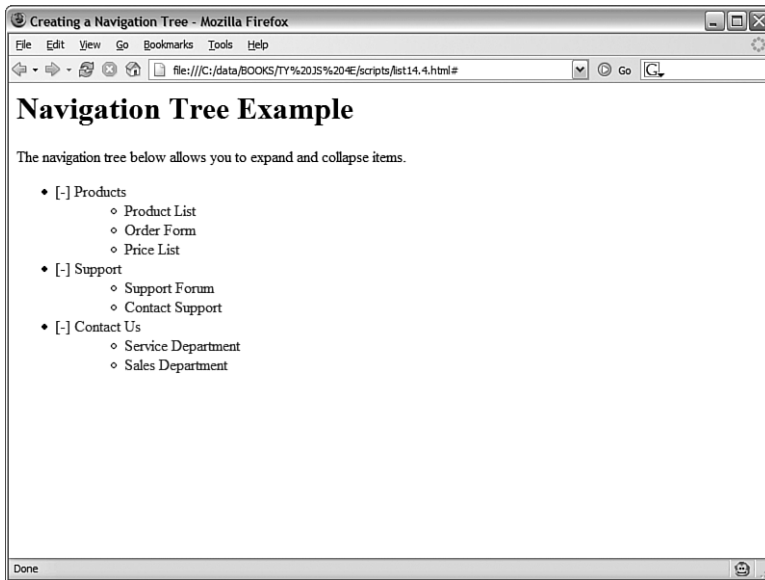
The `Toggle()` function shows or hides a menu. It first determines which of the links triggered the event, and then uses the link's `id` attribute to find the objects for the menu and for the link itself. If the menu is currently visible, it is hidden, and if it is currently hidden, it is revealed. The appropriate symbol `[+]` or `[-]` is added to the link name and displayed by modifying the text node's `nodeValue` attribute.

The last three lines of the script assign the `Toggle()` function as the `onClick` event handler for the three top-level links of the tree.

To use this script, save it as `tree.js` in the same folder as the HTML document you created previously, and load the HTML file into a browser. Figure 14.4 shows the example in action after all three nodes of the tree have been expanded.

**Did you  
Know?**

To add items to the navigation tree, add links to the HTML file. If you add a new submenu, you need to assign an `id` attribute to the link, use the same word plus menu as the `id` of the menu, and assign its `onclick` event to the `Toggle()` function at the end of the script.



**FIGURE 14.4**  
The navigation tree example as displayed by Firefox.

## Summary

In this hour, you learned some of the advanced features of the new W3C DOM (Document Object Model). You learned the functions and properties you can use to manage DOM objects, and used example scripts to hide and show elements within a page, modify text, and add text. Finally, you created a dynamic navigation tree using DOM features.

Congratulations—you've reached the end of Part III! Now that you've learned all about the DOM, you will move on to some advanced aspects of JavaScript. In the next hour, you will learn how to create scripts that unobtrusively handle multiple browsers, and some best practices for more involved scripting.

## Q&A

**Q.** *Can I avoid assigning an id attribute to every DOM object I want to handle with a script?*

**A.** Yes. Although the scripts in this hour typically use the id attribute for convenience, you can actually locate any object in the page by using combinations of node properties such as `firstChild` and `nextSibling`. However, keep in mind that any change you make to the HTML can change an element's place in the DOM hierarchy, so the id attribute is a reliable way to handle this.

- Q.** *Can I include HTML tags, such as `<b>`, in the new text I assign to a text node?*
- A.** Text nodes are limited to text if you use the `nodeValue` attribute. However, the `innerHTML` property does not have this limitation and can be used to insert any HTML.
- Q.** *Is there a reference that specifies which DOM properties and methods work in which browser versions?*
- A.** Yes, several websites are available that keep up-to-date lists of browser features. Some of these are listed in Appendix A, “Other JavaScript Resources.”

## Quiz Questions

Test your knowledge of the DOM by answering the following questions.

- 1.** If `para1` is the DOM object for a paragraph, what is the correct syntax to change the text within the paragraph to “New Text”?
  - a.** `para1.value="New Text";`
  - b.** `para1.firstChild.nodeValue="New Text";`
  - c.** `para1.nodeValue="New Text";`
- 2.** Which of the following DOM objects never has a parent node?
  - a.** `body`
  - b.** `div`
  - c.** `document`
- 3.** Which of the following is the correct syntax to get the DOM object for a heading with the identifier `head1`?
  - a.** `document.getElementById("head1")`
  - b.** `document.GetElementById("head1")`
  - c.** `document.getElementsById("head1")`

## Quiz Answers

1. b. The actual text is the `nodeValue` attribute of the text node, which is a child of the paragraph node.
2. c. The document object is the root of the DOM object tree, and has no parent object.
3. a. `getElementById` has a lowercase *g* at the beginning, and a lowercase *d* at the end, contrary to what you might know about normal English grammar.

## Exercises

If you want to gain more experience using the advanced DOM features you learned in this hour, try the following exercise:

- ▶ Add a third check box to Listing 14.1 to allow the paragraph of text to be shown or hidden. You will need to add an `id` attribute to the `<p>` tag, add a check box to the form, and add the appropriate lines to the script.
- ▶ Add a fourth node to the navigation tree in Listing 14.4, and make the appropriate changes to the script in Listing 14.5 to make the new section of the tree expand and collapse correctly.

*This page intentionally left blank*



---

## **PART IV:**

# **Working with Advanced JavaScript Features**

<b>HOURL 15</b>	Unobtrusive Scripting	<b>235</b>
<b>HOURL 16</b>	Debugging JavaScript Applications	<b>255</b>
<b>HOURL 17</b>	AJAX: Remote Scripting	<b>273</b>
<b>HOURL 18</b>	Greasemonkey: Enhancing the Web with JavaScript	<b>293</b>

*This page intentionally left blank*

## HOURL 15

# Unobtrusive Scripting

---

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

- ▶ Best practices for creating unobtrusive scripts
- ▶ Separating content, presentation, and behavior
- ▶ Following web standards to create cross-browser scripts
- ▶ Reading and displaying browser information
- ▶ Using feature sensing to avoid errors
- ▶ Supporting non-JavaScript browsers

You have now learned enough JavaScript to create some complex effects—and potentially to create some complex problems. In Part IV, you will learn about some of JavaScript's more advanced features, and learn how to avoid problems as you progress to longer and more complicated scripts.

In this hour, you'll learn some guidelines for creating scripts and pages that are easy to maintain, easy to use, and follow web standards. This is known as *unobtrusive scripting*: Scripts add features without getting in the way of the user, the developer maintaining the code, or the designer building the layout of the site. You'll also learn how to make sure your scripts will work in multiple browsers, and won't stop working when a new browser comes along.

## **Scripting Best Practices**

As you start to develop more complex scripts, it's important to know some scripting *best practices*. These are guidelines for using JavaScript that more experienced programmers have learned the hard way. Here are a few of the benefits of following these best practices:

- ▶ Your code will be readable and easy to maintain, whether you're turning the page over to someone else or just trying to remember what you did a year ago.

- ▶ You'll create code that follows standards, and won't be crippled by a new version of a particular browser.
- ▶ You'll create pages that work even without JavaScript.
- ▶ It will be easy to adapt code you create for one site to another site or project.
- ▶ Your users will thank you for creating a site that is easy to use, and easy to fix when things go wrong.

Whether you're writing an entire AJAX web application or simply enhancing a page with a three-line script, it's useful to know some of the concepts that are regularly considered by those who write complex scripts for a living. The following sections introduce some of these best practices.

## **Content, Presentation, and Behavior**

When you create a web page, or especially an entire site or application, you're dealing with three key areas: *content*, *presentation*, and *behavior*.

- ▶ *Content* consists of the words that a visitor can read on your pages. You create the content as text, and mark it up with HTML to define different classes of content—headings, paragraphs, links, and so on.
- ▶ *Presentation* is the appearance and layout of the words on each page—text formatting, fonts, colors, and graphics. Although it was common in the early days of the Web to create the presentation using HTML only, you can now use Cascading Style Sheets (CSS) to define the presentation.
- ▶ *Behavior* is what happens when you interact with a page—items that highlight when you move over them, forms you can submit, and so on. This is where JavaScript comes in, along with server-side languages such as PHP.

It's a good idea to keep these three areas in mind, especially as you create larger sites. Ideally, you want to keep content, presentation, and behavior separated as much as possible. One good way to do this is to create an external CSS file for the presentation and an external JavaScript file for the behavior, and link them to the HTML document.

Keeping things separated like this makes it easier to maintain a large site—if you need to change the color of the headings, for example, you can make a quick edit to the CSS file without having to look through all of the HTML markup to find the right place to edit. It also makes it easy for you to reuse the same CSS and JavaScript on multiple pages of a site. Last, but not least, this will encourage you to use each language where its strengths lie, making your job easier.

## Progressive Enhancement

One of the old buzzwords of web design was *graceful degradation*. The idea was that you could build a site that used all of the bells and whistles of the latest browsers, as long as it would “gracefully degrade” to work on older browsers. This mostly meant testing on a few older browsers and hoping it worked, and there was always the possibility of problems in browsers that didn’t support the latest features.

Ironically, you might expect browsers that lack the latest features to be older, less popular ones, but some of the biggest problems are with brand-new browsers—those included with mobile phones and other new devices, all of which are primitive compared to the latest browsers running on computers.

One new approach to web design that addresses this problem is known as *progressive enhancement*. The idea is to keep the HTML documents as simple as possible, so they’ll definitely work in even the most primitive browsers. After you’ve tested that and made sure the basic functionality is there, you can add features that make the site easier to use or better looking for those with new browsers.

If you add these features unobtrusively, they have little chance of preventing the site from working in its primitive HTML form. Here are some guidelines for progressive enhancement:

- ▶ Enhance the presentation by adding rules to a separate CSS file. Try to avoid using HTML markup strictly for presentation, such as `<b>` for boldface or `<blockquote>` for an indented section.
- ▶ Enhance behavior by adding scripts to an external JavaScript file.
- ▶ Add events without using inline event handlers, as described in Hour 9, “Responding to Events,” and later in this hour.
- ▶ Use feature sensing, described later this hour, to ensure that JavaScript code only executes on browsers that support the features it requires.

The term *progressive enhancement* first appeared in a presentation and article on this topic by Steve Champeon. The original article, along with many more web design articles, is available on his company’s website at <http://hesketh.com/>.

**By the  
Way**

## Adding Event Handlers

In Hour 9, you learned that there is more than one way to set up an event handler. The simplest way is to add them directly to an HTML tag. For example, this `<body>` tag has an event handler that calls a function called `Startup`.

```
<body onLoad="Startup();">
```

This method still works, but it does mean putting JavaScript code in the HTML page, which means you haven't fully separated content and behavior. To keep things entirely separate, you can set up the event handler in the JavaScript file instead, using syntax like this:

```
window.onload=Startup;
```

Right now, this is usually the best way to set up events: It keeps JavaScript out of the HTML file, and it works in all browsers since Netscape 4 and Internet Explorer 4. However, it does have one problem: You can't attach more than one event to the same element of a page. For example, you can't have two different onLoad event handlers that both execute when the page loads.

When you're the only one writing scripts, this is no big deal—you can combine the two into one function. But when you're trying to use two or three third-party scripts on a page, and all of them want to add an onLoad event handler to the body, you have a problem.

### **The W3C Event Model**

To solve this problem and standardize event handling, the W3C created an event model as part of the DOM level 2 standard. This uses a method, `addEventListener()`, to attach a handler to any event on any element. For example, the following uses the W3C model to set up the same onLoad event handler as the previous examples:

```
window.addEventListener('load', Startup, false);
```

The first parameter of `addEventListener()` is the event name without the on prefix—load, click, mouseover, and so on. The second parameter specifies the function to handle the event, and the third is an advanced flag that indicates how multiple events should be handled. (`false` works for most purposes.)

Any number of functions can be attached to an event in this way. Because one event handler doesn't replace another, you use a separate function, `removeEventListener()`, which uses the same parameters:

```
window.removeEventListener('load', Startup, false);
```

The problem with the W3C model is that Internet Explorer (as of versions 6 and 7) doesn't support it. Instead, it supports a proprietary method, `attachEvent()`, which does much the same thing. Here's the Startup event handler defined Microsoft-style:

```
window.attachEvent('onload', Startup);
```

The `attachEvent()` method has two parameters. The first is the event, with the on prefix—onload, onclick, onmouseover, and so on. The second is the function that

will handle the event. Internet Explorer also supports a `detachEvent()` method with the same parameters for removing an event handler.

## Attaching Events the Cross-Browser Way

As you can see, attaching events in this new way is complex and will require different code for different browsers. In most cases, you're better off using the traditional method to attach events, and that method is used in most of this book's examples. However, if you really need to support multiple event handlers, you can use some `if` statements to use either the W3C method or Microsoft's method. For example, the following code adds the `ClickMe()` function as an event for the element with the `id` attribute `btn`:

```
obj = document.getElementById("btn");
if (obj.addEventListener) {
 obj.addEventListener('click',ClickMe,false);
} else if (obj.attachEvent) {
 obj.attachEvent('onclick',ClickMe);
} else {
 obj.onclick=ClickMe;
}
```

This checks for the `addEventListener()` method, and uses it if it's found. Otherwise, it checks for the `attachEvent()` method, and uses that. If neither is found, it uses the traditional method to attach the event handler. This technique is called *feature sensing* and is explained in detail later this hour.

Many universal functions are available to compensate for the lack of a consistent way to attach events. If you are using a third-party library, there's a good chance it includes an event function that can simplify this process for you.

The Yahoo! UI Library includes an event-handling function that can attach events in any browser, attach the same event handler to many objects at once, and other nice features. See <http://developer.yahoo.net/yui/> for details, and see Hour 8, "Using Built-in Functions and Libraries," for information about using third-party libraries.

***Did you  
Know?***

## Web Standards: Avoid Being Browser Specific

The Web was built on standards, such as the HTML standard developed by the W3C. Now there are a lot of standards involved with JavaScript—CSS, the W3C DOM, and the ECMAScript standard that defines JavaScript's syntax.

Right now, both Microsoft and the Mozilla Project are improving their browsers' support for web standards, but there are always going to be some browser-specific, nonstandard features, and some parts of the newest standards won't be consistently supported between browsers.

Although it's perfectly fine to test your code in multiple browsers and do whatever it takes to get it working, it's a good idea to follow the standards rather than browser-specific techniques when you can. This ensures that your code will work on future browsers that improve their standards support, whereas browser-specific features might disappear in new versions.

### By the Way

One reason to make sure you follow standards is that your pages can be better interpreted by search engines, which often helps your site get search traffic. Separating content, presentation, and behavior is also good for search engines because they can focus on the HTML content of your site without having to skip over JavaScript or CSS.

## Documenting Your Code

As you create more complex scripts, don't forget to include comments in your code to document what it does, especially when some of the code seems confusing or is difficult to get working. It's also a good idea to document all of the data structures and variables, and function arguments used in a larger script.

Comments are a good way to organize code, and will help you work on the script in the future. If you're doing this for a living, you'll definitely need to use comments so that others can work on your code as easily as you can.

## Usability

While you're adding cool features to your site, don't forget about *usability*—making things as easy, logical, and convenient as possible for users of your site. Although there are many books and websites devoted to usability information, a bit of common sense goes a long way.

For example, suppose you use a drop-down list as the only way to navigate between pages of your site. This is a common use for JavaScript, and it works well, but is it usable? Try comparing it to a simple set of links across the top of a page.

- ▶ The list of links lets you see at a glance what the site contains; the drop-down list requires you to click to see the same list.
- ▶ Users expect links and can spot them quickly—a drop-down list is more likely to be part of a form than a navigation tool, and thus won't be the first thing they look for when they want to navigate your site.
- ▶ Navigating with a link takes a single click—navigating with the drop-down list takes at least two clicks.



Remember to consider the user's point of view whenever you add JavaScript to a site, and be sure you're making the site easier to use—or at least not harder to use. Also make sure the site is easy to use even without JavaScript.

## Design Patterns

If you learn more about usability, you'll undoubtedly see *design patterns* mentioned. This is a computer science term meaning “an optimal solution to a common problem.” In web development, design patterns are ways of designing and implementing part of a site that webmasters run into over and over.

For example, if you have a site that displays multiple pages of data, you'll have “Next Page” and “Previous Page” links, and perhaps numeric links for each page. This is a common design pattern—a problem many web designers have had to solve, and one with a generally agreed-upon solution. Other common web design patterns include a login form, a search engine, or a list of navigation links for a site.

Of course, you can be completely original and make a search engine, a shopping cart, or a login form that looks nothing like any other, but unless you have a way of making them even easier to use, you're better off following the pattern, and giving your users an experience that matches their expectations.

Although you can find some common design patterns just by browsing sites similar to yours and noticing how they solved particular problems, there are also sites that specialize in documenting these patterns, and they're a good place to start if you need ideas on how to make your site work.

The Yahoo! Developer Network documents a variety of design patterns used on their network of sites, many of which are implemented using JavaScript:  
<http://developer.yahoo.net/ypatterns/>.

***Did you  
Know?***

## Accessibility

One final aspect of usability to consider is *accessibility*—making your site as accessible as possible for all users, including the disabled. For example, blind users might use a text-reading program to read your site, which will ignore images and most scripts. More than just good manners, accessibility is mandated by law in some countries.

The subject of accessibility is complex, but you can get most of the way there by following the philosophy of progressive enhancement: Keep the HTML as simple as possible, keep JavaScript and CSS separate, and make JavaScript an enhancement rather than a requirement for using your site.

## Reading Browser Information

In Hour 4, “Working with the Document Object Model (DOM),” you learned about the various objects (such as `window` and `document`) that represent portions of the browser window and the current web document. JavaScript also includes an object called `navigator` that you can use to read information about the user’s browser.

The `navigator` object isn’t part of the DOM, so you can refer to it directly. It includes a number of properties, each of which tells you something about the browser. These include the following:

- ▶ `navigator.appCodeName` is the browser’s internal code name, usually `Mozilla`.
- ▶ `navigator.appName` is the browser’s name, usually `Netscape` or `Microsoft Internet Explorer`.
- ▶ `navigator.appVersion` is the version of the browser being used—for example, `4.0(Win95;I)`.
- ▶ `navigator.userAgent` is the user-agent header, a string that the browser sends to the web server when requesting a web page. It includes the entire version information—for example, `Mozilla/4.0(Win95;I)`.
- ▶ `navigator.language` is the language (such as `English` or `Spanish`) of the browser. This is stored as a code, such as `“en_US”` for U.S. English. This property is supported only by `Netscape` and `Firefox`.
- ▶ `navigator.platform` is the computer platform of the current browser. This is a short string, such as `Win16`, `Win32`, or `MacPPC`. You can use this to enable any platform-specific features—for example, `ActiveX` components.

### **By the Way**

As you might have guessed, the `navigator` object is named after `Netscape Navigator`, the browser that originally supported JavaScript. Fortunately, this object is also supported by `Internet Explorer` and most other recent browsers.

## Displaying Browser Information

As an example of how to read the `navigator` object’s properties, Listing 15.1 shows a script that displays a list of the properties and their values for the current browser.

**LISTING 15.1** A Script to Display Information About the Browser

---

```
<html>
<head>
<title>Browser Information</title>
</head>
<body>
<h1>Browser Information</h1>
<hr>
<p>
The navigator object contains the following information
about the browser you are using.
</p>

<script language="JavaScript" type="text/javascript">
document.write("Code Name: " + navigator.appCodeName);
document.write("App Name: " + navigator.appName);
document.write("App Version: " + navigator.appVersion);
document.write("User Agent: " + navigator.userAgent);
document.write("Language: " + navigator.language);
document.write("Platform: " + navigator.platform);
</script>

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

---

This script includes a basic HTML document. A script is used within the body of the document to display each of the properties of the navigator object using the `document.write()` statement.

To try this script, load it into the browser of your choice. If you have more than one browser or browser version handy, try it in each one. Firefox's display of the script is shown in Figure 15.1.

## Dealing with Dishonest Browsers

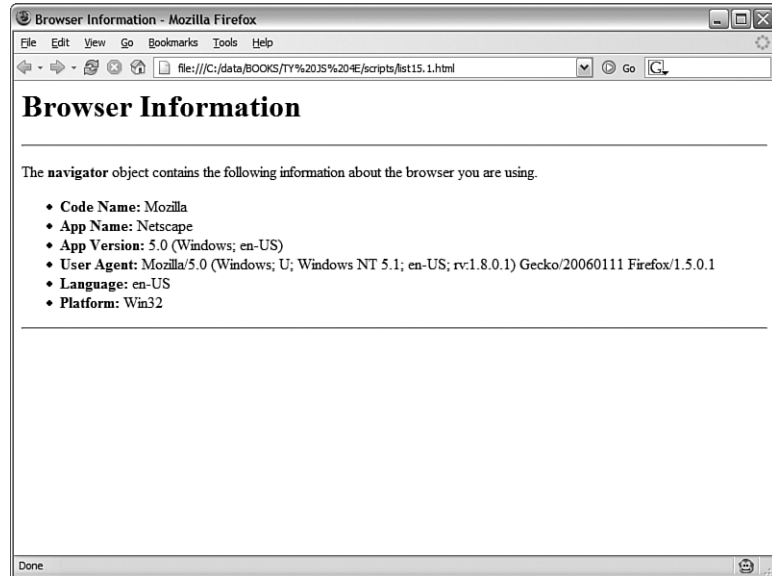
If you tried the browser information script in Listing 15.1 using one of the latest versions of Internet Explorer, you probably got a surprise. Figure 15.2 shows how Internet Explorer 6.0 displays the script.

There are several unexpected things about this display. First of all, the `navigator.language` property is listed as undefined. This isn't much of a surprise because this property isn't yet supported by Internet Explorer.

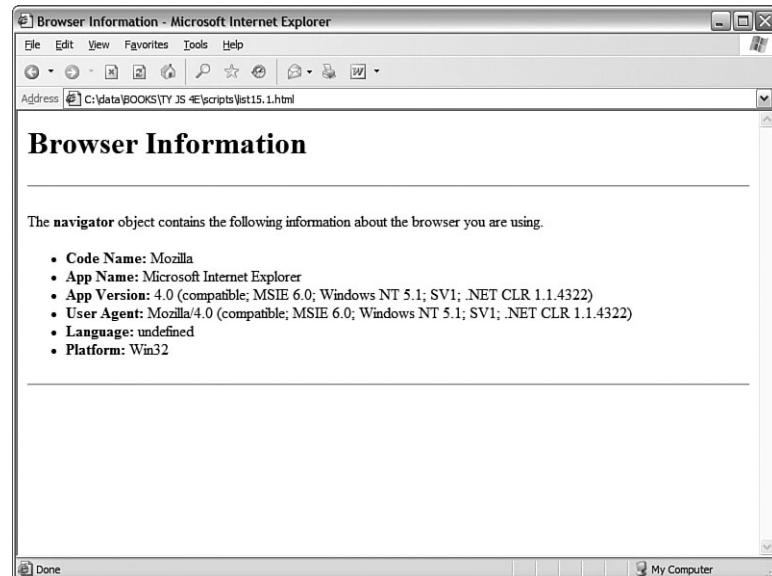
More important, you'll notice that the word Mozilla appears in the Code Name and User Agent fields. The full user agent string reads as follows:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

**FIGURE 15.1**  
Firefox displays  
the browser  
information  
script.



**FIGURE 15.2**  
Internet Explorer  
displays the  
browser infor-  
mation script.



Believe it or not, Microsoft did have a good reason for this. At the height of the browser wars, about the time Netscape 3.0 and IE 3.0 came out, it was becoming common to see “Netscape only” pages. Some webmasters who used features such as frames and JavaScript set their servers to turn away browsers without Mozilla in their user agent string. The problem with this was that most of these features were also supported by Internet Explorer.

Microsoft solved this problem in IE 4.0 by making IE's user agent read *Mozilla*, with the word *compatible* in parentheses. This allows IE users to view those pages, but still includes enough details to tell web servers which browser is in use.

You've probably already noticed the other problem with Internet Explorer 6.0's user agent string: the portion reading *Mozilla/4.0*. Not only is IE claiming to be Netscape, but it's also masquerading as version 4.0. Why?

As it turns out, this was another effort by Microsoft to stay one step ahead of the browser wars, although this one doesn't make quite as much sense. Because poorly written scripts were checking specifically for "*Mozilla/4*" for dynamic HTML pages, Microsoft was concerned that its 5.0 version would fail to run these pages. Since changing it now would only create more confusion, this tradition continues with IE 6.0.

Microsoft isn't alone in confusing browser IDs. Netscape version 6 displays a user agent string beginning with *Mozilla/5*, and an app version of 5.0. (Netscape 5.0 was Netscape's open-source browser, code named *Mozilla*, which formed the foundation of Netscape 6 and Firefox.)

***By the  
Way***

Although these are two interesting episodes in the annals of the browser wars, what does all this mean to you? Well, you'll need to be careful when your scripts are trying to differentiate between IE and Netscape, and between different versions. You'll need to check for specific combinations instead of merely checking the `navigator.appVersion` value. Fortunately, there's a better way to handle this, as you'll learn in the next section.

## Cross-Browser Scripting

If all of those details about detecting different browser versions seem confusing, here's some good news—in most cases, you can write cross-browser scripts without referring to the `navigator` object at all. This is not only easier, it's better, because browser-checking code is often confused by new browser versions, and has to be updated each time a new browser is released.

### Feature Sensing

Checking browser versions is sometimes called *browser sensing*. The better way of dealing with multiple browsers is called *feature sensing*. In feature sensing, rather than checking for a specific browser, you check for a specific feature. For example, suppose your script needs to use the `document.getElementById()` function. You can begin a script with an `if` statement that checks for the existence of this function:

```
if (document.getElementById) {
 // do stuff
}
```

If the `getElementById` function exists, the block of code between the brackets will be executed. Another common way to use feature sensing is at the beginning of a function that will make use of a feature:

```
function changeText() {
 if (!document.getElementById) return;
 // the rest of the function executes if the feature is supported
}
```

If this looks familiar, it's because it's been used in several previous examples in this book. For example, most of the code listings in Hour 14, "Using Advanced DOM Features," make use of feature sensing to prevent errors in browsers that don't support the W3C DOM.

You don't need to check for *every* feature before you use it—for example, there's not much point in verifying that the `window` object exists in most cases. You can also assume that the existence of one feature means others are supported: If `getElementById()` is supported, chances are the rest of the W3C DOM functions are supported.

Feature sensing is a very reliable method of keeping your JavaScript unobtrusive—if a browser supports the feature, it works, and if the browser doesn't, your script stays out of the way. It's also much easier than trying to keep track of hundreds of different browser versions and what they support.

### **By the Way**

Feature sensing is also handy when working with third-party libraries, as discussed in Hour 8. You can check for the existence of an object or a function belonging to the library to verify that the library file has been loaded before your script uses its features.

## **Dealing with Browser Quirks**

So, if feature sensing is better than browser sensing, why do you still need to know about the `navigator` object? There's one situation where it still comes in handy, although if you're lucky you won't find yourself in that situation.

As you develop a complex script and test it in multiple browsers, you might run across a situation where your perfectly standard code works as it should in one browser, and fails to work in another. Assuming you've eliminated the possibility of a problem with your script, you've probably run into a browser bug, or a difference in features between browsers at the very least. Here are some tips for this situation:

- ▶ Double-check for a bug in your own code. See Hour 16, “Debugging JavaScript Applications,” for debugging tips.
- ▶ Search the Web to see if others have run into the same bug. Often you’ll find that someone else has already found a workaround.
- ▶ Try a different approach to the code, and you might sidestep the bug.
- ▶ If the problem is that a feature is missing in one browser, use feature sensing to check for that feature.
- ▶ When all else fails, use the `navigator` object to detect a particular browser and substitute some code that works in that browser. This should be your last resort.

Peter-Paul Koch’s QuirksMode, [www.quirksmode.org](http://www.quirksmode.org), is a good place to start when you’re looking for specific information about browser bugs.

***Did you  
Know?***

## Supporting Non-JavaScript Browsers

Some visitors to your site will be using browsers that don’t support JavaScript at all. These aren’t just a few holdouts using ancient browsers—actually, there are more non-JavaScript browsers than you might think:

- ▶ Both Internet Explorer and Firefox include an option to turn off JavaScript, and some users do so. More often, the browser might have been set up by their ISP or employer with JavaScript turned off by default, usually in a misguided attempt to increase security.
- ▶ Some corporate firewalls and personal antivirus software block JavaScript.
- ▶ Some ad-blocking software mistakenly prevents scripts from working even if they aren’t related to advertising.
- ▶ More and more mobile phones are coming with web browsers these days, and most of these support little to no JavaScript.
- ▶ Some disabled users use special-purpose browsers or text-only browsers that might not support JavaScript.

As you can see, it would be foolish to assume that all of your visitors will support JavaScript. Two techniques you can use to make sure these users can still use the site are discussed in the following sections.

**By the  
Way**

Search engines are another “browser” that will visit your site frequently, and they usually don’t pay any attention to JavaScript. If you want search engines to fully index your site, it’s critical that you avoid making JavaScript a requirement to navigate the site.

## Using the `<noscript>` Tag

One way to be friendly to non-JavaScript browsers is to use the `<noscript>` tag. Supported in most modern browsers, this tag displays a message to non-JavaScript browsers. Browsers that support JavaScript ignore the text between the `<noscript>` tags, whereas others display it. Here is a simple example:

```
<noscript>
This page requires JavaScript. You can either switch to a browser
that supports JavaScript, turn your browser's script support on,
or switch to the Non-JavaScript version of
this page.
</noscript>
```

Although this works, the trouble is that `<noscript>` is not consistently supported by all browsers that support JavaScript. An alternative that avoids `<noscript>` is to send users with JavaScript support to another page. This can be accomplished with a single JavaScript statement:

```
<script language="JavaScript" type="text/javascript">
window.location="JavaScript.html";
</script>
```

This script redirects the user to a different page. If the browser doesn’t support JavaScript, of course, the script won’t be executed, and the rest of the page can display a warning message to explain the situation.

## Keeping JavaScript Optional

Although you can detect JavaScript browsers and display a message to the rest, the best choice is to simply make your scripts unobtrusive. Use JavaScript to enhance rather than as an essential feature, keep JavaScript in separate files, assign event handlers in the JavaScript file rather than in the HTML, and browsers that don’t support JavaScript will simply ignore your script.

In those rare cases where you absolutely need JavaScript—for example, an AJAX application or a JavaScript game—you can warn users that JavaScript is required. However, it’s a good idea to offer an alternative JavaScript-free way to use your site, especially if it’s an e-commerce or business site that your business relies on. Don’t turn away customers with lazy programming.



One place you should definitely *not* require JavaScript is in the navigation of your site. Although you can create drop-down menus and other fancy navigation tools using JavaScript, they prevent users' non-JavaScript browsers from viewing all of your site's pages. They also prevent search engines from viewing the entire site, compromising your chances of getting search traffic.

Google's Gmail application (mail.google.com), one of the most well-known uses of AJAX, requires JavaScript for its elegant interface. However, Google offers a Basic HTML View that can be used without JavaScript. This allows them to support older browsers and mobile phones without compromising the user experience for those with modern browsers.

***By the  
Way***

## Avoiding Errors

If you've made sure JavaScript is only an enhancement to your site, rather than a requirement, those with browsers that don't support JavaScript for whatever reason will still be able to navigate your site. One last thing to worry about: It's possible for JavaScript to cause an error, or confuse these browsers into displaying your page incorrectly.

This is a particular concern with browsers that partially support JavaScript, such as mobile phone browsers. They might interpret a `<script>` tag and start the script, but might not support the full JavaScript language or DOM. Here are some guidelines for avoiding errors:

- ▶ Use a separate JavaScript file for all scripts. This is the best way to guarantee that the browser will ignore your script completely if it does not have JavaScript support.
- ▶ Use feature sensing whenever your script tries to use the newer DOM features, such as `document.getElementById()`.
- ▶ Test your pages with your browser's JavaScript support turned off. Make sure nothing looks strange, and make sure you can still navigate the site.

The developer's toolbars for Firefox and Internet Explorer include a convenient way to turn off JavaScript for testing. See Hour 16 for details.

***Did you  
Know?***



## Try It Yourself

### Creating an Unobtrusive Script

As an example of unobtrusive scripting, you can create a script that adds functionality to a page with JavaScript without compromising its performance in older browsers. In this example, you will create a script that creates graphic check boxes as an alternative to regular check boxes.

### By the Way

Note: See Hour 11, “Getting Data with Forms,” for the basics of working with forms in JavaScript.

Let’s start with the final result: Figure 15.3 shows this example as it appears in Firefox. The first check box is an ordinary HTML one, and the second is a graphic check box managed by JavaScript.

**FIGURE 15.3**

The graphic check box example in action.



The graphic check box is just a larger graphic that you can click on to display the checked or unchecked version of the graphic. Although this could just be a simple JavaScript simulation that acts like a check box, it’s a bit more sophisticated. Take a look at the HTML for this example in Listing 15.2.

### LISTING 15.2 The HTML File for the Graphic Check box Example

```
<html>
<head>
<title>Graphic Checkboxes</title>
</head>
<body>
<h1>Graphic Checkbox Example</h1>
<form name="form1">
<p>
<input type = "checkbox" name="check1" id="check1">
An ordinary checkbox.
</p><p>
<input type = "checkbox" name="check2" id="check2">
```

**LISTING 15.2 Continued**


---

```

A graphic checkbox, created with unobtrusive JavaScript.
</p>
</form>
<script language="JavaScript" type="text/javascript">
 src="checkbox.js">
</script>
</body>
</html>

```

---

If you look closely at the HTML, you'll see that the two check boxes are defined in exactly the same way with the standard `<input>` tag. Rather than substitute for a check box, this script actually replaces the regular check box with the graphic version. The script for this example is shown in Listing 15.3.

**LISTING 15.3 The JavaScript File for the Graphic Check box Example**


---

```

function graphicBox(box) {
 // be unobtrusive
 if (!document.getElementById) return;
 // find the object and its parent
 obj = document.getElementById(box);
 parentobj = obj.parentNode;
 // hide the regular checkbox
 obj.style.visibility = "hidden";
 // create the image element and set its onclick event
 img = document.createElement("IMG");
 img.onclick = Toggle;
 img.src = "unchecked.gif";
 // save the checkbox id within the image ID
 img.id = "img" + box;
 // display the graphic checkbox
 parentobj.insertBefore(img,obj);
}

function Toggle(e) {
 if (!e) var e=window.event;
 // find the image ID
 img = (e.target) ? e.target : e.srcElement;
 // find the checkbox by removing "img" from the image ID
 checkid = img.id.substring(3);
 checkbox = document.getElementById(checkid);
 // "click" the checkbox
 checkbox.click();
 // display the right image for the clicked or unclicked state
 if (checkbox.checked) file = "checked.gif";
 else file="unchecked.gif";
 img.src=file;
}

//replace the second checkbox with a graphic
graphicBox("check2");

```

---

This script has three main components:

- ▶ The `graphicBox()` function converts a regular check box to a graphic one. It starts by hiding the existing check box by changing its `style.visibility` property, and then creates a new image node containing the `unchecked.gif` graphic and inserts it into the DOM next to the original check box. (These DOM features were described in the previous hour.) It gives the image an `id` attribute containing the text `img` plus the check box's `id` attribute to make it easier to find the check box later.
- ▶ The `Toggle()` function is specified by `graphicBox()` as the event handler for the new image's `onClick` event. This function removes `img` from the image's `id` attribute to find the `id` of the real check box. It executes the `click()` method on the check box, toggling its value. Finally, it changes the image to `unchecked.gif` or `checked.gif` depending on the state of the real check box.
- ▶ The last line of the script file runs the `graphicBox()` function to replace the second check box with the `id` attribute `check2`.

Using this technique has three important advantages. First, it's an unobtrusive script. The HTML has been kept simple, and browsers that don't support JavaScript will display the ordinary check box. Second, because the real check box is still on the page but hidden, it will work correctly when the form is submitted to a server-side script. Last but not least, you can use it to create any number of graphic check boxes simply by defining regular ones in the HTML file and adding a call to `graphicBox()` to transform each one.

### By the Way

See Hour 19, "Using Graphics and Animation," for details on the image manipulation features used in this example.

To try this example, save the JavaScript file as `checkbox.js`, and be sure the HTML file is in the same folder. You'll also need two graphics the same size, `unchecked.gif` and `checked.gif`, in the same folder. You can download all of the files you need for this example from this book's website.



## Summary

In this hour, you've learned many guidelines for creating scripts that work in as many browsers as possible, and learned how to avoid errors and headaches when working with different browsers. Most important, you learned how you can use JavaScript while keeping your pages small, efficient, and valid with web standards.

In the next hour, you'll learn about another thing you'll run into frequently when working with more advanced scripts: bugs. Hour 16 shows you how to avoid common JavaScript errors, and how to use debugging tools and techniques to find and fix errors when they happen.

## Q&A

- Q. *Is it possible to create 100% unobtrusive JavaScript that can enhance a page without causing any trouble for anyone?***
- A.** Not quite. For example, the unobtrusive script in the Try It Yourself section of this hour is close—it will work in the latest browsers, and the regular check box will display and work fine in even ancient browsers. However, it can still fail if someone with a modern browser has images turned off: The script will hide the check box because JavaScript is supported, but the image won't be there. This is a rare circumstance, but it's an example of how any feature you add can potentially cause a problem for some small percentage of your users.
- Q. *Can I detect the user's email address using the navigator object or another technique?***
- A.** No, there is no reliable way to detect users' email addresses using JavaScript. (If there was, you would get hundreds of advertisements in your mailbox every day from companies that detected your address as you browsed their pages.) You can use a signed script to obtain the user's email address, but this requires the user's permission and only works in some versions of Netscape.
- Q. *Are there browsers besides Firefox, Netscape, and Internet Explorer that support JavaScript?***
- A.** Yes. Opera is a multiplatform browser that supports JavaScript and the W3C DOM. Apple's Safari browser for Macintosh also supports JavaScript. It's always best to support all browsers if you can, and to focus on web standards rather than particular browsers.

## Quiz Questions

Test your knowledge of unobtrusive scripting by answering the following questions.

1. Which of the following is the best place to put JavaScript code?
  - a. Right in the HTML document
  - b. In a separate JavaScript file
  - c. In a CSS file

2. Which of the following is something you *can't* do with JavaScript?
  - a. Send browsers that don't support a feature to a different page
  - b. Send users of Internet Explorer to a different page
  - c. Send users of non-JavaScript browsers to a different page
3. Which of the following is the best way to define an event handler that works in all modern browsers?
  - a. `<body onLoad="MyFunction()">`
  - b. `window.onload=MyFunction;`
  - c. `window.attachEvent('load',MyFunction,false);`

## Quiz Answers

1. b. The best place for JavaScript is in a separate JavaScript file.
2. c. You can't use JavaScript to send users of non-JavaScript browsers to a different page because the script won't be executed at all.
3. b. The code `window.onload=MyFunction;` defines an event handler in all modern browsers. This is better than using an inline event handler as in (a) because it keeps JavaScript out of the HTML document. Option (c) uses the W3C's standard method, but does not work in Internet Explorer.

## Exercises

If you want to gain more experience creating cross-browser scripts, try the following exercises:

- ▶ Add several check boxes to the HTML document in Listing 15.2, and add the corresponding function calls to the script in Listing 15.3 to replace all of them with graphic check boxes.
- ▶ Modify the script in Listing 15.3 to convert all check boxes with a `class` value of `graphic` into graphic check boxes. You can use `getElementsByTagName()` and then check each item for the right `className` property.

## HOUR 16

# Debugging JavaScript Applications

---

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

- ▶ Using good programming practices to avoid bugs
- ▶ Tips for debugging with the JavaScript console
- ▶ Using alert messages and comments to debug scripts
- ▶ Creating custom error handlers
- ▶ Using advanced debugging tools
- ▶ Debugging an actual script

As you move on to more advanced JavaScript applications in the remaining hours of this book, it's important to know how to deal with problems in your scripts. In this hour, you'll learn a few pointers on keeping your scripts bug-free, and you'll look at the tools and techniques you can use to find and eliminate bugs when they occur.

## **Avoiding Bugs**

A bug is an error in a program that prevents it from doing what it should do. If you've tried writing any scripts on your own, you've probably run into one or more bugs. If not, you will—no matter how careful you are.

Although you'll undoubtedly run into a few bugs if you write a complex script, you can avoid many others by carefully writing and double-checking your script.

## **Using Good Programming Practices**

There's not a single programmer out there whose programs always work the first time, without any bugs. However, good programmers share a few habits that help them avoid some of the more common bugs. Here are a few good habits you can develop to improve your scripts:

- ▶ Format your scripts neatly and try to keep them readable. Use consistent spacing and variable names that mean something. It's hard to determine what's wrong with a script when you can't even remember what a particular line does.
- ▶ Similarly, use JavaScript comments liberally to document your script. This will help if you need to work on the script after you've forgotten the details of how it works—or if someone else inherits the job.
- ▶ End all JavaScript statements with semicolons. Although this is optional, it makes the script more readable. Additionally, it might help the browser to produce meaningful error messages.
- ▶ Declare all variables with the `var` keyword. This is optional in most cases, but it will help make sure you really mean to create a new variable and will avoid problems with variable scope.
- ▶ Divide complicated scripts logically into functions. This will make the script easier to read, and it will also make it easy to pinpoint the cause of a problem.
- ▶ Write a large script in several phases and test the script at each phase before adding more features. This way, you can avoid having several new errors appear at once.

## Avoiding Common Mistakes

Along with following good scripting practices, you should also watch for common mistakes in your scripts. Different people make different mistakes in JavaScript programming, but the following sections explore some of the most common ones.

### Syntax Errors

A syntax error is an incorrect keyword, operator, punctuation mark, or other item in a script. Most often, it's caused by a typing error.

Typical syntax errors include mistyped commands, missing parentheses, and functions with the wrong number of arguments. Syntax errors are usually obvious—both to you when you look at the script and to the browser's JavaScript interpreter when you load the script. These errors usually result in an error message and can easily be corrected.

### Assignment and Equality

One of the most common syntax errors made by beginning JavaScript programmers is confusing the assignment operator (`=`) with the equality operator (`==`). This can be a hard error to spot because it might not result in an error message.



If you're confused about which operator to use, follow this simple rule: Use `=` to change the value of a variable, and use `==` to compare two values. Here's an example of a statement that confuses the two:

```
If (a = 5) alert("found a five.");
```

The statement looks logical enough, but `a = 5` will actually assign the value 5 to the `a` variable rather than compare the two. The browser usually detects this type of error and displays an error message in the JavaScript console, but the opposite type of error (using `==` when you mean `=`) may not be detected.

## Local and Global Variables

Another common mistake is confusing local and global variables, such as trying to use the value of a variable that was declared in a function outside the function. If you actually need to do this, you should either use a global variable or return a value from the function.

Hour 5, "Using Variables, Strings, and Arrays," describes the differences between local and global variables in detail.

**By the  
Way**

## Using Objects Correctly

Another common error is to refer to JavaScript objects incorrectly. It's important to use the correct object names and to remember when to explicitly name the parent of an object.

For example, you can usually refer to the `window.alert()` method as simply `alert()`. However, there are some cases when you must use `window.alert()`, such as in some event handlers. If you find that `alert()` or another method or property is not recognized by the browser, try specifying the window object.

Another common mistake is to assume that you can omit the document object's name, such as using `write()` instead of `document.write()`. This won't work because most scripts have a window object as their scope.

## HTML Errors

Last but not least, don't forget that JavaScript isn't the only language that can have errors. It's easy to accidentally create an error in an HTML document—for example, forgetting to include a closing `</table>` tag, or even a closing `</script>` tag.

Although writing proper HTML is beyond the scope of this book, you should be aware that sometimes improper HTML can cause errors in your JavaScript. When you experience bugs, be sure to double-check the HTML, especially the objects (such as forms or images) that your script manipulates.

**Watch  
Out!**

Your script can also introduce HTML errors if it modifies the DOM, particularly if it uses the `innerHTML` property. Double-check HTML produced by a script to avoid these problems.

## Basic Debugging Tools

If checking your script for common mistakes and obvious problems doesn't fix things, it's time to start debugging. This is the process of finding errors in a program and eliminating them. Some basic tools for debugging scripts are described in the following sections.

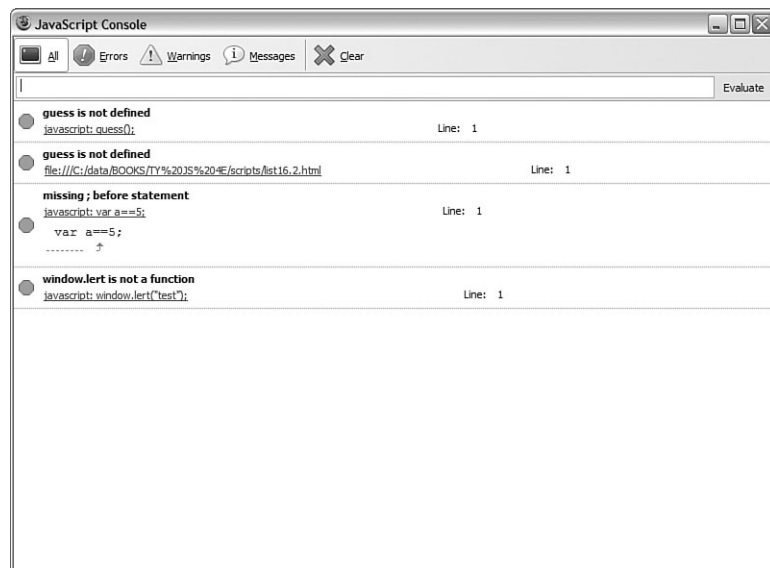
### Firefox's JavaScript Console

The first thing you should do if your script doesn't work is check for error messages. In Firefox and other Mozilla-based browsers, the messages are not displayed by default, but are logged to the JavaScript console.

To access the console, type `javascript:` in the browser's Location field or select Tools, JavaScript Console from the menu. The console displays the last few error messages that have occurred, as shown in Figure 16.1.

Along with reading the error messages, you can use the console to type a JavaScript command or expression and see its results. This is useful if you need to make sure a line of your script uses the correct syntax.

**FIGURE 16.1**  
The JavaScript console displays recent error messages.



## Displaying Error Messages in Internet Explorer

Microsoft Internet Explorer 4.0 and later do not display JavaScript error messages by default. This can make browsing poorly written pages a more pleasant experience, but it can be frustrating to JavaScript programmers.

To enable the display of error messages in Internet Explorer, select Internet Options from the Tools menu. Select the Advanced tab. In the list under Browsing, deselect the Disable Script Debugging option and enable the Display a Notification About Every Script Error option.

If you haven't enabled the display of error messages, Internet Explorer still displays an error icon on the status line when an error occurs. You can double-click this icon to display the error message.

## Alert Messages

If you're lucky, the error messages in the console will tell you how to fix your script. However, your script might not generate any error messages at all—but still fail to work correctly. In this case, the real debugging process begins.

One useful debugging technique is to add temporary statements to your script to let you know what's going on. For example, you can use an `alert` statement to display the value of a variable. After you understand what's happening to the variable, you can figure out what's wrong with the script.

You can also display debugging information in a separate browser window or frame. You can use `document.write` in some cases, but this only works when the document hasn't finished loading yet and thus isn't a reliable debugging tool.

**By the  
Way**

## Using Comments

When all else fails, you can use JavaScript comments to eliminate portions of your script until the error goes away. If you do this carefully, you can pinpoint the place where the error occurred.

You can use `//` to begin a single-line comment, or `/*` and `*/` around a section of any length. Using comments to temporarily turn off statements in a program or a script is called *commenting out* and is a common technique among programmers.

JavaScript comments were introduced and described in more detail in Hour 3, "Getting Started with JavaScript Programming."

**By the  
Way**

## Other Debugging Tools

Although you can use alert messages and a little common sense to quickly find a bug in a simple script, larger scripts can be difficult to debug. Here are a few tools you might find useful as you develop and debug larger JavaScript applications:

- ▶ HTML validators can check your HTML documents to see if they meet the HTML standard. The validation process can also help you find errors in your HTML. The W3C has a validator online at <http://validator.w3.org/>.
- ▶ Mozilla's JavaScript debugger enables you to set breakpoints, display variable values, and perform other debugging tasks. You can download the debugger at <http://www.mozilla.org/projects/venkman/>.
- ▶ Microsoft Script Debugger is similar, but works with Internet Explorer. It is available at [http://msdn.microsoft.com/library/en-us/sdbug/Html/sdbug\\_1.asp](http://msdn.microsoft.com/library/en-us/sdbug/Html/sdbug_1.asp).
- ▶ Although text and HTML editors are good basic editing tools, they can also help with debugging by displaying line numbers and using color codes to indicate valid commands.

### ***Did you Know?***

Appendix B, "Tools for JavaScript Developers," includes links to HTML validators, editors, and other debugging tools.

## Creating Error Handlers

In some cases, there may be times when an error message is unavoidable and, in a large JavaScript application, errors are bound to happen. Your scripts can respond to errors in a friendlier way using error handlers.

### Using the `onerror` Property

You can set up an error handler by assigning a function to the `onerror` property of the window object. When an error occurs in a script in the document, the browser calls the function you specify instead of the normal error dialog. For example, these statements set up a function that displays a simple message when an error occurs:

```
function errmsg(message,url,line) {
 alert("There wasn't an error. Nothing to see here.");
 return true;
}
window.onerror=errmsg;
```

These statements define a function, `errmsg()`, which handles errors by displaying a simple dialog. The last statement assigns the `errmsg()` function to the `window.onerror` property.

The `return true;` statement tells the browser that this function has handled the error, and prevents the standard error dialog from being displayed. If you use `return false;` instead, the standard error dialog will be displayed after your function exits.

You can't define an `onError` event handler in HTML. You must define it using the `window.onerror` property as shown here.

***By the  
Way***

## Displaying Information About the Error

When the browser calls your error-handling function, it passes three parameters: the error message, the URL of the document where the error happened, and the line number. The simple error handler in the previous example didn't use these values. You can create a more sophisticated handler that displays the information.

As usual, you can download this hour's examples from this book's website.

***Did you  
Know?***

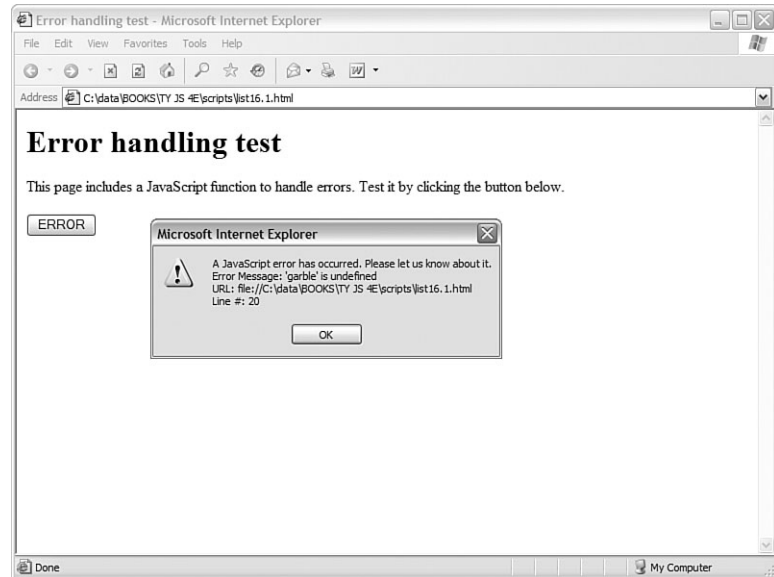
Listing 16.1 shows a complete example including an enhanced `errmsg()` function. This version displays the error message, URL, and line number in a dialog box.

### LISTING 16.1 Handling Errors with a JavaScript Function

```
<html><head>
<title>Error handling test</title>
<script language="JavaScript" type="text/javascript">
function errmsg(message,url,line) {
 amsg = "A JavaScript error has occurred. Please let us know about it.\n";
 amsg += "Error Message: " + message + "\n";
 amsg += "URL: " + url + "\n";
 amsg += "Line #: " + line;
 alert(amsg);
 return true;
}
window.onerror=errmsg;
</script>
</head>
<body>
<h1> Error handling test</h1>
<p>This page includes a JavaScript function to handle errors.
Test it by clicking the button below.</p>
<form>
 <input type="button" value="ERROR" onClick="garble">
</form>
</body>
</html>
```

This example includes a button with a nonsensical event handler. To test the error handler, click the button to generate an error. Figure 16.2 shows the example in action in Internet Explorer with the alert message displayed.

**FIGURE 16.2**  
The error-handler example in action.



### ***Did you Know?***

If you try to use an error handler and still get system error messages, make sure there isn't a syntax error in your error handler itself.

## **Using try and catch**

A more modern way of handling errors, supported by the latest browsers, is the try and catch keywords. To use it, include the try keyword, then a block of code (within braces) that might cause an error, then the catch keyword, and a block of code to handle the error:

```
try {
 DoThis();
} catch(err) {
 alert(err.description);
}
```

The try block of code always executes. If it generates an error, the catch block is executed. If there is no error, the catch block does not execute.

The error-handling code is passed an argument (err in the example) indicating the type of error. This is an object with properties including name (the error name) and description (a description of the error).

Handling errors with try and catch is a good way to deal with browser-specific code that might cause errors when run in the wrong browser. See the next hour for an example that uses try and catch to create a cross-browser AJAX function.

***Did you  
Know?***

## Advanced Debugging Tools

Although it's possible to get a simple script working with an alert message or two, you might find some other tools useful as you build more complex scripts, and especially as you work with scripts that modify the DOM. The following are some useful debugging tools available for Firefox and Internet Explorer.

### Web Developer Toolbar (Firefox)

The Web Developer Extension by Chris Pederick is an open-source extension for Firefox and other Mozilla-based browsers. This extension adds a toolbar to the browser with a variety of functions useful to developers. The following features are useful for JavaScript in particular:

- ▶ **Disable, JavaScript**—Disables JavaScript, useful for making sure pages function on non-JavaScript browsers.
- ▶ **Information, Display ID and Class Details**—Displays the values of `id` and `class` attributes for all of the elements in a page; useful for attaching event handlers or CSS styles.
- ▶ **Information, View JavaScript**—Displays all of the scripts that affect the current page, including those in external files.
- ▶ **View Source, View Generated Source**—Displays the HTML source of the current page. Unlike the browser's regular View Source function, this displays the source after any scripts have acted upon it; useful for debugging scripts that modify the DOM.

Along with these functions, the toolbar includes many useful tools for debugging HTML and CSS, working with forms, and validating pages. To install it or for more information, see its official site at <http://chrispederick.com/work/webdeveloper/>.

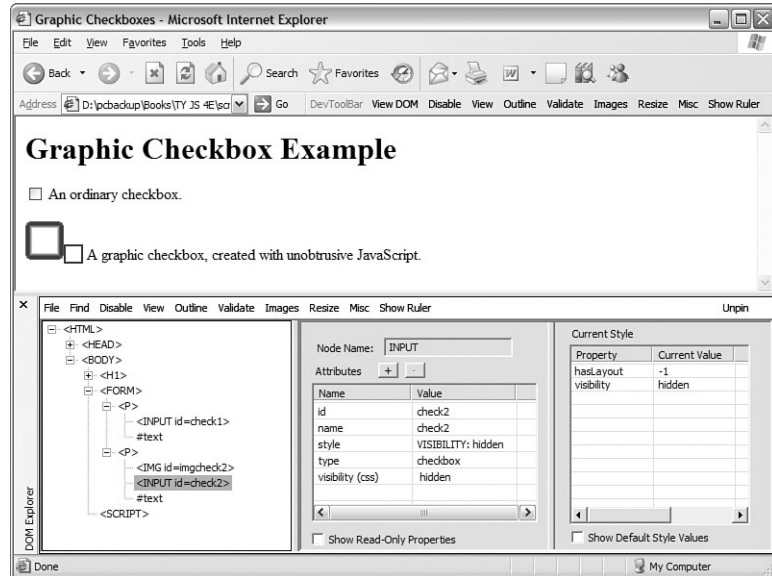
### Developer Toolbar (Internet Explorer)

Inspired by the Web Developer Extension for Firefox, Microsoft created a Developer Toolbar for Internet Explorer. Currently in beta, the toolbar works with Internet Explorer 6.0 or later. Here are some of its features useful for JavaScript programmers:

- ▶ **View DOM**—Allows you to browse the DOM of the current page and view details of elements, similar to Firefox’s DOM Inspector. This feature is shown in Figure 16.3.
- ▶ **Disable, Script**—Disables JavaScript, enabling you to test how your site works without it.
- ▶ **View, Class and ID Information**—Displays id and class attribute values; useful for attaching event handlers or CSS styles.

**FIGURE 16.3**

The Internet Explorer Developer Toolbar’s view DOM feature.



To download the IE Developer Toolbar, go to <http://www.microsoft.com/downloads/> and search for “Developer Toolbar.” The download is available for free and includes an easy installer.

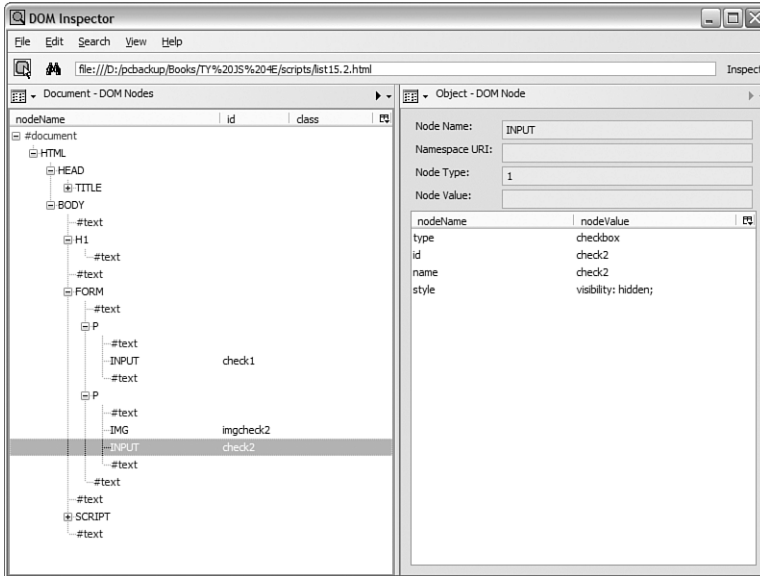
## DOM Inspector (Firefox)

The DOM Inspector is a tool built in to Firefox and other Mozilla-based browsers that enables you to browse the DOM of a web page and view the attributes of elements. You need to specifically select this feature at installation time, so you might need to reinstall Firefox to gain access to this feature. To see if your copy includes the DOM Inspector, open the Tools menu and check for a DOM Inspector menu item.

To use this tool, open the page you wish to inspect and then select Tools, DOM Inspector. You can then browse the DOM by clicking the [+] symbols for each section



of the hierarchy. Select an item within the DOM tree to view its details in the right section of the window. The DOM Inspector is shown in Figure 16.4.



**FIGURE 16.4**  
The DOM  
Inspector  
opened within  
Firefox.

## Viewing Generated Source

When your script modifies the DOM, the browser's View Source feature only gives you part of the picture—you see the source of the page when it was loaded, rather than the source created by your script as it modified the page. To test scripts that modify the DOM, you can view the *generated source* as modified by the script.

In Firefox, this feature is built in: If you select part of a page, right-click, and select View Selection Source, you'll see the generated source. You can also use the Tools menu of the Web Developer Toolbar, discussed previously, to view the generated source.

For Internet Explorer, you can use a *bookmarklet*—a short script stored as a browser bookmark—to view the generated source in a window. This bookmarklet is available at Jesse Ruderman's site at <http://www.squarefree.com/bookmarklets/>.

## JavaScript Shell

Sometimes it's helpful to be able to simply type a few JavaScript commands to see what they do, either to narrow down a bug or simply to remember the syntax of a rarely used feature. The JavaScript Shell is a bookmarklet that opens a shell window that lets you type JavaScript commands and shows the results.

The shell opens in the context of the current document, so you can use it to explore the DOM of a page or to test scripts you're working on. This feature works only in Firefox, but an online version of the shell without the context feature works in Internet Explorer.

The JavaScript Shell is available from <http://www.squarefree.com/bookmarklets/>.



## Try It Yourself

### Debugging a Script

You should now have a good understanding of what can go wrong with JavaScript programs and the tools you have available to diagnose these problems. You can now try your hand at debugging a script.

Listing 16.2 shows a script I wrote to play the classic “Guess a Number” game. The script picks a number between 1 and 100 and then allows the user 10 guesses. If a guess is incorrect, it provides a hint as to whether the target number is higher or lower.

This is a relatively simple script with a twist: It includes at least one bug and doesn't work at all in its present form.

---

#### **LISTING 16.2** The Number-Guesser Script (Complete with Bugs)

---

```
1 <html>
2 <head>
3 <title>Guess a Number</title>
4 <script LANGUAGE="JavaScript" type="text/javascript">
5 var num = Math.random() * 100 + 1;
6 var tries = 0;
7 function Guess() {
8 var g = document.form1.guess1.value;
9 tries++;
10 status = "Tries: " + tries;
11 if (g < num)
12 document.form1.hint.value = "No, guess higher.";
13 if (g > num)
14 document.form1.hint.value = "No, guess lower.";
15 if (g == num) {
16 window.alert("Correct! You guessed it in " + tries + " tries.");
17 location.reload();
18 }
19 if (tries == 10) {
20 window.alert("Sorry, time's up. The number was: " + num);
21 location.reload();
22 }
23 }
24 </script>
25 </head>
26 <body>
```

**LISTING 16.2 Continued**

```
27 <h1>Guess a Number</h1>
28 <hr>
29 <p>I'm thinking of a number between 1 and 100. Try to guess
30 it in less than 10 tries.</p>
31 <form name="form1">
32 <input type="text" size="25" name="hint" value="Enter your Guess.">
33

34 Guess:
35 <input type="text" name="guess1" size="5">
36 <input type="button" value="Guess" onClick="guess();">
37 </form>
38 </body>
39 </html>
```

Here's a summary of how this script should work:

- ▶ The first line within the `<script>` section picks a random number and stores it in the `num` variable.
- ▶ The `Guess()` function is defined in the header of the document. This function is called each time the user enters a guess.
- ▶ Within the `Guess()` function, several `if` statements test the user's guess. If it is incorrect, a hint is displayed in the text box. If the guess is correct, the script displays an alert message to congratulate the user.

## Testing the Script

To test this program, load the HTML document into your browser. It appears to load correctly and does not immediately cause any errors. However, when you enter a guess and press the Guess button, a JavaScript error occurs.

According to the JavaScript console, the error message is this:

Line 36: guess is undefined

Internet Explorer's error message refers to the same line number:

Line 36, character 1: Object expected

## Fixing the Error

As the error message indicates, there must be something wrong with the function call to the `Guess()` function in the event handler on line 36. The line in question looks like this:

```
<input type="button" value="Guess" onClick="guess();">
```

Upon further examination, you'll notice that the first two lines of the function are as follows (lines 7 and 8 of Listing 16.2):

```
function Guess() {
var guess = document.form1.guess1.value;
```

Although this might look correct at first glance, there's a problem here: `guess()` is lowercase in the event handler, whereas the function definition uses a capitalized `Guess()`. This is easy to fix. Simply change the function call in the event handler from `guess()` to `Guess()`. The corrected line will look like this:

```
<input type="button" value="Guess" onClick="Guess();">
```

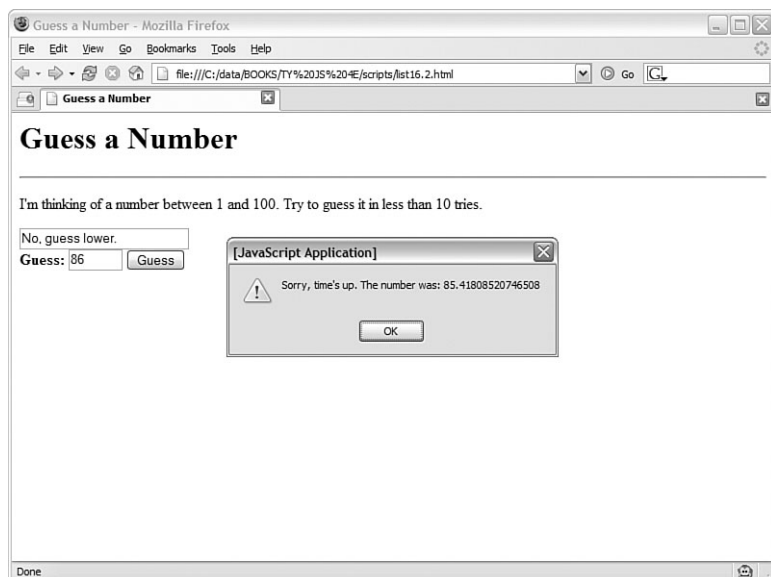
## Testing Again

Now that you've fixed the error, try the script again. This time it loads without an error, and you can enter a guess without an error. The hints about guessing higher or lower are even displayed correctly.

However, to truly test the script, you'll need to play the game all the way through. When you do, you'll discover that there's still another problem in the script: You can't win, no matter how hard you try.

After your 10 guesses are up, an alert message informs you that you've lost the game. Coincidentally, this alert message also tells you what's wrong with the script. Figure 16.5 shows how the browser window looks after a complete game, complete with this dialog box.

**FIGURE 16.5**  
The number guesser script's display after a game is finished.



As you can see from the alert message, it's no wonder you didn't win: The random number the computer picked includes more than 10 decimal places, and you've been guessing integers. You could guess decimal numbers, but you'd need a whole lot more than 10 guesses, and the game would start to lose its simplicity and charm.

To fix this problem, look at the statement at the beginning of the script that generates the random number:

```
var num = Math.random() * 100 + 1;
```

This uses the `Math.random()` method, which results in a random number between 0 and 1. The number is then multiplied and incremented to result in a number between 1 and 100.

This statement does indeed produce a number between 1 and 100, but not an integer. To fix the problem, you can add the `Math.floor()` method to chop off the decimal portion of the number. Here's a corrected statement:

```
var num = Math.floor(Math.random() * 100) + 1;
```

To fix the script, make this change and then test it again. If you play a game or two, you'll find that it works just fine. Listing 16.3 shows the complete, debugged script.

### **LISTING 16.3** The Complete, Debugged Number-Guesser Script

```
<html>
<head>
<title>Guess a Number</title>
<script LANGUAGE="JavaScript" type="text/javascript">
var num = Math.floor(Math.random() * 100) + 1;
var tries = 0;
function Guess() {
var g = document.form1.guess1.value;
tries++;
status = "Tries: " + tries;
if (g < num)
 document.form1.hint.value = "No, guess higher.";
if (g > num)
 document.form1.hint.value = "No, guess lower.";
if (g == num) {
 window.alert("Correct! You guessed it in " + tries + " tries.");
 location.reload();
}
if (tries == 10) {
 window.alert("Sorry, time's up. The number was: " + num);
 location.reload();
}
}
</script>
</head>
<body>
<h1>Guess a Number</h1>
<hr>
<p>I'm thinking of a number between 1 and 100. Try to guess
```

**LISTING 16.3 Continued**

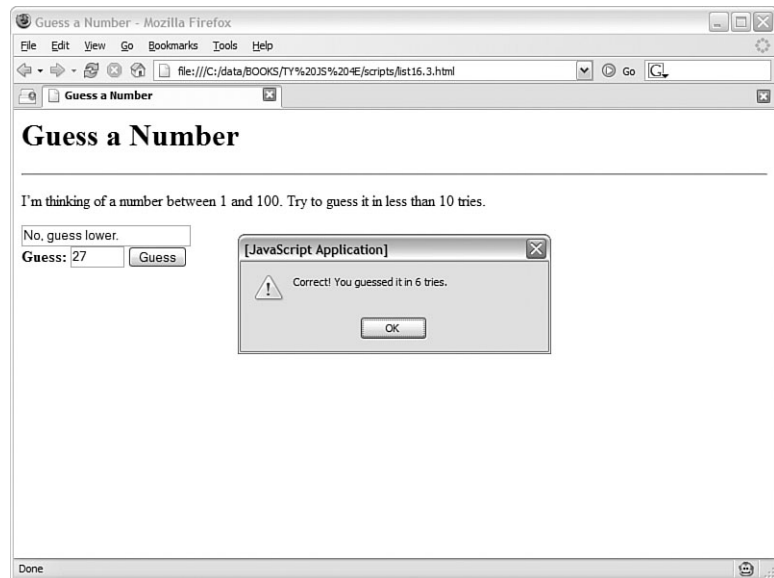
```
it in less than 10 tries.</p>
<form name="form1">
<input type="text" size="25" name="hint" value="Enter your Guess.">

Guess:
<input type="text" name="guess1" size="5">
<input type="button" value="Guess" onClick="Guess();">
</form>
</body>
</html>
```

Figure 16.6 shows the debugged example in action in Firefox after a successful game.

**FIGURE 16.6**

The number-guesser example after a successful game.



## Summary

In this hour, you've learned how to debug JavaScript programs. You examined some techniques for producing scripts with a minimum of bugs and learned about some tools that will help you find bugs in scripts. Finally, you tried your hand at debugging a script.

In Hour 17, "AJAX: Remote Scripting," you'll continue your JavaScript education by learning about AJAX, a technique that lets JavaScript work with server-side files and programs without reloading pages.

## Q&A

**Q.** *Why are some errors displayed after the script runs for a time, whereas others are displayed when the script loads?*

**A.** The JavaScript interpreter looks at scripts in the body or the heading of the document, such as function definitions, when the page loads. Event handlers aren't checked until the event happens. Additionally, a statement might look fine when the page loads, but will cause an error because of the value of a variable it uses later.

**Q.** *What is the purpose of the `location.reload` statements in the number-guesser script?*

**A.** This is an easy way to start a new game because reloading the page reinitializes the variables. This results in a new number being picked, and the default "Guess a Number" message is displayed in the hint field.

**Q.** *Which browser is best for developing and debugging scripts?*

**A.** You may or may not agree, but I find that Firefox offers the best tools for debugging scripts, such as the JavaScript console and the Web Developer Toolbar. Regardless of your preferred browser, be sure to test your scripts in multiple browsers to find any browser-specific issues they might have.

## Quiz Questions

Test your knowledge of debugging JavaScript by answering the following questions.

1. If you mistype a JavaScript keyword, which type of error is the result?
  - a. Syntax error
  - b. Function error
  - c. Pilot error
2. The process of dealing with errors in a script or a program is known as
  - a. Error detection
  - b. Frustration
  - c. Debugging

3. Which of the following is a useful technique when a script is not working but does not generate an error message?
  - a. Rewriting from scratch
  - b. Removing `<script>` tags
  - c. Adding `alert` statements

## Quiz Answers

1. a. A syntax error can result from a mistyped JavaScript keyword.
2. c. Debugging is the process of finding and fixing errors in a program.
3. c. You can add `alert` statements to a script to display variables or the current status of the script and aid in debugging.

## Exercises

If you want to gain more experience debugging scripts, try the following exercises:

- ▶ Although the number-guesser script in Listing 16.3 avoids JavaScript errors, it is still vulnerable to user errors. Add a statement to verify that the user's guess is between 1 and 100. If it isn't, display an alert message and make sure that the guess doesn't count toward the total of 10 guesses.
- ▶ Load Listing 16.2, the number-guesser script with bugs, into Mozilla's JavaScript Debugger or Microsoft's Script Debugger. Try using the watch and breakpoint features and see whether you find this to be an easier way to diagnose the problem.



## HOUR 17

# AJAX: Remote Scripting

---

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

- ▶ How AJAX enables JavaScript to communicate with server-side programs and files
- ▶ Using the XMLHttpRequest object's properties and methods
- ▶ Creating your own AJAX library
- ▶ Using AJAX to read data from an XML file
- ▶ Debugging AJAX applications
- ▶ Using AJAX to communicate with a PHP program

Remote scripting, also known as AJAX, is a browser feature that enables JavaScript to escape its client-side boundaries and work with files on a web server, or with server-side programs. In this hour, you'll learn how AJAX works and create two working examples.

## **Introducing AJAX**

Traditionally, one of the major limitations of JavaScript is that it couldn't communicate with a web server. For example, you could create a game in JavaScript, but keeping a list of high scores stored on a server would require submitting a page to a server-side form.

One of the limitations of web pages in general was that getting data from the user to the server, or from the server to the user, generally required a new page to be loaded and displayed.

AJAX (Asynchronous JavaScript and XML) is the answer to both of these problems. AJAX refers to JavaScript's capability to use a built-in object, XMLHttpRequest, to communicate with a web server without submitting a form or loading a page. Although not part of the DOM standard yet, this object is supported by Internet Explorer, Firefox, and other modern browsers.

Although the term *AJAX* was coined in 2005, XMLHttpRequest has been supported by browsers for years—it was developed by Microsoft and first appeared in Internet Explorer 5. Nonetheless, it has only recently become a popular way of developing applications because browsers that support it have become more common. Another name for this technique is *remote scripting*.

### By the Way

The term *AJAX* first appeared in an online article by Jesse James Garrett of Adaptive Path on February 18, 2005. It still appears here:  
<http://adaptivepath.com/publications/essays/archives/000385.php>

## The JavaScript Client (Front End)

JavaScript traditionally only has one way of communicating with a server—submitting a form. Remote scripting allows for much more versatile communication with the server. The *A* in AJAX stands for *asynchronous*, which means that the browser (and the user) isn't left hanging while waiting for the server to respond. Here's how a typical AJAX request works:

1. The script creates an XMLHttpRequest object and sends it to the web server. The script can continue after sending the request.
2. The server responds by sending the contents of a file, or the output of a server-side program.
3. When the response arrives from the server, a JavaScript function is triggered to act on the data.
4. Because the goal is a more responsive user interface, the script usually displays the data from the server using the DOM, eliminating the need for a page refresh.

In practice, this happens quickly, but even with a slow server, it can still work. Also, because the requests are asynchronous, more than one can be in progress at a time.

## The Back End

The part of an application that resides on the web server is known as the *back end*. The simplest back end is a static file on the server—JavaScript can request the file with XMLHttpRequest, and then read and act on its contents. More commonly, the back end is a server-side program running in a language like PHP, Perl, or Ruby.

JavaScript can send data to a server-side program using GET or POST methods, the same two ways an HTML form works. In a GET request, the data is encoded in the URL that loads the program. In a POST request, it is sent separately, and can contain more data.

## XML

The *X* in AJAX stands for *XML* (extensible markup language), the universal markup language upon which the latest versions of HTML are built. A server-side file or program can send data in XML format, and JavaScript can act on the data using its methods for working with XML. These are similar to the DOM methods you've already used—for example, you can use the `getElementsByTagName()` method to find elements with a particular tag in the data.

Keep in mind that XML is just one way to send data, and not always the easiest. The server could just as easily send plain text, which the script could display, or HTML, which the script could insert into the page using the `innerHTML` property. Some programmers have even used server-side scripts to return data in JavaScript format, which can be easily executed using the `eval` function.

JSON (JavaScript Object Notation) takes the idea of encoding data in JavaScript and formalizes it. See <http://www.json.org/> for details and code examples in many languages.

**By the  
Way**

## Popular Examples of AJAX

Although typical HTML and JavaScript is used to build web pages and sites, AJAX techniques often result in *web applications*—web-based services that perform work for the user. Here are a few well-known examples of AJAX:

- ▶ Google's Gmail mail client (<http://mail.google.com/>) uses AJAX to make a fast-responding email application. You can delete messages and perform other tasks without waiting for a new page to load.
- ▶ Amazon.com uses AJAX for some functions. For example, if you click on one of the Yes/No voting buttons for a product comment, it sends your vote to the server and a message appears next to the button thanking you, all without loading a page.
- ▶ Digg (<http://www.digg.com>) is a site where users can submit news stories and vote to determine which ones are displayed on the front page. The voting happens inside the page next to each story.

These are just a few examples. Subtle bits of remote scripting are appearing all over the Web, and you might not even notice them—you'll just be annoyed a little bit less often at waiting for a page to load.

## Frameworks and Libraries

Because remote scripting can be complicated, especially considering the browser differences you'll learn about later this hour, several frameworks and libraries have been developed to simplify AJAX programming.

For starters, three of the libraries described earlier in this book, Dojo, Prototype, and script.aculo.us, include functions to simplify remote scripting. There are also some dedicated libraries for languages like PHP, Python, and Ruby.

Some libraries are designed to add server-side functions to JavaScript, whereas others are designed to add JavaScript interactivity to a language like PHP. You'll build a simple library later this hour that will be used to handle the remote scripting functions for this hour's examples.

### ***Did you know?***

See this book's website for an up-to-date list of AJAX libraries. See Hour 8, "Using Built-in Functions and Libraries," for information about using third-party libraries with JavaScript.

## Limitations of AJAX

Remote scripting is a relatively new technology, so there are some things it can't do, and some things to watch out for. Here are some of the limitations and potential problems of AJAX:

- ▶ The script and the XML data or server-side program it requests data from must be on the same domain.
- ▶ Internet Explorer 5 and 6 use ActiveX to implement XMLHttpRequest. Although the security settings allow this by default, users with different settings might be unable to use AJAX. (Internet Explorer 7 does not have this problem.)
- ▶ Some older browsers and some less common browsers (such as mobile browsers) don't support XMLHttpRequest, so you can't count on its availability for all users.
- ▶ Requiring AJAX might compromise the accessibility of a site for disabled users.
- ▶ Users are accustomed to seeing a new page load each time they change something, so there might be a learning curve for them to understand an AJAX application.

As with other advanced uses of JavaScript, the best approach is to be unobtrusive—make sure there's still a way to use the site without AJAX support if possible, and use feature sensing to prevent errors on browsers that don't support it. See Hour 15, "Unobtrusive Scripting," for details.

## Using XMLHttpRequest

You will now take a look at how to use XMLHttpRequest to communicate with a server. This might seem a bit complex, but the process is the same for any request. Later, you will create a reusable code library to simplify this process.

### Creating a Request

The first step is to create an XMLHttpRequest object. To do this, you use the new keyword, as with other JavaScript objects. The following statement creates a request object in some browsers:

```
ajaxreq = new XMLHttpRequest();
```

The previous example works with Firefox, Mozilla, and Safari, and with Internet Explorer 7, but not Internet Explorer 5 or 6. For those browsers, you have to use ActiveX syntax:

```
ajaxreq = new ActiveXObject("Microsoft.XMLHTTP");
```

The library section later this hour demonstrates how to use the correct method depending on the browser in use. In either case, the variable you use (ajaxreq in the example) stores the XMLHttpRequest object. You'll use the methods of this object to open and send a request, as explained in the following sections.

### Opening a URL

The open() method of the XMLHttpRequest object specifies the filename as well as the method in which data will be sent to the server: GET or POST. These are the same methods supported by web forms.

```
ajaxreq.open("GET", "filename");
```

For the GET method, the data you send is included in the URL. For example, this command opens the search.php program and sends the value "John" for the query parameter:

```
ajaxreq.open("GET", "search.php?query=John");
```

## Sending the Request

You use the `send()` method of the `XMLHttpRequest` object to send the request to the server. If you are using the POST method, the data to send is the argument for `send()`. For a GET request, you can use the null value instead:

```
ajaxreq.send(null);
```

## Awaiting a Response

After the request is sent, your script will continue without waiting for a result. Because the result could come at any time, you can detect it with an event handler. The `XMLHttpRequest` object has an `onreadystatechange` event handler for this purpose. You can create a function to deal with the response and set it as the handler for this event:

```
ajaxreq.onreadystatechange = MyFunc;
```

The request object has a property, `readyState`, that indicates its status, and this event is triggered whenever the `readyState` property changes. The values of `readyState` range from 0 for a new request to 4 for a complete request, so your event handling function usually needs to watch for a value of 4.

Although the request is complete, it may not have been successful. The `status` property is set to 200 if the request succeeded, or an error code if it failed. The `statusText` property stores a text explanation of the error, or “OK” for success.

### Watch Out!

As usual with event handlers, be sure to specify the function name without parentheses. With parentheses, you’re referring to the *result* of the function; without them, you’re referring to the function itself.

## Interpreting the Response Data

When the `readyState` property reaches 4 and the request is complete, the data returned from the server is available to your script in two properties: `responseText` is the response in raw text form, and `responseXML` is the response as an XML object. If the data was not in XML format, only the text property will be available.

JavaScript’s DOM methods are meant to work on XML, so you can use them with the `responseXML` property. Later this hour, you’ll use the `getElementsByTagName()` method to extract data from XML.

## Creating a Simple AJAX Library

You should be aware by now that AJAX requests can be a bit complex. To make things easier, you can create an AJAX library. This is a JavaScript file that provides functions that handle making a request and receiving the result, which you can reuse any time you need AJAX functions.

This library will be used in the two examples later this hour. Listing 17.1 shows the complete AJAX library.

### LISTING 17.1 The AJAX Library

---

```
// global variables to keep track of the request
// and the function to call when done
var ajaxreq=false, ajaxCallback;
// ajaxRequest: Sets up a request
function ajaxRequest(filename) {
 try {
 // Firefox / IE7 / Others
 ajaxreq= new XMLHttpRequest();
 } catch (error) {
 try {
 // IE 5 / IE 6
 ajaxreq = new ActiveXObject("Microsoft.XMLHTTP");
 } catch (error) {
 return false;
 }
 }
 ajaxreq.open("GET", filename);
 ajaxreq.onreadystatechange = ajaxResponse;
 ajaxreq.send(null);
}
// ajaxResponse: Waits for response and calls a function
function ajaxResponse() {
 if (ajaxreq.readyState !=4) return;
 if (ajaxreq.status==200) {
 // if the request succeeded...
 if (ajaxCallback) ajaxCallback();
 } else alert("Request failed: " + ajaxreq.statusText);
 return true;
}
```

---

The following sections explain how this library works and how to use it.

### The ajaxRequest() Function

The ajaxRequest() function handles all of the steps necessary to create and send an XMLHttpRequest. First, it creates the XMLHttpRequest object. This requires a different command for different browsers, and will cause an error if the wrong one

executes, so try and catch are used to create the request. First the standard method is used, and if it causes an error, the ActiveX method is tried. If that also causes an error, the `ajaxreq` variable is set to `false` to indicate that AJAX is unsupported.

## The `ajaxResponse()` Function

The `ajaxResponse()` function is used as the `onreadystatechange` event handler. This function first checks the `readyState` property for a value of 4. If it has a different value, the function returns without doing anything.

Next, it checks the `status` property for a value of 200, which indicates the request was successful. If so, it runs the function stored in the `ajaxCallback` variable. If not, it displays the error message in an alert box.

## Using the Library

To use this library, follow these steps:

1. Save the library file as `ajax.js` in the same folder as your HTML documents and scripts.
2. Include the script in your document with a `<script src>` tag. It should be included before any other scripts that use its features.
3. In your script, create a function to be called when the request is complete, and set the `ajaxCallback` variable to the function.
4. Call the `ajaxRequest()` function. Its parameter is the filename of the server-side program or file. (This library supports GET requests only, so you don't need to specify the method.)
5. Your function specified in `ajaxCallback` will be called when the request completes successfully, and the global variable `ajaxreq` will store the data in its `responseXML` and `responseText` properties.

The two remaining examples in this hour make use of this library to create AJAX applications.

## Creating an AJAX Quiz Using the Library

Now that you have a reusable AJAX library, you can use it to create JavaScript applications that take advantage of remote scripting. This first example displays quiz questions on a page and prompts you for the answers.



Rather than including the questions in the script, this example reads the quiz questions and answers from an XML file on the server as a demonstration of AJAX.

Unlike most of the scripts in this book, this example requires a web server. It will not work on a local machine due to browsers' security restrictions on remote scripting.

**Watch  
Out!**

## The HTML File

The HTML for this example is straightforward. It defines a simple form with an Answer field and a Submit button, along with some hooks for the script. The HTML for this example is shown in Listing 17.2.

### LISTING 17.2 The HTML File for the Quiz Example

```
<html>
<head><title>Ajax Test</title>
<script language="JavaScript" type="text/javascript"
 src="ajax.js">
</script>
</head>
<body>
<h1>Ajax Quiz Example</h1>
<form>
<p>Question:
...

</p>
<p>Answer:
<input type="text" name="answer" id="answer">
<input type="button" value="Submit" id="submit">
</p>
<input type="button" value="Start the Quiz" id="startq">
</form>
<script language="JavaScript" type="text/javascript"
 src="quiz.js">
</script>
</body>
</html>
```

This HTML file includes the following elements:

- ▶ The `<script>` tag in the `<head>` section includes the AJAX library you created in the previous section from the `ajax.js` file.
- ▶ The `<script>` tag in the `<body>` section includes the `quiz.js` file, which will contain the quiz script.
- ▶ The `<span id="question">` tag sets up a place for the question to be inserted by the script.

- ▶ The text field with the id value answer is where the user will answer the question.
- ▶ The button with the id value submit will submit an answer.
- ▶ The button with the id value startq will start the quiz.

You can test the HTML document at this time, but the buttons won't work until you add the script.

## The XML File

The XML file for the quiz is shown in Listing 17.3. I've filled it with a few JavaScript questions, but it could easily be adapted for another purpose.

---

**LISTING 17.3 The XML File Containing the Quiz Questions and Answers**

---

```
<?xml version="1.0" ?>
<questions>
 <q>What DOM object contains URL information for the window?</q>
 <a>location
 <q>Which method of the document object finds the object for an element?</q>
 <a>getElementById
 <q>If you declare a variable outside a function, is it global or local?</q>
 <a>global
 <q>What is the formal standard for the JavaScript language called?</q>
 <a>ECMAScript
</questions>
```

---

The `<questions>` tag encloses the entire file, and each question and answer are enclosed in `<q>` and `<a>` tags. Remember, this is XML, not HTML—these are not standard HTML tags, but tags that were created for this example. Because this file will be used only by your script, it does not need to follow a standard format.

To use this file, save it as `questions.xml` in the same folder as the HTML document. It will be loaded by the script you create in the next section.

Of course, with a quiz this small, you could have made things easier by storing the questions and answers in a JavaScript array. But imagine a much larger quiz, with thousands of questions, or a server-side program that pulls questions from a database, or even a hundred different files with different quizzes to choose between, and you can see the benefit of using a separate XML file.

## The JavaScript File

Because you have a separate library to handle the complexities of making an AJAX request and receiving the response, the script for this example only needs to deal with the action for the quiz itself. Listing 17.4 shows the JavaScript file for this example.

**LISTING 17.4** The JavaScript File for the Quiz Example

---

```
// global variable qn is the current question number
var qn=0;
// load the questions from the XML file
function getQuestions() {
 obj=document.getElementById("question");
 obj.firstChild.nodeValue="(please wait)";
 ajaxCallback = nextQuestion;
 ajaxRequest("questions.xml");
}
// display the next question
function nextQuestion() {
 questions = ajaxreq.responseXML.getElementsByTagName("q");
 obj=document.getElementById("question");
 if (qn < questions.length) {
 q = questions[qn].firstChild.nodeValue;
 obj.firstChild.nodeValue=q;
 } else {
 obj.firstChild.nodeValue="(no more questions)";
 }
}
// check the user's answer
function checkAnswer() {
 answers = ajaxreq.responseXML.getElementsByTagName("a");
 a = answers[qn].firstChild.nodeValue;
 answerfield = document.getElementById("answer");
 if (a == answerfield.value) {
 alert("Correct!");
 }
 else {
 alert("Incorrect. The correct answer is: " + a);
 }
 qn = qn + 1;
 answerfield.value="";
 nextQuestion();
}
// Set up the event handlers for the buttons
obj=document.getElementById("startq");
obj.onclick=getQuestions;
ans=document.getElementById("submit");
ans.onclick=checkAnswer;
```

---

This script consists of the following:

- ▶ The first var statement defines a global variable, qn, which will keep track of which question is currently displayed. It is initially set to zero for the first question.
- ▶ The getQuestions() function is called when the user clicks the Start Quiz button. This function uses the AJAX library to request the contents of the questions.xml file. It sets the ajaxCallback variable to the nextQuestion() function.

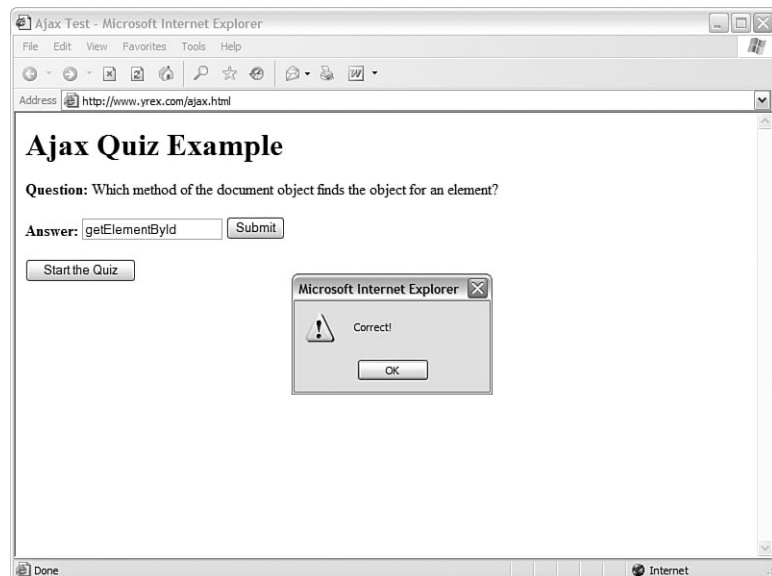
- ▶ The `nextQuestion()` function is called when the AJAX request is complete. This function uses the `getElementsByTagName()` method on the `responseXML` property to find all of the questions (`<q>` tags) and store them in the `questions` array.
- ▶ The `checkAnswer()` function is called when the user submits an answer. It uses `getElementsByTagName()` to store the answers (`<a>` tags) in the `answers` array, and then compares the answer for the current question with the user's answer and displays an alert indicating whether they were right or wrong.
- ▶ The script commands after this function set up two event handlers. One attaches the `getQuestions()` function to the Start Quiz button to set up the quiz; the other attaches the `checkAnswer()` function to the Submit button.

## Testing the Quiz

To try this example, you'll need all four files in the same folder: `ajax.js` (the AJAX library), `quiz.js` (the quiz functions), `questions.xml` (the questions), and the HTML document. All but the HTML document need to have the correct filenames so they will work correctly. Also remember that because it uses AJAX, this example requires a web server.

Figure 17.1 shows the quiz in action. The second question has just been answered.

**FIGURE 17.1**  
The quiz example as displayed by Internet Explorer.



This example should work on Internet Explorer 5–7, Mozilla 1.0 or later, any version of Firefox, or recent versions of Apple Safari. If you have trouble, try the latest Firefox.

**By the  
Way**

## Debugging AJAX Applications

Dealing with remote scripting means working with several languages at once—JavaScript, server-side languages such as PHP, XML, and of course HTML. Thus, when you find an error, it can be difficult to track down. Here are some tips for debugging AJAX applications:

- ▶ Be sure all filenames are correct, and that all files for your application are in the same folder on the server.
- ▶ If you are using a server-side language, test it without the script: Load it in the browser and make sure it works, and try passing variables to it in the URL and checking the results.
- ▶ Check the `statusText` property for the results of your request—an alert message is helpful here. It is often a clear message such as “File not found” that might explain the problem.
- ▶ If you’re using a third-party AJAX library, check its documentation—many libraries have built-in debugging features you can enable to examine what’s going on.

Hour 16, “Debugging JavaScript Applications,” includes more information on JavaScript debugging in general and includes descriptions of some useful debugging tools.

**Did you  
Know?**

### Try It Yourself



#### Making a Live Search Form

One of the most impressive demonstrations of AJAX is *live search*: Whereas a normal search form requires that you click a button and wait for a page to load to see the results, a live search displays results within the page immediately as you type in the search field. As you type letters or press the backspace key, the results are updated instantly to make it easy to find the result you need.

Using the AJAX library you created earlier, live search is not too hard to implement. This example will use a PHP program on the server to provide the search results, and can be easily adapted to any search application.

### **Watch Out!**

Once again, because it uses AJAX, this example requires a web server. You'll also need PHP version 3 or later, which is available on most servers.

## The HTML Form

The HTML for this example simply defines a search field and leaves some room for the dynamic results. The HTML document is shown in Listing 17.5.

### LISTING 17.5 The HTML File for the Live Search Example

```
<html>
<head>
<title>Live Search Ajax Example</title>
<script language="javascript" type="text/javascript"
 src="ajax.js">
</script>
</head>
<body>
<h1>Live Search: Ajax Example</h1>
<form>
<p>
Search for: <input type="text" size="40" id="searchlive">
</p>
<div id="results">
<ul id="list">
Results will display here.

</div>
</form>
<script language="javascript" type="text/javascript"
 src="search.js">
</script>
</body>
</html>
```

This HTML document includes the following:

- ▶ The `<script>` tag in the `<head>` section includes the AJAX library, `ajax.js`.
- ▶ The `<script>` tag in the `<body>` section includes the `search.js` script, which you'll create next.
- ▶ The `<input>` element with the `id` value `searchlive` is where you'll type your search query.

- The `<div>` element with the `id` value `results` will act as a container for the dynamically fetched results. A bulleted list is created with a `<ul>` tag; this will be replaced with a list of results when you start typing.

## The PHP Back End

Next, you'll need a server-side program to produce the search results. This PHP program includes a list of names stored in an array. It will respond to a JavaScript query with the names that match what the user has typed so far. The names will be returned in XML format. For example, here is the output of the PHP program when searching for "smith":

```
<names>
<name>John Smith</name>
<name>Jane Smith</name>
</names>
```

Although the list of names is stored within the PHP program here for simplicity, in a real application it would more likely be stored in a database—and this script could easily be adapted to work with a database containing thousands of names. The PHP program is shown in Listing 17.6.

### LISTING 17.6 The PHP Code for the Live Search Example

---

```
<?php
 header("Content-type: text/xml");
 $names = array (
 "John Smith", "John Jones", "Jane Smith", "Jane Tillman",
 "Abraham Lincoln", "Sally Johnson", "Kilgore Trout",
 "Bob Atkinson", "Joe Cool", "Dorothy Barnes",
 "Elizabeth Carlson", "Frank Dixon", "Gertrude East",
 "Harvey Frank", "Inigo Montoya", "Jeff Austin",
 "Lynn Arlington", "Michael Washington", "Nancy West");
 if (!$query) $query=$_GET['query'];
 echo "<?xml version='1.0' ?>\n";
 echo "<names>\n";
 while (list($k,$v)=each($names)) {
 if (striestr($v,$query))
 echo "<name>$v</name>\n";
 }
 echo "</names>\n";
?>
```

---

This hour is too small to teach you PHP, but here's a summary of how this program works:

- The header statement sends a header indicating that the output is in XML format. This is required for XMLHttpRequest to correctly use the `responseXML` property.

- ▶ The \$names array stores the list of names. You can use a much longer list of names without changing the rest of the code.
- ▶ The program looks for a GET variable called query and uses a loop to output all of the names that match the query.
- ▶ Because PHP can be embedded in an HTML file, the <?php and ?> tags indicate that the code between them should be interpreted as PHP.

### ***Did you know?***

The following books are good resources if you want to learn more on PHP quickly:

- ▶ *Sams Teach Yourself PHP in 10 Minutes*; ISBN: 0672327627
- ▶ *Sams Teach Yourself PHP in 24 Hours*; ISBN: 0672326191

Save the PHP program as search.php in the same folder as the HTML file. You can test it by typing a query such as search.php?query=John in the browser's URL field. Use the View Source command to view the XML result.

## **The JavaScript Front End**

Finally, the JavaScript for this example is shown in Listing 17.7.

### **LISTING 17.7 The JavaScript File for the Live Search Example**

```
// global variable to manage the timeout
var t;
// Start a timeout with each keypress
function StartSearch() {
 if (t) window.clearTimeout(t);
 t = window.setTimeout("LiveSearch()",200);
}
// Perform the search
function LiveSearch() {
 // assemble the PHP filename
 query = document.getElementById("searchlive").value;
 filename = "search.php?query=" + query;
 // DisplayResults will handle the Ajax response
 ajaxCallback = DisplayResults;
 // Send the Ajax request
 ajaxRequest(filename);
}
// Display search results
function DisplayResults() {
 // remove old list
 ul = document.getElementById("list");
 div = document.getElementById("results");
 div.removeChild(ul);
 // make a new list
 ul = document.createElement("UL");
 ul.id="list";
```



**LISTING 17.7 Continued**

---

```
names = ajaxreq.responseXML.getElementsByTagName("name");
for (i = 0; i < names.length; i++) {
 li = document.createElement("LI");
 name = names[i].firstChild.nodeValue;
 text = document.createTextNode(name);
 li.appendChild(text);
 ul.appendChild(li);
}
if (names.length==0) {
 li = document.createElement("LI");
 li.appendChild(document.createTextNode("No results"));
 ul.appendChild(li);
}
// display the new list
div.appendChild(ul);
}
// set up event handler
obj=document.getElementById("searchlive");
obj.onkeydown = StartSearch;
```

---

This script includes the following components:

- ▶ A global variable, `t`, is defined. This will store a pointer to the timeout used later in the script.
- ▶ The `StartSearch()` function is called when the user presses a key. This function uses `setTimeout()` to call the `LiveSearch()` function after a 200-millisecond delay. The delay is necessary so that the key the user types has time to appear in the search field.
- ▶ The `LiveSearch()` function assembles a filename that combines `search.php` with the query in the search field, and launches an AJAX request using the library's `ajaxRequest()` function.
- ▶ The `DisplayResults()` function is called when the AJAX request is complete. It deletes the bulleted list from the `<div id="results">` section, and then assembles a new list using the W3C DOM and the AJAX results. If there were no results, it displays a "No results" message in the list.
- ▶ The final lines of the script set the `StartSearch()` function up as an event handler for the `onkeydown` event of the search field.

## Making It All Work

To try this example, you'll need three files on a web server: `ajax.js` (the library), `search.js` (the search script), and the HTML file. Figure 17.2 shows this example in action.

**FIGURE 17.2**

The live search example as displayed by Firefox.



## Summary

In this hour, you've learned how AJAX, otherwise known as remote scripting, can let JavaScript communicate with a web server. You created a reusable AJAX library that can be used to create any number of AJAX applications, and you created an example using an XML file. Finally, you created a live search form using AJAX and PHP.

You've nearly reached the end of Part IV. In the next hour, you'll learn about Greasemonkey, a Firefox extension that enables you to use JavaScript to enhance sites you visit, even those created by others.

## Q&A

**Q.** *Why would I want to use POST instead of GET when making a request?*

**A.** Although GET is easy to use, it is limited to about 255 characters. If you are using a large amount of data, POST is the only way to send it to the server.

**Q.** *What happens if the server is slow, or never responds to the request?*

**A.** This is another reason you should use AJAX as an optional feature—whether caused by the server or by the user's connection, there will be times when a request is slow to respond or never responds. In this case, the callback function will be called late, or not at all. This can cause trouble with overlapping requests: for example, in the live search example, an erratic server might cause the responses for the first few characters typed to come in a few seconds apart, confusing the user. You can remedy this by checking the `readyState` property to make sure a request is not already in progress before you start another one.

- Q.** *In the live search example, why is the `onkeydown` event handler necessary? Wouldn't the `onchange` event be easier to use?*
- A.** Although `onchange` tells you when a form field has changed, it is not triggered until the user moves on to a different field—it doesn't work for "live" search, so you have to watch for key presses instead. The `onkeypress` handler would work, but in some browsers it doesn't detect the Backspace key, and it's nice to have the search update when you backspace to shorten the query.

## Quiz Questions

Test your knowledge of AJAX by answering the following questions.

- 1.** Which of the following is the A in *AJAX*?
  - a.** Advanced
  - b.** Asynchronous
  - c.** Application
- 2.** Which property of an `XMLHttpRequest` object indicates whether the request was successful?
  - a.** `status`
  - b.** `readyState`
  - c.** `success`
- 3.** Which browsers require ActiveX for remote scripting?
  - a.** Internet Explorer 5–7
  - b.** Firefox 1.0–1.5
  - c.** Internet Explorer 5–6

## Quiz Answers

1. b. AJAX stands for Asynchronous JavaScript and XML.
2. a. The `status` property indicates whether the request was successful; `readyState` indicates whether the request is complete, but does not indicate success.
3. c. Internet Explorer 5 and 6 require ActiveX. Internet Explorer 7 supports the `XMLHttpRequest` object natively.

## Exercises

If you want to gain more experience with AJAX, try the following exercises:

- ▶ Build your own XML file of questions and answers on your favorite topic and try it with the quiz example.
- ▶ Use the AJAX library to add an AJAX feature to your site, or create a simple example of your own.

## Hour 18

# Greasemonkey: Enhancing the Web with JavaScript

---

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

- ▶ How Greasemonkey and user scripts can enhance your web browser
- ▶ How to install and configure Greasemonkey in Firefox
- ▶ Installing and managing user scripts
- ▶ Creating your own user scripts
- ▶ Defining metadata for scripts
- ▶ Using the Greasemonkey API
- ▶ Adding macros to web forms

One of the recent trends is that JavaScript is being used in new ways, both inside and outside web browsers. In this hour, you'll look at Greasemonkey, a Firefox extension that enables you to write scripts to modify the appearance and behavior of sites you visit. User scripts can also work in Internet Explorer, Opera, and Safari with the right add-ons.

## **Introducing Greasemonkey**

So far in this book, you've been using JavaScript to work on your own sites. In this hour, you'll take a break from that and learn about a way to use JavaScript on *other people's sites*. Greasemonkey is an extension for the Firefox browser that enables *user scripts*. These are scripts that run as soon as you load a page and can make changes to the page's DOM.

A user script can be designed to work on all web pages, or only to affect particular sites. Here are some of the things user scripts can do:

- ▶ Change the appearance of one or more sites—colors, font size, and so on.
- ▶ Change the behavior of one or more sites with JavaScript.

- ▶ Fix a bug in a site before the site author does.
- ▶ Add a feature to your browser, such as text macros—see the Try It Yourself section of this hour for an example.

As a simple example, a user script called `Linkify` is provided with Greasemonkey. It affects all pages you visit and turns unlinked URLs into hyperlinks. In other words, the script looks for any string that resembles a URL in the page, and if it finds a URL that is not enclosed in an `<a>` tag, it modifies the DOM to add a link to the URL.

Greasemonkey scripts can range from simple ones such as `Linkify` to complex scripts that add a feature to the browser, rearrange a site to make it more usable, or eliminate annoying features of sites such as pop-up ads.

Keep in mind that Greasemonkey doesn't do anything to the websites you visit—it strictly affects your personal experience with the sites. In this way, it's similar to other browser customizations, such as personal style sheets and browser font settings.

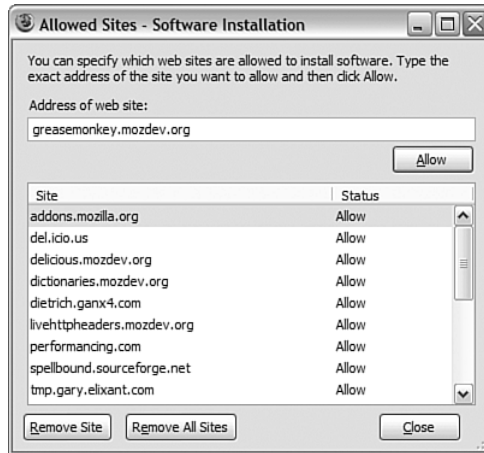
### **By the Way**

Greasemonkey was created in 2004 by Aaron Boodman. Its official site is <http://greasemonkey.mozdev.org/>. At this writing, the current version of Greasemonkey is 0.6.4. The current developers are Aaron Boodman and Jeremy Dunck.

## **Installing Greasemonkey in Firefox**

Greasemonkey works in Firefox for Windows, Macintosh, and Linux platforms. You can install it by visiting the Greasemonkey site and running the installer. Start at <http://greasemonkey.mozdev.org/> and follow these steps:

1. Click the Install Greasemonkey link.
2. You will probably see a message in a yellow bar at the top of the window warning you about installing software. Click the Edit Options button within the yellow bar.
3. In the Allowed Sites dialog, shown in Figure 18.1, click the Allow button to allow the current site to install software, and then click Close.
4. Click the Install link again, and then click the Install button in the Software Installation dialog that appears.
5. Exit and restart Firefox. You should see a small monkey icon in the lower-right hand corner of the browser window if the extension was successfully installed.



**FIGURE 18.1**  
Firefox prompts you to allow a site for installing extensions.

When you first install Greasemonkey, the extension doesn't do anything—you'll need to install one or more user scripts, as described in the next section, to make it useful.

**By the Way**

## Turnabout for Internet Explorer

Greasemonkey was written as a Firefox extension, and does not work on other browsers. Fortunately, there's an alternative for those who prefer Internet Explorer: Turnabout, from Reify, is an open-source add-on for Internet Explorer that supports user scripts. Turnabout is available for free from its official site at <http://www.reifysoft.com/turnabout.php>. Two versions are available:

- ▶ Turnabout Basic, which only supports the scripts bundled with it
- ▶ Turnabout Advanced, which supports any user script, similar to Greasemonkey

Turnabout supports most of Greasemonkey's features, and user scripts for Greasemonkey often work with Turnabout Advanced without modification. The only potential problem is with differences in JavaScript and in the DOM between Internet Explorer and Firefox. If you follow the same cross-browser coding practices you learned throughout this book, there's a good chance you can make a user script that works on both platforms.

## Other Browsers

Although Greasemonkey itself is still relatively new software, user script features have also appeared for other browsers. Along with Turnabout for IE, two other browsers can support user scripts:

- ▶ Opera, the cross-platform browser from Opera Software ASA, has built-in support for user scripts, and supports Greasemonkey scripts in many cases. See Opera's site for details at <http://www.opera.com/>.
- ▶ Creammonkey is a beta add-on for Apple's Safari browser to support user scripts. You can find it at <http://8-p.info/Creammonkey/>.

## User Script Security

Before you get into user scripting, a word of warning: Don't install a script unless you understand what it's doing, or you've obtained it from a trustworthy source. Although the Greasemonkey developers have spent a great deal of time eliminating security holes, it's still possible for a malicious script to cause you trouble—at the very least, it could send information about which sites you visit to a third-party website.

To minimize security risks, be sure you're running the latest version of Greasemonkey or Turnabout. Only enable scripts you are actively using, and limit scripts you don't trust to specific pages so they don't run on every page you visit.

## Working with User Scripts

User scripts are a whole new way of working with JavaScript—rather than uploading them for use on your website, you install them in the browser for your own personal use. The following sections show you how to find useful scripts, and install and manage them.

### Finding Scripts

Anyone can write user scripts, and many people have. Greasemonkey sponsors a directory of user scripts at <http://userscripts.org/>. There you can browse or search for scripts, or submit scripts you've written.

The script archive has thousands of scripts available. Along with general-purpose scripts, many of the scripts are designed to add features to—or remove annoying features from—particular sites.



## Installing a Script

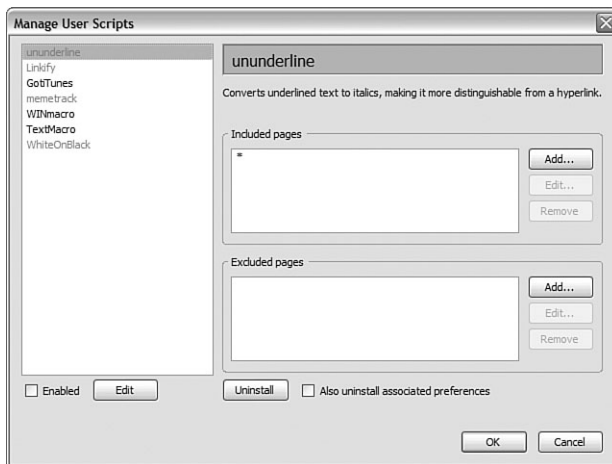
After you've found a script you wish to install, you can install it from the Web:

- ▶ In Firefox with Greasemonkey, open the script in the browser and then select Tools, Install This User Script from the menu.
- ▶ In IE with Turnabout, right-click on a link to the script and select Turnabout, Install Script.

You can also install a script from a local file. You'll use this technique to install your own script later this hour.

## Managing Scripts

After you've installed one or more scripts with Greasemonkey, you can manage them by selecting Tools, Manage User Scripts from the Firefox menu. The Manage User Scripts dialog is shown in Figure 18.2.



**FIGURE 18.2**  
Managing user  
scripts in  
Greasemonkey.

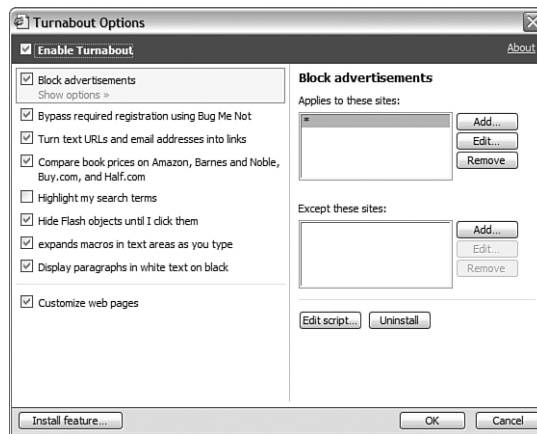
The user scripts you have available are listed in the left column. Click on a script name to manage it:

- ▶ Use the Included Pages and Excluded Pages lists to control which pages the script works on. You can specify wildcards, such as \* for all pages or \*.google.com/\* for all Google pages.
- ▶ Use the Enabled check box to enable or disable each script.

- Click the Uninstall button to remove a script.
- Click Edit to open a script in a text editor. When it is saved, it will immediately take effect on pages you load.

Turnabout for IE has a similar dialog. To access it, click the Reify button in the Turnabout toolbar and select Options. The dialog is similar to Greasemonkey's dialog, except that each script has a separate check box to enable or disable it. There is also an Install Feature button that prompts you for a new script to install. The Turnabout Options dialog is shown in Figure 18.3.

**FIGURE 18.3**  
The Turnabout  
Options dialog.



## Testing User Scripts

If you have a script enabled, it will be activated as soon as you load a page that matches one of the Included Pages specified for the script. (The script is run after the page is loaded, but before the onLoad event.) If you want to make sure Greasemonkey is running, either try one of the scripts available for download, or type in the simple script in the next section.

## Activating and Deactivating Greasemonkey or Turnabout

Sometimes you'll want to turn off Greasemonkey altogether, especially if one of the scripts you've installed is causing an error. To do this, right-click on the monkey icon in the lower-right corner of the browser window and select the Enabled option to deselect it. The monkey icon changes to a gray sad-faced monkey, and no user scripts will be run at all. You can re-enable it at any time using the same option.

With Turnabout for Internet Explorer, the procedure is similar: Click the Reify button in the Turnabout toolbar, and select the Enable Turnabout option. The icon changes to indicate that Turnabout is disabled. Choose Enable Turnabout again to re-enable it.

## Creating Your Own User Scripts

You've already learned most of what you need to know to create user scripts since they're written in JavaScript. In this section, you'll create and test a simple script, and look at some features you'll use when creating more advanced scripts.

### Creating a Simple User Script

One of the best uses for Greasemonkey is to solve annoyances with sites you visit. For example, a site might use green text on an orange background. Although you could contact the webmaster and beg for a color change, user scripting lets you deal with the problem quickly yourself.

As a simple demonstration of user scripting, you can create a user script that changes the text and background colors of paragraphs in sites you visit. Listing 18.1 shows this user script.

---

**LISTING 18.1 A Simple User Script to Change Paragraph Colors**

---

```
// Change the color of each paragraph
var zParagraphs = document.getElementsByTagName("p");
for (var i=0; i<zParagraphs.length; i++) {
 zParagraphs[i].style.backgroundColor="#000000";
 zParagraphs[i].style.color="#FFFFFF";
}
```

---

This script uses the `getElementsByTagName()` DOM method to find all of the paragraph tags in the current document and store their objects in the `zParagraphs` array. The `for` loop iterates through the array and changes the `style.color` and `style.backgroundColor` properties for each one.

### Describing a User Script

Greasemonkey supports *metadata* at the beginning of your script. These are JavaScript comments that aren't executed by the script, but provide information to Greasemonkey. To use this feature, enclose your comments between `// ==UserScript==` and `// ==/UserScript` comments.

The metadata section can contain any of the following directives. All of these are optional, but using them will make your user script easier to install and use.

- ▶ @name—A short name for the script, displayed in Greasemonkey's list of scripts after installation.
- ▶ @namespace—An optional URL for the script author's site. This is used as a namespace for the script: Two scripts can have the same name as long as the namespace is different.
- ▶ @description—A one-line description of the script's purpose.
- ▶ @include—The URL of a site on which the script should be used. You can specify any number of URLs, each in its own @include line. You can also use the wildcard \* to run the script on all sites, or a partial URL with a wildcard to run it on a group of sites.
- ▶ @exclude—The URL of a site on which the script should *not* be used. You can specify a wildcard for @include and then exclude one or more sites that the script is incompatible with. The @exclude directive can also use wildcards.

Listing 18.2 shows the color-changing example with a complete set of metadata comments added at the top.

---

**LISTING 18.2    The Color-Changing Script with Metadata Comments**

---

```
// ==UserScript==
// @name WhiteOnBlack
// @namespace http://www.jsworkshop.com/
// @description Display paragraphs in white text on black
// @include *
// ==/UserScript==
//
// Change the color of each paragraph
var zParagraphs = document.getElementsByTagName("p");
for (var i=0; i<zParagraphs.length; i++) {
 zParagraphs[i].style.backgroundColor="#000000";
 zParagraphs[i].style.color="FFFFFF";
}
```

---

## Testing Your Script

Now that you've added the metadata, installing your script is simple. Follow these steps to install the script in Firefox:

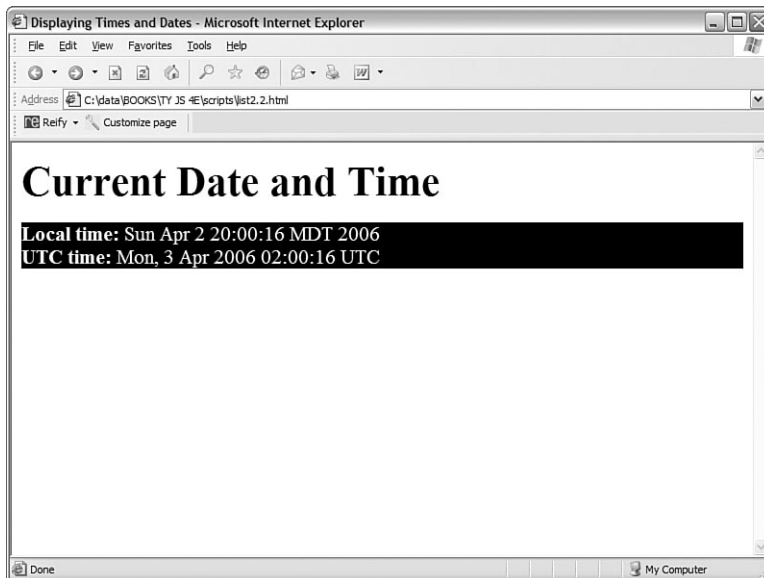
1. Save the script file as `colors.user.js`. The filename must end in `.user.js` to be recognized as a Greasemonkey script.
2. In Firefox, choose File, Open from the menu.
3. Select your script from the Open File dialog.

4. After the script is displayed in the browser, select Tools, Install This User Script.
5. An alert will display to inform you that the installation was successful. The new user script is now running on all sites.

If you're using Turnabout under Internet Explorer, click on the Turnabout toolbar and select Options, and then click the Install Feature button. Select the script and click Open to install it.

Both Greasemonkey and Turnabout for IE will use the metadata you specified to set the script's included pages, description, and other options when you install it.

After you've installed and enabled the script, any page you load will have its paragraphs displayed in white text on a black background. For example, Figure 18.4 shows the user script's effect on the Date and Time example from Hour 2, "Creating Simple Scripts." Because the date and time are within `<p>` tags, they are displayed in white on black.



**FIGURE 18.4**  
The Date and Time example altered by the color-changing user script.

You probably don't want to make a change this drastic to *all* sites you visit. Instead, you can use `@include` to make this script affect only one or two sites whose colors you find hard to read. Don't forget that you can also change the colors in the script to your own preference.

**By the  
Way**

## Greasemonkey API Functions

You can use all of the DOM methods covered in this book to work with pages in user scripts, along with JavaScript's built-in functions. In addition to these, Greasemonkey defines an API (Application Programmer's Interface) with a few functions that can be used exclusively in user scripts:

- ▶ `GM_log(message, level)`—Inserts a message into the JavaScript console. The `level` parameter indicates the severity of the message: 0 for information, 1 for a warning, and 2 for an error.
- ▶ `GM_setValue(variable, value)`—Sets a variable stored by Greasemonkey. These variables are stored on the local machine. They are specific to the script that set them, and can be used in the future by the same script. (These are similar to cookies, but are not sent to a server.)
- ▶ `GM_getValue(variable)`—Retrieves a value previously set with `GM_setValue`.
- ▶ `GM_registerMenuCommand(command, function)`—Adds a command to the browser menu. These commands appear under Tools, User Script Commands. The `command` parameter is the name listed in the menu, and `function` is a function in your script that the menu selection will activate.
- ▶ `GM_xmlHttpRequest(details)`—Requests a file from a remote server, similar to the AJAX features described in Hour 17, "AJAX: Remote Scripting." The `details` parameter is an object that can contain a number of properties to control the request. See the Greasemonkey documentation for all of the properties you can specify.

Turnabout for Internet Explorer also supports all of these API functions, so aside from the usual browser differences, scripts that use these functions should work in both browsers. Because Internet Explorer does not have a JavaScript console, Turnabout includes its own console, available from the menu, where log messages are displayed.

## Creating a Site-Specific Script

You might want to use a user script to fix a problem or add a feature to a specific site. In addition to using `@include` to specify the site's URL, you'll need to know something about the site's DOM.

You can use the DOM Inspector in Firefox (or the similar feature in Internet Explorer's developer toolbar) to browse the DOM for the site and find the objects you want to work with. Depending on how they are marked up, you can access them through the DOM:

- If an element has an id attribute, you can simply use `document.getElementById()` in your script to find its object.
- If a nearby element has an id defined, you can use DOM methods to find it—for example, if the parent element has an id, you can use a method such as `firstChild()` to find the object you need.
- If all else fails, you can use `document.getElementsByTagName()` to find all objects of a certain type—for example, all paragraphs. If you need to refer to a specific one, you can use a loop and check each one for a certain attribute.

See Hour 16, “Debugging JavaScript Applications,” for information about Firefox’s DOM Inspector and IE’s developer toolbar. See Hours 13, “Using the W3C DOM” and 14, “Using Advanced DOM Features,” for information about DOM methods.

**By the  
Way**

As an example, Listing 18.3 shows a simple user script you could use as a site-specific script to automatically fill out certain fields in forms.

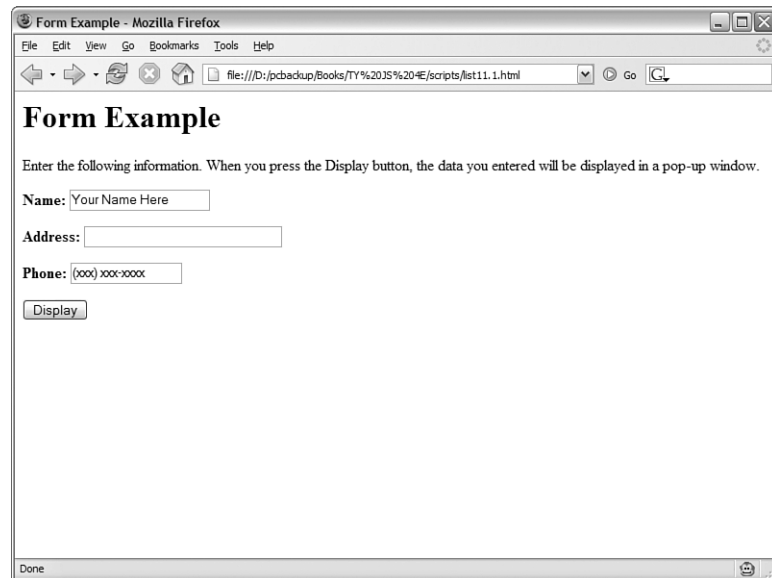
### LISTING 18.3 A User Script to Fill Out Form Fields Automatically

```
// ==UserScript==
// @name AutoForm
// @namespace http://www.jsworkshop.com/
// @description Fills in forms automatically
// @include *
// ==/UserScript==
// this function fills out form fields
//
var zTextFields = document.getElementsByTagName("input");
for (var i=0; i<zTextFields.length; i++) {
 thefield=zTextFields[i].name;
 if (!thefield) thefield=zTextFields[i].id;
 // Set up your auto-fill values here
 if (thefield == "yourname") zTextFields[i].value="Your Name Here";
 if (thefield == "phone") zTextFields[i].value="(xxx) xxx-xxxx";
 alert("field: " + thefield + " value: " + zTextFields[i].value);
}
```

This script uses `getElementsByTagName()` to find all of the `<input>` elements in a document, including text fields. It uses a `for` loop to examine each one. If it finds a field with the name or id value “yourname” or “phone”, it inserts the appropriate value.

To test this script, save it as `autoform.user.js` and install the user script as described earlier in this hour. To test it, load Listing 11.1 from Hour 11, “Getting Data with Forms,” into the browser—it happens to have both of the field names the script looks for. The `yourname` and `phone` fields will be automatically filled out, as shown in Figure 18.5.

**FIGURE 18.5**  
The form-filling  
user script in  
action.



To make it easy to test, Listing 18.3 doesn't include specific sites in the `@include` line. To make a true site-specific script, you would need to find out the field names for a particular site, add `if` statements to the script to fill them out, and use `@include` to make sure the script only runs on the site.

## Debugging User Scripts

Debugging a user script is much like debugging a regular JavaScript program—errors are displayed in the JavaScript Console in Firefox or in an error message in Internet Explorer. Here are a few debugging tips:

- ▶ As with regular scripts, you can also use the `alert()` method to display information about what's going on in your script.
- ▶ The browser may display a line number with an error message, but when you're working with user scripts, these line numbers are meaningless—they refer neither to lines in your user script nor to the page you're currently viewing.
- ▶ Use the `GM_log()` method described earlier in this hour to log information about your script, such as the contents of variables, to the JavaScript console.
- ▶ If you're trying to write a cross-browser user script, watch for methods that are browser specific. See Hour 15, "Unobtrusive Scripting," for information about cross-browser issues.



- ▶ Watch for conflicts with any existing scripts on the page.
- ▶ If you're using multiple user scripts, be sure they don't conflict. Use unique variable and function names in your scripts.

Most of the issues with user scripts are the same as for regular JavaScript. See Hour 16 for information on debugging tools, techniques, and common mistakes.

## Try It Yourself



### Creating a User Script

Now that you've learned the basics of Greasemonkey, you can try a more complex—and more useful—example of a user script.

If you spend much time on the Web, you'll find yourself needing to fill out web forms often, and you probably type certain things—such as your name or URL—into forms over and over. The user script you create here will let you define macros for use in any text area. When you type a macro keyword (a period followed by a code) and then type another character, the macro keyword will be instantly replaced by the text you've defined. For example, you can define a macro so that every time you type `.cu`, it will expand into the text "See you later."

This script has been tested on Greasemonkey 0.6.4 for Firefox and Turnabout Advanced 0.31 b3 for Internet Explorer. Because browsers and extensions are always changing, it might stop working at some point—see this book's website for the latest updates.

**Watch  
Out!**

Listing 18.4 shows the text area macro user script.

#### LISTING 18.4 The Text Area Macro User Script

```
// ==UserScript==
// @name TextMacro
// @namespace http://www.jsworkshop.com/
// @description expands macros in text areas as you type
// @include *
// ==/UserScript==
// this function handles the macro replacements
function textmacro(e) {
 // define your macros here
 zmacros = [
 [".mm", "Michael Moncur"],
 [".js", "JavaScript"],
 [".cu", "See you later."]
];
 if (!e) var e = window.event;
```

**LISTING 18.4** Continued

---

```

// which textarea are we in?
thisarea= (e.target) ? e.target : e.srcElement;
// replace text
for (i=0; i<zmacros.length; i++) {
 vv = thisarea.value;
 vv = vv.replace(zmacros[i][0],zmacros[i][1]);
 thisarea.value=vv;
}
}
// install the event handlers
var zTextAreas = document.getElementsByTagName("textarea");
for (var i=0; i<zTextAreas.length; i++) {
 if (zTextAreas[i].addEventListener)
 zTextAreas[i].addEventListener("keydown",textmacro,0);
 else if (zTextAreas[i].attachEvent)
 zTextAreas[i].attachEvent("onkeydown",textmacro);
}

```

---

**How It Works**

This user script begins with the usual comment metadata. The `@include` command specifies a wildcard, `*`, so the script will work on all sites. The actual work is done in the `textmacro()` function. This function begins by defining the macros that will be available:

```

zmacros = [
 [".mm", "Michael Moncur"],
 [".js", "JavaScript"],
 [".cu", "See you later."]
];

```

This example defines three macros using a two-dimensional array. To make the script useful to you, define your own. You can have any number of macros—just add a comma after the last macro line and add your items before the closing bracket.

Next, the function uses the `target` property to find the text area in which you're currently typing. Next, it uses a `for` loop to do a search and replace within the text area's value property for each of your macros.

The section of code after the `textmacro()` function sets up an event handler for each text area. First, it uses `getElementsByTagName()` to find all of the text areas, and then it uses a `for` loop to add an `onkeydown` event handler to each one.

To avoid conflicts with existing event handlers within web pages, this example uses the `addEventListener()` method to add the event handler. This method defines an event handler without overwriting existing events. In Internet Explorer, it uses the similar `attachEvent()` method. See Hour 15 for more information.

## Using This Script

To use this script, first make sure you've installed and enabled Greasemonkey as described earlier this hour. Save the script as `textmacro.user.js`. You can then install the user script.

After the script is installed, try loading any page with a text area. You should be able to type a macro, such as `.mm` or `.js`, followed by another character such as a space, within the text area and see it instantly expand into the correct text.

This script runs on all sites by default. If you only want the macros to work on certain sites, you can change the `@include` directive to specify them. If the script causes trouble on some sites, you can exclude them with `@exclude`.

***Did you  
Know?***



---

## Summary

In this hour, you've learned how to use Greasemonkey—and its counterpart for Internet Explorer, Turnabout—to enable user scripting in your browser. You've learned how user scripts work and how to install and manage them. Finally, you created two examples of functioning user scripts.

Congratulations! You've reached the end of Part IV of this book, and you're well on your way to becoming a JavaScript expert. In Part V, you'll look at multimedia applications of JavaScript—graphics, animation, sound, and working with plug-ins. Hour 19, "Using Graphics and Animation," starts by helping you move beyond scripts that work with text.

## Q&A

**Q.** *Is there any way to prevent users from using Greasemonkey while viewing my site?*

**A.** Because Greasemonkey only affects the user who installed it, it's usually harmless to allow it. If you still want to prevent its use, this is difficult but not impossible, and varies with different versions of Greasemonkey. Search the Web to find current solutions.

**Q.** *What if I want to do something more sophisticated, such as modifying Firefox's menu?*

**A.** This capability does not exist in Greasemonkey, but Firefox extensions are also written in JavaScript. In fact, you can compile a user script into a Firefox extension and then add more advanced features. See <http://www.letitblog.com/greasemonkey-compiler/> for details.

- Q.** *What happens when a new version of Firefox or Internet Explorer is released?*
- A.** Although I have faith in the Greasemonkey developers, there's no guarantee that this extension will work in future browser versions. If you're concerned about this, you might want to write your own Firefox extension instead.
- Q.** *Are there limits to how much I can modify a page using Greasemonkey?*
- A.** No—in fact, you can yank the entire content of the page's DOM out and replace it with HTML of your choosing using the `innerHTML` property. You'd have to do quite a bit of work to make something as useful as the original page, of course.

## Quiz Questions

Test your knowledge of Greasemonkey and user scripts by answering the following questions.

- 1.** Which of the following offers user scripting for Microsoft Internet Explorer?
  - a.** Greasemonkey
  - b.** Microsoft Live Scripting Toolbar
  - c.** Turnabout
- 2.** Which of the following is not a valid Greasemonkey API function?
  - a.** `GM_log()`
  - b.** `GM_alert()`
  - c.** `GM_setValue()`
- 3.** Which is the correct `@include` directive to run a script on both `www.google.com` and `google.com`?
  - a.** `@include *.google.com`
  - b.** `@include www.google.com.*`
  - c.** `@include google.com`

## Quiz Answers

1. c. Turnabout is a user script add-on for Internet Explorer.
2. b. There is no `GM_alert()` method, although the standard `alert()` method will work in a user script.
3. a. Using `@include *.google.com` will run the script on any page on any site within the `google.com` domain.

## Exercises

If you want to gain more experience with user scripts, try the following exercises:

- ▶ Modify the color-changing user script in Listing 18.2 to use different colors, and add another style attribute—for example, use `style.fontSize` to change the font size.
- ▶ The color-changing example works on paragraphs, but text often appears in other places, such as bullet lists. Modify Listing 18.2 to make the changes to `<li>` tags as well as paragraphs.
- ▶ Currently, the macro example in the Try It Yourself section only works on text inputs that use `<textarea>` tags. Modify the script in Listing 18.3 to work on `<input>` tags also. (You'll need to add a second call to `getElementsByTagName()` and a loop to add the event handlers.)

*This page intentionally left blank*

---

## **PART V:**

# **Building Multimedia Applications with JavaScript**

<b>HOURL 19</b>	Using Graphics and Animation	<b>313</b>
<b>HOURL 20</b>	Working with Sound and Plug-ins	<b>329</b>

*This page intentionally left blank*



## Hour 19

# Using Graphics and Animation

---

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

- ▶ Using JavaScript to swap images within a page
- ▶ Using JavaScript rollovers
- ▶ Using CSS rollovers
- ▶ Creating an image slideshow
- ▶ Adding animation to the slideshow

Welcome to Part V! So far, you've used JavaScript to work with text and forms in web pages. In the next two hours, you'll look at how JavaScript can work with graphics, sounds, and plug-ins. This hour focuses on using JavaScript to manipulate graphics and create animated displays.

## **Using Dynamic Images**

Long before the W3C DOM allowed JavaScript to change any part of a web page, a feature called *dynamic images* enabled you to swap one image for another with JavaScript. This technique is still supported by current browsers, and is still the most convenient (and compatible) way to work with images in JavaScript.

### **Working with image Objects**

You can change images dynamically by using the `image` object associated with each one. The traditional way to do this is with the `document.images` array. This array contains an item for each of the images defined on the page. In the object hierarchy, each `image` object is a child of the `document` object.

With the W3C DOM, you can also assign an `id` attribute to an image within the `<img>` tag, and then use `document.getElementById` to find the object for that image. Each `image` object has the following properties:

- ▶ `complete` is a flag that tells you whether the image has been completely loaded. This is a Boolean value (true or false).
- ▶ `height` and `width` reflect the corresponding image attributes. This is for information only; you can't change an image's size dynamically.
- ▶ `hspace` and `vspace` represent the corresponding image attributes, which define the image's placement on the page. Again, this is a read-only attribute.
- ▶ `name` is the image's name. You can define this with the `NAME` attribute in the image definition.
- ▶ `src` is the image's source, or URL. You can change this value to change images dynamically.

For most purposes, the `src` attribute is the only one you'll use. The `image` object has no methods. It does have three event handlers you can use:

- ▶ The `onLoad` event occurs when the image finishes loading. (Because the `onLoad` event for the entire document is triggered when all images have finished loading, it's usually a better choice.)
- ▶ The `onAbort` event occurs if the user aborts the page before the image is loaded.
- ▶ The `onError` event occurs if the image file is not found or corrupt.

### **By the Way**

Although changing image sources works fine, you can also use the W3C DOM to completely remove or replace image objects, or insert new ones, just like any other object.

## **Preloading Images**

You can also create an independent `image` object. This enables you to specify an image that will be loaded and placed in the cache, but will not be displayed on the page.

This might sound useless, but it's a great way to work with modem-speed connections. After you've preloaded an image, you can replace any of the images on the page with that image—and because it's already cached, the change happens instantly. Even on a fast connection, this avoids flickering and makes animation smoother.

You can cache an image by creating a new `Image` object, using the new keyword.

Here's an example:

```
Image2 = new Image();
Image2.src = "arrow1.gif";
```

You learned about the new keyword and its other uses for object-oriented programming in Hour 6, “Using Functions and Objects.”

**By the  
Way**

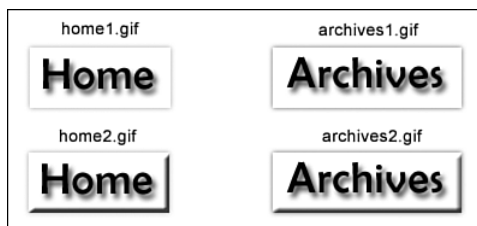
## Creating Rollovers

One of the classic uses of JavaScript is to create *rollovers*—images that change when you move the mouse over them. They are typically used to create navigation links that give the user a bit of guidance by highlighting the one the mouse is over.

In this section, you'll learn how to use JavaScript's dynamic images to create rollovers—and then you'll learn why you shouldn't do this most of the time, and how to create rollovers with no scripting at all.

### JavaScript Rollovers

First, let's take a quick look at how to create rollovers using JavaScript. To do this, you start with regular and highlighted versions of each rollover image. Figure 19.1 shows two examples of navigation buttons in both states.



**FIGURE 19.1**  
Regular and highlighted versions of two button images.

As you might guess, all this requires in JavaScript is to combine an `onMouseOver` event handler with a dynamic image. Adding `onMouseOut` allows your script to restore the original image when the mouse moves away. Listing 19.1 shows a simple way to do this with inline event handlers.

**LISTING 19.1** Using Basic JavaScript Rollovers

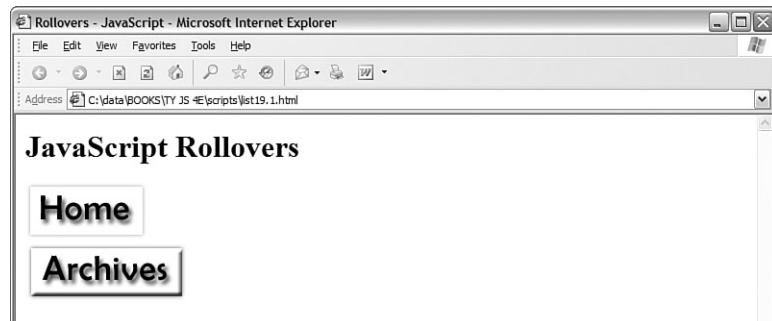
```
<html>
<head>
<title>Rollovers - JavaScript</title>
</head>
<body>
<h1>JavaScript Rollovers</h1>
<a href="home.html"
 onmouseover="document.images[0].src='home2.gif';"
 onmouseout="document.images[0].src='home1.gif';">
 <image border="0" src="home1.gif">

<a href="archives.html"
 onmouseover="document.images[1].src='archives2.gif';"
 onmouseout="document.images[1].src='archives1.gif';">
 <image border="0" src="archives1.gif">

</body>
</html>
```

This is just a basic bit of inline JavaScript, so you can test it by simply loading the HTML file into a browser. The results are shown in Figure 19.2. In the figure, the mouse cursor is over the Archives button.

**FIGURE 19.2**  
The rollover  
example in  
action.



## CSS Rollovers Without JavaScript

Although JavaScript rollovers work fine in today's browsers, the technique was developed in the days before CSS, and there is now a better way to accomplish the same thing. Using the `:hover` directive in CSS, you can create text links that change color when the mouse hovers over them. Listing 19.2 shows a simple example of CSS rollovers.

**LISTING 19.2** JavaScript-Free Rollovers with CSS

```
<html>
<head>
<title>Rollovers - CSS</title>
<style>
#home,#archives {
 font-size: 30px;
 text-decoration: none;
}
#home:hover, #archives:hover {
 background-color: #AAAAAA;
}
</style>
</head>
<body>
<h1>JavaScript-Free Rollovers</h1>
Home

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

To try this example, simply load the HTML document into a browser. When you move the mouse over the links, their background color changes from white to gray. This example is shown in Figure 19.3.



**FIGURE 19.3**  
Simple CSS-only  
rollovers.

This isn't as fancy as the JavaScript rollovers, but it has some advantages—first of all, it doesn't require JavaScript. Second, the links are actual text—this means they'll work even in text-based browsers, primitive mobile phone browsers, and voice-reading browsers for the blind, although the rollover effects won't work in these situations. Third, the page loads faster, and you can add more links without creating graphics.

## Graphic CSS Rollovers

Suppose you're really attached to the nifty graphic look of the first rollover example. Before you do something like that, take a look at Listing 19.3. This listing uses CSS to implement graphic rollovers, which look and work exactly like Figure 19.2, with no JavaScript.

**LISTING 19.3 Graphic Rollovers with CSS**


---

```

<html>
<head>
<title>Rollovers - CSS</title>
<style>
#home {
 display: block;
 height: 60px;
 width: 126px;
 background-image: url("home1.gif");
}
#home:hover {
 background-image: url("home2.gif");
}
#archives {
 display: block;
 height: 60px;
 width: 168px;
 background-image: url("archives1.gif");
}
#archives:hover {
 background-image: url("archives2.gif");
}
#home b, #archives b {
 display: none;
}
</style>
</head>
<body>
<h1>JavaScript-Free Rollovers</h1>
Home

Archives
</body>
</html>

```

---

Here's a summary of how the CSS works:

- ▶ The `#home` and `#archives` rules, which match the `id` attribute of the two links, set their `display` attribute to `block` and the width and height attributes to allow the links to be as large as their corresponding graphics. They then use the `background-image` property to display the unhighlighted graphics (`home1.gif` and `archives1.gif`).
- ▶ The `#home:hover` and `#archives:hover` rules change the background images to the highlighted versions (`home2.gif` and `archives2.gif`).
- ▶ The `#home b` and `#archives b` rule hides the text of the links within the `<b>` tags. This prevents the text from appearing on top of the graphics.

Notice that the HTML portion of this example is identical to the previous example, and it will work exactly the same on text-based browsers and browsers with JavaScript turned off. Users with modern browsers will see the graphic versions of the links instead. This gives you the look of graphic rollovers without JavaScript, and without compromising accessibility.

Another reason to use this type of rollover: Because the links are still in the HTML as text, search engines see them as ordinary links, and can do a better job of indexing your site. See Hour 15, “Unobtrusive Scripting,” for more information on accessibility and search engine optimization.

**By the  
Way**

## A Simple JavaScript Slideshow

Suppose you wanted to create a simple picture slideshow using JavaScript: The page displays the first picture, and when you click on it the next picture replaces it. You can continue to click and view all of the pictures in the slideshow. The obvious way to do this is to change the `.src` attribute of an `image` object, and that will work fine—but here you’ll take a look at a different approach that uses the W3C DOM to make a more flexible slideshow.

### The HTML File

First, you’ll need an HTML document that defines the page and where the images will appear. Before you start on the scripting, take a look at the HTML file in Listing 19.4.

#### LISTING 19.4 The HTML File for the Slideshow

```
<html>
<head>
 <title>Image Slideshow Test</title>
 <script language="javascript" type="text/javascript"
 src="slideshow.js">
 </script>
</head>
<body>
 <h1>Image Slideshow Test</h1>

 <p>Click the image to view the next slide.</p>
</body>
</html>
```

You might notice something peculiar about this document: All five of the images are included with `<img>` tags. If you load the document into a browser before you add the JavaScript file, you'll see all five images on the page at once.

The `slideshow.js` script is included with the `<script>` tag in the header. This script will hide all but the first image, and allow the images to be shown one at a time. This is an example of *unobtrusive scripting*—users without JavaScript can see the images just fine, although they'll have to scroll the page, and they'll miss the nifty slideshow feature.

Because the markup of this example uses ordinary `<img>` tags, we've used a special `class="slide"` attribute on the slide images. The script will check for this class to determine which images belong to the slideshow because there's a good chance you'll have other images on the page.

This is a flexible way to create a slideshow—you can change the order of the slides simply by rearranging the HTML, and you can add more slides just by adding more images with the right class value.

## By the Way

See Hour 15 for more information about keeping JavaScript unobtrusive and optional.

## The JavaScript File

The script that brings the slideshow to life is shown in Listing 19.5. The script consists of two basic functions: `MakeSlideShow()`, which rearranges the images into a slideshow, and `NextSlide()`, which responds to a click and advances to the next image.

### LISTING 19.5 The JavaScript File for the Slideshow

```
var numslides=0,currentslide=0;
var slides = new Array();
function MakeSlideShow() {
 // find all images with the class "slide"
 imgs=document.getElementsByTagName("img");
 for (i=0; i<imgs.length; i++) {
 if (imgs[i].className != "slide") continue;
 slides[numslides]=imgs[i];
 // hide all but first image
 if (numslides==0) {
 imgs[i].style.display="block";
 } else {
 imgs[i].style.display="none";
 }
 imgs[i].onclick=NextSlide;
 numslides++;
 } // end for loop
}
```



**LISTING 19.5** Continued

---

```
} // end MakeSlideshow()
function NextSlide() {
 slides[currentslide].style.display="none";
 currentslide++;
 if (currentslide >= numslides) currentslide = 0;
 slides[currentslide].style.display="block";
}
// create the slideshow when the page loads
window.onload=MakeSlideshow;
```

---

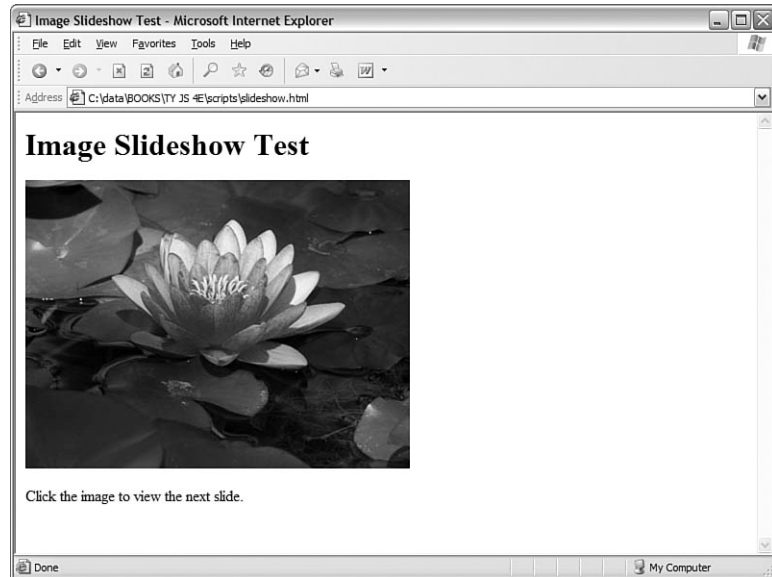
Let's take a look at how this script works:

- ▶ The first lines define three global variables: `numslides` to store the current number of slides, `currentslide` to keep track of the current slide, and the `slides` array to store the image objects for each slide.
- ▶ The `MakeSlides()` function starts by using `getElementsByTagName()` to find all of the images on the page, and iterates through the array with a `for` loop. The first `if` statement in the loop checks the `className` attribute of the image, and if it does not belong to the `slide` class, the loop is continued without any action.
- ▶ The next statements store the image in the `slides` array, and then check `numslides` for a value of zero, meaning the first image in the array. For the first image, the `display` attribute is set to `block`; for all others, `display` is set to `none` so that only one image is visible at a time.
- ▶ The final statements in the loop set the image's `onclick` event handler to the `NextSlide()` function and increment the `numslides` variable.
- ▶ The `NextSlide()` function first hides the current slide by setting its `display` property to `none`. Next, it increments `currentslide`. The `if` statement resets `currentslide` to zero when the last slide is clicked on. Finally, the new slide is displayed by setting its `display` property to `block`.
- ▶ The final line of the script sets an `onLoad` event handler for the window to run the `MakeSlideshow()` function. This rearranges the images into a slideshow as soon as the page loads.

To test the script, save it as `slideshow.js` in the same folder as the HTML document you created previously, and load the HTML document into a browser. Figure 19.4 shows the script in action with the first image displayed.

**FIGURE 19.4**

The JavaScript slideshow shows the first image.



### ***Did you know?***

You might see a brief flicker when you load the page and the five images display before being hidden by the script. You can eliminate this by adding a `display:none` rule in CSS for the `slide` class, making all of the images invisible until the script displays the first one.



## **Try It Yourself**

### **Adding Animation to the Slideshow**

Although the slideshow example works, the transitions between images are instantaneous—somewhat of a utilitarian effect. With a bit more code, you can use JavaScript and the CSS positioning properties to create an animated transition between the slides.

### ***By the way***

See Hour 13, “Using the W3C DOM,” for information about the CSS positioning properties used in this example.

## **The HTML File**

The HTML for this example is similar to that of the basic slideshow, with two differences: First, the images are enclosed in a `<div>` element with the `id` attribute “slideshow”. This element will be used to make the transition between slides work.

Second, a `<link>` tag in the header specifies a style sheet, `slideshow2.css`, because this example will require some CSS styles. The HTML document is shown in Listing 19.6.

### LISTING 19.6 The HTML File for the Animated Slideshow

---

```
<html>
<head>
 <title>Animated Slideshow Test</title>
 <script language="javascript" type="text/javascript"
 src="slideshow2.js">
 </script>
 <link rel="stylesheet" type="text/css" href="slideshow2.css">
</head>
<body>
 <h1>Animated Slideshow Test</h1>
 <div id="slideshow">

 </div>
 <p>Click the image to view the next slide.</p>
</body>
</html>
```

---

As before, if you load this document into a browser without the JavaScript or CSS files, it will display all five images on the page.

## The CSS File

You'll need a bit of CSS to set things up for the slideshow. The style sheet will set the initial position of the images and set the positioning properties so that the animation will work. The CSS file for this example is shown in Listing 19.7. Save the file as `slideshow2.css` in the same folder as the HTML document you created previously.

### LISTING 19.7 The CSS File for the Animated Slideshow

---

```
img.slide {
 position: absolute;
 left:0;
 top:0;
}
#slideshow {
 position: relative;
 overflow: hidden;
 width: 400;
 height: 300;
}
```

---

The `#slideshow` rule defines the styles for the `<div>` element that encloses the images. The `position: relative` rule enables positioning for the element and its children, while leaving it where it landed in the page by default. The `overflow` property hides the part of an image that lies outside the `<div>`, so the new image can “slide in” from the side. Finally, the `width` and `height` properties make the `<div>` as large as the images so that the slideshow is always one size.

The `img.slide` rule sets up the styles for the images themselves. The `position` property is set to `absolute`. In combination with the `relative` value on the `<div>`, this means that the image is positioned relative to its parent. It is set to `left: 0` and `top: 0`, which positions each image at the upper-left corner of the `<div>`—to begin, all of the images will be on top of each other, so only one will be visible.

Instead of using the `display` property, the animated slideshow will use the `z-index` property (`zIndex` in JavaScript). This controls which of the overlapping images is “on top.” To change slides, the script will set the new image to be on the top of the stack and position it off the right edge of the `<div>`, and then gradually slide both the old and new slides to the left until the new one is the only one visible.

## The JavaScript File

Now that you have the HTML and CSS files, all that remains is the script. Listing 19.8 shows the JavaScript file for the animated slideshow.

### LISTING 19.8 The JavaScript File for the Animated Slideshow

---

```
// Global variables
var numslides=0;
var currentslide=0,oldslide=4;
var x = 0;
var slides = new Array();
function MakeSlideShow() {
 // find all images with the class "slide"
 imgs=document.getElementsByTagName("img");
 for (i=0; i<imgs.length; i++) {
 if (imgs[i].className != "slide") continue;
 slides[numslides]=imgs[i];
 // stack images with first slide on top
 if (numslides==0) {
 imgs[i].style.zIndex=10;
 } else {
 imgs[i].style.zIndex=0;
 }
 imgs[i].onclick=NextSlide;
 numslides++;
 } // end for loop
} // end MakeSlideShow()
function NextSlide() {
 // Set current slide to be under the new top slide
 slides[currentslide].style.zIndex=9;
```

**LISTING 19.8** Continued

---

```
// Move older slide to the bottom of the stack
slides[oldslide].style.zIndex=0;
oldslide = currentslide;
currentslide++;
if (currentslide >= numslides) currentslide = 0;
// start at the right edge
slides[currentslide].style.left=400;
x=400;
// Move the new slide to the top
slides[currentslide].style.zIndex=10;
AnimateSlide();
}
function AnimateSlide() {
// Lower moves slower, higher moves faster
x = x - 25;
slides[currentslide].style.left=x;
// previous image moves off the left edge
// (comment the next line for a different effect)
slides[oldslide].style.left=x-400;
// repeat until slide is at zero position
if (x > 0) window.setTimeout("AnimateSlide();",10);
}
// create the slideshow when the page loads
window.onload=MakeSlideShow;
```

---

Here's how this script differs from the original slideshow script:

- ▶ An `oldslide` global variable has been added to keep track of the previous slide, so it can be moved out as the new slide moves in. Another global variable, `x`, will store the current horizontal position of the sliding image.
- ▶ Instead of using the `display` property, the `MakeSlideShow()` function sets the `zIndex` property to 10 for the first image and to zero for the others.
- ▶ The `NextSlide()` function works differently. First, it sets the current slide's `zIndex` property to 9, so it is the second one in the stack. (See the Did You Know? sidebar at the end of this section for the reason.) Next, it sets `zIndex` to zero for the old slide to move it to the bottom. It then assigns the `oldslide` value for next time, and increments the current slide as before.
- ▶ `NextSlide()` finishes by setting the new slide's `left` property to 400, and the `x` variable to the same value. The slide will start off the right edge of the `<div>` and gradually become visible as it moves to the left. It then sets `zIndex` to 10 for the new slide to put it on top of the stack. Last, it calls the new `AnimateSlide()` function to make the transition.
- ▶ `AnimateSlide()` handles the animation. It starts by subtracting 25 from the value of `x` and setting the current slide's `left` property to that value. It also sets the position of the old slide 400 pixels to the left of the current one, so it slides out of the frame as the new one slides in.

- The last line in `AnimateSlide()` checks `x`, and if it has not yet reached zero, it uses `setTimeout()` to call itself after a brief (10 millisecond) delay. This function will be called repeatedly until the new slide reaches its final resting place on the left side.

### ***Did you Know?***

The reason for setting the old slide's `zIndex` to 9 instead of 10 is to allow you to try a different transition effect. If you remove the `slides[oldstyle].style.left` assignment in `AnimateSlide()`, the old slide will stay in one place while the new slide moves over it.

## Putting It All Together

To try out the animated slideshow, make sure you have all three files in the same folder: the HTML document, the style sheet (`slideshow2.css`), and the JavaScript file (`slideshow2.js`). Load the HTML document into a browser; then click on the image to advance the slideshow.

The `AnimateSlide()` function uses a lot of code, but on a reasonably fast machine, the transition will be very fast, taking about half a second. If you want to slow it down to see what's going on, change the 25 value in `AnimateSlide()` to a lower number—a value of 1 will make the transition extremely slow. Figure 19.5 shows the slideshow in action, halfway between the first slide and the second.

**FIGURE 19.5**  
The animated  
slideshow in  
action.



## Summary

In this hour, you learned some techniques for working with graphics in JavaScript. You learned how to use dynamic images to create rollovers, and how to use CSS for JavaScript-free rollovers. Finally, you created a script to turn any group of images on a page into an animated slideshow.

In the next hour, you'll look at how JavaScript works with plug-ins, particularly Flash, and learn how to add sounds to your scripts.

## Q&A

**Q. *Isn't it possible to make JavaScript rollovers unobtrusive?***

**A.** Yes, you could use a separate JavaScript file and do JavaScript rollovers “the right way.” You would still have image links instead of text links, but aside from that it's arguably no worse than CSS rollovers. JavaScript rollovers can also go beyond what CSS can do—for example, the links could change in an animated way rather than simply changing graphics.

**Q. *Can JavaScript work with any type of image?***

**A.** Yes, JavaScript's dynamic image features (and the W3C DOM features you used in the slideshow) will work fine with GIF, JPG, and PNG (Portable Network Graphics, the newest standard) images.

**Q. *Why doesn't the slideshow example require preloading images for fast transitions?***

**A.** Because the images are all on the page in the HTML document, they are loaded with the page, although the JavaScript immediately hides them. Thus, they're available instantly when the slideshow switches them.

**Q. *How do I speed up the transitions in the animated slideshow?***

**A.** There are two ways: Either increase the amount subtracted from `x`, or reduce the timeout in the `setTimeout` statement. Subtracting too much can make the transition jerky, and timeouts below 10 aren't handled well by browsers, so experiment with your changes to reach the best compromise.

## Quiz Questions

Test your knowledge of JavaScript graphics by answering the following questions.

1. Which property of an `image` object stores the filename of the image?
  - a. `href`
  - b. `filename`
  - c. `src`
2. Which of the following languages *cannot* be used to implement rollovers?
  - a. HTML
  - b. CSS
  - c. JavaScript
3. If `image1` is the object for an image on the page, which of the following would you modify to change the image's horizontal position?
  - a. `image1.left`
  - b. `image1.style.left`
  - c. `image1.style.xPosition`

## Quiz Answers

1. c. The `src` property of the image stores its filename.
2. a. You can create rollovers using JavaScript or CSS, but it can't be done in plain HTML.
3. b. The `style.left` property controls the image's horizontal position.

## Exercises

If you want to gain more experience working with graphics in JavaScript, try the following exercises:

- ▶ Change the animated slideshow example to move the slides downward instead of right to left to make the transition. You'll need to change the `style.top` property instead of `style.left`.
- ▶ Firefox and some other browsers support a CSS 3 property, `style.opacity`, which controls how opaque an element is, with a value of 100 being completely opaque and a value of 0 being completely transparent. Try changing the animated slideshow to fade the new slide in from 0 to 100 rather than slide it in from right to left.



## HOUR 20

# Working with Sound and Plug-Ins

---

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

- ▶ How browser plug-ins work
- ▶ How JavaScript works with plug-ins
- ▶ Scripting objects in plug-ins
- ▶ Integrating JavaScript and Flash
- ▶ Testing JavaScript's sound support
- ▶ Creating an application using sounds

Browser plug-ins enable the browser to work with sounds, printer-ready documents, and other formats instead of being limited to HTML. JavaScript can connect with some plug-ins to add interactive features. In this hour, you'll explore JavaScript's plug-in support and look specifically at playing sounds.

## **Introducing Plug-Ins**

Plug-ins were introduced by Netscape in Navigator 3.0. Rather than adding support directly to the browser for media types such as formatted text, video, and audio, Netscape created a modular architecture that allows programmers to write their own browser add-ons for these features.

There are now hundreds of plug-ins available for Netscape, Firefox, and Internet Explorer. Here are a few of the most popular:

- ▶ Macromedia's Shockwave and Flash plug-ins support animation and video.
- ▶ Adobe's Acrobat plug-in supports precisely formatted, cross-platform text.

- ▶ Apple's QuickTime plug-in supports many audio and video formats.
- ▶ RealPlayer supports streaming audio and video.

Firefox and Internet Explorer use different plug-in formats and usually require different versions of a plug-in. Additionally, some plug-ins are available only for one platform, such as Windows or Macintosh.

## The <embed> and <object> Tags

Browsers support two tags for plug-ins, <embed> and <object>. The following is an example of the <embed> tag that embeds a sound in a page:

```
<embed src="sound.wav" autostart="false" loop="false">
```

This example uses the sound.wav file. It sets two parameters: autostart controls whether the sound automatically plays when the page loads, and loop controls whether the sound repeats after it plays the first time. The parameters supported depend on the plug-in being used.

A more standard tag, <object>, is part of the HTML 4.0 specification. Here's the same sound file using <object>:

```
<object type="audio/x-wav" data="sound.wav" width="100" height="50">
 <param name="src" value="sound.wav">
 <param name="autostart" value="false">
</object>
```

### ***Did you Know?***

Although <object> is a more standard way of embedding a file, most current browsers still support <embed>, which works better in some cases. Always try your pages that use plug-ins in different browsers to make sure they work.

## Understanding MIME Types

Multipurpose Internet Mail Extensions types (MIME) is a standard for classifying different types of files and transmitting them over the Internet. The different types of files are known as *MIME types*.

You've already worked with a few MIME types: HTML (MIME type text/html), text (MIME type text/plain), and GIF images (MIME type image/gif). The <script> tag also uses a MIME type to indicate the language: text/javascript. Although web browsers don't normally support many more than these types, external applications and plug-ins can provide support for additional types.

When a web server sends a document to a browser, it includes that document's MIME type in the header. If the browser supports that MIME type, it displays the file. If not, you're asked what to do with the file (such as when you click on a .zip or .exe file to download it).

## How JavaScript Works with Plug-Ins

Some plug-ins, such as the sound plug-ins you'll use later this hour, support scripting with JavaScript. Scripting plug-ins works just like scripting the DOM: You assign an `id` attribute to the `<embed>` or `<object>` tag, and then use `document.getElementById()` to find the object corresponding to the embedded item.

After you've found the object, what you can do with it depends on the file type and the plug-in. For example, most sound plug-ins support a `Play()` method. Here's an example that finds an embedded sound with the `id` attribute `sound1` and plays the sound:

```
obj = document.getElementById("sound1");
obj.Play();
```

Because plug-in methods are not part of the standard DOM, you'll need to consult the plug-in's documentation to find out what methods are supported and what your script can do with the embedded object.

## Plug-In Feature Sensing

Any time you work with plug-ins, it's important to remember that not all browsers will have the needed plug-in installed. Although both Firefox and Internet Explorer will attempt to notify users and let them know where to install the plug-in, expecting users to install software just to view your site is a bit optimistic.

Instead, you should use feature sensing to use the plug-in only when it is supported. For example, you could check for the `Play()` method like this:

```
if (obj.Play) {
 obj.Play();
} else alert("Can't Play.");
```

A more sophisticated method that handles errors as well as feature sensing is presented later in this hour.

Feature sensing is the same technique you've used to make sure browsers support the W3C DOM. See Hour 15, "Unobtrusive Scripting," for information on feature sensing.

## JavaScript and Flash

Adobe (formerly Macromedia) Flash is the Web's most popular format for movies and interactive content that require a bit more graphical splendor than HTML and JavaScript can provide. Flash's programming language is similar to JavaScript, and JavaScript can work with Flash.

### ActionScript

If you program scripts for a Flash movie, you use a language called ActionScript. You may find that ActionScript has a strong similarity to JavaScript, and for good reason—the version of ActionScript used in Flash 5.0 and later is based on the same ECMAScript standard that specifies the syntax for JavaScript.

Although the language is the same, Flash programming is quite different from writing JavaScript for the Web—you are scripting Flash objects rather than working with the DOM. However, you'll find that the basic syntax of the language is the same, which makes it easy for a JavaScript programmer to work with Flash when its capabilities are needed.

### JavaScript and Flash Communication

JavaScript and Flash can communicate and work together. Adobe's Flash/JavaScript Integration Kit, available as a free download, enables JavaScript to call ActionScript functions within Flash objects, and also enables Flash scripts to call JavaScript functions within the page that contains them.

The Flash/JavaScript Integration Kit works best with Flash Player 6.0 or later, although it also includes basic support for earlier versions of Flash. If you are developing a Flash application and need it to communicate with JavaScript, you can download the kit from <http://weblogs.macromedia.com/flashjavascript/>.

If you're using an existing Flash object, the author might have already set it up to work with JavaScript, in which case it will have a list of methods available like other plug-in objects.

### Embedding Flash with JavaScript

One other common use of JavaScript with Flash is to use JavaScript to generate the `<object>` or `<embed>` tag to embed a Flash object. Although you could use HTML directly, using JavaScript enables you to sidestep Internet Explorer's warning dialog that pops up whenever an embedded object is in use. JavaScript can also pass parameters, such as the user's screen size, to Flash by writing them into the `<embed>` or `<object>` tag.

Microsoft added the warning dialog for embedded objects in response to a patent dispute. See the Try It Yourself section later this hour for an example that uses JavaScript to embed objects in a page and avoid this warning.

***Did you  
Know?***

## Playing Sounds with JavaScript

Although the W3C DOM has made advanced effects and applications possible in JavaScript in a painless, cross-browser fashion, no standard has emerged to do the same for JavaScript's sound support. There are a few ways of making JavaScript play sounds, and none of them work consistently in all browsers all of the time. Nonetheless, with a bit of effort, you can play sounds in most browsers.

Because sound support in browsers is inconsistent, there's no guarantee your sounds will work for everyone. Be sure any sound you use in JavaScript applications is optional and that the script still works even on browsers that won't play the sounds.

***By the  
Way***

## Sound Formats

There are a wide variety of sound formats, usually identified by their file extensions. The following are some of the most common sound formats on the Web:

- ▶ **.au (Audio Unit)**—The earliest sound format supported by browsers, and still the most widely supported. Some browsers have built-in support for this format. In Firefox, the QuickTime plug-in supports .au files.
- ▶ **.wav**—The standard Windows sound format (usually played by Media Player on Windows machines).
- ▶ **.mp3**—A compressed format for larger files, such as music. MP3 plug-ins are not included with most browsers, but are often installed by users.
- ▶ **.mid (MIDI)**—Rather than audio, MIDI files store note information to re-create a song using a standard set of instruments. Most computers support MIDI music, although a browser plug-in might be required.

Any of these formats can be supported by most browsers, but unfortunately there is no format that is universally supported. If you're hoping as many visitors as possible will be able to hear your sounds, the best choice is .au if you're using standard audio plug-ins, or .mp3 if you're using Flash.

## Sound-Playing Plug-Ins

Browsers almost always require a plug-in to play sounds. Fortunately, sound plug-ins are widely used and many of your site's visitors already have one or more of them installed. Here are the most common sound-playing plug-ins:

- ▶ **QuickTime**—Apple's sound and video player, installed by default on Macintosh systems. QuickTime plug-ins are also available for Internet Explorer for Windows and for Firefox on Windows and Macintosh.
- ▶ **Windows Media Player**—Microsoft's sound and video player, installed by default on Windows systems.
- ▶ **RealPlayer**—A popular third-party plug-in for playing music and video, available from [http:// www.real.com](http://www.real.com).
- ▶ **Flash**—Although the Flash plug-in doesn't play standard embedded sounds, Flash animations and movies can play sounds, as you'll learn later in this section.

## Embedding Sounds

The following is a simple example of an `<embed>` tag to embed a sound in a page:

```
<embed id="note_c1" src="c1.au" width="0" height="0"
 autostart="false" enablejavascript="true"/>
```

This example works with the most common sound plug-ins. It specifies a source filename for the sound file (`c1.au`) and `autostart="false"` to prevent the sound from playing when the page loads. The `enablejavascript` parameter is required by some plug-ins to allow scripting.

The width and height parameters set the size of the embedded player. If they are not zero, the player will be visible with Play, Pause, and Stop buttons. Setting them to zero hides the player, useful when you intend to control it strictly with JavaScript. (A hidden parameter is supposed to hide the player, but this causes sounds not to play in some browsers.)

## Controlling Sounds with JavaScript

After you've embedded a sound—assuming a browser plug-in supports it—you can use the following methods of the sound object to control the sound:

- ▶ **Play() or DoPlay()**—Starts playing the sound, and stops when the sound is finished. `DoPlay()` is supported by RealPlayer, and `Play()` is supported by most other sound plug-ins.

- ▶ `Stop()`—Stops the currently playing sound.
- ▶ `Rewind()`—Restarts the current sound at the beginning.

Depending on the audio plug-in in use, the methods supported might be different. Always use try and catch when attempting to control sounds to avoid errors.

**Watch  
Out!**

## Detecting Sound Support

Because you can't count on sounds being supported by all browsers, it's a good practice to use try and catch to test the statements and display a message (or take another appropriate action) if sounds are not supported:

```
try {
 sound.DoPlay();
} catch (e) {
 try {
 sound.Play();
 } catch (e) {
 alert("No sound support.");
 }
}
```

This code first tries `RealPlayer's DoPlay()` method. If that doesn't work, it tries the `Play()` method. If neither approach works, it displays an error message.

The try and catch keywords are used to test a risky statement, find out whether it works, and suppress the browser's usual error messages. See Hour 16, "Debugging JavaScript Applications" for more information.

**By the  
Way**

## Using Flash

If you are relying on sounds for an application, you might want to consider using Flash. You can create a simple Flash object that loads sound files and allows JavaScript to play them. This gives you scriptable sounds using one consistent plug-in that works on most platforms.

Scott Schiller's SoundManager provides an easy way to use Flash sounds from JavaScript. SoundManager uses a Flash object to play MP3-formatted sounds you specify in an XML file. After you've created the XML file and included SoundManager using a `<script>` tag, you can use its methods to control the sounds. More information and the download for SoundManager are available at <http://www.schillmania.com/projects/soundmanager/>.

## Testing Sounds in JavaScript

You can now create a simple example that uses JavaScript to play a sound. Listing 20.1 shows an HTML document with an embedded script to play a sound when you click a button.

---

**LISTING 20.1** A Simple Example of Playing Sounds Using JavaScript

---

```
<html>
<head>
<title>Sound Test</title>
<script language="JavaScript" type="text/javascript">
function PlaySound() {
 var sound = document.getElementById("note_c1");
 try {
 // RealPlayer
 sound.DoPlay();
 } catch (e) {
 try {
 // Windows Media / Quicktime
 sound.Play();
 } catch (e) {
 alert("No sound support.");
 }
 }
}
</script>
</head>
<body>
<h1>Sound Test</h1>
<embed id="note_c1" src="c1.au" width="0" height="0"
 autostart="false" enablejavascript="true"/>
<input type="button" value="Play the Sound"
 onClick="PlaySound()">
</body>
</html>
```

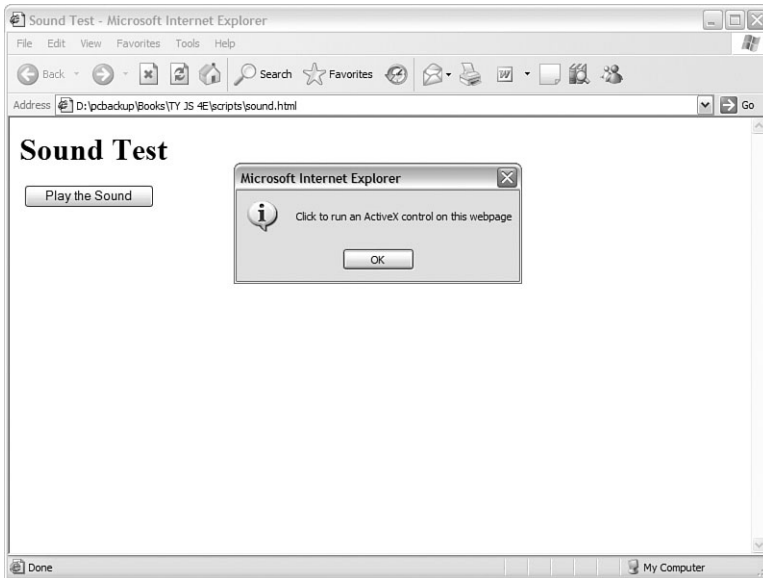
---

To try the example, you'll need a sound file: the `c1.au` file is available at this book's website, or you can substitute the `.au` format sound of your choice. Load the document into a browser and click the button to play the sound.

If you don't hear a sound, or if the "No sound support" message is displayed, try looking at the JavaScript Console in Firefox or clicking the error icon in the lower-left corner of the window in Internet Explorer. You might need to install a plug-in to get it to work.

Internet Explorer might display an alert message when you load the page, as shown in Figure 20.1. Due to a patent dispute, Microsoft made their browser require you to click on something in order for embedded objects to work. Although this is only a minor annoyance in this example, it's possible to eliminate it by using JavaScript to write the `<embed>` tag. The Try It Yourself section of this hour includes an example of this technique.





**FIGURE 20.1**  
Internet Explorer warns you before enabling an embedded object.

## Try It Yourself

### Playing Music with the Mouse

As an example of scripting multiple embedded objects, you can create a simple demonstration that displays a piano keyboard and plays piano notes when you click on the keys. This example requires an .au sound file for each key, which you can download from this book's website.

## The HTML Document

The HTML file for this document includes a series of <div> tags that will act as the black and white piano keys. A <link> tag is used to include a CSS file to style the keys, and a <script> tag includes a script you'll create later in this section. The complete HTML document is shown in Listing 20.2.

### LISTING 20.2 The HTML Document for the Piano Example

```
<html>
<head>
<title>JavaScript Piano</title>
<link rel="stylesheet" type="text/css" href="piano.css">
</head>
<body>
<h1>JavaScript Piano</h1>
<div class="white" id="c1"> </div>
<div class="black" id="cs1"> </div>
<div class="white" id="d1"> </div>
```

**LISTING 20.2 Continued**

---

```
<div class="black" id="ds1"> </div>
<div class="white" id="e1"> </div>
<div class="white" id="f1"> </div>
<div class="black" id="fs1"> </div>
<div class="white" id="g1"> </div>
<div class="black" id="gs1"> </div>
<div class="white" id="a1"> </div>
<div class="black" id="as1"> </div>
<div class="white" id="b1"> </div>
<div class="white" id="c2"> </div>
<p style="clear:left">
Click the piano keys above to play sounds.
</p>
<script language="javascript" type="text/javascript"
 src="piano.js"> </script>
</body>
</html>
```

---

Type this document or download it from this book's website and store it in the same folder as the sound files. You'll also need the CSS and JavaScript files described in the next sections.

## The CSS Style Sheet

Using CSS, you can make the browser display the series of `<div>` tags in the HTML document as something resembling piano keys. Listing 20.3 shows the CSS file for this example.

**LISTING 20.3 The CSS File for the Piano Example**

---

```
.white {
 float: left;
 background-color: white;
 height: 300px;
 width: 30px;
 border: 2px solid black;
}
.black {
 float: left;
 background-color: black;
 height: 225px;
 width: 25px;
}
```

---

This file defines two styles for the two classes used in the HTML document, `white` and `black`. The `float` attribute makes the keys appear as a horizontal set of boxes. The size of the keys is set using `width` and `height` attributes, and `background-color` sets the colors to differentiate the keys.

## Playing the Sounds

The `PlaySound()` function will be called when a key is clicked to play a sound. The first lines of this function detect which key was clicked and use the `id` attribute of the key `<div>` element to construct the `id` attribute of the corresponding sound:

```
function PlaySound(e) {
 if (!e) var e = window.event;
 // which key was clicked?
 thiskey = (e.target) ? e.target: e.srcElement;
 var sound = document.getElementById("note_" + thiskey.id);
```

The remainder of `PlaySound()` will attempt to play the piano note using the `try` and `catch` routine described earlier in this hour.

## Embedding the Sounds

This example will use JavaScript `document.write()` statements to write out an `<embed>` tag for each note. Although this is a roundabout way of doing things, it conveniently avoids Internet Explorer's warning dialog about embedded objects, which would otherwise pop up 13 times—once for each embedded sound. Here are the lines that write an `<embed>` tag:

```
document.write('<embed id="' + 'note_' + divs[i].id + '"');
document.write(' src="' + divs[i].id + '.au" width="0" height="0"');
document.write(' autostart="false" enablejavascript="true">');
```

The `src` attribute of the `<embed>` tag is set using the `id` attribute of each `<div>` element to embed the corresponding sound file for each key.

## Putting It All Together

To get the piano working, you can combine the techniques discussed previously with a bit more JavaScript. Listing 20.4 shows the JavaScript file for this example.

### LISTING 20.4 The JavaScript File for the Piano Example

---

```
function Setup() {
 if (!document.getElementById) return;
 // Set up event handlers and embed the sounds
 divs = document.getElementsByTagName("div");
 for (i=0; i<divs.length; i++) {
 // embed the appropriate sound using document.write
 document.write('<embed id="' + 'note_' + divs[i].id + '"');
 document.write(' src="' + divs[i].id + '.au" width="0" height="0"');
 document.write(' autostart="false" enablejavascript="true">');
 // set up the event handler
 divs[i].onclick = PlaySound;
 }
}
```

**LISTING 20.4 Continued**


---

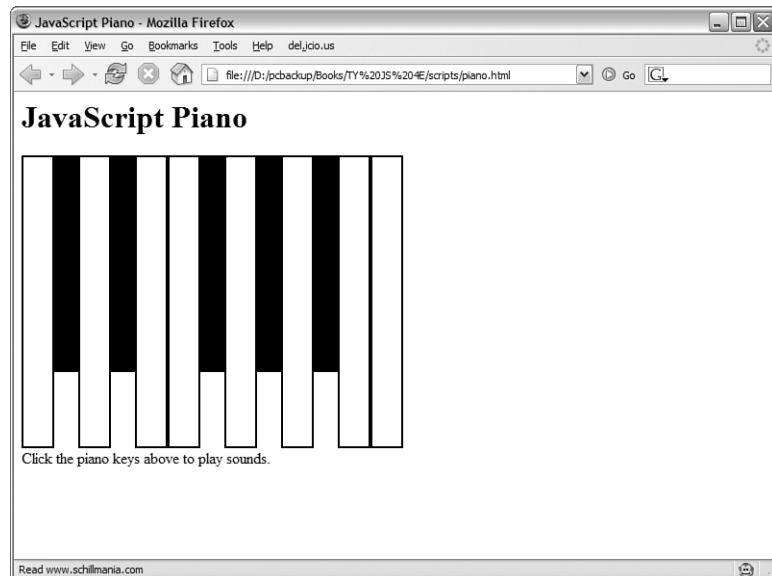
```
function PlaySound(e) {
 if (!e) var e = window.event;
 // which key was clicked?
 thiskey = (e.target) ? e.target : e.srcElement;
 var sound = document.getElementById("note_" + thiskey.id);
 try {
 // RealPlayer
 sound.DoPlay();
 } catch (e) {
 try {
 // Windows Media / Quicktime
 sound.Play();
 } catch (e) {
 alert("No sound support.");
 }
 }
}
// Run the setup routine when this script executes
Setup();
```

---

The `Setup()` function executes when the script loads. Because the `<script>` tag appears after the `<div>` elements in the HTML file, it can set event handlers for each `<div>` and write out the `<embed>` tags. `Setup()` uses `document.getElementsByTagName` and a `for` loop to do this for each of the keys.

To test the piano, make sure you have everything in one folder: The HTML document, the CSS file (`piano.css`), the JavaScript file (`piano.js`), and all 13 sound files. The complete example is shown in Figure 20.2.

**FIGURE 20.2**  
The JavaScript piano example in action.



## Summary

In this hour, you learned about browser plug-ins and how they work with JavaScript. You also learned about JavaScript's support for sound (or the lack thereof) and how you can use JavaScript to detect and work with common sound-playing plug-ins. Finally, you created a piano keyboard with audio using JavaScript.

Congratulations—you've reached the end of Part V of this book. In part VI, you'll apply your JavaScript knowledge to create some complex applications. This begins in Hour 21, "Building JavaScript Drop-Down Menus."

## Q&A

**Q.** *Is there a way to list all of the plug-ins installed in the browser?*

**A.** Yes. Type `about:plugins` to display a list of plug-ins installed in Netscape or Firefox. These browsers also support a proprietary `navigator.plugins` object, an array that contains information about each installed plug-in, which you can access with JavaScript. Unfortunately, this is not a standard part of the DOM and is not supported by other browsers.

**Q.** *Can I add sounds to a site's navigation bar or user interface?*

**A.** Yes, this can be done using the techniques in this hour and `onMouseOver` or `onClick` event handlers. However, given the inconsistency of sound support in browsers, this is a lot of trouble for a feature that will probably annoy your visitors anyway.

**Q.** *Can the browser play more than one sound at the same time?*

**A.** This ultimately depends on the audio plug-in, but none of the current ones support playing more than one sound at a time.

## Quiz Questions

Test your knowledge of JavaScript's sound and plug-in features by answering the following questions.

1. Which HTML tag is often used to include a plug-in object within a web page?
  - a. `<sound>`
  - b. `<embed>`
  - c. `<plugin>`

2. Which of the following is not a sound-playing plug-in?
  - a. RealPlayer
  - b. QuickTime
  - c. Acrobat
3. Which is the correct statement to play a sound?
  - a. `sound.Go();`
  - b. `sound.Play();`
  - c. `sound.Submit();`

## Quiz Answers

1. b. The `<embed>` tag embeds a plug-in object in a page.
2. c. The Acrobat plug-in displays PDF files.
3. b. The `Play()` method plays a sound.

## Exercises

If you want to gain more experience working with sounds in JavaScript, try the following exercises:

- ▶ Expand the piano keyboard in Listing 20.2 to include more notes. (Additional sound files are available from this book's website.) You should only need to change the HTML file.
- ▶ Try adding one or more sounds to the animated slideshow in the previous hour (refer to Listing 19.8). You can adapt the `PlaySound()` function from this hour to play a specific sound as the slideshow advances.

## **PART VI:**

# **Creating Complex Scripts**

<b>HOURL 21</b>	<b>Building JavaScript Drop-down Menus</b>	<b>345</b>
<b>HOURL 22</b>	<b>Creating a JavaScript Game</b>	<b>359</b>
<b>HOURL 23</b>	<b>Creating JavaScript Applications</b>	<b>377</b>
<b>HOURL 24</b>	<b>Your Future with JavaScript</b>	<b>393</b>

*This page intentionally left blank*



## HOUR 21

# Building JavaScript Drop-Down Menus

---

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

- ▶ How to create drop-down menus using JavaScript
- ▶ Defining menus using bullet lists
- ▶ Using CSS to lay out menus
- ▶ Using JavaScript to display and hide submenus
- ▶ Using CSS to improve the menu's appearance

Welcome to Part VI! Now that you've spent some time learning both beginning and advanced JavaScript techniques, it's time to put them into action with some more complicated examples. In this hour, you'll use HTML, CSS, and JavaScript to create a drop-down menu navigation system.

## **Designing Drop-Down Menus**

One of the most common uses for JavaScript and the DOM is to create drop-down menus, similar to those used in Windows and MacOS, as a navigation system for a page. Figure 21.1 shows a drop-down menu in action.

Why use drop-down menus? Ideally, you should use them when a website or web application has too many options to conveniently fit on the page. Adding a drop-down menu to a site with only a few pages will just make it more confusing to visitors.

Another potential problem with drop-down menus is that they traditionally require some messy browser-specific code and some awkward HTML markup. Thanks to the now standard W3C DOM, you can create menus using simple markup and a script that works in all modern browsers.

**FIGURE 21.1**

A drop-down menu.



## Creating the HTML Markup

There will always be browsers that don't support drop-down menus correctly—in particular, mobile phone browsers are unlikely to work with this script. You can avoid problems with compatibility by making an unobtrusive script using standard markup. The HTML document for this example, shown in Listing 21.1, uses bullet lists (`<ul>` and `<li>` tags) to organize the links into menus.

### LISTING 21.1 The HTML for the Drop-Down Menu

```
<html>
<head>
<title>A DOM drop-down menu</title>
<link rel="stylesheet" type="text/css" href="dropdown.css">
<script language="javascript" type="text/javascript"
 src="dropdown.js">
</script>
</head>
<body>
<h1>Menu Test</h1>
<ul id="menu">
<li class="menu">Home
<li class="menu">Products

 Sub-item 1
 Sub-item 2
 Item 3

<li class="menu">Support

 Sub-item 1
 Sub-item 2

<li class="menu">Employment

 Sub-item 1
 Sub-item 2

<li class="menu">Contact Us

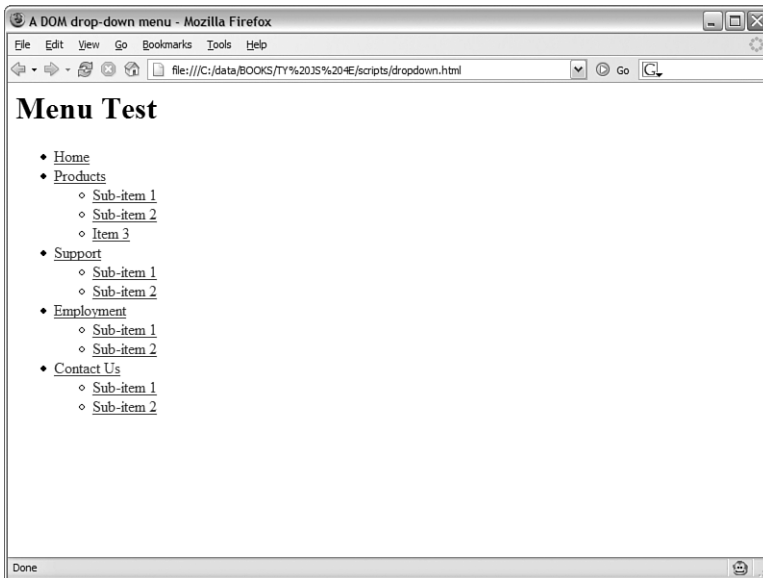
 Sub-item 1
 Sub-item 2

</body>
</html>
```

The top-level links (Home, Products, Support, Employment, and Contact Us) are formatted as a bullet list. Most of the links have subitems, listed in a nested bullet list. These subitems will be displayed as drop-down menus using CSS formatting and JavaScript.

Although you have not yet created the CSS or JavaScript for this example, you can try the HTML document in a browser—it will be displayed as a simple bullet list, as shown in Figure 21.2.

Notice the class and id attributes in the HTML—these will be used by the CSS and JavaScript code to format the menu and add behavior. The main `<ul>` tag that encloses the top-level items has an id attribute of `menu`, and each top-level item's `<li>` tag has the class attribute `menu`.



**FIGURE 21.2**  
Without formatting, the links display as bullet lists.

The links in this example all link to a nonexistent URL, #. To use the menu on your site, you'll need to replace them with actual links.

**Watch  
Out!**

## Laying Out the Menu with CSS

As you can see in Figure 21.2, the list of links doesn't look much like a drop-down menu yet. You can now use CSS to format the links to appear in the right format.

The first step is to make the main list display in a horizontal format. This can be done with two CSS rules:

```
#menu li {
 float: left;
 list-style-type: none;
}
```

The selector, `#menu li`, looks for any list item directly under the `#menu` list. The `float: left` rule causes the items to display left to right instead of vertically, and `list-style-type: none` prevents a bullet from being displayed. Next, a couple of rules for the subitems:

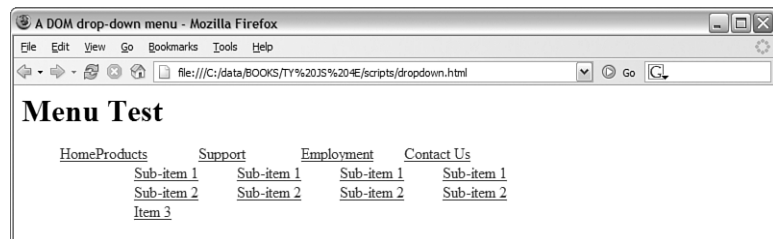
```
#menu li ul li {
 float: none;
 list-style-type: none;
}
```

The selector here, `#menu li ul li`, looks for `<li>` items nested under the main `<li>` items. Once again, `list-style-type: none` is used to eliminate bullets. The `float: none` rule is necessary because we want the subitems to be listed vertically rather than inheriting the floating behavior of the main list.

Figure 21.3 shows what the list looks like with the styles so far. As you can see, the menu is beginning to take shape: The main links are displayed in a horizontal row, and each subitem list appears vertically underneath its corresponding item. The spacing and alignment needs work, but it's a start.

**FIGURE 21.3**

The list of links with some basic styles.



### **By the Way**

As you develop a complex layout using CSS, be sure to test in multiple browsers. Floats are one area where Internet Explorer shows its quirks, and you may need to adjust a few rules to make it work cross-browser.

To make the menu look more like a menu, you just need some padding, width, and other settings. Listing 21.2 shows the complete style sheet for the drop-down menu.

**LISTING 21.2** The CSS File for the Drop-Down Menu

---

```
/* The whole menu */
#menu {
 position: absolute;
}
/* Each menu name */
#menu li {

 float: left;
 list-style-type: none;
 padding-right: 20px;
 width: 100px;
 background-color: silver;
}
/* The entire submenu */
#menu li ul {
 background-color: silver;
 margin: 0px;
 padding: 0px;
}
/* Each submenu item */
#menu li ul li {
 padding: 0px;
 margin: 0px;
 float: none;
 list-style-type: none;
 width: 80px;
}
}
```

---

This style sheet uses padding and width values to make sure the submenus line up with their headings. Some background-color attributes are applied to make the menu appear more solid.

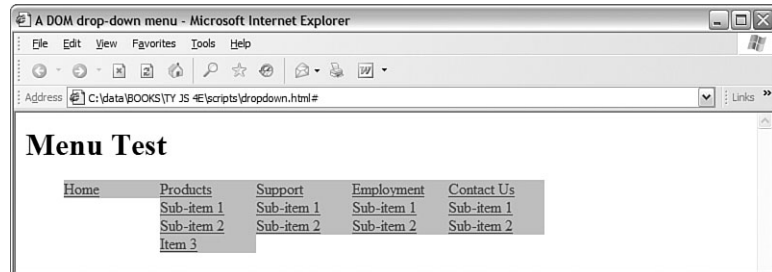
The position: absolute rule is used so the menus can overlap the content of the page when they drop down. There's no content in the example, but on a real site you don't want to leave room for the menus—if you have that kind of room, you might as well just display the links all of the time.

Using position: absolute has a downside—because the menu isn't positioned in the normal flow of the page, the main menu can overlap part of your page unless you avoid it by positioning the other content around it. The ideal situation would be for the main menu to use relative positioning while the submenus use absolute positioning—unfortunately, this does not work consistently in Internet Explorer.

The styled menu is shown in Figure 21.4. As you can see, the entire menus are shown at this time—the submenus will be hidden by the script. This ensures that the menu will still be accessible to browsers that support CSS but not JavaScript.

**FIGURE 21.4**

The menu with full CSS styling.



### By the Way

When the script is added, the full menus will display for an instant before the script hides them. If you find this annoying, you can add a `display:none` rule to the CSS for the submenu `<ul>`. This eliminates the flicker, but makes the menu less useful to browsers without JavaScript support.

## Scripting Drop-Down Menu Behavior

You now have a list of links that looks like a drop-down menu. All you need now is a script to make it act like one. Your script will set up the menu when the page loads, and respond to event handlers to show and hide the submenus.

### Setting Up the Menu

The `SetupMenu()` function will run when the page loads, and then configure the drop-down menu. This mainly consists of hiding the submenus and configuring some event handlers. The function will use a loop to look at all of the `<li>` elements in the page, and if they have a `class` attribute of `menu`, they're considered part of the menu. The following lines set up the event handlers for the link and hide the submenu:

```
thelink=findChild(items[i],"A");
thelink.onmouseover=ShowMenu;
thelink.onmouseout=StartTimer;
//is there a submenu?
if (ul=findChild(items[i],"UL")) {
 ul.style.display="none";
}
```

The `findChild()` function is used twice here. This function will also be defined in your script, and will return the first child item of a particular type it finds for an object. In the preceding lines, it is used to find the link (`<a>` tag) under the list item, and to find the nested list of subitems (`<ul>` tag). The `style.display` property is used to hide each submenu.

## Showing a Submenu

The `ShowMenu()` function will be called by the `onmouseover` event handler when you move over a link. Here's an excerpt from this function that handles showing the submenu:

```
// find the submenu, if any
ul = findChild(thislink,"UL");
if (!ul) return;
ul.style.display="block";
```

Once again, `findChild()` is used to find the `<ul>` element under the current item, and the `display` property is set to `block` to display the menu.

## Hiding Submenus

The logic for showing the submenus is simple—whenever the mouse pointer is over a menu heading, the corresponding submenu is displayed. Hiding a submenu is a bit more complicated—the menu needs to stay open while you select an item, but get out of the way quickly when you're not using it. The `HideMenu()` function will accomplish this:

```
function HideMenu(thelink) {
 // find the submenu, if any
 ul = findChild(thelink,"UL");
 if (!ul) return;
 ul.style.display="none";
}
```

One time you definitely want a menu to be hidden is when the user opens another menu, so the `ShowMenu()` function will call `HideMenu()` to hide the previous menu. You also want the menu to disappear if you move out of it, but a simple `onmouseout` event handler won't work because the user could have moved off the menu heading and into the submenu. Instead, the `onmouseout` event calls the `StartTimer()` function:

```
function StartTimer() {
 t = window.setTimeout("HideMenu(current)",200);
}
```

This function sets a timeout to hide the menu in 200 milliseconds. If the user moves over any of the submenu items during the delay, the timer is reset with the `ResetTimer()` function:

```
function ResetTimer() {
 if (t) window.clearTimeout(t);
}
```

This function cancels the timeout using the `clearTimeout()` method, keeping the menu on the screen until the `onmouseout` event starts the timer again. Finally, some additional lines in the `SetupMenu()` function will set up event handlers to call `StartTimer()` and `ResetTimer()` for each subitem:

```
for (j=0; j<ul.childNodes.length; j++) {
 ul.childNodes[j].onmouseover=ResetTimer;
 ul.childNodes[j].onmouseout=StartTimer;
}
```

## Completing the Script

You can now combine all of the functions discussed above to create working drop-down menus. The complete drop-down menu script is shown in Listing 21.3.

### LISTING 21.3 The Complete JavaScript File for the Drop-Down Menus

---

```
// global variables for timeout and for current menu
var t=false,current;
function SetupMenu() {
 if (!document.getElementsByTagName) return;
 items=document.getElementsByTagName("li");
 for (i=0; i<items.length; i++) {
 if (items[i].className != "menu") continue;
 //set up event handlers
 thelink=findChild(items[i],"A");
 thelink.onmouseover=ShowMenu;
 thelink.onmouseout=StartTimer;
 //is there a submenu?
 if (ul=findChild(items[i],"UL")) {
 ul.style.display="none";
 for (j=0; j<ul.childNodes.length; j++) {
 ul.childNodes[j].onmouseover=ResetTimer;
 ul.childNodes[j].onmouseout=StartTimer;
 }
 }
 }
}
// find the first child object of a particular type
function findChild(obj,tag) {
 cn = obj.childNodes;
 for (k=0; k<cn.length; k++) {
 if (cn[k].nodeName==tag) return cn[k];
 }
 return false;
}
function ShowMenu(e) {
 if (!e) var e = window.event;
 // which link was the mouse over?
 thislink = (e.target) ? e.target : e.srcElement;
 ResetTimer();
 // hide the previous menu, if any
 if (current) HideMenu(current);
 // we want the LI, not the link
```



**LISTING 21.3 The Complete JavaScript File for the Drop-Down Menus**

```
/ thislink = thislink.parentNode;
current=thislink;
// find the submenu, if any
ul = findChild(thislink,"UL");
if (!ul) return;
ul.style.display="block";
}
function HideMenu(thelink) {
 // find the submenu, if any
 ul = findChild(thelink,"UL");
 if (!ul) return;
 ul.style.display="none";
}
function ResetTimer() {
 if (t) window.clearTimeout(t);
}
function StartTimer() {
 t = window.setTimeout("HideMenu(current)",200);
}
// Set up the menu when the page loads
window.onload=SetupMenu;
```

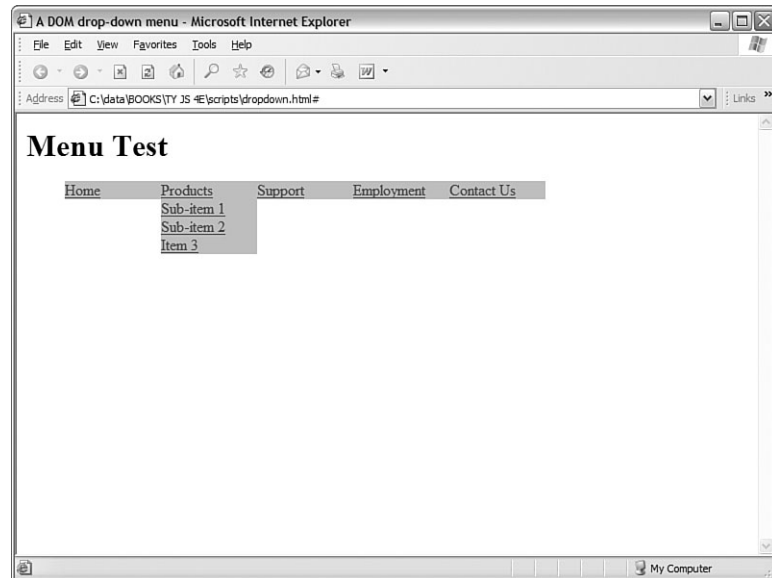
Here's a summary of how the script works from top to bottom:

- ▶ The first line defines two global variables: `t` stores a reference to the timeout so that it can be canceled, and `current` is the object for the currently open menu.
- ▶ The `SetupMenu()` function sets up event handlers to call `ShowMenu()`, `StartTimer()`, and `ResetTimer()`, and hides the submenus.
- ▶ The `findChild()` function is used by several of the other functions to find a child object.
- ▶ The `ShowMenu()` function shows a menu.
- ▶ The `HideMenu()` function hides a menu when the timeout expires.
- ▶ The `StartTimer()` and `ResetTimer()` functions manage the timeout discussed earlier.
- ▶ The final line of the script sets the window's `onload` event handler to the `SetupMenu()` function to set up the menu when the page loads.

To try the menu, first be sure you have all three files in the same folder: the HTML document, the CSS file (`dropdown.css`), and the JavaScript file (`dropdown.js`). You can then load the HTML document into a browser. Figure 21.5 shows the drop-down menu in action.

**FIGURE 21.5**

The drop-down menu in action.



## Try It Yourself

### Enhancing the Menu with CSS

Although the menu works as it is, the CSS could use some improvement. The menus are not well delineated, and there are no rollover effects to let you know you're moving over menu items. Also, to make a menu appear, you have to move the mouse over the *text* of the menu name—for this menu to work like users expect, the entire block that contains the menu name should be active.

An improved CSS style sheet can solve these problems. You might also want to add more CSS rules to fine-tune its formatting. Here are some suggestions:

- ▶ Change the fonts and colors to match your site.
- ▶ Add an `a:hover` selector to make the subitems change color as you move over them.
- ▶ Use border attributes to add borders around menus or subitems.
- ▶ Use margin attributes to add space between menu items.

Listing 21.4 shows a modified style sheet that makes the menu work as it should, and implements several of these ideas to create a menu with a different style.

**LISTING 21.4** A Style Sheet for a Different Style of Menu

---

```
/* The whole menu */
#menu {
 position: absolute;
 font-family: sans-serif;
 font-size: 100%;
}
/* Each menu name */
#menu li {
 float: left;
 list-style-type: none;
 width: 102px;
 background-color: silver;
 border: 1px solid black;
 text-indent: 0px;
 margin-left: 3px;
}
/* each main menu link */
#menu li a {
 color: black;
 text-decoration: none;
 width: 100%;
 display: block;
}
#menu li a:hover {
 color: white;
}
/* The entire submenu */
#menu li ul {
 background-color: silver;
 margin: 0px;
 padding: 0px;
}
/* Each submenu item */
#menu li ul li {
 padding: 0px;
 margin: 0px;
 float: none;
 list-style-type: none;
 width: 100px;
 text-indent: 0px;
 border: none;
}
#menu li ul li a {
 color: black;
 text-decoration: none;
}
#menu li ul li a:hover {
 color: black;
 background-color: aqua;
}
}
```

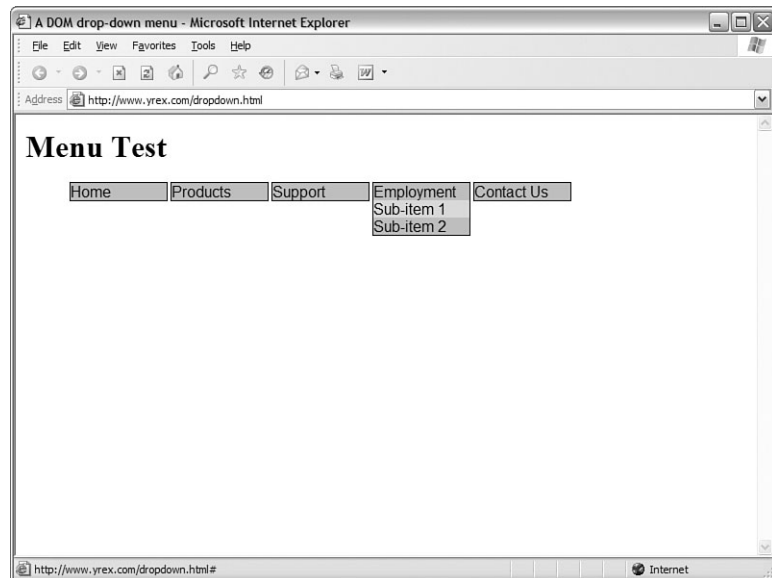
---

This style sheet has the following features:

- ▶ A sans-serif font is used for a more modern appearance.
- ▶ Borders and margins are used to make the menu names appear as separate boxes.
- ▶ An `a:hover` selector is used to make the menu names change color when the mouse is over them.
- ▶ The `width: 100%` and `display: block` rules for the menu names make the entire box active, not just the text.
- ▶ Another `a:hover` selector makes the submenu items change color when the mouse is over them.
- ▶ The `width: 100%` rule for submenu items makes the entire width of the submenu active, not just the text.

To use this style sheet, save it as `dropdown2.css` in the same folder as the HTML document, and change the `<link>` tag in the HTML document to refer to the new file. Figure 21.6 shows the drop-down menu with this style sheet.

**FIGURE 21.6**  
The drop-down menu with an alternative style sheet.



## Summary

In this hour, you've developed a complete application that uses HTML, CSS, and JavaScript to create drop-down menus for navigating a site. You learned how to create a simple HTML document using nested lists, and how to use CSS to format it as a horizontal menu with vertical drop-downs. You used JavaScript to make the drop-down menu work. Finally, you created an alternative style sheet to give the menu a different look.

In the next hour, you'll create another complex JavaScript application—a card game that uses JavaScript, images, and CSS to interact with the user.

## Q&A

**Q.** *Can I make a vertical pop-out menu using the same script in Listing 21.3?*

**A.** Yes. You'll need a different style sheet that doesn't use `float` for the menu headings, but uses `float:left` for the submenu. The same script and HTML document can be used with a vertical menu.

**Q.** *Can I add some space before each menu heading?*

**A.** Yes, but be aware that Internet Explorer has some bugs involving margins and padding when `float` is in use. Be sure to test in multiple browsers.

**Q.** *Which browsers support the drop-down menus?*

**A.** The drop-down menus you created in this hour should work in Internet Explorer 5.0 and later, Netscape 6.0 and later, and all versions of Firefox. Most important, because it uses the standard W3C DOM, it should work in all standards-compliant browsers—but watch out for formatting quirks in different browsers when you change the styles.

## Quiz Questions

Test your knowledge of the techniques used in this hour by answering the following questions.

1. Which of the following CSS rules makes the menu horizontal instead of vertical?
  - a. `float: left`
  - b. `position: absolute`
  - c. `orientation: horizontal`

2. Which of the following CSS selectors refers to an `<li>` element directly under the `<ul>` element with the id value `menu`?
  - a. `#menu ul li`
  - b. `#menu li`
  - c. `ul #menu li`
3. Which of the following is the correct command to cancel a timeout set with the command `t=window.setTimeout("HideMenu(current)",500);`?
  - a. `t = window.clearTimeout();`
  - b. `window.clearTimeout(t);`
  - c. `window.setTimeout("HideMenu(current)",0);`

## Quiz Answers

1. a. The drop-down menu uses `float:left` to make a horizontal menu.
2. b. The correct selector is `#menu li`.
3. b. The correct command is `window.clearTimeout(t)`.

## Exercises

If you want to gain more experience working with JavaScript drop-down menus, try the following exercises:

- Change the drop-down menu to contain the appropriate links for your site, or for an imaginary site. (You only need to change the HTML file, but each menu item needs the `class` value of `menu`.)
- Modify the CSS file for the drop-down menu to use colors, borders, or other attributes of your choice.

## Hour 22

# Creating a JavaScript Game

---

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

- ▶ How to design a JavaScript Game
- ▶ Creating game graphics
- ▶ Laying out the game board in HTML
- ▶ Using CSS to style the board
- ▶ Creating gameplay scripts
- ▶ Finalizing and testing the game

In this hour, you'll look at another complex application of JavaScript: a Poker Solitaire game that uses the W3C DOM, graphics, and some JavaScript logic to interact with the user quickly and responsively.

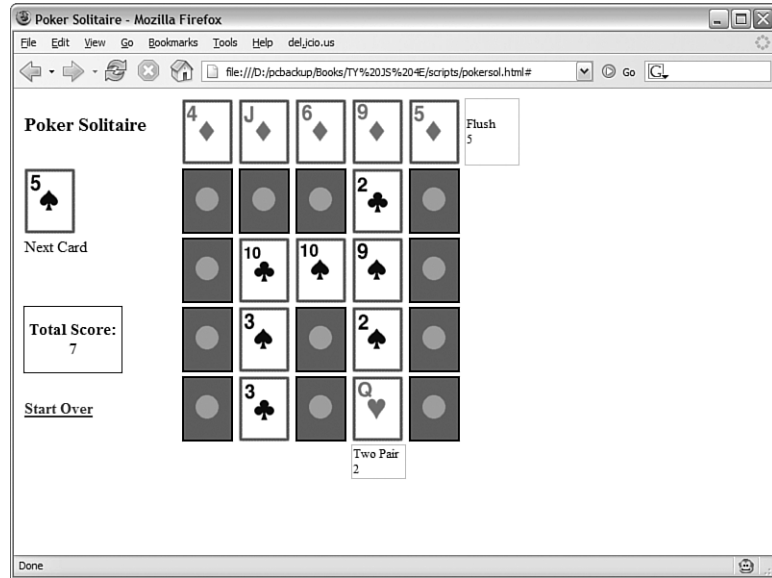
## **About the Game**

Although it's possible to create just about any game with JavaScript, a card game is a simple choice because the graphics are easy to create and the gameplay is relatively simple. In this hour, you'll create a Poker Solitaire game using HTML, JavaScript, and a bit of CSS.

## **How to Play**

Poker Solitaire is played on a five by five board. The deck of cards is shuffled, and you draw one card from the deck at a time and place it anywhere on the board. Your goal is to make each column, row, and diagonal row form the best possible poker hand. For example, in Figure 22.1, several cards have been placed on the board and the score for the completed column and row is shown.

**FIGURE 22.1**  
Playing Poker  
Solitaire.



## Scoring

Because there are no other players, the game will be scored. The script will calculate the score for each column, row, and diagonal on the board, and combine them for a total score. Points are awarded for poker hands, with more difficult (and less likely) combinations scoring higher:

- ▶ **Pair**—1 point (Two cards of the same number and different suits)
- ▶ **Two pair**—2 points (Two pairs)
- ▶ **Three of a kind**—3 points (Three cards of the same number)
- ▶ **Straight**—4 points (Five cards in numeric sequence)
- ▶ **Full house**—8 points (One pair plus three of a kind)
- ▶ **Four of a kind**—25 points (All four cards of the same number)
- ▶ **Flush**—5 points (Five cards of the same suit)
- ▶ **Straight flush**—50 points (Five cards of the same suit, in sequence)
- ▶ **Royal flush**—250 points (10, Jack, Queen, King, and Ace of the same suit)

In the JavaScript version of the game, the score for each row or column will be displayed as you complete it, and the total score will be updated in real time as you place each card on the board.



## Creating Graphics

This game will require some graphics—you'll need 52 images, one for each card in a standard deck. One more image, `blank.gif`, will be used to mark the spaces on the board that don't yet contain cards. You can download all of the graphics for the game from this book's website.

All of the graphics will be the same size, including the blank space image. The board will consist entirely of blanks at the start of a game, and images will be switched to the appropriate card when the user clicks to place a card. The images I used in the example are all  $53 \times 68$  pixels.

When you're working with a large number of graphics, filenames become important. It will make coding easier if you decide in advance on a naming scheme for the images. In this case, the filenames will include a number for the card's rank (1–13, with 1 representing Ace, and 11, 12, and 13 representing Jack, Queen, and King) and a letter for the suit. For example, the Seven of Hearts image would be `7h.gif`, and the Queen of Spades would be `12s.gif`.

## Creating the HTML Document

The HTML document for the game is straightforward. In keeping with the unobtrusive scripting strategies you've learned in previous hours, it contains no JavaScript—just a `<script>` tag that imports a script that will handle the game. Similarly, a separate CSS file will be used for styles. Listing 22.1 shows the HTML document.

### LISTING 22.1 The HTML Document for the Poker Solitaire Game

```
<html>
<head>
<title>Poker Solitaire</title>
<script language="JavaScript" type="text/javascript"
 src="pokersol.js">
</script>
<link rel="stylesheet" type="text/css" href="pokersol.css">
</head>
<body>
<table>
<tr>
 <td colspan="2"><h1>Poker Solitaire</h1></td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td class="score" id="row0"> </td>
</tr>
<tr>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td class="score" id="row1"> </td>
</tr>
<tr>
 <td valign="top" id="status"> Next Card</td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td class="score" id="row2"> </td>
</tr>
<tr>
 <td id="total"> Total Score:
 <div id="totalscore">0</div></td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td class="score" id="row3"> </td>
</tr>
<tr>
 <td> Start Over</td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td> </td>
 <td class="score" id="row4"> </td>
</tr>
<tr>
 <td> </td>
 <td class="score" id="diag1"> </td>
 <td class="score" id="col0"> </td>
 <td class="score" id="col1"> </td>
 <td class="score" id="col2"> </td>
 <td class="score" id="col3"> </td>
 <td class="score" id="col4"> </td>
 <td class="score" id="diag2"> </td>
</table>
</body>
</html>

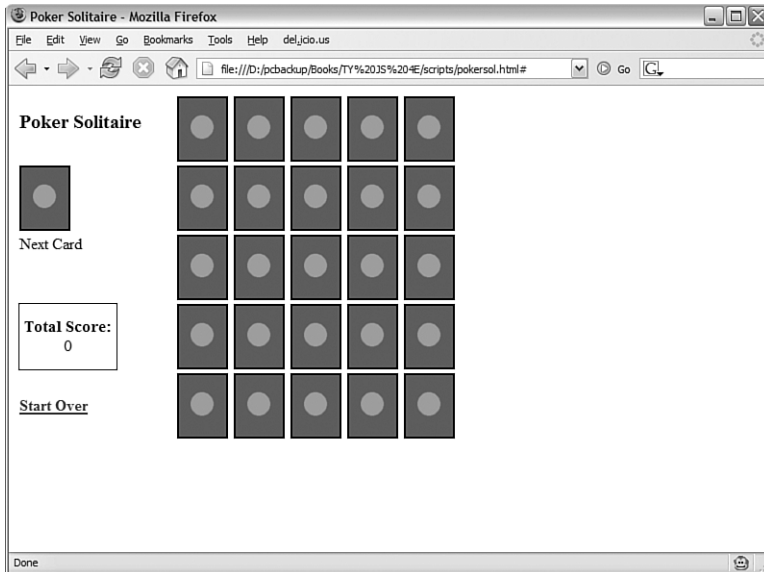
```

The game board is laid out using an HTML table. Although the game board is five by five squares, the table contains eight columns and six rows. The leftmost column

will be used for displaying the next card and the score, as well as a Start Over link. The remaining columns and rows will be used to display the score for each column and row as they are filled with cards.

The 25 spaces on the board are given unique id values, card1 through card25. These will be used by the script to determine which card the user clicks on, and also to replace the appropriate image when a card is placed. The table cells for displaying scores are given the id values row0-4, col0-4, diag1, and diag2.

Save the HTML document in a folder, or download it from this book's website. You can load it into a browser to see the game layout, as shown in Figure 22.2. The game won't be playable until you add the script you'll develop in the next section.



**FIGURE 22.2**  
The Poker  
Solitaire game  
layout.

## Creating the Script

Because this is the longest script in this book, it will be easier to understand if you look at some of its key functions before the entire script. The following sections discuss how the script will manage the game.

### Using Objects to Store Cards

Because JavaScript is designed to work with numbers but not specifically with playing cards, you can create a custom object to make it easier to manage the card game. The following code is the constructor for a Card object:

```
// make a filename for an image, given Card object
function fname() {
 return this.num + this.suit + ".gif";
}
// constructor for Card objects
function Card(num,suit) {
 this.num = num;
 this.suit = suit;
 this.fname = fname;
}
```

Each Card object will represent a space on the board. It has two properties, num and suit, and an fname() method that returns the filename for the graphic representing the card.

## Setting Up the Board

Along with the graphics on the screen, the board array will represent the game board by storing 25 Card objects, one for each space. Here's the code that will set up the board:

```
// array for board contents
board = new Array(26);
for (i=1; i<26; i++) {
 board[i] = new Card(0,"x");
 obj=document.getElementById("card"+i);
 obj.src = "blank.gif";
 obj.onclick = PlaceCard;
}
```

The first line creates the board array. The for loop then sets up each space on the board. First, it places a blank card in the array. Next, it finds the object for the corresponding space on the screen. It makes sure blank.gif is displayed in each space, and sets an event handler for onClick events to call the PlaceCard() function, which will handle the user's clicks on the board.

## Shuffling the Deck

The deck array will be used to simulate a deck of cards. The following code fills the array with Card objects:

```
deck = new Array(53);
for (i=1; i<14; i++) {
 deck[i] = new Card(i,"c");
 deck[i+13] = new Card(i,"h");
 deck[i+26] = new Card(i,"s");
 deck[i+39] = new Card(i,"d");
}
```

To save time, the statements in the for loop insert cards of each of the four suits into the deck. At this point, the cards are in order. The next step is to shuffle the deck:

```
n = Math.floor(100 * Math.random() + 200);
for (i=1; i<n; i++) {
 c1 = Math.floor(52*Math.random() + 1);
 c2 = Math.floor(52*Math.random() + 1);
 temp = deck[c2];
 deck[c2] = deck[c1];
 deck[c1] = temp;
}
```

This code starts by choosing a random number, *n*, ranging from 200 to 299. It then loops *n* times using a for loop. Each iteration of the loop chooses two random cards in the deck and swaps their positions. This ensures a reasonably random deck that still contains exactly 52 unique cards.

## Placing Cards on the Board

The `PlaceCard()` function will be called by an event handler when the user clicks on a space on the board. This function begins by determining which space was clicked:

```
function PlaceCard(e) {
 if (!e) var e = window.event;
 // which space on the board was clicked?
 thiscard = (e.target) ? e.target : e.srcElement;
 pos = thiscard.id.substring(4);
 if (board[pos].suit != "x") {
 return;
 }
}
```

These statements use the `target` or `srcElement` property to determine which space was clicked. The `pos` variable stores the numeric position on the board (1–25), calculated by removing “card” from the `id` value using the `substring()` method. The final if statement checks whether a card is already in place, and returns to prevent placing a card over an existing card.

The next section of `PlaceCard()` does the actual card placement:

```
drawcard=document.getElementById("dcard");
thiscard.src = deck[nextcard].fname();
drawcard.src = "blank.gif";
board[pos] = deck[nextcard];
nextcard++;
Score();
```

The `nextcard` variable keeps track of the next card in the deck, starting at one for the top card. This function uses `getElementById()` to find the object for the “next

card" display, and then uses the `fname()` method to assign the appropriate filename to the `src` property of the `image` object. The board array is updated with the new card, `nextcard` is incremented, and the `Score()` function is called to update the scores.

The next task for `PlaceCard()` is to check whether the game is over:

```
// Game over?
if (nextcard > 25) {
 EndGame();
}
else {
 drawcard.src = deck[nextcard].fname();
 // cache next image for draw pile
 nexti = new Image(53,68);
 nexti.src = deck[nextcard+1].fname();
}
}
```

If 25 cards have been placed, the `EndGame()` function is called to end the game. Otherwise, the next card is displayed in the display. The next card image (not yet displayed) is also preloaded so the game will respond quickly.

## Scoring Columns, Rows, and Diagonals

The `Score()` function will update the scores for each column, row, and diagonal each time a card is placed. Here is the code for the `Score()` function:

```
function Score() {
 score=document.getElementById("totalscore");
 totscore = 0;
 // rows
 for (x=0; x<5; x++) {
 r = x * 5 + 1;
 a =
 AddScore(board[r],board[r+1],board[r+2],board[r+3],board[r+4],"row"+x);
 totscore += a;
 }
 // columns
 for (x=0; x<5; x++) {
 r = x + 1;
 a =
 AddScore(board[r],board[r+5],board[r+10],board[r+15],board[r+20],"col"+x);
 totscore += a;
 }
 // diagonals
 a = AddScore(board[5],board[9],board[13],board[17],board[21],"diag1")
 totscore += a;
 a = AddScore(board[1],board[7],board[13],board[19],board[25],"diag2")
 totscore += a;
 score.firstChild.nodeValue = totscore;
}
```

This function uses for loops to process each row and each column. It then handles the diagonals. A separate function, `AddScore()`, will handle the actual detection of poker hands in each of these.

The `totscore` variable stores a total of all of the scores. Each time a card is placed, the updated total score is displayed in the left column.

## Adding Up Scores

The `AddScore()` function is called by `Score()` for each column, row, and diagonal. This function determines which poker hand, if any, is represented by the cards passed to it. It then updates the appropriate score box on the board with the row's score, and returns the numeric value to be used by `Score()`. The `AddScore()` function begins by setting up some variables:

```
function AddScore(c1,c2,c3,c4,c5,scorebox) {
 obj=document.getElementById(scorebox);
 straight = false;
 flush = false;
 royal = false;
 pairs = 0;
 three = false;
 // sorted array for convenience
 nums = new Array(5);
 nums[0] = c1.num;
 nums[1] = c2.num;
 nums[2] = c3.num;
 nums[3] = c4.num;
 nums[4] = c5.num;
 nums.sort(numsort);
```

The function first sets up a number of flag variables, such as `straight` and `flush`, to keep track of which poker hand it finds. It then stores the five card values in an array and sorts it to make it easy to detect straights. The function continues by testing for each hand, one at a time. For example, this `if` statement tests for a flush by comparing card suits:

```
// flush
if (c1.suit == c2.suit && c2.suit == c3.suit
 && c3.suit == c4.suit && c4.suit == c5.suit) {
 flush = true;
}
```

After doing each test, `AddScore()` updates the board with a description of the poker hand and score for the row and returns a numeric score:

```
if (flush) {
 obj.innerHTML="Flush
5"
 return 5;
}
```

## Ending the Game

The game ends when all 25 spaces on the board have been filled with cards and the `EndGame()` function is called. Because the score is updated in real time and no moves can be made after all cards are placed, the only thing left for this function to do is to display a “Game Over” message:

```
function EndGame() {
 stat=document.getElementById("status");
 stat.innerHTML="Game Over";
}
```

This uses `innerHTML` to display a message in the `status` element, which normally displays “Next Card” to label the draw card.

## Adding Style with CSS

The game will also need a small CSS file to define the appearance of some of the game elements. Listing 22.2 shows the CSS file for the Poker Solitaire game.

---

**LISTING 22.2** The CSS File for the Poker Solitaire Game

---

```
h1 {
 font-size: 125%;
}
td.score {
 font-size: 80%;
 border: 1px solid silver;
 width: 53px;
}
#total {
 border: 1px solid black;
 font-size: 105%;
 padding: 5px;
}
#totalscore {
 text-align: center;
}
```

---

The CSS rules set the size of H1 headers, and then define a border, width, and font size for `td` elements in the `score` class, which will display each row’s score. Finally, a border, font size, and padding are defined for the “Total Score” display, and the numeric score is centered.



## Try It Yourself



### Putting It All Together

To get the game working, you'll need to use the complete JavaScript file that incorporates the functions you learned about earlier in this hour. Listing 22.3 shows the JavaScript file for the game.

#### LISTING 22.3 The Complete JavaScript File for the Poker Solitaire Game

```
// global variables
var tally = new Array(14)
var nextcard = 1;
var nexti = new Image(53,68);
// numeric comparison for sort()
function numsort(a, b) {
 return a - b;
}
function InitGame() {
 if (!document.getElementById) return;
 stat=document.getElementById("status");
 stat.innerHTML="Next Card";
 nextcard = 1;
// array for board contents
 board = new Array(26);
 for (i=1; i<26; i++) {
 board[i] = new Card(0,"x");
 obj=document.getElementById("card"+i);
 obj.src = "blank.gif";
 obj.onclick = PlaceCard;
 }
// fill the deck (in order, for now)
 deck = new Array(53);
 for (i=1; i<14; i++) {
 deck[i] = new Card(i,"c");
 deck[i+13] = new Card(i,"h");
 deck[i+26] = new Card(i,"s");
 deck[i+39] = new Card(i,"d");
 }
// Clear the scores
 Score();
// shuffle the deck
 n = Math.floor(100 * Math.random() + 200);
 for (i=1; i<n; i++) {
 c1 = Math.floor(52*Math.random() + 1);
 c2 = Math.floor(52*Math.random() + 1);
 temp = deck[c2];
 deck[c2] = deck[c1];
 deck[c1] = temp;
 }
// draw the first card on screen
 next=document.getElementById("dcard");
 next.src = deck[nextcard].fname();
// preload the next image
 nexti.src = deck[nextcard+1].fname();
```

**LISTING 22.3 Continued**


---

```

 obj=document.getElementById("newgame")
 obj.onclick=InitGame;
} // end InitGame
// place the draw card on the board where clicked
function PlaceCard(e) {
 if (!e) var e = window.event;
 // which space on the board was clicked?
 thiscard = (e.target) ? e.target: e.srcElement;
 pos = thiscard.id.substring(4);
 if (board[pos].suit != "x") {
 return;
 }
 drawcard=document.getElementById("dcard");
 thiscard.src = deck[nextcard].fname();
 drawcard.src = "blank.gif";
 board[pos] = deck[nextcard];
 nextcard++;
 Score();
 // Game over?
 if (nextcard > 25) {
 EndGame();
 }
 else {
 drawcard.src = deck[nextcard].fname();
 // cache next image for draw pile
 nexti = new Image(53,68);
 nexti.src = deck[nextcard+1].fname();
 }
}
// check for completed rows and display row scores
function Score() {
 score=document.getElementById("totalscore");
 totscore = 0;
 // rows
 for (x=0; x<5; x++) {
 r = x * 5 + 1;
 a =
AddScore(board[r],board[r+1],board[r+2],board[r+3],board[r+4],"row"+x);
 totscore += a;
 }
 // columns
 for (x=0; x<5; x++) {
 r = x + 1;
 a =
AddScore(board[r],board[r+5],board[r+10],board[r+15],board[r+20],"col"+x);
 totscore += a;
 }
 // diagonals
 a = AddScore(board[5],board[9],board[13],board[17],board[21],"diag1")
 totscore += a;
 a = AddScore(board[1],board[7],board[13],board[19],board[25],"diag2")
 totscore += a;
 score.firstChild.nodeValue = totscore;
}
// check for poker hands
function AddScore(c1,c2,c3,c4,c5,scorebox) {

```

**LISTING 22.3** Continued

---

```

 obj=document.getElementById(scorebox);
 straight = false;
 flush = false;
 royal = false;
 pairs = 0;
 three = false;
// sorted array for convenience
 nums = new Array(5);
 nums[0] = c1.num;
 nums[1] = c2.num;
 nums[2] = c3.num;
 nums[3] = c4.num;
 nums[4] = c5.num;
 nums.sort(numsort);
// no score if row is not filled
 if (c1.num == 0 || c2.num == 0 || c3.num == 0
 || c4.num == 0 || c5.num == 0) {
 obj.innerHTML="";
 return 0;
 }
// flush
 if (c1.suit == c2.suit && c2.suit == c3.suit
 && c3.suit == c4.suit && c4.suit == c5.suit) {
 flush = true;
 }
// straight
 if (nums[4] - nums[3] == 1
 && nums[3] - nums[2] == 1
 && nums[2] - nums[1] == 1
 && nums[1] - nums[0] == 1) {
 straight = true;
 }
// royal straight (10, J, Q, K, A)
 if (nums[1] == 10 && nums[2] == 11 && nums[3] == 12
 && nums[4] == 13 && nums[0] == 1) {
 straight = true;
 royal = true;
 }
// royal flush, straight flush, straight, flush
 if (straight && flush && royal) {
 obj.innerHTML="Royal Flush
250";
 return 250;
 }
 if (straight && flush) {
 obj.innerHTML="Straight Flush
50";
 return 50;
 }
 if (straight) {
 obj.innerHTML="Straight
4";
 return 4;
 }
 if (flush) {
 obj.innerHTML="Flush
5"
 return 5;
 }
// tally array is a count for each card value

```

**LISTING 22.3 Continued**

---

```

 for (i=1; i<14; i++) {
 tally[i] = 0;
 }
 for (i=0; i<5; i++) {
 tally[nums[i]] += 1;
 }
 for (i=1; i<14; i++) {
// four of a kind
 if (tally[i] == 4) {
 obj.innerHTML="Four of a Kind
25";
 return 25;
 }
 if (tally[i] == 3) three = true;
 if (tally[i] == 2) pairs += 1;
 }
// full house
 if (three && pairs == 1) {
 obj.innerHTML="Full House
8";
 return 8;
 }
// two pair
 if (pairs == 2) {
 obj.innerHTML="Two Pair
2";
 return 2;
 }
// three of a kind
 if (three) {
 obj.innerHTML="Three of a Kind
3";
 return 3;
 }
// just a pair
 if (pairs == 1) {
 obj.innerHTML="Pair
1";
 return 1;
 }
// nothing
 obj.innerHTML="No Score
0";
 return 0;
// end AddScore()
}
// game over - final score
function EndGame() {
 stat=document.getElementById("status");
 stat.innerHTML="Game Over";
}
// make a filename for an image, given Card object
function fname() {
 return this.num + this.suit + ".gif";
}
// constructor for Card objects
function Card(num,suit) {
 this.num = num;
 this.suit = suit;
 this.fname = fname;
}
// event handlers to start game
window.onload=InitGame;

```

---

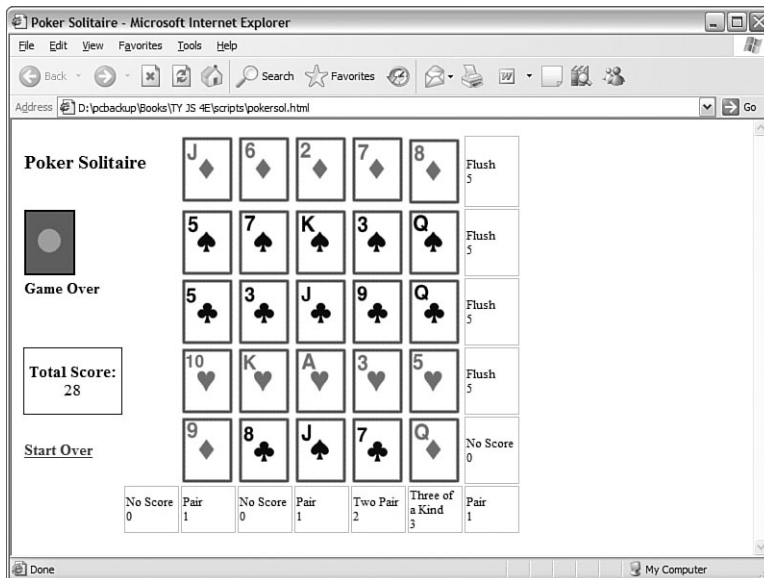
Because this is the longest code listing in this book, I recommend you download the files from this book's website rather than type it all in. You'll need the card graphics to make it work anyway.

**Watch  
Out!**

To try the game, make sure you have everything you need in one folder:

- ▶ The HTML document
- ▶ The CSS file (pokersol.css)
- ▶ The JavaScript file (pokersol.js)
- ▶ All 53 graphics (52 cards plus blank.gif)

You can now load the HTML file to test the game. Figure 22.3 shows the Poker Solitaire game after a complete game—it shouldn't take you long to beat my score.



**FIGURE 22.3**  
The Poker Solitaire example at the end of a game.

## Summary

In this hour, you've applied your JavaScript knowledge to create a complete application—a playable game. Along the way, you've used objects to represent playing cards, used graphics and the W3C DOM to display the game, and learned some of the issues involved in a complex application.

In the next hour, you'll return to practical applications of JavaScript with some advanced examples using the W3C DOM.

## Q&A

- Q.** *Because the spaces on the board aren't links, is there a way to make the cursor indicate that they can be clicked on?*
- A.** Yes. You can do this with an `onMouseOver` event handler that changes the `style.cursor` property to `pointer` for the spaces on the board. You could also use a rollover effect that changes the graphic, as demonstrated in Hour 19, "Using Graphics and Animation."
- Q.** *Can I add images or other HTML to the page without messing up the script?*
- A.** Yes. Because the game script works with `id` attributes rather than making any assumptions about which image objects to change, it shouldn't be affected by anything you add to the page, unless you use a conflicting `id` value.
- Q.** *What's a good strategy for playing this game?*
- A.** A simple approach is to dedicate each of the first four rows to a suit, so you have very good odds of scoring a flush on each row. As you do this, try to place cards where they'll form pairs with cards in other columns.

## Quiz Questions

Test your knowledge of the JavaScript techniques you used in this hour by answering the following questions.

- 1.** Which property of an `image` object do you change to display a different image?
  - a.** `href`
  - b.** `src`
  - c.** `fname`
- 2.** Which of the following statements converts the text "card21" to the number 21?
  - a.** `pos = thiscard.id.substring(4);`
  - b.** `pos = thiscard.id.numValue;`
  - c.** `pos = 1 * thiscard.id;`

3. Assuming Card objects have been defined as in this hour, which statement creates a new Card object?

- a. `c = Card(12, "s");`
- b. `c = new Card(12, "s");`
- c. `var c (Card);`

## Quiz Answers

- 1. b. You change the `src` property to display a different image.
- 2. a. The `substring()` method removes the first four letters of the string.
- 3. b. The `new` keyword is used to create a new instance of an object.

## Exercises

If you want to gain more experience creating games in JavaScript, try the following exercises:

- ▶ Spend some time playing the game and see if you find any bugs in the script. Notice how difficult it can be to fully test an application like this—you won't know for certain that it scores a royal flush correctly until you get one.
- ▶ Using the techniques described in Hour 19, try adding a rollover effect that changes the `blank.gif` appearance when you move over a square where you can drop the current card. Make sure the image does not change if there is already a card placed on the space.

*This page intentionally left blank*



## HOUR 23

# Creating JavaScript Applications

---

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

- ▶ Using the DOM to create a scrolling window
- ▶ Switching between CSS style sheets using JavaScript
- ▶ Using the DOM to create dynamic forms

You've learned quite a bit about JavaScript in the last 22 hours. In this hour, you'll apply this knowledge to create three quick, practical examples of JavaScript applications that could be useful for just about any website.

## **Creating a Scrolling Window**

One of the most common, and the most unfortunate, early uses of JavaScript was for scrolling messages, which crept across the browser's status line giving you information one letter at a time rather than making use of the whole page.

In this section, you'll create a different kind of scrolling message. This one scrolls a large block of text vertically within a window, similar to the credits at the end of a movie. This type of scrolling message is easier to read, is standards compliant, and can include links or other HTML features.

This example uses the same techniques as the animated slideshow in Hour 19, "Using Graphics and Animation." The only difference is that the animated text is only visible within a box, making it appear to scroll.

***By the  
Way***

---

## The HTML Document

The HTML document for this example includes a link to the script, a link to a CSS style sheet, the text displayed on the page, and the text that will be scrolled within the box. Listing 23.1 shows the HTML for this example.

---

**LISTING 23.1** The HTML Document for the Scrolling Window

---

```
<html>
<head>
<title>A DOM Scrolling Window</title>
<script language="JavaScript" type="text/javascript"
 src="scroll.js">
</script>
<link rel="stylesheet" type="text/css" href="scroll.css">
</head>
<body>
<h1>Scrolling Window Example</h1>
<p>This example shows a scrolling window created using JavaScript and
the W3C DOM. The red-bordered window below is actually a layer that
shows a clipped portion of a larger layer.</p>
<div id="thewindow">
<div id="thetext">
<p>This is the first paragraph of the scrolling message. The message
is created with ordinary HTML.</p>
<p>Entries within the scrolling area can use any HTML tags. They can
contain Links.</p>
<p>There's no limit on the number of paragraphs that you can include
here. They don't even need to be formatted as paragraphs.</p>

For example, you could format items using a bulleted list.

<p>The scrolling ends when the last part of the scrolling text
is on the screen. You've reached the end.</p>
</div>
</div>
</body>
</html>
```

---

The `<div>` tags in this document create two nested layers: One, `thewindow`, will form the small window for text to display in. The other, `thetext`, contains the text to scroll. You can use any HTML here, although it should be able to wrap to the small window.

## The CSS File

The CSS file for this example, shown in Listing 23.2, sets margins and borders for the two `<div>` elements. The box's position property is set to relative, so it will be laid out normally within the document, and the position property for the scrolling text is set to absolute so it can be repositioned by the script.

**LISTING 23.2** The CSS Style Sheet for the Scrolling Window

---

```
#thewindow {
 position:relative;
 width:180;
 height:150;
 overflow:hidden;
 border: 2px solid red;
}
#thetext {
 position: absolute;
 width: 170;
 left: 5;
 top: 100;
}
```

---

Because the text doesn't all fit in the small window, you'll only see part of it at a time. The `overflow` property on the window layer prevents the rest of the content from showing. Your script will manipulate the scrolling text's `style.top` property to move it relative to the window, creating a scrolling effect.

The text layer is actually 10 pixels narrower than the window layer. This, along with the `left` property, creates a small margin of white space on either side of the window, preventing any of the text from being obstructed.

**By the  
Way**

## The JavaScript File

The JavaScript code for this example uses a function, `Scroll()`, that is called repeatedly by a timeout. Listing 23.3 shows the JavaScript file for this example.

**LISTING 23.3** The JavaScript File for the Scrolling Window

---

```
// global variable for position of the scrolling window
var pos=100;
function Scroll() {
 if (!document.getElementById) return;
 obj=document.getElementById("thetext");
 pos -=1;
 if (pos < 0-obj.offsetHeight+130) return;
 obj.style.top=pos;
 window.setTimeout("Scroll();",30);
}
// Start scrolling when the page loads
window.onload = Scroll;
```

---

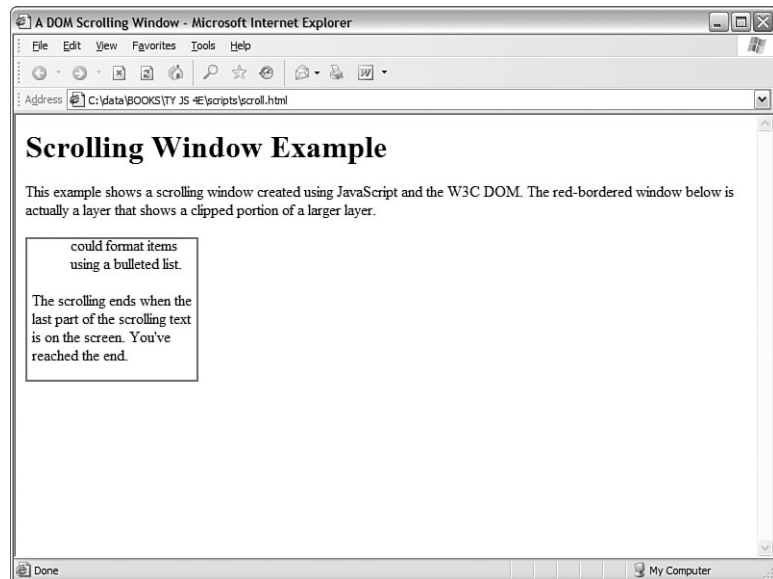
The first line defines a global variable, `pos`, to store the current scroll position. The `Scroll()` function subtracts 1 from `pos` and checks its value. If the scrolling has reached the end, the function exits; otherwise, it sets the object position and calls the `Scroll()` function again using a timeout.

***Did you know?***

Notice the if statement at the beginning of the function. This is a simple example of feature sensing, described in Hour 15, “Unobtrusive Scripting”—if the browser doesn’t support the `getElementById()` method, the function exits rather than cause errors.

To try this example, make sure you have all three files in the same folder: the HTML document, the CSS file (`scroll.css`), and the JavaScript file (`scroll.js`) and load the HTML document into a browser. Figure 23.1 shows this example in action, after the scrolling text has reached the end.

**FIGURE 23.1**  
The scrolling text box example in action.



## Style Sheet Switching with JavaScript

Suppose you want to offer your visitors a choice of different ways of viewing your site—for example, a choice of large or small fonts, or different background colors. Although you can use the `style` properties of elements within a page to make these changes individually, it would take a lot of code to change a page between drastically different styles.

One alternative is to create two or more completely separate style sheets, and use JavaScript to switch between them. This allows the user to have a large amount of control over the site’s appearance without using a large and complex script.

## Creating the HTML Document

First, you can create a basic HTML document for the style-switching example. This document will include a `<script>` tag for the script you'll create later, as well as links to two different style sheets. The HTML document for this example is shown in Listing 23.4.

### LISTING 23.4 The HTML Document for the Style-Switching Example

---

```
<html>
<head>
<title>Style Sheet Example</title>
<link rel="stylesheet" type="text/css" href="style1.css">
<link rel="stylesheet" type="text/css" href="style2.css" disabled>
<script language="javascript" type="text/javascript"
 src="styleswitch.js">
</script>
</head>
<body>
<h1>multiple-choice styles</h1>
<p>This is a standard paragraph of text. Its font, margins,
colors, justification, and other attributes depend on the style
sheet you select. This paragraph includes some text in
bold and <i>italics</i>.
</p>
<p>You can select one of three styles for this document:
</p>

Style sheet # 1
Style sheet # 2
No style sheet

<p>These links call a short JavaScript function that enables one
of this document's two linked external style sheets. You can edit
the style sheets to style this document in two different ways,
without changing any HTML.</p>
</body>
</html>
```

---

Although most of the document is just sample text to show off the styles of the different style sheets, it has several important components to make this technique work:

- ▶ The `<script>` tag uses the `src` attribute to include a script, `styleswitch.js`.
- ▶ There are two `<link>` tags to attach two external style sheets, `style1.css` and `style2.css`. The second tag includes the `disabled` attribute, so the document will be styled using only `style1.css` by default.
- ▶ The three links within the `<li>` list items have event handlers that call the `Style()` function to switch styles.

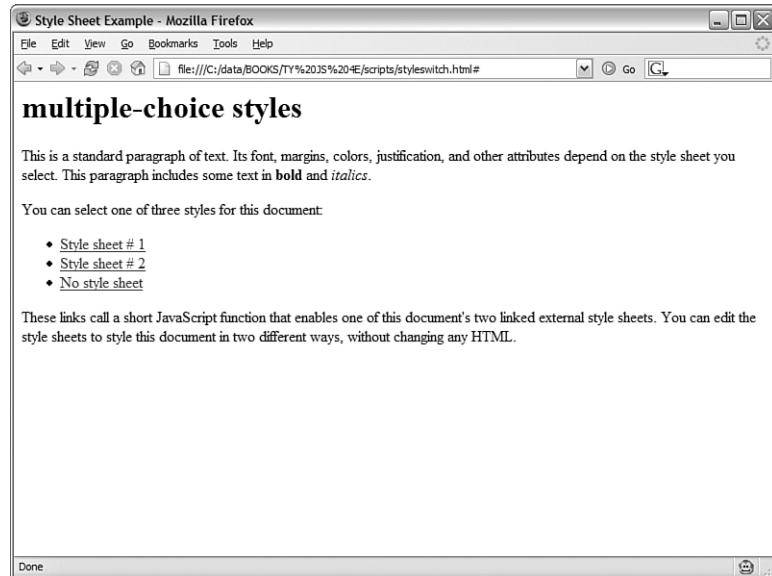
## By the Way

Some browsers don't correctly support the disabled attribute in HTML. The script you create later will use JavaScript to disable the second style sheet by default to ensure that only one style sheet is used, regardless of the browser.

Save the HTML document in a folder. You'll be adding two style sheets and a script file to the folder to complete the example. If you load the document into a browser before creating the style sheets, it will be displayed without styles. Figure 23.2 shows how the document looks with no styles applied.

**FIGURE 23.2**

The style-switching example displayed without styles.



## Creating the First Style Sheet

Next, you can create the first of the two style sheets. Listing 23.5 shows the complete style sheet `style1.css`.

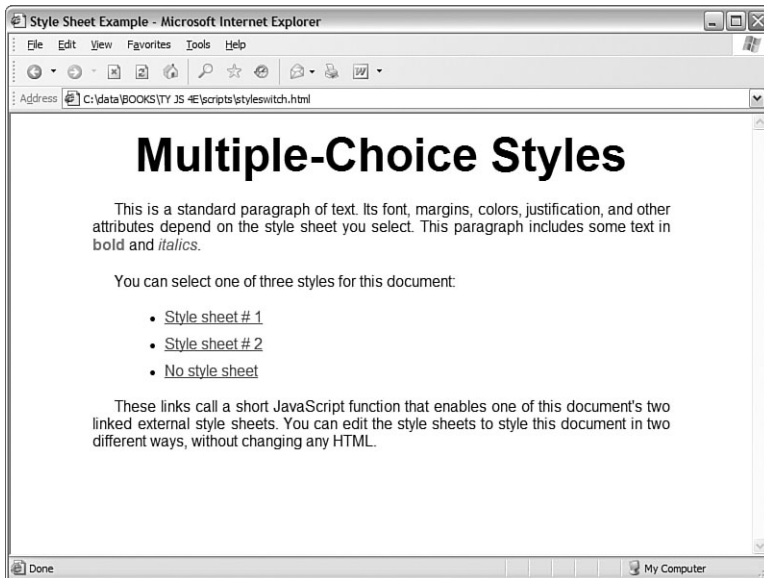
**LISTING 23.5** The First Style Sheet for the Style-Switching Example (`style1.css`)

```
body {
 font-family: Arial, Helvetica, sans-serif;
 font-size: 12pt;
}
P {
 margin-left: 10%;
 margin-right: 10%;
 text-align: justify;
 text-indent: 3%;
}
```

**LISTING 23.5** Continued

```
B { color: red; }
I { color: DarkViolet; }
H1 {
 font-size: 300%;
 text-align: center;
 text-transform: capitalize;
}
UL {
 margin-left: 20%;
 margin-right: 20%;
}
LI { margin-top: 10px;}
```

Save this style sheet as `style1.css` in the same folder as the HTML document. This style sheet assigns some basic styles to the body, and to specific tags: `<p>`, `<h1>`, and so on. Because this is the default style sheet, it will be used if you load the HTML document now. Figure 23.3 shows the document as styled by this style sheet.



**FIGURE 23.3**  
The style-switching example using the first style sheet.

## Creating the Second Style Sheet

The second style sheet, `style2.css`, uses some more dramatic styles and is unlikely to be suited to all viewers. This sheet is disabled by default. Listing 23.6 shows the second style sheet.

**LISTING 23.6** The Second Style Sheet for the Style-Switching Example (style2.css)

---

```
body {
 font-family: Times, "Times New Roman", sans-serif;
 font-size: 14pt;
}
P {
 margin-left: 20%;
 margin-right: 20%;
 text-align: left;
 text-indent: 0%;
}
B {
 color: black;
 background-color: aqua;
}
I { color: red;}
H1 {
 font-size: 200%;
 text-align: right;
 text-transform: uppercase;
}
UL {
 margin-left: 30%;
 margin-right: 30%;
 background-color: yellow;
}
LI { margin-top: 20px;}
```

---

Save this style sheet as style2.css in the same folder as the HTML document.

## Creating the Script

You can use JavaScript to enable or disable style sheets. The <link> elements that you used to attach the two style sheets to the HTML document are objects in the DOM, and you can manipulate them using DOM methods. In this example, you will use the `getElementsByTagName()` method to find all of the <link> elements, and then enable one and disable the other. Listing 23.7 shows the complete JavaScript file.

**LISTING 23.7** The JavaScript File for the Style-Switching Example (styleswitch.js)

---

```
function Style(n,enable) {
 if (!document.getElementsByTagName) return;
 links = document.getElementsByTagName("link");
 links[n].disabled=!enable;
 links[1-n].disabled=true;
}
Style(0,true);
```

---



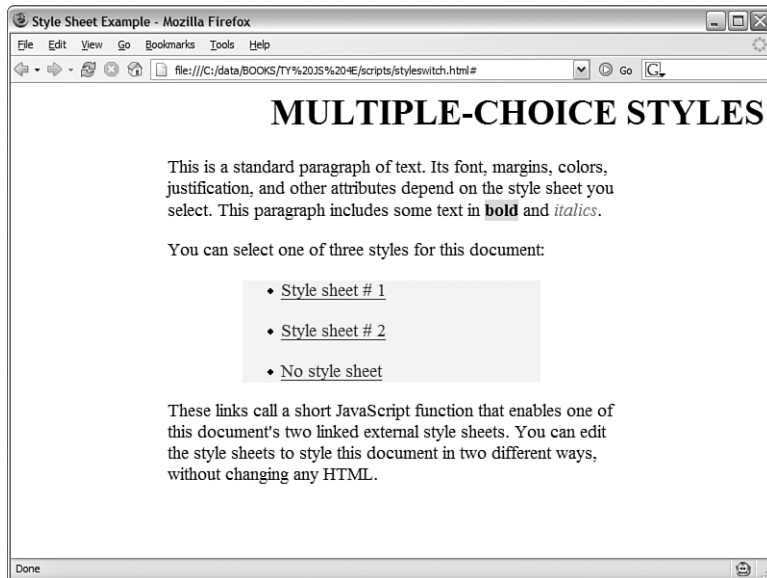
This script defines the `Style()` function, which accepts two parameters. The first, `n`, specifies the style sheet to activate. The second parameter, `enable`, specifies whether to enable the new style sheet (`true`) or to disable all style sheets (`false`). This feature is used by the No Style Sheet link.

This example uses `getElementsByTagName`, but you could also assign an `id` attribute to each `<link>` tag and then use `document.getElementById` to find the object for each one individually.

***Did you  
Know?***

The script enables (or disables, depending on the parameter) the chosen style sheet, and always disables the other sheet. The last line of the script calls the `Style()` function to select the first style sheet, just in case the browser doesn't support the disabled attribute.

To try the example, make sure you have all four files in the same folder: The HTML document, the two style sheets (`style1.css` and `style2.css`), and the script file (`styleswitch.js`). Load the HTML document into a browser and try clicking the links to change styles. Figure 23.4 shows the document after the second style sheet has been selected.



**FIGURE 23.4**  
The style-switching example with the second style sheet selected.



## Try It Yourself

### Creating a Dynamic Form

In Hour 11, “Getting Data with Forms,” you learned how JavaScript can work with data from HTML forms, and change form elements. Using the W3C DOM, you can take this one step further, creating a script that can add elements to a form or show or hide sections of a form.

### Creating the HTML Document

Listing 23.8 shows the HTML document for this example, which defines an order form. The form will have two dynamic features: first, the Ship To address fields aren’t shown unless they’re needed, and second, a button enables you to add additional item fields to the form.

---

#### **LISTING 23.8** The HTML Document for the Dynamic Form Example

---

```
<html>
<head>
<title>Dynamic Order Form</title>
<script language="JavaScript" type="text/javascript"
 src="dform.js">
</script>
</head>
<body>
<h1>Order Form</h1>
<hr>
<form name="form1">
Bill to:

Name: <input type="text" name="customer" size="20">

Address 1: <input type="text" name="addr1" size="20">

Address 2: <input type="text" name="addr2" size="20">

City: <input type="text" name="city" size="15">
State: <input type="text" name="state" size="4">
Zip: <input type="text" name="zip" size="9">
<hr>
Ship to:

<input type="radio" name="shiptopt" value="same" checked onClick="Show(0);">
Same Address
<input type="radio" name="shiptopt" value="other" onClick="Show(1);">
Other Address
<div ID="shipto" style="display: none;">

Name: <input type="text" name="shipname" size="20">

Address 1: <input type="text" name="shipaddr1" size="20">

Address 2: <input type="text" name="shipaddr2" size="20">

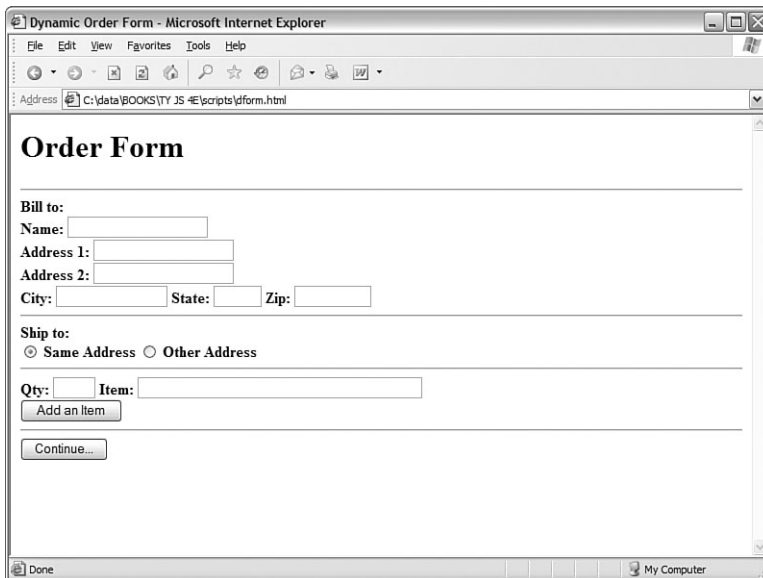
City: <input type="text" name="shipcity" size="15">
State: <input type="text" name="shipstate" size="4">
Zip: <input type="text" name="shipzip" size="9">
</div>
<hr>
<div ID="items">
Qty:
```

**LISTING 23.8 Continued**

```
<input type="text" name="qty1" size="3">
Item:
<input type="text" name="item1" size="45">

<input type="button" value="Add an Item"
onClick="AddItem();" ID="add">
</div>
<hr>
<input type="submit" value="Continue...">
</form>
</body>
</html>
```

Save this HTML document in a folder. If you load it into a browser, you'll see the form's default appearance, but the dynamic features won't work yet. Figure 23.5 shows the default look of the form.

A screenshot of a Microsoft Internet Explorer browser window displaying an "Order Form". The browser's address bar shows the file path "C:\data\BOOKS\TY JS -E\scripts\dford.html". The form itself has a title "Order Form" and is divided into sections. The "Bill to:" section includes input fields for "Name:", "Address 1:", "Address 2:", "City:", "State:", and "Zip:". Below this is the "Ship to:" section with radio buttons for "Same Address" (selected) and "Other Address". Further down are input fields for "Qty:" and "Item:", followed by an "Add an Item" button. At the bottom of the form is a "Continue..." button. The browser's status bar at the bottom shows "Done" and "My Computer".

**FIGURE 23.5**  
The dynamic  
form before the  
script is added.

## Adding the Script

The script for this example will include two functions: `AddItem()`, for adding items to the form, and `Show()`, for showing or hiding the ship-to address. Listing 23.9 shows the script file.

**LISTING 23.9 The JavaScript File for the Dynamic Form Example**


---

```
// global variable
var items=1;
function AddItem() {
 if (!document.getElementById) return;
 // Add an item to the form
 div=document.getElementById("items");
 button=document.getElementById("add");
 items++;
 newitem="Qty: ";
 newitem+="<input type=\"text\" name=\"qty\" + items;
 newitem+="\" size=\"3\"> ";
 newitem+="Item: ";
 newitem+="<input type=\"text\" name=\"item\" + items;
 newitem+="\" size=\"45\">
";
 newnode=document.createElement("span");
 newnode.innerHTML=newitem;
 div.insertBefore(newnode,button);
}
function Show(a) {
 if (!document.getElementById) return;
 //Hide or show ship-to address
 obj=document.getElementById("shipto");
 if (a) obj.style.display="block";
 else obj.style.display="none";
}
```

---

Here's a breakdown of how this script works:

- ▶ The first line defines a global variable, `items`, to keep track of the number of items. This is used to assign a unique name attribute to each `<input>` tag as they are added.
- ▶ The `AddItem()` function adds additional Quantity and Item fields to the form using the `insertBefore()` DOM method.
- ▶ The `Show()` function uses the `style.display` property to either show or hide the section with the id value `shipto`.

To try the script, save it as `dform.js` (or download the files from this book's website) and load the HTML document into a browser. Figure 23.6 shows the document with two additional item fields added and the ship-to address displayed.

**Dynamic Order Form - Mozilla Firefox**

File Edit View Go Bookmarks Tools Help

file:///C:/data/BOOKS/TV%20JS%20%E/scripts/dform.html

## Order Form

**Bill to:**

Name:

Address 1:

Address 2:

City:  State:  Zip:

**Ship to:**

☐ Same Address ☒ Other Address

Name:

Address 1:

Address 2:

City:  State:  Zip:

Qty:  Item:

Qty:  Item:

Qty:  Item:

Done

**FIGURE 23.6**  
The dynamic  
form in action.

## Summary

In this hour, you put your knowledge of JavaScript and the DOM to work with three examples: a scrolling text box, a page with user-selectable styles, and a dynamic form. Each of these could serve as the basis for a much more sophisticated feature for a site.

Your 24-hour tour of JavaScript is nearly over. In the final hour of this book, you'll learn about the future of JavaScript, learn what other web languages and disciplines you might want to learn next, and create one final code example.

## Q&A

**Q. Can I make text scroll horizontally rather than vertically?**

**A.** Yes, the scrolling text example could easily work horizontally by changing the `left` property rather than the `top` property. However, this will be confusing unless the text is designed for horizontal scrolling—a single line would work fine.

**Q. Why don't more sites use dynamic forms?**

**A.** There are some usability and accessibility issues with a dynamic form—for starters, if JavaScript is disabled, the form in this hour would be limited to ordering a single item. You could compensate for this with some server-side code to allow adding additional items (slowly), but it would be far more complex than a simple form.

- Q.** *Can my script enable more than one style sheet at a time?*
- A.** Yes, you can have any number of `<link>` tags for style sheets, and any or all of them can be enabled. One obvious approach is to have one common style sheet that is always enabled, and use the script to enable or disable additional style sheets for user preferences.

## Quiz Questions

Test your knowledge of the DOM by answering the following questions.

1. In the scrolling-text example, which CSS rule prevents the scrolling text from being visible outside the box?
  - a. `overflow: hidden;`
  - b. `position: relative;`
  - c. `position: absolute;`
2. Which property of a `<link>` element determines whether the style sheet affects the document?
  - a. `enabled`
  - b. `disabled`
  - c. `active`
3. In the dynamic forms example, which DOM method is used to add additional fields to the form?
  - a. `insertAfter`
  - b. `addItem`
  - c. `insertBefore`

## Quiz Answers

1. a. The `overflow: hidden;` rule prevents the text from being visible outside the box.
2. b. The `disabled` property controls whether the style sheet affects the document.
3. c. The `insertBefore` method is used to add additional items before the Add an Item button.

## Exercises

If you want to gain more experience working with the techniques you explored in this hour, try the following exercises:

- ▶ Create your own text for the scrolling text box, and try modifying the HTML document (refer to Listing 23.1) to scroll your text.
- ▶ Create a third style sheet for the style-switching example, and modify the HTML document and the script (refer to Listings 23.4 and 23.7) to allow switching between three different style sheets.
- ▶ Right now, the dynamic forms example in Listings 23.8 and 23.9 will never display the Ship To address if JavaScript is disabled. To improve the accessibility of the form, make the script hide the Ship To section rather than having it hidden by default in the HTML document.

*This page intentionally left blank*



## Hour 24

# Your Future with JavaScript

---

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

- ▶ Where to go to learn more about JavaScript
- ▶ How future versions of JavaScript might affect your scripts
- ▶ An introduction to XML (Extensible Markup Language)
- ▶ XHTML (Extensible Hypertext Markup Language)
- ▶ How to be sure you're ready for future web technologies
- ▶ How to move on to other web languages
- ▶ Implementing drag-and-drop using JavaScript

You've reached the last hour of this book. In this final hour, you'll find some ideas of where to go next—whether you want to learn more about JavaScript or move on to other languages and technologies. You'll also learn some tips for future-proofing your scripts, and you'll create a drag-and-drop script as a final example.

## **Learning Advanced JavaScript Techniques**

Although you've now learned all of the essentials of the JavaScript language, there is still much to learn. JavaScript can be used to script environments other than the Web, and you can move beyond simple scripts to develop entire applications that combine JavaScript with server-side programming.

Here are some ways you can further your JavaScript education:

- ▶ See Appendix A, "Other JavaScript Resources," for a list of JavaScript books and web pages with further information.
- ▶ While the core JavaScript language is in place, be sure to follow the latest developments. The websites in Appendix A and this book's site will let you know when changes are on the way.

- Be sure to spend some time practicing the JavaScript techniques you’ve learned throughout this book. You can use them to create much more complex applications than those you’ve worked with so far.

One advanced technique that is becoming popular is AJAX (Asynchronous JavaScript and XML), which allows JavaScript to communicate with a server without reloading pages. You learned the basics of AJAX in Hour 17, “AJAX: Remote Scripting”.

Because this is one of the most exciting features of JavaScript, it’s a good one to learn more about. Try using the AJAX library you created in Hour 17 to add remote scripting to a site, or explore the Web’s AJAX sites to learn more sophisticated techniques.

## Future Web Technologies

The Web has changed dramatically in the last 10 years, and is continually changing. In the following sections, you will explore some of the upcoming—and already developed—technologies that will affect your pages and scripts.

### Future Versions of JavaScript

JavaScript has gone through several versions to reach its current one, 1.6. Fortunately, the core language hasn’t changed much through these version changes, and nearly all scripts written for older versions will work on the latest one.

The next version of JavaScript, 2.0, is currently being developed by the Mozilla Foundation and ECMA. Version 2.0’s main change will be the addition of true object-oriented features, such as classes and inheritance.

As with previous versions, 2.0 should be backward compatible with older versions. To be sure your scripts will work under version 2.0, follow the standard language features and do not rely on any undocumented or browser-specific features.

### Future DOM Versions

Currently, the W3C DOM level 1 is an official specification, whereas level 2 is only a recommendation. Level 2 adds features such as event handling and better style sheet support, and is already partially supported by the latest browsers.

In the future, expect better browser support for the DOM, and less compatibility issues between browsers.

## XML (Extensible Markup Language)

HTML was originally created as a language for the Web, and was based on an older standard for documentation called SGML (Standard Generalized Markup Language). HTML is a much-simplified version of SGML, specifically designed for web documents.

A relatively new language on the scene is XML (Extensible Markup Language). XML is also a simplified version of SGML, but it isn't nearly as simple as HTML. Although HTML has a specific purpose, XML can be used for virtually any purpose.

The W3C (World Wide Web Consortium) developed XML, and has published a specification to standardize the language.

**By the  
Way**

Strictly speaking, XML isn't a language in itself—there is no concise list of XML tags because XML has no set list of tags. Instead, XML enables you to create your own markup languages for whatever purpose you choose.

So what use is a language without any specific commands? Well, XML enables you to define tags, similar to HTML tags, for any purpose. If you were storing recipes, for example, you could create tags for ingredients, ingredient quantities, and instructions.

XML uses a DTD (Document Type Definition) to define the tags used in a particular document. The DTD can be in a separate file or built into the document, and specifies which tags are allowed, their attributes, and what they can contain.

XML is already in use today. Although it isn't directly supported by web browsers, you can use a program on the server to parse XML documents into HTML documents before sending them to the browser.

To return to the recipe example, an XML processor could convert each recipe into HTML. The reason for doing this is simple: By changing the rules in the parser, you could change the entire format of all of the recipes—a difficult task to perform manually if you had thousands of recipes.

## XHTML (Extensible Hypertext Markup Language)

The HTML specification, at version 4.01, is still considered valid, but the W3C has been working on the successor to HTML, XHTML, now at version 1.1. XHTML is a reformulated version of HTML that fits the strict rules of XML and can be processed with software designed to work with XML.

In practice, XHTML looks very similar to HTML. Here are some of the most obvious changes you will need to make to adapt a page to XHTML:

- ▶ All tags should be lowercase: `<p>`, `<body>`, and so forth.
- ▶ Most tags require closing tags: `</p>`, and so forth.
- ▶ For standalone tags that don't enclose other elements, such as `<img>` and `<br>`, a special syntax combines opening and closing tags with a slash before the closing brace: `<br/>`.
- ▶ The document must follow strict rules of structure and tags must be nested correctly.
- ▶ A `<!DOCTYPE>` tag is required to specify the XML DTD used for the document. The following specifies the XHTML Transitional DTD:

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

The transitional DTD allows some deprecated HTML tags, such as `<center>`, for compatibility. There is also an XHTML Strict DTD that does not allow any deprecated tags.

### **By the Way**

Browser support for XHTML isn't perfect, especially when it comes to the Strict DTD. It's also difficult to meet XHTML's strict validation requirements while dealing with issues such as user-generated content and scripts. For these reasons, most webmasters use either XHTML Transitional or the still-valid HTML 4.01.

## **XSL (Extensible Stylesheet Language)**

XML documents focus strictly on the meaning of the tags—content—and ignore presentation. The presentation of XML can be determined by creating an XSL (Extensible Stylesheet Language) style sheet.

XSL is based on XML, but specifies presentation—parameters such as font size, margins, and table formatting—for an XML document. When you use an XML processing program to create HTML output, it uses an XSL style sheet to determine the HTML formatting of the output.

### **By the Way**

XSL documents are actually XML documents, using their own DTD that specifies style sheet tags. XSL is a newer W3C specification.

## Planning for the Future

In the history of JavaScript, there has never been such a major change to the language that a great number of scripts written using the older version have stopped working. Nevertheless, many scripts have been crippled by new browser releases—chiefly those that used browser-specific features.

The following sections offer some guidelines you can follow in writing scripts to ensure that the impact of future JavaScript versions and browser releases will be minimal.

### Keeping Scripts Compatible

Years ago, Netscape and Microsoft introduced separate and incompatible versions of DHTML (Dynamic HTML), which allowed scripts to modify any element of a page for the first time. Early adopters jumped in to write many scripts, some of which you can still find online today. These scripts made some serious mistakes:

- ▶ Browser detection was used to separately support browsers, or in some cases a specific browser was required.
- ▶ Scripts were written to work around bugs in browsers, or sometimes even take advantage of them.
- ▶ The process of writing scripts often involved trial and error rather than consulting official documentation.

This messy scripting gave DHTML—and JavaScript—a bad name among serious programmers. Fortunately, the standardized W3C DOM has now replaced the proprietary browser DHTML features, and it's easier than ever to create scripts the right way—but as time goes by, there will undoubtedly be cutting-edge features that aren't quite standard.

One obvious example is AJAX (Asynchronous JavaScript and XML), which is only now being developed as a standard by the W3C, despite working (in sometimes confusingly different ways) in the major browsers.

There's nothing wrong with using these cutting-edge features—but if you do, you should be aware that you're going to need to test the scripts on several different browsers. You should use feature sensing rather than detecting (or expecting) particular browsers. Finally, you should be prepared to do a bit of rewriting when the standard arrives.

## Staying HTML Compliant

One trend as browsers advance is that newer browsers tend to do a better job of following the W3C standard for HTML—and, often, relying on it. This means that although a page that uses completely standard HTML will likely work in future browsers, one that uses browser-specific features or workarounds is bound to have problems eventually.

### By the Way

In particular, the first release of Netscape 6.0 received many complaints about “breaking” previously working pages. In most cases, the page used bad HTML, and previous browsers happened to handle the error more gracefully.

To avoid these problems, try to use completely valid HTML whenever possible. This means not only using standard tags and attributes, but following certain formatting rules: For example, always using both opening and closing `<p>` tags, and enclosing numbers for table widths and other parameters in quotation marks.

To be sure your documents follow the HTML standard, see Appendix B, “Tools for JavaScript Developers,” for suggested HTML validation programs and services. These will examine your document and point out any areas that do not comply with the HTML standard.

## Document Everything

Last but not least, be sure you understand everything your scripts are doing.

Document your scripts using comments, and particularly document any statements that might look cryptic or are particularly hard to get working correctly.

If your scripts are properly documented, it will be a much easier process if you have to modify them to be compatible with a future browser, JavaScript, or DOM version.

### Did you Know?

See Hour 15 for more tips on future-proofing your scripts by using unobtrusive scripting techniques.

## Moving on to Other Languages

Assuming you’ve spent the last 24 hours learning JavaScript to further your career (or hobby) as a web developer, where will you go next? As you should know by now, JavaScript can’t do everything, and there are many other languages that work on the Web. Here are some you might want to explore:

- ▶ **Java** is useful for more complex client-side programs. Although Java applets aren't as integrated with web pages as JavaScript, you can build applications that go beyond JavaScript's capabilities. Java's syntax is similar to JavaScript, although the language is more complicated. See <http://java.sun.com/> for more information.
- ▶ **Flash** is also a popular choice for more sophisticated client-side programs, and is an especially good choice if you want to create games or applications that work with video. Flash's ActionScript is based on the same ECMAScript standard as JavaScript, so you have a headstart. See <http://www.adobe.com/products/flash/flashpro/> for more information.
- ▶ **Ruby** is a relatively new server-side language that has taken the web development world by storm, particularly thanks to the Ruby on Rails framework. Ruby on Rails includes features for easily integrating JavaScript and AJAX features into sites. See <http://www.ruby-lang.org/en/> and <http://www.rubyonrails.org/> for details.
- ▶ **PHP** is the workhorse of server-side languages, and a popular choice for back-end development, whether with basic HTML or JavaScript and AJAX front ends. See <http://www.php.net/> for details.
- ▶ **Python** is another popular server-side language, noted for its simple coding style and the excellent libraries available for adapting it to various purposes. See <http://www.python.org/> for more information.

There are many other languages on the Web, but these are five popular choices. It's worth taking the time to learn a bit about these languages and others even if you don't plan on making them your primary development tool.

## Try It Yourself



### Creating Drag-and-Drop Objects

Just to prove JavaScript can do many things beyond what you've learned so far, here is a final example. Although desktop operating systems support drag-and-drop actions (for example, moving a file into the trash can), web pages have traditionally lacked this feature. Using JavaScript and the DOM, you can unobtrusively create objects that the user can pick up, drag, and drop.

This is a simple implementation of drag-and-drop. Full-featured dragging and dropping leads to a very complex script. Fortunately, you can use JavaScript libraries such as [Script.aculo.us](http://script.aculo.us) to add drag-and-drop to your pages without any scripting. See Hour 8, "Using Built-in Functions and Libraries," for more details.

**Did you  
Know?**

## The HTML Document

The HTML document for this example exists mainly to define four draggable objects with `<div>` tags. Listing 24.1 shows the complete HTML document.

---

**LISTING 24.1** The HTML Document for the Drag-and-Drop Example

---

```
<html>
<head>
<title>Drag and Drop</title>
<link rel="stylesheet" type="text/css" href="dragdrop.css">
<script language="javascript" type="text/javascript"
 src="dragdrop.js">
</script>
</head>
<body>
<h1>Drag and Drop in JavaScript</h1>
<div class="drag" id="drag1">
<h3>Box #1</h3>
<p>Click one of these boxes and hold the
mouse button down to move it to a new location.</p>
</div>
<div class="drag" id="drag2">
<h3>Box #2</h3>
<p>This is another box you can drag and drop.</p>
</div>
<div class="drag" id="drag3">
<h3>Box #3</h3>
<p>This is yet another box you can drag and drop.</p>
</div>
<div class="drag" id="drag4">
<h3>Box #4</h3>
<p>This is the fourth and final draggable box.</p>
</div>
</body>
</html>
```

---

Each of the `<div>` tags with the `class="drag"` attribute will be a draggable object. The document also includes a `<script>` tag to attach a script and a `<link>` tag for a style sheet.

## The CSS Style Sheet

The style sheet sets up the four positionable objects with an initial position as well as a distinctive border. Listing 24.2 shows the CSS file for this example.

---

**LISTING 24.2** The CSS File for the Drag-and-Drop Example

---

```
.drag {
 position: absolute;
 width: 150px;
 border: 2px solid black;
 border-top: 20px solid black;
```

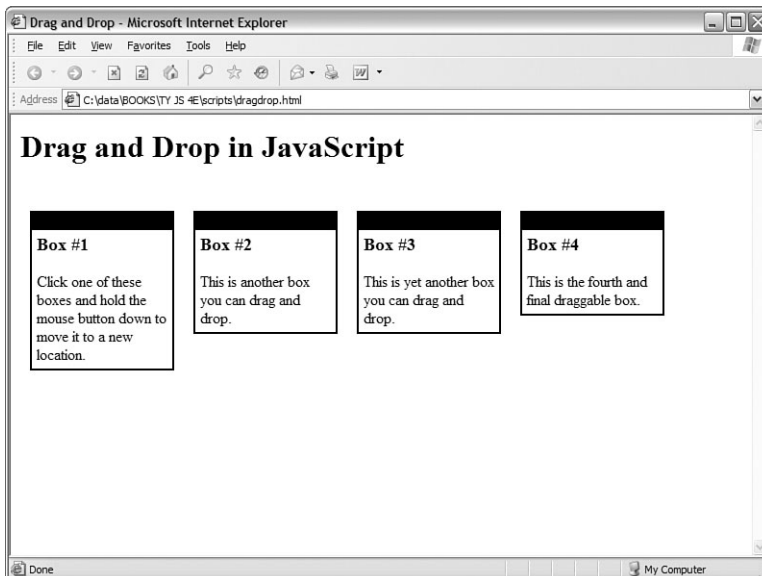


**LISTING 24.2** Continued

```
 top: 100px;
 padding: 5px;
}
#drag1 { left: 20px; }
#drag2 { left: 190px; }
#drag3 { left: 360px; }
#drag4 { left: 530px; }
```

The `position: absolute` rule makes the elements positionable. The `top` property sets the vertical position of all four elements, and the `left` property is set for each one to space them across the page. The width and border properties make the `<div>` elements look like boxes, and the `border-top` property creates a thick top border for dragging.

Save this file as `dragdrop.css` in the same folder as the HTML document. If you load the HTML document into a browser at this point, you can see the styled boxes, but they won't be movable until you add the script. Figure 24.1 shows this example before adding the script.



**FIGURE 24.1**  
The initial display of the draggable objects.

## Implementing Drag-and-Drop

Because drag-and-drop isn't built in to the DOM, your script will have to do it the hard way. When the user clicks on an element, an `onmousedown` event handler will begin dragging the object. After that starts, an `onmousemove` event handler will update the object's position, and `onmousedown` will "drop" the object.

One tricky part is determining the mouse position in the `onmousemove` event handler. This is stored as a property of the event object, but Netscape and Firefox use the `pageX` and `pageY` properties, whereas Internet Explorer uses the `clientX` and `clientY` properties. A series of `if` statements finds the `x` and `y` values, regardless of the browser:

```
if (!e) var e = window.event;
if (e.pageX) {
 x = e.pageX;
 y = e.pageY;
} else if (e.clientX) {
 x = e.clientX;
 y = e.clientY;
} else return;
```

### **By the Way**

See Hour 9, “Responding to Events,” for more information on event handlers and the event object.

One more issue: Objects are positioned based on their top-left corner, but you can click anywhere on the object with the mouse. This will result in a “jump” effect when you pick up an object. The solution is to calculate an offset between the mouse position and the object’s position:

```
dx = x - obj.offsetLeft;
dy = y - obj.offsetTop;
```

When the object is moved, these offsets will be subtracted from the mouse position. This way, the object is anchored to the mouse pointer wherever you click it, and does not jump to a new position.

## **The JavaScript File**

Now all you need is the JavaScript file to add the drag-and-drop feature to the document. Listing 24.3 shows the complete script.

### **LISTING 24.3 The JavaScript File for the Drag-and-Drop Example**

```
// global variables
var obj,x,y,dx,dy;
// set up draggable elements
function Setup() {
 // exit if the browser doesn't support the DOM
 if (!document.getElementsByTagName) return;
 divs = document.getElementsByTagName("DIV");
 for (i=0; i<divs.length; i++) {
 if (divs[i].className != "drag") continue;
 // set event handler for each div with class="drag"
 divs[i].onmousedown = Drag;
 }
}
```

**LISTING 24.3** Continued

---

```
}
function Drag(e) {
 // Start dragging an object
 if (!e) var e = window.event;
 // which object was clicked?
 obj = (e.target) ? e.target : e.srcElement;
 obj.style.borderColor="red";
 // calculate object offsets from mouse position
 dx = x - obj.offsetLeft;
 dy = y - obj.offsetTop;
}
function Move(e) {
 // track mouse movements
 if (!e) var e = window.event;
 if (e.pageX) {
 x = e.pageX;
 y = e.pageY;
 } else if (e.clientX) {
 x = e.clientX;
 y = e.clientY;
 } else return;
 if (obj) {
 obj.style.left = x - dx;
 obj.style.top = y - dy;
 }
}
function Drop() {
 // let go!
 if (!obj) return;
 obj.style.borderColor="black";
 obj = false;
}
// Detect mouse movement
document.onmousemove = Move;
// drop current object on mouse up
document.onmouseup = Drop;
// Set up when the page loads
window.onload = Setup;
```

---

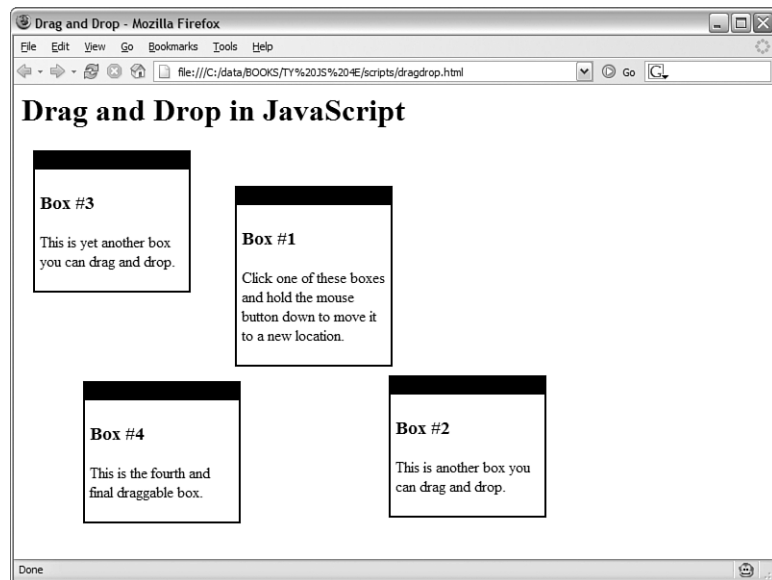
Here's a rundown of how this script works:

- ▶ The first line declares five global variables: `obj` to keep track of the current object being dragged, `x` and `y` for the mouse position, and `dx` and `dy` for the object's offset from the mouse position.
- ▶ The `Setup()` function runs when the page loads. This function uses `getElementsByTagName` to find all of the `<div>` elements in the page. For each one with the `class="drag"` attribute, it sets up an `onmousedown` event handler to call the `Drag()` function.
- ▶ The `Drag()` function sets `obj` to the correct object, sets the object's border color to red to indicate it's being dragged, and calculates the `dx` and `dy` offsets.

- ▶ The `Move()` function is where the action happens. After calculating the mouse pointer's x and y position, it sets the object's left and top properties to move it to follow the mouse.
- ▶ The `Drop()` function ends the process by setting the object's border color back to black, and then setting `obj` to false, so mouse movements won't move any object.
- ▶ The final lines set some global event handlers: `onmousemove` to call the `Move()` function, `onmouseup` to call the `Drop()` function, and `onload` to call `Setup()`.

Save this file as `dragdrop.js`. To try the example, make sure you have all three files in the same folder: the HTML document, the CSS file (`dragdrop.css`), and the JavaScript file (`dragdrop.js`). Load the HTML document into a browser. Figure 24.2 shows the example after all four objects have been dragged to new positions.

**FIGURE 24.2**  
The drag-and-drop example in action.



## Summary

In this hour, you've learned how the future of JavaScript and the Web might affect your web pages and scripts, and learned some of the upcoming technologies that might change the way you work with the Web. Finally, you learned how to create drag-and-drop effects using JavaScript.

Time's up—you've reached the end of this book. I hope you've enjoyed spending 24 hours learning JavaScript, and that you'll continue to learn more about it on your own. See Appendix A for starting points to further your knowledge.

## Q&A

- Q.** *Besides parsing documents into HTML, what other practical uses are there for XML?*
- A.** XML is a great way to store any type of marked-up text in a standardized way. Developers of many software applications, including popular word processors, are considering using XML-based files.
- Q.** *In the drag-and-drop example, the objects overlap each other. Is there a way to avoid this?*
- A.** Yes, if you set a `background-color` property for the objects in the style sheet, they won't overlap. However, you'll notice that sometimes you're dragging the current object behind the others. To avoid this, you can set the `style.zIndex` property for the current object in the script to keep it on top.
- Q.** *What if I have a JavaScript question that isn't answered in this book?*
- A.** Start with the resources in Appendix A. You should also stop by the author's website ([www.jsworkshop.com](http://www.jsworkshop.com)) for a list of updates to the book, frequently asked questions, and a forum where you can discuss JavaScript with the author and other users.

## Quiz Questions

Test your knowledge of JavaScript's future by answering the following questions.

1. Which of the following is the latest DOM recommendation from the W3C?
  - a. DOM 1.5
  - b. DOM level 1
  - c. DOM level 2
2. When should you use a new JavaScript feature?
  - a. Immediately
  - b. As soon as it's supported by browsers
  - c. As soon as it's part of a standard, and browsers that support it are widely available

3. Which of the following is an important way of making sure your scripts will work with future browsers?
  - a. Follow HTML, JavaScript, and DOM standards.
  - b. Spend an hour a day downloading the newest browsers and testing your scripts.
  - c. Wait until the very last browsers are released before writing any scripts.

## Quiz Answers

1. c. DOM level 2 is the latest W3C recommendation.
2. c. Wait until JavaScript features are standardized and widely available before implementing them.
3. a. Following HTML, JavaScript, and DOM standards is an important way of making sure your scripts will work with future browsers.

## Exercises

If you want to gain more experience working with JavaScript, try the following exercises:

- ▶ Try adding another `<div>` element to the drag-and-drop example and make the appropriate changes to the style sheet so it will respond correctly.
- ▶ Try changing the drag-and-drop example to move a different type of element, such as paragraphs of text.

## PART VII:

# Appendixes

<b>APPENDIX A</b>	Other JavaScript Resources	<b>409</b>
<b>APPENDIX B</b>	Tools for JavaScript Developers	<b>411</b>
<b>APPENDIX C</b>	Glossary	<b>415</b>
<b>APPENDIX D</b>	JavaScript Quick Reference	<b>419</b>
<b>APPENDIX E</b>	DOM Quick Reference	<b>427</b>

*This page intentionally left blank*



## APPENDIX A

# Other JavaScript Resources

Although you've learned a lot about JavaScript in 24 hours, there's still a lot to know. If you'd like to move on to advanced features of JavaScript or learn more, the resources listed in this appendix will be helpful.

## Other Books

The following books, also from Sams.net, discuss JavaScript and DHTML in more detail:

- ▶ *Sams Teach Yourself JavaScript in 21 Days* by Jinjer Simon, Andrew Watt, Jonathan Watt. ISBN 0672322978.
- ▶ *JavaScript Unleashed, Fourth Edition* by Jason D. Gilliam, R. Allen Wyke ISBN 0672324318.
- ▶ *JavaScript Developer's Dictionary* by Alexander J. Vincent. ISBN 0672322013.
- ▶ *Sams Teach Yourself DHTML in 24 Hours* by Michael Moncur. ISBN 0672323028.

## JavaScript Websites

The following websites will help you learn more about JavaScript:

- ▶ The JavaScript Workshop is a weblog about JavaScript written by Michael Moncur, the author of this book. There you'll find updates on the JavaScript language and the DOM, as well as detailed tutorials on beginning and advanced tasks.  
<http://www.jsworkshop.com/>
- ▶ The DOM Scripting Task Force, part of the Web Standards Project, works toward better use of standards in scripting, and has an informative weblog with the latest on JavaScript and DOM standards.  
<http://domscripting.webstandards.org/>
- ▶ The Mozilla Project's JavaScript section has information on the latest updates to the JavaScript language, as well as documentation, links to resources, and information about JavaScript implementations.  
<http://www.mozilla.org/js/>

## Web Development Sites

The following sites have news and information about web technologies, including JavaScript, XML, and the DOM, as well as basic HTML:

- ▶ The W3C (World Wide Web Consortium) is the definitive source for information about the HTML and CSS standards.

<http://www.w3.org/>

- ▶ WebReference.com has information and articles about web technologies ranging from Java to plug-ins.

<http://www.webreference.com/>

- ▶ *Digital Web Magazine* features regular online articles on everything from JavaScript and web design to running a web business.

<http://www.digital-web.com/>

## This Book's Website

Be sure to register your book at [www.sampublishing.com/register](http://www.sampublishing.com/register) by entering this book's ISBN number. You'll find updates to the book, information on new browsers and new JavaScript features, and other useful resources there, as well as a place to download all of this book's examples and the files you will need to try them out.

## APPENDIX B

# Tools for JavaScript Developers

One of the best things about JavaScript is that it requires no specialized tools—all you need to start scripting is a web browser and a simple text editor. Nonetheless, tools are available that will make scripting easier. Some of these are described in this appendix.

## HTML and Text Editors

Although they aren't specifically intended for scripting, a wide variety of HTML editors are available. These allow you to easily create web documents, either by automating the process of entering tags, or by presenting you with an environment for directly creating styled text.

### HomeSite

HomeSite, from Adobe, is a full-featured HTML editor. It is similar to a text editor, but includes features to automatically add HTML tags, and to easily create complicated HTML elements such as tables.

HomeSite also includes a number of JavaScript features, such as creating tags automatically and coloring script commands to make them easy to follow.

A demo version of HomeSite is available for download from Macromedia's site:

<http://www.macromedia.com/software/homesite/>

### TopStyle

TopStyle, from NewsGator Technologies, Inc., is a CSS and HTML editor written by Nick Bradbury, who originally created HomeSite. It specializes in CSS editing and includes powerful tools for editing style sheets, but it also works as an editor for HTML and JavaScript:

<http://www.newsgator.com/>

## FrontPage

Microsoft FrontPage is a popular WYSIWYG (What You See Is What You Get) HTML editor that allows you to easily create HTML documents. The latest version, FrontPage 2000, includes a component to create simple scripts automatically.

You can download FrontPage from Microsoft's site:

<http://www.microsoft.com/frontpage/>

## BBEdit

For Macintosh users, BBEdit is a great HTML editor that also includes JavaScript features. You can download it from Bare Bones Software's website:

<http://www.bbedit.com/>

## Text Editors

Often, a simple text editor is all you need to work on an HTML document or script. Here are some editors that are available for download:

- ▶ TextPad, from Helios Software Solutions, is a Windows text editor intended as a replacement for the basic Notepad accessory. It's a fast, useful editor, and also includes a number of features for working with HTML. TextPad is shareware, and a fully working version can be downloaded from its official site:

<http://www.textpad.com/>

- ▶ UltraEdit-32, from IDM Computer Solutions, is another good Windows text editor, with support for hexadecimal editing for binary files as well as simple text editing. The shareware version is available for download here:

<http://www.ultraedit.com/>

- ▶ SlickEdit, from MicroEdge, is a sophisticated programmer's editor for Windows and UNIX platforms:

<http://www.slickedit.com/>

- ▶ TextWrangler, from Bare Bones Software (the developers of BBEdit) is a text editor for the Macintosh that works great for general text files, HTML, and JavaScript:

<http://www.barebones.com/products/textwrangler/>

## HTML Validators

Writing web pages that comply with the HTML specifications is one way to avoid JavaScript errors, as well as to ensure that your pages will work with future browser versions. Here are some automated ways of checking the HTML compliance of your pages:

- ▶ CSE HTML Validator, from AI Internet Solutions, is an excellent standalone utility for Windows that checks HTML documents against your choice of HTML versions. It can also be integrated with HomeSite, TextPad, and several other HTML and text editors. Although the Pro version of this product is commercial, a Lite version is available for free download. Visit their website:

<http://www.htmlvalidator.com/>

- ▶ The W3C's HTML Validation Service is a web-based HTML validator. Just enter your URL, and it will be immediately checked for HTML compliance. Access this service at this URL:

<http://validator.w3.org/>

- ▶ The WDG HTML Validator offers a different perspective, and is also an easy-to-use web-based service. Access it at this URL:

<http://www.htmlhelp.com/tools/validator/>

## Debugging Tools

You might find the following tools useful in debugging your JavaScript applications:

- ▶ The Web Developer Extension for Firefox includes several helpful features for debugging JavaScript and for analyzing pages. (See Hour 16, "Debugging JavaScript Applications," for more information.)

<http://chrispederick.com/work/webdeveloper/>

- ▶ The Mozilla project's Venkman is a sophisticated debugger for JavaScript in Mozilla or Firefox. Find out more here:

<http://www.mozilla.org/projects/venkman/>

- ▶ Microsoft Script Debugger works with JavaScript and VBScript in Internet Explorer:

[http://msdn.microsoft.com/library/en-us/sdbug/html/sdbug\\_1.asp](http://msdn.microsoft.com/library/en-us/sdbug/html/sdbug_1.asp)

*This page intentionally left blank*

# APPENDIX C

## Glossary

The following are some terms relating to JavaScript and web development that are used throughout this book. Although most of them are explained in the text of the book, this section can serve as a useful quick reference while reading the book, or while reading other sources of JavaScript information.

**ActiveX** A technology developed by Microsoft to allow components to be created, primarily for Windows computers. ActiveX components, or controls, can be embedded in web pages.

**AJAX (Asynchronous JavaScript and XML)** a combination of technologies that allows JavaScript to send requests to a server, receive responses, and update sections of a page without loading a new page.

**anchor** In HTML, a named location within a document, specified using the `<a>` tag. Anchors can also act as links.

**applet** A Java program that is designed to be embedded in a web page.

**argument** A parameter that is passed to a function when it is called. Arguments are specified within parentheses in the function call.

**array** A set of variables that can be referred to with the same name and a number, called an index.

**attribute** A property value that can be defined within an HTML tag. Attributes specify style, alignment, and other aspects of the element defined by the tag.

**Boolean** A type of variable that can store only two values: true and false.

**browser sensing** A scripting technique that detects the specific browser in use by clients to provide compatibility for multiple browsers.

**Cascading Style Sheets (CSS)** The W3C's standard for applying styles to HTML documents. CSS can control fonts, colors, margins, borders, and positioning.

**concatenate** The act of combining two strings into a single, longer string.

**conditional** A JavaScript statement that performs an action if a particular condition is true, typically using the `if` statement.

**debug** The act of finding errors, or bugs, in a program or script.

**decrement** To decrease the value of a variable by one. In JavaScript, this can be done with the decrement operator, `--`.

**deprecated** A term the W3C applies to HTML tags or other items that are no longer recommended for use, and may not be supported in the future. For example, the `<font>` tag is deprecated in HTML 4.0 because style sheets can provide the same capability.

**Document Object Model (DOM)** The set of objects that JavaScript can use to refer to the browser window and portions of the HTML document. The W3C (World Wide Web Consortium) DOM is a standardized version supported by the latest browsers, and allows access to every object within a web page.

**Dynamic HTML (DHTML)** The combination of HTML, JavaScript, CSS, and the DOM, which allows dynamic web pages to be created. DHTML is not a W3C standard or a version of HTML.

**element** A single member of an array, referred to with an index. In the DOM, an element is a single node defined by an HTML tag.

**event** A condition, often the result of a user's action, that can be detected by a script.

**event handler** A JavaScript statement or function that will be executed when an event occurs.

**expression** A combination of variables, constants, and operators that can be evaluated to a single value.

**feature sensing** A scripting technique that detects whether a feature, such as a DOM method, is supported before using it to avoid browser incompatibilities.

**function** A group of JavaScript statements that can be referred to using a function name and arguments.

**global variable** A variable that is available to all JavaScript code in a web page. It is declared (first used) outside any function.

**Greasemonkey** An extension for the Firefox browser that allows user scripts to modify the appearance and behavior of web pages.

**Hypertext Markup Language (HTML)** The language used in web documents. JavaScript statements are not HTML, but can be included within an HTML document.

**increment** To increase the value of a variable by one. In JavaScript, this is done with the increment operator, `++`.



**interpreter** The browser component that interprets JavaScript statements and acts on them.

**Java** An object-oriented language developed by Sun Microsystems. Java applets can be embedded within a web page. JavaScript has similar syntax, but is not the same as Java.

**JavaScript** A scripting language for web documents, loosely based on Java's syntax, developed by Netscape. JavaScript is now supported by the most popular browsers.

**layer** An area of a web page that can be positioned and can overlap other sections in defined ways. Layers are also known as positionable elements.

**local variable** A variable that is available to only one function. It is declared (first used) within the function.

**loop** A set of JavaScript statements that are executed a number of times, or until a certain condition is met.

**method** A specialized type of function that can be stored in an object, and acts on the object's properties.

**Navigator** A browser developed by Netscape, and the first to support JavaScript.

**node** In the DOM, an individual container or element within a web document. Each HTML tag defines a node.

**object** A type of variable that can store multiple values, called properties, and functions, called methods.

**operator** A character used to divide variables or constants used in an expression.

**parameter** A variable sent to a function when it is called, also known as an argument.

**progressive enhancement** The approach of building a basic page that works on all browsers, and then adding features such as scripting that will work on newer browsers without compromising the basic functionality of the page.

**property** A variable that is stored as part of an object. Each object can have any number of properties.

**rule** In CSS, an individual element of a style block that specifies the style for an HTML tag, class, or identifier.

**scope** The part of a JavaScript program that a variable was declared in and is available to.

**selector** In a CSS rule, the first portion of the rule that specifies the HTML tag, class, or identifier that the rule will affect.

**statement** A single line of a script or program.

**string** A group of text characters that can be stored in a variable.

**tag** In HTML, an individual element within a web document. HTML tags are contained within angle brackets, as in <body> and <p>.

**text node** In the DOM, a node that stores a text value rather than an HTML element. Nodes that contain text, such as paragraphs, have a text node as a child node.

**unobtrusive scripting** A set of techniques that make JavaScript accessible and avoid trouble with browsers by separating content, presentation, and behavior.

**variable** A container, referred to with a name, that can store a number, a string, or an object.

**VBScript** A scripting language developed by Microsoft, with syntax based on Visual Basic. VBScript is supported only by Microsoft Internet Explorer.

**World Wide Web Consortium (W3C)** An international organization that develops and maintains the standards for HTML, CSS, and other key web technologies.

**XHTML (Extensible Hypertext Markup Language)** A new version of HTML developed by the W3C. XHTML is similar to HTML, but conforms to the XML specification.

**XML (Extensible Markup Language)** A generic language developed by the W3C (World Wide Web Consortium) that allows the creation of standardized HTML-like languages, using a DTD (Document Type Definition) to specify tags and attributes.

## APPENDIX D

# JavaScript Quick Reference

This appendix is a quick reference for the JavaScript language. It includes the built-in objects and the objects in the basic object hierarchy, JavaScript statements, and built-in functions.

## Built-in Objects

The following objects are built in to JavaScript. Some can be used to create objects of your own; others can only be used as they are. Each is detailed in the following sections.

### Array

You can create a new array object to define an array—a numbered list of variables. (Unlike other variables, arrays must be declared.) Use the `new` keyword to define an array, as in this example:

```
students = new Array(30)
```

Items in the array are indexed beginning with 0. Refer to items in the array with brackets:

```
fifth = students[4];
```

Arrays have a single property, `length`, which gives the current number of elements in the array. They have the following methods:

- ▶ `join` quickly joins all of the array's elements together, resulting in a string. The elements are separated by commas, or by the separator you specify.
- ▶ `reverse` returns a reversed version of the array.
- ▶ `sort` returns a sorted version of the array. Normally this is an alphabetical sort; however, you can use a custom sort method by specifying a comparison routine.

### String

Any string of characters in JavaScript is a string object. The following statement assigns a variable to a string value:

```
text = "This is a test."
```

Because strings are objects, you can also create a new string with the `new` keyword:

```
text = new String("This is a test.");
```

String objects have a single property, `length`, which reflects the current length of the string. There are a variety of methods available to work with strings:

- ▶ `substring` returns a portion of the string.
- ▶ `toUpperCase` converts all characters in the string to uppercase.
- ▶ `toLowerCase` converts all characters in the string to lowercase.
- ▶ `indexOf` finds an occurrence of a string within the string.
- ▶ `lastIndexOf` finds an occurrence of a string within the string, starting at the end of the string.
- ▶ `link` creates an HTML link using the string's text.
- ▶ `anchor` creates an HTML anchor within the current page.

There are also a few methods that allow you to change a string's appearance when it appears in an HTML document:

- ▶ `string.big` displays big text using the `<big>` tag in HTML 3.0.
- ▶ `string.blink` displays blinking text using the `<blink>` tag in Netscape.
- ▶ `string.bold` displays bold text using the `<b>` tag.
- ▶ `string.fixed` displays fixed-font text using the `<tt>` tag.
- ▶ `string.fontcolor` displays the string in a colored font, equivalent to the `<fontcolor>` tag in Netscape.
- ▶ `string.fontsize` changes the font size using the `<fontsize>` tag in Netscape.
- ▶ `string.italics` displays the string in italics using the `<i>` tag.
- ▶ `string.small` displays the string in small letters using the `<small>` tag in HTML 3.0.
- ▶ `string.strike` displays the string in a strike-through font using the `<strike>` tag.
- ▶ `string.sub` displays subscript text, equivalent to the `<sub>` tag in HTML 3.0.
- ▶ `string.sup` displays superscript text, equivalent to the `<sup>` tag in HTML 3.0.

## Math

The `Math` object is not a “real” object because you can't use it to create your own objects. A variety of mathematical constants are also available as properties of the `Math` object:

- ▶ `Math.E` is the base of natural logarithms (approximately 2.718).
- ▶ `Math.LN2` is the natural logarithm of two (approximately 0.693).
- ▶ `Math.LN10` is the natural logarithm of 10 (approximately 2.302).
- ▶ `Math.LOG2E` is the base 2 logarithm of  $e$  (approximately 1.442).
- ▶ `Math.LOG10E` is the base 10 logarithm of  $e$  (approximately 0.434).
- ▶ `Math.PI` is the ratio of a circle's circumference to its diameter (approximately 3.14159).
- ▶ `Math.SQRT1_2` is the square root of one half (approximately 0.707).
- ▶ `Math.SQRT2` is the square root of two (approximately 1.4142).

The methods of the `Math` object allow you to perform mathematical functions. The methods are listed in the following categories.

### Algebraic Functions

- ▶ `Math.acos` calculates the arc cosine of a number in radians.
- ▶ `Math.asin` calculates the arc sine of a number.
- ▶ `Math.atan` calculates the arc tangent of a number.
- ▶ `Math.cos` calculates the cosine of a number.
- ▶ `Math.sin` returns the sine of a number.
- ▶ `Math.tan` calculates the tangent of a number.

### Statistical and Logarithmic Functions

- ▶ `Math.exp` returns  $e$  (the base of natural logarithms) raised to a power.
- ▶ `Math.log` returns the natural logarithm of a number.
- ▶ `Math.max` accepts two numbers and returns whichever is greater.
- ▶ `Math.min` accepts two numbers and returns the smaller of the two.

### Basic Math and Rounding

- ▶ `Math.abs` calculates the absolute value of a number.
- ▶ `Math.ceil` rounds a number up to the nearest integer.
- ▶ `Math.floor` rounds a number down to the nearest integer.
- ▶ `Math.pow` calculates one number to the power of another.

- ▶ `Math.round` rounds a number to the nearest integer.
- ▶ `Math.sqrt` calculates the square root of a number.

## Random Numbers

- ▶ `Math.random` returns a random number between 0 and 1.

## Date

The `Date` object is a built-in JavaScript object that allows you to conveniently work with dates and times. You can create a `Date` object any time you need to store a date, and use the object's methods to work with the date:

- ▶ `setDate` sets the day of the month.
- ▶ `setMonth` sets the month. JavaScript numbers the months from 0 to 11, starting with January (0).
- ▶ `setYear` sets the year. `SetFullYear` is a four-digit, Y2K-compliant version.
- ▶ `setTime` sets the time (and the date) by specifying the number of milliseconds since January 1, 1970.
- ▶ `setHours`, `setMinutes`, and `setSeconds` set the time.
- ▶ `getDate` gets the day of the month.
- ▶ `getMonth` gets the month.
- ▶ `getFullYear` gets the year.
- ▶ `getTime` gets the time (and the date) as the number of milliseconds since January 1, 1970.
- ▶ `getHours`, `getMinutes`, and `getSeconds` get the time.
- ▶ `getTimeZoneOffset` gives you the local time zone's offset from GMT.
- ▶ `toGMTString` converts the date object's time value to text using GMT (Greenwich Mean Time, also known as UTC).
- ▶ `toLocaleString` converts the `Date` object's time value to text using local time.
- ▶ `Date.parse` converts a date string, such as "June 20, 2003" to a `Date` object (number of milliseconds since 1/1/1970).
- ▶ `Date.UTC` converts a `Date` object value (number of milliseconds) to a UTC (GMT) time.

# Creating and Customizing Objects

This is a brief summary of the keywords you can use to create your own objects and customize existing objects. These are documented in detail in Hour 6, “Using Functions and Objects.”

## Creating Objects

There are three JavaScript keywords used to create and refer to objects:

- ▶ `new` is used to create a new object.
- ▶ `this` is used to refer to the current object. This can be used in an object’s constructor function or in an event handler.
- ▶ `with` makes an object the default for a group of statements. Properties without complete object references will refer to this object.

To create a new object, you need an object constructor function. This simply assigns values to the object’s properties using this:

```
function Name(first,last) {
 this.first = first;
 this.last = last;
}
```

You can then create a new object using `new`:

```
Fred = new Name("Fred","Smith");
```

## Customizing Objects

You can add additional properties to an object you have created just by assigning them:

```
Fred.middle = "Clarence";
```

Properties you add this way apply only to that instance of the object, not to all objects of the type. A more permanent approach is to use the `prototype` keyword, which adds a property to an object’s prototype (definition). This means that any future object of the type will include this property. You can include a default value:

```
Name.prototype.title = "Citizen";
```

You can use this technique to add properties to the definitions of built-in objects as well. For example, this statement adds a property called `num` to all existing and future string objects, with a default value of 10:

```
string.prototype.num = 10;
```

## JavaScript Statements

This is an alphabetical listing of the statements available in JavaScript and their syntax.

### Comments

Comments are used to include a note within a JavaScript program, and are ignored by the interpreter. There are two different types of comment syntax:

```
//this is a comment
/* this is also a comment */
```

Only the second syntax can be used for multiple-line comments; the first must be repeated on each line.

### break

This statement is used to break out of the current `for` or `while` loop. Control resumes after the loop, as if it had finished.

### continue

This statement continues a `for` or `while` loop without executing the rest of the loop. Control resumes at the next iteration of the loop.

### for

This statement defines a loop, usually to count from one number to another using an index variable. In this example, the variable `i` counts from 1 to 9:

```
for (i=1;i<10;i++;) { statements }
```

### for...in

This is a different type of loop, used to iterate through the properties of an object, or the elements of an array. This statement loops through the properties of the `Scores` object, using the variable `x` to hold each property in turn:

```
for (x in Scores) { statements }
```

### function

This statement defines a JavaScript function that can be used anywhere within the current document. Functions can optionally return a value with the `return` statement. This example defines a function to add two numbers and return the result:



```
function add(n1,n2) {
 result = n1 + n2;
 return result;
}
```

## if...else

This is a conditional statement. If the condition is true, the statements after the `if` statement are executed; otherwise, the statements after the `else` statement (if present) are executed. This example prints a message stating whether a number is less than or greater than 10:

```
if (a > 10) {
 document.write("Greater than 10");
}
else {
 document.write("10 or less");
}
```

A shorthand method, known as the “hook and colon” conditional, can also be used for these types of statements, where `?` indicates the `if` portion and `:` indicates the `else` portion. This statement is equivalent to the previous example:

```
document.write((a > 10) ? "Greater than 10" : "10 or less");
```

Conditional statements are explained further in Hour 7, “Controlling Flow with Conditions and Loops.”

## return

This statement ends a function, and optionally returns a value. The `return` statement is necessary only if a value is returned.

## var

This statement is used to declare a variable. If you use it within a function, the variable is guaranteed to be local to that function. If you use it outside the function, the variable is considered global. Here’s an example:

```
var students = 30;
```

Because JavaScript is a loosely typed language, you do not need to specify the type when you declare the variable. A variable is also automatically declared the first time you assign it a value:

```
students = 30;
```

Using `var` will help avoid conflicts between local and global variables. Note that arrays are not considered ordinary JavaScript variables; they are objects. See Hour 5, “Using Variables, Strings, and Arrays,” for details.

## `while`

The `while` statement defines a loop that iterates as long as a condition remains true. This example waits until the value of a text field is “go”:

```
while (document.form1.text1.value != "go") {statements }
```

# JavaScript Built-in Functions

The functions in the following sections are built in to JavaScript, rather than being methods of a particular object.

## `eval`

This function evaluates a string as a JavaScript statement or expression, and either executes it or returns the resulting value. In the following example, a function is called using variables as an argument:

```
a = eval("add(x,y);");
```

`eval` is typically used to evaluate an expression or a statement entered by the user.

## `parseInt`

This function finds an integer value at the beginning of a string and returns it. If there is no number at the beginning of the string, “NaN” (not a number) is returned.

## `parseFloat`

Finds a floating-point value at the beginning of a string and returns it. If there is no number at the beginning of the string, “NaN” (not a number) is returned.

## APPENDIX E

# DOM Quick Reference

This appendix presents a quick overview of the DOM objects available, including the basic level 0 DOM and the W3C level 1 DOM.

## DOM Level 0

The level 0 DOM includes objects that represent the browser window, the current document, and its contents. The following is a basic summary of level 0 DOM objects.

The level 0 DOM was an informal standard developed by Netscape when JavaScript was introduced. Its objects and properties are now formalized in the W3C DOM level 1 recommendation.

***By the  
Way***

### window

The window object represents the current browser window. If multiple windows are open or frames are used, there might be more than one window object. These are given aliases to distinguish them:

- ▶ `self` represents the current window, as does `window`. This is the window containing the current JavaScript document.
- ▶ `top` is the window currently on top (active) on the screen.
- ▶ `parent` is the window that contains the current frame.
- ▶ The `frames` array contains the window object for each frame in a framed document.

The window object has three child objects:

- ▶ `location` stores the location (URL) of the document displayed in the window.
- ▶ `document` stores information about the current web page.
- ▶ `history` contains a list of sites visited before or after the current site in the window.

## location

The `location` object contains information about the current URL being displayed by the window. It has a set of properties to hold the different components of the URL:

- ▶ `location.hash` is the name of an anchor within the document, if specified.
- ▶ `location.host` is a combination of the host name and port.
- ▶ `location.hostname` specifies the host name.
- ▶ `location.href` is the entire URL.
- ▶ `location.pathname` is the directory to find the document on the host, and the name of the file.
- ▶ `location.port` specifies the communication port.
- ▶ `location.protocol` is the protocol (or method) of the URL.
- ▶ `location.query` specifies a query string.
- ▶ `location.target` specifies the `TARGET` attribute of the link that was used to reach the current location.

## history

The `history` object holds information about the URLs that have been visited before and after the current one in the window, and includes methods to go to previous or next locations:

- ▶ `history.back` goes back to the previous location.
- ▶ `history.forward` goes forward to the next location.
- ▶ `history.go` goes to a specified offset in the history list.

## document

The `document` object represents the current document in the window. It includes the following child objects:

- ▶ `document.forms` is a collection with an element for each form in the document.
- ▶ `document.links` is a collection containing elements for each of the links in the document.
- ▶ `document.anchors` is a collection with elements for each of the anchors in the document.

- ▶ `document.images` contains an element for each of the images in the current document.
- ▶ `document.applets` is a collection with references to each embedded Java applet in the document.

## navigator

The `navigator` object includes information about the current browser version:

- ▶ `appCodeName` is the browser's code name, usually "Mozilla."
- ▶ `appName` is the browser's full name.
- ▶ `appVersion` is the version number of the browser. (Example: "4.0(Win95;I).")
- ▶ `userAgent` is the user-agent header, which is sent to the host when requesting a web page. It includes the entire version information, such as "Mozilla/4.5(Win95;I)."
- ▶ `plugins` is a collection, which contains information about each currently available plug-in (Netscape and Firefox only).
- ▶ `mimeType` is a collection containing an element for each of the available MIME types (Netscape and Firefox only).

# DOM Level 1

The level 1 DOM is the first cross-browser DOM standardized by the W3C. Its objects are stored under the `document` object of the level 0 DOM.

## Basic Node Properties

Each object has certain common properties:

- ▶ `nodeName` is the name of the node (not the ID). The name is the tag name for HTML tag nodes, `#document` for the document node, and `#text` for text nodes.
- ▶ `nodeType` is a number describing the node's type: 1 for HTML tags, 3 for text nodes, and 9 for the document.
- ▶ `nodeValue` is the text contained within a text node.
- ▶ `innerHTML` is the HTML contents of a container node.
- ▶ `id` is the value of the ID attribute for the node.
- ▶ `className` is the value of the class attribute for the node.

## Relationship Properties

The following properties describe an object's relationship with others in the hierarchy:

- ▶ `firstChild` is the first child node for the current node.
- ▶ `lastChild` is the last child object for the current node.
- ▶ `childNodes` is an array of all the child nodes under a node.
- ▶ `previousSibling` is the sibling before the current node.
- ▶ `nextSibling` is the sibling after the current node.
- ▶ `parentNode` is the object that contains the current node.

## Offset Properties

Although not part of the W3C DOM, both Netscape and Internet Explorer support the following properties that provide information about a node's position:

- ▶ `offsetLeft` is the distance from the left side of the browser window or containing object to the left edge of the node object.
- ▶ `offsetTop` is the distance from the top of the browser window or containing object to the top of the node object.
- ▶ `offsetHeight` is the height of the node object.
- ▶ `offsetWidth` is the width of the node object.

## Style Properties

The style child object under each DOM object includes its style sheet properties. These are based on attributes of a style attribute, `<style>` tag, or external style sheet. See Hour 12, "Working with Style Sheets," for details on these properties.

## Node Methods

The following methods are available for all DOM nodes:

- ▶ `appendChild(node)` adds a new child node to the node after all its existing children.
- ▶ `insertBefore(node,oldnode)` inserts a new node before the specified existing child node.

- ▶ `replaceChild(node, oldnode)` replaces the specified old child node with a new node.
- ▶ `removeChild(node)` removes an existing child node.
- ▶ `hasChildNodes()` returns a Boolean value of `true` if the node has one or more children, or `false` if it has none.
- ▶ `cloneNode()` returns a copy of the current node.
- ▶ `getAttribute(attribute_name)` gets the value of the attribute you specify and stores it in a variable.
- ▶ `setAttribute(attribute_name, value)` sets the value of an attribute.
- ▶ `removeAttribute(attribute_name)` removes the attribute you specify.
- ▶ `hasAttributes()` simply returns `true` if the node has attributes, and `false` if it has none.

## Document Object Methods and Properties

The following are methods and properties of the document object:

- ▶ `document.getElementById(ID)` returns the element with the specified ID attribute.
- ▶ `document.getElementsByTagName(tag)` returns an array of the elements with the specified tag name. You can use the asterisk (\*) as a wildcard to return an array containing all of the nodes in the document.
- ▶ `document.createElement(tag)` creates a new element with the specified tag name.
- ▶ `document.createTextNode(text)` creates a new text node containing the specified text.
- ▶ `document.documentElement` is an object that represents the document itself, and can be used to find information about the document.

*This page intentionally left blank*



# Index

## Symbols

**&&** (and operator), 103  
**=** (assignment operator), 103  
**{}** (braces), for loop syntax, 110  
**==** (equality operator), 103, 256  
**++** (increment operator), 66, 115  
**!** (not operator), 104  
**+=** operator, 66  
**—** operator, 66  
**||** (or operator), 103  
**+** (plus sign), 28  
**;** (semicolon), 27, 38  
**.au** files, 333  
**.css** external files, creating, 201  
**.is** extension, 11  
**.mid** files, 333  
**.mp3** files, 333  
**.wav** files, 333

## A

abbreviating statements with shorthand expressions, 105  
accessibility, 241  
accessing  
    array elements, 77  
    JavaScript console (Firefox), 258  
ActionScript, 332  
ActiveX, 18  
addEventListener() function, 142  
addEventListener() method, W3C event model, 238-239  
adding  
    comments to scripts, 43-44, 240, 259  
    event handlers to HTML tags, 237-238  
        cross-browser method, 239  
    items to navigation trees, 228  
    Script.aculo.us library to HTML document, 132  
    scripts to HTML documents, 28  
    temporary statements to scripts, 259  
    text to pages, 225-228  
AddItem() function, 387  
AddScore() functions, 367  
AddText() function, 226

Adobe Dreamweaver, 25  
Adobe GoLive, 25  
AJAX (Asynchronous JavaScript and XML), 17, 273  
    applications, debugging, 285  
    back end, 274  
    client operation, requests, 274  
    client/server processing, XMLHttpRequest object, 277-278  
    examples of, 275-276  
    frameworks, 276  
    libraries, 130, 276  
        ajaxRequest function, 279  
        ajaxResponse function, 280  
        creating, 279-280  
    limitations of, 276-277  
    live searches, 285  
        back end, 287-288  
        front end, 288-289  
        HTML file example, 286-287  
        requirements for, 289  
    quiz  
        creating, 280, 282-284  
        testing, 284  
ajaxRequest() function, 279  
ajaxResponse() function, 280  
alert messages, 259  
alert() function, 45  
anchor objects, 54  
And operator (&&), 103  
animated slideshows, creating, 322-326  
AnimateSlide() function, 326  
APIs (application programming interfaces), Greasemonkey support for, 302  
appendChild() method, 221  
applets, 17  
applications for JavaScript  
    improving navigation, 16  
    remote scripting, 17  
    special effects, 17  
    validating forms, 16  
applying  
    classes to elements, 195  
    styles to specific elements, 194  
arguments, 86

arrays, 76. *See also* string arrays  
    assigning values to, 76  
    creating, 76  
    declaring, 76  
    elements, accessing, 77  
    frame, 167-168  
    length property, 77  
    sorting, 79-81  
ASCII (American Standard Code for Information Interchange), 149  
assigning values  
    to arrays, 76  
    to strings, 71-72  
    to variables, 65-66  
assignment and equality, 256  
assignment operator (=), 256  
attributes of form tag, 173  
audio file formats, 333  
avoiding  
    browser-specific scripting, 239  
    common scripting mistakes, 256  
        HTML errors, 257  
    assignment and equality, 256  
    confusing local and global variables, 257  
    improper object usage, 257  
    syntax errors, 256  
    errors, 249-252

## B

back button, creating, 56-57  
back end, 274  
    of AJAX live search example, 287-288  
background images, CSS properties, 196-197  
background property (CSS), 196  
background-attachment property (CSS), 196  
background-color property (CSS), 196, 213  
background-image property (CSS), 196, 213  
background-position property (CSS), 196

**background-repeat property (CSS)**

**background-repeat property (CSS)**, 196

**behavior, separating from content and presentation**, 236

**best practices**, 44

for unobtrusive scripting, 235

**bookmarklets**, 265

JavaScript Shell, 265

**Boolean operators**, *See* logical operators

**Boolean values**, 69

**Boodman, Aaron**, 294

**border-color property (CSS)**, 213

**border-style property (CSS)**, 213

**border-width property (CSS)**, 213

**borders, CSS properties**, 197-198

**break statement, escaping from infinite loops**, 113

**browser sensing**, 245-246

**browser-specific scripting, avoiding**, 239

**browsers**

dialog boxes, displaying, 164-165

graceful degradation, 237

information about, displaying, 242-243

for Internet Explorer 6.0, 243, 245

non-JavaScript, 247

detecting, 248

progressive enhancement, 237

script compatibility, 397

timeouts

enabling, 162

repeating, 163

user script support, 296

**windows**

closing, 159-160

creating, 158-159

resizing, 160

**built-in objects**, 39

definitions, extending, 94-96

Math object, 121-122

**buttons**, 178-179

**C**

**calculating**

length of arrays, 77

length of strings, 72

**calling functions**, 87-88

**case sensitivity**, 42

of event handlers, 140

**catch keyword, error handling**, 262

**CGI (Common Gateway Interface)**, 19

**Champeon, Steve**, 237

**changebody() function**, 204

**changehead() function**, 204

**charAT() method**, 75

**check boxes**, 179

**child objects**, 91, 209

**childNodes property**, 220

**classes**, 195

applying to elements, 195

**clear property (CSS)**, 197

**clearDesc function**, 153

**clearing timeouts**, 162

**clearTimeout() method**, 162

**client/server processing (AJAX), XMLHttpRequest object**, 277-278

**clip property (CSS)**, 212

**cloneNode() method**, 221

**closing windows**, 159-160

**CLR (Common Language Runtime)**, 16

**color property (CSS)**, 196

**colors**

CSS properties, 196-197

selecting from forms, 202

**combining**

conditions, 103-104

values of strings, 71

**comments**, 43-44

adding to code, 240, 259

**common scripting mistakes**

confusing local and global variables, 257

HTML errors, 257

improper object usage, 257

**communication between**

JavaScript and Flash, 332

**compatibility issues**

of browsers and scripts, 397

of drop-down menus, 346

**conditional expressions**, 102, 223

**conditional operators**, 103

**conditional statements**, 40

**conditions, combining**, 103-104

**constructor functions**, 92

**containers**, 208

**content**, 236

separating from behavior and presentation, 236

**continue statement**, 113

**controlling**

operator precedence, 68

sounds, 334

styles, 201-204

**controls**, 18

**converting**

ASCII code to string character, 149

case of strings, 73

data types, 69-70

date formats, 128

**createElement() method**, 221

**createTextNode() method**, 221

**creating**

AJAX libraries, 279-280

ajaxResponse function, 280

ajaxRequest function, 279

AJAX quiz, 280, 282-284

arrays, 76

back/forward buttons, 56-57

Date objects, 126

drag-and-drop objects, 399

CSS style sheet, 400-401

HTML document, 400

JavaScript file, 402-404

drop-down menus, 345

CSS, fine-tuning, 354, 356

example JavaScript file, 352-353

dynamic forms, 385-388

dynamic styles, 202-204

error handlers, 260

event handlers, 140

external .css files, 201

global variables, 65

layers, 210-211

local variables, 65

loops

for, in loops, 115-116

with do statement, 112

with for statement, 109

with while statement, 110-111

movable layers, 214-215

## DOM (Document Object Model)

- navigation trees, 226-228
- objects, 90
- poker solitaire game script, 363-368
- rollovers, 315-316
  - without JavaScript, 316-317
- rules, 193
- scripts, required tools
  - browsers, 25
  - text editors, 23-25
- scrolling windows, 377-380
- separate JavaScript and HTML files, 33-34
- site-specific user scripts, 302-304
- slideshows, 319-323, 325-326
- string arrays, 78
- string objects, 71
- stylesheets, 198-200
- user scripts, 299, 305-306
- windows, 158-159
- cross-browser scripting, feature sensing, 245-246**
- cross-platform compatibility, of drop-down menu support, 346**
- CSS (Cascading Style Sheets), 192**
  - adding styles to poker solitaire game, 368-372
  - drop-down menu links, formatting, 347-350
  - fine-tuning for drop-down menus, 354-356
  - for drag-and-drop objects, 400-401
  - for scrolling window, 378-379
  - graphic rollovers, 317-319
  - hover directive, creating rollovers, 316-317
  - movable layers, creating, 214-215
  - properties
    - background-color=, 213
    - background-image, 213
    - background images, 196-197
    - border-color, 213
    - border-style, 213
    - border-width, 213
    - borders, 197-198
    - clip, 212
    - colors, 196-197
    - display, 212
    - fonts, 197
    - hyphenating, 213
    - margins, 197-198
    - overflow, 212
    - text alignment, 195
    - units of measurement, 198
    - visibility, 212
  - rules, 194
- custom objects, 40**
- D**
- data types, 68-69**
  - converting between, 69-70
- date and time, displaying, 25-30**
  - with large clock display, 30-31
- date formats, converting, 128**
- Date objects, 27**
  - creating, 126
  - get methods, 127
  - reading values, 127
  - setting values, 126
- Date.parse() method, 128**
- Date.UTC() method, 128**
- debugging**
  - AJAX applications, 285
  - user scripts, 266-267, 304
- debugging tools, 259-260**
  - Firefox's JavaScript Console, 258
- decimal numbers, rounding, 121**
- declaring**
  - arrays, 76
  - variables, 64
- decrementing variables, 66**
- defining**
  - event handlers, 140-142
  - forms, 173-174
  - functions, 86-87
  - multiple classes for elements, 195
  - objects, 92
- describing user scripts with meta-data, 299-300**
- design patterns, 241**
- detecting**
  - browser features, 245-246
  - non-JavaScript browsers, 248
  - sound support, 335
- Developer Toolbar (Internet Explorer), 263-264**
- DHTML (dynamic HTML), 51, 207**
- dialog boxes, displaying, 164-165**
- display property (CSS), 212**
- displaying**
  - browser information, 242-243
    - for Internet Explorer 6.0, 243-245
  - date and time, 25-30
    - with large clock display, 30-31
  - dialog boxes, 164-165
  - error messages, 31, 259-261
  - form data, 182-183
  - generated source, 265
  - link descriptions, 151-154
  - submenus within drop-down menus, 351
  - typed characters, 150
- do...while loops, creating, 112**
- document node methods, 221**
- document object, 52-53**
  - methods, 53
  - properties, 52
- document.getElementById() method, 213**
- document.write statement, 10, 27**
- documenting scripts, 398**
- Dolt function, 161**
- Dojo library, 130**
- DOM (Document Object Model), 12, 49**
  - children, 209
  - history of, 50
  - layers, creating, 210
  - level standards, 51
  - methods, 50
  - nodes
    - document node, methods, 221
    - properties, 220
    - relationship properties, 220
  - objects, 39
    - document, 52-53
    - hierarchy, 50
    - positioning, 211
    - window, 51
  - parents, 209

## DOM (Document Object Model)

- properties, 50
- siblings, 210
- structure, 208
- DOM Inspector, 264**
- DoPlay() method, 334**
- downloading Script.aculo.us library, 131**
- Drag() function, 403**
- drag-and-drop objects**
  - creating, 399
  - CSS style sheet, 400-401
  - HTML document, 400
  - JavaScript file, 402-404
- Drop() function, 404**
- drop-down lists, 181-182**
- drop-down menus**
  - compatibility issues, 346
  - creating, 345, 352-353
  - CSS, fine-tuning, 354, 356
  - FindChild() function, 350
  - links, formatting in CSS, 347-350
  - SetupMenu() function, 350
  - submenu
    - displaying, 351
    - hiding, 351-352
    - positioning, 349
- DTD (Document Type Definition), 395**
- Dunck, Jeremy, 294**
- dynamic forms, creating, 385-388**
- dynamic HTML, 150**
- dynamic images, 313**
- dynamic styles, creating, 202-204**

## E

- ECMA (European Computer Manufacturing Association), 15**
- elements, 76**
  - buttons, 178-179
  - check boxes, 179
  - drop-down lists, 181-182
  - radio buttons, 180
  - referring to as arrays, 176
  - text areas, supported methods, 176-178
- else keyword, 104**
  - testing multiple conditions, 105-107
- em property (CSS), 198**

- emailing form results, 184-186**
- embedding Flash with JavaScript, 332**
- embedding sounds, 334**
- enabling timeouts on browsers, 162**
- EndGame() function, 368**
- equality operator (==), 256**
- error handlers**
  - adding to HTML tags, 237-239
  - catch keyword, 262
  - creating, 260
  - try keyword, 262
- error messages**
  - displaying, 259
  - handling, 31
- errors**
  - avoiding, 249-252
  - displaying information about, 261
  - fixing, 267
- escaping from infinite loops, 113**
- event handlers, 11, 41, 139.**
- See also timeouts**
  - case sensitivity, 140
  - creating, 140
  - defining, 140-142
  - example of, 45
  - for image object, 314
  - multiple, executing, 142
  - onClick, 146-148
  - onLoad, 151
  - onMouseDown, 146
  - onMouseMove, 145
  - onMouseOver, 145
  - onMouseUp, 146
  - parentheses, use of, 278
  - syntax, 140
  - W3C event model, 238-239
- event object, 142-143**
  - properties, 143-144
- events, 12**
  - onMouseOver, 140
- evolution of JavaScript, 14**
- ex property (CSS), 198**
- examples**
  - of AJAX, 275-276
  - of for loops, 110
- executing multiple event handlers, 142**

- expressions, 67**
  - operators, precedence rules, 67-68
- extending built-in object definitions, 94-96**
- external .css files, creating, 201**
- external scripts, 11**

## F

- feature sensing, 153, 239, 245-246, 331**
- findChild() function, configuring drop-down menus, 350**
- fine-tuning CSS for drop-down menus, 354-356**
- Firefox, 13**
  - DOM Inspector, 264
  - Greasemonkey extension. *See* Greasemonkey
  - JavaScript console, accessing, 258
  - Web Developer Extension, 263
- firstChild property, 220**
- fixing errors in scripts, 267**
- Flash, 399**
- Flash/JavaScript Integration Kit, 332**
- float property (CSS), 197**
- flow control, 101**
  - if statement, 102
    - conditional expressions, 102
    - logical operators, 103-104
- font property (CSS), 197**
- font-family property (CSS), 197**
- font-size property (CSS), 197**
- font-style property (CSS), 197**
- font-variant property (CSS), 197**
- font-weight property (CSS), 197**
- for loops, parameters, 109**
- for statement, 40, 109**
  - example of, 110
- for...in loops, 114**
  - creating, 115-116
- form object**
  - onReset event, 175
  - onSubmit event, 175
- formats of audio files, 333**
- formatting drop-down menu link in CSS, 347-350**

## hyphenated CSS property names

**forms**

- colors, selecting, 202
- data, displaying, 182-183
- defining, 173-174
- elements
  - buttons, 178-179
  - check boxes, 179
  - drop-down lists, 181-182
  - radio buttons, 180
  - text areas, 177-178
  - text fields, 176-178
- referring to as arrays, 176
- results, emailing, 184-186
- submitting, 175
- validating, 16, 185-186

**Forth programming language, 110****forward button, creating, 56-57****frame object, 166****frames array, 167-168****front end of AJAX live search****example, 288-289****Fuchs, Thomas, 129****functions, 11, 38, 85**

- addEventListener, 142
- AddItem(), 387
- AddScore(), 367
- AddText(), 226
- Alert(), 45
- AnimateSlide(), 326
- arguments, 86
- assigning as event handler, 141
- calling, 38, 87-88
- Changebody(), 204
- Changehead(), 204
- Cleardesc, 153
- constructor, 92
- defining, 86-87
- DisplayKey, 150
- Drag(), 403
- Drop(), 404
- EndGame(), 368
- GraphicBox, 252
- HideMenu(), 351
- HideSquare(), 216
- Hover(), 153
- local variables, creating, 65
- MakeSlideShow(), 320
- Move(), 404
- naming conventions, 43
- NextSlide(), 320

- parseFloat, 70
- PartInt(), 70
- Setup(), 340
- Show(), 387
- ShowHide(), 223
- ShowMenu(), 351
- showSquare(), 216
- Style(), 385
- Toggle(), 228, 252
- Update(), 163
- Validate(), 186
- values, returning, 88-89
- with multiple parameters,
  - defining, 87

**G****Garret, Jesse James, 274****generated source, viewing, 265****generating random numbers, 122-123****example script, 123-125****get methods for dates, 127****getElementById() method, 202, 216, 221****getElementsByTagName() method, 221****getTimeZoneOffset() function, 127****getUTCDate() function, 128****getUTCDay() function, 128****getUTCFullYear() function, 128****getUTCMonth() function, 128****global variables, 64****confusing with local variables, 257****Gmail, 249****GMT (Greenwich Mean Time), 26****Google Gmail, 249****graceful degradation, 237****graphic check box as unobtrusive scripting technique, 250-252****graphic rollovers, 317-319****graphicBox() function, 252****graphics, creating for poker solitaire game, 361****Greasemonkey**

- activating/deactivating, 298
- API functions, 302
- installing, 294
- metadata, 299-300

- security issues, 296
- user scripts
  - installing, 297
  - managing, 297-298
  - creating, 299, 305-306
  - debugging, 304
  - locating, 296
  - site-specific, creating, 302-304
  - testing, 300-301
  - text area macro user script, 306-307

**H****handling JavaScript errors, 31****hasChildNodes() method, 221****headings, modifying, 223****height property (CSS), 197****HideMenu() function, 351****hiding**

- objects, 222
- submenus within drop-down menus, 351-352

**HideSquare() function, 216****history.back() method, 55****history.forward() method, 55****history.go() method, 55****history.length property, 55****history object, 55****history**

- of DOM, 50
- of JavaScript, 8

**HomeSite, 24****hover function, 153****href property (window object), 55****HTML**

- application compliance, 398
- documents for drag-and-drop objects, 400
- editors, 24-25
- errors, avoiding, 257
- inline styles, 194
- layers, creating, 210-211
- tags
  - adding, 237-239
  - event handlers, 41
  - id attribute, 194
- hyphenated CSS property names, 213

## id attribute of HTML tags

### I

**id attribute of HTML tags, 194**

**IE (Internet Explorer), 13**

- browser information, displaying, 243-245
- Developer Toolbar, 263-264
- error messages, displaying, 259
- security settings, 29
- Turnabout, 295

**if statement, 102**

- conditional expressions, 102, 223
- logical operators, 103
  - And, 103
  - else keyword, 104
  - Not, 104
  - Or, 103
- testing multiple conditions, 105
  - time and greeting example, 106-107

**image object**

- event handlers, 314
- properties, 314

**images**

- preloading, 314
- rollovers, creating, 315-317
- slideshows, transitioning between, 322-326

**implementing drag-and-drop, 402**

**incompatibility with web browsers, 12**

**Increment operator (++), 115**

**incrementing variables, 66**

**IndexOf() method, 75-76**

**infinite loops, 112**

- escaping from, 113

**initial expression, 109**

**inline styles, 194**

**innerHTML property, 220**

**insertBefore() method, 221**

**installing**

- Greasemonkey, 294
- user scripts, 297

**instances, creating, 93**

**Internet Explorer. See IE**

**interpreted languages, 8**

### J-K

**Java, 399**

**JavaScript Shell, 265**

**JavaScript-free rollovers, creating, 316-317**

**join() method, 81**

**JScript, 15**

**JSON (JavaScript Object Notation), 275**

**keyboard events, 149**

- ASCII code, converting to string character, 149
- DisplayKey function, 150

**Koch, Peter-Paul, 247**

### L

**large clock display, adding to time and date script, 30-31**

**lastChild property, 220**

**lastIndexOf() method, 75**

**layers, 51, 207**

- creating, 210-211
- positioning, 213

**length of arrays, calculating, 77**

**length property, 72**

- of arrays, 77

**letter-spacing property (CSS), 195**

**levels (DOM), 51**

**libraries. See also third-party libraries**

- AJAX, creating, 279-280
- Script.aculo.us, library effects example, 132-133
- Yahoo! UI Library, 131

**limitations of AJAX, 276-277**

**line-height property (CSS), 196**

**link descriptions, displaying, 151-154**

**link objects, 54**

**linking to external stylesheets, 201**

**live searches**

- performing, 285-289
- requirements for, 289

**local variables, 64-65**

- confusing with global variables, 257

**localtime variable, 27**

**locating**

- strings within strings, 75-76
- user scripts, 296

**location object, 55-56**

**location.reload() method, 56**

**location.replace() method, 56**

**logical operators, 103**

- And, 103
- Not, 104
- Or, 103

**loops, 40**

- continue statement, 113
- creating
  - with for statement, 109-110
  - with while statement, 111
- for...in, 114
  - creating, 115-116
- infinite, 112
- escaping from, 113

### M

**Macintosh-based systems, HTML editors, 25**

**Macromedia Dreamweaver, 16**

**maintaining optional JavaScript code, 248-249**

**MakeSlideShow() function, 320**

**managing user scripts, 297-298**

**margin property (CSS), 197-198**

**Math object, 121-122**

**Math.random method, 123-125**

**messages**

- displaying in dialog boxes, 164-165
- scrolling, 377-380

**metadata, 299-300**

**methods, 39, 91**

- CharAT(), 75
- ClearTimeout(), 162
- document.getElementById(), 213
- DoPlay(), 334
- GetElementById(), 202
- history.back(), 55
- history.forward(), 55
- history.go(), 55
- indexOf(), 75-76
- join(), 81
- LastIndexOf(), 75
- location.reload, 56
- location.replace(), 56

- Math.random, 123-125
- Play(), 334
- Reset(), 175
- Rewind(), 335
- SetTimeout, 162
- SetTimeout(), 163
- Sort(), 79-81
- Split(), 78-79
- Stop(), 335
- Submit(), 175
- Substring(), 74
- window.open(), 159
- Microsoft Frontpage 2003, 24
- MIME (Multipurpose Internet Mail Extensions), 330-331
- MochiKit library, 131
- modifying text, 223
- mouse
  - events
    - onClick, 146-148
    - onMouseDown, 146
    - onMouseMove, 145
    - onMouseOver, 145
    - onMouseUp, 146
  - rollovers, creating, 315-317
- mousestatus function, 147
- movable layers, creating, 214-215
- Move() function, 404
- Mozilla Foundation, 13, 15
- multiple conditions, testing, 105-107
- multiple event handlers, executing, 142
- multiple scripts, order of operation, 42

## N

- naming conventions, 43
- NaN (non a number), 70
- navigation, improving, 16
- navigation tools, creating
  - back/forward buttons, 56-57
- navigation trees, adding, 226-228
- navigator object, 242
  - properties, 242-243
- nested framesets, 167
- Netscape 4.0, event object
  - properties, 144
- Netscape Communications Corporation, 8
- NextSibling property, 220
- NextSlide() function, 320
- NodeName property, 220
- nodes, 209
- NodeType property, 220
- NodeValue property, 220
- non-JavaScript browsers, 247
  - detecting, 248
- Not operator, 104
- null value, 69
- number-guesser script, debugging, 266-270
- numeric arrays, sorting, 79-81

## O

- object hierarchy (DOM), 50
- objects, 39, 90
  - built-in, 39
    - definitions, extending, 94-96
  - child objects, 91
  - creating, 90, 93
  - defining, 92
  - document, 52-53
    - methods, 53
    - properties, 52
  - DOM, 39, 49
  - event, 142-143
    - properties, 143-144
  - for Netscape 4.0, 144
  - forms, properties, 174-175
  - frame, 166
  - hiding/showing, 222
  - improper usage, avoiding, 257
  - instances, creating, 93
  - location, 55-56
  - methods, 39
  - naming conventions, 43
  - navigator, 242
    - properties, 242-243
  - positioning, 211
  - properties, 39, 91
  - resizing, 211
  - siblings, 210
  - visibility property, 222
  - window, 51
- onClick event handler, 146-148
- onLoad event handler, 151
- onMouseDown event handler, 146
- onMouseMove event handler, 145
- onMouseOver event handler, 140
- onMouseOver event handler, 145
- onMouseUp event handler, 146
- onReset event, 175
- onSubmit event, 175
- Opera, 14
- operators, 67
  - precedence rules, 67-68
- Or operator, 103
- overflow property (CSS), 212

## P

- pages, updating, 163
- parameters, 38
- parent objects, 209
- parentheses, event handler syntax, 278
- parseFloat() function, 70
- parseInt() function, 70
- Pederick, Chris, 263
- performing live searches, 285
  - front end, 288-289
  - back end, 287-288
  - HTML file example, 286-287
- PHP, 399
- piano keyboard script, 337-340
- Play() method, 334
- plug-ins, 329
  - feature sensing, 331
  - sound, detecting, 335
  - sound-playing, 334
- plus sign (+), 28
- poker solitaire game, 369
  - CSS style, adding, 368-372
  - graphics, creating, 361
  - HTML document, creating, 361-363
  - scoring, 360
  - script, creating, 363-368
- pop-up windows, displaying form data in, 182
- pos() function, 216
- positioning
  - objects, 211
  - submenus in drop-down menus, 349
- preloading images, 314



## presentation, separating from content and behavior

presentation, separating from  
 content and behavior, 236  
**PreviousSibling** property, 220  
 programming languages, Java, 17  
 progressive enhancement, 237  
 properties, 39, 91  
   of CSS  
     clip, 212  
     for background images,  
       196-197, 213  
     for borders, 197-198, 213  
     for colors, 196-197  
     for fonts, 197  
     for margins, 197-198  
     for text alignment, 195  
     for units of measurement,  
       198  
     hyphenating, 213  
     overflow, 212  
     visibility, 212  
   of document object, 52  
   of DOM nodes, 220  
   of event object, 143-144  
   of form object, 174-175  
   of image object, 314  
   of navigator object, 242-243  
   of window object, 158  
   of window.screen object, 158  
   values, reading, 91  
**prototype** keyword, 94  
**Prototype** third-party library, 129  
**pt** property (CSS), 198  
**px** property (CSS), 198  
 Python, 399

## Q

**QuirksMode**, 247  
 quiz questions  
   creating in AJAX, 280, 282-284  
   testing, 284  
 quotation marks, event handler  
   syntax, 140

## R

radio buttons, 180  
 random numbers, generating,  
   122-125  
 recommended web browsers, 25  
 relationship properties of DOM  
   nodes, 220

remote scripting, 17, 130, 274  
**RemoveChild()** method, 221  
 repeating timeouts, 163  
**ReplaceChild()** method, 221  
 requests, AJAX, 274  
 required JavaScript, avoiding,  
   248-249  
 requirements for AJAX live search  
   example, 289  
**reserved words**, 43  
**Reset()** method, 175  
**resetting forms**, 175  
**resizing**  
   objects, 211  
   windows, 160  
**return** keyword, 89  
**returning**  
   single character from  
     strings, 75  
   UTC time, 128  
**reusable AJAX libraries**  
   ajaxResponse function, 280  
   ajaxRequest function, 279  
   creating, 279-280  
**Rewind()** method, 335  
**rollovers**, 145  
   creating, 315-316  
     without JavaScript,  
       316-317  
   CSS graphic rollovers,  
     creating, 317-319  
**rounding decimal numbers**, 121  
 Ruby, 399  
 rules, creating, 193

## S

Safari, 14  
**scope of variables**, 64  
**scoring for poker solitaire**  
   game, 360  
**Script.aculo.us** library, 129-130  
   adding to HTML document,  
     132  
   downloading, 131  
   library effects example,  
     132-133  
 scripting languages, 8  
   VBScript, 18  
**scripts**, 8  
   adding to HTML  
     documents, 28  
   assignment and equality, 256  
   commenting out, 43-44, 259  
   common mistakes, avoiding,  
     256-257  
   compatibility with browsers,  
     397  
   creating, required tools,  
     23-25  
   debugging, 266-267  
   documenting, 398  
   errors, fixing, 267  
   event handlers, 41  
   for displaying date and time,  
     creating, 26-31  
   generating random numbers,  
     123-125  
   incompatibility with web  
     browsers, 12  
   order of operation, 42  
   separating from HTML files,  
     33-34  
   syntax errors, avoiding, 256  
   testing, 267-269  
**scrolling window**  
   creating, 377-378  
   CSS style sheet, 378-379  
   JavaScript file, 379-380  
**security issues with user**  
   scripts, 296  
**security settings** (IE 6.0), 29  
**selecting**  
   colors from forms, 202  
   names for variables, 64  
**self** keyword, 157  
**separating**  
   content, presentation, and  
     behavior, 236  
   JavaScript and HTML files,  
     33-34  
**server-side scripting**, 19  
**SetTimeout()** method, 162-163  
**Setup()** function, 340  
**SetupMenu()** function, configuring  
   drop-down menus, 350  
**SGML** (Standard Generalized  
   Markup Language), 395



## ToLocalizedString() function

- shorthand conditional expressions, 105
- Show() function, 387
- ShowHide() function, 223
- showing objects, 222
- ShowMenu() function, 351
- ShowSquare() function, 216
- siblings, 210
- site-specific user scripts, creating, 302-304
- sizing objects, 211
- slideshows, creating, 319-326
- sort() method, 79-81
- sorting
  - numeric arrays, 79-81
  - string arrays, 79
- sound file formats, 333
- sound-playing plug-ins, 334
- SoundManager, 335
- sounds
  - controlling, 334
  - embedding, 334
  - piano keyboard script, 337-340
  - testing, 336
- special effects, 17
- specifying versions of JavaScript, 15
- Split() method, 78-79
- splitting strings, 78-79
- statements, 38
  - conditional, 40
  - function calls, 38
- Stephenson, Sam, 129
- Stop() method, 335
- string arrays, 77
  - creating, 78
  - sorting, 79
- strings, 27, 69-70
  - assigning values to, 71-72
  - case, converting, 73
  - length of, calculating, 72
  - length property, 72
  - objects, creating, 71
  - returning single character from, 75
  - splitting, 78-79
  - substrings, locating, 75-76
- structure of DOM, 208

- style sheets
  - classes, 195
  - CSS. *See* CSS
  - switching between, 380-388
- Style() function, 385
- styles
  - applying to specific elements, 194
  - controlling, 201-204
- style sheets
  - controlling styles, 201-204
  - creating, 198-200
  - CSS
    - adding styles to poker solitaire game, 368-372
    - graphic rollovers, 317-319
  - dynamic styles, creating, 202-204
  - inline styles, 194
  - linking to external, 201
  - rules, 194
- submenus
  - displaying in drop-down menus, 351
  - hiding in drop-down menus, 351-352
  - positioning in drop-down menus, 349
- Submit() method, 175
- submitting forms, 175
- Substring() method, 74
- substrings, 74
  - index values, 74
  - locating, 75-76
- supported sound plug-ins, detecting, 335
- switch statement, 108
- switching between style sheets, 380-388
- syntax
  - case sensitivity, 42
  - comments, 43-44
  - errors, avoiding, 256
  - for event handlers, quotation marks, 140
  - for switch statement, 108
  - naming conventions, 43
  - reserved words, 43

## T

- tags
  - id attribute, 194
  - script, specifying versions of JavaScript, 15
- temporary statements, adding to scripts, 259
- testing
  - AJAX quiz, 284
  - date and time script, 29
  - multiple conditions, 105
  - time and greeting example, 106-107
  - scripts, 267-269
  - sounds, 336
  - user scripts, 298-301
- text
  - adding to pages, 225-228
  - alignment, CSS properties, 195
  - modifying, 223
  - scrolling, 377-380
- text area macro user script, 305-307
- text areas, supported methods, 177-178
- text fields, 176-178
- text-align property (CSS), 196
- text-decoration property (CSS), 195
- text-indent property (CSS), 196
- text-transform property (CSS), 196
- TextPad, 24
- third-party libraries
  - AJAX frameworks, 130
  - Prototype, 129
  - Script.aculo.us, 129-130
- time, displaying, 25-31
- time and greeting example, 106-107
- time zones, 127
- timeouts
  - clearing, 162
  - enabling on browsers, 162
  - repeating, 163
- Toggle() function, 228, 252
- ToLocalizedString() function, 127

**ToLowerCase() method**

**ToLowerCase() method**, 73  
**ToUpperCase() method**, 73  
**ToUTCString() function**, 127  
 transitioning between slideshow images, 322-326  
 tree structure of DOM, 209  
 try keyword, error handling, 262  
**Turnabout**, 295
 

- activating/deactivating, 298
- API functions, 302
- Options dialog, accessing, 298

 typed characters, displaying, 150

**U**

units of measurement, CSS properties, 198  
 unobtrusive scripting, 235, 320
 

- best practices, 235
- errors, avoiding, 249-252
- feature sensing, 245-246
- maintaining optional JavaScript code, 248-249

**Update() function**, 163  
 updating pages in browsers, 163  
 usability, 240
 

- accessibility, 241
- design patterns, 241

 user scripts, 293
 

- creating, 299, 305-306
- debugging, 304
- describing, 299-300
- installing, 297
- locating, 296
- managing, 297-298
- security, 296
- site-specific scripts, creating, 302-304
- testing, 298-301
- text area macro user script, 306-307

**UTC (Universal Time Coordinated)**, 26
 

- time values, returning, 128

 utctime variable, 27

**V**

**validate() function**, 186  
**validating forms**, 185-186  
**variables**, 26, 39
 

- arguments, 86
- assigning values to, 65-66
- declaring, 64
- decrementing, 66
- expressions, 67
- global, creating, 65
- incrementing, 66
- local, 65
- naming conventions, 43
- operators, 67
  - precedence rules, 67-68
- scope of, 64
- selecting names for, 64

**VBScript**, 18  
**verifying date and time script**, 29  
**versions of JavaScript**, 14-15  
**vertical-align property (CSS)**, 195  
**viewing**

- browser information, 242-243
  - for Internet Explorer 6.0, 243-245
- error information, 261
- form data, 182-183
- generated source, 265

**virtual machines**, 18  
**visibility property (CSS)**, 212, 222

**W**

**W3C (World Wide Web Consortium)**, 13, 51
 

- DOM. *See* DOM
- event model, 238-239

**web browsers**

- compatibility with Javascript, 12
- Firefox, 13
- IE, 13
- Mozilla, 13
- Opera, 14
- Safari, 14
- user script support, 296

**web design**

- design patterns, 241
- graceful degradation, 237
- progressive enhancement, 237

**Web Developer Extension**, 263  
**web pages**

- text, adding, 225-228
- text, modifying, 223

**while loops**, example of, 111  
**whitespace**, 43  
**width property (CSS)**, 197  
**window objects**, 51, 157-158  
**window.close() method**, 159-160  
**window.moveBy() method**, 161  
**window.moveTo() method**, 161  
**window.open() method**, 159  
**window.resizeBy() method**, 161  
**window.resizeTo() method**, 161  
**window.screen object**, properties, 158  
**window.setTimeout method**, 162  
**windows**

- closing, 159-160
- creating, 158-159
- resizing, 160

**Windows-based systems**, HTML editors, 24-25  
**with keyword**, 125  
**WSH (Windows Scripting Host)**, 16

**X-Z**

**XHTML (Extensible Hypertext Markup Language)**, 395-396  
**XML (Extensible Markup Language)**, 395
 

- AJAX. *See* AJAX

**XMLHttpRequest object**, 277-278  
**XSL (Extensible Stylesheet Language)**, 396  
  
**Yahoo Developer Network**, 241  
**Yahoo! UI Library**, 131, 239

Wouldn't it be great  
if the world's leading technical  
publishers joined forces to deliver  
their best tech books in a common  
digital reference platform?

They have. Introducing  
**InformIT Online Books**  
powered by Safari.

■ **Specific answers to specific questions.**

InformIT Online Books' powerful search engine gives you  
relevance-ranked results in a matter of seconds.

■ **Immediate results.**

With InformIT Online Books, you can select the book you  
want and view the chapter or section you need immediately.

■ **Cut, paste, and annotate.**

Paste code to save time and eliminate typographical errors.  
Make notes on the material you find useful and choose  
whether or not to share them with your workgroup.

■ **Customized for your enterprise.**

Customize a library for you, your department, or your entire  
organization. You pay only for what you need.

POWERED BY  
**safari**  
**informIT**  
**Online Books**

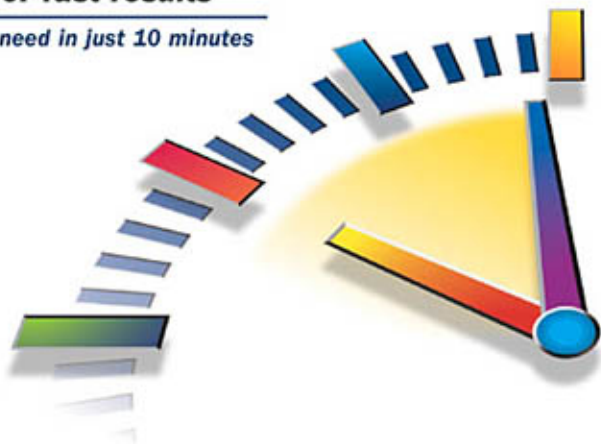
**Get your first 14 days FREE!**

InformIT Online Books is offering its members a 10-book subscription risk free  
for 14 days. Visit <http://www.informit.com/onlinebooks> for details.

**informit.com/onlinebooks**

**Quick steps for fast results™**

*Get the skills you need in just 10 minutes*



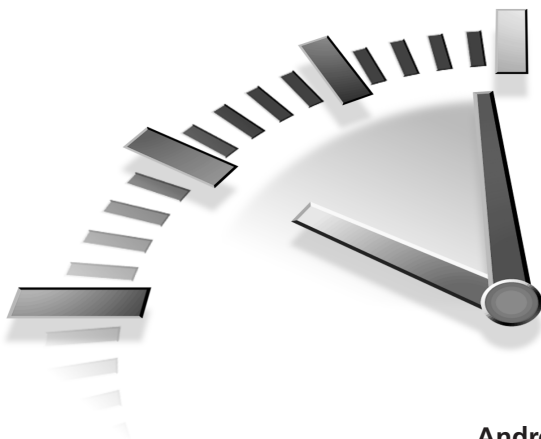
**SAMS**  
**Teach Yourself**

**XML**

**in 10**  
**Minutes**

Andrew H. Watt

**SAMS**  
**Teach Yourself**  
**XML**



Andrew H. Watt

in **10 Minutes**

**SAMS**

800 East 96th St., Indianapolis, Indiana, 46240 USA

# Sams Teach Yourself XML in 10 Minutes

## Copyright © 2003 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32471-7

Library of Congress Catalog Card Number: 2002110227

Printed in the United States of America

First Printing: October 2002

05 04 4 3

## Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**  
**1-800-382-3419**  
**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**  
**international@pearsoned.com**

**EXECUTIVE EDITOR**  
Michael Stephens

**ACQUISITIONS EDITOR**  
Todd Green

**DEVELOPMENT EDITOR**  
Kevin Howard

**MANAGING EDITOR**  
Charlotte Clapp

**PROJECT EDITOR**  
George E. Nedeff

**COPY EDITOR**  
Krista Hansing

**INDEXER**  
Sandra Henselmeier

**TECHNICAL EDITORS**  
Steve Heckler  
Mary C. Ecsedy

**TEAM COORDINATOR**  
Lynne Williams

**MULTIMEDIA DEVELOPER**  
Dan Scherf

**INTERIOR DESIGNER**  
Gary Adair

**COVER DESIGNER**  
Aren Howell

# Table of Contents

Introduction.....	1
-------------------	---

## **PART 1 XML Documents**

---

<b>1 What Is XML?</b>	<b>5</b>
What Is XML For? .....	5
XML Is a Markup Language .....	9
XML Is a Meta Language.....	10
How Does XML Relate to HTML? .....	12
Separating Content from Presentation .....	13
How Is XML Written? .....	14
Summary .....	16
<b>2 The Structure of an XML Document</b>	<b>17</b>
An XML Document .....	17
Prolog .....	18
Document Type Declaration .....	22
Document Element .....	24
CDATA Sections .....	25
Content After the Document Element End Tag .....	28
Summary .....	28
<b>3 XML Must Be Well-Formed</b>	<b>29</b>
Well-Formed XML Documents .....	29
XML Names .....	30
Elements .....	32
Attributes.....	33
Other Characteristics of Well-Formedness .....	35
Well-Formedness and XML Processor Type .....	39
Summary .....	40

<b>4</b>	<b>Valid XML—Document Type Definitions</b>	<b>41</b>
	Shared Documents: Why We Need DTDs .....	41
	What Is a Valid XML Document? .....	43
	What a DTD Is .....	43
	Declaring Elements in DTDs .....	46
	Declaring Attributes in DTDs .....	50
	Declaring Entities in the DTD .....	52
	Summary .....	52
<b>5</b>	<b>XML Entities</b>	<b>53</b>
	What Is an Entity? .....	53
	Parsed Entities .....	58
	Unparsed Entities .....	61
	Summary .....	64
<b>6</b>	<b>Characters in XML</b>	<b>65</b>
	Internationalization .....	65
	XML and Internationalization .....	69
	Unicode .....	72
	Fonts, Characters, and Glyphs .....	74
	Summary .....	76
<b>7</b>	<b>The Logic Hidden in XML</b>	<b>77</b>
	Modeling Data As XML .....	77
	W3C XML Data Models .....	85
	XPath .....	86
	The XML Information Set .....	87
	Summary .....	88
<b>8</b>	<b>Namespaces in XML</b>	<b>89</b>
	What Is a Namespace, and Why Do You Need Them? .....	89
	Using Namespaces in XML .....	93
	Using Multiple Namespaces in a Document .....	99
	Summary .....	101



---

## **PART 2    Manipulating XML**

---

<b>9</b>	<b>The XML Path Language—XPath</b>	<b>102</b>
	How XPath Is Used .....	102
	Accessing Elements .....	109
	Accessing Attributes .....	111
	XPath Functions .....	112
	Summary .....	114
<b>10</b>	<b>XSLT—Creating HTML from XML</b>	<b>115</b>
	XSLT Basics .....	115
	Creating a Simple HTML Page .....	118
	Creating an HTML List .....	122
	Creating an HTML Table .....	126
	Summary .....	128
<b>11</b>	<b>XSLT—Transforming XML Structure</b>	<b>129</b>
	Why Change Structure? .....	129
	Copying Elements.....	131
	Creating New Elements .....	135
	Creating New Attributes .....	140
	Summary .....	142
<b>12</b>	<b>XSLT—Sorting XML</b>	<b>143</b>
	Conditional Processing and Sorting Data.....	143
	Conditional Processing .....	144
	The <code>xs1:choose</code> Element .....	149
	Sorting Output.....	152
	Multiple Sorts .....	155
	Summary .....	158
<b>13</b>	<b>Styling XML with CSS</b>	<b>159</b>
	Cascading Style Sheets and XML .....	159
	Associating a Stylesheet .....	161
	Using CSS Rules with XML .....	161
	Some Examples Using CSS Styling.....	164
	Using CSS with XSLT .....	167
	Summary .....	171

<b>14</b>	<b>Linking in XML—XLink</b>	<b>172</b>
	The XML Linking Language .....	172
	XLink Attributes .....	175
	XLink Examples .....	175
	Document Fragments and XPointer .....	178
	XPointer and XPath .....	180
	Summary .....	187

## **PART 3    Programming XML**

---

<b>15</b>	<b>Presenting XML Graphically—SVG</b>	<b>188</b>
	What Is SVG? .....	188
	Advantages of SVG .....	190
	Creating SVG .....	191
	Some SVG Examples .....	193
	Summary .....	198
<b>16</b>	<b>The Document Object Model</b>	<b>199</b>
	The Document Object Model .....	199
	DOM Interfaces .....	201
	DOM Interfaces Properties and Methods .....	205
	Summary .....	209
<b>17</b>	<b>The Document Object Model—2</b>	<b>210</b>
	Creating a New Element .....	210
	Retrieving Information from the DOM .....	215
	Summary .....	220

## **PART 4    Where XML is Going**

---

<b>18</b>	<b>SAX—The Simple API for XML</b>	<b>221</b>
	What SAX Is and How It Differs from DOM .....	221
	Basics of SAX Programming .....	222
	Installing a SAX Parser .....	224
	Simple SAX Example .....	226
	Summary .....	230

<b>19</b>	<b>Beyond DTDs—W3C XML Schema</b>	<b>231</b>
	W3C XML Schema Basics.....	231
	Declaring Elements.....	233
	Defining Complex and Simple Types.....	238
	Summary.....	240

---

## **PART 5 Appendices**

---

<b>A</b>	<b>XML Online Resources</b>	<b>241</b>
	Web Sites .....	241
	Mailing Lists.....	243
<b>B</b>	<b>XML Tools</b>	<b>246</b>
	XML Editors .....	246
	XSLT Tools.....	247
	XLink and XPointer Tools .....	251
<b>C</b>	<b>XML Glossary</b>	<b>252</b>
	<b>Index</b>	<b>263</b>

## About the Author

**Andrew Watt** is an independent consultant and author with knowledge and interest in XML and graphics topics. He is the author of *Designing SVG Web Graphics* (New Riders, 2001) and *XPath Essentials* (John Wiley & Sons, 2002). He is a co-author of *XML Schema Essentials* (John Wiley & Sons, 2002), *Sams Teach Yourself JavaScript in 21 Days* (Sams, 2002), and *SVG Unleashed* (Sams, 2002). He is also a contributing author to *Platinum Edition Using XHTML, XML, and Java 2* (Que, 2000), *Professional XML, Second Edition* (Wrox Press, 2001), *Professional XSL* (Wrox Press, 2001), *Professional XML Meta Data* (Wrox Press, 2001), and *Special Edition Using XML, Second Edition* (Que, 2002).

# Dedication

*I would like to dedicate this book to the memory of my late father, George Alec Watt, a very special human being.*

# Acknowledgments

I would like to thank all the people at Sams Publishing who have made this book possible. Any book is the work of a team, not simply of a single person.

I would like to thank Todd Green and George Nedeff for keeping progress from idea to completion on course. I am grateful to Kevin Howard for careful developing, and Krista Hansing for concise editing.

I am also grateful to Steve Heckler for all his sensible suggestions. I couldn't take them all on board in a book of this size, but they were appreciated.

# We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an executive editor for Sams Publishing, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that I cannot help you with technical problems related to the *topic* of this book. We do have a User Services group, however, where I will forward specific technical questions related to the book.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:        [feedback@sampublishing.com](mailto:feedback@sampublishing.com)

Mail:         Michael Stephens  
                Sams Publishing  
                201 West 103rd Street  
                Indianapolis, IN 46290 USA

For more information about this book or another Sams Publishing title, visit our Web site at [www.sampublishing.com](http://www.sampublishing.com). Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

# Introduction

XML, the Extensible Markup Language, is the basis of many key technologies on the Web and many growing areas of software development. An understanding of at least the basics of XML is essential for an increasing number of Web developers and software developers.

XML bears many similarities to HTML, but it is sufficiently different to cause problems for many newcomers to XML. To write XML code that won't cause an XML parser to choke, you need to get many things right. Some are simple—for example, XML is case sensitive. Others are more subtle. It's important that you get on top of these issues right at the beginning of using XML.

This book teaches you how an XML document is correctly formed—"well-formed," in XML jargon—and how to make use of several of the key XML-related technologies, including XPath, XSLT, the Document Object Model, SAX, and the W3C XML Schema. By the time you finish reading this book, you will understand what these XML-related technologies do and will have grasped many key facts about each of these important technologies so that you can begin to use them for yourself.

## **Who Is *Sams Teach Yourself XML in 10 Minutes* For?**

This book is for you if

- You are new to XML and you want to get a handle on what XML is really about.
- You want to quickly learn the key facts about using XML and its important associated technologies.
- You want to know how to create well-formed XML.
- You want to get a handle on Document Type Definitions (DTDs).

- You want to understand how you can create HTML from XML using XSLT.
- You want to be able to change the structure of an XML document using XSLT.
- You want to learn the basics of programming XML using the Document Object Model and the Simple API for XML (SAX).
- You want to understand the basics of how W3C XML Schema works.

## What This Book Covers

This book shows you how an XML document is correctly structured. It also explains and demonstrates how to write XML so that it is well-formed and, therefore, acceptable to an XML parser.

XML is full of jargon, so new terms are explained as they are introduced. The book also has a glossary that explains or reminds you of what many terms mean and when they are used.

You'll be introduced to Document Type Definitions (DTD), which are used to define the allowed structure of a class of XML documents. You'll also learn about using entities in XML, to allow reuse of XML content.

In addition, the book discusses and demonstrates the use of Cascading Style Sheets (CSS) and Extensible Stylesheet Language Transformations (XSLT). You'll also be introduced to linking in XML using the XML Linking Language (XLink) and the XML Pointer Language (XPointer).

The book goes on to cover Scalable Vector Graphics (SVG), and it demonstrates programming XML using the Document Object Model and the Simple API for XML. The final chapter of the book introduces W3C XML Schema, a schema technology that goes beyond the capabilities of the DTD.

Of course, a book of this length can't tell you everything about XML and its associated family of technologies. You're pointed to sources of further information on XML in Appendix A, "XML Online Resources."



## What You Need to Use This Book

You don't need expensive tools to create XML code similar to the code you see in this book. If you want, you can use a plain text editor such as Windows Notepad. However, an XML-aware editor such as XML Writer (a 30-day free trial download is available at [www.xmlwriter.net](http://www.xmlwriter.net)) is better.

You will need XSLT software to run some of the code. Three free suitable downloads of XSLT software are listed in Appendix B, "XML Tools."

To view SVG, you will need an SVG viewer, such as the Adobe SVG Viewer; it can be downloaded free from [www.adobe.com/svg/](http://www.adobe.com/svg/).

To run the code for the DOM and SAX chapters, you will need a JavaScript interpreter (present in almost all Web browsers) and a Java Virtual Machine (already installed on most operating systems).

## Source Code and Updates

For updates to this book and to download the source code and examples presented in this book, visit [www.sampublishing.com](http://www.sampublishing.com). From the home page, type this book's ISBN (0672324717) into the search window and click Search to access information about the book and for a direct link to the source code.



# LESSON 1

## What Is XML?



*In this lesson, you will learn what XML is and the basics of how it is written.*

## What Is XML For?

Many newcomers to XML, the Extensible Markup Language, find it difficult to grasp what XML is and what XML is for. Part of the difficulty arises from the fact that XML is pretty abstract, and part is because XML can do many things, not just one. Only after you have explored XML for some time do the parts begin to come together to make sense. In addition, the number of XML language standards, such as XSLT, XPath, SVG, and XML Schema, is potentially intimidating.

XML documents are intended for the storage or exchange of data or information. XML “documents” can be used to store data that you would traditionally store as documents—letters, reports, manuals and so on—or data that you might associate with databases.



**Note** In XML, a *document* can be what you would normally think of as a document, but it also might be a complex, highly structured hierarchical data store.

A simple XML document could express a letter with a structure similar to the following code:

```
<?xml version="1.0" ?>
<letter>
 <salutation>Dear John</salutation>
 <paragraph>
 I look forward to meeting you on Saturday morning at the
 agreed location.
 </paragraph>
 <paragraph>
 It will be great to see you again after so many years.
 </paragraph>
 <ending>
 Yours sincerely,
 </ending>
 <signature>
 Janet
 </signature>
</letter>
```



**Note** Element type names are case sensitive in XML, unlike the situation that arises using HTML. This means that an element type named `myOrder` is a different element from `MyOrder` or `myorder`.

Equally, an XML “document” can store data that you might associate as being appropriate for storage in a database-management system. For example, you could use XML to store information for a human resources department using a structure similar to the following:

```
<Employees>
 <Employee>
 <Name>
 <FirstName>John</FirstName>
 <MiddleInitials>Q</MiddleInitials>
 <LastName>Campbell</LastName>
 </Name>
 <EmployeeID>12345</EmployeeID>
```

```
</Employee>
<Employee>
 <Name>
 <FirstName>Joan</FirstName>
 <MiddleInitials>D</MiddleInitials>
 <LastName>Dupois</LastName>
 </Name>
 <EmployeeID>01234</EmployeeID>
</Employee>
</Employees>
```

In addition to being suitable for storing many types of data, XML documents can be used to transmit messages, such as those sent across the Internet. If a single character is corrupted in an XML message, the message remains largely intact because each character is no more than that—a single character. Binary formats might be impossible to interpret if one byte is corrupted.

## XML Is Human-Readable

XML is, at least for developers, human-readable. Typically, an XML developer uses *element type names* (informally called tag names) that are meaningful.

The readability of a data store expressed in a format such as this

DE239, 0123, 01/12/24, 200.87, 02/01/03

is much improved by expressing it like this:

```
<order>
 <OrderNumber>DE239</OrderNumber>
 <CustomerID>0123</CustomerID>
 <OrderDate>01/12/24</OrderDate>
 <OrderAmount>200.87</OrderAmount>
 <DespatchDate>02/01/03</DespatchDate>
</order>
```

This is because someone looking at the information for the first time can quickly gather what the document is about.



**Tip** Be sure to use meaningful names for elements in your XML documents. XML doesn't make you use meaningful element type names for elements that you invent, but using meaningful names makes it easier for you to maintain the code and for newcomers to the code to understand the meaning of the documents that you create.

The improvement in readability is bought at the price of increasing file size and, if included in a message, increasing transmission time.

## Is XML Usable on the Web?

When XML 1.0 was first finalized in 1998, there was a lot of expectation—at least some of it hype—that XML would be transmitted on the Web as XML. At the time of this writing, this hasn't happened, partly because conventional (HTML) Web browsers have added only limited XML capabilities to the existing HTML functionality. Another factor has been the prolonged delays in completing the W3C recommendations for the XML Linking Language (XLink, completed in mid-2001) and the XML Pointer Language (XPointer, not yet a recommendation at the time of this writing). Using XML as XML on the Web has been unrealistic without functional XML linking (XLink) to link to external documents and fragment identifier (XPointer) technologies to link to specified parts of documents.

In addition, XML has lacked forms capabilities for data collection. At the time of this writing, the W3C XForms specification is two steps from being finalized and will likely be finalized soon after the time this book is published.

After XLink, XPointer, and XForms are finalized, using XML application languages on the Web—including the XML-based Scalable Vector Graphics (SVG) specification for vector graphics—becomes more realistic. New XML-dedicated browsers, such as the X-Smiles browser (see <http://www.x-smiles.org>), are appearing, making all-XML Web sites

and browsers potentially viable. Users will decide whether the functionality offered is sufficiently improved to displace HTML's current dominant position.

In the meantime, the practical solution that many have chosen is to store information as XML (or in a relational database-management system that can export XML) and present that XML content after transformation to HTML, which is displayed in the conventional way in a Web browser. XSLT transformations are described in Chapter 10, "XSLT—Creating HTML from XML"; Chapter 11, "XSLT—Transforming XML Structure"; and Chapter 12, "XSLT—Sorting XML."



**Note** *Transformation* is the process of selecting desired data in an XML document or data store and either restructuring it as another XML document or producing a non-XML document, such as HTML.

## XML Is a Markup Language

As its name suggests, XML is a markup language. Markup is used to convey some information about text or other data. XML has similarities to other markup languages.

Markup may be used to indicate how text is to be presented. In a word-processing program, for example, hidden codes communicate to the word-processor software various settings that are applied to a document but that are not visible. Typically, the codes used in word-processing programs are not easy for a typical user to read or decipher.

The Hypertext Markup Language (HTML) uses angled brackets and tags to convey something about the structure of a document to be presented on the World Wide Web. An `h1` tag, for example, indicates a top-level heading. HTML markup typically merges information about the presentation of text content with its meaning. The `h1` tag, for example, indicates not only a top-level heading but likely also specifies, according to browser settings, a particular font size, and so on.

Markup may also be used to indicate something about the meaning of the text content. For example, if you wanted to express some information about this book using XML syntax, you could write this:

```
<?xml version="1.0" ?>
<book>
 <title>Sams Teach Yourself XML in 10 Minutes</title>
 <author>Andrew Watt</author>
 <publisher>Sams Publishing</publisher>
</book>
```

The use of meaningful element type names results in a logical structure that is better expressed than when using HTML tags. Notice that the preceding code says nothing about how the information is to be presented.

## XML Is a Meta Language

XML differs from markup languages such as HTML because XML can have any number of *element type names* that often are informally called *tag names*. This gives application languages that use XML syntax the functionality to represent data from any subject domain. Because all these XML application languages use the same syntax rules, those languages can be processed using a set of common core software tools that understand XML syntax rules, with custom tools being necessary for fewer aspects of processing. This makes possible efficiencies that facilitate data exchange between users and between applications.

A meta language can be thought of as a set of grammar rules. The application languages that follow the specified set of rules can be thought of as a vocabulary. Broadly, in the natural language domain there are sets of rules that specify how words are used with punctuation and so on. Many Western languages broadly follow similar sets of rules at that level, with uppercase initial letters to start a sentence, periods to complete a sentence, and so on. But the words used in each language, such as in English and French, are very different.

Of course, the syntax rules that apply to XML-based application languages are much more tightly defined and consistent than the situation in the natural language realm, but the principles are similar. Using this single



set of XML syntax rules, you—or, often more importantly, the World Wide Web Consortium (W3C)—can define multiple languages that follow XML syntax rules precisely. This facilitates processing of code written in those languages because a generic XML processor (also called an XML parser) can parse the characters of files written in that language and can extract its logical structure and pass that logical information to another application more specific to the needs of the application language.

## You Can Create Your Own Vocabulary

XML 1.0 is a set of syntax rules. You can create your own set of element type names, as you learned in the short examples earlier in this chapter.

Already the W3C has published specifications for many application languages of XML, including languages to describe the transformation of XML (XSLT, the Extensible Stylesheet Language Transformations, described in Chapters 10–12), to create links among XML documents (the XML Linking Language, XLink, described in Chapter 14, “Linking in XML—XLink”), and to describe vector graphics (SVG, Scalable Vector Graphics).

The syntax rules of XML can also be used by corporations or business consortia to create XML application languages in various business or technical domains. Many business and technical domains have already created common vocabularies to assist data exchange. Common vocabularies use schemas to define the allowed content and structure of a class of XML documents. Schemas can be written using non-XML syntax—Document Type Definitions, described in Chapter 4, “Valid XML—Document Type Definitions”—or using XML syntax, the W3C XML Schema language described in Chapter 19, “Beyond DTDs—W3C XML Schema.”

This enormous flexibility of XML needs some order applied to it—after all, if everyone used element type names just as they personally wanted, individuals or companies would use different element type names to describe the same thing or would use the same element type names to describe different concepts. In fact, these problems are pretty much impossible to avoid; this potential for element name clashes led to the

development of namespaces in the XML specification. Namespaces are described in Chapter 8, “Namespaces in XML.”

## How Does XML Relate to HTML?

XML is a meta language, a set of syntax rules. HTML is an application language, a predefined vocabulary that uses the rules of another meta language, Standard Generalized Markup Language (SGML).

Both XML and HTML derive from SGML, but in different ways. XML uses a subset of the syntax rules allowed in SGML documents. In principle, HTML can use all the syntax rules of SGML, but only as a defined vocabulary that makes use of the SGML syntax rules.

HTML has a defined number of tags defined by specification documents produced by W3C. An HTML processor—a Web browser—should process only those tags that are officially approved. With the exception of some legacy or unofficial tags, such as the `embed` tag, this is what an HTML processor does. XML documents, by contrast, can contain any element names that you choose (subject to the rules for legal characters in XML names, described in Chapter 3, “XML Must Be Well-Formed”).

HTML element names are not case sensitive. A Web browser processes a paragraph identically whether it is written as

```
<p>This is a test paragraph</p>
```

or as

```
<P>This is a test paragraph</P>
```

XML, by contrast, *is* case sensitive. For the names of two elements to be treated as identical, each character must match exactly. The `Title` and `title` elements in the following code snippet are treated as different elements by an XML processor:

```
<book>
 <Title>Sams Teach Yourself XML in 10 Minutes</Title>
 <author>
 <title>Mr.</title>
 <Name>Andrew Watt</Name>
 </author>
</book>
```

Because the `Title` and `title` elements contain different types of information, it might be desirable to distinguish them. But it is less confusing to choose element type names such as `BookTitle` and `PersonalTitle`.

## Separating Content from Presentation

When most Web programming was done on a small scale by individuals, the problems arising from mixing content and presentation existed but weren't a major problem for small-scale HTML users. If you had to make many individual changes, the total number was unlikely to be enormous.

One of the disadvantages of HTML for large projects is that content and presentation are entangled. An `h1` tag indicates that the contained text is a heading, but, depending on the way the browser is configured, it also indicates something about how the text is to be displayed. Increasingly, HTML documents separate presentation from content by using Cascading Style Sheets (CSS).

In XML, by contrast, typically a document describes only the logical structure of the data:

```
<book>
<title>The Bible</title>
<TestamentTitle>The Old Testament</TestamentTitle>
<TestamentTitle>The New Testament</TestamentTitle>
</book>
```

You might expect the text content of the `title` and `TestamentTitle` elements in the preceding code to be displayed as headings, but the XML document contains no information about exactly how they should be presented. Typically, an XML document is styled using CSS style sheets or using XSLT.



**Note** For historical reasons, a CSS style sheet is two words and an XSLT stylesheet is one word.

## How Is XML Written?

XML is written using tags. An XML element has a *start tag* that is delimited by angled brackets—written as `<myStartTagName>`—and an *end tag*—written as `</myStartTagName>`—which is also delimited by angled brackets. For example, to create a `title` element in XML, you could write this:

```
<title>Some book title</title>
```

Unlike in some parts of HTML, tags in XML must be used in pairs. Omitting an end tag causes the XML processor to generate an error. Each start tag in XML must have a matching end tag.



**Note** An *XML processor* is a piece of software that processes the characters of an XML document and makes available to another piece of software—often called the application—the logical structure of the XML document.

In HTML, you could write this

```
<p>This is a paragraph
<p>This is another paragraph
```

and the Web browser would figure out that the first `p` tag ends at the end of the first line of code. In HTML, you could also write this to close the tags explicitly:

```
<p>This is a paragraph</p>
<p>This is another paragraph</p>
```

Allowing syntax options in this way makes an HTML processor more complex than if a rule existed that all tags had to be closed.

XML uses that principle to simplify the writing of XML processors. All XML elements must have a matching end tag for each start tag. This means that you must write XML documents correctly or the XML processor will signal an error and stop processing.

If you needed an XML element to describe the title of a book, you could write the title element as follows:

```
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt</author>
```

An error would result if you completed the document and did not provide an end tag, `</title>`, to balance the `<title>` start tag.

## Adding Attributes to Elements

XML elements often contain qualifying information contained in *attributes*. XML attributes are written inside the start tag of an element, after the element type name and separated from it and any other attributes by whitespace. Finally, optionally, the attribute name/value pair is separated from the closing angled bracket of the start tag by a space character.

Attribute values in XML must be surrounded by paired quotation marks, either a pair of double quotes or apostrophes.

An XML element with one or more attributes could be written like the following code:

```
<Invoice date="2002/12/23">
<InvoiceNumber Dept='ID4'>DA890</InvoiceNumber>
<BillTo>John Smith</BillTo>
<!-- And so on -->
</Invoice>
```

The value of the date attribute on the Invoice element uses paired double quotes, and the value of the Dept attribute is delimited by paired apostrophes. Either paired character is acceptable to an XML processor.

It is an error to omit the paired quotation marks. The following is not legal XML:

```
<Invoice date=2002/12/23>
<!-- And so on -->
</Invoice>
```

## Try It Yourself

You have seen how to create simple XML documents that contain elements and attributes. Many XML editors, such as those described in Appendix B, “XML Tools,” check that you have written XML documents correctly and help you find errors. Some XML editors with that capability, such as XML Writer (see Appendix B), have a free download that you can try if you want.

## Summary

In this lesson, you learned what XML is, what XML can be used for, and how to correctly write XML elements and attributes.

## LESSON 2

# The Structure of an XML Document



*In this lesson you will learn the permitted structure of an XML document and see examples of allowed variations on the basic structure.*

## An XML Document

An XML document has the following general structure but some parts are optional:

- A *prolog*, which may optionally be empty
- At least one element, the *document element*, although a typical XML document has a nested structure of elements
- Optional content following the end tag of the document element

Each of these permitted parts of an XML document is described in the remaining sections of this chapter.

A minimal XML document can consist of a single *document element*, such as this:

```
<someElementTypeName>The content</someElementTypeName>
```

Typically, XML documents are much more complex and have a nested structure of elements. An XML document that uses more of the allowed structures might look like this:

```
<?xml version="1.0" ?>
<!-- This is an example XML document. -->
<!DOCTYPE book >
<book>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt</author>
</book>
```

First let's look at what can go in the prolog.

## Prolog

The *prolog* of an XML document, when present, precedes the *document element*.

The prolog may, but need not, contain the following:

- An XML declaration
- Miscellaneous content—processing instructions or comments
- A Document Type Declaration, also called a DOCTYPE declaration

Let's look at each of these in turn.

## The XML Declaration

The XML declaration can be as simple as this:

```
<?xml version="1.0" ?>
```

This declaration is an optional part of all XML documents. However, when an XML declaration is present in an XML document, it must occur in the first line of the XML document and must have no characters—not even a single space character—before it.

An XML declaration, if present, must have a `version` attribute. For XML documents that correspond to version 1.0 of XML, the only permitted



value for the version attribute is 1.0. Future versions of XML might use different version numbers, allowing XML processors to process only versions of XML that they recognize.



**Note** At the time of this writing, only a value of 1.0 is meaningful. However, version 1.1 of XML is currently in draft at the W3C.

Optionally, an XML declaration may include an encoding attribute. When present, this attribute indicates the character encoding being used in the document. All XML processors must be capable of processing XML documents encoded in the UTF-8 and UTF-16 character encodings. So, if an XML document is encoded in UTF-8 or UTF-16, no encoding attribute is needed because all conforming XML processors will be capable of processing the document. Optionally, XML processors may choose to support additional character encodings. When using other encodings, it is advisable to specify an encoding attribute and the appropriate value. Character encoding is discussed further in Chapter 6, “Characters in XML.”



**Note** Strictly speaking, an XML declaration is not a processing instruction, although the initial delimiter, `<?`, and the closing delimiter, `?>`, are the same as those used by processing instructions. These are described in the following section.

The XML declaration may also include a standalone attribute. The standalone attribute takes the values of yes or no. If external *markup declarations* supply default values for attributes or if entities (other than built-in entities—see Chapter 5, “XML Entities”) are declared, the standalone attribute must have the value of no.



**Note** A *markup declaration* declares the existence of, for example, an element or attribute in the relevant class of XML documents. It defines permitted or default values in some circumstances.

## Comments and Processing Instructions

XML documents may contain comments, which contain information intended for human consumption or processing instructions.



**Note** XML comments and processing instructions described in the following sections may occur in the prolog, within the document element, and after the end tag of the document element.

### XML Comments

XML comments may occur anywhere outside other markup. In other words, comments cannot be used within the start or end tags of elements, within processing instructions, or within entity references, empty element tags, character references, CDATA section delimiters, Document Type Declarations, XML declarations or text declarations. Any unfamiliar terms are explained in more detail later.

XML comments use the same syntax as HTML comments:

```
<!-- This is an XML comment -->
```

The character sequence `<!--` is the starting delimiter of an XML comment, and the character sequence `-->` is the ending delimiter. The text contained between these delimiters can contain any characters (with exceptions described in the next paragraph), including those that must be escaped when present within element or attribute values (such as the left angle bracket or right angle bracket). Implicitly, the text content of an XML comment is marked as unparsed information and, therefore, need not satisfy the requirements for other parsed content.

For compatibility with SGML, the character string `--` must not occur within an XML comment. Also, it is illegal to end an XML comment with the character sequence `-->`, which has three consecutive dashes.

## XML Processing Instructions

An XML document is viewed in the XML 1.0 Recommendation as being parsed by an XML parser that then passes the results of that parsing to an *application*. Sometimes it might be appropriate to pass to the application instructions intended for the use of the application rather than the results of parsing human-readable markup. Because processing instructions are intended for use outside the XML processor, they are not considered part of the character content of the XML document.

An XML processing instruction is delimited by the starting character sequence `<?` and by the ending character sequence `?>`.

An XML processing instruction takes the following general form:

```
<? target characterSequence ?>
```

Here, *target* is any XML name, excluding the character sequence `xml` or `XML`, in any combination of upper- or lowercase. The *target* identifies the application to which the *characterSequence* should be passed. The *characterSequence* that constitutes the message to the application must not include the character sequence `?>`, otherwise, an XML processor would assume that the processing instruction had been completed.

One common example of the use of a processing instruction is the `xml-stYLESHEET` processing instruction used to associate a stylesheet—either XSLT (Extensible Stylesheet Language Transformations) or CSS (Cascading Style Sheet)—with an XML document.

For example, to associate a stylesheet stored in a file called `myStylesheet.xml`, you might use the following processing instruction to associate the XSLT stylesheet (introduced in Chapter 10, “XSLT—Creating HTML from XML”) with the XML document, assuming that the stylesheet is stored in the same directory as the XML document:

```
<?xml-stYLESHEET href="myStylesheet.xml" type="text/xml"?>
```

## Document Type Declaration

The Document Type Declaration, also called the DOCTYPE declaration, is optional. If present, it must occur before the document element. The DOCTYPE declaration specifies the element type name of the document element and, optionally, references external markup declarations or may include markup declarations.

The simplest form of the Document Type Declaration may be written as follows:

```
<!DOCTYPE theDocumentElement >
<theDocumentElement>
<!-- The content of the document element would go here -->
</theDocumentElement>
```

As written, the preceding DOCTYPE declaration simply indicates that the document element of the XML document has the element type name of `theDocumentElement`.

In addition to indicating the element type name of the document element, the DOCTYPE declaration contains or points to the markup declarations that optionally define the permitted structure of an XML document. The markup declarations provide a grammar for a class of XML documents.

A DOCTYPE Declaration, when present, may have two additional optional parts:

- An indication of the location of the *external subset* of the Document Type Definition, DTD (see Chapter 4, “Valid XML—Document Type Definitions,” for further explanation)
- A set of markup declarations that constitute the *internal subset* of the DTD (also discussed in Chapter 4)

When an external subset of the DTD exists, it may be expressed as a relative URI, as follows:

```
<!DOCTYPE documentElementName SYSTEM "myDTD.dtd" >
```

Here, `SYSTEM` is a keyword indicating that the location of the external subset of the DTD is specified relative to the current system. The *system*

*identifier*—myDTD.dtd, in this case—is a uniform resource identifier (URI) that indicates the location of the external subset.

A system identifier may be identified by a relative URI, as in the preceding example, or a full URI reference, as follows:

```
<!DOCTYPE myElement SYSTEM "http://www.XMML.com/myDTD.dtd">
```

An alternate syntax is to specify a *public identifier*, using the keyword PUBLIC, together with a system identifier:

```
<!DOCTYPE documentElementName PUBLIC publicidentifier SYSTEM
myDTD.dtd >
```

This form is used when a public identifier is appropriate—for example, in the DOCTYPE declaration of the XML application language Extensible Hypertext Markup Language (XHTML) documents. For XHTML version 1.0, the DOCTYPE declaration is as follows:

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

The public identifier indicates an organization—in this case, the W3C—and also includes an indication of the language—in this case, English, indicated by EN.

The syntax for defining the internal subset of the DTD is described in Chapter 4.



**Caution** The Document Type Declaration and the Document Type Definition (DTD) are not identical, although they are closely related. The Document Type Declaration defines the location of the external subset of the DTD and optionally may contain the internal subset of the DTD.

## Document Element

Each XML document must have one—and only one—*document element*, which is sometimes referred to as the *root element*.



**Caution** Be careful if you use or read the term *root* without any further clarification. The XML 1.0 specification uses the term apparently as a synonym for the document entity (Chapter 2 of the XML 1.0 Recommendation) and also as a synonym for the document element (Chapter 2.1).

## Elements Nested Inside the Document Element

Typically, nested within the document element is a hierarchy of other elements, represented as start tag/end tag pairs or as empty element tags.

Elements must be nested correctly. The following code shows a correctly nested pair of elements:

```
<chapter>
 <section number="1">Some content
</section>
</chapter>
```

The start tag of the `section` element follows the start tag of the `chapter` element, so the end tag of the `section` element must come before the end tag of the `chapter` element.

When correctly nested as shown in the preceding code, the `chapter` element is termed the parent element of the `section` element. The `section` element is termed the child element of the `chapter` element. An element may have zero, one, or more child elements. An element may have zero or one parent elements. All elements except the document element have one parent element. The document element does not have a parent element.

The following is not legal XML because the elements are not nested correctly:

```
<chapter>
<section number="1">Some content
</chapter>
</section>
```

## Attributes

An element may be qualified by adding attribute name/value pairs to its start tag. An attribute is indicated by an XML name, followed by the equal sign and a pair of either quotation marks or apostrophes. The *value* of the attribute is contained within those quotation marks or apostrophes.

So, if you want to indicate that this is the first edition of this book and that it is published in English, you could add `edition` and `language` attributes to the start tag of the `book` element:

```
<book edition="1" language='English'>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt</author>
<publisher>Sams Publishing</publisher>
</book>
```

The value of the `edition` attribute is contained in a pair of quotation marks. The value of the `language` attribute is contained in a pair of apostrophes.

It is an error to mix double quotes and single quotes as the delimiters of an attribute. In the following code, both the `edition` and `language` attributes would generate an error from the XML processor because the attribute value delimiters are not correctly paired.

```
<book edition="1" language='English' >
```

## CDATA Sections

An XML document may contain information expressed in a non-XML syntax. The XML mechanism for indicating that such content is not to be parsed as XML is the *CDATA section*.

The starting delimiter of a CDATA section is the character sequence `<![CDATA[`. The ending delimiter is the character sequence `]]>`. The character sequence `]]>` cannot be used as the content of a CDATA section.

CDATA sections can be used to store XML code, such as code snippets used in this book, without having to escape all characters that an XML processor would recognize as markup. For example, if the text of this chapter were written and stored in XML, you would not want example code to be parsed. Thus, if you wanted to create in XML the text for a section of this book that referred to example text expressed as XML, you could write something like this:

```
<example>
<![CDATA[
<book>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt</author>
</book>
]]>
</example>
```

If you didn't use a CDATA section, you would have to write this:

```
<example>
<book>
<title>Sams Teach Yourself XML in 10
 Minutes</title>
<author>Andrew Watt</author>
</book>
</example>
```

It very quickly becomes tedious having to write `&lt;` for each `<` character and `&gt;` for each `>` character in a section of example code. The CDATA section is more convenient.

One common use of CDATA sections appears in Scalable Vector Graphics (SVG) documents (SVG is an XML application language for two-dimensional graphics), which can contain scripting code written, for example, in JavaScript. The general structure using the SVG script element would look as follows:



```
<script type="text/javascript" >
<![CDATA[

//JavaScript code goes here
]]>
</script>
```

CDATA sections cannot be nested within each other because the starting delimiter `<![CDATA[` is recognized only as a sequence of characters, not as a starting delimiter of a nested CDATA section. Only the ending delimiter character sequence, `]]>`, is recognized as markup within a CDATA section.

## Text Content

Text, which is basically a sequence of characters, may occur between the start tag and end tag of an element; it is said to be (or to form part of) the element's *content*.

Most English alphabetic or numeric characters can simply be typed as normal. Certain characters must not be used in text content, however. The following simple description of an arithmetic axiom in XML generates an error in an XML processor:

```
<axiom> 1 < 2 </axiom>
```

An XML processor recognizes the less than sign between 1 and 2 as the starting angle bracket of a new tag. An error results upon finding a space and then a number (which is not allowed to start an XML name). The following characters must be escaped to use them in text content:

- `<` (The less than symbol)—Must be written as `&lt;`;
- `>` (The greater than symbol)—Must be written as `&gt;`;
- `'` (The single quotation mark)—Must be written as `&apos;`;
- `"` (The double quotation mark)—Must be written as `&quot;`;
- `&` (The ampersand)— Must be written as `&amp;`;

The alternative is to use these characters written literally (that is, not escaped) within a CDATA section. The choice of whether to escape characters or to enclose them inside a CDATA section often depends on how many

characters in a particular section of text require escaping. The more characters need escaping, the more likely it is that using a CDATA section offers the most convenient solution.

## **Content After the Document Element End Tag**

The XML 1.0 specification allows content to follow the end tag of the document element. However, permitted content is restricted to only comments, processing instructions, and whitespace. In practice, this means that all document content must be nested within the document element. Markup after the end tag of the document element can contain only information intended for a human reader, given in comments, or one or more processing instructions for the XML processor or the application. No constraint affects the ordering of comments or processing instructions.

## **Summary**

In this lesson, you learned about the structure of an XML document and saw examples of how to write the most common parts of that structure.

## LESSON 3

# XML Must Be Well-Formed



*In this chapter you will learn the rules that an XML document must satisfy to be considered well-formed.*

## Well-Formed XML Documents

If XML is to be used as a format for data interchange, it must adhere to a consistent syntax so that programs can reliably produce and parse XML documents. An XML document that adheres to proper XML syntax is said to be well-formed.

If the results of parsing are to be presented by an XML processor (also known as an XML parser) to its associated application, the XML document must be *well-formed*. If the document is not well-formed, the XML processor should report one or more errors encountered, and normal processing, including the passing of parsed data to the application, should stop. Ensuring that the XML documents that you write are well-formed is crucial to achieving the desired processing of the data that they contain.

Some of the rules for well-formedness are straightforward. Some can seem pretty obscure the first time you read them, so if some of the rules in this chapter don't make too much sense the first time through, don't worry too much. As you learn more about other aspects of XML in later chapters, the pieces of the syntax jigsaw will fit together more clearly.

In Chapter 2, "The Structure of an XML Document," you learned about the structure that an XML document must conform to. All well-formed XML documents must follow the permitted options of that structure. In addition to those rules, an XML document must satisfy several other rules to be considered well-formed.



**Note** This chapter gives a complete description of well-formedness constraints. To do so, it is necessary to refer to concepts described more fully in later chapters. You might find it helpful to reread parts of this chapter after reading Chapter 4, “Valid XML—Document Type Definitions,” and Chapter 5, “XML Entities.”

The term *well-formed* is used to describe the rules that all XML documents must satisfy. If an XML document is not well-formed, an XML processor signals an error and stops normal processing. It is crucial that you understand the well-formedness constraints in XML 1.0, to ensure that the XML documents that you create will be processed correctly and without errors.

To be well-formed, an XML document must satisfy each of three broad rules or sets of rules:

- The structure of the document must follow that described in Chapter 2—an optional prolog, followed by a required document element (and any content that it has) and, finally, an optional miscellaneous section.
- The document must satisfy the well-formedness constraints described in the following sections of this chapter.
- Any parsed entities referenced from the document, whether directly or indirectly, must themselves be well-formed.

The following several sections consider each of the XML 1.0 well-formedness constraints.

## XML Names

The XML 1.0 Recommendation places restrictions on the characters that may be used in legal XML names and imposes tighter restrictions on the characters that may be used as the first character in an XML name.

## Initial Characters of XML Names

In English, the initial character of an XML name must be either a letter (from A to Z—both upper- and lowercase are legal), the colon character (:), or the underscore character (\_).



**Tip** Avoid the colon character as the first character in an XML name. Using that character is legal but could cause confusion when you create XML documents using elements from several XML namespaces (described in Chapter 8, “Namespaces in XML”).

XML is case sensitive, so the following two elements are considered in XML to be different elements because of the difference in case:

```
<p></p>
```

```
<P></P>
```

In some other languages, ideographic characters may also be used as the initial character of an XML name.

It is illegal to start an XML name with a numeric character. The following code generates an error because 2d is not a legal XML name:

```
<?xml version='1.0'?>
<myElement>
 <2d>
 Some content.
 </2d>
</myElement>
```



**Caution** Names in XML 1.0 must not begin with the character sequence `xml` or `XML`, in any combination of upper- or lowercase.

All characters that are legal as the first character of an XML name can be used in later positions within an XML name as well.

## Non-Initial Characters of XML Names

The non-initial characters of an XML name are allowed to include characters not permitted as the first character of an XML name. The additional allowed characters are numeric characters from 0 to 9 inclusive, the hyphen character, and the period character.



**Tip** Again, avoid using the colon character later in XML names. Later you will want to mix XML documents from different namespaces (discussed in Chapter 8), and the colon character has special meaning in those circumstances. Avoiding the colon character except in namespace-aware documents means that you won't have to change your documents if you want to use multiple namespaces later.

## Elements

XML elements must have as their element type name a legal XML name as defined in the preceding section.

Additionally, the element type name in the start tag must match the element type name in the end tag. It is important to remember that XML is case sensitive. Any of the following tag pairs will generate well-formedness errors because of case differences:

```
<title></TITLE>
```

```
<title></Title>
```

```
<title></Title>
```

## Balanced Start and End Tags

Each start tag must have a corresponding end tag, properly nested. The following example is correctly nested:

```
<oneElement>Some text
 <anotherElement>Some other text
</anotherElement>
</oneElement>
```

The end tag of the `anotherElement` element must appear before the end tag of the `oneElement` element.

If an element is empty (that is, it has no content, not even a single white-space character), the start tag/end tag pair can be written as an empty element tag. The following

```
<someElement myAttribute="someInformation"></someElement>
```

is equivalent to writing this:

```
<someElement myAttribute="someInformation"/>
```

## Attributes

An attribute is the association of an attribute name with an attribute value. For example, the markup describing this book might be represented as follows:

```
<book edition="1">
<title>Sams Teach Yourself XML in 10 Minutes</title>
</book>
```

The `book` element has an `edition` attribute, which has the value of 1.

An attribute is allowed only in the start tag of an element. The value of an attribute must be enclosed between paired double quotation marks, such as

```
<book edition="1">
```

or between paired apostrophes:

```
<book edition='1'>
```

It is an error to mix these two types of delimiters. The next two line of code would cause a well-formedness error because the delimiters of the attribute value are not paired.

In the following line, the quotation mark before the attribute value is not paired with a matching quotation mark:

```
<book edition="1'>
```

Here, the apostrophe before the attribute value does not have a matching apostrophe to delimit the end of the attribute value:

```
<book edition='1">
```

## Attributes Must Be Unique

The start tag of any XML element must not contain duplicate attribute names.

For example, the following code will generate an error because there are two number attributes in the start tag of the chapter element.

```
<chapter number="1" author="DPT" number='1'>
<!-- Some text would go here -->
</chapter>
```

## No External Entity References

Attribute values are not allowed to contain external entity references.

For example, imagine that you had declared an external entity called `copyright`:

```
<!ENTITY copyright SYSTEM "copyright.xml">
```

You could not use it in an attribute value, such as in the following code:

```
<book status="©right;">
<!-- Content goes here -->
</book>
```

However, attribute values are allowed to contain references to internal parsed entities.

For example, imagine that the internal subset of the DTD included an entity declaration as follows:

```
<!ENTITY BigText "font-size:72">
```



It could be used to define the style of text nested in an SVG text element, as follows:

```
<text style="&BigText;">This text is big!!</text>
```

## No < in Attribute Values

The value of an XML attribute is not allowed to include the < character, either directly or indirectly.

It is not legal to write this:

```
<math comparison="3<4"></math>
```

It is also an error to include an internal entity reference to an entity, such as

```
<!ENTITY lessthan "3<4">
```

which is referenced like this:

```
$$
```

After the entity reference was replaced by its replacement text, it would result in the same illegal code:

```
<math comparison="3<4"></math>
```

Following the well-formedness constraints described for elements and attributes in the preceding sections will help you avoid many of the common well-formedness errors. As your documents become more complex, other well-formedness constraints might become important.

## Other Characteristics of Well-Formedness

This section covers the well-formedness constraints as they apply to areas other than elements and attributes. Several of the well-formedness rules will make more sense after you have read Chapters 4 and 5.

## Comments

To be well formed, XML comments must not end with the character sequence `-->`. A legal XML comment ends with `-->`— that is, two dashes and a greater than character.

## Entities Must Be Declared

If a reference to an entity is present in an XML document and any of the following are true, any entity referenced must be declared:

- There is no Document Type Definition (DTD).
- A document has only an internal DTD subset with no parameter entity references.
- The value of the `standalone` attribute in the XML declaration is `yes`.

The exception to this rule is that the built-in entities `amp`, `apos`, `gt`, `lt`, and `quot` need not be declared.

## External Parsed Entities

External parsed entities must be well-formed. They optionally begin with a *text declaration* (described more fully in Chapter 5). A text declaration is similar to an XML declaration but may have only `version` and `encoding` attributes. A text declaration does not have a `standalone` attribute. Following the text declaration, the structure need not all be contained in a single element—the document element is contained in the XML document entity that references the external parsed entity.

The allowed content is any combination of the following:

- Character data
- Elements
- Entity references
- Character references
- CDATA sections

- Processing instructions
- Comments

The content following the text declaration is essentially the same as the permitted content of an element anywhere in the document entity. That is not surprising because an external parsed entity can have replacement text that constitutes the content of an element in the document entity.

An internal parsed entity is well-formed if the replacement text matches the content that was listed in the preceding list.

No element or other markup may begin in one external entity and end in another.

## **Parsed Entities: No Recursion**

Entities are discussed in more detail in Chapter 5. A parsed entity is not permitted to directly or indirectly reference itself.

## **Parameter Entity References in the DTD**

Parameter entity references may appear only in the DTD.

## **Parameter Entity References in Internal Subset**

Parameter entity references in the internal subset of the DTD, which is described more fully in Chapter 4, can occur only where markup declarations can occur and not within other markup declarations.

## **External Subset of the DTD**

The external subset of the DTD may have the following structure. It may optionally begin with a text declaration (described in Chapter 5) and may also contain markup declarations or conditional sections (both described in Chapter 4).

Two separators are allowed between markup declarations, parameter entities and whitespace.

## Parameter Entities in Markup Declarations

A parameter entity has replacement text. The replacement text of a parameter entity must satisfy the constraints on declarations, as described in the preceding section, so that the replacement text nests properly.

### Replacement Text

When an entity reference appears in an attribute value or a parameter entity appears in an entity declaration, the replacement text might contain either a double quotation mark or an apostrophe. In this situation, when the double quotation mark or the apostrophe is applied as replacement text, it is treated as a literal character of the relevant type, not the closing delimiter of the attribute value.

For example, with the entity declaration

```
<!ENTITY myEntity "something with an apostrophe">
```

this entity reference is well-formed:

```
<someElement someAttribute='&myEntity' />
```

The same is true with a parameter entity such as this one:

```
<!ENTITY % hisStatement '"I agree."'>
<!ENTITY aSentence "He said, &hisStatement;">
```

The replacement text for the `hisStatement` parameter entity contains two double quotation marks, the first of which would normally be the closing delimiter of the replacement text of the `aSentence` entity. However, both double quotation marks contained in the parameter reference are treated literally, not as the closing delimiter.

### Character References

In XML documents, a character may be referenced using a character reference. You might want to use a character reference when, for example, a character cannot be typed from the keyboard. A character reference beginning with `&x#` indicates a hexadecimal reference to a character's code point. For example, the uppercase A may be written as the character reference, `&x#0041;`.

```
<?xml version='1.0'?>
<capitalA>A</capitalA>
```

Character references can also be expressed using the `&#` syntax, indicating a decimal reference to a character's code point. Using this syntax, you can represent the uppercase A as `&#0065`;, as shown here:

```
<capitalA>A</capitalA>
```

## Declaring Predefined Entities

If compatibility with SGML is not an issue in your use of XML, this well-formedness constraint can perhaps be ignored. However, it might help you understand what at first sight could appear to be strange entity declarations in XML documents created by others.

All XML processors must recognize the entities `amp`, `apos`, `gt`, `lt`, and `quot`, whether they are explicitly declared or not. However, if compatibility with SGML is a relevant issue, these predefined entities must be explicitly declared in the internal subset of the DTD.

You might recall that you cannot use the literal characters `&`, `'`, `>`, `<`, and `"` in well-formed character data without causing a well-formedness error. Therefore, you cannot use any of these characters literally as the replacement text of an entity declaration. The solution is to use character references for each of the characters.

Therefore, entity declarations used for compatibility with SGML can be written as follows:

```
<!ENTITY amp "& &">
<!ENTITY apos "' '">
<!ENTITY gt "> >">
<!ENTITY lt "< <">
<!ENTITY quot "" "">
```

## Well-Formedness and XML Processor Type

XML processors can be viewed as being of two types, validating processors and nonvalidating processors. Validation is discussed further in Chapter 4.

Both validating processors and nonvalidating processors detect any well-formedness errors in the document entity, including the internal subset of the DTD. However, their behavior might differ with external entities and the external subset of the DTD.

A validating processor must process the XML document entity, any external entities, and the DTD (both internal subset and external subset). It must access those fully to validate the XML document. In doing so, it detects any well-formedness errors in any physical part of an XML document.

Nonvalidating processors, on the other hand, need not access an external subset of the DTD (if it exists) or external entities. So, well-formedness errors might not be detected by nonvalidating processors if the well-formedness errors occur outside the document entity itself.

Another potential source of confusion with nonvalidating processors is that although they are not obliged to access parts of the document other than the document entity, nothing in the XML specification prevents them doing so. Therefore, one nonvalidating processor might detect errors that another nonvalidating processor misses because the former processor accessed external entities that the latter ignored. This could be confusing if you find no errors in your code but a recipient of a document that you have written finds errors. The recipient might simply be using a nonvalidating processor that checks more parts of the document than the nonvalidating processor that you used to run the code.

## Summary

In this lesson, you learned all the XML 1.0 well-formedness constraints. Those relating to elements and attributes likely will be most directly relevant to straightforward XML documents. Other well-formedness constraints less likely will be directly useful to simple code. However, you might find that knowing these constraints helps you write code that runs correctly.

## LESSON 4

# Valid XML— Document Type Definitions



*In this lesson, you will learn what valid XML documents are, why document type definitions for XML documents are needed, and how to write a DTD for XML documents.*

## Shared Documents: Why We Need DTDs

The well-formedness rules examined in Chapter 2, “The Structure of an XML Document,” and Chapter 3, “XML Must Be Well-Formed,” either ensure that XML processors handle XML documents that satisfy XML’s syntax rules or signal an error indicating that the document isn’t suitable for further processing. For some purposes, that is sufficient.

For many purposes, particularly when documents are being shared among business partners, for example, it is useful for XML documents also to conform to a known, predictable structure. Of course, it is possible to write custom code in Java or some other programming language to make appropriate checks of a received document’s structure. However, it is potentially more convenient to check structure using the validation tools that form part of a validating XML processor.

In XML, a document type definition (DTD) defines the allowed structure of a class of XML documents. A validating XML processor can use the DTD to confirm that a document conforms to the relevant DTD. Using XML-based validation cuts down on the need to write custom code.



**Note** Two broad types of XML processor exist. A *nonvalidating XML processor* checks that documents conform to the rules of XML syntax, but it doesn't check for any specific structure of elements or attributes. A *validating XML processor* checks for well-formedness and also checks that the document conforms to a defined structure.

XML documents are ideal for sharing information using a standard syntax. The recipient of information expressed in XML might want to check that the structure of the information received corresponds to what he expects. If the structure of the received information can be checked automatically rather than by writing customized code, this is significantly more efficient.

## Document Structure Is Defined in a DTD

The only schema mechanism provided in the XML 1.0 Recommendation is the document type definition (DTD).



**Note** A *schema* is a document that defines the allowed structure, or its variants, of a *class* of XML documents. Schemas for XML 1.0 documents may be written in non-XML syntax, a DTD, or, more recently in XML syntax, using a variety of schema languages. For example, the W3C XML Schema is described in Chapter 19, “Beyond DTDs—W3C XML Schema.”



A DTD contains markup declarations in two subsets, the *internal subset* and the *external subset*. Taken together, these define the allowed structure of a class of XML documents.

Typically, a group of companies or other organizations with a common interest in a particular type of information will agree (after sometimes lengthy discussion) to automate important processes. That will generate a list of documents that they want to exchange and, in turn, agree on a common structure to be used for the exchange of certain types of business or technical information. The formal expression of the agreed-upon structure is a DTD or other type of schema.

## What Is a Valid XML Document?

An XML document that contains a DOCTYPE declaration and that complies with the constraints for that class of XML document expressed in the document type declaration is said to be *valid*. In other words, a valid XML document must comply with a defined logical structure.

All valid XML documents must be well-formed and must satisfy the well-formedness constraints described in Chapter 3. Some well-formed XML documents lack a DOCTYPE declaration and cannot be valid; others might have a DOCTYPE declaration but not comply fully with its constraints. Those documents are not valid XML, either.

## What a DTD Is

A DTD is a description of the allowed structure of a class of XML documents.

In a DTD, you *declare* elements, attributes, and so on that are allowed in the structure of a corresponding class of XML documents. Elements and other parts of an XML document are declared in *markup declarations*.

The following are the types of markup declaration in XML 1.0:

- Element declarations
- Attribute list declarations

- Entity declarations
- Notation declarations

Markup declarations may be contained in part or entirely within parameter entities.

## The DTD Is Not the DOCTYPE Declaration

Earlier, when defining a valid document, this chapter indicated that a valid document complies with the constraints expressed in the DOCTYPE declaration.

It is important to be clear about the differences between the document type *definition*, the DTD, and the document type *declaration* (also called the DOCTYPE declaration). They are not the same, although they are closely related.

The document type declaration contains or refers to information about the permitted structure of an XML document. In other words, the document type declaration may contain part of the DTD, refer to the location of part of the DTD, or do both. The DTD can be spread across two subsets: the *external subset* and the *internal subset*. Either subset may be empty in any particular situation, or both subsets may contain markup declarations. If markup declarations are present in the external subset and include default values for attributes, the `standalone` attribute in the XML declaration of an *instance XML document* must have the value `no`.



**Note** An *instance document* is a document that conforms to the defined structure for the class of XML documents to which it belongs.

## The External Subset

The external subset of the DTD typically exists as a separate file with a file extension of `.dtd`.

For example, this short XML document

```
<?xml version="1.0" ?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
<title>1984</title>
<author>George Orwell</author>
</book>
```

would have a DTD contained in a file, book.dtd, with the following structure:

```
<!ELEMENT book (title,author) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
```

That would be contained in the same directory as the XML document.



**Note** The character content of elements is termed PCDATA, meaning parsed character data. The character content of attributes is termed CDATA, meaning character data.

## The Internal Subset

The internal subset of the DTD is contained within the DOCTYPE declaration. The opening delimiter of the internal subset is the [ character, and the closing delimiter is the ] character.

To incorporate the markup declarations in an internal subset, the XML document would be rewritten as this:

```
<?xml version="1.0" ?>
<!DOCTYPE book [
<!ELEMENT book (title,author) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
]>
<book>
<title>1984</title>
<author>George Orwell</author>
</book>
```

The internal subset, if it exists for a particular document, is contained within the DOCTYPE declaration. The start of the internal subset is indicated by a square bracket character, [, and the end of the internal subset is indicated by the corresponding square bracket character, ]. The DOCTYPE declaration is then closed by a right angle bracket, >.

## Conditional Sections

A DTD may contain *conditional sections*. Conditional sections can be used to control when to use or ignore single markup declarations or a list of markup declarations.

Two options exist: INCLUDE (the default) and IGNORE:

```
<!INCLUDE[
oneOrMoreMarkupDeclarations

```

or

```
<!IGNORE[
oneOrMoreMarkupDeclarations

```

Conditional sections can be used with parameter entities, which are described more fully in Chapter 5, “XML Entities.”

## Declaring Elements in DTDs

Defining the structure of an XML document must always involve the declaration of elements because all XML documents contain at least one element: the document element.

An element declaration takes this general form:

```
<!ELEMENT elementType contentModel >
```

The allowed content models are listed here:

- Text only—Indicated by (#PCDATA). No element content is allowed.
- Empty element—Indicated by EMPTY. Not even whitespace is allowed as content.

- Any content—Indicated by ANY. Any well-formed content is allowed.
- Mixed content—Indicated by MIXED. This allows text content to be mixed with element content declared in the element declaration.
- Child elements—Indicated by one or more element names contained in parentheses, with any appropriate cardinality indicators.

Consider a simple XML document, such as the following:

```
<simpleMessage>
Here is a simple message.
</simpleMessage>
```

The following element declaration could be used to indicate that the `simpleMessage` element may contain only parsed character data, indicated by `#PCDATA` in the element declaration.

```
<!ELEMENT simpleMessage (#PCDATA) >
```

The `simpleMessage` element is constrained to contain parsed character data only. The presence of any elements in the content of the `simpleMessage` element would render that instance document invalid.

Typically, you would want to allow element content within a document element. To do so, you must declare the allowed elements. Consider an instance document with the following structure:

```
<book>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt</author>
<publisher>Sams Publishing</publisher>
</book>
```

You could indicate that a `book` element is allowed to contain a `title` element, an `author` element, and a `publisher` element, in that order, as follows:

```
<!ELEMENT book (title, author, publisher) >
```

You have not yet defined the allowed content of the `title`, `author`, and `publisher` elements—in this case, each has parsed character data. You would do so by completing the DTD as follows:

```
<!ELEMENT book (title, author, publisher) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT publisher (#PCDATA) >
```

The preceding DTD indicates that the `book` element is the document element. It may contain one and exactly one `title` element, followed by a single `author` element, then followed by a single `publisher` element.

In many settings, an element has more than one child element of a particular element type. Thus, you need ways to express the allowed frequency of occurrence—the cardinality—of child elements.

## Cardinality

In XML 1.0, the default cardinality is exactly one occurrence of, for example, an element. Therefore, the absence of any of the cardinality operators in the following list indicates that an element is allowed to occur exactly once.

In addition, the DTD can express three choices of cardinality that must be explicitly expressed within markup declarations:

- Optional, but may only occur once at most—Zero or one occurrences. This is indicated by the `?` character.
- Optional, but may occur many times—A minimum of zero occurrences and an unlimited maximum. This is indicated by the `*` character.
- Required, but may occur many times—A minimum of one occurrence and an unlimited maximum. This is indicated by the `+` character.



**Note** A DTD cannot express the notion that an element must occur, say, a minimum of 3 times and a maximum of 20 times.

For example, consider a customer order with a structure similar to the following:

```
<order>
<date>
2003/04/01
</date>
<customerID>
AB987
</customerID>
<items>
<item productID="1234" quantity="10">
3.5" floppy disks
</item>
<item productID="2345" quantity="20">
Write once CDROMs
</item>
</items>
<customerComment>I need the floppy disks as soon as possible.
</customerComment>
<customerComment>
Don't attempt delivery on a Friday.
</customerComment>
</order>
```

You could express that using the following markup declarations:

```
<!ELEMENT order (date, customerID, items, customerComment*) >
<!ELEMENT date (#PCDATA)>
<!ELEMENT customerID (#PCDATA) >
<!ELEMENT items (item)+ >
<!ELEMENT customerComment (#PCDATA) >
<!ELEMENT item (#PCDATA) >
<!ATTLIST item
 productID CDATA #REQUIRED
 quantity CDATA #REQUIRED>
```

In the first line of the code, the `date`, `customerID`, and `items` elements are declared without any cardinality operator. They are required and can occur only once. The `customerComment` element is declared with a `*` cardinality operator, indicating that it is optional but can occur more than once.

The declaration of the `items` element specifies that there must be at least one `item` element as its child, but that the `item` element may occur more than once.

## Declaring Attributes in DTDs

An XML element in a well-formed document may not have two attributes of the same name. Therefore, cardinality operators are not required when attributes are declared.

The declaration of attributes takes the following general form:

```
<!ATTLIST
 attributeName attributeType defaultDeclaration
>
```

The *attributeType* is any of the values in the following list:

- CDATA—Any legal XML string.
- ENTITY—Value that must match the name of an external unparsed entity.
- ENTITIES—An ENTITY, except that more than one whitespace-separated name may occur.
- ID—Value that must begin with a letter and then must consist of letters, numeric characters, hyphens, underscores, and period characters. At most, one attribute on any element can be of type ID. An ID attribute value must be unique in the XML document.
- IDREF—The value of the attribute must match the value of an ID attribute elsewhere in the same XML document.
- IDREFS—An IDREF, except that it may match more than one ID attribute value elsewhere in an XML document.
- NMTOKEN—The attribute value may contain only letters, numeric characters, and colons. No whitespace is allowed.
- NMTOKENS—An NMTOKEN, except that multiple values that do not contain whitespace are separated by whitespace characters.

The *defaultDeclaration* indicates whether a value is required, is optional, is fixed, or has a default value:



- #FIXED "*someValueInQuotes*"—The value of the attribute is fixed to the value given inside the quotation marks.
- #IMPLIED—A value for the attribute is optional.
- #REQUIRED—A value for the attribute is required.
- "*someValueInQuotes*"—A value for the attribute is optional. If no value is specified for the attribute in the XML document, the value "*someValueInQuotes*" is applied as a default.

For example, suppose that you have a very short XML document, as follows:

```
<order date="2002/11/30" orderNumber="NOV123"
customerID="DB998">
<items>
<!-- And so on -->
</items>
</order>
```

This example shows three attributes of the order element; for business purposes, each of these is essential. These are declared as a list of attributes associated with the order element. The ATTLIST keyword implies an attribute list, but you can define as few as one attribute.

```
<!ELEMENT order (items)>
<!-- An element declaration for the items element would go
 here -->
<!ATTLIST order
customerID CDATA #REQUIRED
date CDATA #REQUIRED
orderNumber CDATA #REQUIRED
>
```

## Types of Attributes

XML 1.0 DTDs provide very limited typing of attribute values, as discussed in the preceding section. For data-centric XML documents, this might be insufficient. This is one of the reasons behind the development of W3C XML Schema, described in Chapter 19.

## Specifying Default Attribute Values

You can specify default values for attributes when no value is given. For example, consider a business document structured as follows:

```
<Report>
<Paragraph status="public">Some text.</Paragraph>
<Paragraph status="confidential">Some confidential
 text</Paragraph>
<Paragraph>Some text.</Paragraph>
</Report>
```

To preserve confidentiality, you can make it essential that a human actually decide to make any information public by having the following attribute declaration:

```
<!ATTLIST Paragraph status "confidential" #REQUIRED>
```

If an author specifies `status="public"`, the default value is not applied. However, if the author overlooks the need to assign a `status` attribute to a `Paragraph` element, the `Paragraph` element's content is confidential until a human author overrides the default.

## Declaring Entities in the DTD

Entities may also be declared in a DTD. This is described in Chapter 5, where entities are discussed in more detail.

## Summary

The benefits of sharing XML documents with a predictable structure were discussed in this chapter. You also learned about the concept of a *valid* XML document and gained insight into the use of a Document Type Definition, internal subsets and external subsets, and the correct way to express *markup declarations* for the declaration of elements and attribute lists.

# LESSON 5

## XML Entities



*In this chapter, you will learn about XML entities, what they are, and how you can use them.*

### What Is an Entity?

An entity is an expression of the physical, rather than logical, structure of an XML document. An entity is a physical data object. When your XML documents are short and simple, you likely will seldom use entities other than the built-in entities. As you begin to create XML documents of greater length and complexity, the usefulness of entities will become more apparent.

One situation in which entities are sometimes used in relatively short documents is in Scalable Vector Graphics (SVG), an XML application language that you will meet in Chapter 15, “Presenting XML Graphically—SVG.” In SVG, for example, an entity can be used to define a particular style. If you had an entity called `BlackAndRed`, you could declare it like this:

```
<!ENTITY BlackAndRed "fill:black;stroke:red">
```

Then you could reuse the entity many times in attribute values in the document, like this:

```
<rect style="&BlackAndRed;" />
```

Even when creating the simplest XML documents, you are using at least one entity, although you might not be aware of it. Each XML document has at least one physical entity: the document entity.

An XML document can be viewed as being contained within the document entity. The document entity is not expressed within the syntax of an XML document; instead, it is the container for the syntax that makes up the document.



**Note** Most XML entities have a *name*, which is used to reference the entity. The exceptions are the document entity and the external subset of the DTD; these have no name, although both have filenames.

For example, the description of this book used in earlier examples has one logical structure but could be expressed by either of the physical structures shown in the following examples. The simplest expression of the logical structure exists in a single document entity with the description contained in one XML file with the following content, as shown in Listing 5.1.

#### **LISTING 5.1** SingleEntity.xml: A Description of This Book in a Single Document Entity

```
<?xml version="1.0" ?>
<book>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<author>Andrew Watt </author>
<publisher>Sams Publishing</publisher>
</book>
```

Alternatively, the same logical structure could be expressed in several different physical structures. One possibility is to use an external parsed entity to express title information, as in Listing 5.2. For a document as simple as this, there is little practical point in splitting it this way, but the example serves to illustrate the principle.

**LISTING 5.2** SplitEntities.xml: A Description of the Book with an External Parsed Entity

---

```
<?xml version="1.0" ?>
<!DOCTYPE book [
<!ENTITY bookTitle SYSTEM "title.xml">
]>
<book>
<title>&bookTitle;</title>
<author>Andrew Watt</author>
<publisher>Sams Publishing</publisher>
</book>
```

The file title.xml specified in the entity declaration is shown in Listing 5.3. In a typical external parsed entity in real-life use, the content would be much more extensive.

**LISTING 5.3** Title.xml: A Brief External Entity Referenced in Listing 5.2

---

Sams Teach Yourself XML in 10 Minutes

In Listing 5.2 the entity reference &bookTitle; is used by the XML processor together with the corresponding entity declaration to find the file Title.xml and to insert the content of that file between the start tag and end tag of the title element in Listing 5.2:

```
<!ENTITY bookTitle SYSTEM "title.xml">
```

So, after the external parsed entity has been retrieved, the title element is processed as if it read as follows:

```
<title>Sams Teach Yourself XML in 10 Minutes</title>
```



**Caution** If the parsed entity is defined in the external subset of the DTD, some nonvalidating XML parsers might not retrieve external entity declarations.

One use of external entities is to centralize frequently referenced information used by multiple files. In lengthy, complex XML documents, it can be very useful to split documents into entities. For example, a change made in an external parsed entity can be reflected at each place where the entity reference occurs in the XML document and other documents that reference the same external parsed entity.

You might structure financial results using separate XML files for each quarter's figures. For example, the sales figures for Quarter 1 2003 might be represented as follows:

```
<Q12003>
 <Total Sales>$74,300,000</TotalSales>
 <GrossProfit>$2,900,000</GrossProfit>
 <NetProfit>$1,500,000</NetProfit>
</Q12003>
```

If you stored that content in a file named Q12003.xml, you could reference that data in several places after declaring an entity:

```
<!ENTITY Q12003Sales system "Q12003.xml">
```

It makes sense to store the data once rather than risk it being stored in several places with inconsistent data. If that data was referenced several times, such as in department reports and company reports, it would make sense to store it once and then reference it each time it is used.

## Entities and Entity References

An entity is a data object. An *entity reference* refers to a parsed entity or parameter entity. The entity referenced may be either a *parsed entity* or a *parameter entity*. The syntax for referencing these two types of entities differs.



**Note** A parsed entity can be *internal*—declared in the document entity—or *external*—contained in a file (entity) physically separate from the document entity. A parameter entity is declared and referenced within the DTD, in either the internal or the external subset.

Parsed entities are referenced by an initial & character followed immediately by the entity's name and a semicolon. If you had declared an internal parsed entity called `myEntity`

```
<!ENTITY myEntity "This is my own entity">
```

you would reference it as follows:

```
&myEntity;
```

You can, of course, choose any name that makes sense in your context.

Parameter entities, which are used only in the DTD, use a different syntax, both in declaration

```
<!ENTITY % myParameterEntity "Class">
```

and in references to them:

```
%myParameterEntity;
```

Unparsed entities are referenced by names contained in attribute values declared to be of type `ENTITY` or `ENTITIES`.

## Predefined Entities

XML processors recognize a number of entity references as referring to five characters that have special meaning when used in XML documents. This means that a character being used literally in content can be distinguished from its use as part of markup. The five entity references and the characters that they represent are listed here:

- `amp`—Represents the ampersand character (&) in parsed character data
- `apos`—Represents the apostrophe (') in parsed character data
- `gt`—Represents the right angle bracket (>) in parsed character data
- `lt`—Represents the left angle bracket (<) in parsed character data
- `quot`—Represents a single double quotation mark (") in parsed character data

Let's consider parsed entities and parameter entities in more detail.

## Parsed Entities

In XML, an entity may contain parsed data or unparsed data. The term *parsed data* refers to data in XML syntax that is to be parsed—it doesn't mean that it has already been parsed. Similarly, the term *unparsed data* refers to data that is not intended to be parsed by an XML processor.



**Note** Parsed entities are also *general entities* because they are used within the XML document rather than in the DTD. An internal parsed entity and an internal general entity are the same thing.

Parsed data consists of characters, which may represent either character data or markup.

```
<name>John Smith</name>
```

The preceding line of code consists of characters. The characters John Smith are character data, and the start tag `<name>` and the end tag `</name>` make up the markup.

You can represent the preceding code in an entity declaration for either an internal parsed entity or an external parsed entity. An internal parsed entity would have an entity declaration that declares a value:

```
<!ENTITY myNameInfo "<name>John Smith</name>" >
```

Notice that it is permissible to use tags with literal `<` and `>` characters within the quotation marks that contain the replacement text for the parsed entity.

An external parsed entity would provide an external reference using an appropriate combination of the `SYSTEM` and `PUBLIC` keywords. For example, if the replacement text for the entity was contained in a file `myNameInfo.xml` in the same directory as the document entity, you could declare the external parsed entity as follows:

```
<!ENTITY myNameInfo SYSTEM "myNameInfo.xml">
```



The external file would contain the replacement text:

```
<name>John Smith</name>
```

## Internal Parsed Entities

Internal entities are defined in the same XML document (the same document entity) as they are used in.

For example, in a document about XML 1.0, you might have the following structure:

```
<?xml version="1.0" ?>
<!DOCTYPE document [
 <!ENTITY xml1.0 "Extensible Markup Language 1.0">
]>
<document>
 <description>
 In February 1998 &xml1.0; was finalized by the World Wide Web
 Consortium and released as a W3C Recommendation.
 </description>
</document>
```

The following entity declaration associates the entity name `xml1.0` with the text `Extensible Markup Language 1.0` as the *replacement text*:

```
<!ENTITY xml1.0 "Extensible Markup Language 1.0">
```



**Note** You may use the character sequence `XML` or `xml` to begin the name of an entity. You cannot use that character sequence to begin an element type name.

Then, in the content of the `description` element, the `xml1.0` entity is referenced.

The XML processor processes the description element as if it reads as follows:

```
<description>
In February 1998 Extensible Markup Language 1.0 was finalized
 by the World Wide Web Consortium and released as a W3C
 Recommendation.
</description>
```

One example of entities' use in short documents is in SVG documents to define style information for several elements. For example, you could use an entity to define a red stroke on text and a rectangle shape:

```
<?xml version="1.0" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
http://www.w3.org/Graphics/SVG/1.0/DTD/svg10.dtd[
<!ENTITY myRedStroke "fill:red">
]>
<svg>
<rect x="20" y="20" width="100" height="100"
 style="&myRedStroke" />
<text x="140" y="20" style="&myRedStroke">
This text is red
</text>
</svg>
```

Of course, typically, the style information in an SVG document is significantly longer to justify use of an entity in this way.

## External Parsed Entities

External parsed entities are contained in a file external to the document entity.

The simplest situation is one in which an external parsed entity is referenced using an entity reference in the document entity, as shown earlier.

However, an external parsed entity may contain any arbitrary well-formed XML, including entity references. That means that an external parsed entity may itself contain an entity reference to yet another external parsed entity. Of course, in principle, that external parsed entity could contain yet further entity references to more external parsed entities.

It is in situations such as the one just described that an entity may indirectly refer back to itself. If that occurs, the XML document, taken as a whole, is not well formed. Remember that recursion is not allowed (refer back to Chapter 3, “XML Must Be Well-Formed”), so an error is signaled.

## Text Declaration

An external parsed entity may optionally begin with a *text declaration*.

A full XML document can contain an XML declaration to indicate that the document is XML, to indicate the version of XML being used, and, optionally, to indicate whether the XML document is standalone and which character encoding is being used. In an external parsed entity, there is, by definition, no possibility that it is standalone. A text declaration, therefore, cannot have a standalone attribute.

A text declaration takes two forms. This first is this:

```
<?xml version="1.0" ?>
```

Here, the only attribute is the compulsory version attribute. An encoding attribute also is possible, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

## Unparsed Entities

Unparsed entities are used in attribute values and are referenced by name. If the name of an unparsed entity appears in an attribute value declared as being of type ENTITY or ENTITIES, a validating XML processor must pass both the system identifier and the public identifier for the unparsed entity to the application.

The content of an unparsed entity may be text or other types of data. If it is text, then it can be XML or some other format.

An unparsed entity is declared using the following format:

```
<!ENTITY entityName SYSTEM "myPicture.gif" NDATA gif>
```

The *entityName* is used in attribute values to reference the unparsed entity. The filename of the entity is given in quotation marks following the SYSTEM keyword. The keyword NDATA indicates that the data is non-XML data. In this example, the unparsed entity is identified as being in GIF format, by means of a gif notation declared elsewhere in the DTD.



**Note** A *notation* identifies by name the format of unparsed entities.

## Declaring Notations

Notations are declared using the following general format:

```
<!NOTATION notationName locationOfInformation >
```

The *notationName* specifies the name of the notation. The *locationOfInformation* variable uses the SYSTEM and/or PUBLIC keywords to identify a resource where the application can access further information about the non-XML data indicated by the notation. This enables the application to further process the non-XML data, either using its own facilities or by accessing a helper application.

## Types of Unparsed Data

Unparsed data might be an image, text, or binary data. The only responsibility of the XML processor is to pass to the application the name of the unparsed entity and the notation associated with it.

## Parameter Entities

Parameter entities are used only in DTDs. They allow reuse of information within the DTD.

Suppose that you had a document of the following structure:

```
<Reports>
<Report>
<Introduction>
<!-- Introductory text goes here. -->
</Introduction>
<!-- Main report information goes here. -->
<Comment>
This contains a comment about an individual report
</Comment>
</Report>
</Reports>
```

If you were declaring a single introduction element, you might write the declaration as follows:

```
<!ELEMENT introduction (#PCDATA)>
```

Similarly, if you were declaring the comment element, you might write this:

```
<!ELEMENT comment (#PCDATA)>
```

The literal text (`#PCDATA`) is being used more than once, so you could use a parameter entity to replace it.

Remember that parameter entities are not permitted in the internal subset of a DTD; you need two separate documents.

The modified XML document would look like this:

```
<?xml version='1.0'?>
<!DOCTYPE Names SYSTEM "myDTD.dtd">
<Reports>
<Report>
<Introduction>
<!-- Introductory text goes here. -->
</Introduction>
<!-- Main report information goes here. -->
<Comment>
This contains comment about an individual report
</Comment>
</Report>
</Reports>
```

The external subset of the DTD could look like this:

```
<!ENTITY % myPC "(#PCDATA)" >
<!ELEMENT Reports (Report)+ >
<!ELEMENT Report (Introduction, Comment)>
<!ELEMENT Introduction %myPC; >
<!ELEMENT Comment %myPC; >
```

The literal text `(#PCDATA)` has been defined as the replacement text for the `myPC` parameter entity, declared as follows:

```
<!ENTITY % myPC "(#PCDATA)" >
```

You can then use the `myPC` parameter entity in the entity declarations for the `Introduction` and `Comment` elements:

```
<!ELEMENT Introduction %myPC; >
<!ELEMENT Comment %myPC; >
```

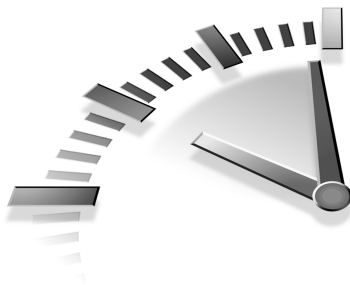
This example is very simple, but it indicates how a parameter entity can be used.

## Summary

In this lesson, you learned what an entity is and what types of entities are defined for XML 1.0. You also learned how to declare parsed entities, unparsed entities, and parameter entities. In addition, you saw examples of how entities can be used.

## LESSON 6

# Characters in XML



*In this lesson, you will learn about character sets and encodings and how they can be used with XML documents.*

## Internationalization

Only a few years ago, the World Wide Web was primarily an English-language medium, at least in the eyes of people from the United States and the United Kingdom. Of course, English is the native language of perhaps 10% of the world's population and is used in economically influential countries, with international trade, and particularly in countries where English is spoken natively. Support for English on the Web was obviously essential, but over several years it became increasingly obvious that support for many other languages was essential, too, if the Web is to be truly worldwide.



**Note** In this lesson, you will learn many terms relating to characters that have very specific meanings. If you are unfamiliar with this material, you might find the new terminology a little confusing. It will make sense if you work through it slowly.

Support for multiple languages raises many issues for people who think primarily or exclusively in English. For example, how do you express characters that cannot simply be typed using an English-language keyboard? For example, how do you express characters in German that use

the umlaut, such as ü, or letters in French that have an acute accent above a vowel, such as é?



**Note** The idea of internationalization is an important one in the XML world. Because *internationalization* is a long word, you will often see it abbreviated as i18n.

To understand how to solve this general problem, you need to consider how characters are encoded.

## Character Encodings

This section looks at how individual characters and sets of characters are encoded on a computer.

As mentioned earlier, English characters can be entered into computer memory most simply from the keyboard. However, at a fundamental level, computers understand only numbers. A character must be represented in some way as a number so that you can use text in a computer. A mapping from a set of numbers to a set of characters is stored internally.

English-language characters (plus some other characters) can be represented using the American Standard Coding for Information Interchange (ASCII) coding. This is an 8-bit coding system. All English characters can be represented using the 256 characters ( $2^8$ ) characters of ASCII. The hexadecimal number 21, for example, corresponds to the exclamation mark character (!).

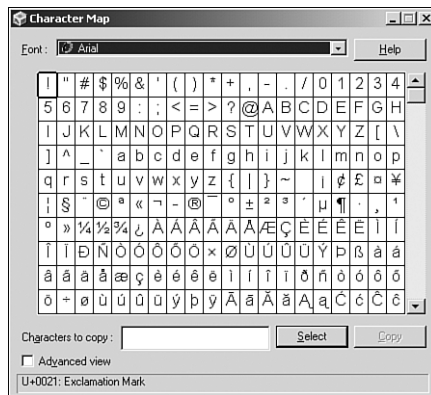


**Note** The characters up to hexadecimal 0020 don't display a visible character onscreen, but they might affect screen appearance. Character 0020 (decimal 32), for example, is the space character.



Let's look at ASCII characters in a little more detail. If you are using a computer that is running a recent version of Microsoft Windows, you will have access to the Character Map, which allows you to express the ASCII character set as well as many non-English characters. You will use that to begin to explore some issues related to characters, their representation, and their display.

In Windows 2000, to access the Character Map, choose the Start button, Programs, System Tools, Character Map. When you run the Character Map utility, you will see the window shown in Figure 6.1.



**FIGURE 6.1** The Character Map utility in Windows 2000.

In this figure, the font applied initially is the Arial font. So, all characters displayed use the Arial font unless the user chooses an alternative font. We will return to the discussion of fonts a little later.

First, hover the mouse over the exclamation mark (!) in the top-left corner of the Character Map program. You will see a ToolTip:

U+0021 - Exclamation Mark

The meaning of the final part of the ToolTip is obvious: It is simply the natural English term for the character indicated by the mouse.

The U+0021 corresponds to the hexadecimal number 21, (decimal 33) mentioned earlier. However, it is an abbreviation indicating that the Unicode system (explained in greater detail later in the chapter) is in use and that the particular character is 0021. The 0021 is expressed in hexadecimal notation and would be expressed as 0033 in decimal notation.

Listing 6.1 shows an XML representation of the exclamation mark in hexadecimal and decimal notation, as well the literal exclamation mark character.

---

**LISTING 6.1** Exclamations.xml: Expressing Characters in Hexadecimal and Decimal

---

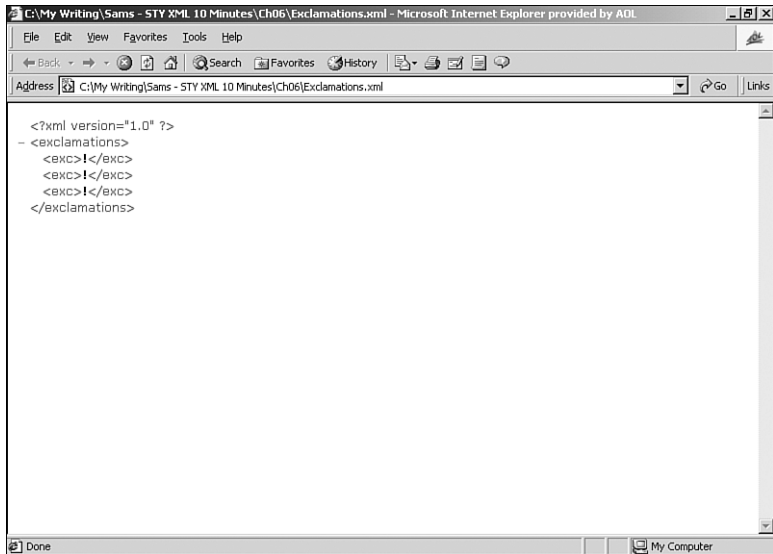
```
<?xml version='1.0'?>
<exclamations>
<exc>!</exc>
<exc>!</exc>
<exc>!</exc>
</exclamations>
```

If you run the code and display it in the Internet Explorer 5.5 browser, you will see something similar to Figure 6.2. Any other XML-compatible browser should give a similar appearance. Both *character references* display as an exclamation mark contained within the exc element, just like the literal exclamation mark in the third exc element.

Of course, what you have done so far you could have done more easily using an English-language keyboard. When you need to include characters that either are not available from an English language keyboard or can be achieved only using obscure key combinations, the Character Map utility becomes more helpful.

Suppose that you want to express the simple idea in German, “Übung macht den Meister” (meaning, “Practice makes perfect”). You need the *u* with the umlaut (also called a diaeresis). This can be copied from the Character Map by clicking the desired character and clicking the Copy button.

This enables you to include foreign-language characters that use the Roman alphabet in English-language documents. If you scroll down in the Character Map utility, you will see characters used in such languages as Russian, Hebrew, and Arabic.



**FIGURE 6.2** Character references.

So, the type of facility supplied by the Character Map provides an answer sufficient for expressing many European and some Middle East languages. Of course, many languages cannot be expressed using the Character Map. For example, some Asian languages, such as Chinese, Japanese, and Korean, use many thousands of ideographic characters. An ideographic character represents a word or idea rather than an individual letter. A more general solution to supplying a desired character is needed to accommodate any written language.

## XML and Internationalization

From the beginning, XML was intended to be used internationally. To support authentic international use, XML must be capable of expressing character sets used in all (or at least the most widely used) world languages. To achieve this international usage and express the ideas and words of many languages, XML has built on several other agreed approaches for expressing characters.



**Note** Many of the issues discussed in this chapter are important if you plan to use more than one language in your XML documents.

The 1-byte (8-bit) approach of the ASCII code is insufficient for expressing more than a few languages. Increasing the encoding to 2 bytes enables users to express far more characters (65,536 instead of 256).

Having 2-byte character codes opens up the possibility for many more codes than necessary to encode the characters of English and other languages that use the Roman alphabet. At the present time, the most widely used internationally accepted standard uses 16-bit encoding. It doesn't achieve representation of every single character that might be needed, but it does go further toward a truly global solution for character encoding.

All XML processors are required to support two particular character forms: UTF-8 and UTF-16. These are 8-bit and 16-bit forms of character encoding, respectively. In hexadecimal terms, these are numeric values from 0000 to FFFF inclusive, with minor gaps because some codes are used for other purposes.

The section "Unicode" looks a little more closely at these international codes. First, the next section looks again at how the encodings for individual characters can be expressed in a way that an XML processor will understand.

## Character References

*Character references* are the XML technique for directly expressing the numeric character encoding of a character.



**Note** A *character reference* is a technique for using a numerical value, which can be expressed in hexadecimal or decimal notation, to refer to an individual character.

In XML, character references are used. To express a character reference using hexadecimal notation, use the character sequence `&#x` followed by four numeric characters or the characters A (decimal 10) to F (decimal 15) inclusive to express any character that can be expressed using 16-bit encoding. The end of the character reference is signaled by the semicolon (`;`).

Decimal numbers are expressed as `&#` followed by numbers up to 65,535 that are followed by a closing semicolon character.

An example using both hexadecimal and decimal notations is shown in Listing 6.2.

### **LISTING 6.2** Good.xml: A Statement in English and German

```
<?xml version='1.0'?>
<Good>
 <InEnglish>That is good.</InEnglish>
 <Deutsch>Das ist güt.</Deutsch>
 <Deutsch>Das ist güt.</Deutsch>
</Good>
```

The desired character in German is represented numerically as hexadecimal `00FC`.

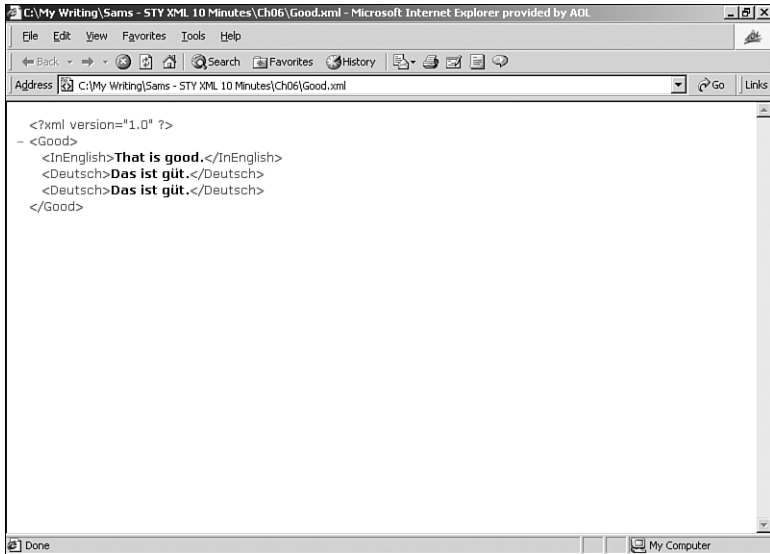
If the code is displayed in the Internet Explorer browser, the desired character appears similar to Figure 6.3.

The character code `00FC` is a 2-byte code.



**Note** Strictly, a byte need not be 8 bits in size. Because many familiar computer systems use an 8-bit byte, this code is referred to as 2-byte code. An alternative description is a 16-bit character code.

If you open the document shown in Listing 6.3 in your browser, you can explore the onscreen appearance of the full number of 16-bit characters by replacing the 16-bit number. Unfortunately, your computer probably will not have the necessary information to display many of the possible characters.



**FIGURE 6.3** Displaying a foreign-language character using a character reference.

### **LISTING 6.3** Explore.xml: Explore the 16-Bit Character Encoding

```
<?xml version='1.0' ?>
<Explore>
 <aCharacter>﫼</aCharacter>
</Explore>
```

For many of the characters, you will find that a browser simply displays a square or other placeholder because your computer does not have a font to display the needed character.

## Unicode

This section looks a little more closely at the character set that XML uses: Unicode.



**Note** Strictly speaking, XML uses the international Standard ISO/IEC 10646. Unfortunately, the International Organization for Standardization (ISO) does not make its standards freely available on the Web in the way that the World Wide Consortium (W3C) does. The Unicode encoding follows ISO/IEC 10646 and is accessible via the Web.

The Unicode organization created a character encoding that has global acceptance. Before the emergence of Unicode, many encoding schemes existed that, for many practical purposes, were incompatible. There is, of course, no intrinsic reason why a particular character—the exclamation mark, for example—should be represented by a particular number. Therefore, the encoding is arbitrary, to a degree. The important thing is that everybody agrees to use the same encoding.

Of course, it made sense for the English encoding to be compatible with the long-standing (in computer terms) ASCII encoding. So, English is expressed in Unicode codes beginning with a double zero, such as representing the exclamation mark with hexadecimal 0021.



**Note** Unicode is also used by many modern programming languages, including Java and ECMAScript (JavaScript). An understanding of at least the basics of Unicode is useful for the Web developer.

Unicode uses an initially intimidating technical vocabulary. See <http://www.unicode.org/glossary/> for further details.

## Unicode Supplementary Code Points

The latest versions of Unicode go beyond the hexadecimal range of 0000 to FFFF. Also included are *supplementary code points* in the range 100000

to 10FFFF. This provides for the encoding of additional characters that cannot be encoded in the 16-bit encoding.

Thus extended, Unicode allows the encoding of more than one million characters, which is anticipated to accommodate all of the world's characters. Of course, most of the most commonly used characters are included in the 16-bit encoding scheme.

The first 65,000 or so characters are referred to as the Basic Multilingual Plane—BMP, for short.

## Unicode Encoding Forms

Unicode provides three *encoding forms*.

- UTF-8—An 8-bit encoding form. This must be supported by all XML processors. UTF-8 maps one to one with ASCII.
- UTF-16—A 16-bit encoding form. This must be supported by all conforming XML processors.
- UTF-32—A 32-bit encoding form. XML 1.0 processors are not obligated to support this encoding form.

## Fonts, Characters, and Glyphs

This section examines the meaning of the terms *font*, *character*, and *glyph*.

A Unicode character point is a numerical representation of a conceptual character. For example, suppose that you refer to “uppercase A.” If you are familiar with English, you know which character is being referred to. However, you can't say with certainty exactly how this conceptual character will be displayed onscreen or on paper. This is where glyphs and fonts come in.

For the purposes of this discussion, a *font* is a set of glyphs. A *glyph* is a particular visual representation of a character. Any character, such as the uppercase A, can be displayed in any number of visual appearances.



This truth is illustrated using Scalable Vector Graphics (SVG). Listing 6.4 shows a simple SVG document with elements that contain the uppercase character sequence XML, displayed in several different fonts. To run and view the code, you will need an SVG viewer, such as the Adobe SVG Viewer ([www.adobe.com/svg/](http://www.adobe.com/svg/)).

---

**LISTING 6.4** myXML.svg: Displaying the Same Characters As Different Glyphs Specified by Different Fonts

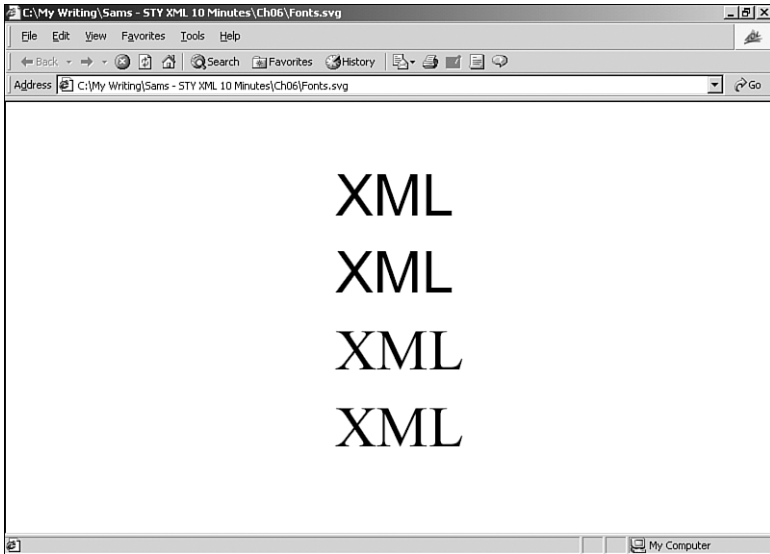
---

```
<?xml version='1.0'?>
<svg>
<text x="20" y="30" style="font-size:30; font-
 family:Arial;">XML</text>
<text x="20" y="70" style="font-size:30; font-family:Arial;">
 XML</text>
<text x="20" y="110" style="font-size:30; font-family:'Times
 New Roman';">
XML</text>
<text x="20" y="150" style="font-size:30; font-family:'Times
 New Roman';">XML</text>
</svg>
```

Notice the difference between the second and third lines of characters. The glyphs that are displayed in the second line form part of the Arial font. Glyphs in the Arial font share visual features in common. An important one is that they all lack a *serif*, those little marks at the ends of strokes on some characters. The same characters, XML, are represented by different glyphs when the Times New Roman font is used, as in the third line in Figure 6.4. The glyphs in that font commonly possess a serif on many letters. Compare the exact shape of the characters XML as displayed in Figure 6.4 to see the differences.

Whether the XML is specified as literal characters or as character references, the displayed characters are the same if the font is the same. You can see this in the first two myXML elements shown in Figure 6.4.

The range of glyphs available for English-language characters is enormous, in part expressing the creativity of font designers. For XML application languages, the character point is conceptually the same regardless of the visual appearance chosen to display it.



**FIGURE 6.4** Literal characters or character references yield the same result.

## Summary

In this lesson, you learned about the need for internationalization and how XML supports it. Several technical terms, including *character*, *font*, and *glyph*, were introduced and explained. The concepts of character encoding, character set, Unicode, and character references also were discussed, along with examples.

## LESSON 7

# The Logic Hidden in XML



*In this chapter, you will learn how data can be modeled using XML and how the W3C models of logical structure behind well-formed or valid XML can be represented and manipulated.*

## Modeling Data As XML

Earlier chapters in this book described XML mostly in terms of a set of syntax rules that define how sequences of characters are to be used so that an XML processor can process an XML document without throwing errors and that also define the physical structure, expressed as entities. But XML documents also have a logical structure that is expressed by the nesting of elements and the presence of attributes on selected elements. This highly flexible document structure means that you can model many types of data. This is useful for modeling both highly regular structured data that might otherwise be stored in a relational database and highly flexible narrative documents.

First, let's look at issues relating to markup languages and see why they are increasingly used in place of or alongside binary file formats.

## Binary Files and Markup

Binary files play an important role in data storage because they are—or can be—very compact for any given amount of data to be stored. When RAM was measured in kilobytes rather than hundreds of megabytes and when disk storage was measured in megabytes rather than tens of gigabytes, compactness of storage was crucially important to developers of

software for personal computers. As quantities of RAM and hard disk storage have multiplied in recent years, compactness of data has become less important for many uses. The increase in the availability of RAM and hard disk space has outpaced the availability of developer time to create programs to handle data.

Many more types of data are being stored than were in use a decade or so ago. Typically binary files are specific to a particular application or vendor; programs to make them comprehensible to humans with onscreen displays or paper reports are becoming progressively more complex. Thus, new types of data storage must be easy to create and maintain, not just be compact. Markup languages in general, and XML in particular, play a part in the process of making data easier to structure and describe. Clear data structures that can be easily modified or adapted, together with processors that access the data contained in those structures, make it easier to create, maintain, and modify data storage than when using binary files alone.

One of the best known attempts to use markup languages to store or display structured data is the Hypertext Markup Language (HTML).

## XML More Structured Than HTML

HTML was initially designed to provide structured documents. HTML has enjoyed enormous success as the language used by Web browsers to display information, but it has fundamental limitations with structuring data. Look at the short employment record expressed in the HTML code in Listing 7.1.

---

### **LISTING 7.1** An HTML Representation of Structured Data

---

```
<html>
<body>
<h1>Peter Jones</h1>
<h2>Basic data</h2>
<p>1958/10/29</p>
<p>ABC123</p>
<h2>Employee History</h2>
<p>Peter Jones was first employed by the company on January
 21st 1980.
```

**LISTING 7.1** Continued

---

```
He has performed well since then and has been promoted four
times.</p>
</body>
</html>
```

An `h1` element is used to provide a heading for the employee data that follows. The `h1` element has a hybrid function: It communicates something about how its content will be displayed and also indicates something (rather vaguely) about the nature of the content in the overall structure of the document. An `h1` element indicates a heading, but an `h1` element in one HTML document might have a different importance or meaning than an `h1` element in another HTML document in the same subject domain.

The fact that an `h1` element indicates a header is useful up to a point, but you learn nothing about what an `h1` header *means*. Nor does the `h1` element—or other HTML header elements—actually contain all the data related to that heading. For example, the `h2` element in this example does not actually contain the `p` element that describes Peter Jones's employment history. The content of the `h2` element and the following `p` element are logically related but are structurally only loosely related.

If the HTML document contained four employee records, you have to depend on the good sense of the Web page author for any consistency in structure. You might use an `h1` element to indicate that multiple records are involved but that the `h1` element is only a descriptor, not a container element.

Taken together, these characteristics of HTML mean that it can structure data, but this structure is loosely expressed and is mixed with presentation data. For simple data, that might be acceptable. However, as the complexity of data and its volume increases—and as the prospect of more automatic processing of data appears on the horizon—the fuzzy edges of the structure that HTML provides become substantive disadvantages.

XML provides a cleaner, more consistent framework for expressing structured data. In part, this improvement relies on the fact that you can create meaningful element type names for each element in an XML document. In addition, by appropriately nesting XML elements, you can express more clearly the logical relationships among elements.

An XML document broadly equivalent to Listing 7.1 might look something like Listing 7.2.

---

**LISTING 7.2** An XML Representation of Employee Data

---

```
<employee>
<name>Peter Jones</name>
<dateOfBirth>1958/10/29</dateOfBirth>
<employeeID>ABC123</employeeID>
<employeeHistory>
Peter Jones was first employed by the company on January 21st
1980. He has performed well since then and has been promoted
four times.
</employeeHistory>
</employee>
```

An enormous advantage of the XML approach is that there is a container element—in this case, the `employee` element—that contains all the data in an XML document. Everything nested within the `employee` element is logically related to that `employee` element. This is a structure that more closely approximates reality.

Assuming that the element type name of the document element in an XML document is chosen wisely, this also hints at what the data is about. For example, when you see an `employees` element, you can guess that the content likely relates to data about a group of employees. An `h1` element type name can't communicate that—at least, not in the element name.

When element type names are chosen sensibly, XML can be termed self-describing data. An `employee` element tells much more about the data contained in it than, for example, an `html` or `body` element in an HTML document. In addition to creating new element type names, you can create new attributes to express information about individual elements for particular purposes.

Similarly, each component part of the `employee` element—whether attributes or children or descendant elements—can be viewed as further describing the `employee` element. In this respect, the content of an XML element has similarities to the properties of a programming object. Just as encapsulation is useful in programming languages, nesting of elements is useful for data description.

Let's move on to look at how you can use XML to describe various broad types of data.

## Modeling Relational-Type Data

XML allows the modeling of data that would conventionally be stored in a relational database-management system.



**Note** In some systems, data is served as XML but is stored in a relational database-management system (RDBMS) because of the highly optimized storage provided.

Relational data is typically expressed as rows (records) and columns (fields). A simple RDBMS table might contain employee records. Each record might consist of an employee's ID number, surname, first name, initials, and date of birth. This can be easily modeled in XML. Listing 7.3 shows one approach.

### LISTING 7.3 Modeling an Employee Record in XML

```
<?xml version='1.0'?>
<employees>
<employee>
 <employeeID>ABC123</employeeID>
 <surname>Cameron</surname>
 <firstName>Ewen</firstName>
 <initials></initials>
 <dateOfBirth>1975/12/28</dateOfBirth>
</employee>
<!-- Other "records" can go here -->
</employees>
```

XML provides flexibility in how the logical relationships are expressed. For example, you can use attributes to express information that might be contained in a child element. For example, you could use an attribute to express the employee ID, as in Listing 7.4.

**LISTING 7.4** An Alternate Structure for Representing the Data of Listing 7.3

---

```
<?xml version='1.0'?>
<employees>
<employee employeeID='ABC123'>
 <surname>Cameron</surname>
 <firstName>Ewen</firstName>
 <initials></initials>
 <dateOfBirth>1975/12/28</dateOfBirth>
</employee>
<!-- Other "records" can go here -->
</employees>
```

Thus, the same information has multiple possible representations in XML. The flexibility of XML creates potential problems because one logical structure can be expressed in more than one way. As you will see in Chapter 11, “XSLT—Transforming XML Structure,” you can use the Extensible Stylesheet Language Transformations (XSLT) to transform XML data from one equivalent structure to another (as well as perform other transformation tasks described in Chapter 10, “XSLT—Creating HTML from XML,” and Chapter 12, “XSLT—Sorting XML”).

In the approach shown in Listing 7.3 and Listing 7.4, the XML document corresponds broadly to an Employees table in an RDBMS system. Each employee element holds a record, in RDBMS terms. Each column is represented by an XML element, such as the employeeID element.

You can even express a single row as follows:

```
<employee
employeeID="ABC123"
surname="Cameron"
firstName="Ewen"
initials=""
dateOfBirth="1975/12/28"
>
```

Using only attributes confines you to the equivalent of a row structure of an RDBMS approach.



When the data naturally follows a relational structure, ultimate storage may best be in a proprietary format.

## Hierarchical Data in XML

Some types of data do not follow the regular structure of rows and columns that occur in RDBMS tables. For example, you could express a date store for the support calls from a customer as follows:

```
<customer customerID="DCE789">
 <call date="2002/08/30">
 <subject>Program crashes when started</subject>
 <supportOperator>Jim</supportOperator>
 <status>Resolved</status>
 </call>
 <call date="2002/09/25">
 <subject>Was unable to locate command to print.</subject>
 <supportOperator>Karen</supportOperator>
 <status>Resolved</status>
 </call>
</customer>
```

This support record for a single customer has a hierarchy—each customer element has nested inside it one or more call elements. In turn, each call element has three child elements—subject element, support element, and status element.

In data structures that model hierarchical relationships, XML can cope with hierarchies of arbitrary complexity. As hierarchies in data increase in depth, it becomes increasingly problematic for an RDBMS to model the data.

## Loosely Structured Data in XML

XML is derived from the Standard Generalized Markup Language (SGML). SGML often is used for complex documents, such as aircraft maintenance manuals. Because SGML is useful for expressing complex documents, it is not surprising that XML is also used to express documents of significant complexity.



**Note** All XML files are usually referred to as “documents.” XML documents are often referred to as document-centric and data-centric. The former describes simple letters or other correspondence, as well as lengthy documents that would take up multiple volumes if printed out. Data-centric XML “documents” use XML to store data that conventionally would be stored, for example, in an RDBMS or a hierarchical database.

Some lengthy document-centric XML documents are very rigidly structured, although they are documents rather than “data.” On the other hand, XML can be used to contain documents that form a very loose or flexible structure that might build on this sort of form:

```
<book>
<introduction>
<!-- Text for the introduction goes here, perhaps in multiple
 <section> or <paragraph> elements -- >
</introduction>
<chapter number="1">
<!-- In more highly structured documents, we may also have
 <section> elements -- >
<paragraph>Some paragraph text</paragraph>
<paragraph>Some more paragraph text</paragraph>
</chapter>
<!-- An appendix or several might go here. -->
<!-- An index, with many variant structures might be added
 here. -->
</book>
```

The structure shown is simple and clear, but many real-life uses are much more complex. In some cases, you might add elements such as header, subheading, footnote, sidebar, and other possible elements that convey the great diversity of structures used in books. XML can express all these structures.

As you have seen, XML can express the logical relationships contained in relational-type data, hierarchical data, and loosely structured data. For certain uses, non-XML approaches might be more efficient or appropriate,

but the sheer flexibility of XML makes it an important technology to master if you are handling anything but small amounts of data.

The final sections of this chapter briefly look at how XML data can be accessed and manipulated programmatically. Some of the topics introduced, including the Document Object Model and XPath, are discussed later in the book.

## W3C XML Data Models

XML documents follow the syntax rules of well-formedness but also represent *data objects* that have a hierarchical structure. The hierarchical structure must exist for each XML document, given that there is a document entity within which the prolog (if it exists), the document element, and all other elements exist. The simplest hierarchy for a well-formed XML document is a document entity with a single document element contained inside it.

The World Wide Web Consortium has produced three families of specifications that express, in different ways, the logical structure of an XML document: the Document Object Model, the XML Path Language, and the XML Information Set.

### The Document Object Model

The Document Object Model (DOM) for XML documents is, at its simplest, closely related to the DOM for HTML documents. The W3C has created specifications that express increasing functionality as subsequent *levels* of the DOM. The development of the DOM is ongoing—Level 1 and Level 2 have been released as full W3C specifications. DOM Level 3 is under development at the time of this writing.

As its name suggests, the DOM *models* an XML *document* in terms of *objects*. Technically, the DOM specifications govern *interfaces*. Interfaces can be viewed as contracts with an object. It doesn't matter what the exact structure of an object is. However, if it is a DOM object, it must behave as if particular properties and methods exist for the object.

The DOM enables developers to manipulate the in-memory representation of an XML document, with each element and attribute represented as a node in the in-memory hierarchy. The fundamental interface in the DOM is the `Node` interface. The nodes for elements, attributes, and so on extend the `Node` interface. You can add nodes that represent elements, attributes, and so on. The in-memory representation can be discarded after the application has performed any necessary actions; it also can be explicitly saved to a new file or can alter the file that was loaded.

The Document Object Model is discussed in more detail in Chapter 16, “The Document Object Model,” and Chapter 17, “The Document Object Model, 2.”

## XPath

The XML Path Language (XPath) also models an XML document as a set of nodes. However, the hierarchy of nodes in the XPath representation of an XML document differs in several respects from the DOM representation.

XPath uses a path syntax to express the hierarchy of the content in an XML document. XPath bears similarities to the *paths*—hence the name XML Path Language—used to express the hierarchy of the file system on your computer.

The *root node* of an XPath document, which is equivalent to the document entity, is expressed as a single forward slash character (/). Nodes can then be accessed relative to the root node. For example, the document element of a document with a document element called `myDocumentElement` can be accessed using path syntax, `/myDocumentElement`.

XPath is intended for use with other XML specifications, such as XSLT and XForms (a new forms language expressed in XML). XPath specifies which part of an XML document is to be processed. For example, XPath can be used with XSLT to select a set of nodes that, not surprisingly, it terms a *node-set*. The node-set is then processed in one of the ways supported by the XSLT specification.

XPath is discussed in greater detail in Chapter 9, “The XML Path Language—XPath.”

## The XML Information Set

The XML Information set, also termed the infoset, is an abstract data model for most of an XML document and is intended to be used by other XML-related specifications from the W3C.

An information item is similar to a node as used by DOM and XPath. However, the detail of how an XML document is modeled differs from both DOM and XPath.

The XML information set is the newest of the data models specified by the W3C but is viewed as the foundation for many future XML-related specifications produced by the W3C. For example, the infoset is used as a basis for the W3C XML Schema language, together with some augmentation of that infoset to express notions specific to the activity of schema validation.

## Which Data Model?

The fact that the W3C has released three data models that model the same thing—an XML document—can seem confusing. In practice, the choice of data model to use depends on what you want to do.

The DOM is used when you want to create, manipulate, or delete parts of an XML document using programming languages such as Java. XPath is used primarily to navigate the in-memory representation of an XML document, to make selected parts of that document available to other XML-based application languages such as XSLT, the XML Pointer Language, and XForms.

The information set is used in W3C XML Schema and is proposed as the basis for the XML Query Language (XQuery) and version 2.0 of XSLT and XPath (currently under active development at the W3C).

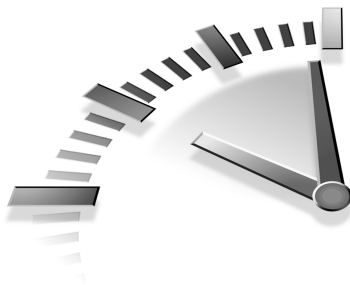
## Summary

XML can express, or model, many types of data structures, including structures that are similar to relational data, hierarchical data, and loosely structured data.

Data models provide the way for programming languages to efficiently and conveniently create, modify, or manipulate XML data. Three data models were introduced in this chapter—the Document Object Model, the XML Path Language, and the XML Information Set.

## LESSON 8

# Namespaces in XML



*In this lesson, you will learn the reasons for using namespaces in XML documents and the correct syntax for XML namespaces.*

## What Is a Namespace, and Why Do You Need Them?

XML is useful for exchange of documents. A finite number of element type names is available to use to contain the document content. In a global Web, how do we handle the possibility of two element type names being the same? The solution that the W3C chose is namespaces.



**Note** A *namespace* is a collection of names. In XML, a namespace refers to a collection of element type names and attribute names.

So, exactly why do we need namespaces? You might have document fragments such as the following:

```
<html>
<head>
<title>My XHTML document</title>
</head>
<body>Some content.</body>
</html>
```

and

```
<persons>
<person>
<title>President</title>
<firstName>George</firstName>
<lastName>Bush</lastName>
</person>
</persons>
```

Even with simple documents that mix these two XML structures, there is a problem. In this example, how do you distinguish unambiguously the title element that belongs to XHTML from the title element in the persons data store? When you mix longer documents, possibly using more than two vocabularies, the potential for problems is greater.

Three important problems almost inevitably might arise:

- It is difficult to recognize the application to be used for processing particular elements.
- Elements with the same element type name are used for different real-world meanings.
- Different element type names are used by different users or groups of users to represent the same real-world notion.

XML namespaces were designed to overcome the first two problems. A solution to the third problem using XSLT is described in Chapter 11, “XSLT—Transforming XML Structure.”



## Recognizing Which Application to Use

In the preceding code, how do you signal that a Web browser should process the `title` element from the XHTML code snippet and that another application should process the `title` element from the persons data store?

The first code snippet could be nested within the second, or vice versa. So, you need a mechanism that solves the problem, regardless of which essentially infinite number of structures is used. You need to be able to identify the individual elements as belonging to a particular namespace.

## Element Type Name Clashes

Element type names clash when two or more XML document authors use the same element type name to represent different real-world ideas or values.

Because the community using XML is already large and will grow further, the possibility that element type names will clash in shared documents becomes very real. The likelihood of element name clashes also increases as more XML documents are shared outside defined groups.

Consider the possibility of sharing information from several sources, each of which uses an `order` element:

```
<order>
 <orderNumber>AB123</orderNumber>
 <Items>
 <Item number="20">Ink cartridges</Item>
 <Item number="2">3.5" Floppy Disks (Box of 10)</Item>
 </Items>
</order>
```

The `order` element has a different meaning in the following code:

```
<order>
 <givenBy>General B. Smart</givenBy>
 <Content>Relocate 6 jet fighters to MacDill Air Force
 Base</Content>
</order>
```

It is different again in the following code:

```
<order>
 <Location>Seattle</Location>
 <Description>A riot lasted several hours following clashes
 with anti-globalization protesters.
</order>
```



**Note** These examples illustrate the problem, but more subtle issues also arise. Suppose that several companies each use an order element to refer to the placing of an order for goods and services. Those companies are not obligated to use an order element with the same structure of child and descendant elements nested within it.

If these XML documents remained within commercial, military, or police organizations, possibly no confusion would arise. However, suppose that the police or the Air Force ordered a number of items from the supplier that uses the order element in the first of the three ways shown. How is the reader—and, more importantly, an XML processor—to know which order element is intended in any particular context?

Clearly, a mechanism is needed to distinguish a commercial order from a military order or an order recorded by a police department. You need to be able to express in XML similar distinctions.

In XML, elements are distinguished by using *namespaces*.



**Note** A namespace is simply a collection of names. In XML 1.0, a namespace doesn't imply any particular document structure. It simply expresses the notion that these elements belong together.

A concept similar to XML namespaces exists in some programming languages. For example, in Java, a particular class must have a unique name, to avoid ambiguity. In Java, a *package* provides a broad equivalent to the concept of XML namespaces. A class must have a name that is unique within a package. Classes in other packages might have the same class name, but because they are in a different package, there is no risk of confusion as the code is processed.

As in Java, each XML element in an XML namespace must have a unique name; otherwise, confusion can arise.

Let's move on to examine exactly how XML distinguishes namespaces.

## Using Namespaces in XML

To clearly distinguish the order element here you would need to provide more information about the element type name than simply order:

```
<order>
 <givenBy>General B. Smart</givenBy>
 <Content>Relocate 6 jet fighters to MacDill Air Force
 Base</Content>
</order>
```

from the order element here

```
<order>
 <Location>Seattle</Location>
 <Description>A riot lasted several hours following clashes
 with anti-globalization protesters.
</order>
```

The solution to this, in XML 1.0, is a *qualified name*. This is often abbreviated as QName.



A *qualified name* is an XML element type name that consists of two parts—a *namespace prefix* and a *local part* separated by the colon character (:).

## Qualified Names

You can distinguish elements using qualified names (QNames). Referring back to the earlier example, the military version of an order could use the QName `mil:order`. This would distinguish it from possible QNames for the other two order element types, perhaps expressed as `business:order` and `civil:order`.

Each QName consists of a namespace prefix, a colon character, and a local part. The local part is what we have called the element type name in a non-namespace-aware document.



**Note** The colon character can legally be used for any purpose in XML. To avoid confusion and unpredictable results, it is wise to reserve the use of the colon character as a separator in QNames only.

If you use QNames on their own, you might run into problems similar to those you are trying to avoid. For example, the `mil:order` element might itself lead to ambiguity. Are you referring to military orders or militia orders? Similarly, if you refer to `business:order` elements, which of potentially many element types created by individual businesses or consortia are you referring to?

You need a more universal way to distinguish namespaces than simply using namespace prefixes alone. You achieve potentially unique identification of a namespace using a uniform resource identifier and mapping a namespace prefix to it.

## URIs Represent Namespaces

In XML namespaces, the *namespace name* is a uniform resource identifier (URI).

A URI is a potentially lengthy sequence of characters. For example, you might want to create a document type for a particular structure of document. You could choose a namespace URI `http://www.XMML.com/myVeryOwnNamespace`.

So, why aren't URIs used directly in QNames? Three reasons are worth mentioning:

First, URIs often make use of the colon character as a separator. With the following start tag, ambiguity exists regarding which of the two colon characters is the separator between the namespace prefix and the local part:

```
<http://www.XMML.com/myVeryOwnNamespace:document>
```

Does this refer to an element whose namespace prefix is `http` and local part is `//www.XMML.com/myVeryOwnNamespace:document`, or does this refer to an element whose namespace prefix is `http://www.XMML.com/myVeryOwnNamespace` and local part is `document`? For the human reader, it is pretty obvious that the second possibility is much more likely; for an XML processor, however, serious ambiguity arises.

Second, if you write elements using the literal URI throughout the XML document, it would soon become difficult for a human reader to decipher what the document is about. This is particularly true if lines must be broken to squeeze the lengthy URI onto the page:

```
<http://www.XMML.com/myVeryOwnNamespace:document>
 <http://www.XMML.com/myVeryOwnNamespace:introduction>
 Some content goes here.
 </http://www.XMML.com/myVeryOwnNamespace:introduction>
<!-- Many more lengthy elements could go here. -->
</http://www.XMML.com/myVeryOwnNamespace:document>
```

Third, URIs can contain characters that are not allowed in XML names.

The problem is solved by using a succinct namespace prefix that complies with the rules for XML names and that is mapped to a namespace URI. Declaring a namespace involves associating the namespace prefix with the namespace URI. In XML, you do this using a special type of attribute called a *namespace declaration*.

## Namespace Declarations

To be used in an XML document, a namespace must be declared. A namespace declaration is made in the start tag of the element to which it refers.

A namespace declaration has a special structure. For most namespace declarations, the attribute name begins with the character sequence `xmlns`, followed by a colon and the namespace prefix. These are followed by an equal sign (=) and the namespace URI enclosed in a pair of double or single quotation marks:

```
xmlns:namespacePrefix='namespaceURI'
```



**Note** XML names that begin with the character sequence XML (in any case combination) are reserved for W3C use, as in the namespace declarations demonstrated in this section. As a result, do not attempt to create your own namespace named `xml:`.

The document element in the `http://www.XMML.com/myVeryOwnNamespace` namespace can be declared as follows, assuming that you map the namespace URI to the namespace prefix `XMML`:

```
<XMML:document xmlns:XMML="http://www.XMML.com/
myVeryOwnNamespace">
```

The short sample document then would be written as follows:

```
<XMML:document
 xmlns:XMML="http://www.XMML.com/myVeryOwnNamespace">
 <XMML:introduction>
 Some content goes here.
 </XMML:introduction>
 <!-- Many more lengthy elements could go here. -->
</XMML:document>
```

The alternative form of namespace declaration is the character sequence `xmlns` followed by the equal sign and the namespace URI enclosed in paired quotation marks:

```
xmlns="namespaceURI"
```



**Note** A namespace declared using the `xmlns="namespaceURI"` namespace declaration syntax is termed the *default namespace*.

Using this approach, you could express your document as follows:

```
<document
 xmlns="http://www.XMML.com/myVeryOwnNamespace">
 <introduction>
 Some content goes here.
 </introduction>
 <!-- Many more lengthy elements could go here. -->
</document>
```

For the human reader, the immediately preceding version is perhaps easier to read, but the earlier version using the XMML namespace prefix is less ambiguous. For the XMML processor, these documents are identical in a namespace sense. All the elements in each document are associated with the namespace URI `http://www.XMML.com/myVeryOwnNamespace`, and the XML processor uses the namespace URI for identifying namespaces.



**Caution** Namespace URIs in XML must be identical before they are considered to refer to the same namespace. So, if you create your own namespace, be sure that you are totally consistent in how you use upper-case and lowercase characters.

If document authors use URIs for domains that they own or otherwise legitimately use, a namespace URI should be unique. In addition, it is important that namespace URIs are persistent, to instill confidence that URIs and their related XML vocabularies remain stable.



**Note** The namespace URI need not point to any particular document. In particular, it need not contain any schema for the class of documents. The primary purpose of the namespace URI is to provide a unique and persistent identifier for an XML vocabulary.

Namespace declarations should be explicit in the start tag of an appropriate element or should be declared in the internal subset of the DTD. Namespace-aware processors are not required to be validating processors. Therefore, external declarations may not be accessed by a nonvalidating processor.

## Namespaces and Attributes

An attribute is assumed to be in the same namespace as the element with which it is associated. The namespace prefix of the element need not be expressed on the attribute, too. For example, both the `XMML:chapter` element and the `number` attribute in the following code are in the same namespace:

```
<XMML:chapter
xmlns:XMML=http://www.XMML.com/ABookNamespace
number="3">
<!-- Chapter content goes here. -->
</XMML:chapter>
```

An attribute need not share the same namespace prefix as the start tag on which it is placed. You have seen this already for the reserved attributes whose qualified names begin with the character sequence `xmlns`.

More generally, to use an attribute with a different namespace, the namespace prefix must be declared on either the element or an ancestor of it. Consider this example:

```
<someNamespace:someElement
xmlns:someNamespace=http://www.XMML.com/someNamespace
xmlns:someOtherNamespace=http://www.XMML.com/someOtherNamespace
someOtherNamespace:someAttribute="something"
>
```



In principle, the value of an XML attribute may contain the colon character. However, in namespace-aware XML documents, if the attribute is declared to be of type ID, IDREF, IDREFS, ENTITY, ENTITIES, or NOTATION, use of the colon character is not permitted.

## Namespace Well-Formedness

The “Namespaces in XML” Recommendation (<http://www.w3.org/TR/1999/REC-xml-names-19990114>), or, more precisely, an erratum to it (<http://www.w3.org/XML/xml-names-19990114-errata>), specifies a concept of namespace well-formedness.

Namespace well-formedness includes the XML well-formedness criteria defined in Chapter 2, “The Structure of an XML Document,” and Chapter 3, “XML Must Be Well-Formed,” together with the following constraints:

- Element type names and attribute names may contain either zero or one colon characters.
- No entity names or processing instruction targets or notation names may contain a colon character.
- No attribute that is declared to be of type ID, IDREF, ENTITY, ENTITIES, or NOTATION may contain a colon character in its value.

## Using Multiple Namespaces in a Document

Namespaces remove ambiguity from element type names. This assists the exchange of XML documents between users, but it also allows elements from different namespaces—in context implicitly different XML application languages—to be mixed in the same document.

You can mix elements in the same document if you understand the scope of namespaces.

## Scope of Namespaces

The scope of a namespace is limited to all elements nested inside the element on which the namespace was declared. However, the namespace may be altered by a namespace declaration on any of the nested elements.

Look at a simple example:

```
<somePrefix:anElement xmlns:somePrefix=
 http://www.XMML.com/someNamespace>
 <anotherPrefix:anElement
 xmlns:anotherPrefix="http://www.XMML.com/anotherNamespace">
 Some content
 </anotherPrefix:anElement>
</somePrefix:anElement>
```

The `somePrefix:anElement` element has an `anotherPrefix:anElement` element nested inside it. The `somePrefix:anElement` element is associated with the namespace URI `http://www.XMML.com/someNamespace`. The namespace declaration on the `anotherPrefix:anElement` element associates the `anotherPrefix` namespace prefix with a different namespace URI: `http://www.XMML.com/anotherNamespace`.

Let's return to the earlier example and explore how order elements from different namespaces might be used in a single XML document.

You might have a document with a structure similar to the following from a police department to a supplier:

```
<PurchaseOrder
xmlns="http://MyWonderfulLocalPoliceDepartment.com/
 purchaseOrders">
 <Reason>
 <order>
 <Location>Seattle</Location>
 <Description>A riot lasted several hours following clashes
 with anti-globalization protesters.
 </order>
 </Reason>
 <Supplier>
 <business:order
 xmlns:business="http://SomeFriendlyLocalBusiness.com/
 PurchaseOrders">
```

```
<business:orderNumber>
 AB123
</business:orderNumber>
<business:Items>
 <business:Item number="20">Tear Gas Cartridges
 </business:Item>
 <business:Item number="10">Personal Protection Masks
 </business:Item>
</business:Items>
</business:order>
</Supplier>
<PurchaseOrder>
```

Two namespace declarations exist. The first associates the default namespace with the namespace URI `http://MyWonderfulLocalPoliceDepartment.com/purchaseOrders`. The second, on the `business:order` element, associates the `business` namespace prefix with the namespace URI `http://SomeFriendlyLocalBusiness.com/PurchaseOrders`. You can clearly distinguish the `order` element in the default namespace from the `business:order` element. The XML processor uses the namespace URIs rather than the namespace prefixes to distinguish the namespaces.

## Summary

The need for XML namespaces to accommodate the likelihood of element type name clashes was explained in this chapter.

XML 1.0 uses a namespace URI to uniquely identify a namespace. A namespace URI is mapped to a namespace prefix by means of a namespace declaration. A namespace-aware element type name consists of a *namespace prefix* and a *local part*.



## LESSON 9

# The XML Path Language—XPath

*In this lesson, you will learn about the XML Path Language (XPath) and its view of an XML document. You also will learn how to use XPath to access nodes representing elements and attributes of XML documents.*

## How XPath Is Used

XPath is written in a non-XML syntax that enables you to define which part or parts of an XML document are selected by an XML processor or an application built on it. For example, you might want to select information to include in a company report from elements in a data store representing sales during a specified period.

Understanding basic XPath syntax is essential to being able to use Extensible Stylesheet Language Transformations (XSLT), described in Chapter 10, “XSLT—Creating HTML from XML”; Chapter 11, “XSLT—Transforming XML Structure”; and Chapter 12, “XSLT—Sorting XML.”



**Note** The description of XPath in this chapter focuses mostly on the selection of elements and attributes using the abbreviated XPath syntax. XPath also has an unabbreviated syntax.

To understand how to use XPath, you need to see how XPath models an XML document and how XPath syntax is used.

## An XML Document As a Hierarchy of Nodes

The description of XML documents in earlier chapters focused mostly on the syntax of the document. However, in XPath, an XML document is treated as a logical structure and is viewed as a hierarchy of nodes.

Each XML document has an in-memory hierarchy that represents the logical structure of the document, not its surface syntax. The document entity is represented in XPath as the *root node*. The representation of any XML document has exactly one root node. Because all other parts of an XML document are logically related to the document entity, the root node that represents the document entity is the apex of the hierarchy of nodes.

For an XML document, as distinct from an external parsed entity, each root node must have exactly one *element node* child.

For a very simple XML document such as the following, the XPath structure is similar to that shown in Figure 9.1:

```
<?xml version="1.0" ?>
<document>
George Bush is the son of George Bush.
</document>
```

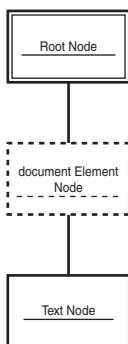


Figure 9.1 A representation of an XML document as an XPath hierarchy.

In this short example document, only three nodes are shown as representing the document. The root node is at the apex of the hierarchy. It has one

child node: the document element node. Notice that there is no node to represent the XML declaration in our document. The XPath 1.0 data model has no representation for the XML declaration or for the DOCTYPE declaration.



**Note** In XPath, element nodes and attribute nodes have names that correspond to the element type name and attribute name. Not all XPath node types have names.

The document element node has a single text node as its only child node. The value of the text node is the text string `George Bush is the son of George Bush`, which is the content of the document element.

A full representation of the document would also include a namespace node, representing the implicit default XML namespace, whose namespace URI is `http://www.w3.org/XML/1998/namespace`.

## XPath Axes

XPath is intended to allow a processor to navigate around the in-memory representation of an XML document. To describe how to navigate, it's necessary to express where that navigation starts and which direction(s) to take from that starting point—much like a set of street directions.

The starting point for navigation around a document using the XPath axes is called the *context node*. The context node can be the root node or any other node. A special form of syntax, beginning with the forward slash (/) character, indicates that the root node is the context node.

XPath processors navigate around the in-memory representation of an XML document by means of *axes*. Thirteen types of axes exist in XPath. Each axis represents a “direction” that the processor can take, beginning from the context node.

The following list briefly describes the XPath 1.0 axes. The `child` axis and `attribute` axis are the most frequently used and are considered in more detail later in this chapter.



**Note** All axis names in XPath begin with lowercase letters.

- **child axis**—Contains the child nodes (including element nodes) of the context node
- **attribute axis**—When the context node is an element node, contains an attribute node for each attribute on the element
- **descendant axis**—Contains the child nodes of the context node, their child nodes, and so on
- **self axis**—Contains the context node itself
- **descendant-or-self axis**—Contains the nodes in both the descendant axis and the self axis
- **parent axis**—Contains the parent node of the context node
- **ancestor axis**—Contains the parent node of the context node (if it has one), that node's parent node, and so on
- **ancestor-or-self axis**—Contains the nodes in the ancestor axis for the context node and the self axis
- **namespace axis**—When the context node is an element node, contains a namespace node for each in-scope namespace declaration
- **following axis**—Contains nodes later in document order than the context node, excluding nodes in the descendant axis, attribute axis, or namespace axis
- **following-sibling axis**—Contains nodes in the following axis, but only if the nodes have the same parent node as the context node
- **preceding axis**—Contains nodes that occur earlier in document order than the context node, excluding nodes in the ancestor axis

- preceding-sibling axis—Contains nodes that satisfy the criteria for the preceding axis but that also have the same parent node as the context node

## The Node Types in XPath 1.0

In XPath 1.0, seven types of node are specified, each of which corresponds to a structure in the source XML document:

- Root node—Represents the document entity
- Element node—Corresponds to an element in the source document
- Attribute node—Corresponds to an attribute in the source document
- Namespace node—Corresponds to each namespace declaration in scope for an element
- Processing instruction node—Corresponds to a processing instruction in the source document
- Comment node—Corresponds to a comment in the source document
- Text node—Corresponds to character content of an element in the source document

Each XPath axis has a *principal node type*. For the attribute axis, the principal node type is the attribute node. For the namespace axis, the principal node type is the namespace node. For all other axes, the principal node type is the element node.

Let's move on to consider how XPath expressions are written to define the axis that you plan to use and other parts of the expression.

## XPath Syntax

In XPath, an *expression* is used to select nodes to be processed in a way appropriate to the application in which XPath is being used. The most



commonly used type of XPath expression is the *location path* that returns a set of nodes, called a node-set.



A *location path* is an XPath expression that returns a node-set.

XPath uses two forms of syntax for an expression: unabbreviated syntax and abbreviated syntax.

We will briefly use the unabbreviated syntax to demonstrate the general principles of XPath syntax. Mainly abbreviated syntax is used in the rest of the chapter.

Unabbreviated syntax for a location path takes the following general form if the context node is the root node:

```
/axisName::nodeTest[predicate]/axisName::nodeTest[predicate]
```

For other context nodes, this form is used:

```
axisName::nodeTest[predicate]/axisName::nodeTest[predicate]
```

Let's break down the first of the two forms into its component parts. When a location path begins with the forward slash character (/), that indicates that the context node is the root node. The next part of the location path is the *location step*:

```
axisName::nodeTest[predicate]
```

This location step follows the normal form for a location step—an axis name followed by a pair of colon characters as a separator (:), followed by a node test and, optionally, by one or more predicates contained in square brackets.

Then you see the / character as a separator between location steps and a second location step that follows the same form as the first. The axis name can be any of the 13 axes listed earlier in the chapter.

The node test can be a name or a wildcard. If the location step was as follows, this would select element nodes named Document present in the child axis:

```
child::Document
```

Alternatively, you could use a wildcard, \*, and select all element nodes in the child axis by writing this:

```
child::*
```

The abbreviated syntax follows the same general structure as the unabbreviated syntax, but parts of the structure might not be explicitly expressed. For example, this location path

```
child::myElement
```

is equivalent to this in abbreviated syntax:

```
myElement
```

The child axis is essentially the default axis in XPath 1.0 and doesn't need to be expressed.

Similarly, if you want to select an edition attribute node, perhaps in code like this

```
<book edition="2nd">
```

you can use this unabbreviated syntax

```
attribute::edition
```

or this abbreviated syntax:

```
@edition
```

Here, the @ character is equivalent to the attribute axis name plus the :: separator.

The syntax of XPath bears similarities to the syntax of directory paths used in some operating systems. For example, consider this a document:

```
<myDocument>
 <myIntroduction>
 Some introduction content.
 </myIntroduction>
</myDocument>
```

You then could select the `myDocument` element node simply by writing this:

```
/myDocument
```

The preceding location path has a single location step. The `/` character indicates the root node. The following characters, `myDocument`, select all `myDocument` element nodes that are in the `child` axis of the root node. In this case, there is one `myDocument` element node that corresponds to the document element of the source document. No predicate is present in the location path.

As stated earlier, location paths may consist of multiple location steps. For example, you could write a location path to select the `myIntroduction` element node:

```
/myDocument/myIntroduction
```

Again, the initial `/` character indicates that the root node is the context node. The first location step, `myDocument`, selects the `child` axis implicitly. The node test selects only `myDocument` element nodes. There is no predicate. The second `/` character is a separator between node steps. The second location step, `myIntroduction`, implicitly selects the `child` axis and applies a node test that selects only `myIntroduction` element nodes. Again, there is no predicate.

Let's move on to look at some examples of selecting elements and attributes.

## Accessing Elements

A common and important use of XPath involves being able to access elements—or, more precisely, element nodes. XPath provides ways to select element nodes either by their relationship to another node or by name.

Consider a source document of the following structure:

```
<report>
 <author>John Smith</author>
 <date>2002/12/19</date>
 <title>Safety Assessment for Greenland Tropical Tours</title>
 <chapter number="1">Some text</chapter>
 <chapter number="2">Some text</chapter>
 <chapter number="3">Some text</chapter>
 <appendix>Some appendix stuff</appendix>
</report>
```

You can select all the child element nodes of the report element node by the following XPath location path:

```
/report/*
```

The / at the beginning of the location path indicates that you are starting at the root node. Then you select any report element nodes that are children of the root node. The next / character is a separator of one location step from another. The \* character is a wildcard that indicates any child element node.

In the example document, the root node has exactly one report element node child, which, in turn, has seven element node children. The node-set returned by the location path would contain seven element nodes—one author element node, one date element node, one title element node, three chapter element nodes, and an appendix element node.

Often you will want to select element nodes of a particular type. For example, if you wanted to select only title element nodes, you would simply incorporate the name of the desired type of element node in the location path:

```
/report/title
```

XPath would select only title element node children of report element node children of the root node.

The use of predicates when selecting elements is discussed later in this chapter when we look at some XPath functions.

## Accessing Attributes

Selecting attributes is another important use of XPath. You learned earlier that you can use the @ character as an abbreviation for `attribute::`. You can use the following short XML document to illustrate how to select attributes.

```
<book edition="1st" language="English">
<introduction>Some introduction text</introduction>
<chapter number="1">
Some Chapter 1 text.
</chapter>
<chapter number="2">
Some Chapter 2 text.
</chapter>
<chapter >
Some Chapter 3 text.
</chapter>
<appendix designation="A">
Appendix A's content
</appendix>
</book>
```

If you want to select the `edition` attribute on the `book` element, you can write this:

```
/book/@edition
```

It might help you understand this to look at the unabbreviated form:

```
/child::book/attribute::edition
```

Start with the root node as context node (as indicated by the initial / character). Then follow the `child` axis and apply a node test of `book`. This selects `book` element nodes, of which there is exactly one in this document. Using the node-set selected by the first location step, you then follow the `attribute` axis from that single `book` element node and apply a node test of `edition`.

When you select the nodes in the attribute axis, two attribute nodes are selected: the `edition` and `language` attribute nodes. When you apply the `edition` node test to that node-set, only the `edition` attribute node

matches. Thus, the location path selects a single attribute node corresponding to the `edition` attribute on the `book` element.

Now that you have seen how to select an attribute node, let's look at a way to select element nodes depending on the attribute node(s) that they possess. In the preceding example document, notice that the third chapter element has no `number` attribute. You can use that fact to select the first two chapter element nodes. The syntax to make that selection follows:

```
/book/chapter[@number]
```

The syntax `[@number]` is a predicate used to filter chapter element nodes. The part of the location path before the predicate selects element nodes named `chapter` that are child element nodes of the `book` element node. You then apply the predicate `[@number]` to the node-set that contains chapter element nodes. Only the first two chapter element nodes in document order possess `number` attribute nodes, so the third chapter element node (which has no `number` attribute node) is filtered out of the node-set.

On the other hand, if you wanted to select `number` attribute nodes on chapter element nodes, we could write this:

```
/book/chapter/@number
```

Follow the `child` axis from the root node to the single `book` element node and then the `child` axis to the chapter element nodes (of which there are three). Then, for each of the chapter element nodes, follow the `attribute` axis and apply a `number` node test. Only two of the chapter element nodes have a `number` attribute node, so only two `number` attribute nodes are selected.

## XPath Functions

XPath provides a function library that can manipulate or return four data types—node-sets, Boolean values, strings, and numbers. This section briefly looks at how two of those functions can be used.

## The `position()` Function

The `position()` function is widely used. Suppose that you have an XML document similar to the following:

```
<book>
<title>Sams Teach Yourself XML in 10 Minutes</title>
<chapter number="1" title="What is XML?">
<!-- The text of Chapter 1 could go here -->
</chapter>
<chapter number="2" title="The Structure of an XML document">
<!-- The text of Chapter 2 could go here -->
</chapter>
<!-- Other <chapter> elements and their content would go
 here -->
</book>
```

You might want to select the chapter element node that is second in document order. You can do that using the `position()` function by writing this:

```
/book/chapter[position()=2]
```

An abbreviated syntax exists for the `position()` function. The following syntax similarly selects the second chapter element node in document order:

```
/book/chapter[2]
```

## The `count()` Function

Often it is useful to be able to count how many nodes are in a node-set. The `count()` function can be used to do that.

For example, suppose that you wanted to offer special offers to customers who had placed more than four orders with the company. Imagine that you store information about orders in a format similar to the following:

```
<customer CustID="DEF876">
<customerName>Greenland Tropical Tours</customerName>
<orders>
<order date="2001/12/22">
```

```
<!-- Order details would go here. -->
</order>
<order date="2002/01/31">
<!-- Order details would go here. -->
</order>
<order date="2002/07/16">
<!-- Order details would go here. -->
</order>
<!-- More <order> elements can go here. -->
</orders>
</customer>
```

You could count the number of order element nodes using the `count()` function in the following XPath expression:

```
/customer/orders/count(order)
```

This would return a number that you could use, for example, in constructing a table in HTML, if you used the XPath expression in an XSLT stylesheet.

Many other XPath functions exist that cannot be described in the space available here.

## Summary

In this chapter, you were introduced to the XPath model of an XML document as a hierarchy of nodes. The node types were described, and the abbreviated XPath syntax to select elements and attributes was discussed. The chapter also gave some examples of widely used XPath functions.



# LESSON 10

## XSLT— Creating HTML from XML



*In this lesson, you will learn how to use the Extensible Stylesheet Language Transformations (XSLT) to create HTML documents from data stored as XML.*

### XSLT Basics

XSLT is designed to be used with XML documents to *transform* data into a form appropriate for presentation in a particular context or into an alternate XML structure. In this chapter, you examine the basics of how to use XSLT to create HTML documents.

The output from an XSLT transformation can be another XML format (this is discussed in Chapter 11, “XSLT—Transforming XML Structure”), HTML documents (the main subject of this chapter), or plain text (not discussed in this book).

### Why XSLT Is Needed

XML was originally intended to be transmitted across the Web, but since XML 1.0 was finalized in 1998, the implementation of XML in Web browsers has been slow and patchy. XML’s failure to take over the Web browser does not mean that it is of no use as a data storage medium for the Web; however, at least for now, XML must be transformed into formats that are convenient for most Web users if XML data is to be

available to them. That means that you need to be able to create HTML documents from XML data stores.

XSLT is ideally suited to creating HTML documents from XML server-side with the HTML produced being processed by a user's Web browser in the normal way. XSLT has an output mode to enable HTML output to be specified. In addition, XSLT can be used to produce HTML for desktop browsers, and Wireless Markup Language (WML) for mobile browsers from the same XML data store.

## **XSLT Tools**

A large and growing number of XSLT tools is available. Appendix B, "XML Tools," lists some commonly used XSLT tools together with URLs where they can be downloaded. Appendix B also contains information about how to install these tools.

The examples in this chapter and the following two chapters illustrate the use of XSLT using the Instant Saxon XSLT processor.

## **A Basic XSLT Stylesheet**

All XSLT stylesheets have either a `stylesheet` element or a `transform` element as their document element. Many stylesheet authors find it helpful to express XSLT elements with a namespace prefix of `xsl`, which is the indicative namespace prefix used in the XSLT 1.0 Recommendation.

XSLT elements in an XSLT stylesheet must be explicitly declared as belonging to the XSLT namespace. The XSLT namespace URI is `http://www.w3.org/1999/XSL/Transform`. Thus, a stylesheet will have either of the following basic structures, assuming that the namespace prefix `xsl` is declared as being associated with the XSLT namespace:

```
<?xml version="1.0" ?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform
version="1.0">
<!-- The other XSLT elements and the literal result elements
go here. -->
</xsl:stylesheet>
```

or

```
<?xml version="1.0" ?>
<xsl:transform
xmlns:xsl="http://www.w3.org/1999/XSL/Transform
version="1.0">
<!-- The other XSLT elements and the literal result elements
go here. -->
</xsl:transform>
```



**Note** The `xsl:stylesheet` or `xsl:transform` element must possess a version attribute. In XSLT 1.0, the only permitted value for the version attribute is 1.0.

Notice the XML declaration in both versions. All XSLT stylesheets are XML documents and, therefore, must follow all the well-formedness rules described earlier in Chapter 2, “The Structure of an XML Document,” and Chapter 3, “XML Must Be Well-Formed.”

## The Structure of an XSLT Stylesheet

The `xsl:stylesheet` element is allowed to have only certain specified XSLT elements as its children. These are referred to, a little confusingly (because they are second-level elements), as *top-level elements*. The top-level elements are shown in the following list. If an `xsl:import` element is present, it must come before top-level elements of any other type. Apart from that, the ordering of top-level elements is open to the developer’s preferences.

- `xsl:import`—Imports the content of one stylesheet (module) into the stylesheet containing the `xsl:import` element
- `xsl:include`—Includes the content of one stylesheet (module) in the stylesheet containing the `xsl:include` element
- `xsl:template`—Defines an XSLT template
- `xsl:output`—Defines parameters of the output document

- `xsl:attribute-set`—Used to define a named set of attributes
- `xsl:decimal-format`—Used in relation to the `format-number()` function
- `xsl:key`—Declares a named key to be used in relation to the `key()` function
- `xsl:namespace-alias`—Enables an XSLT stylesheet to be used to output another XSLT stylesheet as its result document
- `xsl:param`—(When a top-level element) Specifies a global parameter
- `xsl:preserve-space`—Controls whitespace handling in conjunction with `xsl:strip-space`
- `xsl:strip-space`—Controls whitespace handling in conjunction with `xsl:preserve-space`
- `xsl:variable`—(When a top-level element) Defines a global variable

Other XSLT elements are introduced as you meet them in the examples in this chapter and in Chapter 11 and Chapter 12, “XSLT—Sorting XML.”

Having taken a brief look at the basics of what an XSLT stylesheet contains, let’s move on to look at how to create an HTML Web page as the output of an XSLT transformation.

## Creating a Simple HTML Page

It is traditional in many introductory programming texts to create a program that provides a “Hello World!” greeting.

Listing 10.1 shows a very short XML document that stores the message.

---

### **LISTING 10.1** XSLTMessage.xml: A Simple Message in XML

---

```
<?xml version='1.0'?>
<XSLTMessage>
Hello World!
</XSLTMessage>
```

You can use the XSLT stylesheet shown in Listing 10.2 to create an HTML Web page that extracts the message from Listing 10.1 and places it in the HTML page.

**LISTING 10.2** XSLTMessage.xsl: A Stylesheet to Extract the Message from Listing 10.1

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 version="1.0"
 >

 <xsl:template match="/">
 <html>
 <head>
 <title><xsl:value-of select="name(/XSLTMessage)" /></title>
 </head>
 <body>
 <xsl:apply-templates select="/XSLTMessage" />
 </body>
 </html>
 </xsl:template>

 <xsl:template match="XSLTMessage">
 <p><xsl:value-of select="text()" /></p>
 </xsl:template>
 </xsl:stylesheet>
```

Before looking at the HTML output document that you can create, let's analyze what the XSLT stylesheet does.

The first line is an XML declaration. That is followed by the `xsl:stylesheet` element. It has a `version` attribute, which is compulsory, and a namespace declaration that associates the namespace prefix `xsl` with the XSLT namespace URI, `http://www.w3.org/1999/XSL/Transform`. So, it is clear that the `xsl:stylesheet` element and the other elements in the stylesheet with a namespace prefix `xsl` are XSLT elements.

When an XSLT processor is satisfied that a document is a well-formed XSLT stylesheet, it looks for an `xsl:template` element whose `match` attribute has a value of `/`. In other words, the template is applied to the root node.

Let's look at the content of that `xsl:template` element a little more closely:

```
<xsl:template match="/">
<html>
<head>
<title><xsl:value-of select="name(/XSLTMessage)" /></title>
</head>
<body>
<xsl:apply-templates select="/XSLTMessage" />
</body>
</html>
</xsl:template>
```

The first two lines nested within the `xsl:template` element contain *literal result elements*. In other words, you create an `html` start tag followed by `head` and `title` start tags. The content of the `title` element is defined using an XSLT `xsl:value-of` element, which you will use in many of your XSLT stylesheets. One question immediately arises: “The value of what?” That is answered by the value of the `select` attribute of the `xsl:value-of` element.

In this case, the value of the `select` attribute is this:

```
name (/XSLTMessage)
```

You use the `name()` function to extract the name of the document element. We have expressed that here by giving the element type name literally.

You then create the end tags for the `title` and `head` elements. Then you create the start tag for the `body` element.

The content of the `body` element is defined by the `xsl:apply-templates` element. Its `select` attribute indicates that a template that matches the context node defined by the XPath expression `/XSLTMessage` is to be instantiated.

```
<xsl:template match="XSLTMessage">
<p><xsl:value-of select="text()" /></p>
</xsl:template>
```

The only other template in the stylesheet matches the value in the `select` attribute of the `xsl:apply-templates` element. The content of that

template defines the content of the body element of the HTML output document.

That content begins with a `p` start tag. It is followed by content defined by an `xsl:value-of` element. The value of the `select` attribute is `text()`; it selects the text node that is a child of the context node, which is the `XSLTMessage` element node. The template is completed by the creation of an end tag of the `p` element.

When that template is complete, you return to the template from which the template was instantiated. In other words, processing goes back to the template that matches the root node. Processing completes by outputting an end tag for the body and html elements—both literal result elements—of the HTML output document.

To run the transformation, assuming that Instant Saxon and Listings 10.1 and 10.2 are in the same directory, simply navigate to that directory and type this:

```
saxon XSLTMessage.xml XSLTMessage.xsl > XSLTMessage.html
```

Listing 10.3 shows the HTML document output by the Instant Saxon XSLT processor.

---

**LISTING 10.3** XSLTMessage.html: The Output of Applying Listing 10.2 to Listing 10.1

---

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html;
 charset=utf-8">
 <title>XSLTMessage</title>
</head>
<body>
 <p>
 Hello World!
 </p>
</body>
</html>
```

The Instant Saxon processor has added a `meta` element to the head of the HTML document.

If you plan to regularly create HTML documents using XSLT, it is useful to have an XSLT template specific to HTML and save some repeated typing. Listing 10.4 gives a bare outline template that you might want to adapt to your own needs.

---

**LISTING 10.4** HTMLTemplate.xsl: An XSLT Stylesheet to Create a Basic HTML Document

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/">
<html>
<head>
<title><!--title goes here--></title>
</head>
<body>
<!-- XSLT code to create page content goes here. -->
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

If you routinely want to include metadata about the author of the HTML document, its keywords, and so on, you can add those to the section that creates the head of the HTML document.

## Creating an HTML List

This section looks at how to create an HTML list from an XML source document.

The XML source document contains information about a series of reports produced for XMML.com. In this HTML Web page, you will choose to display only reports for which the year of the report is 2002. Listing 10.5 shows the source XML document.



**LISTING 10.5** XMMLReports.xml: Reports Presented to XMML.com

---

```
<?xml version='1.0'?>
<XMMLReports>
 <Report year="2000">
 <Title>Sales Opportunities</Title>
 <Author>Peter Mallan</Author>
 <Summary>The opportunities for XML consultancy look good for
 2001.</Summary>
 <Content>
 <!-- Main text would go here. -->
 </Content>
 </Report>
 <Report year="2001">
 <Title>SVG - A Graphics Standard</Title>
 <Author>Pamela Askew</Author>
 <Summary>Scalable Vector Graphics, SVG, looks to have enormous
 potential in multnamespace XML documents.</Summary>
 <Content>
 <!-- Main text would go here. -->
 </Content>
 </Report>
 <Report year="2002">
 <Title>Market Conditions</Title>
 <Author>Stephen J. Doppelganger</Author>
 <Summary>Market conditions are much less favorable than in
 2001.</Summary>
 <Content>
 <!-- Main text would go here. -->
 </Content>
 </Report>
 <Report year="2002">
 <Title>XML Schema Languages</Title>
 <Author>Karen Clark</Author>
 <Summary>W3C XML Schema and RelaxNG both have positive
 aspects.</Summary>
 <Content>
 <!-- Main text would go here. -->
 </Content>
 </Report>
</XMMLReports>
```

Listing 10.6 shows the XSLT stylesheet that selects for display reports dated 2002 and displays the title, year, and author name for each such report.

---

**LISTING 10.6** XMMLReports.xsl: A Stylesheet to Display Year 2002 Reports

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<h1>XMML.com Reports for 2002</h1>

<xsl:apply-templates select="//Report[@year='2002']" />

</body>
</html>
</xsl:template>

<xsl:template match="Report">

<xsl:value-of select="Title" />(<xsl:value-of
 select="@year"/>):<xsl:text>
</xsl:text><xsl:value-of select="Author" />

</xsl:template>

</xsl:stylesheet>
```

The preceding stylesheet bears many similarities to those you saw earlier in this chapter.

Notice the `xsl:apply-templates` element in the template that matches the root node. It is nested between the start and end tags of a `ul` element. So, the `xsl:apply-templates` element is used to create the content of the unordered list. The value of its `select` attribute is

`//Report[@year='2002']`. The pair of forward slash characters (`//`) is abbreviated syntax for the descendant-or-self axis. Essentially, `//Report` means to select any `Report` element node in the document. The predicate `[@year='2002']` filters that node-set so that it includes only `Report` element nodes that possess a `year` attribute whose value is `2002`.

The XSLT template that matches the nodes in that node-set is then processed. The `xsl:template` element with the `match` attribute whose value is `Report` is instantiated. A list item element, `li`, is created. The `xsl:value-of` element is used three times:

```
<xsl:value-of select="Title" />(<xsl:value-of
 select="@year"/>):<xsl:text> </xsl:text>
<xsl:value-of select="Author" />
```

First, the text content of the `Title` element is obtained, followed by the value of the `year` attribute of the `Report` element in parentheses. The third `xsl:value-of` element selects the text content of the `Author` element node for the `report`.

The `xsl:text` element is used to insert whitespace—in this case, a single space character—between the colon character and the author's name. Using the `xsl:text` element to output whitespace ensures that it is preserved in the output document.

The output of the transformation is shown in Listing 10.7.

---

**LISTING 10.7** XMMLReports.html: The Output of Applying Listing 10.6 to Listing 10.5

---

```
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html;
 charset=utf-8">
 <title></title>
 </head>
 <body>
 <h1>XMML.com Reports for 2002</h1>

 Market Conditions(2002): Stephen J. Doppelganger
 XML Schema Languages(2002): Karen Clark

 </body>
</html>
```

## Creating an HTML Table

In the final example in this chapter, you create an HTML table in the output document.

In this example, you look at how information about major news items stored in XML can be transformed using XSLT to produce an HTML page with links to the full news items. The structure of the source XML document is shown in Listing 10.8.

---

### LISTING 10.8    XMMLNews.xml: An XML-Based Data Store Containing News Information

---

```
<?xml version='1.0'?>
<XMMLNews>
 <Story>
 <Headline>Teddy Bear's Picnic a Success</Headline>
 <Header>2002 Teddy Bear's Picnic a great success.</Header>
 <MainText>The Drum Castle (http://www.drum-castle.org.uk)
 Teddy Bear's Picnic for 2002 was a great success.
 etc</MainText>
 </Story>
 <Story>
 <Headline>Snow Falls in Antarctica</Headline>
 <Header>Heavy snow falls reported in Antarctica.</Header>
 <MainText>The first snows of the Antarctic winter fell
 yesterday.</MainText>
 </Story>
 <Story>
 <Headline>Brazil Win Cup.</Headline>
 <Header>The Brazilian soccer team won the 2002 World
 Cup.</Header>
 <MainText>After an exciting game, Brazil confirmed their
 dominance of world soccer with a convincing win.</MainText>
 </Story>
</XMMLNews>
```

The number of stories and the components parts of each story have been kept short to save space.

The aim is to use XSLT to create an HTML Web page with a table, with each item in the table having a link to the full text of the story. Listing 10.9 shows a stylesheet that can produce the desired HTML Web page.

---

**LISTING 10.9** XMMLNews.xsl: A Stylesheet to Produce a List of Stories in HTML
 

---

```

<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/" >
<html>
<head>
<title>XMMLNews - Latest Stories</title>
</head>
<body>
<h1>XMML News Service</h1>
<table>
<tr>
<td>Headline</td>
<td>Main Text</td>
</tr>
<xsl:apply-templates select="/XMMLNews/Story"/>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="Story" >
<tr>
<td><xsl:value-of select="Headline" /></td>
<td><xsl:value-of select='MainText' /></td>
</tr>
</xsl:template>

</xsl:stylesheet>

```

The HTML document output is shown in Listing 10.10.

---

**LISTING 10.10** XMMLNews.html: A Table of Headlines and Stories
 

---

```

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

```

**LISTING 10.10**   Continued

```
<title>XMMLNews - Latest Stories</title>
</head>
<body>
 <h1>XMML News Service</h1>
 <table>
 <tr>
 <td>Headline</td>
 <td>Main Text</td>
 </tr>
 <tr>
 <td>Teddy Bear's Picnic a Success</td>
 <td>The Drum Castle (http://www.drum-castle.org.uk) Teddy
 Bear's Picnic for 2002 was a great success. etc</td>
 </tr>
 <tr>
 <td>Snow Falls in Antarctica</td>
 <td>The first snows of the Antarctic winter fell
 yesterday.</td>
 </tr>
 <tr>
 <td>Brazil Win Cup.</td>
 <td>After an exciting game, Brazil confirmed their
 dominance of world soccer with a convincing win.</td>
 </tr>
 </table>
</body>
</html>
```

Onscreen, the output is a basic HTML table.

## Summary

In this lesson, you were introduced to the ways in which XSLT plays a valuable role in the current XML world. The structure of an XSLT stylesheet was described, and examples were given to show using XSLT to create several types of HTML pages, ranging from very basic on up to pages that include a list and a table.

# LESSON 11

## XSLT— Transforming XML Structure



*In this lesson, you will learn how to use XSLT to restructure content in XML documents.*

### Why Change Structure?

Chapter 8, “Namespaces in XML,” mentioned problems that can occur frequently as XML documents are exchanged among increasing numbers of individuals and companies. XML namespaces do a lot to solve the problem in which two identical element type names are used with different meanings. This chapter looks at how XSLT can be used to provide a solution to the problem of different element type names being used to refer to the same concept or real-world value. In addition, it looks more generally at how XSLT can be used to restructure XML documents.

For example, one company might store an order like this, with date information in an element:

```
<order>
<date>2002-12-29</date>
<!-- More content here. -->
</order>
```

It might deal with a company that stores information about an order like this, with date information stored in an attribute:

```
<order date="2002-12-29">
<!-- More content here. -->
</order>
```

When the companies exchange documents, the first company could send information in its own format. So, the second company would need to transform the XML so that the `date` element is removed and a `date` attribute is added to the `order` element. When the second company sends information back, the opposite process would need to be carried out.

Historically, companies and other organizations have had their own ways of describing the data that they use in the course of their business. When there was no direct exchange of data, that didn't matter too much. However, as businesses have started exchanging data—orders or shared information, for example—issues relating to the structure of that data have become increasingly important.

It would be pretty unusual for two companies to have identical data structures. But if both companies use XML to store their data, XSLT can be used to move from one format to another.

A number of possible solutions exist in any one setting. For example, each company could create an XSLT stylesheet to convert from the other company's data format. If a very small number of business partners are involved, that might work well. However, if a large number of companies in the same business sector work together, an alternate approach might be better.

If all the companies in the business sector can agree on a common data format for a particular type of data, each company needs only two XSLT stylesheets: one stylesheet to transform the company's format to the common format and another stylesheet to transform the common format to the company's format. It doesn't matter how large the number of business partners grows. If all use the common format, they don't need to add to the two stylesheets just mentioned.

Let's look at how transformations between formats can be carried out using XSLT. In the process of transforming XML documents from one structure to another, you need to be able to carry out three tasks:

- Copy elements from one document to another, possibly in a different part of the structure of the document



- Create new elements (for example when a value is contained in an attribute value in the source document and you need an element in the output document)
- Create new attributes (for example, when a value is contained in text element content in the source document and you need an attribute to contain the value in the output document)

First, let's look at how you can copy elements from one document to another.

## Copying Elements

When you copy elements from one XML document to another, you will likely place the element in a new place in the structure. Sometimes you will want to copy an element only (without any child elements); other times, you might want to copy an element together with any content that it has. The first type of copy is called a *shallow copy*, and the second type is called a *deep copy*.

### Shallow Copy

In XSLT, a shallow copy is carried out using the `xsl:copy` element. Listing 11.1 shows a simple purchase order for training services.

---

**LISTING 11.1** PurchaseOrder.xml: A Purchase Order in XML

---

```
<?xml version='1.0'?>
<PurchaseOrder>
 <Date>2003/02/20</Date>
 <To>XMML Training Services</To>
 <From>Acme Computing</From>
 <Items>
 <Item>
 <StockNumber>DBI99</StockNumber>
 <Description>Database Introduction</Description>
 <Quantity>2</Quantity>
 </Item>
 <Item>
 <StockNumber>MSVG101</StockNumber>
```

**LISTING 11.1** Continued

---

```

 <Description>Introduction to Mobile SVG</Description>
 <Quantity>1</Quantity>
 </Item>
</Items>
</PurchaseOrder>

```

When such an XML purchase order is sent to XMML Training Services, the data contained in it can be reused for purposes of the receiving business partner. Listing 11.2 shows a stylesheet to convert the purchase order for the purposes of the recipient. The `xsl:copy` element is used in this transformation.

**LISTING 11.2** PurchaseToOrder.xsl: An XSLT Stylesheet to Create an XML Order Received File

---

```

<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>

<xsl:template match="/">
<XMMLOrder>
<xsl:apply-templates select="/PurchaseOrder/Date" />
<xsl:apply-templates select="/PurchaseOrder/From" />
<xsl:apply-templates select="/PurchaseOrder/Items" />
</XMMLOrder>
</xsl:template>

<xsl:template match="Date|From">
<xsl:copy>
<xsl:value-of select="." />
</xsl:copy>
</xsl:template>

<xsl:template match="Items">
<xsl:copy>
<xsl:value-of select="." />
</xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

This stylesheet takes you only part way to the desired output, as you can see by examining the content of the output document in Listing 11.3.

**LISTING 11.3** XMMLOrder.xml: The Result of Applying Listing 11.2 to Listing 11.1

---

```
<?xml version="1.0" encoding="UTF-8"?>
<XMMLOrder>
 <Date>2003/02/20</Date>
 <From>Acme Computing</From>
 <Items>

 DBI99
 Database Introduction
 2

 MSVG101
 Introduction to Mobile SVG
 1

</Items>
</XMMLOrder>
```

You have copied across the element node and, for the `Date` and `From` element nodes, included the element's content using the `xsl:value-of` element. So far, so good. However, when you use the `xsl:value-of` element with `xsl:copy` for the `Items` element node, you output only the text content of its descendant elements—but without outputting the corresponding start and end tags of the `Item` element and its child elements.

As you have seen, the `xsl:copy` element produces a shallow copy. You supply the content of the copied element by using the `xsl:value-of` element.

To correctly output and copy all the content of the `Items` element node, including the start and end tags of the descendant elements, you need to carry out a deep copy.

## Deep Copy

The `xsl:copy-of` element is used in XSLT to carry out a deep copy.

Listing 11.4 shows an XSLT stylesheet that carries out the desired transformation. In this case, however, it uses the `xsl:copy-of` element in place of the `xsl:copy` element used in Listing 11.3.

### **LISTING 11.4** PurchaseToOrder2.xsl: Using `xsl:copy-of` to Produce a Deep Copy

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="xml"
 indent="yes"
 encoding="UTF-8" />
<xsl:template match="/">
<XMMLOrder>
<xsl:apply-templates select="/PurchaseOrder/Date" />
<xsl:apply-templates select="/PurchaseOrder/From" />
<xsl:apply-templates select="/PurchaseOrder/Items" />
</XMMLOrder>
</xsl:template>

<xsl:template match="Date|From">
<xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="Items">
<xsl:copy-of select="." />
</xsl:template>

</xsl:stylesheet>
```

In Listing 11.4, the `xsl:copy` elements have been replaced with `xsl:copy-of` elements, which have a `select` attribute that selects the content of the context node. As you can see in Listing 11.5, the output of the transformation, using the `xsl:copy-of` element gives the text content of the `Date` and `From` elements and also gives the full hierarchy of elements and their content contained within the `Items` element.

**LISTING 11.5** XMMLOrder2.xml: The Results Document After Applying Listing 11.4 to Listing 11.1

---

```
<?xml version="1.0" encoding="UTF-8"?>
<XMMLOrder>
 <Date>2003/02/20</Date>
 <From>Acme Computing</From>
 <Items>

 <Item>

 <StockNumber>DBI99</StockNumber>

 <Description>Database Introduction</Description>

 <Quantity>2</Quantity>

 </Item>

 <Item>

 <StockNumber>MSVG101</StockNumber>

 <Description>Introduction to Mobile SVG</Description>

 <Quantity>1</Quantity>

 </Item>

 </Items>
</XMMLOrder>
```

In some transformations, you cannot simply copy elements from source document to output document because there is some fundamental change in structure. Often you will need to create new elements or attributes in the output document. First let's look at how to create new elements.

## Creating New Elements

In this section, you will look at two reasons why you might need to create new elements.

In some transformations, the element names in the source document and the corresponding element name in the output document differ. For example, suppose that a company based in the United States is doing business with a company based in the United Kingdom. The U.S. company wants to place an order for white shirts. In the United States, the company might describe a white shirt, using this:

```
<Color>white</Color>
```

The U.K. company might use this line, however:

```
<Colour>white</Colour>
```

Only one letter is different in the element type name, but that is enough to trip up an XML parser.

Consider also that you might want to create a new element in the output document if the source document uses an attribute to store data that needs to be contained in an element in the output document. One company could have this

```
<Shirt size="medium" />
```

and want to share the information with a company that stores data like this:

```
<Shirt>
<Size>medium</Size>
</Shirt>
```

You can explore both issues in Listing 11.6. The U.S. company uses a `Color` element that stores information about the color of the shirt. Because of differences in spelling, this information is held in a `Colour` element in the U.K. company's data store. Also, the U.K. company stores shirt size information in a `Size` element.

---

**LISTING 11.6** USShirts.xml: An Order for Shirts from a Company Based in the United States

---

```
<?xml version='1.0'?>
<USShirts>
<Order>
<Date>2003/12/13</Date>
```

**LISTING 11.6** Continued

---

```
<From>US Shirt Company</From>
<To>UK Shirt Company</To>
<Shirt size="medium">
<Color>Cerise</Color>
<Quantity>100</Quantity>
</Shirt>
</Order>
</USShirts>
```

Listing 11.7 shows the type of output document required. Notice the new Colour element and the Size element. In fact, Listing 11.7 is the result of applying the stylesheet in Listing 11.8 to Listing 11.6.

**LISTING 11.7** UKShirts.xml: The U.K. Company's Form of XML for the Order

---

```
<?xml version="1.0" encoding="UTF-8"?>
<UKShirts>
 <Date>2003/12/13</Date>
 <From>US Shirt Company</From>
 <To>UK Shirt Company</To>
 <Shirt>
 <Size>Medium</Size>
 <Colour>Cerise</Colour>
 <Quantity>100</Quantity>
 </Shirt>
</UKShirts>
```

Notice that the Shirt element no longer has a size attribute; instead, it has a Size element child.

Listing 11.8 shows an XSLT stylesheet that creates two new elements using the `xsl:element` element. One of the new elements replaces a Color element with a Colour element. The other replaces a size attribute with a Size element.

**LISTING 11.8** USShirtToUK.xsl: An XSLT Stylesheet to Transform to the U.K. Company's Data Structure

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
```

**LISTING 11.8**    Continued

---

```
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="xml" indent="yes" encoding="UTF-8" />
<xsl:template match="/">
<UKShirts>
<xsl:apply-templates select="/USShirts/Order" />
</UKShirts>
</xsl:template>

<xsl:template match="Order" >
<xsl:apply-templates select="Date" />
<xsl:apply-templates select="From" />
<xsl:apply-templates select="To" />
<xsl:apply-templates select="Shirt" />
</xsl:template>

<xsl:template match="Date" >
<xsl:copy-of select="." />
</xsl:template>

<xsl:template match="From|To" >
<xsl:copy-of select="." />
</xsl:template>

<xsl:template match="Shirt">
<xsl:copy>
<xsl:element name="Size">
<xsl:value-of select="@size" />
</xsl:element>
<xsl:apply-templates select="Color" />
<xsl:apply-templates select="Quantity" />
</xsl:copy>
</xsl:template>

<xsl:template match="Color">
<xsl:element name="Colour">
<xsl:value-of select="." />
</xsl:element>
</xsl:template>

<xsl:template match="Quantity">
<xsl:copy-of select="." />
</xsl:template>

</xsl:stylesheet>
```



The `xsl:element` element is used in two templates in the XSLT stylesheet.

```
<xsl:template match="Shirt">
<xsl:copy>
<xsl:element name="Size">
<xsl:value-of select="@size" />
</xsl:element>
<xsl:apply-templates select="Color" />
<xsl:apply-templates select="Quantity" />
</xsl:copy>
</xsl:template>
```

First, you use `xsl:copy` to create a shallow copy of the `Shirt` element. Because you have used a shallow copy, that leaves you free to create new content for that element. You use the `xsl:element` element to create a new element—the `Size` element—to contain the content of the `size` attribute. Notice that the `name` attribute of the `xsl:element` element is the same as the element type name of the element you are creating.

The `xsl:apply-templates` element instantiates a template that matches the `Color` element:

```
<xsl:template match="Color">
<xsl:element name="Colour">
<xsl:value-of select="." />
</xsl:element>
</xsl:template>
```

Here the `xsl:element` is used to create a new `Colour` element (U.K. spelling) to replace the `Color` element (U.S. spelling). The content of the new element is identical to the content of the `Color` element, so you can simply use the `xsl:value-of` element to select the text content of the `Color` element as the content of the new `Colour` element.

By using the `xsl:copy`, `xsl:copy-of`, and `xsl:element` elements, you have been able to transform from the U.S. data format to the format desired by the U.K. company.

However, if you want to transform the data from the U.K. format to the U.S. format, you need to learn how to create new attributes.

## Creating New Attributes

To create a new attribute in the output document, you need to use the `xsl:attribute` element.

On this occasion, you will transform a source document in the U.K. company's format to an XML output document in the U.S. company's format. Listing 11.9 shows an XSLT stylesheet that can carry out the transformation.

---

### LISTING 11.9   UKShirtsToUS.xsl: An XSLT Stylesheet to Transform to the U.S. Company's Data Format

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
 <xsl:output method="xml" indent="yes" encoding="UTF-8" />
 <xsl:template match="/">
 <USShirts>
 <Order>
 <xsl:apply-templates select="/UKShirts" />
 </Order>
 </USShirts>
 </xsl:template>

 <xsl:template match="Order" >
 <xsl:apply-templates select="Date" />
 <xsl:apply-templates select="From" />
 <xsl:apply-templates select="To" />
 <xsl:apply-templates select="Shirt" />
 </xsl:template>

 <xsl:template match="Date" >
 <xsl:copy-of select="." />
 </xsl:template>

 <xsl:template match="From|To" >
 <xsl:copy-of select="." />
 </xsl:template>

 <xsl:template match="Shirt">
 <xsl:copy>
```

**LISTING 11.9** Continued

---

```
<xsl:attribute name="size">
<xsl:value-of select="Size" />
</xsl:attribute>
<xsl:apply-templates select="Colour" />
<xsl:apply-templates select="Quantity" />
</xsl:copy>
</xsl:template>

<xsl:template match="Colour">
<xsl:element name="Color">
<xsl:value-of select="." />
</xsl:element>
</xsl:template>

<xsl:template match="Quantity">
<xsl:copy-of select="." />
</xsl:template>

</xsl:stylesheet>
```

Use the `xsl:attribute` in the template that matches the `Shirt` element node:

```
<xsl:template match="Shirt">
<xsl:copy>
<xsl:attribute name="size">
<xsl:value-of select="Size" />
</xsl:attribute>
<xsl:apply-templates select="Colour" />
<xsl:apply-templates select="Quantity" />
</xsl:copy>
</xsl:template>
```

First copy the `Shirt` element using `xsl:copy`. Then use the `xsl:attribute` element to add an attribute to the `Shirt` element. The value of that new shirt attribute is obtained from the content of the `Size` element. The first `xsl-apply-templates` element is used to create a new `Color` element to replace the `Colour` element used by the U.K. company's format.

If you can carry out shallow and deep copies and create new elements and new attributes, you can accomplish many of the basic tasks that are necessary in converting one XML format to another.

## Summary

In this lesson, you learned about the need to transform one XML vocabulary to another. You saw how to use the `xsl:copy` element to make a shallow copy and the `xsl:copy-of` element to make a deep copy of elements from the source document. You also saw how to create new elements using the `xsl:element` element and create new attributes in the output document using the `xsl:attribute` element.

# LESSON 12

## XSLT— Sorting XML



*In this lesson, you will learn how to sort selected content to produce output documents sorted on one or multiple criteria.*

## Conditional Processing and Sorting Data

As you saw in Chapter 11, “XSLT—Transforming XML Structure,” an XML document may not be in the precise form that you want to work with. In addition to providing tools to copy or to create new elements and attributes, XSLT provides tools to process data according to criteria that you set. Among the important functionality that XSLT provides is the capability to process elements (or not) based on criteria that you define or to sort data according to criteria that you specify.

Many programming languages have `if ... then ... else` statements or similar constructs. In XSLT, you can use the `xs1:if` element for similar purposes. For more complicated choices, conventional programming languages have a `switch/case` statement or similar construct. In XSLT, you can use the `xs1:choose` element to make choices when more than two options are involved.

Data stored in an XML document may be ordered according to some arbitrary criteria, perhaps as simple as the sequence in which elements and their content were first entered into the data store. For some purposes, you likely will want to use data in various orders—including alphabetical order, date order, and by value of element content or attribute value. XSLT possesses the `xs1:sort` element to provide sorting functionality.

First, let's look more closely at conditional processing and how it is supported in XSLT.

## Conditional Processing

Controlling choices in XSLT as to how and whether a node is to be processed falls into two categories:

- Choice of two processing alternatives, one of which is to do nothing
- Choice of multiple (greater than two) options

### The **xsl:if** Element

The `xsl:if` element in XSLT corresponds broadly to `if...then... else` type statements in other programming languages, but in XSLT there is no `else` option. If you want an `else` option, you must use `xsl:choose`, described later in this chapter.

An `xsl:if` element is always nested within an `xsl:template` element. XSLT elements that are nested within templates are termed *instructions* or *instruction elements*.

The general form is like this:

```
<xsl:template>
<!-- Other content can go here. -->
<xsl:if test="XPathExpression">
<!-- Anything in here is executed if the test attribute
 returns true. -->
</xsl:if>
<!-- Other content can go here. -->
</xsl:template>
```

The value of the `test` attribute is converted to a Boolean value. If the result is `true`, the content of the `xsl:if` element is instantiated. If the result is `false`, the content of the `xsl:if` element is skipped.

The following list summarizes the rules for conversion to Boolean values:

- If the expression is a node-set, the Boolean value `true` is returned if the node-set contains one or more nodes.
- If the expression is a number, the Boolean value returned is `true` if the number is not zero.
- If the expression is a string, the Boolean value returned is `true` if the string is not the empty string.

Some possible uses of `xsl:if` can be more succinctly expressed using a predicate.

For example, if you wanted to specify that a document was to be included in a results document only if a `version` attribute had the value `final`, you could use `xsl:if` inside a template that matched the `Document` element node:

```
<xsl:template match="Document">
 <!-- All Document elements get to here. -->
 <xsl:if test="@version='final'">
 <!-- Conditional processing of final version documents only
 goes here. -->
 </xsl:if>
 <!-- Any other processing that applies to all Document element
 nodes could go here. -->
</xsl:template>
```

However, you could just as easily control things using a predicate `[@version="final"]` in the location path in an `xsl:apply-templates` element's `select` attribute.

However, when you want to process all `Document` element nodes but process them differently depending on whether they are `final` or `draft`, you can make use of the `xsl:if` element. Listing 12.1 shows an example XML source document.

### **Listing 12.1** Documents.xml: An XML Data Store Containing Final and Draft Documents

---

```
<?xml version='1.0'?>
<Documents>
 <Document version="outdated">
 <Title>XMML.com Training Courses</Title>
 <Author>Karen Karenstein</Author>
 <Date>1999/12/20</Date>
 <Content>December 1999 content.</Content>
 </Document>
 <Document version="final">
 <Title>XMML.com Training Courses</Title>
 <Author>Camilla Zukowski</Author>
 <Date>2002/12/29</Date>
 <Content>December 2002 content.</Content>
 </Document>
 <Document version="draft">
 <Title>XMML.com Training Courses</Title>
 <Author>Camilla Zukowski</Author>
 <Date>2002/07/31</Date>
 <Content>July 2002 draft content.</Content>
 </Document>
 <Document version="final">
 <Title>XMML.com Consultancy Services</Title>
 <Author>Paul Hartington</Author>
 <Date>2003/04/29</Date>
 <Content>April 2003 consultancy services
 information.</Content>
 </Document>
</Documents>
```

Listing 12.2 is an XSLT stylesheet that outputs the information about final version documents with a red h1 header and full information about the document author, together with the document content. For documents with a version attribute equal to draft or outdated, the document title is output in an h2 element in blue and only the document status is output for each such document.

### **Listing 12.2** Documents.xsl: An XSLT Stylesheet Using the xsl:if Element

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
```



**Listing 12.2** Continued

---

```

 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/">
<html>
<head>
<title>XMMML.com Documents</title>
<style type="text/css">
h2{color:red}
h3{color:blue}
</style>
</head>
<body>
<h1>All XMMML.com documents.</h1>
<xsl:apply-templates select="//Document" />

</body>
</html>
</xsl:template>

<xsl:template match="Document">
<xsl:if test="@version='final'">
<h2>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h2>
</xsl:if>
<xsl:if test="not(@version='final')">
<h3>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h3>
</xsl:if>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
<xsl:if test="@version='final'">
<p>Document Author:<xsl:text> </xsl:text><xsl:value-of select=
 "Author" /></p>
<p>Document Content:<xsl:text> </xsl:text><xsl:value-of
select=
 "Content" /></p>
</xsl:if>
</xsl:template>

</xsl:stylesheet>

```

Listing 12.3 contains the output from the XSLT transformation.

**Listing 12.3** Documents.html: The Output Document After Applying Listing 12.2 to Listing 12.1

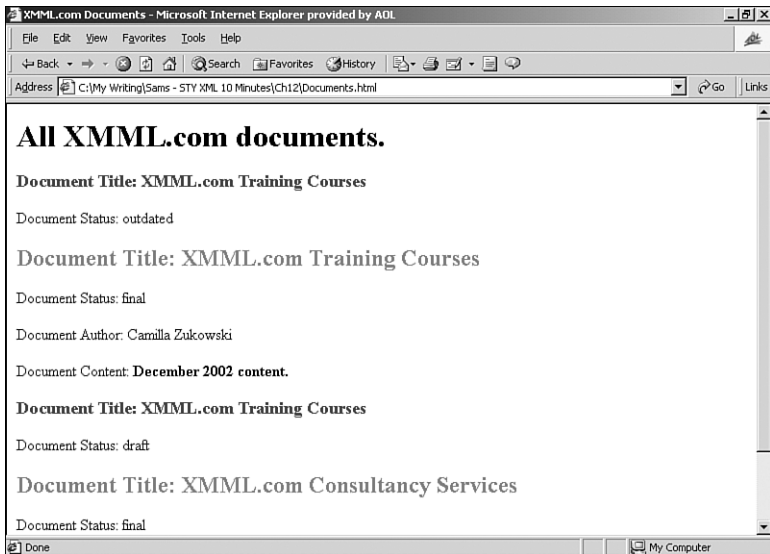
---

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

 <title>XMML.com Documents</title><style type="text/css">
h2{color:red}
h3{color:blue}
</style></head>
 <body>
 <h1>All XMML.com documents.</h1>
 <h3>Document Title: XMML.com Training Courses</h3>
 <p>Document Status: outdated</p>
 <h2>Document Title: XMML.com Training Courses</h2>
 <p>Document Status: final</p>
 <p>Document Author: Camilla Zukowski</p>
 <p>Document Content: December 2002 content.</p>
 <h3>Document Title: XMML.com Training Courses</h3>
 <p>Document Status: draft</p>
 <h2>Document Title: XMML.com Consultancy Services</h2>
 <p>Document Status: final</p>
 <p>Document Author: Paul Hartington</p>
 <p>Document Content: April 2003 consultancy services
 information. </p>
 </body>
</html>
```

Figure 12.1 shows Listing 12.3 displayed in the Internet Explorer 5.5 browser.

To use only `xsl:if` elements to achieve output like this can be a little clumsy at times. Let's modify the output by using the `xsl:choose` element in an XSLT transformation.



**Figure 12.1** Differential display of documents depending on the value of the version attribute.

## The `xs1:choose` Element

The `xs1:choose` element enables you to make multiple choices about how nodes should be processed. For each specified test, you use an `xs1:when` element with a test attribute. If you want to create a default type of processing when none of the tests on `xs1:when` elements is satisfied, you can use an `xs1:otherwise` element.

Listing 12.4 shows a modified stylesheet. When the value of the `version` attribute has the value `final`, you will output the full document as in the earlier example. This time, you will create different outputs when the value of the `version` attribute is `draft` or `outdated`.

### Listing 12.4 Documents2.xsl: A Modified XSLT Stylesheet Using `xsl:choose`

---

```

<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/" >
<html>
<head>
<title>XMML.com Documents</title>
<style type="text/css">
h2{color:red}
h3{color:blue}
</style>
</head>
<body>
<h1>All XMML.com documents.</h1>
<xsl:apply-templates select="//Document" />

</body>
</html>
</xsl:template>

<xsl:template match="Document">
<xsl:choose>
<xsl:when test="@version='final'">
<h2>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h2>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
<p>Document Author:<xsl:text> </xsl:text><xsl:value-of select=
 "Author" /></p>
<p>Document Content:<xsl:text> </xsl:text><xsl:value-of
 select="Content" /></p>
</xsl:when>
<xsl:when test="@version='draft'">
<h3>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h3>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
</xsl:when>

```

**Listing 12.4** Continued

---

```
<xsl:otherwise >
<!-- The version attribute is "outdated". -->
<!-- Do nothing -->
</xsl:otherwise>
</xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

Two `xsl:when` elements are nested inside the `xsl:choose` element. The content of each `xsl:when` element is processed when a Document element satisfies the test in the `test` attribute of the `xsl:when` element.

Listing 12.5 shows the modified HTML output document. Notice that the outdated document does not appear in this output document because the `xsl:otherwise` element does nothing.

**Listing 12.5** Documents2.html: The Output of the Stylesheet Using `xsl:choose`

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
 <xsl:output method="html"
 indent="yes" />
 <xsl:template match="/" >
 <html>
 <head>
 <title>XMML.com Documents</title>
 <style type="text/css">
 h2{color:red}
 h3{color:blue}
 </style>
 </head>
 <body>
 <h1>All XMML.com documents.</h1>
 <xsl:apply-templates select="//Document" />

 </body>
 </html>
 </xsl:template>
```

**Listing 12.5**    Continued

---

```

<xsl:template match="Document">
<xsl:choose>
<xsl:when test="@version='final'">
<h2>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h2>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
<p>Document Author:<xsl:text> </xsl:text><xsl:value-of select=
 "Author" /></p>
<p>Document Content:<xsl:text> </xsl:text><xsl:value-of
 select="Content" /></p>
</xsl:when>
<xsl:when test="@version='draft'">
<h3>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h3>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
</xsl:when>
<xsl:otherwise >
<!-- The version attribute is "outdated". -->
<!-- Do nothing -->
</xsl:otherwise>
</xsl:choose>
</xsl:template>

</xsl:stylesheet>

```

The output of each type of document is not in any particular order—it merely reflects document order in Listing 12.1. In some situations, you would want to sort the order of elements in the output document.

## Sorting Output

XSLT provides the `xsl:sort` element to enable you to sort output from a transformation into the order that you want. If you want to sort by two criteria, you can nest `xsl:sort` elements inside each other.

Listing 12.6 uses the `xsl:sort` element to output all documents that are final before those that are drafts. Outdated documents are governed by the `xsl:otherwise` element and are not output.

### Listing 12.6 Documents3.xml: Sorting Documents in Descending Alphabetical Order by version Attribute Value

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/" >
<html>
<head>
<title>XMML.com Documents</title>
<style type="text/css">
h2{color:red}
h3{color:blue}
</style>
</head>
<body>
<h1>All XMML.com documents.</h1>
<xsl:apply-templates select="//Document" >
<xsl:sort select="@version"
 order="descending"
 data-type="text"/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Document">
<xsl:choose>
<xsl:when test="@version='final'">
<h2>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h2>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
<p>Document Author:<xsl:text> </xsl:text><xsl:value-of select=
 "Author" /></p>
<p>Document Content:<xsl:text> </xsl:text><xsl:value-of
 select="Content" /></p>
</xsl:when>
<xsl:when test="@version='draft'">
<h3>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h3>
```

**Listing 12.6** Continued

---

```

<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
</xsl:when>
<xsl:otherwise >
 <!-- The version attribute is "outdated". -->
 <!-- Do nothing -->
</xsl:otherwise>
</xsl:choose>
</xsl:template>

</xsl:stylesheet>

```

The key change to the XSLT stylesheet is in the `xsl:apply-templates` element in the main template.

```

<xsl:apply-templates select="//Document" >
<xsl:sort select="@version"
 order="descending"
 data-type="text" />
</xsl:apply-templates>

```

Instead of having an empty `xsl:apply-templates` element, an `xsl:sort` element is nested inside the `xsl:apply-templates` element. You specify the sort key using the `select` attribute. The order—ascending or descending—is specified using the `order` attribute. The data type—text, number, or QName—is specified using the `data-type` attribute.

Listing 12.7 shows the sorted output. The two documents that are final come before the single draft document because you are sorting in descending order and because final comes after draft alphabetically.

**Listing 12.7** Documents3.html: Output Sorted by Value of the version Attribute

---

```

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

 <title>XMML.com Documents</title><style type="text/css">
h2{color:red}
h3{color:blue}
</style></head>

```



**Listing 12.7** Continued

```
<body>
 <h1>All XMML.com documents.</h1>
 <h2>Document Title: XMML.com Training Courses</h2>
 <p>Document Status: final</p>
 <p>Document Author: Camilla Zukowski</p>
 <p>Document Content: December 2002 content.</p>
 <h2>Document Title: XMML.com Consultancy Services</h2>
 <p>Document Status: final</p>
 <p>Document Author: Paul Hartington</p>
 <p>Document Content: April 2003 consultancy services
 information. </p>
 <h3>Document Title: XMML.com Training Courses</h3>
 <p>Document Status: draft</p>
</body>
</html>
```



**Note** The `xs1:sort` element can also be used with the `xs1:for-each` element. That usage is not considered in this book, however.

You can sort on more than one criterion using multiple `xs1:sort` elements.

## Multiple Sorts

When you use multiple `xs1:sort` elements, you put the major sort criterion first, then the next most important, and so on if you are using more than two sort criteria.

The `xs1:sort` element must be a direct child element of the `xs1:apply-templates` element (in this example). It is an error to nest `xs1:sort` elements inside each other.

Listing 12.8 shows the XSLT stylesheet modified so that it outputs all (here, both) the final documents first, but those final documents are themselves sorted in ascending alphabetical order.

**Listing 12.8** Documents4.xsl: Applying Two Sorts to the Source Document

---

```

<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/" >
<html>
<head>
<title>XMML.com Documents</title>
<style type="text/css">
h2{color:red}
h3{color:blue}
</style>
</head>
<body>
<h1>All XMML.com documents.</h1>
<xsl:apply-templates select="//Document" >
<xsl:sort select="@version"
 order="descending"
 data-type="text" />
<xsl:sort select="Title"
 order="ascending"
 data-type="text" />
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Document">
<xsl:choose>
<xsl:when test="@version='final'">
<h2>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h2>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version" /></p>
<p>Document Author:<xsl:text> </xsl:text><xsl:value-of select=
 "Author" /></p>
<p>Document Content:<xsl:text> </xsl:text><xsl:value-of
 select="Content" /></p>
</xsl:when>

```

**Listing 12.8** Continued

---

```

<xsl:when test="@version='draft'">
<h3>Document Title:<xsl:text> </xsl:text><xsl:value-of select=
 "Title" /></h3>
<p>Document Status:<xsl:text> </xsl:text><xsl:value-of select=
 "@version"/></p>
</xsl:when>
<xsl:otherwise >
<!-- The version attribute is "outdated". -->
<!-- Do nothing -->
</xsl:otherwise>
</xsl:choose>
</xsl:template>

</xsl:stylesheet>

```

Because “Consultancy Services” is alphabetically before “Training Services,” the consultancy services document appears first in the output document, as you can see in Listing 12.9.

**Listing 12.9** Documents4.html: The Output Document After Applying Listing 12.8

---

```

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

 <title>XMML.com Documents</title><style type="text/css">
h2{color:red}
h3{color:blue}
</style></head>
 <body>
 <h1>All XMML.com documents.</h1>
 <h2>Document Title: XMML.com Consultancy Services</h2>
 <p>Document Status: final</p>
 <p>Document Author: Paul Hartington</p>
 <p>Document Content: April 2003 consultancy services
 information.</p>
 <h2>Document Title: XMML.com Training Courses</h2>
 <p>Document Status: final</p>
 <p>Document Author: Camilla Zukowski</p>
 <p>Document Content: December 2002 content.</p>

```

**Listing 12.9** Continued

---

```
<h3>Document Title: XMML.com Training Courses</h3>
<p>Document Status: draft</p>
</body>
</html>
```

## Summary

In this lesson, you were introduced to how the `xsl:if` and `xsl:choose` elements can be used to carry out conditional processing when transforming XML documents. In addition, you learned how to use the `xsl:sort` element to sort nodes according to single and multiple criteria.

# LESSON 13

## Styling XML with CSS



*In this lesson, you will learn how to associate a Cascading Style Sheets (CSS) style sheet with an XML document and use CSS rules to style XML documents. Combining CSS and XSLT to style XML documents is also discussed and demonstrated.*

### Cascading Style Sheets and XML

Cascading Style Sheets (CSS) is a styling technology that uses non-XML syntax to style elements in markup languages such as HTML. It also is suitable for some tasks in the styling of XML.

CSS style sheets can be used on their own with XML or can be used with XML and XSLT. Both uses are described and illustrated in this lesson.



**Note** In CSS, the words *style sheet* are two separate words. In XSLT, the term *stylesheet* is a single word. This difference in spelling arose because these of the separate initial development of these two technologies at W3C.

First let's look at some of the background that explains why CSS was invented and what problems it solves.

## Separating Content and Presentation

Having a site-wide coherent visual appearance is desirable for all but the most anarchic Web sites. A coordinated style with good design characteristics can make a positive impression on visitors to a site. On that basis alone, it is useful to be able to easily create a style that applies across a whole Web site. But there were difficulties in doing that without CSS.

One of the problems with HTML—and one of the problems that led to the development of XML—was that content and presentation were intertwined. For example, an `h1` element indicated a heading, but, almost inevitably, it also indicated a larger size for the contained text.

In the early days of HTML, it was common for the same person to create all aspects of an HTML Web page. The Web page creator carried out design tasks and content tasks pretty much seamlessly. When a page is created, and particularly when a site consists of a small number of pages, one-person authoring using HTML (with content and presentation intertwined) can work well. However, problems begin when a site grows and when it must be updated, perhaps by a different person and perhaps with a site-wide change in color or other style aspects. If style information is contained in HTML tags alone, updating style information in every individual page on a large site becomes a tedious, time-consuming, and expensive process.

Another factor that is increasingly relevant is that many Web pages, particularly on larger Web sites, are generated dynamically. If styling information was applied to each individual element in an HTML page created dynamically, the problems of updating potentially become even more severe. Any change in styling must be made within dynamically created HTML code contained, for example, in a Java servlet. This could mean even more time in amending styling information at potentially greater cost. Separating styling information for the site into a separate CSS stylesheet enables the Java code (or code in another language) to be shorter and more easily maintained.

Given these factors, a way obviously must be found to update styling information across a site efficiently, speedily, and not too expensively. By separating styling information into CSS style sheets, any necessary

changes in content can be made independent of styling changes. Equally, styling can be changed without changing content or having to individually edit each HTML page. Taken together, these factors can save a lot of time and money in the ongoing costs of supporting a Web site.

Let's move on to look at how you can associate an external CSS style sheet with an XML document.

## Associating a Stylesheet

The `xml-stylesheet` processing instruction is used to associate an XML document with a CSS style sheet or with an XSLT stylesheet of the type you saw in earlier chapters.



**Tip** Place the `xml-stylesheet` processing instruction in the prolog of the XML document, in the line immediately after the XML declaration, if you used one.

The general form of the necessary processing instruction is as follows:

```
<?xml-stylesheet href="CSSStyleSheet.css" type="text/css" ?>
```

The second part of the `xml-stylesheet` processing instruction consists of two *pseudoattributes* (these aren't true attributes because they aren't associated with an element)—the `href` and `type` pseudoattributes.

An XML processor can use this information to recognize that there is a CSS file of type `text/css`, named `CSSStyleSheet.css`, which is associated with the XML document.

## Using CSS Rules with XML

A CSS style sheet is made up of *rules*.



A *rule* is the association of an element type name, a class, or other part of an XML document with a CSS declaration.

If an XML document includes a `title` element and you want the text content to be displayed at a font size of 24 points, you could write this:

```
title {font-size:24pt}
```

## CSS Syntax

Whether it is internal (in a `style` element) or external, a CSS style sheet consists of rules. The following CSS rule associates the Arial font of size 36 with the `h1` element:

```
h1 {font-family:Arial, sans-serif;
font-size:36;}
```

The part of the rule outside the curly brackets is called a *selector*. Selectors can be grouped by separating them using commas. The following rule would apply to both `p` and `li` elements:

```
p, li {font-family:"Times Roman", serif;
font-size:12;}
```

A selector may consist of one or more element type names or may be more focused and include only certain elements that have a `class` attribute of a particular value. Imagine applying a rule to `p` elements with a `class` attribute with the value `confidential`:

```
<p class="confidential" ...>Some text<p>
```

To make the text red in color, you can use this rule:

```
p.confidential {color:#FF0000;}
```

The period in `p.confidential` separates the element type name, `p`, from the value of the `class` attribute—in this case, `confidential`.



Inside the curly brackets, you can have one or more *declarations*. A declaration such as this one consists of a property—in this example, `font-family`, separated by a colon from its value or values:

```
font-family:Arial,sans-serif;
```

When more than one value exists in a declaration, they are separated by a comma from each other. The end of a declaration is signaled by a semi-colon character.

## Limitations of CSS Styling

When CSS style sheets are used in the absence of XSLT stylesheets, they have significant limitations with XML documents.

Suppose that you had an XML document with the following structure:

```
<FaultReports>
<Fault status="resolved">
Internet Explorer will not save HTML files correctly.
</Fault>
<Fault status="ongoing">
Noisy telephone line to West building. Intermittent loss of
 Internet connection.
</Fault>
<Fault status="resolved">
Modem fails to dial external numbers correctly.
</Fault>
</FaultReports>
```

For example, imagine that you wanted to sort the data so that all ongoing fault reports were grouped and were followed in the displayed document by all resolved fault reports. CSS alone doesn't enable you to carry out that restructuring of an XML document.

Suppose also that you want to display an image to illustrate something about our data. CSS cannot link images.



**Note** Some XML application languages, such as Scalable Vector Graphics (see Chapter 15, "Presenting XML Graphically—SVG"), do have functionality to display vector and bitmap images.

## Some Examples Using CSS Styling

If your XML data is structured as you want to display it, you can effectively use CSS to display XML content.

Listing 13.1 shows a CSS style sheet to display heading information in red and paragraph text in a smaller, black font.

### **LISTING 13.1** Reports.css: A CSS Style Sheet to Display Reports Stored in XML

---

```
header{font-family:Arial, sans-serif;
 font-size:18;
 color:blue;}
content{ {font-family:"Times New Roman", serif;
 font-size:12;
 color:black;}
```

The XML source document is shown in Listing 13.2.

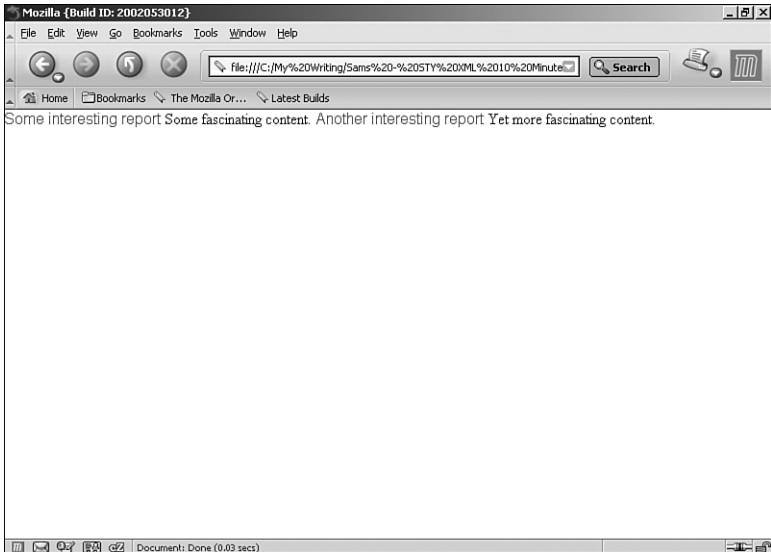
### **LISTING 13.2** Reports.xml: A Brief XML Data Store of Reports

---

```
<?xml version='1.0'?>
<?xml-stylesheet href="Reports.css" type="text/css" ?>
<Reports>
 <Report>
 <header>Some interesting report</header>
 <content>Some fascinating content.</content>
 </Report>
 <Report>
 <header>Another interesting report</header>
 <content>Yet more fascinating content.</content>
 </Report>
</Reports>
```

The `xml-stylesheet` processing instruction associates the XML document with the Reports.css CSS style sheet.

Figure 13.1 shows the onscreen appearance when Listing 13.2 is displayed in the Mozilla 1.0 browser.



**FIGURE 13.1** Basic CSS display with all element content on same line.

The display in Figure 13.1 is pretty rudimentary. It doesn't even split headings (in blue) into separate lines from the content of an individual report. That limitation in XML documents occurs because some effects seemingly produced by CSS in HTML Web pages are the result of the characteristics of the HTML elements, such as the `h1` element or the `p` element, which automatically create a block display.

Listing 13.3 shows how you can modify the CSS style sheet, using the `display` property, to display the content of each element on a separate line.

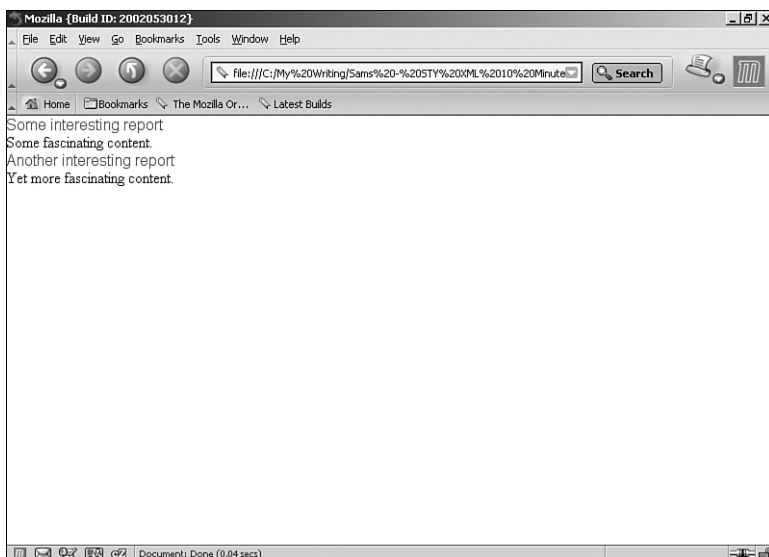
### **LISTING 13.3** Reports2.css: Adding Block Display to the CSS Style Sheet

```
header{font-family:Arial, sans-serif;
 font-size:18;
 color:blue;
 display:block;}
```

**LISTING 13.3**    Continued

```
content{ {font-family:"Times New Roman", serif;
 font-size:12;
 color:black;
 display:block;}
```

After the XML file has been amended to point to the revised CSS file, the onscreen appearance in the Mozilla browser should look like Figure 13.2. The altered XML document, Reports2.xml, is available in the code download.



**FIGURE 13.2**    Appearance after the CSS has been modified to specify block display.

If you want to target only browsers that fully support CSS2, you can use absolute positioning to further refine the appearance onscreen. But, in practice, you will run into a problem when you want to reorder elements or display certain elements only. Of course, you can begin to modify the source XML document and add `class` attributes to enable you to hide a

class by specifying `display:none` in a rule. However, taking that approach is ill advised. You are beginning to modify the structure of your content to control presentation, which is where HTML caused difficulties. It's an approach that runs against the principles that XML was designed to follow.

Rather than stretching CSS and modifying XML documents to accommodate CSS's limitations, it makes more sense, at least for the moment, to use CSS in conjunction with XSLT.

## Using CSS with XSLT

If CSS (at least at Level 2) can't do all that you want with your XML data, you need to explore alternative approaches to displaying that data for some uses. One productive possibility is to use CSS and XSLT together with a source XML data store.

## Using XSLT and CSS with HTML Output

If you use both XSLT to create HTML output documents (Web pages) and CSS styling together with the HTML, all four ways of using CSS to style HTML documents are, in principle, available:

- Linking to an external CSS style sheet using the `link` element
- Using the `@import` directive
- Using the `style` element
- Styling individual HTML elements

One of the reasons for using CSS is that it makes it easier to update styles site-wide, so you most likely will want to use an external CSS style sheet. An external CSS style sheet can be accessed using the `link` element or the `@import` directive. The `link` element is used in the example that follows.

Suppose that you wanted to use the CSS style sheet shown in Listing 13.4 site-wide in an HTML site whose pages are generated using XSLT.

**LISTING 13.4**    MySite.css: A Brief CSS Style Sheet

---

```
/* This style sheet is to be used site-wide with HTML pages
 generated using XSLT */
h1 {font-family:Arial, sans-serif;
 font-size:28;
 color:#FF0000;}

h2 {font-family:Arial, sans-serif;
 font-size:20;
 color:#0000FF;}

p {font-family:"Times New Roman", serif;
 font-size:16;
 color:black;}

li {font-family:"Times New Roman", serif;
 font-size:14;
 color:#999999;}
```

The source XML document is shown in Listing 13.5.

**LISTING 13.5**    CSSInformation.xml: A Brief Data Store of Information About CSS

---

```
<?xml version='1.0'?>
<CSSInformation>
<Techniques>
<Technique>
The <link> element
</Technique>
<Technique>
The @import directive
</Technique>
<Technique>
The <style> element
</Technique>
<Technique>
Styling individual elements
</Technique>
</Techniques>
</CSSInformation>
```

Listing 13.6 is an XSLT stylesheet to create the desired HTML output document.

**LISTING 13.6** CSSInformation.xsl: An XSLT Stylesheet  
Creating a Link to a CSS Style Sheet

---

```
<?xml version='1.0'?>
<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
 >
<xsl:output method="html"
 indent="yes" />
<xsl:template match="/" >
<html>
<head>
<title>Techniques for using CSS in HTML</title>
<link rel="stylesheet" href="MySite.css" />
</head>
<body>
<h1>Techniques for using CSS in HTML</h1>
<h2>Techniques</h2>
<p>The following are the techniques available in CSS2 to use
 CSS.</p>

<xsl:apply-templates select="//Technique" />

</body>
</html>
</xsl:template>

<xsl:template match="Technique">
<xsl:value-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

Notice the `link` element in the head section of the literal result elements in the XSLT stylesheet. The `rel` and `href` attributes of the `link` element indicate that the file `MySite.css` (see Listing 13.4) is to be linked as the CSS style sheet for the HTML document.

Listing 13.7 shows the HTML output document.

**LISTING 13.7**    CSSInformation.html: The Result of the XSLT Transformation

---

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html;
 charset=utf-8">
 <title>Techniques for using CSS in HTML</title>
 <link rel="stylesheet" href="MySite.css">
</head>
<body>
 <h1>Techniques for using CSS in HTML</h1>
 <h2>Techniques</h2>
 <p>The following are the techniques available in CSS2
 to use CSS.</p>

 The <link> element

 The @import directive

 The <style> element

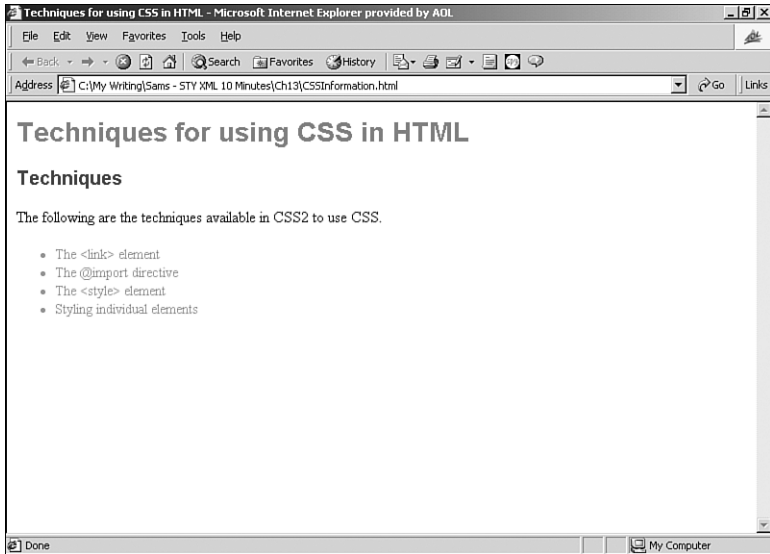
 Styling individual elements

</body>
</html>
```

Figure 13.3 shows the onscreen appearance. When you view the document onscreen, you can see by the size, font family, and color of the text that the CSS styling in MySite.css has been applied to the text in the HTML page.

Combining CSS with XSLT in this way maintains the separation of content and presentation in XML documents. At the same time, linking external CSS style sheets offers the maintenance benefits of CSS.





**FIGURE 13.3** An HTML document displayed using CSS but created using an XSLT transformation.

## Summary

This lesson discussed the need for Cascading Style Sheets (CSS) to ease maintenance of Web sites. It also described and demonstrated the way to link an XML document to an external CSS style. In addition, you learned about the styling of XML documents and the advantages of using CSS together with XSLT, for example, when elements need to be reordered.



## LESSON 14

# Linking in XML—XLink

*In this lesson, you will learn how to use the XML Linking Language (XLink) simple links and the XML Pointer Language (XPointer).*

## The XML Linking Language

The XML Linking Language (XLink)—and the associated XML Pointer Language (XPointer) specification, discussed later in this chapter—are intended to provide linking functionality for XML on the Web. This will provide XML-based linking functionality equivalent to HTML, but with significant enhancements. Relevant parts of XLink and XPointer have already been implemented in Scalable Vector Graphics (see Chapter 15, “Presenting XML Graphically—SVG”) and likely will be implemented in at least some other XML specifications.

The original vision for XML was that it should be used on the Web. However, in practice it has displaced HTML much more slowly than was originally envisioned. Perhaps a contributing factor to that was the lengthy delay in finishing development of the XLink and the (still unfinished) XPointer specifications.

XLink became a full W3C Recommendation in June 2001 (see [www.w3.org/TR/2001/REC-xlink-20010627/](http://www.w3.org/TR/2001/REC-xlink-20010627/)). At the time of this writing, four new XPointer Working Drafts have just been issued.

XLink provides *simple links*, which are similar to HTML hyperlinks, and *extended links* that introduce functionality beyond that provided in HTML. XPointer is a much more powerful fragment-identifier mechanism than HTML anchors.

First let’s look at HTML hyperlinks and their strengths and limitations. XPointer is discussed later in the chapter.

## XLink and HTML Hyperlinks

HTML hyperlinking has been immensely successful as a pivotal part of the World Wide Web. The Web is almost unimaginable without such hyperlinking functionality.

In HTML, the hyperlinking mechanism uses the `a` element. It is possible to use the `a` element to link externally to other HTML Web pages or other resources, or to link internally or externally to specified document fragments.

These simple mechanisms are immensely useful, but they do have limitations. For example, a link is expressed at one end only. If you click a link on page A and move to page B, the browser Back button typically allows you to link back to page A. But if you visit page B directly, there is likely no way to link to page A at all.

To take another example, if you want to link to a fragment of page B but the document author hasn't provided an anchor at the point that interests you, you can't link directly to that point. You are limited to linking to the Web page and providing instructions for what part of the document you want a user to scroll to.

XLink and XPointer were intended to address limitations such as these.

## Simple Links and Extended Links

XLink provides two significantly different types of links: *simple links* and *extended links*.

An XLink simple link behaves on its own very much like an HTML hyperlink. When used with XPointer, it is potentially much more flexible in linking to specified document fragments.

XLink extended links allow links to be created among more than two resources.

## XLink Jargon

XLink brings with it a lot of jargon, which is needed to precisely express what is happening with XLink extended links.

In XLink terminology, a *link* is simply an explicit association between two or more resources. The link is made explicit by an XLink *linking element*.



**Note** No elements exist in the XLink namespace—only attributes. An element in a non-XLink namespace that possesses XLink attributes is termed an XLink *linking element*.

So, an XLink simple link might look like this:

```
<myPrefix:myElement xlink:href="SomeResource.xml" ... />
```

The preceding code assumes that the XLink namespace URI, `www.w3.org/1999/xlink`, has been declared at an appropriate place in the document.

Using or following an XLink link is termed *traversal*. The resource that contains the XLink linking element is known as the *starting resource*, and the destination of the link is the *ending resource*.

A *local resource* is an XML element that participates in a link by virtue of having a linking element as its parent or being itself a linking element. A *remote resource* is addressed by means of a URI reference.

A link that has a *local starting resource* and a *remote ending resource* is termed *outbound*. XLink simple links, like HTML hyperlinks, are of this type.

XLink also allows two other types of arc: *inbound* and *third-party*. An inbound arc exists when the XLink linking element is expressed on a local resource—in other words, there is a local ending resource and a remote starting resource. A third-party link exists when the XLink linking element is expressed in neither the starting resource nor the ending resource. These types of links, which are XLink extended links, can be used to form link databases, also called *linkbases*.

## XLink Attributes

The XLink specification creates no new elements but does add attributes that are in the XLink namespace to elements in other XML application languages. Typically, the XLink namespace must be declared.

The `xlink:href` attribute specifies a URI for the remote resource. The type of an XLink—simple or extended—is expressed using the `xlink:type` attribute. For a simple link that replaces the local resource when the arc is traversed, you need only those two attributes:

```
<myPrefix:myElement xlink:type="simple"
xlink:href="someURI" >
```

The `xlink:show` attribute controls where the remote resource is displayed. The default value is `replace`. To display a resource in a new browser window, the `xlink:show` attribute has the value `new`.

The `xlink:actuate` attribute controls when the arc of the link is traversed. The default value is `onRequest`. To traverse an arc upon document loading, the `xlink:actuate` attribute has the value `onLoad`. At the time of this writing, no browser implements that feature.

An extended link has the `xlink:type` attribute with the value `extended`. An extended link may have `xlink:href`, `xlink:show`, and `xlink:actuate` attributes. In addition, it may have `xlink:title`, `xlink:resource`, `xlink:arc`, `xlink:arcrole`, `xlink:label`, `xlink:from`, and `xlink:to` attributes. The latter attributes will not be discussed further.

## XLink Examples

At the time of this writing, Internet Explorer browser has no support for XLink. The Netscape 6.x and Mozilla 1.x browsers support XLink simple links only.

Listings 14.1 and 14.2 show two brief XML documents that each contain a single XLink simple link.

**LISTING 14.1** Document1.xml: A Document with a Single XLink Simple Link

---

```
<?xml version='1.0'?>
<?xml-stylesheet href="BigText.css" type="text/css" ?>
<Document1 xmlns:xlink="http://www.w3.org/1999/xlink">
 <myPrefix:myElement xlink:href="Document2.xml"
 xlink:type="simple"
 xmlns:myPrefix="http://www.XMML.com/">
 Click here to go to Document 2.
 </myPrefix:myElement>
</Document1>
```

**LISTING 14.2** Document2.xml: A Document with Another Simple Link

---

```
<?xml version='1.0'?>
<?xml-stylesheet href="BigText.css" type="text/css" ?>
<Document1 xmlns:xlink="http://www.w3.org/1999/xlink">
 <myPrefix:myElement xlink:href="Document1.xml"
 xlink:type="simple"
 xmlns:myPrefix="http://www.XMML.com/">
 Click here to go back to Document 1.
 </myPrefix:myElement>
</Document1>
```

Notice that the XLink namespace is declared in each listing on the document element. For ease of display in the screenshots, the documents have been linked to a CSS style sheet shown in Listing 14.3.

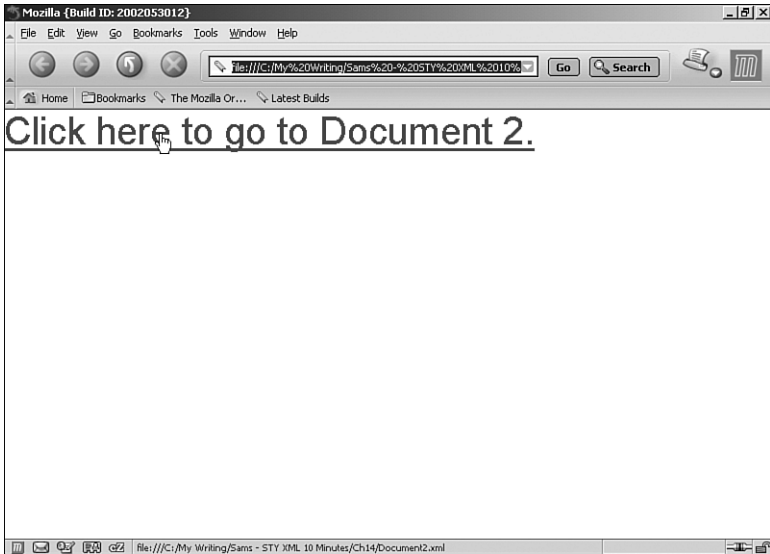
**LISTING 14.3** BigText.css: A CSS Style Sheet Controlling the Appearance of Text

---

```
myElement {
font-family:Arial, sans-serif;
font-size:30pt;
color:blue;
text-decoration:underline;
}
```

Without the CSS stylesheet, the XML text would be displayed as black, without underlining, and in the default font.

Figure 14.1 shows Listing 14.1 displayed in the Mozilla 1.0 browser. Note that the cursor changes to a pointing finger cursor over the linking text.



**FIGURE 14.1** Linking text in a simple XML document in the Mozilla browser.

If you wanted Listing 14.2 to display in a new browser window, you could simply add an `xlink:show` attribute with the value of `new`, as shown in Listing 14.4.

#### **LISTING 14.4** Document3.xml: A Link to Open a New Browser Window

```
<?xml version='1.0'?>
<?xml-stylesheet href="BigText.css" type="text/css" ?>
<Document1 xmlns:xlink="http://www.w3.org/1999/xlink">
<myPrefix:myElement xlink:href="Document2.xml"
 xlink:type="simple"
 xlink:show="new"
 xmlns:myPrefix="http://www.XMML.com/">
Click here to go to Document 2.
</myPrefix:myElement>
</Document1>
```

## XLink in SVG

One of the XML application languages that already implements XLink simple links is the W3C's Scalable Vector Graphics (SVG) specification. SVG 1.0 does not support XLink extended links. An XLink example in SVG is shown in Chapter 15.

Having looked at how you can link whole Web pages, let's move on to see how XPointer handles XML fragment identifiers.

## Document Fragments and XPointer

In HTML documents, the process of addressing a specific part of the document involves using anchors. This can be a useful mechanism, but it does have significant limitations.



**Caution** Immediately before this writing, the XPointer specification was split into four W3C Working Drafts. The description that follows is based on those drafts and, therefore, is potentially subject to change.

Let's suppose that you want to link to the HTML document shown in Listing 14.5. It includes many `br` tags to space out the anchors.

### **LISTING 14.5**    Anchors.html: An HTML Document with Anchors

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>Teach Yourself XML in 10 Minutes - Chapter 14</TITLE>
</HEAD>
<BODY>
<h1>
Linking in XML - XLink
</h1>

```



**LISTING 14.5** Continued

---

```

<p>First fascinating text.</p>

<p>Second fascinating text.</p>

<p>Third fascinating text.</p>

<p>The REALLY fascinating text.</p>

</BODY>
</HTML>

```

If you want to link from another HTML document to any of the specified anchors in Listing 14.5, writing a link is straightforward:

```
 LinkText
```

If you don't have write privileges on the target document, you have limited options. You can ask the Web page author to add an anchor at the point that you want to link to (often unrealistic in a rapidly growing Web), link to an anchor close to the real point of interest (if there is such an anchor), or accept that it isn't possible to link to a desired fragment in that page.

XPointer is designed to provide improved functionality for linking to document fragments in XML documents. A correctly written XPointer should be capable of identifying any arbitrary part of an XML document. How that part of the document is processed after it is identified depends on the application with which the XPointer processor is associated.

For example, suppose you wanted to link to a long XML document, perhaps a lengthy report, and display it in a browser or similar software at exactly the point of interest. You could write an XPointer to access any part of the document that is of interest without having write access to the target document.



**Note** At the time of this writing, no XPointer tools conform fully to the newly issued XPointer draft documents.

To understand how to do that, you should look first at the relationship between XPath (discussed in Chapter 9, “The XML Path Language—XPath”), and XPointer.

## XPointer and XPath

XPath models an XML document as a hierarchy of nodes. XPointer incorporates the concept of a node and also makes use of XPath functions, some of which were discussed in Chapter 9.

The XPath notion of a node is generalized in XPointer to include the additional notions of a *point* and a *range*. The following XML code snippet can help illustrate these concepts.

```
<text>Here is some text.</text>
```

All the text between the start and end tags of the `text` element is the value of a corresponding text node. The point exactly between the initial `H` and the first `e` of `Here` is a point. That type of point is a *character-point*.

A range could be the text string `is some`, with the *starting point* between the space character and the initial `i` of `is`, and the *ending point* between the final `e` of `some` and the following space character.

XPath operates using location paths. The output from one location step is used as the input for the following location step. XPointer works similarly. The *location set* returned by one pointer part is the start for processing by the next pointer part, if there is one.

XPointer incorporates XPath functions and adds functions that manipulate or return points and ranges.

Node tests in XPath have a counterpart—a *test*—in XPointer. An XPointer test can be applied to a node, point, or range.

## The XPointer Framework and Schemes

As this book was being written, a significant rewrite of the XPointer specification emerged after (from a public viewpoint) a static period of several months. The XPointer draft specification was split into four documents: the XPointer Framework Working Draft and three draft specifications that describe XPointer *schemes*—the `xpointer()`, `xmlns()`, and `element()` schemes.

An XPointer processor takes as input an XML document and a URI that includes a fragment identifier. The output from an XPointer processor is either identification of a part of the XML document corresponding to the fragment identifier or an error.

## The XPointer Framework

The XPointer Framework is a specification that defines the context for the `xpointer()`, `xmlns()`, and `element()` schemes.

## Scheme-Based XPointers

A scheme-based XPointer consists of one or more *pointer parts*, each of which is of the following general form:

*schemeName* (*characterSequence*)<sup>+</sup>

In other words, the *pointer part* begins with the scheme name followed by an opening parenthesis. A sequence of XML characters follows, and the pointer part is completed by a closing parenthesis.

So, an XPointer from the `xpointer()` scheme to select Chapter element nodes would look like this:

```
xpointer(//Chapter)
```

If the sequence of XML characters contains an unbalanced parenthesis character, that unbalanced parenthesis must be escaped.

As indicated by the `+` cardinality operator, an XPointer may use more than one scheme.

You will look at each of the proposed XPointer schemes in turn. First, let's look at the `xpointer()` scheme.

## The `xpointer()` Scheme

The `xpointer()` scheme is the most extensive of the XPointer schemes. In fact, it was originally envisaged as being the only scheme until technical issues led to the development of the `xmlns()` scheme (discussed later in this chapter).

The `xpointer()` scheme is associated with the namespace URI `www.w3.org/2001/05/XPointer`.



**Caution** The namespace URI given is associated with a Working Draft for the `xpointer()` scheme. It is possible that the namespace URI will change for the final version of the specification.

### Points in the `xpointer()` Scheme

The `xpointer()` scheme recognizes two types of points: a *node-point* and a *character-point*. Both types of points are defined in terms of a container node (the node within whose content the point is situated) and an index.

A node-point is a point between nodes that are children of the *container node* of the point. The index for a node-point lies between zero (the index of the node-point immediately before the first node in the container node) and the number of child nodes that the container node has.

A node-point corresponds conceptually to a gap between nodes. Because character-points occur within nodes, they are envisaged as occurring between the node-points before and after their container node.

The `self` axis and the `descendant-or-self` axis of a point location contain the point itself. The `parent` axis contains the container node of the point. The `ancestor` axis contains the container node and its ancestors. The `ancestor-or-self` axis also contains the point itself. All other axes are empty.

Points do not have an expanded name, and the string value of a point is the empty string.

## Ranges in the `xpointer()` Scheme

A range is defined by its *start point* and its *end point*. A range consists of all the XML structure and content between the start point and the end point of the range. The start point of a range need not be in the same node as the end point if the container node of the start point is of type root, element, or text. However, both points must be in the same XML document or external parsed entity. The start point must not come later in the document than the end point.

A special case arises when the start point and the end point are the same point. In that case, the range is referred to as a *collapsed range*.

A range does not have an expanded name. The string value of a range consists of the character content of text nodes inside the range.

The axes of a range are identical to the axes of its start point. The parent axis of the range contains the parent node of the start point.

The XPointer `start-point()` and `end-point()` functions can be used to navigate to the start point and end point, respectively, of a range.

## Functions in the `xpointer()` Scheme

The `xpointer()` scheme adds functions to those available from the XPath function library.

The `string-range()` function takes two required arguments (a location set and a string) and two optional arguments (numbers). The `string-range()` function returns a location for each occurrence of the string argument in the location set argument.

The `range()` function takes a location set argument and returns a location set. The `range()` function returns a covering range for each location in the argument location set.



**Note** A *covering range* is a range that totally encompasses a location. For a range, the covering range is the same range. For a point, the start point and end point of the covering range are the point itself. Definitions of other covering ranges are included in the XPointer specification.

The `range-inside()` function returns a location set and takes a single location set argument.

The `range-to()` function returns a range for each location in the context whose start point is returned by the `start-point()` function and whose end point is returned by the `end-point()` function.

The `start-point()` and `end-point()` functions respectively address the starting point and ending point of a range.

The `here()` function is meaningful only when the context is an XML document or an external parsed entity. If the expression being evaluated is in a text node inside an element node, the `here()` function returns the element node. Otherwise the `here()` function returns the node that directly contains the expression being evaluated.

The `origin()` function is meaningful only when it is processed in response to traversal of a link expressed in an XML document.

## Some xpointer() Scheme Examples

To locate an element node that has an ID attribute of value "CRES99", you can write this:

```
xpointer(id("CRES99"))
```

If you want to reference a range that includes the first and second chapters of a document, you could write this:

```
xpointer(//chapter[number='1']/range-to(//chapter[number='2']))
```

This assumes that the document contains chapter elements with a `number` attribute corresponding to the chapter number.

## The xmlns() Scheme

The xmlns() scheme is intended for use with the XPointer Framework to ensure correct interpretation of namespace prefixes in XPointers.

You might assume that using namespace prefixes would be possible using only the xpointer() scheme, but take a look at the following XML code snippet and think of the ambiguity it introduces:

```
<myPrefix:myElement
 xmlns:myPrefix="http://www.XMML.com/Namespcae">
 <AnElement>
 First piece of text.
 </AnElement>
</myPrefix:myElement
 xmlns:myPrefix="http://www.XMML.com/AnotherNamespcae">
 <AnElement>
 Second piece of text.
 </AnElement>
 <!-- Some content could go here -->
</myPrefix:myElement>
</myPrefix:myElement>
```

If you had the XPointer that follows, which XPointer location(s) is it intended to refer to?

```
xpointer(/myPrefix:myElement/AnElement)
```

Is it intended to refer to both AnElement elements? Or only one? If so, which? The myPrefix:myElement is declared to be associated with two different namespaces. For an XML processor, that doesn't cause difficulties because it uses the namespace URI, not the namespace prefix. But for XPointer you need to specify which namespace URI you are referring to.

To remove that ambiguity, the xmlns() scheme has been provided.

You can refer unambiguously to the outer myPrefix:myElement element using the following XPointer:

```
xmlns(myPrefix:http://www.XMML.com/Namespcae)
 xpointer(/myPrefix:myElement/AnElement)
```

Or, you can refer unambiguously to the inner one using this:

```
xmlns(myPrefix:http://www.XMML.com/AnotherNamespcae)
 xpointer(/myPrefix:myElement/AnElement)
```

Remember that an XML processor uses the expanded name rather than the namespace prefix for processing. So, if you want to access both `AnElement` elements, you could use both `xmlns()` pointer parts:

```
xmlns(a:http://www.XMML.com/Namespce)
 xmlns(b:http://www.XMML.com/AnotherNamespce)
```

You could use `a` or `b` as the namespace prefix in further pointer parts using the `xpointer()` scheme.

## The `element()` Scheme

The `element()` scheme is intended to be used with the `XPointer` Framework to provide basic addressing of elements in XML documents.

The `element()` scheme can use two forms of syntax: a name or a child sequence.

Suppose you had the following XML document:

```
<myDocument>
<Introduction>Some text</Introduction>
<MainText>Some main text</MainText>
<Postscript>Some postscript text</Postscript>
</myDocument>
```

You could select the `MainText` element by name using this line:

```
element(/MainText)
```

Or, you could select it as a child sequence using this code:

```
element(/1/2)
```

The syntax of the child sequence is to be understood as follows. The initial `/` character indicates that the root node is the initial context location. The `1` indicates that you are selecting the first element child of the root node—in this case, the `myDocument` element node. The next `/` character is a separator. The `2` indicates that you are selecting the second element child node of the `myDocument` node—in this case, the `MainText` element node.



## Summary

In this lesson, you were introduced to the XML Linking Language and learned how to use XLink simple links. The chapter also described the XML Pointer Language and introduced the characteristics of the XPointer Framework and the `xpointer()`, `xmlns()`, and `element()` schemes.



## LESSON 15

# Presenting XML Graphically— SVG

*In this lesson, you will be introduced to Scalable Vector Graphics (SVG), an XML application language that expresses two-dimensional vector graphics.*

## What Is SVG?

SVG is an XML application language that is intended to replace many uses of bitmap graphics on the Web and provide a vector graphics standard that is not vendor-specific (compare Microsoft's VML) and that is open source (compare Macromedia's Flash/SWF). SVG is written in XML-compliant syntax.

Version 1.0 of SVG became a W3C Recommendation in September 2001. At the time of this writing, version 1.1 of SVG is a W3C Candidate Recommendation ([www.w3.org/TR/SVG11/](http://www.w3.org/TR/SVG11/)) and will provide modularization of SVG so that SVG can be used on mobile browsers as well as on traditional desktop browsers.

SVG code is always well-formed SVG and can be validated against a publicly available DTD. SVG has elements for displaying text—`text` and `tspan`—as well as several elements that represent commonly used graphics shapes—for example, `rect`, `circle`, `ellipse`, and `polygon`. A `path` element can be used to represent any arbitrary two-dimensional graphics shape.

An SVG shape has its onscreen position, its dimensions, and its style information specified by attributes. The following code specifies a circle shape of radius 50 pixels filled with green and having a red outline (*stroke* in SVG jargon):

```
<circle cx="100px" cy="100px" r="50px"
style="fill:#00FF00; stroke:#FF0000; stroke-width:3" />
```

Alternatively, styling information for an SVG image or Web page can be contained in an internal style sheet (contained in a `style` element) or in an external CSS style sheet, referenced using the `xml-stylesheet` processing instruction. An internal style sheet contains non-XML text, so it must be contained in a CDATA section:

```
<style type="text/css">
<![CDATA[
/* style rules go here. */
]]>
</style>
```

In addition to static shapes and text, SVG provides five animation elements—`set`, `animate`, `animateMotion`, `animateColor`, and `animateTransform`—that singly or combined can produce an essentially unlimited number of animation effects.

SVG code is XML, so data stored as generic XML can be transformed into SVG using an XSLT stylesheet. This allows SVG charts to be created to express bar charts, line charts, and so on. Another growing use of SVG is to express map data.

In addition to containing SVG elements that define graphics shapes and text, SVG can be used to display external vector or bitmap images. For example, the following SVG code could be used to display a PNG image called `myBitmap.png`:

```
<image xlink:href="myBitmap.png" x="240px" y="90px"
width="350px" height="125px" />
```

In addition, SVG enables the scripting of a Document Object Model (DOM), which incorporates the DOM Level 2 (to be discussed in Chapter 16, “The Document Object Model,” and Chapter 17, “The Document

Object Model—2). Script code may be contained in a script element, as shown here:

```
<script type="text/javascript" >
[[CDATA[
// JavaScript code is not XML and needs to be in a
// CDATA section.
]]>
</script>
```

Or, it may be referenced in an external JavaScript or ECMAScript file:

```
<script type="text/javascript" xlink:href="myJavaScript.js" />
```

SVG also includes several powerful bitmap filters that, to take a simple example, can add a drop shadow to SVG text or shapes.

## Advantages of SVG

If you were around in the early days of the Web, you might remember the excitement as newcomers to HTML were able to learn the new technology rapidly because the HTML source code was always accessible in a Web browser. SVG offers the same advantage: A student of SVG can study the source code of an interesting or impressive graphic and need not deal with a steep learning curve.

The Adobe SVG Viewer can be downloaded from [www.adobe.com/svg/viewer/install/main.html](http://www.adobe.com/svg/viewer/install/main.html) and is a plug-in for conventional Web browsers. The SVG source code of an image displayed in the Adobe SVG Viewer is accessed by simply right-clicking the SVG image and selecting the View Source option.

The Batik standalone SVG viewer is available for download from <http://xml.apache.org/batik/>. Batik can also be used in Java applications to dynamically create and display SVG.

SVG can be combined with other XML application languages. New-generation browsers such as the X-Smiles browser ([www.x-smiles.org](http://www.x-smiles.org)) enable multiple XML languages to be combined into XML-based Web pages. For example, you could use SVG images within an XHTML Web page that also contains forms using the XML-based XForms specification.

An alternative approach is to create all-SVG Web pages, such as those you can see at [www.XMML.com/](http://www.XMML.com/).

## Creating SVG

SVG is XML. In principle, it can be created by any text editor. Of course, having syntax checking for well-formedness and color highlighting is an improvement over an editor such as Windows Notepad.

Listing 15.1 shows a simple SVG document that animates some text from a position offscreen onto the screen shortly after the document loads.

### LISTING 15.1 HelloVector.svg: An Animated Greeting Expressed in SVG

---

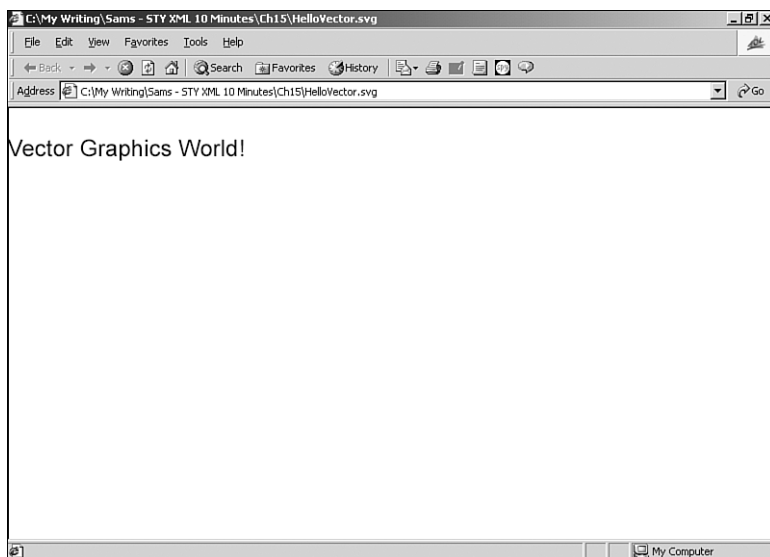
```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg>
<text x="-320" y="50" style="font-family:Arial; font-
size:24;">
Hello Vector Graphics World!
<animate attributeName="x" from="-320" to="20"
begin="1s" dur="3s" fill="freeze" />
</text>
</svg>
```

The XML declaration and the DOCTYPE declaration should be familiar from earlier chapters in this book. The document element is an `svg` element. Remember that SVG is XML, so element type names are case sensitive. The `svg` element must be written using lowercase characters only.

An SVG text element is used to contain the short message. One of the SVG animation elements is used to animate the text from a position just offscreen to the left onto the screen, starting one second after the document loads.

This *declarative animation* provides powerful and flexible animation facilities in SVG, which don't need to use scripting languages for many common effects. However, SVG has the flexibility to add script code to augment animation effects when it is appropriate.

Figure 15.1 shows the onscreen appearance partway through the animation of the text.



**FIGURE 15.1** A simple SVG document that animates text onscreen.

To view the SVG content onscreen, you need either a Web browser that has the Adobe SVG plug-in installed (see [www.Adobe.com/svg/](http://www.Adobe.com/svg/) for further details) or a dedicated SVG viewer such as Batik (see <http://xml.apache.org/batik/> for further information). Some of the examples in Chapters 16 and 17 use SVG to illustrate programmatic control of the XML Document Object Model, so you will find it useful to install an SVG viewer.

Examples of SVG used in geographical mapping can be seen at [www.carto.net/projects/](http://www.carto.net/projects/).

Another approach to creating SVG is to use a drawing tool with SVG export capabilities. For example, Jasc WebDraw is a dedicated SVG

drawing tool. More information is available at [www.Jasc.com](http://www.Jasc.com). Other well-known vector-drawing tools such as Adobe Illustrator (version 9 and onward, [www.Adobe.com](http://www.Adobe.com)) and CorelDraw (version 10 and onward, [www.Corel.com](http://www.Corel.com)) have SVG export facilities. Macromedia Freehand does not support SVG export at the time of this writing.

Because SVG is XML, it can be generated dynamically from XML data stores on the server before being transmitted to an SVG-enabled Web browser. Among the server-side tools that can be used to generate SVG dynamically are XSLT, Java, and Perl.

## Some SVG Examples

This section shows a few short examples of SVG code, including how to use SVG to create a rollover and how XLink is used in SVG.

### SVG Rollovers

In a conventional HTML Web page, it is necessary to use JavaScript to produce rollover effects. In SVG, rollover effects can be produced using SVG declarative syntax alone. Listing 15.2 shows a simple rollover with a message about SVG.

---

**LISTING 15.2** Mouseover.svg: A Message About SVG

---

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg>
<rect id="myRect" x="20" y="30" rx="10"
 ry="10" width="250" height="50"
 style="fill:white; stroke:blue; stroke-width:4">
<set attributeName="fill" from="white" to="#FFFF00"
 begin="mouseover" end="mouseout" />
</rect>
<text x="35" y="65"
 style="fill:blue; stroke:none; font-family:Arial,sans-serif;
 font-size:28; pointer-events:none " visibility="visible">
```

**LISTING 15.2** Continued

---

```
<animate begin="myRect.mouseover"
 attributeName="visibility" from="visible"
to="hidden" dur="0.1s" fill="freeze" />
<animate begin="myRect.mouseout"
 attributeName="visibility" from="hidden"
to="visible" dur="0.1s" fill="freeze" />
SVG is cool!
</text>
<text x="35" y="65"
 style="fill:red; stroke:none; font-family:Arial,sans-serif;
 font-size:28; pointer-events:none; visibility:hidden;">
<animate begin="myRect.mouseover"
 attributeName="visibility" from="hidden"
to="visible" dur="0.1s" fill="freeze" />
<animate begin="myRect.mouseout"
 attributeName="visibility" from="visible"
to="hidden" dur="0.1s" fill="freeze" />
SVG is RED HOT!
</text>
</svg>
```

When the document loads a simple text message, `SVG is Cool!` is visible against a white background inside a rectangle. When the rectangle is rolled over, its fill color changes to yellow, the first text message is hidden, and the message `SVG is RED HOT!` is displayed. The change in text is achieved by animating the `visibility` property of the two text elements in the document.

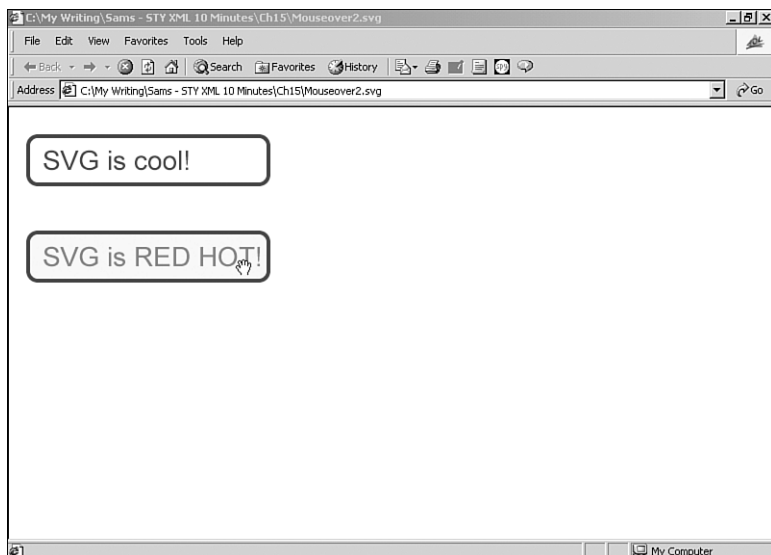
Figure 15.2 shows a composite image that includes both the rolled-over and unrolled-over versions of the rectangle.

## **XLink Links in SVG**

SVG uses XLink linking mechanisms to link to external resources and uses a subset of XPointer to address fragments in the same SVG document.

Listing 15.3 shows an example of using XLink in an SVG document.





**FIGURE 15.2** Rolled-over and unrolled-over versions of the rectangle.

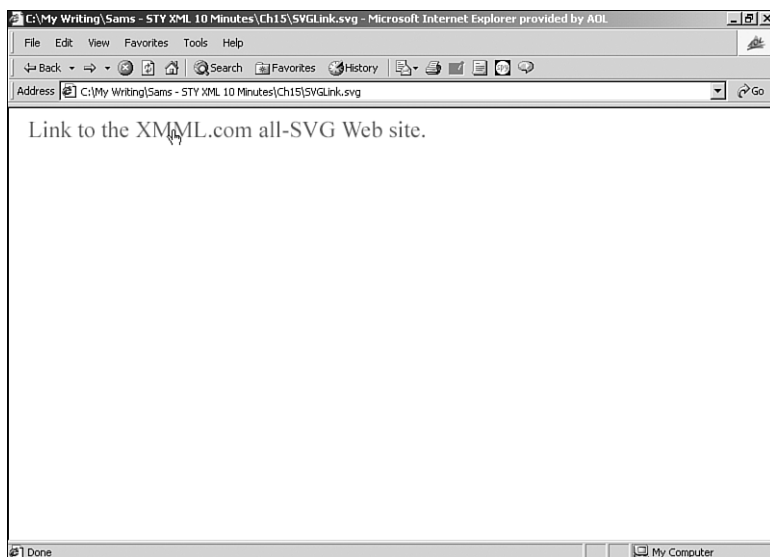
### LISTING 15.3 SVGLink.svg: An Example of Using an XLink Link in SVG

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg>
<rect x="0" y="0" width="100%" height="100%"
style="fill:#FFFFFF" />
<a xlink:href="http://www.XMML.com/" >
<text x="20" y="30"
style="fill:#666666; stroke:none; font-family:
'Times New Roman', serif; font-size:24" >
Link to the XMML.com all-SVG Web site.
</text>

</svg>
```

The SVG `a` element uses an `xlink:href` attribute to specify the resource to be traversed to. As you can see in Figure 15.3, rolling over the text

causes a pointing-finger cursor to appear. Clicking the text links to the [www.XMML.com](http://www.xmml.com) all-SVG Web site.



**FIGURE 15.3** An XLink hyperlink to an external resource.

## Using XPointer to Reference Definitions

SVG documents, other than very short ones, likely will include a definitions section contained in a `defs` element.

Listing 15.4 shows an example of using a *bare names* XPointer (now called *shorthand form* in the XPointer drafts that came out after the SVG 1.0 Recommendation was completed) to reference the definition of an SVG filter that applies a drop shadow to the text.

### LISTING 15.4 Reference.svg: Applying an SVG Filter on Mouseover

---

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg>
```

**LISTING 15.4** Continued

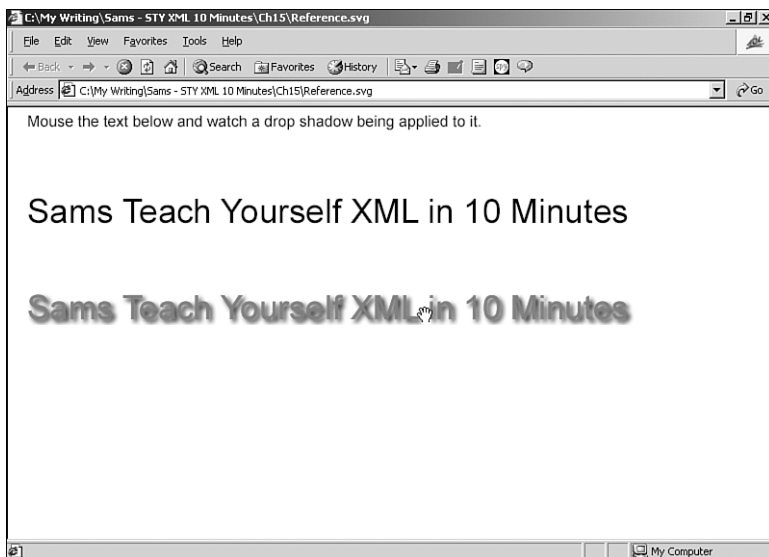
---

```

<defs>
<style type="text/css">
<![CDATA[
text {font-family:Arial, sans-serif;
font-size:16;
fill:black;
stroke:none;
}
text.big {font-family:Arial, sans-serif;
font-size:35;
fill:black;
stroke:none;
}
]]>
</style>
<filter id="myFilter" width="140%" height="140%" y="-.20%">
<feGaussianBlur in="SourceAlpha" stdDeviation="2.5"
 result="Blur" />
<feOffset in="Blur" dx="3" dy="3" result="OffsetBlur" />
<feMerge>
<feMergeNode in="OffsetBlur" />
<feMergeNode in="SourceGraphic" />
</feMerge>
</filter>
</defs>
<text x="20" y="20" >
Mouse the text below and watch a drop shadow being applied to
it.
</text>
<text class="big" x="20" y="120" filter="none">
<set begin="mouseover" end="mouseout"
 attributeName="fill" from="black" to="red" />
<set begin="mouseover" end="mouseout"
 attributeName="filter" from="none" to="url(#myFilter)" />
Sams Teach Yourself XML in 10 Minutes
</text>
<text class="big" x="20" y="220" filter="none">
<set begin="mouseover" end="mouseout"
 attributeName="fill" from="black" to="red" />
<set begin="mouseover" end="mouseout"
 attributeName="filter" from="none" to="url(#myFilter)" />
Sams Teach Yourself XML in 10 Minutes
</text>
</svg>

```

Figure 15.4 is a composite image with both rolled-over and unrolled-over text.



**FIGURE 15.4** A drop shadow created using SVG filter elements in response to rolling over the text

This short chapter has been able to indicate only a few of SVG's capabilities. Appendix B, "XML Tools," gives some information about Web sites and mailing lists where you can explore SVG further.

## Summary

This lesson introduced the reasons for the development of SVG and explained some of its advantages. It also described and demonstrated a number of SVG 1.0 elements, the use of XLink in SVG and an example of using an SVG filter.

## LESSON 16

# The Document Object Model



*In this lesson, you will learn the basics of how to program the XML Document Object Model.*

## The Document Object Model

The Document Object Model (DOM) is a series of W3C specifications that provide increasing functionality to access and manipulate XML (and HTML) documents programmatically. The DOM provides a practical way to manipulate, create, and modify XML documents programmatically.

At the time of this writing, the DOM Level 2 specifications (see Appendix B, “XML Tools,” for links) are the current versions. DOM Level 3 specifications are in development.



**Note** Only the DOM Level 2 Core interfaces are considered in this chapter and in Chapter 17, “The Document Object Model—2.” The Document Object Model Level 2 also specifies more specialized interfaces to be used—for example, with HTML documents and CSS style sheets.

## Object and Interfaces

The name of the Document Object Model refers to *objects*, but the DOM is defined in terms of *interfaces*. An object packages a specified group of *properties* and *methods* in a convenient object, for want of a better term.

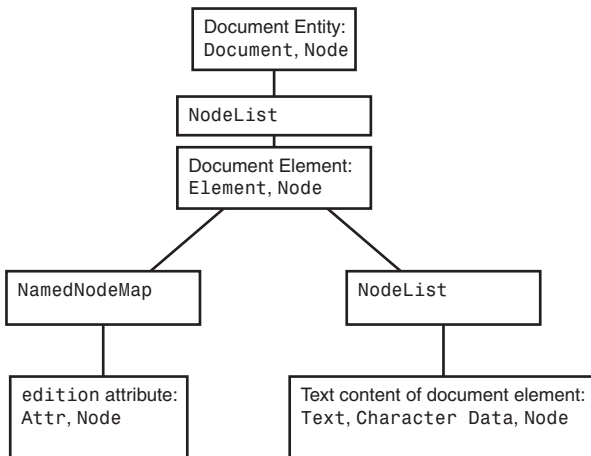
A property can be thought of as a characteristic of an object. For example, a car might have a `color` property and a `number_of_wheels` property. Each of these properties tells about some characteristic of the car. Similarly, a car object might have `go_forward()`, `go_backwards()`, and `stop()` methods. These methods would tell something about what the object can do.

An interface can be thought of as a convenient package of properties and methods. An object can implement an interface—a specified and named package of properties and methods. It can either add properties and methods specific to that object or can implement the properties and methods defined in one or more other interfaces.

You learned earlier that an XML document can be viewed as a logical hierarchy. In the DOM, you can model that hierarchy using several nodes.

Let's look at a simple XML document and consider how it is represented in the DOM. Figure 16.1 shows a hierarchy representing the interfaces and objects that make up the DOM representation of the document.

```
<book edition="1">
Sams Teach Yourself XML in 10 Minutes
</book>
```



**FIGURE 16.1** A hierarchical representation of the example document.

Let's first look at Figure 16.1 and consider each part of the hierarchy. Later you will look at individual interfaces shown in the figure in more detail.

The node at the apex of the hierarchy represents the (invisible) document entity of an XML document. That node implements the `Document` interface and the `Node` interface.

The next node in the hierarchy implements the `NodeList` interface. In this simple document there is only a single node that is the child of the `NodeList`—the node representing the document element, the book element, of the XML document. That node implements the `Node` interface and the `Element` interface.

The `Element` node has a child node that implements the `NamedNodeList` interface as its child node. That node has a single child node that represents the `edition` attribute in the XML document. The node implements the `Attr` and `Node` interfaces.

That `Element` node also has a node that implements the `NodeList` interface as a child node. In this simple document, there is only a single node that contains the text content of the `Book` element node. That node implements the `Node`, `Text`, and `CharacterData` interfaces.

## DOM Interfaces

Let's examine in more detail the interfaces in Figure 16.1.

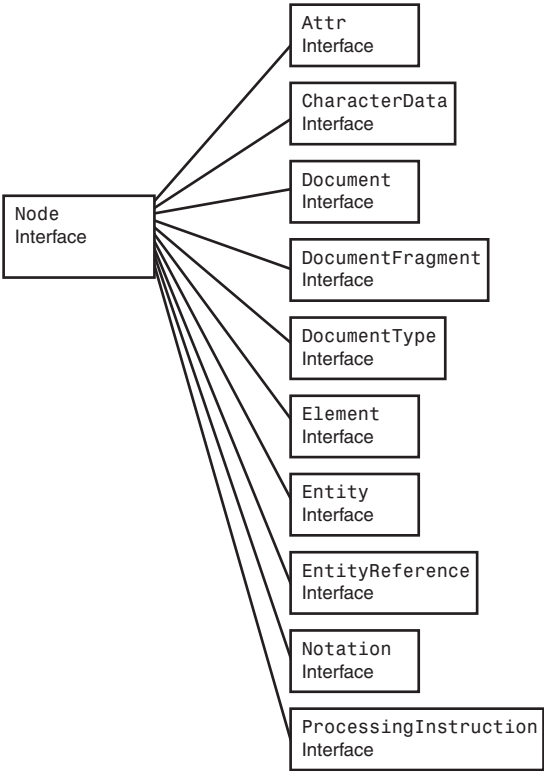
### The Node Interface

As you saw in Figure 16.1, many DOM nodes implement the `Node` interface. They might also have other properties and methods in addition to those that the `Node` interface provides.

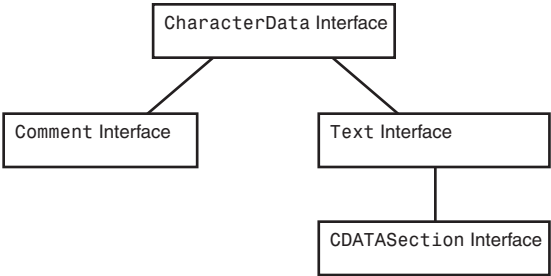
These DOM interfaces are said to *extend* the `Node` interface. Figure 16.2 shows the interfaces in DOM Level 2 Core that extend the `Node` interface.

In general, the names of the node types that inherit from the `Node` interface can be readily understood from their names. The `DocumentType` interface, for example, corresponds to the `DOCTYPE` declaration.

DOM interfaces can be extended several levels deep. Figure 16.3 shows the node types that extend the `CharacterData` interface.



**FIGURE 16.2** The DOM Level 2 interfaces that extend the Node interface.



**FIGURE 16.3** The interfaces that extend the CharacterData interface.



Both the `Comment` and `Text` interfaces extend the `CharacterData` interface. The `Text` interface itself is extended by the `CDATASection` interface. You will look at these text-oriented interfaces later, but first let's look briefly at the properties and methods of the `Node` interface.

The `Node` interface has the following properties:

- `attributes`—Read-only, of type `NamedNodeList`
- `childNodes`—Read-only, of type `NodeList`
- `firstChild`—Read-only, of type `Node`
- `lastChild`—Read-only, of type `Node`
- `localName`—Read-only, of type `String`
- `namespaceURI`—Read-only, of type `String`
- `nextSibling`—Read-only, of type `Node`
- `nodeName`—Read-only, of type `String`
- `nodeType`—Read-only, of type `Number`
- `nodeValue`—Of type `String`
- `ownerDocument`—Read-only, of type `Document`
- `parentNode`—Read-only, of type `Node`
- `prefix`—Of type `String`
- `previousSibling`—Read-only, of type `Node`

The `Node` interface has the following methods:

- `appendChild(newChild)`—Returns a `Node` object
- `cloneNode(deep)`—Returns a `Node` object
- `hasAttributes()`—Returns a Boolean value
- `hasChildNodes()`—Returns a Boolean value
- `insertBefore(newChild, refChild)`—Returns a `Node` object
- `isSupported(feature, value)`—Returns a Boolean value

- `normalize()`—Has no return value
- `replaceChild(newChild, oldChild)`—Returns a Node object
- `removeChild(oldChild)`—Returns a Node object

## The NodeList Interface

The child nodes of the Document node implement the NodeList interface.

The NodeList interface has the `length` property, which is a read-only property of type Number. The NodeList interface has the `item(index)` method, which returns a Node object.

## The NamedNodeMap Interface

You can locate the attributes of an element using the NamedNodeMap interface.

The NamedNodeMap interface has the `length` property, a read-only property of type Number.

The NamedNodeMap interface has the following methods, all of which return a Node object:

- `getNamedItem(name)`
- `getNamedItemNS(namespaceURI, localName)`
- `item(index)`
- `removeNamedItem(name)`
- `removeNamedItemNS(namespaceURI, localName)`
- `setNamedItem(arg)`
- `setNamedItemNS(namespaceURI, localName)`

Before looking in more detail at the interfaces in Figure 16.1, you will look briefly at generally relevant interfaces.

## The DOMImplementation Interface

The DOMImplementation interface has no properties, but it does provide methods that allow a programmer to determine what is supported by that implementation.

The `hasFeature(feature, version)` method returns a Boolean value enabling you to test the availability of particular features. The `createDocument(namespaceURI, qualifiedName, doctype)` method enables you to create a new XML document. The `createDocumentType(qualifiedName, publicId, systemId)` method enables you to create a new DOCTYPE declaration.

## The DOMException Interface

Several programming languages use a concept called an *exception* to handle errors that occur while a program is running. The DOM provides a DOMException interface. An exception is often said to be *thrown* and is *caught* by an *exception handler*.

The DOMException interface has a single code property, which is of type number. The value of the code property corresponds to the type of exception that has been raised. When programming, you should consider the likely types of errors that might occur and provide error-handling code that provides appropriate responses to those problems.

## DOM Interfaces Properties and Methods

Because of space constraints, this section looks at the properties and methods of selected interfaces only.

### The Document Interface

The Document interface represents an XML document. The Document interface has the following properties:

- `doctype`—Read-only, of type `DocumentType`
- `implementation`—Read-only, of type `DOMImplementation`
- `documentElement`—Of type `Element`

The methods of the `Document` interface can be used to create new parts of an XML document or to retrieve information about the document. The `Document` interface has the following methods:

- `createAttribute(name)`—Returns an `Attr` object
- `createAttributeNS(namespaceURI, qualifiedName)`—Returns an `Attr` object
- `createCDATASection(data)`—Returns a `CDATASection` object
- `createComment(data)`—Returns a `Comment` object
- `createDocumentFragment()`—Returns a `DocumentFragment` object
- `createElement(tagName)`—Returns an `Element` object
- `createElementNS(namespaceURI, qualifiedName)`—Returns an `Element` object
- `createEntityReference(name)`—Returns an `EntityReference` object
- `createProcessingInstruction(target, data)`—Returns a `ProcessingInstruction` object
- `createTextNode(data)`—Returns a `Text` object
- `getElementsByTagName(tagname)`—Returns a `NodeList` object
- `getElementsByTagNameNS(namespaceURI, localName)`—Returns a `NodeList` object
- `importNode(importedNode, deep)`—Returns a `Node` object

## The DocumentType Interface

The DocumentType interface is the DOM representation of the DOCTYPE declaration.

The DocumentType interface has the following properties:

- `entities`—Read-only, of type `NamedNodeMap`
- `internalSubset`—Read-only, of type `String`
- `name`—Read-only, of type `String`
- `notations`—Read-only, of type `NamedNodeMap`
- `publicId`—Read-only, of type `String`
- `systemId`—Read-only, of type `String`

## The Element Interface

The Element interface represents an element in an XML document.

The Element interface has the `tagName` property, which is a read-only property of type `String`.

The Element interface has the following methods:

- `getAttribute(name)`—Returns a value of type `String`
- `getAttributeNS(namespaceURI, localName)`—Returns a value of type `String`
- `getAttributeNode(name)`—Returns a value of type `Attr`
- `getAttributeNodeNS(namespaceURI, localName)`—Returns a value of type `Attr`
- `getElementsByTagName(tagname)`—Returns a `NodeList` object
- `getElementsByTagNameNS(namespaceURI, localName)`—Returns a `NodeList` object
- `hasAttribute(name)`—Returns a value of type `Boolean`

- `hasAttributeNS(namespaceURI, localName)`—Returns a value of type `Boolean`
- `removeAttribute(name)`—Returns no value
- `removeAttributeNode(oldAttr)`—Returns a value of type `Attr`
- `removeAttributeNS(namespaceURI, localName)`—Returns no value
- `setAttribute(name, value)`—Returns no value
- `setAttributeNode(newAttr)`—Returns a value of type `Attr`
- `setAttributeNodeNS(newAttr)`—Returns a value of type `Attr`
- `setAttributeNS(namespaceURI, localName)`—Returns no value

## The Attr Interface

The `Attr` interface is the DOM representation of an attribute in an XML document.

The `Attr` interface has the following properties:

- `name`—Read-only, of type `String`
- `ownerElement`—Read-only, of type `Element`
- `specified`—Read-only, of type `Boolean`
- `value`—Of type `String`

The `Attr` interface has no methods specific to it.

## The CharacterData Interface

The `CharacterData` interface is intended to contain character data. The `Comment` and `Text` interfaces extend the `CharacterData` interface.

The `CharacterData` interface has the following properties:

- `data`—Of type `String`
- `length`—Read-only, of type `Number`

The `CharacterData` interface has the following methods:

- `appendData(arg)`—Has no return value
- `deleteData(offset, count)`—Has no return value
- `insertData(offset, arg)`—Has no return value
- `replaceData(offset, count, arg)`—Has no return value
- `substringData(offset, count)`—Returns a value of type `String`

## The Text Interface

The `Text` interface has no properties specific to it.

The `Text` interface has the `splitText(offset)` method, which returns a `Text` object.

In Chapter 17 you will create some examples using DOM properties and methods.

## Summary

In this lesson, you were introduced to the DOM Level 2. The chapter also discussed the concept of an interface and how an interface can be *extended*. In addition, you learned about the DOM `Node` interface and DOM interfaces, many of which extend the `Node` interface.



## LESSON 17

# The Document Object Model—2

*In this lesson, you will see examples of using the Document Object Model to create, retrieve, and manipulate parts of XML documents.*

In the examples in this chapter, you will use SVG, which was introduced in Chapter 15, “Presenting XML Graphically—SVG,” as an example of an XML application language. SVG has a number of extensions to the core XML DOM, but you won’t use any of those to manipulate the DOM. Our purpose is to demonstrate the properties and methods of the XML DOM Level 2 Core, all of which are available in SVG 1.0.

In each of the following examples, the programming language used is JavaScript because many Web developers already have some experience using it. Similar techniques can be applied using Java or other programming languages.

## Creating a New Element

The DOM provides more than one way to create an element. You will examine some options in the examples that follow.

### Using the `createElement()` Method

In this example, you will create a new SVG element using the `createElement()` method of the Document interface. You also will set the values of the attributes of the newly created element.

Listing 17.1 shows the code. Apart from the `circle` element that you will create using the DOM, the SVG document would be blank.



## LISTING 17.1 CreateCircle.svg: Creating a circle Element Using the DOM

---

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg onload="Initialize(evt)">
<script type="text/javascript" >
<![CDATA[
var SVGDoc;
var SVGRoot;
var myCircle;
function Initialize(){
SVGDoc = evt.getTarget().getOwnerDocument();
SVGRoot = SVGDoc.getDocumentElement();
createCircle(evt);
}

function createCircle(){
myCircle = SVGDoc.createElement("circle");
myCircle.setAttribute("cx", "50px");
myCircle.setAttribute("cy", "50px");
myCircle.setAttribute("r", "30px");
myCircle.setAttribute("style", "fill:none;
stroke:red; stroke-width:3");
SVGRoot.appendChild(myCircle);
}
]]>
</script>

</svg>
```

The onload attribute of the svg element calls the Initialize() function when the document loads.

Notice that the content of the script element is enclosed in a CDATA section to inform the parser that the content should not be treated as XML suitable for parsing.

The new circle element is created using the createElement() method of the Document node in the following line of code:

```
myCircle = SVGDoc.createElement("circle");
```

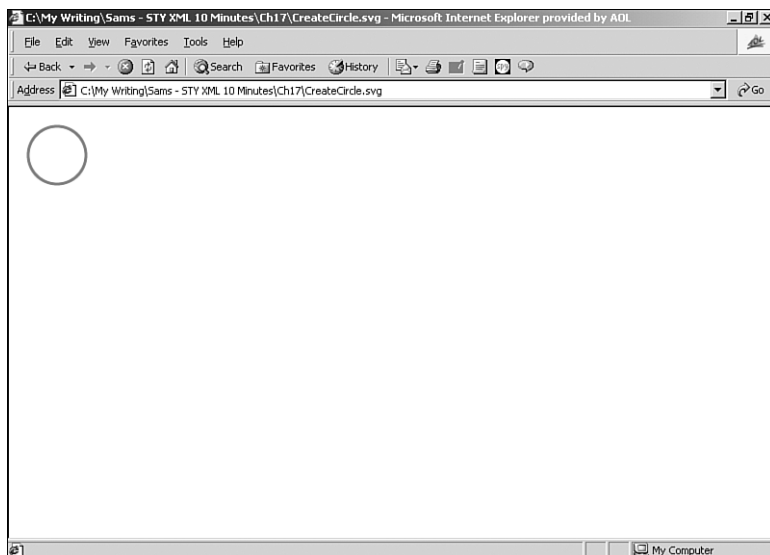
The `circle` element node exists, but it does not yet have any values for its attributes. The following lines of code use the `setAttribute()` method of the `Element` node to assign values to the attributes of the `circle` element that define the position of its center (`cx` and `cy` attributes), its radius (`r` attribute), and its style (`style` attribute):

```
myCircle.setAttribute("cx", "50px");
myCircle.setAttribute("cy", "50px");
myCircle.setAttribute("r", "30px");
myCircle.setAttribute("style", "fill:none; stroke:red;
 stroke-width:3");
```

You have successfully created the `circle` element and set its attribute values. Finally, you need to append the newly created element as a child of the `Element` node that represents the `svg` document element. This is done using the `appendChild()` method of the `Node` interface:

```
SVGRoot.appendChild(myCircle);
```

Figure 17.1 shows the onscreen appearance when Listing 17.1 is run.



**FIGURE 17.1** A circle element created using the `createElement()` method.

## Using the `createElementNS()` Method

When you have a document in which XML namespaces are not being used or in which a single namespace is using a default namespace declaration, you can use the `createElement()` method to create new elements. If you have documents in which namespace prefixes are used or in which multiple namespaces are (or might be) in use, it makes sense to remove any doubt and use the `createElementNS()` method. This makes it absolutely clear what namespace the newly created element belongs to.

Listing 17.2 shows an example that creates an SVG `rect` element. This creates a rectangle onscreen. However, because all SVG elements in this example use the namespace prefix `svg`, you have to use the `createElementNS()` method to produce the rectangle shape.

### LISTING 17.2 CreateRectNS.svg: Using the `createElementNS()` Method to Create a Rectangle

---

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg:svg
 xmlns:svg="http://www.w3.org/2000/svg"
 onload="Initialize(evt)">
<svg:script type="text/javascript" >
<![CDATA[
var SVGDoc;
var SVGRoot;
var myRectangle;
function Initialize(){
SVGDoc = evt.getTarget().getOwnerDocument();
SVGRoot = SVGDoc.getDocumentElement();
createRectangle(evt);
}

function createRectangle(){
myRectangle = SVGDoc.createElementNS
 ("http://www.w3.org/2000/svg", "rect");
myRectangle.setAttribute("x", "50px");
myRectangle.setAttribute("y", "50px");
myRectangle.setAttribute("width", "300px");
myRectangle.setAttribute("height", "100px");
myRectangle.setAttribute("style", "fill:#CCCCC;
 stroke:green; stroke-width:4");
```

**LISTING 17.2** Continued

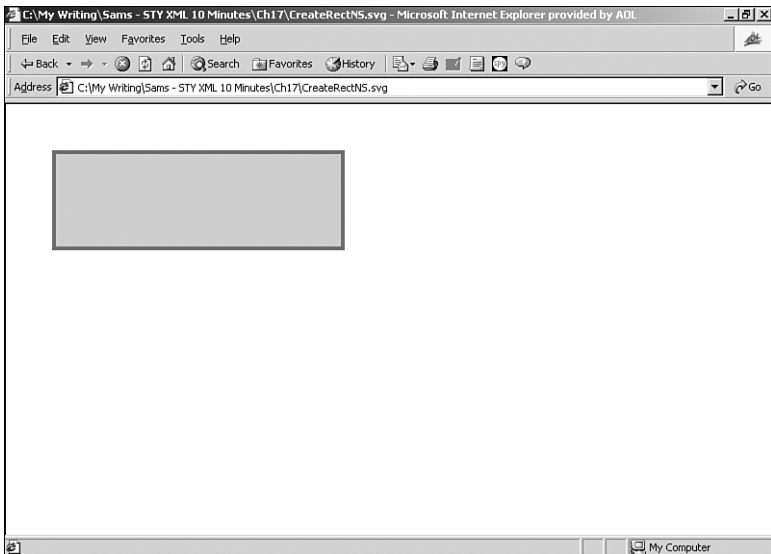
```
SVGRoot.appendChild(myRectangle);
}
]]>
</svg:script>

</svg>
```

The `createElementNS()` method has two arguments. The first is the namespace URI for the element that is to be created. The second argument is the local part of the QName for the element. The namespace URI for SVG 1.0 is `www.w3.org/2000/svg`.

Notice that it isn't necessary to specify the namespace prefix in the call to the `createElementNS()` method. The namespace URI is already associated with a namespace prefix by means of the namespace declaration `xmlns:svg="http://www.w3.org/2000/svg"` contained in the start tag of the `svg:svg` document element.

Figure 17.2 shows the onscreen appearance when Listing 17.2 is run.



**FIGURE 17.2** A `rect` element created using the `createElementNS()` method.

# Retrieving Information from the DOM

In earlier examples in this chapter, you created elements and set values for attributes belonging to newly created elements. You can also access information already contained in the DOM and use it for additional purposes.

## The DocumentType Interface's Properties

If you want to retrieve the information contained in the DOCTYPE declaration, you can access and process the properties of the DocumentType object. Listing 17.3 shows an example.

### LISTING 17.3 DoctypeProps.svg: Retrieving and Displaying the DocumentType Object's Properties

---

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg:svg
 xmlns:svg="http://www.w3.org/2000/svg"
 onload="Initialize(evt)">
<svg:script type="text/javascript" >
<![CDATA[
var SVGDoc;
var SVGRoot;
var SVGDoctype;
function Initialize(){
SVGDoc = evt.getTarget().getOwnerDocument();
SVGRoot = SVGDoc.getDocumentElement();
SVGDoctype = SVGDoc.doctype;
getDoctype(evt);
}

function getDoctype(){
var docElem = SVGDoctype.name;
var firstString = "The document element is: " + docElem;
var firstText = SVGDoc.getElementById("docelem");
var stars = firstText.firstChild;
stars.replaceData(0,5, firstString);

var docPubID = SVGDoctype.publicId;
var secondString = "The public identifier is: " + docPubID;
```

**LISTING 17.3** Continued

---

```
var secondText = SVGDoc.getElementById("pub");
var stars2 = secondText.firstChild;
stars2.replaceData(0,5, secondString);

var docSystID = SVGDoctype.systemId;
var thirdString = "The system identifier is: " + docSystID;
var thirdText = SVGDoc.getElementById("syst");
var stars3 = thirdText.firstChild;
stars3.replaceData(0,5, thirdString);
}
]]>
</svg:script>
<text id="docelem" x="20" y="40">

</text>
<text id="pub" x="20" y="100">

</text>
<text id="syst" x="20" y="160">

</text>
</svg:svg>
```

In the `Initialize()` function, the `SVGDoctype` variable is assigned the value of the `doctype` property of the `SVGDoc` variable. So, the `SVGDoctype` variable is a `DocumentType` object. Therefore, you can retrieve the values of various properties of the `DocumentType` object.

When you call the `getDoctype()` function, you retrieve each of three properties of the `DocumentType` object. You use the value that they contain to replace the data in a `Text` object (which extends a `CharacterData` object) that is a child node of each of three SVG text elements. Let's look in detail at the first of these.

This code declares a variable named `docElem` and assigns to it the value of the `name` property of the `SVGDoctype` variable (which itself contains the value of the `doctype` property of the `Document` object):

```
var docElem = SVGDoctype.name;
```

This assigns the element type name of the document element to the `docElem` variable.

Then you declare a variable `firstString` and create a message for display that incorporates the value of the `docElem` variable:

```
var firstString = "The document element is: " + docElem;
```

Then you use the `getElementById()` method of the `Document` object to uniquely retrieve the `Element` node corresponding to the SVG text element with an `id` attribute of the value `docelem`.

```
var firstText = SVGDoc.getElementById("docelem");
```

Next declare a `stars` variable and assign it the value of the `firstChild` property (a property of the `Element` interface) of the `firstText` variable (the first SVG text element, identified by its `id` attribute):

```
var stars = firstText.firstChild;
```

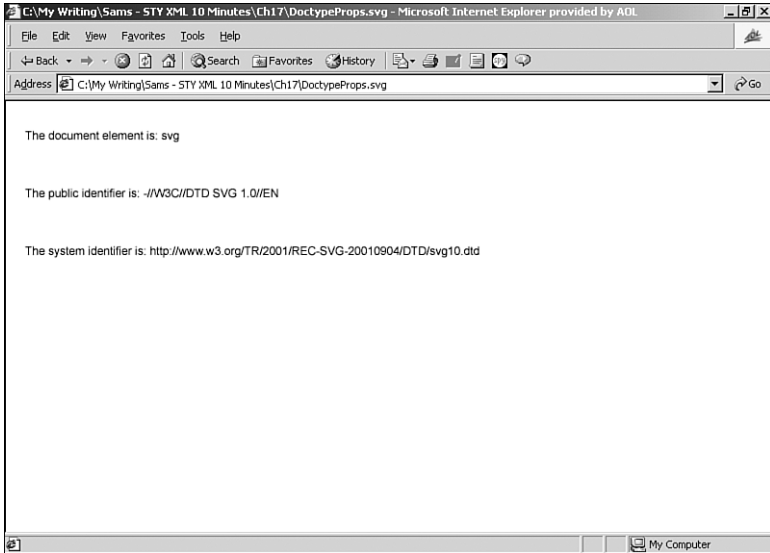
Finally, you use the `replaceData()` method of the `CharacterData` interface to replace the three stars, which is the original content of the text element.

```
stars.replaceData(0,5, firstString);
```

A similar process is carried out for each of the other two text elements. When the document loads, the script instantly replaces the asterisks with three messages that display the values of the `name`, `publicId`, and `systemId` properties of the `DocumentType` interface. Figure 17.3 shows the onscreen appearance.

## Displaying a List of Child Nodes

In this example, you will retrieve information about the `NodeList` object. This object contains information about the child nodes of the node that represents the `svg` element in Listing 17.4.



**FIGURE 17.3** Displaying properties of the DocumentType interface.

### LISTING 17.4 ChildNodes.svg: Retrieving Information About Child Nodes

```
<?xml version='1.0'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg
 xmlns:svg="http://www.w3.org/2000/svg"
 onload="Initialize(evt)">
<script type="text/javascript" >
<![CDATA[
var SVGDoc;
var SVGRoot;
var SVGDoctype;
function Initialize(){
SVGDoc = evt.getTarget().getOwnerDocument();
SVGRoot = SVGDoc.getDocumentElement();
SVGDoctype = SVGDoc.doctype;
getChildNodes(evt);
}
```



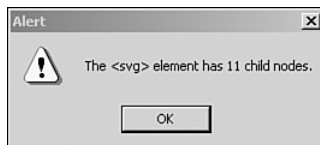
**LISTING 17.4** Continued

```
function getChildNodes(){
var Length = SVGRoot.childNodes.length;
alert("The <svg> element has " + Length + " child nodes.");

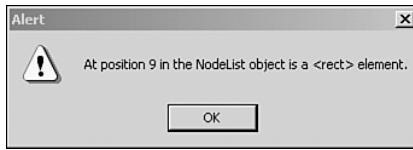
for (i=0; i<Length; i++){
if (SVGRoot.childNodes.item(i).nodeType==1){
alert("At position " + i + " in the NodeList object
 is a <" + SVGRoot.childNodes.item(i).tagName + ">
 element.");
} // end if

} // end for loop
} // end getChildNodes() function
]]>
</script>
<text id="first" x="20" y="40">
This is the first <text> element.
</text>
<text id="second" x="20" y="100">
This is the second <text> element.
</text>
<text id="third" x="20" y="160">
This is the third <text> element.
</text>
<rect x="20" y="200" width="200" height="50"
 style="stroke:red; stroke-width:4; fill:none;" />
</svg>
```

Before considering the explanation of how the code works, first look at the output of the code onscreen. Figures 17.4 and 17.5 show two of the alert boxes produced by the code. First, you might be surprised by the first alert box shown in Figure 17.4. Why does it say that there are 11 child nodes? If you look carefully at the code, you will see that each element starts on a new line. So, there is a Text node consisting only of whitespace separating each element.



**FIGURE 17.4** An alert box showing the number of child nodes for the `<svg>` element.



**FIGURE 17.5** An alert box for the `<rect>` element.

Now look at how the code works. Inside the `getChildNodes()` function, you declare the `Length` variable. To the `Length` variable you assign the `length` property of the `NodeList` object that contains information about the child nodes of the document element.

```
var Length = SVGRoot.childNodes.length;
```

Next, you use the JavaScript `alert()` function to output the value of the `Length` variable:

```
alert("The <svg> element has " + Length + " child nodes.");
```

Then you use a `for` loop to iterate through the child nodes. The `if` statement tests whether the value of the `nodeType` property retrieved by the `item()` method of the `NodeList` object equals 1. When the value of the `nodeType` property is 1, you know that it is an `Element` node. If the node isn't an `Element` node (because it is whitespace), you do nothing. But when the value of the `nodeType` property indicates that the node is an `Element` node, you output the position of the node and its element type name using the `tagName` property of the `Element` interface.

```
for (i=0; i<Length; i++){
 if (SVGRoot.childNodes.item(i).nodeType==1){
 alert("At position " + i + " in the NodeList object is a <" +
 SVGRoot.childNodes.item(i).tagName + "> element.");
 } // end if

} // end for loop
} // end getChildNodes() function
```

## Summary

In this lesson, you examined examples of using several DOM properties and methods. You learned how to use some DOM methods to create new elements and how to retrieve the values of various DOM properties.

## LESSON 18

# SAX—The Simple API for XML



*In this lesson you will learn about the Simple API for XML, SAX, and the basics of how SAX programming is done.*

## What SAX Is and How It Differs from DOM

As you learned in Chapter 16, “The Document Object Model,” and Chapter 17, “The Document Object Model—2,” DOM programming depends on a tree-like hierarchy of nodes that implement a specified number of interfaces. SAX takes a very different approach. It uses *events* that occur during parsing of an XML document, and it doesn’t build a tree hierarchy in memory.

SAX programming is often done using either Java or Visual Basic. In this chapter, you will use Java to illustrate how SAX can be coded.

## Brief History of SAX

Unlike most of the XML-related topics covered in this book, SAX is not a product of the W3C. It was created by members of the XML-Dev mailing list to fill a perceived gap in available tools in the early days around the time XML 1.0 was finalized. SAX version 1 was completed in May 1998. SAX version 2 was completed in May 2000.

## Pro and Cons of SAX

This section discusses a number of issues relating to SAX and its suitability, compared to DOM programming.

## Large Documents

To manipulate a document using DOM programming requires the complete in-memory hierarchy of nodes to be built before manipulation using DOM can begin. As document size increases, the time needed to build the in-memory tree increases.

Also, as XML document size increases, the amount of RAM needed to contain the in-memory hierarchy of nodes increases as well. Beyond a certain document size, which varies according to installed RAM and other factors, the amount of memory available will be inadequate and swapping to disk will be needed. As expected, this will cause deterioration in performance.

In principle, SAX is free from this type of memory limitation because events occur during parsing of an XML document and because the appropriate processing in response to those events takes place without the need to create a potentially large in-memory hierarchy.

## Programmer Mindset

It is widely accepted that many XML programmers find the concepts of programming using SAX much less natural than using DOM programming. Perhaps that preference is partly because DOM programming is familiar from scripting HTML Web pages. Whatever the cause, many programmers aren't too comfortable using SAX.

Writing code to handle a cascade of events is certainly different from writing typical JavaScript or Java procedural code.

# Basics of SAX Programming

SAX programming depends on recognizing events that occur during the process of parsing an XML document.

## Parsing Events

In this section, you will use pseudocode to see what happens as an XML document is parsed using a SAX parser.

Listing 18.1 shows a short XML document that you will use to illustrate the SAX approach.

---

**LISTING 18.1** SAXSource.xml: A Short XML Document

---

```
<?xml version='1.0'?>
<?xml-stylesheet href="myCSS.css" type="text/css" ?>
<!-- This is an XML comment. -->
<myDocument>
Some text content.
</myDocument>
```

A SAX parser would respond to parsing an XML document like this by signaling events, similar to the following:

```
start_document;
processing_instruction;
start_element (<myDocument>);
characters;
end_element (</myDocument>);
end_document;
```

The existence of the XML declaration and the comment are ignored.

Clearly, in anything but a very short document, a very large number of events will be signaled. It is up to the programmer to write code to define what to do in response to all such events.

## SAX 2 Interfaces

When discussing the Document Object Model, we defined an interface as a specified grouping of properties and methods. SAX uses the following important interfaces:

- `ContentHandler`—Defines methods that process XML document content
- `DTDHandler`—Defines methods that process DTDs
- `ErrorHandler`—Defines methods that process errors

These interfaces are implemented by classes in SAX parsers.

## Installing a SAX Parser

To run a Java SAX-capable parser, you need the following installed on your computer:

- A Java Software Development Kit, version 1.1 or higher.
- A SAX2-compatible XML parser installed on your Java CLASSPATH.
- A SAX2 distribution on your Java CLASSPATH. This would likely be included with the SAX2-compatible XML parser.

## Installing the JSDK

You might already have a JSDK installed. If not, you can download a JSDK from <http://java.sun.com/j2se/>. The URL gives you access to information about the Standard Edition of Java—j2se.

You can check if you have a JSDK installed by searching for a file named `javac.exe`. If `javac.exe` is present, you have a JSDK (formerly called a JDK) installed. Up to Java version 1.3, you might find `javac.exe` in a directory named something like `c:\jdk1.3.1\bin`. In Java 2 version 1.4, you will find it in a directory named something like `c:\j2sdk1.4\bin`.

Download the JSDK appropriate to your operating system and install it according to the instructions supplied by Sun.

Take note of the exact name of the directory that you install the JSDK into. You will add that to your computer's path environment variable in a moment.

## Installing the Xerces Parser

Information about the Java version of the Xerces 2 XML parser is located at <http://xml.apache.org/xerces2-j/index.html>. From the Downloads link, select the latest stable version of the Xerces-J parser appropriate to your operating system.

When the download has completed, install the Xerces-J parser to a directory. We installed Xerces-J in `c:\Xerces-J2.0.2`.

## Setting path and CLASSPATH Environment Variables

Your computer needs to be capable of locating the Java programs and the Xerces parser.

The directory into which you installed the JSDK must be added to the path environment variable. In Windows 2000, go to the Control Panel and select the System option. The System Properties window should open. Select the Advanced tab. Click the Environment Variables button halfway down the page. Scroll down the list of System Variables until the path variable is highlighted. Click the Edit button. A window will open with the current value of the path variable.

Your current path might be something like this:

```
c:\WINNT\system32;c:\WINNT
```

You need to add the directory where the JSDK is installed. In my case, the JSDK is in `c:\j2sdk1.4.0\bin`, so that is added to the existing value of the path variable, separated by a semicolon from the existing value.

```
c:\WINNT\system32;c:\WINNT;c:\j2sdk1.4.0\bin
```



**Caution** Check if another Java installation already exists in the path variable. If so, you might want to simply use that. Having more than one Java installation specified in the path environment variable is a recipe for problems.

If you have done no Java programming on your computer, you will likely have to create a new CLASSPATH environment variable. Otherwise, edit the existing CLASSPATH environment variable to add `c:\Xerces-J2.0.2` (or the directory you installed Xerces in) to it, separated by a semicolon from any existing paths.



**Tip** If you don't want to permanently change environment variables, create a short batch file that will set the path and CLASSPATH variables from the command line.

You also need to add the Xerces-J version 2 jar files to the CLASSPATH—`xercesImpl.jar` and `xmlParserAPIs.jar`.

Now that we've discussed a number of issues about how to use SAX, let's move on and use a Java example to illustrate the basics of how SAX can be used.

## Simple SAX Example

This example creates a Java program that you can run from the command line. You will be able to specify an XML document to be parsed, and messages will be output onscreen in response to events generated by the SAX parser.



**Note** Java, like XML, is case sensitive. All names of interfaces, classes, and so on in the following code must use the correct case if your application is to run correctly.

Listing 18.2 shows a simple SAX example.

### **LISTING 18.2** `myHandler.java`: A Java Program That Provides Screen Output in Response to SAX Events

```
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

import org.apache.xerces.parsers.SAXParser;
```



**LISTING 18.2** Continued

---

```
public class myHandler extends DefaultHandler
{
 public static void main(String[] argv) throws Exception {
 if (argv.length == 0) {
 System.out.println("You need to specify a file name");
 System.exit(0);
 }
 System.out.println("The program myHandler has started ...");
 myHandler reader = new myHandler();
 reader.read(argv[0]);
 }

 public void read(String fileName) throws Exception{
 System.out.println("read() method entered ...");
 XMLReader parser = XMLReaderFactory.createXMLReader();

 parser.setContentHandler(this);
 parser.parse(fileName);
 }

 public void startDocument() throws SAXException {
 System.out.println("The document has been opened.");
 }

 public void processingInstruction(String target,
 String data) throws SAXException {
 System.out.println("A processing instruction with target,
 " +target+ " and data " +data+ ".");
 }

 public void startElement(String uri, String localName,
 String QName, Attributes attributes) throws SAXException{
 System.out.println("Start tag of element "
 + localName + " was found.");
 }

 public void endElement(String uri, String localName,
 String QName) throws SAXException{
 System.out.println("End tag of element "
 + localName + " was found.");
 }

 public void characters(char[] characters, int start,
 int length) throws SAXException {
```

**LISTING 18.2** Continued

---

```
System.out.println("Character content encountered.");
}

public void endDocument() throws SAXException {
 System.out.println("The document has been completed.");
}

}
```

The Java file must be compiled. Using the javac compiler, you can issue this command to create a class file:

```
javac myHandler.java
```

To run the class file, issue this command:

```
java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
 myHandler SAXSource.xml
```

That command assigns the class `org.apache.xerces.parsers.SAXParser` to the environment variable `org.xml.sax.driver`.

When the code is run, you will see an onscreen appearance like that shown in Figure 18.1.

Let's look briefly at what the code does.

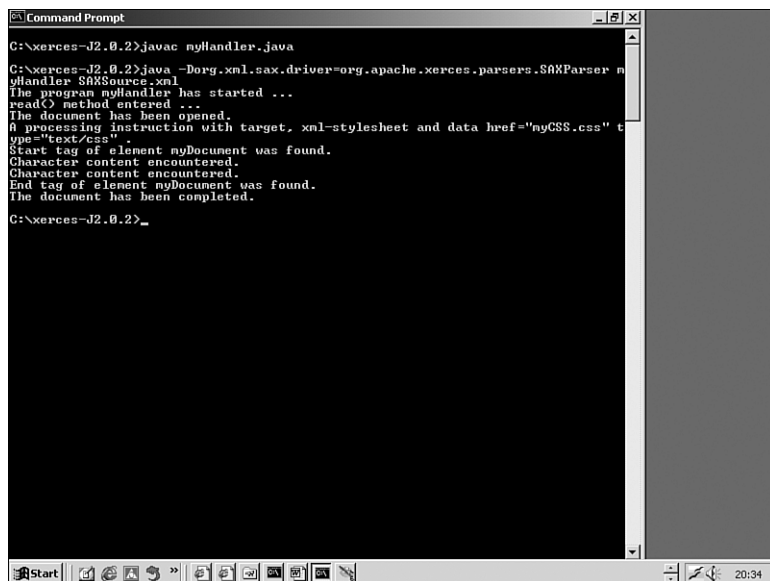
You first create a class called `myHandler`. The `main()` method accepts string arguments. So, when you enter the following command, the `SAXSource.xml` is the sole string argument.

```
java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
 myHandler SAXSource.xml
```

This next `if` statement checks to see if a filename has been supplied as an argument:

```
if (argv.length == 0) {
 System.out.println("You need to specify a file name");
 System.exit(0);
}
```

If no filename is supplied, an error message is output and the program exits.



```
Command Prompt
C:\xerces-j2.0.2>javac myHandler.java
C:\xerces-j2.0.2>java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser myHandler SAXSource.xml
The program myHandler has started ...
read() method entered ...
The document has been opened.
A processing instruction with target, xmlstylesheet and data href="myCSS.css" type="text/css" .
Start tag of element myDocument was found.
Character content encountered.
End tag of element myDocument was found.
The document has been completed.
C:\xerces-j2.0.2>
```

**FIGURE 18.1** The output when the myHandler class is run.

This code indicates that the program has started successfully:

```
System.out.println("The program myHandler has started ...");
```

The read() method creates an XMLReader that parses the file supplied in the argument at the command line.

The parser raises events at many points as it parses the document. The startDocument() method outputs a message when the start of the document has been encountered. Then when the `<?xml-stylesheet ?>` processing instruction is encountered, the processingInstruction() method outputs a message that tells what the target and data of the processing instruction are.

When the start of the myDocument element is found, the startElement() method is called and a message is output. When the character content of the myDocument element is encountered, the characters() method is called and outputs a message.

Finally, the `endElement()` and `endDocument()` methods are called and appropriate messages are output to the screen.

In this short example, when each of the methods of the `org.xml.sax.helpers.DefaultHandler` interface is called as a result of the appropriate event happening during parsing, we have simply output a message to the screen that says what event(s) has been encountered.

Of course, in more serious use of SAX, the way in which you implement an interface is very flexible, and you can write Java code to do whatever is appropriate in response to particular events encountered by the SAX parser.

## Summary

This lesson introduced you to the Simple API for XML, SAX. It described the event-based approach of SAX and gave an example to demonstrate simple usage of SAX.

## LESSON 19

# Beyond DTDs—W3C XML Schema



*In this chapter, you will learn about W3C XML Schema, an alternative technology to Document Type Definitions (DTDs) described in Chapter 4, “Valid XML—Document Type Definitions.”*

## W3C XML Schema Basics

W3C XML Schema is a schema-definition language expressed in XML syntax. To avoid ambiguity, W3C XML Schema is often referred to as *XSD Schema* because in an earlier version it was called XML Schema Definition Language. Recently, the abbreviation WXS has also come into use to refer to the W3C XML Schema language.



**Note** Other schema languages are expressed in XML syntax, such as RELAX NG (a combination of TREX and RELAX) and XDR (XML Data Reduced, from Microsoft).

In this section, you are introduced to some of the reasons why W3C XML Schema was developed as an alternative schema mechanism to the Document Type Definition (DTD).



**Note** The W3C XML Schema specification is lengthy and very complex. This chapter can give you only an indication of some straightforward W3C XML Schema structures.

## Limitations of DTDs

DTDs were inherited by XML from the Standard Generalized Markup Language (SGML). SGML was (and is) commonly used for document-centric data storage such as very large documents, including technical manuals. A DTD that describes most data as #PCDATA is adequate for many document-centric purposes because one piece of text is pretty much like another—simply a sequence of characters.

However, for many uses of XML to store data that might otherwise be stored in a relational or other type of database-management system, you will likely want to say more about the *type* of pieces of data that an element can contain.



A piece of data conforms to a *type* if the characters it contains express a defined idea. For example, you might have a date type as the allowed content of an element. If the characters contained were 2002/12/25, using an internationally recognized date format convention, you can interpret that as a date. If the element contained the characters \$100.50, you would conclude that the type of the data contained in the element didn't conform to a date type.

In a DTD, when mixed content was allowed, very few constraints could be imposed on the allowed content. For example, using a DTD, with mixed content it isn't possible to impose a defined order on elements. W3C XML Schema provides greater control in this situation.

W3C XML Schema also gives greater control over how many occurrences of an element are allowed. For example, it allows you to define that an element occurs at least twice and at most five times:

```
<xsd:element name="someName" minOccurs="2" maxOccurs="5" />
```

You can't do that in a DTD.

W3C XML Schema also specifies many additional datatypes for element content, and so on. W3C XML Schema has many built-in datatypes and also allows you to create your own, for example, by restricting allowed content to enumerated values or values defined by a regular expression.

## W3C XML Schema Jargon

Let's look briefly at some terminology. A W3C XML Schema document defines the allowed content for a *class* of XML documents. A single document of that class is called an *instance document*.

Elements and attributes are said to be *declared* in a W3C XML Schema document. The content of elements and attributes has a *type*, which can be either of *simple type* or *complex type*. Types can be built-in (that is, they are defined in the W3C XML Schema specification itself) or can be *defined* by a schema developer. Elements and attributes have *declarations*. Simple types and complex types have *definitions*.



**Note** Typically, anything but very simple schemas are created semi-automatically by programs such as XML Spy. The examples shown in this chapter are intended to show you basic structures within a W3C XML Schema.

## Declaring Elements

In W3C XML Schema terminology, elements and attributes are *declared*. Both elements and attributes can occur in an instance document.

Listing 19.1 shows a simple example of an XML instance document.

Listing 19.2 shows a W3C XML Schema document against which Listing 19.1 can be validated.

**LISTING 19.1** BasicDocument.xml: A Short XML Instance Document

---

```
<?xml version="1.0"?>
<basicDocument>
Some text content.
</basicDocument>
```

**LISTING 19.2** BasicDocument.xsd: A Schema for Listing 19.1

---

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="basicDocument" type="xsd:string" />
</xsd:schema>
```

The indicative namespace prefix in the W3C XML Schema document is `xsd`. The namespace declaration in the start tag of the `xsd:schema` element associates the `xsd` namespace prefix with the namespace URI `www.w3.org/2001/XMLSchema`.

An `xsd:element` element is used to declare the `basicDocument` element that is found in the instance document. The content of the `basicDocument` element is text content only and is declared, using the `type` attribute of the `xsd:element` element, to be of type `xsd:string`. The `xsd:string` type is one of many built-in types specified in the W3C XML Schema Recommendations.

If you have a slightly more complex instance document, such as the one in Listing 19.3, you must make use of a complex type definition as well as an element declaration.

**LISTING 19.3** LessBasicDocument.xml: An XML Document with Attributes and Nested Elements

---

```
<?xml version='1.0'?>
<lessBasicDocument>
 <Person category="celebrity" status="alive">
 <Name>Tiger Woods</Name>
 <Citizenship>United States</Citizenship>
 <Occupation>Professional Golfer</Occupation>
 </Person>
</lessBasicDocument>
```



As you can see, the content of the `lessBasicDocument` element is a hierarchy of nested elements. In addition, the `Person` element has two attributes, `category` and `status`.

Listing 19.4 shows one approach to a schema for Listing 19.3. W3C XML Schema provides flexible tools that allow several approaches to how the structure of an instance document is represented.

### LISTING 19.4 `LessBasicDocument.xsd`: A Schema for Listing 19.3

---

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="lessBasicDocument" type="myPersonType"/>

 <xsd:complexType name="myPersonType">
 <xsd:element name="Person" type="PersonType"/>
 </xsd:complexType>

 <xsd:complexType name="PersonType">
 <xsd:sequence>
 <xsd:element name="Name" type="xsd:string" />
 <xsd:element name="Citizenship" type="xsd:string" />
 <xsd:element name="Occupation" type="OccupationType" />
 </xsd:sequence>
 <xsd:attribute name="category" type="xsd:string" />
 <xsd:attribute name="status" type="StatusType" />
 </xsd:complexType>

 <xsd:simpleType name="OccupationType">
 <xsd:restriction base="xsd:string">
 <xsd:enumeration value="Professional Golfer" />
 <xsd:enumeration value="Actor" />
 <xsd:enumeration value="Professional Footballer" />
 </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="StatusType">
 <xsd:restriction base="xsd:string">
 <xsd:pattern value="alive|dead" />
 </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>
```

The document element is declared using this:

```
<xsd:element name="lessBasicDocument"
type="myPersonType" />
```

For the allowed content of the `lessBasicDocument` element, you need to find the definition for the `myPersonType` complex type:

```
<xsd:complexType name="myPersonType">
 <xsd:element name="Person" type="PersonType" />
</xsd:complexType>
```

To find the allowed content of the `Person` element, you need to find the definition of the `PersonType` complex type:

```
<xsd:complexType name="PersonType">
 <xsd:sequence>
 <xsd:element name="Name" type="xsd:string" />
 <xsd:element name="Citizenship" type="xsd:string" />
 <xsd:element name="Occupation" type="OccupationType" />
 </xsd:sequence>
 <xsd:attribute name="category" type="xsd:string" />
 <xsd:attribute name="status" type="StatusType" />
</xsd:complexType>
```

This definition specifies that a `Person` element is allowed to have a sequence of child elements, as defined by the content of the `xsd:sequence` element. The `Person` element may also have two attributes named `category` and `status`.

The values of some elements and attributes are simply strings, as indicated by the `xsd:string` value for the type attribute. The `xsd:string` type is one of many built-in datatypes in W3C XML Schema.

The allowed content of the `Occupation` element is defined in the definition for the `OccupationType` type:

```
<xsd:simpleType name="OccupationType">
 <xsd:restriction base="xsd:string">
 <xsd:enumeration value="Professional Golfer" />
 <xsd:enumeration value="Actor" />
 <xsd:enumeration value="Professional Footballer" />
 </xsd:restriction>
</xsd:simpleType>
```

The `xsd:restriction` element is used to restrict (or constrain) allowed values. The `base` attribute of `xsd:restriction` indicates the base type that is being restricted. In this case, the base type is `xsd:string`. The `xsd:enumeration` element is used to specify allowed values.

An alternative type of restriction uses the `xsd:pattern` element. The content of the `value` attribute of `xsd:pattern` that defines the allowed values can be a single literal value or a choice of values (as in the following code):

```
<xsd:simpleType name="StatusType">
 <xsd:restriction base="xsd:string">
 <xsd:pattern value="alive|dead" />
 </xsd:restriction>
</xsd:simpleType>
```

Or, it can be a regular expression.

## Declaring Attributes

To declare an attribute, you use the `xsd:attribute` element. The `name` attribute of the `xsd:attribute` element contains the attribute name. The value of an attribute is always a simple type. If the permitted values of the attribute are restricted, the `xsd:simpleType` element is used with the `xsd:restriction` child element to constrain the permitted values of the attribute.

Declaration of an attribute inside an `xsd:complexType` element was shown in Listing 19.4. An alternate approach is to declare the attribute inside an element declaration:

```
<xsd:element name="Person">
 <xsd:complexType >
 <xsd:sequence>
 <xsd:element name="Name" type="xsd:string" />
 <xsd:element name="Citizenship" type="xsd:string" />
 <xsd:element name="Occupation" type="OccupationType" />
 </xsd:sequence>
 <xsd:attribute name="category" type="xsd:string" />
 <xsd:attribute name="status" type="StatusType" />
 </xsd:complexType>
</xsd:element>
```

## Defining Complex and Simple Types

In W3C XML Schema there are two basic types of element content, simple types and complex types.

Simple types contain only text content and have no child elements or any attributes on the element. If an element has one or more attributes or has one or more child elements, it is said to be of complex type.

### Defining Simple Types

Suppose you have a document with the following structure:

```
<memo>
<from>John Smith</from>
<email>JSmith@XMML.com</email>
<to>Peter Roehampton</to>
<emailto>Peter@SVGenius.com</emailto>
<message>Hello Peter. I attach the SVG graphic you wanted to
 see.</message>
</memo>
```

Several elements have simple string content of type `xsd:string`. You can define a simple type, such as for the `email` element, as in this code, if the allowed content is a built-in W3C XML Schema datatype:

```
<xsd:element name="email" type="xsd:string" />
```

However, you might want to define your own simple type. If you want to constrain allowed values for element content, you can use an anonymous simple type definition (that is, using no name attribute on the `xsd:complexType` element), like this:

```
<xsd:element name="from">
 <xsd:simpleType>
 <xsd:restriction base="xsd:string">
 <xsd:enumeration value="John Smith" />
 <xsd:enumeration value="Janet Smith" />
 </xsd:restriction>
 </xsd:simpleType>
</xsd:element>
```

The allowed values for the element content are the strings "John Smith" and "Janet Smith". This would ensure that only authorized senders of email could be identified in the `from` element.

## Defining Complex Types

The type of an element that has either element content or attribute(s) or both is of *complex type*. A complex type can be named or anonymous.

A named complex type definition is referenced using the type attribute of an `xsd:element` element. An anonymous complex type definition is nested inside an `xsd:element`.

If a sequence of elements is the allowed content, as in this code

```
<parent>
<firstChild>content</firstChild>
<secondChild>more content</secondChild>
</parent>
```

then an `xsd:sequence` element is nested within the `xsd:complexType` element and the permitted elements are listed in the allowed order:

```
<xsd:complexType>
 <xsd:sequence>
 <xsd:element name="firstChild" type="xsd:string" />
 <xsd:element name="secondChild" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

Alternatively, if the allowed structures in the instance document were

```
<parent>
<firstChoice>Some content</firstChoice>
</parent>
```

or

```
<parent>
<secondChoice>Some other content</secondChoice>
</parent>
```

this can be expressed in a schema using the `xsd:choice` element:

```
<xsd:complexType>
 <xsd:choice>
 <xsd:element name="firstChoice" type="xsd:string" />
 <xsd:element name="secondChoice" type="xsd:string" />
 </xsd:choice>
</xsd:complexType>
```

Much more to W3C XML Schema exists than has been mentioned in this brief introduction. However, the structures illustrated give you a flavor of the syntax of W3C XML Schema.

## Summary

This lesson presented some of the reasons why XML developers need something with more functionality than traditional DTDs.

You learned how to declare elements and attributes in W3C XML Schema, and you also learned how to define W3C XML Schema simple types and complex types.

# APPENDIX A

## XML Online Resources



The number of online XML-related resources is huge and growing. This appendix lists some resources that you might find useful as you look to build on the knowledge you have gained in *Sams Teach Yourself XML in 10 Minutes*.

### Web Sites

Web sites with useful XML information abound. This section can list only a few.

#### The World Wide Web Consortium

The World Wide Web Consortium (W3C) published the XML 1.0 Recommendation and a large number of associated specifications for languages created in XML or, like XPath, in non-XML syntax but designed to be used with XML documents.

All W3C technical documents—full Recommendations and non-final versions of specifications—can be accessed at [www.w3.org/TR/](http://www.w3.org/TR/).

The following list contains URLs that will take you directly to selected W3C Recommendations for some of the XML technologies discussed in this book.

- The XML 1.0 Recommendation (2nd Edition)—  
[www.w3.org/TR/2000/REC-xml-20001006](http://www.w3.org/TR/2000/REC-xml-20001006)
- The Namespaces in XML Recommendation—  
[www.w3.org/TR/REC-xml-names-19990114](http://www.w3.org/TR/REC-xml-names-19990114)

- The Associating Stylesheets with XML Documents 1.0 Recommendation—[www.w3.org/1999/06/REC-xml-stylesheet-19990629](http://www.w3.org/1999/06/REC-xml-stylesheet-19990629)
- The XML Path Language (XPath) Version 1.0—[www.w3.org/TR/1999/REC-xpath-19991116](http://www.w3.org/TR/1999/REC-xpath-19991116)
- XSL Transformations (XSLT) Version 1.0—[www.w3.org/TR/1999/REC-xslt-19991116](http://www.w3.org/TR/1999/REC-xslt-19991116)
- The Document Object Model (DOM) Level 2 Core Specification—[www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113](http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113)
- The Document Object Model (DOM) Level 2 Events Specification—[www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113](http://www.w3.org/TR/2000/REC-DOM-Level-2-Events-20001113)
- Scalable Vector Graphics (SVG) 1.0 Specification—[www.w3.org/TR/2001/REC-SVG-20010904/](http://www.w3.org/TR/2001/REC-SVG-20010904/)
- XML Linking Language (XLink) Version 1.0—[www.w3.org/TR/2001/REC-xlink-20010627/](http://www.w3.org/TR/2001/REC-xlink-20010627/)
- XPointer Framework—[www.w3.org/TR/xptr-framework/](http://www.w3.org/TR/xptr-framework/)
- XPointer `xpointer()` Scheme—[www.w3.org/TR/xptr-xpointer/](http://www.w3.org/TR/xptr-xpointer/)
- XPointer `xmlns()` Scheme—[www.w3.org/TR/xptr-xmlns/](http://www.w3.org/TR/xptr-xmlns/)
- XPointer `element()` Scheme—[www.w3.org/TR/xptr-element/](http://www.w3.org/TR/xptr-element/)
- XML Schema Part 1: Structures—[www.w3.org/TR/2001/REC-xmlschema-1-20010502/](http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/)
- XML Schema Part 2: Datatypes—[www.w3.org/TR/2001/REC-xmlschema-2-20010502/](http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/)

In addition to the specification documents for the individual technologies discussed in this book, the W3C site provides Web pages that describe ongoing developments, available tools, and other useful information.



These pages can be accessed from the menu of choices at the left of the W3C home page at <http://www.w3.org/>.

## **XML.com**

XML.com is a general XML Web site that includes tutorial articles and general discussion of an extensive range of XML-related topics. XML.com is an excellent site that many people with more than a passing interest in the XML family of technologies visit regularly.

## **XMLHack.com**

The XMLHack.com Web site covers many news-related XML items. It has an archive of news items browsable by subject. Visitors can subscribe to an announcement mailing list giving information on the latest news items and developments on the XMLHack.com Web site.

## **The Apache XML Web Site**

The Apache Foundation has several active XML projects. Information on those projects can be accessed at <http://xml.apache.org/>.

## **Google**

At <http://groups.google.com/>, you can access Usenet discussions on numerous topics, including XML.

## **SVGSpider.com**

Now showing its age, <http://www.SVGSpider.com> is the world's first continuing all-SVG Web site. Other example all-SVG sites can be viewed at <http://www.XMML.com> and <http://www.EditITWrite.com>.

## **Mailing Lists**

Many mailing lists are devoted to general or very specific XML-related topics.

## The XML-DEV Mailing List

The XML-DEV mailing list is very active and includes discussions of any XML-related topic. The level of discussion tends to be fairly high. It isn't typically a good place to ask beginner questions. To subscribe, send an email to [xml-dev-request@lists.xml.org](mailto:xml-dev-request@lists.xml.org) with the word **Subscribe** in the subject line.

## The XSL Mailing List

The XSL mailing list is hosted at <http://www.mulberrytech.com/xsl/xsl-list/index.html>. The XSLT community is a large and active one, and the volume of posts on the XSL list can be overwhelming at times.

## The XSLTalk Mailing List

Hosted at [www.yahoogroups.com/group/XSLTalk](http://www.yahoogroups.com/group/XSLTalk), this mailing list tends to have a Microsoft flavor to its discussions.

## The SVG-Developers Mailing List

Despite reaching W3C Recommendation status as recently as September 2001, SVG has a very active developers' mailing list hosted at <http://www.yahoogroups.com/group/svg-developers/>. Subscription information is available at the site.

The SVG-Developers mailing list is also a very active mailing list, with posts often exceeding 30 per day on a sustained basis.

A mailing list dedicated to the use of SVG on mobile platforms has recently been formed. Further information is located at [www.yahoogroups.com/group/SVG-Mobile](http://www.yahoogroups.com/group/SVG-Mobile).

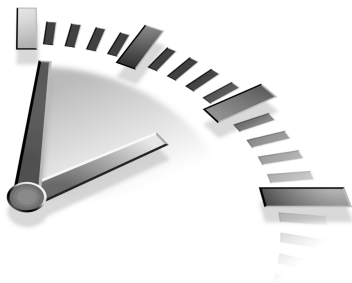
## The www-svg Mailing List

The W3C has a mailing list devoted to SVG. Activity tends to focus on details of the SVG specification, and the volume of posts is much lower than on the SVG-Developers mailing list.

## W3C XML Schema Mailing Lists

The W3C has a mailing list that you can join by sending an email to `xmlschema-dev-request@w3.org` with **subscribe** in the subject line.

There is also an XSD Schema mailing list on YahooGroups.com. Details are found at [www.yahoogroups.com/group/XSDSchema](http://www.yahoogroups.com/group/XSDSchema). To join, send an email to `XSDSchema-subscribe@yahoogroups.com`.



## **APPENDIX B**

# **XML Tools**

This appendix describes some XML tools. First, let's look at some XML editors.

## **XML Editors**

Many XML editors are on the market, with varied functionality and varying prices. The absence of a particular editor in this section does not indicate that it is an inappropriate editor for your use.

### **XML Writer**

XML Writer is a basic but very useful XML editor that is easy to use and relatively inexpensive. Further details are available at [www.xmlwriter.com/](http://www.xmlwriter.com/).

XML Writer features syntax color highlighting and can check XML documents for well-formedness and validity. You can create and save document templates. Examples of this include an XSLT stylesheet with a basic HTML document as literal result elements, and an SVG document with the elements already in place for adding JavaScript code.

At the time of this writing, the current version is 1.2.1 and the upcoming release of version 2.0 has been hinted at for a very long time. This delay means that the current version has no support for W3C XML Schemas, although DTDs are well-supported. Screen shots of version 2.0 are now being shown, which are a good sign that a more full-featured version 2.0 is not too far away. In version 2.0, support for W3C XML Schema is promised along with many other improvements.

A 30-day download of XML Writer is available to enable you to test the program's capabilities.

## XML Spy

XML Spy is a well-featured XML editor with capabilities that go far beyond the capability to edit XML documents. The higher price of the product reflects its multiple capabilities. The current version at the time of writing is 5.0, and further information is available at [www.xmlspy.com](http://www.xmlspy.com).

XML Spy supports XML document editing, either as text or as a logical hierarchy. XML Spy supports XSLT, XML Schema, and XSL-FO, and it also can generate XML from any ODBC-compliant data source. If you use multiple XML languages, you might find XML Spy particularly appropriate to your needs.

A 30-day download is available to enable you to test the capabilities of XML Spy. Given the many aspects of the program, you will likely be able to explore only part of the program in that time.

## XSLT Tools

This section describes several commonly used XSLT tools. XML Spy Suite, mentioned in the preceding section, also includes an XSLT Designer.

### Saxon and Instant Saxon

Saxon is a Java-based XSLT processor. Instant Saxon is a Windows-specific executable version that can be run on Windows 95, 98, Me, and 2000.



**Note** If you want to run Instant Saxon on Windows XP, you will need to download the Microsoft Java Virtual Machine, the JVM, separately from the Microsoft Web site. Visit [www.microsoft.com/java/default.htm](http://www.microsoft.com/java/default.htm) for current information.

The various versions of Saxon—stable and developer versions—and Instant Saxon are described at <http://saxon.sourceforge.net>.

The full Saxon download contains API documentation and examples, as well as source code. Instant Saxon contains an executable with a single HTML Web page in a Zip file.



**Caution** Older versions of the Microsoft JVM can cause Saxon to output blank documents. This problem is now not common, but you might want to check that you have the latest version of the JVM using the Windows Update facility in Internet Explorer.

To install Instant Saxon on a Windows platform, assuming that you have WinZip ([www.winzip.com](http://www.winzip.com)) or a similar utility installed, simply double-click the Zip file and select an appropriately named directory (perhaps C:\Instant Saxon) to install Saxon into. To check whether Instant Saxon has installed correctly, open a command prompt, change to the installation directory you chose, and type **Saxon**. The command prompt window should show a message indicating the version of Instant Saxon that you installed and a series of messages explaining the basic syntax for using Instant Saxon. If you see that message, the installation has been successful.

Likely, you will want to store XML and XSLT files in the Instant Saxon directory or add the appropriate directory to the path environment variable.

## MSXML

Microsoft started development of an “XSL” processor at a time before XSL (in W3C terminology) split into XSLT and XSL Formatting Objects (XSL-FO).



**Note** A lot of practical advice about installing and using the various versions of the MSXML parser is available at [www.netcrucible.com/](http://www.netcrucible.com/).

MSXML version 3 (and above) includes an XML processor and an XSLT processor (among other things). If you have Microsoft Internet Explorer version 5.5 or earlier, you will not have MSXML version 3 installed by default. Earlier versions of MSXML support a Microsoft-specific, non-standard approximation of XSLT. Versions of MSXML earlier than 3 should be avoided to ensure future compatibility with W3C standards.

To download the MSXML parser and XSLT processor, visit [www.microsoft.com/XML/](http://www.microsoft.com/XML/).



**Note** Microsoft has begun to refer to the MSXML software as Microsoft XML Core Services. If you can't find any reference to MSXML 3 or 4, search for the newer term.

Chris Bayes has written an MSXML Sniffer utility that can detect whether you have MSXML version 3 or above installed and whether it is in replace mode (which is what you want for XSLT processing). Go to [www.bayes.co.uk/xml/index.xml?xml/main.xml](http://www.bayes.co.uk/xml/index.xml?xml/main.xml) and look for the link to MSXML Sniffer (left panel of the Web page, at the time of writing).



**Note** Internet Explorer version 6 has MSXML version 3 already installed.

## Xalan

Xalan is a Java-based XSLT processor under ongoing development by the Apache Foundation (<http://xml.apache.org>). Details of the Java version

of the Xalan XSLT processor can be found at <http://xml.apache.org/xalan-j/index.html>. At the time of this writing, Xalan-J, the Java version of Xalan, is at version 2.4.



**Note** Xalan version 1 is no longer supported and has been removed from the [xml.apache.org](http://xml.apache.org) Web site. If your computer uses only early versions of Java, you might be able to download the necessary files to run Xalan 1 by searching in Google.com.

Xalan can be used from the command line, can be used in an applet or servlet, or can be incorporated in a full application.

By default, Xalan-J uses the Xerces XML parser, also from the Apache Foundation. Typically the Xalan download includes the appropriate version of the Xerces parser. If you want to run Xalan with another XML processor, configuration instructions are available from the Apache Web site.

To run the currently supported versions of Xalan-J and Xerces, you need either a Java Runtime or a Java Software Developer's Kit installed on your computer. Version 1.2 or later is required. If you do not already have a suitable Java Runtime or JSDK installed, you can download one from <http://java.sun.com>. Further information on the latest versions is also available there.

Be sure to add the Xalan and Xerces jar files to the classpath on your computer.



**Caution** If you use multiple Java applications that make use of the Xerces XML parser, it is very easy to have multiple versions of Xerces on your computer at one time. If more than one version has been added to the classpath, you might find that the wrong version is accessed first and that unexpected and puzzling errors occur.





**Tip** If you plan to use multiple Java-based applications, create a simple batch file that you run when you open the command window. The batch file can set the path and classpath to settings that are specific to the Java application that you want to run. This avoids potential clashes between different versions of Java software.

## XLink and XPointer Tools

At the time of this writing, XLink and XPointer tools are very limited in number.

The Mozilla 1.0 browser ([www.mozilla.org](http://www.mozilla.org)) implements simple XLink links.

The most full-featured XLink processor is the XLiP processor from Fujitsu ([www.labs.fujitsu.com/free/xlip/en/](http://www.labs.fujitsu.com/free/xlip/en/)), which is a Java application.

At the time of this writing, XLiP was not updated to take into account the July 2002 XPointer working drafts. Despite that, it is the most full-featured XLink and XPointer tool generally available. At the time of this writing, a free evaluation download is available.

XLiP also supports a demo of the Extensible Business Reporting Language, XBRL. To run it, you need the Tomcat server available from <http://jakarta.apache.org/tomcat/>.



## APPENDIX C

# XML Glossary

This appendix provides definitions of selected XML-related terms.

**arc** XLink construct that provides information about how to traverse a pair of resources.

**attribute axis** XPath axis that contains only attribute nodes. The abbreviated syntax for the attribute axis is the @ character, usually followed immediately by the attribute name or a wildcard.

**axis** XPath term referring to the way in which the in-memory representation of an XML document is navigated.

**axis name** The part of a *location step* that specifies which XPath axis is to be used in the location step.

**CDATA section** Character data not intended to be parsed by an XML processor is enclosed in a CDATA section.

**character encoding** A way to represent a character, or set of characters by one or more numeric values.

**character point** Unicode term indicating the numeric representation of a character.

**character-point** XPointer `xpointer()` scheme term indicating a point within a container node whose content is character content.

**child axis** XPath axis that contains element nodes, comment nodes, processing instruction nodes, and text nodes.

**child element** Element that is nested completely within another XML element, which is termed the parent element.

**child node** Node in the XPath child axis.

**code point** Numeric code for a character in character encodings such as Unicode.

**collapsed range** XPointer term referring to a range where the start point and the end point are the same point.

**complex type** One of two types of element content permitted in W3C XML Schemas. An element is said to be of complex type if it has one or more attributes or one or more child elements. An element with only text content is said to be of simple type.

**container node** XPointer term indicating a node within whose content a point is located.

**context** XPath term indicating the starting point for interpretation of a location path or expression.

**context location** XPointer term corresponding to and extending the XPath notion of a context node.

**context node** XPath term referring to the node that navigation starts from.

**covering range** XPointer term designating a range that completely encloses a location.

**current node** XSLT term that often, but not always, refers to the same node as the XPath context node.

**deep copy** Copy that results from copying an element in an XSLT transformation with its content. The `xs1:copy-of` element is used.

**default namespace** A namespace name (also called a namespace URI) declared using a namespace declaration of the form  
`xmlns='namespaceURI'.`

**descendant node** Node in the XPath descendant axis.

**DOCTYPE declaration** Informal synonym for the Document Type Declaration.

**document element** Also called the root element. All elements of an XML document are nested between the start tag and end tag of the document element.

**Document Object Model** A W3C-approved way to model content of XML documents.

**document order** XPath term referring to the order in which elements occur in an XML document. A document precedes a second element in document order if the start tag of the element precedes the start tag of the second element.

**Document Type Declaration** XML 1.0 construct that expresses the element type name of the document element; references the external subset of the DTD, if one exists; and contains the internal subset of the DTD, if one exists.

**Document Type Definition** Often referred to as a DTD. A DTD consists of two parts—the internal subset and the external subset. The internal subset consists of markup declarations that are contained within the DOCTYPE declaration.

**DOM** Abbreviation for the Document Object Model.

**DTD** Abbreviation for the Document Type Definition.

**element content** The content between the start tag and end tag of an element.

**element node** In XPath, node that represents an element in the source XML document.

**element type name** The name of an element type. The element type name of a `<myElement></myElement>` tag pair is `myElement`.

**empty element** XML element that has no content. It may be represented by a start tag and end tag pair with no content (not even a single whitespace character in between) or as a shorthand empty element tag, `<anEmptyTag/>`.

**empty element tag** Shorthand form of expressing a start tag/end tag pair when the element is empty. Instead of writing `<Tag></Tag>`, the empty element tag `<Tag/>` can be used. The empty element tag cannot be used if there is any element content, including a single whitespace character.

**encoding form** Unicode term that defines how a character is represented in bits. XML supports UTF-8 and UTF-16.

**end point** XPointer term indicating the final point that defines a range. See also *start point*.

**end tag** The closing delimiter of an element. Each end tag in a well-formed XML document must have a matching start tag.

**ending resource** XLink term for the resource that is the destination of a link expressed in an XLink linking element in the starting resource.

**evaluation context** XPointer term corresponding to (and extending) the XPath concept of a context. An evaluation context consists of a location (the context location), a nonzero position, a nonzero context size, a set of variable bindings, a library of functions, a namespace binding context, and (where applicable) properties for the values returned by the `here()` and `origin()` functions.

**expanded name** Namespace term for a name consisting of the namespace URI and the local part of the QName of the element node or other node.

**expression** Term used in XPath to express how to address a selected part of the in-memory representation of an XML document. The most commonly used type of XPath expression is the location path.

**external parsed entity** An external entity whose content can be parsed by an XML processor.

**external subset** The part of the DTD that is contained in a separate file from the XML document to which it applies. The location of the external subset is indicated within the Document Type Declaration.

**fragment identifier** An identifier for a part (fragment) of a document, including XML documents. See *XML Pointer Language*.

**general entity** An entity used within the document element. A general entity may be a parsed entity or an unparsed entity.

**here() function** XPointer function that returns the context location.

**i10n** Abbreviation for *localization*.

**i18n** A widely used abbreviation for *internationalization*.

**inbound** XLink term that refers to a link where the linking element is expressed on the ending resource.

**indicative namespace prefix** The namespace prefix typically used with elements from a particular namespace. For example, the indicative namespace prefix for XSLT is `xs1`.

**information item** A part of the XML information set. An item is broadly equivalent to a node in other models.

**information set** An abstract data model that represents the information contained in an XML document as a *set* of information items.

**infoset** Abbreviation for the XML Information Set.

**instance document** A document that is an example of a class of XML documents, whose structure is defined by a schema that can be a DTD or a schema expressed in XML.

**instantiate** XSLT term used to refer to the processing of an XSLT template.

**instruction** XSLT term referring to an XSLT element that is contained within an XSLT template.

**instruction element** Synonym for an XSLT instruction.

**interface** Collection of properties and methods that can be implemented by one or more objects.

**internal subset** The part of the DTD that is contained within an XML document. See also *external subset*.

**link** XLink term for an association between two or more resources.

**linkbase** XLink term, short for link database, that uses extended links of inbound and third-party types.

**linking element** XLink term for an element from a non-XLink namespace that has XLink attributes expressing an XLink link.

**literal result element** An element not in the XSLT namespace that is contained in an XSLT stylesheet. The literal result element is output literally in the result document (output document).

**local part** The final part of a QName that follows the colon character.

**local resource** XLink term for a resource that is an XLink linking element or that has an XLink linking element as its parent.

**location** XPointer term that includes XPath nodes and XPointer points and ranges.

**location path** XPath expression that returns a node set.

**location set** XPointer `xpointer()` scheme term referring to an unordered set of XPointer locations. A location set is a generalization of the XPath notion of a node set.

**location step** A part of an XPath location path that consists of an axis specifier, a node test, and an optional predicate.

**LRE** Abbreviation sometimes used to refer to a literal result element.

**markup declaration** In a Document Type Definition, the declaration of elements, attributes, entities, and so on as being present in the permitted structure of a class of XML documents.

**named template** An XSLT term that refers to an `xsl:template` element that has a `name` attribute and that can be called by name using `xsl:call-template`.

**namespace** In XML, is a collection of names identified by a uniform resource identifier (URI).

**namespace declaration** XML 1.0 term indicating an attribute that associates a namespace prefix with a namespace URI.

**namespace name** Synonym for the namespace URI.

**namespace prefix** The initial part of a `QName` that is followed by a colon character and the local part.

**namespace URI** The unique identifier of an XML namespace. Also called a namespace name. The namespace URI together with the local part form the expanded name of a node.

**NCName** An XML name that does not contain a colon character (:). Both the namespace prefix and the local part of a qualified name are NCNames.

**node** Term used in the Document Object Model and the XML Path Language (XPath) to indicate a logical component of an XML document. Nodes may represent elements, attributes, or other structures present in an XML document. The term node is also used in XPointer.

**node-point** An XPointer `xpointer()` scheme point consisting of a point relative to a container node that can have child nodes.

**node-set** An unordered set of XPath nodes. The result of applying an XPath location path.

**node test** One of three parts of an XPath location step. The first part is the axis specifier, the second is the node test, and finally is an optional predicate. The node test refines the selection of nodes made by the axis specifier.

**origin() function** An XPointer function.

**outbound** An XLink term used to refer to a link with a local starting resource and a remote ending resource.

**output document** A synonym for the result document.

**output tree** An XSLT term; a synonym for the result tree.

**parameter entity** Entity declared and used within the Document Type Definition.

**parent element** Element that contains another XML element nested between its start tag and its end tag (without any intervening level of nesting). The element so nested is termed a child element.

**parsed entity** Entity used within the document element of an XML document. A parsed entity is always a general entity.

**pattern** An XPath expression that evaluates to a node set. Commonly, a pattern is used to specify which nodes a template is applied to.

**point** XPointer `xpointer()` scheme term, indicating a precise point (for example, between two characters) in an XML document. An XPointer point type is broadly equivalent to a DOM Level 2 position.

**pointer** XPointer term indicating a string that conforms to the XPointer Framework specification.

**pointer part** XPointer term referring to part of a pointer that consists of a scheme name and pointer data that conforms to the specification of that scheme.

**position** DOM Level 2 term broadly equivalent to an XPointer point.



**post schema validation infoset** The information set of a document that has been validated using the W3C XML Schema specification by a conforming processor. The post schema validation infoset contains additions to the infoset that describe the results of the validation attempt.

**predicate** Optional part of an XPath location step that filters the node set selected by the axis specifier and the node test.

**principal node type** XPath term that specifies the type of XPath node selected in an axis by default. For example, in the child axis, the element node is the principal node type.

**processing instruction** XML processors pass information to associated applications by means of processing instructions that consist of a target (which identifies the application to which the information is to be passed) and a sequence of characters that is the information passed to the application.

**PSVI** Abbreviation for the post schema validation infoset.

**public identifier** A globally applicable way of identifying the external subset of a DTD. The public identifier is always accompanied by use of a corresponding system identifier.

**qualified name** A name consisting of a namespace prefix, followed by a colon character and then a local part.

**QName** Abbreviation for *qualified name*.

**range** XPointer `xpointer()` scheme term. A range is measured between two points. A range is similar to what can be selected by dragging across XML text onscreen. A range can span more than one node.

**RELAX NG** A schema language expressed in XML syntax. It is an alternative to W3C XML Schemas.

**remote resource** XLink term for a resource that participates in an XLink link and that is addressed using a URI reference.

**result document** The document produced as the result of applying an XSLT stylesheet to a source document. Also called an output document. The result document is produced by serializing the result tree.

**result tree** The in-memory hierarchical structure produced by an XSLT transformation. This structure typically represents an XML, HTML, or other document.

**root element** A synonym for the document element.

**schema** A document that describes the permitted structure of a class of XML documents. In XML 1.0, the Document Type Definition (DTD) is the specified schema. Alternatively, schemas may be expressed in XML syntax, as in W3C XML Schemas or RELAX NG.

**scheme** XPointer term that refers to a pointer data format that has a name and is defined in a (W3C) specification.

**shallow copy** In XSLT, copy made when an element is copied without its content.

**simple type** One of two types of element content allowed in W3C XML Schemas. If an element has only text content and no attributes or child elements, it is of simple type. See also *complex type*.

**source document** XSLT term referring to the XML document to which an XSLT transformation is applied.

**start point** XPointer term indicating the point at the beginning of a node or a range. See also *end point*.

**start tag** The opening delimiter of any content that an element might have. Each start tag (except for the special case of the shorthand tag for an empty element) must have a matching end tag.

**starting resource** XLink term for the resource on which an XLink linking element expresses a link.

**style sheet** Term used in Cascading Style Sheets (CSS) to refer to a set of CSS rules. These rules may be embedded within an HTML or XML file, or they may reside in an external file.

**stylesheet** Term used in XSLT to refer to an XSLT file. Note that it is one word, compared to the two words used in a CSS style sheet. Sometimes called a transformation sheet.

**SVG** Abbreviation for Scalable Vector Graphics. SVG is an application language of XML intended to describe two-dimensional vector graphics.

**system identifier** A URI reference contained in the Document Type Declaration that indicates the location of the external subset of the DTD.

**template** XSLT term indicating a collection of instructions nested in an `xsl:template` element.

**test** XPointer term that corresponds to an XPath node test, generalized to include points and ranges.

**third party** XLink term that refers to a link where the XLink linking element is in neither the starting resource nor the ending resource.

**top-level element** A potentially misleading XSLT term that refers to elements that are child elements of the `xsl:stylesheet` element.

**transformation sheet** A synonym for an XSLT stylesheet.

**traversal** XLink term for following or using an arc in an XLink link.

**Unicode** The encoding scheme used by XML. Unicode has both 8-bit and 16-bit encodings used by XML. It also has supplementary code points that allow about one million characters to be encoded. See [www.unicode.org](http://www.unicode.org) for further information.

**uniform resource identifier** See *URI*.

**unparsed entity** An entity referenced from an attribute declared to be of type ENTITY or ENTITIES. Such an entity is not intended to be parsed by an XML processor.

**URI** Abbreviation for uniform resource identifier. A reference to a resource.

**UTF-8** Unicode term indicating an encoding form that must be supported by conforming XML processors and that encodes character points in 8-bit numbers.

**UTF-16** Unicode term indicating an encoding form that must be supported by conforming XML processors and that encodes character points in 16-bit numbers.

**W3C XML Schema** Officially called simply XML Schema. W3C XML Schema is the W3C language for expressing XML schemas.

**well-formed** XML documents are said to be well-formed when they satisfy the well-formedness criteria, including nesting start and end tags correctly.

**whitespace** In XML, a collective term for the space character (`#x20`), the carriage return character (`#x9`), the line feed character (`#xD`) and the tab character (`#xA`).

**XML Pointer Language** A language specified by the W3C that provides a fragment identifier syntax for XML documents.

**XML Schema** A slightly ambiguous term that can refer, when written as XML Schema, to the W3C XML Schema specification specifically, or, when written as XML schema (initial lowercase), to a single schema or to XML schema languages generically.

**XPath** The XML Path Language. XPath uses a non-XML syntax and is used to address selected parts of a source XML document.

**XPointer** Abbreviation for the XML Pointer Language.

**XPointer framework** W3C specification that provides a framework for the XPointer schema specifications.

**XPointer scheme** One of three syntax options that can be included in XPointer expressions. The three XPointer schemes are `xpointer()`, `xmlns()`, and `element()`.

**XSD Schema** Also called W3C XML Schema.

**XSLT** Extensible Stylesheet Language Transformation language. XSLT is used to transform an XML document, or selected parts of it, into another XML document or a document in another syntax, such as HTML.

**XSLT namespace** The namespace URI for the XSLT namespace is `www.w3.org/1999/XSL/Transform`.

**XSLT template instruction** Any element in the XSLT namespace that occurs inside an `xsl:template` element.

# INDEX



## SYMBOLS

---

' (apostrophe), 25, 57  
\* (asterisk) wildcard, XPath axes, 108  
@ (at sign), XPath (XML Path Language), 108, 111  
+ cardinality operator, 181  
: (colon), 31-32, 93-96  
, (comma), declarations, 163  
{ } (curly brackets), 162-163  
:: (double colons), XPath (XML Path Language) axes, 107  
// (double forward slashes), 125  
" " (double quotation marks), 25, 38  
= (equal sign), 25, 96  
! (exclamation mark), 67-68  
/ (forward slash), 86, 104, 107  
( [opening parenthesis], pointer parts, 181  
. (period), 32, 162  
; (semicolon), 57, 71, 163  
" (single double quotation mark), quot entity reference, 57  
\_ (underscore), names, 31  
- (hyphen), names, 32  
-- (double hyphens), character string, 21  
--> ending delimiter, 20, 36  
---> ending delimiter, 21, 36

<![CDATA[ starting delimiter, 26  
< (left angle bracket), lt entity reference, 57  
< character, none in attribute values, 35  
< > (angle brackets), HTML (Hypertext Markup Language), 9  
> (right angle bracket), gt entity reference, 57  
<? starting delimiter, 21  
<!-- starting delimiter, 20  
> ending delimiter, 21  
] (right square bracket), closing delimiter (DTD internal subsets), 45  
[ (left square bracket), opening delimiter (DTD internal subsets), 45  
]]> ending delimiter, 26  
& (ampersand), amp entity reference or parsed entities, 57  
&# (decimal notation), character references, 71  
&#x (hexadecimal notation), character references, 71

## NUMBERS

---

1-byte ASCII code, 70  
2-byte ASCII code, 70  
16-bit ASCII code, 70-72

## A

- a element (HTML), 173
- A-F (hexadecimal notation), character references, 71
- abbreviated XPath (XML Path Language) syntax, 107-108
- accessing XPath (XML Path Language), 109-112
- Adobe Illustrator Web site, 193
- Adobe SVG Viewer
  - downloading, 190-192
  - Web site, 75
- alert boxes, child nodes, 219
- alert() function, 220
- American Standard Coding for Information Interchange (ASCII)
  - characters, 66-69
  - code (1-byte, 2-byte, 16-bit), 70
- ampersand (&) (amp entity reference or parsed entities), 57
- ancestor axis, 105, 182
- ancestor-or-self axis, 105, 182
- anchors, HTML (Hypertext Markup Language) documents, 178-179
- Anchor.html file (code), 178-179
- angle brackets
  - < >, HTML (Hypertext Markup Language), 9
  - < (left), lt entity reference, 57
  - > (right), gt entity reference, 57
- animations
  - greetings, HelloVector.svg file (code), 191
  - SVG (Scalable Vector Graphics), 189-191
  - text, 192
- Any content content model (DTDs), 47
- Apache XML Web site, 243
- APIs. *See* SAX
- apostrophe (')
  - apos entity reference, 57
  - attributes, 25
- appendChild(newChild) method, 203
- appendData(arg) method, 209
- applications
  - Java, 250-251
  - namespaces, 91
  - parsers, 21
- arcs (inbound, outbound, third-party), 174
- ASCII (American Standard Coding for Information Interchange)
  - characters, 66-69
  - code (1-byte, 2-byte, 16-bit), 70
- asterisk (\*), XPath (XML Path Language) axes, 108
- at sign (@), XPath (XML Path Language), 108, 111
- ATTLIST keyword, 51
- Attr interface (DOM), properties, 208
- attributes
  - ' (apostrophe), 25
  - = (equal sign), 25
  - " " (quotation marks), 25
  - adding to elements, 15
  - attributeType, values, 50
  - axis, 105
  - CDATA value, 50
  - creating, 140-141
  - declaring in DTDs (Document Type Definitions), 50-51
  - default values, specifying in DTDs (Document Type Definitions), 52
  - defaultDeclaration, values, 50-51
  - of elements, 25
  - encoding, 19
  - ENTITIES value, 50
  - ENTITY value, 50
  - external entity references (none), 34-35
  - #FIXED "some ValueInQuotes" value, 51
  - ID value, 50
  - IDREF value, 50
  - IDREFS value, 50
  - #IMPLIED value, 51
  - LessBasicDocument.xml file (code), 234
  - LessBasicDocument.xsd file (code), 235
  - names, 33
  - namespaces, 98-99
  - NMTOKEN value, 50
  - NMTOKENS value, 50
  - nodes, 104-106
  - property, 203

- #REQUIRED value, 51
- “some ValueInQuotes” value, 51
- standalone, 19
- SVG (Scalable Vector Graphics)
  - shapes, 189
- types in DTDs (Document Type Definitions), 51
- unique, 34
- values, 33-35
- version, 18, 117, 148-155
- W3C XML Schema, 233, 237
- well-formed documents (XML), 33-35
- XLink (XML Linking Language), 175
- xlink:actuate, 175
- xlink:href, 175
- xlink:show, 175
- xlink:type, 175
- XPath (XML Path Language), 111-112
- xsd:choice element, 239
- xsd:complexType element, 237-239
- xsd:element element, 239
- xsd:pattern element, 237
- xsd:restriction element, 237
- xsd:sequence element, 239
- xsd:simpleType element, 237
- xsd:string element, 236
- attributeType, values, 50
- axes
  - ancestor, 105, 182
  - ancestor-or-self, 105, 182
  - attribute, 105
  - child, 105
  - descendant, 105
  - descendant-or-self, 105, 182
  - following, 105
  - following-sibling, 105
  - namespace, 105
  - parent, 105, 182
  - preceding, 105
  - preceding-sibling, 106
  - self, 105, 182
  - XPath (XML Path Language), 104-108

## B

- balancing start and end tags, 32-33
- bare names (XPointer), 196
- Basic Multilingual Plane (BMP) (Unicode), 74
- BasicDocument.xml file (code), 233-234
- BasicDocument.xsd file (code), 234
- batch files (Java applications), 251
- Batik viewer, downloading, 190-192
- Bayes, Chris, 249
- BigText.css file (code), 176
- binary files, modeling data as XML, 77-78
- bitmaps
  - images, displaying, 163
  - SVG (Scalable Vector Graphics), 189-190
- bits
  - 16-bit code, 70-72
  - UTF-8, UTF-16, UTF-32 encoding forms (Unicode), 74
- block displays (reports), 166
- BMP (Basic Multilingual Plane) (Unicode), 74
- books, descriptions
  - external parsed entities, 54-55
  - single document entities, 54
- Boolean values, conversion rules, 145
- brackets
  - angle (< >), HTML (Hypertext Markup Language), 9
  - curly ( { } ), 162-163
  - left angle (<), It entity reference, 57
  - left square ( [ ) opening delimiter (DTD internal subsets), 45
  - right angle (>), gt entity reference, 57
  - right square ( ] ) closing delimiter (DTD internal subsets), 45
- browsers. *See* Web browsers
- bytes
  - 1-byte or 2-byte ASCII code, 70
  - sizes, 71

## C

- cardinality
  - elements in DTDs (Document Type Definitions), 48-49
  - operator (+), 181
- Cascading Style Sheets. *See* CSS
- case sensitivity
  - element type names, 6
  - Java, 226
  - names, 31
  - XML (Extensible Markup Language), 31
  - XPath (XML Path Language)
    - axis names, 105-106
- CDATA (character data), 25, 45
  - ]]> ending delimiter, 26
  - <![CDATA[ starting delimiter, 26
  - escape characters, 27
  - text, 27-28
  - value, 50
- Character Map
  - English language, 68
  - foreign languages, 68-69
  - ToolTip, 67
- character references
  - ; (semicolon), 71
  - decimal notation (&#), 71
  - hexadecimal notation (&#x) or (A-F), 71
  - predefined entities, 39
  - well-formed documents (XML), 38-39
- character sequences
  - <!-- starting delimiter, 20
  - <![CDATA[ starting delimiter, 26
  - (double hyphens), character string, 21
  - > ending delimiter, 20-21, 36
  - <? starting delimiter, 21
  - ?> ending delimiter, 21
  - ]]> ending delimiter, 26
  - XML or xml, 59, 96
- character strings, -- (double hyphens), 21
- character-points, 182
- CharacterData interface (DOM), 201, 208-209
- characters
  - 16-bit encoding (code), 71-72
  - ASCII (American Standard Coding for Information Interchange), 66-70
  - bytes, sizes, 71
  - CDATA (character data), 45
  - data, types, 232
  - displaying (code), 75
  - encodings, 66-69
  - English language, 65
  - escape, CDATA sections, 27
  - Explore.xml file, 16-bit character encoding (code), 71-72
  - expressing in hexadecimal and decimal notation (code), 68
  - fonts, 74-75
  - foreign-language, displaying, 71
  - glyphs, 74-75
  - Good.xml file, statement in
    - English and German (code), 71
  - hexadecimal, screen visibility, 66
  - initial, XML names, 31
  - internationalization, 65-69
  - non-initial, XML names, 32
  - parsed data, 58
  - PCDATA (parsed character data), 45
  - references, 68-72
  - serif fonts, 75
  - U+0021 (Unicode), 68
  - Unicode, 19, 70-74
  - XML and internationalization, 69-72
- characterSequence, 21
- child
  - axis, 105
  - nodes, 103, 217-220
- Child elements content model (DTDs), 47
- ChildNodes.svg file (code), 217-219
- circle elements, creating, 210-212
- classes
  - myHandler, output, 228-230
  - W3C XML Schema, 233
- CLASSPATH environment variable, setting, 225-226
- cloneNode(deep) method, 203



## code

Anchors.html file (code), 178-179  
 ASCII (1-byte, 2-byte, or 16.bit),  
 70  
 BasicDocument.xml file, 233-234  
 BasicDocument.xsd file, 234  
 bytes, sizes of, 71  
 characters, 68, 75  
 ChildNodes.svg file, 217-219  
 CreateCircle.svg file, 210-211  
 CreateRectNS.svg file, 213-214  
 CSS (Cascading Style Sheets),  
 162-163, 176  
 CSSInformation.html file,  
 169-170  
 CSSInformation.xml file, 168  
 CSSInformation.xsl file, 168-169  
 deep copy, creating, 134  
 DoctypeProps.svg file, 215-216  
 Document1.xml file, 175-176  
 Document2.xml file, 176  
 Documents.html file, 147-148  
 Documents.xml file, 145-146  
 Documents.xsl file, 146-147  
 Documents2.html file, 151-152  
 Documents2.xml file, 149-151  
 Documents3.html file, 154-155  
 Documents3.xsl file, 152-154  
 Documents4.html file, 157-158  
 Documents4.xsl file, 155-157  
 DocumentType object, displaying  
 properties, 215-216  
 Explore.xml file, 16-bit character  
 encoding, 71-72  
 Good.xml file, statement in  
 English and German, 71  
 HelloVector.svg file, 191  
 HTML (Hypertext Markup  
 Language), structured data,  
 78-79  
 HTMLTemplate.xsl file, 122  
 LessBasicDocument.xml file, 234  
 LessBasicDocument.xsd file, 235  
 links, opening Web browser win-  
 dows, 177  
 Mouseover.svg file, 193-194  
 myHandler class output, 228-230  
 myHandler.java file, 226-228  
 MySite.css file, 167-168

myXML.xml file, displaying char-  
 acters, 75  
 property, 205  
 PurchaseOrder.xml file, 131-132  
 PurchaseToOrder.xsl file, 132  
 PurchaseToOrder2.xsl file, 134  
 Reference.svg file, 196-197  
 Reports.css file, 164  
 Reports.xml file, 164  
 Reports2.css file, 165-166  
 rules, 10-11  
 SAXSource.xml file, 223  
 SingleEntity.xml file, book  
 descriptions, 54  
 SplitEntities.xml file, book  
 descriptions, 54  
 statements in English and  
 German, 71  
 supplementary code points  
 (Unicode), 73-74  
 SVG (Scalable Vector Graphics),  
 188  
 SVGLink.svg file, 194-195  
 Title.xml file, book descriptions,  
 55  
 UKShirts.xml file, 137  
 UKShirtsToUS.xsl file, 140-141  
 USShirts.xml file, 136-137  
 USShirtToUK.xsl file, 137-139  
 XLink (XML Linking Language),  
 176-177  
 XML, 80-82, 189  
 XMMLNews.html file, 127-128  
 XMMLNews.xml file, 126  
 XMMLNews.xsl file, 126-127  
 XMMLOrder.xml file, 133  
 XMMLOrder2.xml file, 134-135  
 XMMLReports.html file, 125  
 XMMLReports.xml file, 122-123  
 XMMLReports.xsl file, 124  
 XPath (XML Path Language),  
 106-109  
 XSLTMessage.html file, 121  
 XSLTMessage.xml file, 118  
 XSLTMessage.xsl file, 119  
 collapsed ranges, 183  
 colon (:)  
   double (::), XPath (XML Path  
   Language) axes, 107  
   names, 31-32

- namespace declarations, 96
- Qnames (qualified names), 93-94
- comma (,), declarations, 163
- Comment interface (DOM), 203
- comment node, 106
- comments
  - <!-- starting delimiter, 20
  - (double hyphens), character string, 21
  - > ending delimiter, 20-21, 36
  - document prolog, 20-21
  - well-formed documents (XML), 36
- comparing
  - HTML (Hypertext Markup Language)
    - and *XLink (XML Linking Language)*, 173
    - and *XML (Extensible Markup Language)*, 12-13, 78-81
  - SAX (Simple API for XML) and DOM (Document Object Model), 221
  - stylesheets and style sheets, 159
- complex types of W3C XML Schema, 233, 239-240
- conditional processing of data, 143-144
- conditional sections, DTDs (Document Type Definitions), 46
- constraints, well-formed documents (XML), 30
- container nodes, 182
- content
  - after document element end tag, 28
  - models (DTDs), 46-47
  - and presentation, separating, 13, 160-161
  - in XML documents, restructuring, 129-131
- ContentHandler interface, 223
- context nodes, XPath (XML Path Language), 104
- conversion rules for Boolean values, 145
- converted ranges, 184
- copying elements, 131-135
- Core interfaces, DOM (Document Object Model) Level 2, 199-201
- CorelDraw Web site, 193
- count() function, 113-114
- createAttribute(name) method, 206
- createAttributeNS(namespaceURI, qualifiedName) method, 206
- createCDATASection(data) method, 206
- CreateCircle.svg file (code), 210-211
- createComment(data) method, 206
- createDocument(namespaceURI, qualifiedName, doctype) method, 205
- createDocumentFragment() method, 206
- createDocumentType(qualifiedName, publicId, systemId) method, 205
- createElement() method, 210-212
- createElement(tagName) method, 206
- createElementNS() method, 213-214
- createElementNS(namespaceURI, qualifiedName) method, 206
- createEntityReference(name) method, 206
- createProcessingInstruction(target,data) method, 206
- CreateRectNS.svg file (code), 213-214
- createTextNode(data) method, 206
- creating
  - attributes, 140-141
  - circle elements, 210-212
  - DOM (Document Object Model) elements, 210-214
  - elements, 135-139
  - HTML (Hypertext Markup Language)
    - lists*, 122-125
    - pages*, 118-122
    - tables*, 126-128
  - rect elements, 214
  - rectangles (code), 213-214
  - SVG (Scalable Vector Graphics), 191-193
  - vocabularies, 11-12
  - W3C XML Schema, 233
- CSS (Cascading Style Sheets), 13
  - CSSInformation.html file (code), 169-170
  - CSSInformation.xml file (code), 168

CSSInformation.xml file (code),  
     168-169  
 data, styling, 164-167  
 documents (XML), associating  
     with style sheets, 161  
 HTML (Hypertext Markup  
     Language), 160, 170-171  
 MySite.css file (code), 167-168  
 reports, displaying, 164-166  
 Reports.css file (code), 164  
 Reports.xml file (code), 164  
 Reports2.css file (code), 165-166  
 style sheets, 159-163, 167, 189  
 text appearance, controlling  
     (code), 176  
 Web sites, separating content and  
     presentation, 160-161  
 XSLT (Extensible Stylesheet  
     Language Transformations),  
     HTML (Hypertext Markup  
     Language) output, 167-170  
 CSSInformation.html file (code),  
     169-170  
 CSSInformation.xml file (code), 168  
 CSSInformation.xml file (code),  
     168-169  
 CSSStyleSheet.css file, 161  
 curly brackets ({}), 162-163

## D

### data

CDATA (character data), 45  
 conditional processing, 143-144  
 employee, XML code, 80  
 hierarchical in XML, 83  
 human readable, 7-8  
 loosely structured in XML, 83-85  
 modeling as XML, 77-85  
 models, XML Information Set  
     (Infoset), 87  
 objects, hierarchical structure, 85  
 parsed, 58  
 PCDATA (parsed character data),  
     45  
 property, 208  
 relational-type, modeling, 81-83  
 sorting, 143-158  
 structured, 78-81  
 styling (CSS), 164-167  
 types, 232  
 unparsed, types of, 62  
 XML models (W3C), 85-86  
 xsl:choose element, 144, 149-152  
 xsl:for-each element, 155  
 xsl:if element, 144-148  
 xsl:otherwise element, 149  
 xsl:sort element, 152-158  
 xsl:template element, 144  
 xsl:when element, 149  
 databases, RDBMS (relational data-  
     base-management system), 81  
 decimal notations, characters  
     &#, references, 71  
     expressing (code), 68  
 declarations  
     , (comma), 163  
     { } (curly brackets), 163  
     ; (semicolon), 163  
     DOCTYPE, 22-23, 44  
     document prolog, 18-19  
     DTDs (Document Type  
         Definitions), external or internal  
         subsets, 22  
     encoding attribute, 19  
     markup, 19-20, 38, 43-44  
     namespaces, 96-98  
     standalone attribute, 19  
     SVG (Scalable Vector Graphics),  
         191  
     text, external parsed entities, 61  
     version attributes, 18  
     W3C XML Schema, 233  
 declarative animations (SVG), 191  
 declaring  
     attributes, 50-51, 237  
     elements, 46-49, 233-237  
     entities, 36, 52-53  
     notations, 62  
     predefined entities in well-formed  
         documents (XML), 39  
 deep copy (elements), 134-135  
 defaults  
     namespaces, 97  
     values of attributes, specifying, 52  
 defaultDeclaration, values, 50-51

## defining

- document structures in DTDs (Document Type Definitions), 42-43
- W3C XML Schema, 233
  - complex types*, 239-240
  - simple types*, 238

## definitions

- entity, 53
- links, 174
- namespace, 89
- referencing with XPointer (XML Pointer Language), 196-198
- rule, 162
- schemas, 42
- transformations, 9

## deleteData(offset, count) method, 209

## delimiters

- <!-- starting, 20
- > ending, 20, 36
- > ending, 21, 36
- <? starting, 21
- ?> ending, 21
- closing or opening, 26, 45

## descendant axis, 105

## descendant-or-self axis, 105, 182

## description element, 59

## descriptions of books, 54-55

## developers, SVG-Developers mailing list Web site, 244

## displaying

- bitmap images, 163, 189
- characters, code, 75
- child nodes, 217-220
- documents, 148-149
- DocumentType, 215-217
- foreign-language characters, 71
- HTML (Hypertext Markup Language) documents, 170-171
- reports, 164-166
- vector images, 163, 189

## DOCTYPE declarations, 22-23, 44

## doctype property, 206

## DoctypeProps.svg file (code), 215-216

## Document interface (DOM), 205-206

Document Object Model. *See* DOMDocument Type Definitions. *See* DTDs

## Document1.xml file (code), 175-176

## Document2.xml file (code), 176

## Document3.xml file (code), 177

## documentElement property, 206

## documents (XML)

- attributes, creating, 140-141
- BasicDocument.xml file (code), 233-234
- BasicDocument.xsd file (code), 234
- CDATA sections, 25-28
- content restructuring, 129-131
- CSS (Cascading Style Sheets), 161
- displaying, 148-149
- DOCTYPE declarations, 22-23
- document element end tag, content after, 28
- DTDs (Document Type Definitions), 23
- element end tag, content after, 28
- elements, 8, 17-18, 24-25, 131-139
- entity, 53
- external parsed entities, book descriptions, 54-55
- fragments, XPointer (XML Pointer Language), 178-180
- HTML (Hypertext Markup Language), 170-171, 178-179
- HTMLTemplate.xml file (code), 122
- human readable, 7-8
- instance, 44, 233
- LessBasicDocument.xml file (code), 234
- LessBasicDocument.xsd file (code), 235
- markup declarations, 43-44
- namespaces, 90, 99-101
- nodes, hierarchy, 103-104
- optional content, 17
- parsers, 21
- parts of, 17
- prolog, 17-21
- PurchaseOrder.xml file (code), 131-132
- PurchaseToOrder.xml file (code), 132
- PurchaseToOrder2.xml file (code), 134
- purposes of, 5-7
- SAX (Simple API for XML), 222

- SAXSource.xml file (code), 223
- schemas, definition, 42
- shared, 41-43
- single document entities, book descriptions, 54
- structure, defining in DTDs (Document Type Definitions), 42-43
- SVG (Scalable Vector Graphics), text animations, 192
- text, linking in Mozilla browser, 176-177
- transforming, 115
- UKShirts.xml file (code), 137
- UKShirtsToUS.xsl file (code), 140-141
- USShirts.xml file (code), 136-137
- USShirtToUK.xsl file (code), 137-139
- UTF-8 character encoding, 19
- UTF-16 character encoding, 19
- valid, 43
- well-formed, 29-40
- XML files, 84
- XMMLOrder.xml file (code), 133
- XMMLOrder2.xml file (code), 134-135
- XPath (XML Path Language), 86, 103
- XSLT (Extensible Stylesheet Language Transformations), 115
- Documents.html file (code), 147-148
- Documents.xml file (code), 145-146
- Documents.xsl file (code), 146-147
- Documents2.html file (code), 151-152
- Documents2.xsl file (code), 149-151
- Documents3.html file (code), 154-155
- Documents3.xsl file (code), 152-154
- Documents4.html file (code), 157-158
- Documents4.xsl file (code), 155-157
- DocumentType interface, properties, 207, 215-217
- DocumentType object, displaying properties (code), 215-216
- DOM (Document Object Model), 85-86
  - appendChild(newChild) method, 203
  - appendData(arg) method, 209
  - Attr interface properties, 208
  - attributes property, 203
  - CharacterData interface, 208-209
  - child nodes, information retrieval (code), 217-219
  - ChildNodes.svg file (code), 217-219
  - cloneNode(deep) method, 203
  - code property, 205
  - Comment interface, 203
  - createAttribute(name) method, 206
  - createAttributeNS(namespaceURI, qualifiedName) method, 206
  - createCDATASection(data) method, 206
  - CreateCircle.svg file (code), 210-211
  - createComment(data) method, 206
  - createDocument(namespaceURI, qualifiedName, doctype) method, 205
  - createDocumentFragment() method, 206
  - createDocumentType(qualifiedName, publicId, systemId) method, 205
  - createElement(tagName) method, 206
  - createElementNS(namespaceURI, qualifiedName) method, 206
  - createEntityReference(name) method, 206
  - createProcessingInstruction(target, data) method, 206
  - CreateRectNS.svg file (code), 213-214
  - createTextNode(data) method, 206
  - data property, 208
  - deleteData(offset, count) method, 209
  - doctype property, 206
  - DoctypeProps.svg file (code), 215-216
  - Document interface, 205-206
  - documentElement property, 206
  - DocumentType interface, properties, 207

- DOMException interface, properties, 205
- DOMImplementation interface, methods, 205
- Element interface, 207-208
  - elements, creating, 210-214
  - firstChild property, 203
  - getAttribute(name) method, 207
  - getAttributeNode(name) method, 207
  - getAttributeNodeNS(namespaceURI, localName) method, 207
  - getAttributeNS(namespaceURI, localName) method, 207
  - getElementsByTagName(tagname) method, 206-207
  - getElementsByTagNameNS(namespaceURI, localName) method, 206-207
  - getNamedItem(name) method, 204
  - getNamedItemNS(namespaceURI, localName) method, 204
  - hasAttribute(name) method, 207
  - hasAttributeNS(namespaceURI, localName) method, 208
  - hasAttributes() method, 203
  - hasChildNodes() method, 203
  - hasFeature(feature, version) method, 205
  - importNode(importedNode, deep) method, 206
  - information retrieval, 215-220
  - insertBefore(newChild, refChild) method, 203
  - insertData(offset, arg) method, 209
  - interfaces, 85, 199-201
  - isSupported(feature, value) method, 203
  - item(index) method, 204
  - lastChild property, 203
  - length property, 204
- Level 2 Core
  - Events Specification Web site*, 242
  - interfaces*, 199-201
  - Specification Web site*, 242
  - specifications*, 199
- localName property, 203
  - methods, 199
  - name property, 208
- NamedNodeMap interface (methods or properties), 204
- Node interface, 201-204
- NodeList interface, 204
- nodeName property, 203
- nodeType property, 203
- nodeValue property, 203
- normalize() method, 204
- notations property, 207
- objects, 199-201
- prefix property, 203
- properties, 199
- removeAttribute(name) method, 208
- removeAttributeNode(oldAttr) method, 208
- removeAttributeNS(namespaceURI, localName) method, 208
- removeChild(oldChild) method, 204
- removeNamedItem(name) method, 204
- removeNamedItemNS(namespaceURI, localName) method, 204
- replaceChild(newChild, oldChild) method, 204
- replaceData(offset, count, arg) method, 209
- and SAX (Simple API for XML), comparing, 221
- scripting with SVG (Scalable Vector Graphics), 189
- setAttribute(name, value) method, 208
- setAttributeNode(newAttr) method, 208
- setAttributeNodeNS(newAttr) method, 208
- setAttributeNS(namespaceURI, localName) method, 208
- setNamedItem(arg) method, 204
- setNamedItemNS(namespaceURI, localName) method, 204
- splitText(offset) method, 209
- substringData(offset, count) method, 209
- systemID property, 207
- tagname property, 207

- Text interface, 203, 209
  - value property, 208
- DOMException interface, properties, 205
- DOMImplementation interface, methods, 205
- double forward slashes (//), 125
- double quotation marks (“ ”)
  - attributes, 25
  - replacement text, 38
  - single (“), quot entity reference, 57
- downloading
  - Adobe SVG Viewer, 190-192
  - Batik viewer, 190-192
  - JSDK (Java Software Development Kit), 224
  - JVM (Java Virtual Machine), 247
  - MSXML parser, 249
  - Xerces parser, 224
- drop shadows, SVG (Scalable Vector Graphics) filters, 198
- DTDHandler interface, 223
- DTDs (Document Type Definitions)
  - Any content content model, 47
  - attributes, 50-52
  - attributeType, values, 50
  - CDATA (character data), 45
  - Child elements content model, 47
  - conditional sections, 46
  - content models, 46-47
  - defaultDeclaration, values, 50-51
  - DOCTYPE declarations, 22-23, 44
  - document structure, defining, 42-43
  - elements, declaring, 46-49
  - Empty element content model, 46
  - entities, declaring, 52
  - external parsed entities, 55
  - external subsets, 37, 44-45
  - instance documents, 44
  - internal subsets, 37, 45
  - limitations, 232-233
  - markup declarations, 43-44
  - Mixed content content model, 47
  - parameter entities, 57, 62-64
  - parameter entity references, 37
  - PCDATA (parsed character data), 45

- shared documents, 41-43
- subsets (external or internal), 22
- Text only content model, 46
- valid XML documents, 43
- W3C XML Schema, 231-240, 245
- XML processors, 40-42

## E

- editors
  - XML Spy, 247
  - XML Writer, 246
- Element interface (DOM), 207-208
- element( ) scheme, 186
- elements
  - a (HTML), 173
  - attributes, 15, 25
  - cardinality in DTDs (Document Type Definitions), 48-49
  - circle, creating, 210-212
  - copying, 131-135
  - creating, 135-139
  - declaring in DTDs (Document Type Definitions), 46-49
  - description, 59
  - document element end tag, content after, 28
  - documents (XML), 17-18, 24
  - DOM (Document Object Model), creating, 210-214
  - link, accessing CSS (Cascading Style Sheets), 167
  - linking, 174
  - literal result, 120
  - meta, 121
  - nested, 24-25, 234-235
  - nodes, 103-106
  - qualified names, 94
  - rect, creating, 214
  - root, 24
  - start and end tags, balancing, 32-33
  - stylesheet, 116
  - SVG (Scalable Vector Graphics), 188-191
  - <svg>, child nodes, 219
  - title, 120

- top-level (XSLT), 117
- transform, 116
- type names, 6, 32, 91-93, 162
- of W3C XML Schema, declaring, 233-237
- well-formed documents (XML), 32-33
- XLink (XML Linking Language), 174-175
- in XML documents, naming, 8
- XPath (XML Path Language),
  - accessing, 109-110
- xsd:choice, 239
- xsd:complexType, 237-239
- xsd:element, 239
- xsd:pattern, 237
- xsd:restriction, 237
- xsd:sequence, 239
- xsd:simpleType, 237
- xsd:string, 236
- xsl:apply-templates, 120, 124, 139-141
- xsl:attribute, 140-141
- xsl:attribute-set, 118
- xsl:choose, 143-144, 149-152
- xsl:copy, 131-133, 141
- xsl:copy-of, 134
- xsl:decimal-format, 118
- xsl:element, 137-139
- xsl:for-each, sorting data, 155
- xsl:if, 143-148
- xsl:import, 117
- xsl:include, 117
- xsl:key, 118
- xsl:namespace-alias, 118
- xsl:otherwise, sorting data, 149
- xsl:output, 117
- xsl:param, 118
- xsl:preserve-space, 118
- xsl:sort, 143, 152-158
- xsl:strip-space, 118
- xsl:stylesheet, 117
- xsl:template, 117-119, 125, 144
- xsl:text, 125
- xsl:transform, version attribute, 117
- xsl:value-of, 120-121, 133, 139
- xsl:variable, 118
- xsl:when, sorting data, 149
- email, XML-DEV mailing list, 244
- employees (XML code)
  - data, 80
  - records, modeling, 81-82
- Empty element content model (DTDs), 46
- encoding
  - 16-bit characters (code), 71-72
  - attribute, 19
  - characters (Unicode), 66-69, 73
  - forms (Unicode), 74
  - UTF-8 or UTF-16, 19, 74
  - UTF-32 encoding form (Unicode), 74
- end-point( ) function, 184
- end points, 183
- end tags, 14, 28, 32-33
- ending delimiters
  - >, 20, 36
  - >, 21, 36
  - ?>, 21
  - ] (right square bracket), DTD internal subsets, 45
  - ]]>, 26
- ending resources (XLink), 174
- English language, 65, 68, 71-73
- entities
  - amp reference (&), 57
  - apos reference ('), 57
  - declaring, 36, 52-53
  - definition, 53
  - description element, 59
  - document, 53
  - external parsed, 54-56
  - external references, none for attributes, 34-35
  - general, 58
  - gt reference (>), 57
  - lt reference (<), 57
  - names, 54, 59
  - parameter, 38, 56-57, 62-64
  - parameter references in DTDs (Document Type Definitions), 37
  - parsed, 36-37, 56-61
  - predefined, 39, 57
  - quot reference ("), 57
  - references, 56-57
  - single document, book descriptions, 54



- SingleEntity.xml file, book descriptions (code), 54
  - SplitEntities.xml file, book descriptions (code), 54
  - SVG (Scalable Vector Graphics), 53, 60
  - Title.xml file, book descriptions (code), 55
    - unparsed, 57, 61-62
  - ENTITIES value, 50
  - ENTITY value, 50
  - entityName, unparsed entities, 62
  - environment variables, 225-226
  - equal sign (=), 25, 96
  - ErrorHandler interface, 223
  - escape characters, CDATA sections, 27
  - events (SAX), parsing, 222-223
  - exclamation mark (!), 67-68
  - Explore.xml file, 16-bit character encoding (code), 71-72
  - expressions, location paths, 107
  - extended links (XLink), 172-173
  - Extensible Business Reporting Language (XBRL), 251
  - Extensible Markup Language. *See* XML
  - Extensible Stylesheet Language Transformations. *See* XSLT
  - external entity references, none for attributes, 34-35
  - external parsed entities, 36-37, 54-61
  - external subsets of DTDs (Document Type Definitions), 22, 37, 44-45
- ## F
- 
- files
    - Anchors.html (code), 178-179
    - BasicDocument.xml (code), 233-234
    - BasicDocument.xsd (code), 234
    - batch, Java application, 251
    - BigText.css (code), 176
    - binary, modeling data as XML, 77-78
    - ChildNodes.svg (code), 217-219
    - CreateCircle.svg (code), 210-211
    - CreateRectNS.svg (code), 213-214
    - CSSInformation.html (code), 169-170
    - CSSInformation.xml (code), 168
    - CSSInformation.xsl (code), 168-169
    - CSSStyleSheet.css, 161
    - DoctypeProps.svg (code), 215-216
    - Document1.xml (code), 175-176
    - Document2.xml (code), 176
    - Document3.xml (code), 177
    - documents, known as, 84
    - Documents.html (code), 147-148
    - Documents.xml (code), 145-146
    - Documents.xsl (code), 146-147
    - Documents2.html (code), 151-152
    - Documents2.xsl (code), 149-151
    - Documents3.html (code), 154-155
    - Documents3.xsl (code), 152-154
    - Documents4.html (code), 157-158
    - Documents4.xsl (code), 155-157
    - Explore.xml, 16-bit character encoding (code), 71-72
    - Good.xml, statement in English and German (code), 71
    - HelloVector.svg (code), 191
    - HTMLTemplate.xsl (code), 122
    - LessBasicDocument.xml (code), 234
    - LessBasicDocument.xsd (code), 235
    - Mouseover.svg (code), 193-194
    - myHandler.java (code), 226-228
    - MySite.css (code), 167-168
    - myXML.xml, displaying characters (code), 75
    - PurchaseOrder.xml (code), 131-132
    - PurchaseToOrder.xsl (code), 132
    - PurchaseToOrder2.xsl (code), 134
    - Reference.svg (code), 196-197
    - Reports.css (code), 164
    - Reports.xml (code), 164
    - Reports2.css (code), 165-166
    - SAXSource.xml (code), 223
    - SingleEntity.xml, book descriptions (code), 54

- SplitEntities.xml, book descriptions (code), 54
- SVGLink.svg (code), 194-195
- Title.xml, book descriptions (code), 55
- UKShirts.xml (code), 137
- UKShirtsToUS.xml (code), 140-141
- USShirts.xml (code), 136-137
- USShirtToUK.xml (code), 137-139
- XMMLNews.html (code), 127-128
- XMMLNews.xml (code), 126
- XMMLNews.xml (code), 126-127
- XMMLOrder.xml (code), 133
- XMMLOrder2.xml (code), 134-135
- XMMLReports.html (code), 125
- XMMLReports.xml (code), 122-123
- XMMLReports.xml (code), 124
- XSLTMessage.html (code), 121
- XSLTMessage.xml (code), 118
- XSLTMessage.xml (code), 119
- filters, SVG (Scalable Vector Graphics), 190, 196-198
- firstChild property, 203
- #FIXED “some ValueInQuotes” value, 51
- following axis, 105
- following-sibling axis, 105
- fonts, 74-75
- foreign languages
  - Character map, 68
  - characters, displaying, 71
- forms
  - encoding (Unicode), 74
  - shorthand (XPath), 196
  - W3C XForms specification, 8
- forward slash (/)
  - double (//), 125
  - root nodes, 86, 104
  - XPath (XML Path Language) axes, 107
- fragments of documents, XPath (XML Pointer Language), 178-180

- frameworks, XPath (XML Pointer Language) Framework
  - element() scheme, 186
  - schemes, 181
  - xmlns() scheme, 185-186
  - xpointer() scheme, 182-184
- Fujitsu Web site, 251

- functions
  - alert(), 220
  - count(), 113-114
  - end-point(), 184
  - getChildNodes(), 220
  - getDoctype(), 216
  - here(), 184
  - Initialize(), 216
  - name(), 120
  - origin(), 184
  - position(), 113
  - range(), 183
  - range-inside(), 184
  - range-to(), 184
  - start-point(), 184
  - string-range(), 183
  - XPath (XML Path Language), 112
  - xpointer() scheme, 183-184

## G

- general entities, 58
- German language, 68, 71
- getAttribute(name) method, 207
- getAttributeNode(name) method, 207
- getAttributeNodeNS(namespaceURI, localName) method, 207
- getAttributeNS(namespaceURI, localName) method, 207
- getChildNodes() function, 220
- getDoctype() function, 216
- getElementById() method, 217
- getElementsByTagName(tagname) method, 206-207
- getElementsByTagNameNS(namespaceURI, localName) method, 206-207
- getNamedItem(name) method, 204
- getNamedItemNS(namespaceURI, localName) method, 204

glyphs, 74-75  
 Good.xml file, statement in English  
   and German (code), 71  
 Google Web site, 243  
 graphics. *See* SVG  
 greetings, animated, HelloVector.svg  
   file (code), 191  
 gt entity reference (>), 57

## H

hasAttribute(name) method, 207  
 hasAttributeNS(namespaceURI,  
   localName) method, 208  
 hasAttributes( ) method, 203  
 hasChildNodes( ) method, 203  
 hasFeature(feature, version) method,  
   205  
 headlines, XMMLNews.html file  
   (code), 127-128  
 HelloVector.svg file (code), 191  
 here( ) function, 184  
 hexadecimal  
   characters, screen visibility, 66  
   notations, 68, 71  
 hierarchies  
   data in XML, 83  
   DOM (Document Object Model)  
     interfaces and objects, 200  
     nodes, 103-104  
   structures, data objects, 85  
   XPath (XML Path Language),  
     XML documents, 103  
 histories, SAX (Simple API for XML),  
   221  
 HTML (Hypertext Markup Language)  
   < > (angle brackets), 9  
   a element, 173  
   Anchors.html file (code), 178-179  
   CSS (Cascading Style Sheets),  
     160, 167-170  
   documents, 170-171, 178-179  
   hyperlinks and XLink (XML  
     Linking Language), comparing,  
     173  
   lists, creating, 122-125  
   pages, creating, 118-122  
   structured data (code), 78-79

tables, creating, 126-128  
 tags, 9  
 and XML (Extensible Markup  
   Language), comparing, 12-13,  
   78-81  
 XSLT (Extensible Stylesheet  
   Language Transformations),  
   167-170  
 HTMLTemplate.xml file (code), 122  
 human readable (XML), 7-8  
 hyperlinks  
   browser windows, opening (code),  
     177  
   definition, 174  
   extended (XLink), 172-173  
 HTML (Hypertext Markup  
   Language) and XLink (XML  
   Linking Language), comparing,  
   173  
   inbound arc, 174  
   outbound arc, 174  
   simple (XLink), 172-176  
   third-party arc, 174  
   transversal, 174  
   XLink (XML Linking Language),  
     173-174, 194-196  
 Hypertext Markup Language. *See*  
   HTML  
 hyphen (-), names, 32

## I

i18n (internationalization), 65-72  
 ID value, 50  
 identifiers, URIs (uniform resource  
   identifiers), 23, 94-95, 116  
 IDREF value, 50  
 IDREFS value, 50  
 IGNORE option, conditional sections,  
   46  
 Illustrator (Adobe), Web site, 193  
 images, bitmap or vector, 163, 189  
 #IMPLIED value, 51  
 importNode(importedNode, deep)  
   method, 206  
 inbound arc (XLink), 174  
 INCLUDE option, conditional sections,  
   46

- information retrieval, DOM
  - (Document Object Model), 215-220
- infoset (XML Information Set), data models, 87
- initial characters of XML names, 31
- Initialize() function, 216
- insertBefore(newChild, refChild) method, 203
- insertData(offset, arg) method, 209
- installations, Java (Path environment variable), 225
- installing
  - JSDK (Java Software Development Kit), 224
  - SAX (Simple API for XML) parsers, 224-226
  - Xerces parser, 224
- instance documents
  - and DTDs (Document Type Definitions), 44
  - W3C XML Schema, 233
- Instant Saxon (XSLT), 247-248
- instructions, processing
  - <? starting delimiter, 21
  - ?> ending delimiter, 21
  - document prolog, 20-21
  - xml-stylesheet, 21, 161
- interfaces
  - Attr (DOM), properties, 208
  - CharacterData (DOM), 201, 208-209
  - COM Level 2 Core, 199-201
  - Comment (DOM), 203
  - ContentHandler, 223
  - Document (DOM), 205-206
  - DocumentType, properties, 207, 215-217
  - DOM (Document Object Model), 85, 199-209
  - DOMException, properties, 205
  - DOMImplementation, methods, 205
  - DTDHandler, 223
  - Element (DOM), 207-208
  - ErrorHandler, 223
  - NamedNodeMap (DOM), methods or properties, 204
  - Node (DOM), 201-204
  - NodeList (DOM), 204

- SAX (Simple API for XML) 2, 223
- Text (DOM), 203, 209
- internal parsed entities, 56, 59-60
- internal style sheets, SVG (Scalable Vector Graphics), 189
- internal subsets, DTDs (Document Type Definitions), 22, 37, 45
- internationalization (i18n), 65-72
- Internet Explorer, 175, 249
- ISO (International Organization for Standardization), ISO/IEC 10646, 73
- isSupported(feature, value) method, 203
- item(index) method, 204

---

## J-K

---

- jargon
  - W3C XML Schema, 233
  - XLink (XML Linking Language), 173-174
- Jasc WebDraw Web site, 192
- Java
  - applications, 250-251
  - case sensitivity, 226
  - installations, Path environment variable, 225
  - myHandler class, output, 228-230
  - myHandler.java file, 226-228
- JSDK (Java Software Development Kit), downloading and installing, 224
- JVM (Java Virtual Machine), 247-248
- keywords
  - ATTLIST, 51
  - NDATA, 62
  - PUBLIC, 23

---

## L

---

- languages. *See also* HTML; XLink; XML; XPath; XPointer; XSLT
  - BMP (Basic Multilingual Plane) (Unicode), 74
  - English, 65, 68, 73
  - foreign, 68

- German, 68
- markup, 9-10, 77-78
- meta, 10-12
- programming, Unicode, 73
- schemas, 231
- Web support, 65
- WML (Wireless Markup Language), 116
- XBRL (Extensible Business Reporting Language), 251
- lastChild property, 203
- left angle bracket ( [ ), lt entity reference, 57
- length property, 204
- LessBasicDocument.xml file (code), 234
- LessBasicDocument.xsd file (code), 235
- Level 2 (DOM)
  - Core interfaces, 199-201
  - specifications, 199
- link element, accessing CSS (Cascading Style Sheets), 167
- linking
  - elements, 174
  - text in Mozilla browser, 176-177
- links. *See* hyperlinks
- lists
  - HTML (Hypertext Markup Language), creating, 122-125
  - mailing, 244-245
- literal result elements, 120
- local parts of Qnames, 93
- local resources (XLink), 174
- localName property, 203
- location paths (XPath), 107
- locationOfInformation variable, unparsed entities, 62
- logic in XML, 77
- loosely structured data in XML, 83-85
- lt entity reference ( [ ), 57

## M

- mailing lists, 244-245
- mapping, geographical examples (Web site), 192
- maps, Character Map, 67-69

- markup declarations, 19-20, 38, 43-44
- markup languages, 9-10. *See also* HTML; XML
  - data, modeling as XML, 77-78
  - WML (Wireless Markup Language), 116
- messages (code), 118-121
- meta element, 121
- meta languages, 10-12
- methods
  - appendChild(newChild), 203
  - appendData(arg), 209
  - CharacterData interface (DOM), 209
  - cloneNode(deep), 203
  - ContentHandler interface, 223
  - createAttribute(name), 206
  - createAttributeNS(namespaceURI, qualifiedName), 206
  - createCDATASection(data), 206
  - createComment(data), 206
  - createDocument(namespaceURI, qualifiedName, doctype), 205
  - createDocumentFragment(), 206
  - createDocumentType(qualifiedName, publicId, systemId), 205
  - createElement(), 210-212
  - createElement(tagName), 206
  - createElementNS(), 213-214
  - createElementNS(namespaceURI, qualifiedName), 206
  - createEntityReference(name), 206
  - createProcessingInstruction(target, data), 206
  - createTextNode(data), 206
  - deleteData(offset, count), 209
  - Document interface (DOM), 206
  - DOM (Document Object Model), 199
  - DOMImplementation interface, 205
  - DTDHandler interface, 223
  - Element interface (DOM), 207-208
  - ErrorHandler interface, 223
  - getAttribute(name), 207
  - getAttributeNode(name), 207
  - getAttributeNodeNS(namespaceURI, localName), 207

getAttributeNS(namespaceURI, localName), 207  
 getElementById( ), 217  
 getElementsByTagName(tag-name), 206-207  
 getElementsByTagNameNS(namespaceURI, localName), 206-207  
 getNamedItem(name), 204  
 getNamedItemNS(namespaceURI, localName), 204  
 hasAttribute(name), 207  
 hasAttributeNS(namespaceURI, localName), 208  
 hasAttributes( ), 203  
 hasChildNodes( ), 203  
 hasFeature(feature, version), 205  
 importNode(importedNode, deep), 206  
 insertBefore(newChild, refChild), 203  
 insertData(offset, arg), 209  
 isSupported(feature, value), 203  
 item(index), 204  
 NamedNodeMap interface (DOM), 204  
 Node interface (DOM), 203-204  
 normalize( ), 204  
 removeAttribute(name), 208  
 removeAttributeNode(oldAttr), 208  
 removeAttributeNS(namespaceURI, localName), 208  
 removeChild(oldChild), 204  
 removeNamedItem(name), 204  
 removeNamedItemNS(namespaceURI, localName), 204  
 replaceChild(newChild, oldChild), 204  
 replaceData( ), 217  
 replaceData(offset, count, arg), 209  
 setAttribute(name, value), 208  
 setAttributeNode(newAttr), 208  
 setAttributeNodeNS(newAttr), 208  
 setAttributeNS(namespaceURI, localName), 208  
 setNamedItem(arg), 204  
 setNamedItemNS(namespaceURI, localName), 204

splitText(offset), 209  
 substringData(offset, count), 209  
 Text interface (DOM), 209  
 Mixed content content model (DTDs), 47  
 modeling  
   data as XML, 77-85  
   employee records in XML (code), 81-82  
   relational-type data, 81-83  
 models. *See also* DOM  
   DTDs (Document Type Definitions)  
     *Any content content*, 47  
     *Child elements content*, 47  
     *Empty element content*, 46  
     *Mixed content content*, 47  
     *Text only content*, 46  
   data, XML Information Set (infoset), 87  
   XML data (W3C), 85-86  
 Mouseover.svg file (code), 193-194  
 mouseovers, SVG (Scalable Vector Graphics) filters, 196-197  
 Mozilla 1.0 browser, 164-166, 176-177, 251  
 Mozilla 1.x, XLink (XML Linking Language), 175  
 MSXML (XSLT), 248-249  
 myHandler class, output, 228-230  
 myHandler.java file (code), 226-228  
 MySite.css file (code), 167-168  
 myXML.xml file, displaying characters (code), 75

---

## N

name( ) function, 120  
 NamedNodeMap interface (DOM), methods or properties, 204  
 names  
   : (colon), 31-32, 93-94  
   - (hyphen), 32  
   . (period), 32  
   \_ (underscore), 31  
   attribute nodes, 104  
   of attributes, 33  
   bare (XPointer), 196

- case sensitivity, 31
- element, 162
- element nodes, 104
- element type, 6, 32, 91-93
- of entities, 54, 59
- elements in XML documents, 8
- entityName, unparsed entities, 62
- initial characters, 31
- non-initial characters, 32
- notationName, unparsed entities, 62
- property, 208
- Qnames (qualified names), 93-94
- qualified, 94
- well-formed documents (XML), 30-32
- namespaces
  - in applications, 91
  - attributes, 98-99
  - axis, 105
  - declarations, 96-98
  - default, 97
  - definition, 89
  - in documents, 90, 99-101
  - element type names, clashing, 91-93
  - local parts, 93
  - nodes, 106
  - packages, 93
  - prefixes, 93
  - Qnames (qualified names), 93
  - qualified names, 94
  - URIs (uniform resource identifiers), 94-95
  - well-formed, 99
  - XLink (XML Linking Language), 175
  - XML or xml character sequence, 96
  - XSLT URI, 116
- NDATA keyword, 62
- nested elements, 24-25, 234-235
- Netscape 6.x, XLink (XML Linking Language), 175
- news
  - XMMLNews.html file (code), 127-128
  - XMMLNews.xml file (code), 126
  - XMMLNews.xml file (code), 126-127
- NMTOKEN value, 50
- NMTOKENS value, 50
- Node interface (DOM), 201-204
- node-points, 182
- NodeList interface (DOM), 204
- nodeName property, 203
- nodes
  - attribute, 104-106
  - child, 217-220
  - comment, 106
  - container, 182
  - context, XPath (XML Path Language), 104
  - element, 104-106
  - element node child, 103
  - namespace, 106
  - principal types (XPath axes), 106
  - processing instruction, 106
  - root, 86, 103-106
  - text, 106
  - XPath (XML Path Language) 1.0 types, 106
- nodeType property, 203
- nodeValue property, 203
- non-initial characters of XML names, 32
- nonvalidating XML processors, 42
- normalize( ) method, 204
- notationName, unparsed entities, 62
- notations
  - decimal, characters, 68, 71
  - declaring, 62
  - hexadecimal, characters, 68, 71
  - property, 207
  - unparsed entities, 62

---

## O

---

- objects. *See also* DOM
  - data, hierarchical structure, 85
  - DocumentType, displaying properties (code), 215-216
- online resources, mailing lists, 244-245. *See also* Web sites
- opening Web browser windows (code), 177
- opening parentheses [ ( ], pointer parts, 181

operators, + cardinality, 181  
 origin( ) function, 184  
 outbound arc (XPath), 174  
 output, myHandler class, 228-230

## P

packages, namespaces, 93  
 pages (Web), 118-122, 191  
 parameters, entities, 37-38, 56-57, 62-64  
 parent axis, 105, 182  
 parsed character data (PCDATA), 45  
 parsed data, 58  
 parsed entities  
   & (ampersand), 57  
   ; (semicolon), 57  
   external, 54-61  
   general entities, 58  
   internal, 56, 59-60  
   unparsed, 57, 61-62  
   in well-formed documents (XML), 37  
 parsers  
   applications, 21  
   documents (XML), 21  
   MSXML, downloading, 249  
   SAX (Simple API for XML), 224-226  
   Xerces, 224, 250  
 parsing  
   entities in well-formed documents (XML), 36-37  
   events (SAX), 222-223  
 Path environment variable, 225-226  
 paths, location (XPath), 107  
 PCDATA (parsed character data), 45  
 period (.), 32, 162  
 pointer languages. *See* XPointer  
 pointers, parts, 181  
 points, 73-74, 182-183  
 position( ) function, 113  
 preceding axis, 105  
 preceding-sibling axis, 106  
 predefined entities, 39, 57  
 prefix property, 203  
 prefixes, namespace, Qnames (qualified names), 93

presentation and content, separating, 13, 160-161  
 principal node types, XPath (XML Path Language) axes, 106  
 processing  
   data, conditional, 143-144  
   instructions, 20-21, 106, 161  
 processors  
   UTF-8 character encoding, 19  
   UTF-16 character encoding, 19  
   XML (Extensible Markup Language), 14, 39-42, 70, 161  
 programming  
   languages (Unicode), 73  
   SAX (Simple API for XML), 222-223  
 prolog, documents (XML), 17-21  
 properties  
   Attr interface (DOM), 208  
   attributes, 203  
   CharacterData interface (DOM), 208  
   code, 205  
   data, 208  
   doctype, 206  
   Document interface (DOM), 205  
   documentElement, 206  
   DocumentType, 207, 215-217  
   DOM (Document Object Model), 199  
   DOMException interface, 205  
   Element interface (DOM), 207  
   firstChild, 203  
   lastChild, 203  
   length, 204  
   localName, 203  
   name, 208  
   NamedNodeMap interface (DOM), 204  
   Node interface (DOM), 203  
   nodeName, 203  
   nodeType, 203  
   nodeValue, 203  
   notations, 207  
   prefix, 203  
   systemID, 207  
   tagname, 207  
   value, 208



public identifiers, 23  
 PUBLIC keyword, 23  
 PurchaseOrder.xml file (code),  
 131-132  
 PurchaseToOrder.xml file (code), 132  
 PurchaseToOrder2.xml file (code), 134

## Q-R

Qnames (qualified names), 93-94

quotation marks

double (" "), 25, 38  
 single double ("), quot entity reference, 57

range() function, 183

range-inside() function, 184

range-to() function, 184

ranges, 183-184

RDBMS (relational database-management system), 81

readability, human readable (XML),  
 7-8

records, employee, modeling in XML  
 (code), 81-82

rect elements, creating, 214

rectangles, 194, 213-214

Reference.svg file (code), 196-197

references

; (semicolon), 38-39, 71  
 amp entity (&), 57  
 apos entity ('), 57  
 character, 39, 68-72  
 definitions with XPointer (XML  
 Pointer Language), 196-198  
 entity, 56-57  
 external entities, none for attributes, 34-35  
 foreign-language characters, displaying, 71  
 gt entity (>), 57  
 lt entity (<), 57  
 parameter entities in DTDs  
 (Document Type Definitions),  
 37  
 quot entity ("), 57

relational database-management system  
 (RDBMS), 81

relational-type data, modeling, 81-83

remote resources (XLink), 174

removeAttribute(name) method, 208

removeAttributeNode(oldAttr) method,  
 208

removeAttributeNS(namespaceURI,  
 localName) method, 208

removeChild(oldChild) method, 204

removeNamedItem(name) method, 204

removeNamedItemNS(namespaceURI,  
 localName) method, 204

replaceChild(newChild, oldChild)  
 method, 204

replaceData() method, 217

replaceData(offset, count, arg) method,  
 209

replacement text, 38

reports, 122-125, 164-166

Reports.css file (code), 164

Reports.xml file (code), 164

Reports2.css file (code), 165-166

#REQUIRED value, 51

resources. *See also* Web sites

mailing lists, 244-245

XLink (XML Linking Language)  
 (ending, local, remote, starting),  
 174

retrieving information, DOM

(Document Object Model), 215-220

right angle bracket (>), gt entity reference, 57

rollovers, 193-194

root elements, 24

root nodes, 86, 103-106

rules

Boolean values, conversions to,  
 145

CSS (Cascading Style Sheets),  
 161-163

definition, 162

well-formed documents (XML),  
 30

XML syntax, 10-11

## S

- SAX (Simple API for XML)
  - ContentHandler interface, 223
  - documents, 222
  - and DOM (Document Object Model), comparing, 221
  - DTDHandler interface, 223
  - ErrorHandler interface, 223
  - examples, 226
  - history, 221
  - myHandler class, output, 228-230
  - myHandler.java file (code), 226-228
  - parsers, 224-226
  - programmer mindset, 222
  - programming, 222-223
  - SAXSource.xml file (code), 223
- Saxon (XSLT), 247-248
- SAXSource.xml file (code), 223
- Scalable Vector Graphics. *See* SVG
- schemas
  - definition, 42
  - languages, 231
  - W3C XML Schema, 231-240, 245
- schemes
  - XPointer (XML Pointer Language), 181-186
  - xpointer(), 182-184
- scope, namespaces in documents, 100-101
- screens
  - hexadecimal characters, visibility, 66
  - SVG (Scalable Vector Graphics)
    - content, viewing, 192
    - text animations, 192
- scripting DOM (Document Object Model) with SVG (Scalable Vector Graphics), 189
- sections
  - CDATA sections, 25-27
  - conditional, 46
- selectors, CSS (Cascading Style Sheets) rules or curly brackets ({ }), 162
- self axis, 105, 182
- semicolon (;), 57, 71, 163
- sensitivity, case
  - element type names, 6
  - Java, 226
  - names, 31
  - XML, 31
  - XPath (XML Path Language) axis names, 105-106
- sequences of characters. *See* character sequences
- serif fonts, 75
- servers, Tomcat Web site, 251
- setAttribute(name, value) method, 208
- setAttributeNode(newAttr) method, 208
- setAttributeNodeNS(newAttr) method, 208
- setAttributeNS(namespaceURI, localName) method, 208
- setNamedItem(arg) method, 204
- setNamedItemNS(namespaceURI, localName) method, 204
- SGML (Standard Generalized Markup Language), limitations of DTDs (Document Type Definitions), 232-233
- shadows, drop (SVG filters), 198
- shallow copy (elements), 131-133
- shapes, SVG (Scalable Vector Graphics), 189
- shared documents, 41-43
- shirts (code), 136-141
- shorthand forms (XPointer), 196
- Simple API for XML. *See* SAX
- simple links (XLink), 172-176
- simple types of W3C XML Schema, 233, 238
- single document entity, book descriptions, 54
- single double quotation mark ("), quot entity reference, 57
- SingleEntity.xml file, book descriptions (code), 54
- sizes of bytes, 71
- slashes
  - / (forward), 86, 104, 107
  - // (double forward), 125
- Sniffer utility (MSXML) Web site, 249
- "some ValueInQuotes" value, 51
- sorting data, 143-158

- specifications
  - DOM (Document Object Model)
    - Level 2, 199
  - SVG (Scalable Vector Graphics), 8
  - W3C (World Wide Web Consortium), 8, 178, 231
- SplitEntities.xml file, book descriptions (code), 54
- splitText(offset) method, 209
- Spy (XML), 247
- square brackets, delimiters for DTD internal subsets
  - [ (left) opening, 45
  - ] (right) closing, 35
- standalone attribute, 19
- Standard Generalized Markup Language (SGML), limitations of DTDs (Document Type Definitions), 232-233
- start points, 183
- start tags, 14, 32-33
- start-point( ) function, 184
- starting delimiters
  - <!--, 20
  - <![CDATA[, 26
  - <?, 21
  - [ (left square bracket), DTD internal subsets, 45
- starting resources (XLink), 174
- statements in English and German (code), 71
- stories
  - XMMLNews.html file (code), 127-128
  - XMMLNews.xml file (code), 126-127
- string-range( ) function, 183
- strings, character, -- (double hyphens), 21
- structures
  - data, 78-81
  - of documents, defining, 42-43
  - hierarchical, data objects, 85
  - namespace declarations, 96
  - XSLT (Extensible Stylesheet Language Transformations) stylesheet, 117-118
  - style sheets, SVG (Scalable Vector Graphics), 189. *See also* CSS stylesheets. *See also* XSLT
    - Documents.html file (code), 147-148
    - Documents.xml file (code), 146-147
    - Documents2.html file (code), 151-152
    - Documents2.xml file (code), 149-151
    - Documents3.html file (code), 154-155
    - Documents3.xml file (code), 152-154
    - Documents4.html file (code), 157-158
    - Documents4.xml file (code), 155-157
    - HTMLTemplate.xml file (code), 122
    - stylesheet element, 116
    - xml-stylesheet processing instruction, 21
    - XMMLNews.html file (code), 127-128
    - XMMLNews.xml file (code), 126
    - XMMLNews.xml file (code), 126-127
    - XMMLReports.html file (code), 125
    - XMMLReports.xml file (code), 122-123
    - XMMLReports.xml file (code), 124
    - XSLTMessage.html file (code), 121
    - XSLTMessage.xml file (code), 119
  - xsl:apply-templates element, 120, 124, 139, 141
  - xsl:attribute element, 140-141
  - xsl:attribute-set element, 118
  - xsl:copy element, 131-133, 141
  - xsl:copy-of element, 134
  - xsl:decimal-format element, 118
  - xsl:element element, 137-139
  - xsl:import element, 117
  - xsl:include element, 117

- xsl:key element, 118
- xsl:namespace-alias element, 118
- xsl:output element, 117
- xsl:param element, 118
- xsl:preserve-space element, 118
- xsl:strip-space element, 118
- xsl:stylesheet element, 117
- xsl:template element, 117-119, 125
- xsl:text element, 125
- xsl:transform element, 117
- xsl:value-of element, 120-121, 133, 139
- xsl:variable element, 118
- styling data (CSS), 164-167
- styling limitations, CSS (Cascading Style Sheets) rules, 163
- subsets, DTDs (Document Type Definitions), external or internal, 22, 37, 44-45
- substringData(offset, count) method, 209
- supplementary code points (Unicode), 73-74
- SVG (Scalable Vector Graphics)
  - Adobe Illustrator Web site, 193
  - Adobe SVG Viewer, 75, 190-192
  - advantages, 190
  - animations, 189-191
  - Batik viewer, downloading, 190-192
  - bitmaps, 189-190
  - ChildNodes.svg file (code), 217-219
  - code, 188
  - content, viewing, 192
  - CorelDraw Web site, 193
  - CreateCircle.svg file (code), 210-211
  - CreateRectNS.svg file (code), 213-214
  - creating, 191-193
  - declarations, 191
  - DoctypeProps.svg file (code), 215-216
  - elements, 188, 191
  - entities, 53, 60
  - filters, 196-198
  - geographical mapping examples (Web site), 192
  - HelloVector.svg file (code), 191
  - Jasc WebDraw Web site, 192
  - Mouseover.svg file (code), 193-194
  - rectangles, rolled-over and unrolled-over versions, 194
  - Reference.svg file (code), 196-197
  - rollovers, 193-194
  - scripting, DOM (Document Object Model), 189
  - shapes, attributes, 189
  - specification, 8
  - Specification Web site, 242
  - style sheets, 189
  - SVGLink.svg file (code), 194-195
  - text animations, 192
  - vector images, displaying, 189
  - W3C Candidate Recommendation Web site, 188
  - Web pages, XMMML Web site, 191
  - X-Smiles browser, Web site, 190
  - XLink (XML Linking Language), 178, 194-196
  - XML code, 189
  - XPointer (XML Pointer Language), 196-198
- <svg>, child nodes, 219
- SVG-Developers mailing list Web site, 244
- SVGLink.svg file (code), 194-195
- SVGSpider.com Web site, 243
- syntax. *See* code
- system identifier, 23
- systemID property, 207

---

## T

- tables, HTML (Hypertext Markup Language), creating, 126-128
- tags
  - document element end, content after, 28
  - HTML (Hypertext Markup Language), 9
  - names (element type names), 6, 91-93, 207
  - start and end, balancing, 32-33
  - XML, end and start, 14

## text

- animating, 192
- appearance, controlling (code), 176
- CDATA sections, 27-28
- declarations, external parsed entities, 61
- linking in Mozilla browser, 176-177
- node, 106
- replacement, 38
- Text interface (DOM), 203, 209
- Text only content model (DTDs), 46
- third-party arc (XLink), 174
- title element, 120
- Title.xml file, book descriptions (code), 55
- Tomcat server Web site, 251
- tools
  - XLink (XML Linking Language), 251
  - XML editors, 246
  - XPointer (XML Pointer Language), 251
  - XSLT (Extensible Stylesheet Language Transformations), 116, 247
- ToolTip, ! (exclamation mark), 67
- top-level elements (XSLT), 117
- transform element, 116
- transformations, 9
- transforming documents (XML), 115
- transversal (XLink), 174
- types
  - of attributes in DTDs (Document Type Definitions), 51
  - data, 62, 232
  - W3C XML Schema, 233
    - complex, defining, 239-240*
    - simple, defining, 238*

---

## U

- U+0021 (Unicode), 68
- UKShirts.xml file (code), 137
- UKShirtsToUS.xml file (code), 140-141

- unabbreviated XPath (XML Path Language) syntax, 107
- underscore (\_), names, 31
- Unicode, 68, 72-74
- uniform resource identifiers (URIs), 23, 94-95, 116
- unique attributes, 34
- unparsed data, 58, 62
- unparsed entities, 57, 61-62. *See also* parsed entities
- URIs (uniform resource identifiers), 23, 94-95, 116
- USShirts.xml file (code), 136-137
- USShirtToUK.xml file (code), 137-139
- UTF-8 bit or UTF-16 bit, 19, 70, 74
- UTF-32 encoding form (Unicode), 74
- utilities
  - Character Map, 67-69
  - MSXML Sniffer Web site, 249

---

## V

- valid XML documents, 43
- validating XML processors, 42
- values
  - of attributes, 33-35
  - attributeType, 50
  - Boolean, conversion rules, 145
  - CDATA, 50
  - default attribute, specifying, 52
  - defaultDeclaration, 50-51
  - ENTITIES, 50
  - ENTITY, 50
  - #FIXED "some ValueInQuotes," 51
  - ID, 50
  - IDREF, 50
  - IDREFS, 50
  - #IMPLIED, 51
  - NMTOKEN, 50
  - NMTOKENS, 50
  - property, 208
  - #REQUIRED, 51
  - "some ValueInQuotes," 51
  - of version attribute, 148-155

## variables

- CLASSPATH environment, setting, 225-226
- locationOfInformation, unparsed entities, 62
- Path environment, setting, 225-226

vector graphics. *See* SVG

## vector images, 163, 189

## version attribute, 18, 117, 148-155

## viewers

- Adobe SVG Viewer, 75, 190-192
- Batik, downloading, 190-192

## viewing SVG (Scalable Vector Graphics) content, 192

## vocabularies

- creating, 11-12
- jargon, 173-174, 233

## W

---

## W3C (World Wide Web Consortium)

- Candidate Recommendation, Web site, 188

## DOM (Document Object Model)

- Level 2 Core or Events Specification Web site, 242

## mailing list, 244

## specifications, XPointer (XML Pointer Language), 178

## SVG (Scalable Vector Graphics)

- 1.0 Specification Web site, 242

## Web site, 241-243

## XForms specification, 8

## XLink (XML Linking Language)

- Version 1.0 Web site, 172, 242

## XML, 85-87

## XML Schema, 231-240, 245

## XPath (XML Path Language), 86-87

## XSLT (Extensible Stylesheet Language Transformations)

- Version 1.0 Web site, 242

## Web, WWW (World Wide Web), 8-9, 65

## Web browsers

- Internet Explorer, XLink (XML Linking Language), 175
- Mozilla 1.0, 164-166, 251

## Mozilla 1.x, XLink (XML Linking Language), 175

## Netscape 6.x, XLink (XML Linking Language), 175

## windows, opening (code), 177

## X-Smiles, Web site, 8, 190

## Web pages, 118-122, 191

## Web sites

## Adobe Illustrator, 193

## Adobe SVG Viewer, 75, 190-192

## Apache XML, 243

## Batik viewer, downloading, 190-192

## content and presentation, separating, 160-161

## CorelDraw, 193

## DOM (Document Object Model)

- Level 2 Core or Events Specifications, 242

## Fujitsu, 251

## geographical mapping examples (SVG), 192

## Google, 243

## Instant Saxon, 248

## Jasc WebDraw, 192

## JSDK (Java Software

- Development Kit), downloading, 224

## JVM (Java Virtual Machine), downloading, 247

## Mozilla 1.0 browser, 251

## MSXML Sniffer utility, 249

## Saxon, 248

## SVG (Scalable Vector Graphics)

- 1.0 Specification, 242

## SVG-Developers mailing list, 244

## SVGSpider.com, 243

## Tomcat server, 251

## Unicode, 73

## W3C (World Wide Web

- Consortium), 172, 188, 241-245

## WinZip, 248

## X-Smiles browser, 8, 190

## Xerces parser, downloading, 224

## XLink (XML Linking Language), 196, 242

## XLiP, 251

## XML Spy, 247

## XML.com, 243

## XMLHack.com, 243

XMML, SVG (Scalable Vector Graphics) Web pages, 191  
 XSL mailing list, 244  
 XSLT (Extensible Stylesheet Language Transformations) Version 1.0, 242  
 XSLTalk mailing list, 244  
 well-formed documents (XML), 29-39  
 well-formed namespaces, 99  
 windows, opening Web browsers (code), 177  
 WinZip Web site, 248  
 WML (Wireless Markup Language), 116  
 World Wide Web Consortium. *See* W3C  
 Writer (XML), 246  
 writers, XML Writer, 246  
 WWW (World Wide Web), 8-9, 65

## X-Y-Z

X-Smiles browser, Web site, 8, 190  
 Xalan (XSLT), 249-251  
 XBRL (Extensible Business Reporting Language), 251  
 Xerces parser, 224, 250  
 XForms specification, 8  
 XLink (XML Linking Language), 8  
   attributes, 175  
   BigText.css file (code), 176  
   CSS (Cascading Style Sheets), controlling text appearance (code), 176  
   Document3.xml file (code), 177  
   elements, 174-175  
   ending resources, 174  
   extended links, 172-173  
   HTML (Hypertext Markup Language) hyperlinks, comparing, 173  
   hyperlinks, 174, 196  
   inbound arc, 174  
   Internet Explorer, 175  
   jargon, 173-174  
   linking elements, 174  
   links, opening browser windows (code), 177

local resources, 174  
 Mozilla 1.x, 175  
 namespace, 175  
 Netscape 6.x, 175  
 outbound arc, 174  
 remote resources, 174  
 simple links, 172-176  
 starting resources, 174  
 SVG (Scalable Vector Graphics), 178, 194-196  
 SVGLink.svg file (code), 194-195  
 third-party arc, 174  
 tools, 251  
 W3C Recommendation Web site, 172  
 Web browsers, 175  
   xlink:actuate attribute, 175  
   xlink:href attribute, 175  
   xlink:show attribute, 175  
   xlink:type attribute, 175  
   Version 1.0 Web site, 242  
 xlink:actuate attribute, 175  
 xlink:href attribute, 175  
 xlink:show attribute, 175  
 xlink:type attribute, 175  
 XLiP Web site, 251  
 XML (Extensible Markup Language), 5, 9-10  
   case sensitivity, 31  
   code, SVG (Scalable Vector Graphics), 189  
   content and presentation, separating, 13  
   data models (W3C), 85-86  
   documents. *See* documents (XML)  
   editors, 246-247  
   elements, adding attributes, 15  
   employee data (code), 80-82  
   end tags, 14  
   files, 84  
   hierarchical data, 83  
   and HTML (Hypertext Markup Language), comparing, 12-13, 78-81  
   human readable, 7-8  
   Information Set (infoset), data models, 87  
   internationalization, 69-72  
   loosely structured data, 83-85

- meta language, 10-12
- processors, 14, 39-42, 70
- Spy, 233, 247
- start tags, 14
- syntax rules, 10-11
- valid documents, 43
- vocabularies, creating, 11-12
- writing, 14-16
- WWW (World Wide Web), 8-9
- or xml character sequences, 59, 96
- XML Linking Language. *See* XLink
- XML Path Language. *See* XPath
- XML Pointer Language. *See* XPointer
- XML-DEV mailing list, 244
- xml-style sheet processing instruction, 21, 161
- XML.com Web site, 243
- XMLHack.com Web site, 243
- xmlns (namespace declarations), 96
- xmlns( ) scheme, 185-186
- xmlns="namespaceURI" namespace declaration, 97
- XMML Web site (SVG Web pages), 191
- XMMLNews.html file (code), 127-128
- XMMLNews.xml file (code), 126
- XMMLNews.xsl file (code), 126-127
- XMMLOrder.xml file (code), 133
- XMMLOrder2.xml file (code), 134-135
- XMMLReports.html file (code), 125
- XMMLReports.xml file (code), 122-123
- XMMLReports.xsl file (code), 124
- XPath (XML Path Language), 86-87, 102
  - ancestor axis, 105
  - ancestor-or-self axis, 105
  - attributes, 104-106, 111-112
  - axes, 105-108
  - child axis, 105
  - comment node, 106
  - count( ) function, 113-114
  - descendant axis, 105
  - descendant-or-self axis, 105
  - documents, node hierarchy, 103-104
  - elements, 109-110
  - expressions, location paths, 107
  - following axis, 105
  - following-sibling axis, 105
  - functions, 112
  - hierarchies, XML documents, 103
  - namespaces, 105-106
  - node types, 106
  - parent axis, 105
  - position( ) function, 113
  - preceding axis, 105
  - preceding-sibling axis, 106
  - processing instruction node, 106
  - root nodes, 86, 103-106
  - self axis, 105
  - syntax, 106-109
  - text node, 106
  - XPointer (XML Pointer Language), 180
- XPointer (XML Pointer Language), 8, 172
  - + cardinality operator, 181
  - Anchors.html file (code), 178-179
  - bare names, 196
  - character-points, 182
  - definitions, referencing, 196-198
  - document fragments, 178-180
  - end-point( ) function, 184
  - end points, 183
  - Framework, 181-186
  - here( ) function, 184
  - HTML (Hypertext Markup Language) document anchors, 178-179
  - mouseover, SVG (Scalable Vector Graphics) filters, 196-197
  - origin( ) function, 184
  - pointer parts, 181
  - points, axes, 182
  - range( ) function, 183
  - range-inside( ) function, 184
  - range-to( ) function, 184
  - ranges
    - collapsed*, 183
    - converted*, 184
  - Reference.svg file (code), 196-197
  - schemes, 181-186
  - shorthand forms, 196
  - start-point( ) function, 184
  - start points, 183



- string-range( ) function, 183
- SVG (Scalable Vector Graphics), 196-198
- tools, 251
- W3C (World Wide Web Consortium) specification, 178
- XPath (XML Path Language), 180
- xpointer( ) scheme, 182-184
- xsd:choice element, 239
- xsd:complexType element, 237-239
- xsd:element element, 239
- xsd:pattern element, 237
- xsd:restriction element, 237
- xsd:sequence element, 239
- xsd:simpleType element, 237
- xsd:string element, 236
- XSL mailing list Web site, 244
- XSL Transformations (XSLT) Version 1.0 Web site, 242
- XSLT (Extensible Stylesheet Language Transformations). *See also* documents (XML)
  - CSS (Cascading Style Sheets), HTML (Hypertext Markup Language) output, 167-170
  - CSSInformation.html file (code), 169-170
  - CSSInformation.xml file (code), 168-169
  - data, conditional processing or sorting, 143-144
  - HTML (Hypertext Markup Language), 118-128, 170-171
  - HTMLTemplate.xml file (code), 122
  - Instant Saxon, 247-248
  - MSXML, 248-249
  - namespace URI, 116
  - purpose, 115-116
  - Saxon, 247-248
  - and style sheets, comparing, 159
  - stylesheet, 116-118
  - tools, 116, 247
  - top-level elements, 117
  - Version 1.0 Web site, 242
  - Xalan, 249-251
  - xml-stylesheet processing instruction, 161
  - XMMLNews.html file (code), 127-128
  - XMMLNews.xml file (code), 126
  - XMMLNews.xml file (code), 126-127
  - XMMLReports.html file (code), 125
  - XMMLReports.xml file (code), 122-123
  - XMMLReports.xml file (code), 124
  - XSL Transformations, 13
  - XSLTMessage.html file (code), 121
  - XSLTMessage.xml file (code), 118
  - XSLTMessage.xml file (code), 119
  - xsl:choose element, 144, 149-152
  - xsl:for-each element, 155
  - xsl:if element, 144-148
  - xsl:otherwise element, 149
  - xsl:sort element, 152-158
  - xsl:template element, 144
  - xsl:when element, 149
  - XSLTalk mailing list Web site, 244
  - XSLTMessage.html file (code), 121
  - XSLTMessage.xml file (code), 118
  - XSLTMessage.xml file (code), 119
  - xsl:apply-templates element, 120, 124, 139-141
  - xsl:attribute element, 140-141
  - xsl:attribute-set element, 118
  - xsl:choose element, 143-144, 149-152
  - xsl:copy element, 131-133, 141
  - xsl:copy-of element, 134
  - xsl:decimal-format element, 118
  - xsl:element element, 137-139
  - xsl:for-each element, sorting data, 155
  - xsl:if element, 143-148
  - xsl:import element, 117
  - xsl:include element, 117
  - xsl:key element, 118
  - xsl:namespace-alias element, 118
  - xsl:otherwise element, sorting data, 149
  - xsl:output element, 117
  - xsl:param element, 118
  - xsl:preserve-space element, 118
  - xsl:sort element, 143, 152-158
  - xsl:strip-space element, 118

- xsl:stylesheet element, 117
- xsl:template element, 117-119, 125, 144
- xsl:text element, 125
- xsl:transform element, 117
- xsl:value-of element, 120-121, 133, 139
- xsl:variable element, 118
- xsl:when element, 149



**SAMS**  
**Teach**  
**Yourself**



**Covers**  
**PHP 5.0**

# PHP

Chris Newman

*in* **10**  
**Minutes**

***SAMS***  
**Teach  
Yourself**

**PHP**

*in* **10**  
**Minutes**

Chris Newman

***SAMS***

800 East 96th Street, Indianapolis, Indiana, 46240 USA

# Sams Teach Yourself PHP in 10 Minutes

**Copyright © 2005 by Sams Publishing**

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32762-7

Library of Congress Catalog Card Number: 2004098028

Printed in the United States of America

First Printing: April 2005

08 07 06 05 4 3 2 1

## Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**  
**1-800-382-3419**  
**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**  
**international@pearsoned.com**

**ACQUISITIONS EDITOR**  
Shelley Johnston

**DEVELOPMENT EDITOR**  
Damon Jordan

**MANAGING EDITOR**  
Charlotte Clapp

**SENIOR PROJECT EDITOR**  
Matthew Purcell

**COPY EDITOR**  
Kitty Jarrett

**INDEXER**  
Chris Barrick

**PROOFREADER**  
Paula Lowell

**TECHNICAL EDITOR**  
Sara Goleman

**PUBLISHING COORDINATOR**  
Vanessa Evans

**INTERIOR DESIGNER**  
Gary Adair

**COVER DESIGNER**  
Aren Howell

**PAGE LAYOUT**  
Susan Geiselman

# Contents

Introduction .....	1
--------------------	---

## **PART I      PHP Foundations**

---

<b>1    Getting to Know PHP</b>	<b>5</b>
PHP Basics .....	5
Your First Script .....	8
<b>2    Variables</b>	<b>13</b>
Understanding Variables .....	13
Data Types .....	17
<b>3    Flow Control</b>	<b>20</b>
Conditional Statements .....	20
Loops .....	26
<b>4    Functions</b>	<b>30</b>
Using Functions .....	30
Arguments and Return Values .....	32
Using Library Files .....	36

## **PART II      Working with Data**

---

<b>5    Working with Numbers</b>	<b>39</b>
Arithmetic .....	39
Numeric Data Types .....	42
Numeric Functions .....	44

<b>6</b>	<b>Working with Strings</b>	<b>47</b>
	Anatomy of a String .....	47
	Formatting Strings .....	50
	String Functions .....	54
<b>7</b>	<b>Working with Arrays</b>	<b>57</b>
	What Is an Array? .....	57
	Array Functions .....	61
	Multidimensional Arrays .....	65
<b>8</b>	<b>Regular Expressions</b>	<b>68</b>
	Introducing Regular Expressions .....	68
	Using ereg .....	69
<b>9</b>	<b>Working with Dates and Times</b>	<b>78</b>
	Date Formats .....	78
	Working with Timestamps .....	80
<b>10</b>	<b>Using Classes</b>	<b>86</b>
	Object-Oriented PHP .....	86
	What Is a Class? .....	87
	Creating and Using Objects .....	88

---

## **PART III    The Web Environment**

---

<b>11</b>	<b>Processing HTML Forms</b>	<b>93</b>
	Submitting a Form to PHP .....	93
	Processing a Form with PHP .....	98
	Creating a Form Mail Script .....	101



<b>12</b>	<b>Generating Dynamic HTML</b>	<b>103</b>
	Setting Default Values .....	103
	Creating Form Elements .....	107
<b>13</b>	<b>Form Validation</b>	<b>113</b>
	Enforcing Required Fields .....	113
	Displaying Validation Warnings .....	114
	Enforcing Data Rules .....	117
	Highlighting Fields That Require Attention .....	118
<b>14</b>	<b>Cookies and Sessions</b>	<b>121</b>
	Cookies .....	121
	Sessions .....	125
<b>15</b>	<b>User Authentication</b>	<b>128</b>
	Types of Authentication .....	128
	Building an Authentication System .....	131
<b>16</b>	<b>Communicating with the Web Server</b>	<b>137</b>
	HTTP Headers .....	137
	Server Environment Variables .....	142

---

## **PART IV    Using Other Services from PHP**

---

<b>17</b>	<b>Filesystem Access</b>	<b>146</b>
	Managing Files .....	146
	Reading and Writing Files .....	150
<b>18</b>	<b>Host Program Execution</b>	<b>156</b>
	Executing Host Programs .....	156
	The Host Environment .....	159
	Security Considerations .....	162

<b>19</b>	<b>Using a MySQL Database</b>	<b>164</b>
	Using MySQL .....	164
	Executing SQL Statements .....	166
	Debugging SQL .....	170
<b>20</b>	<b>Database Abstraction</b>	<b>174</b>
	The PEAR DB Class .....	174
	Database Portability Issues .....	181
<b>21</b>	<b>Running PHP on the Command Line</b>	<b>185</b>
	The Command-Line Environment .....	185
	Writing Scripts for the Command Line .....	189
<b>22</b>	<b>Error Handling</b>	<b>194</b>
	Error Reporting .....	194

---

## **PART V     Configuring and Extending PHP**

---

<b>23</b>	<b>PHP Configuration</b>	<b>203</b>
	Configuration Settings .....	203
	Configuration Directives .....	206
	Loadable Modules .....	211
<b>24</b>	<b>PHP Security</b>	<b>214</b>
	Safe Mode .....	214
	Other Security Features .....	218
<b>25</b>	<b>Using PEAR</b>	<b>223</b>
	Introducing PEAR .....	223
	Using PEAR .....	225

---

## PART VI    Appendix

---

<b>A    Installing PHP</b>	<b>230</b>
Linux/Unix Installation .....	230
Windows Installation .....	234
Troubleshooting .....	236

# About the Author

**Chris Newman** is a consultant programmer specializing in the development of custom web-based database applications to a loyal international client base.

A graduate of Keele University, Chris lives in Stoke-on-Trent, England, where he runs Lightwood Consultancy Ltd., the company he founded in 1999 to further his interest in Internet technology. Lightwood operates web hosting services under the DataSnake brand and is proud to be one of the first hosting companies to offer and support SQLite in addition to PHP as a standard feature on all accounts.

More information on Lightwood Consultancy Ltd. can be found at [www.lightwood.net](http://www.lightwood.net), and Chris can be contacted at [chris@lightwood.net](mailto:chris@lightwood.net).

## We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: [opensource@sampublishing.com](mailto:opensource@sampublishing.com)

Mail: Mark Taber  
Associate Publisher  
Sams Publishing  
800 East 96th Street  
Indianapolis, IN 46240 USA

## Reader Services

For more information about this book or another Sams Publishing title, visit our Web site, at [www.sampublishing.com](http://www.sampublishing.com). Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.



# Introduction: Welcome to PHP

This book is about PHP, one of the most popular web scripting languages around. It is a book for busy people. Each lesson takes just 10 minutes to work through, so if you have wanted to learn PHP for a while but have never really had the chance, don't put it off any longer!

## Who This Book Is For

This book is aimed at those who want to learn PHP, even if they don't have any previous programming or scripting experience. You can even use this book to learn PHP as a first programming language if you do not have any previous experience.

If you have some previous programming experience but have not written for the web before, you can use this book to learn about the PHP language and how to apply programming techniques to the web environment.

This book does not teach you HTML. Although knowledge of HTML is not a prerequisite, having published web pages in the past will be an advantage—even if you do not usually hand-code HTML.

## How This Book Is Organized

This book is organized into five parts.

### Part I: PHP Foundations

The lessons in Part I introduce the basic building blocks of the PHP language:

- **Lesson 1: Getting to Know PHP.** This chapter introduces you to what PHP is all about and gives some simple examples to show how PHP is used inside a web page.
- **Lesson 2: Variables.** This chapter explains how you assign values to variables and demonstrates some simple expressions.
- **Lesson 3: Flow Control.** This chapter examines the conditional and looping constructs that allow you to control the flow of a PHP script.
- **Lesson 4: Functions.** This chapter explains how you can modularize and reuse a frequently used section of code as a function.

## Part II: Working with Data

The lessons in Part II examine in more detail the different types of data that can be manipulated by PHP:

- **Lesson 5: Working with Numbers.** This chapter gives more detailed examples of the numeric manipulation you can perform in PHP.
- **Lesson 6: Working with Strings.** This chapter examines the powerful set of string functions that PHP provides.
- **Lesson 7: Working with Arrays.** This chapter explains how arrays work and examines the PHP functions that can manipulate this powerful data type.
- **Lesson 8: Regular Expressions.** This chapter shows how to perform complex string manipulation by using powerful regular expressions.
- **Lesson 9: Working with Dates and Times.** This chapter examines how to use date and time values in a PHP script.
- **Lesson 10: Using Classes.** This chapter introduces you to object-oriented PHP and examines how you define and access a class in a script.

## Part III: The Web Environment

The lessons in Part III deal with using PHP specifically in the web environment:

- **Lesson 11: Processing HTML Forms.** This chapter shows how you use PHP to process user-submitted input from an HTML form.
- **Lesson 12: Generating Dynamic HTML.** This chapter examines some techniques for creating HTML components on-the-fly from PHP.
- **Lesson 13: Form Validation.** This chapter examines some techniques for validating user-submitted input from an HTML form.
- **Lesson 14: Cookies and Sessions.** This chapter shows how to pass data between pages by using PHP sessions and how to send cookies to a user's browser.



- **Lesson 15: User Authentication.** This chapter examines some techniques for validating user-submitted input from an HTML form.
- **Lesson 16: Communicating with the Web Server.** This chapter looks at ways in which PHP can interact with a web server.

## Part IV: Using Other Services from PHP

Part IV looks at how PHP can communicate with external programs and services:

- **Lesson 17: Filesystem Access.** This chapter examines the PHP functions that enable you to access the filesystem.
- **Lesson 18: Host Program Execution.** This chapter examines the PHP functions that enable you to execute programs on the host system.
- **Lesson 19: Using a MySQL Database.** This chapter shows how to use a MySQL database for data storage and retrieval from PHP.
- **Lesson 20: Database Abstraction.** This chapter explains how you can access a database through an abstraction layer to make scripts more portable.
- **Lesson 21: Running PHP on the Command Line.** This chapter shows how you can use PHP as a powerful shell scripting language.
- **Lesson 22: Error Handling and Debugging.** This chapter discusses some techniques for finding and fixing bugs in scripts.

## Part V: Configuring and Extending PHP

The final part of the book deals with PHP administration:

- **Lesson 23: PHP Configuration.** This chapter explains some of the popular configuration options that can be set at runtime to change the behavior of PHP.
- **Lesson 24: PHP Security.** This chapter discusses security issues in PHP scripts and shows how you can use Safe Mode on a shared web server.

- **Lesson 25: Using PEAR.** This chapter introduces the freely available classes that are available in the PHP Extension and Application Repository.

## Versions of Software Covered

At the time of writing, the current version of PHP is PHP 5.0.3. Unless otherwise stated, all code examples in this book will work with PHP 4.1.0 and higher.

## Conventions Used in This Book

This book uses different typefaces to differentiate between code and regular English, and also to help you identify important concepts.

Text that you type and text that should appear on your screen is presented in monospace type.

It will look like this to mimic the way text looks on your screen.

Placeholders for variables and expressions appear in *monospace italic* font. You should replace the placeholder with the specific value it represents.



A Note presents interesting pieces of information related to the surrounding discussion.



A Tip offers advice or teaches an easier way to do something.



A Caution advises you about potential problems and helps you steer clear of disaster.

# LESSON 1

## Getting to Know PHP



*In this lesson you will find out what PHP is all about and see what it is able to do.*

### PHP Basics

There is a good chance you already know a bit about what PHP can do—that is probably why you have picked up this book. PHP is hugely popular, and rightly so. Even if you haven't come across an existing user singing its praises, you've almost certainly used a website that runs on PHP. This lesson clarifies what PHP does, how it works, and what it is capable of.

PHP is a programming language that was designed for creating dynamic websites. It slots into your web server and processes instructions contained in a web page before that page is sent through to your web browser. Certain elements of the page can therefore be generated on-the-fly so that the page changes each time it is loaded. For instance, you can use PHP to show the current date and time at the top of each page in your site, as you'll see later in this lesson.

The name PHP is a recursive acronym that stands for *PHP: Hypertext Preprocessor*. It began life called PHP/FI, the "FI" part standing for *Forms Interpreter*. Though the name was shortened a while back, one of PHP's most powerful features is how easy it becomes to process data submitted in HTML forms. PHP can also talk to various database systems, giving you the ability to generate a web page based on a SQL query.

For example, you could enter a search keyword into a form field on a web page, query a database with this value, and produce a page of matching

results. You will have seen this kind of application many times before, at virtually any online store as well as many websites that do not sell anything, such as search engines.

The PHP language is flexible and fairly forgiving, making it easy to learn even if you have not done any programming in the past. If you already know another language, you will almost certainly find similarities here. PHP looks like a cross between C, Perl, and Java, and if you are familiar with any of these, you will find that you can adapt your existing programming style to PHP with little effort.

## Server-Side Scripting

The most important concept to learn when starting out with PHP is where exactly it fits into the grand scheme of things in a web environment. When you understand this, you will understand what PHP can and cannot do.

The PHP module attaches to your web server, telling it that files with a particular extension should be examined for PHP code. Any PHP code found in the page is executed—with any PHP code replaced by the output it produces—before the web page is sent to the browser.



**File Extensions** The usual web server configuration is that `somefile.php` will be interpreted by PHP, whereas `somefile.html` will be passed straight through to the web browser, without PHP getting involved.

The only time the PHP interpreter is called upon to do something is when a web page is loaded. This could be when you click a link, submit a form, or just type in the URL of a web page. When the web browser has finished downloading the page, PHP plays no further part until your browser requests another page.

Because it is only possible to check the values entered in an HTML form when the submit button is clicked, PHP cannot be used to perform

client-side validation—in other words, to check that the value entered in one field meets certain criteria before allowing you to proceed to the next field. Client-side validation can be done using JavaScript, a language that runs inside the web browser itself, and JavaScript and PHP can be used together if that is the effect you require.

The beauty of PHP is that it does not rely on the web browser at all; your script will run the same way whatever browser you use. When writing server-side code, you do not need to worry about JavaScript being enabled or about compatibility with older browsers beyond the ability to display HTML that your script generates or is embedded in.

## PHP Tags

Consider the following extract from a PHP-driven web page that displays the current date:

```
Today is <?php echo date('j F Y');?>
```

The `<?php` tag tells PHP that everything that follows is program code rather than HTML, until the closing `?>` tag. In this example, the `echo` command tells PHP to display the next item to screen; the following `date` command produces a formatted version of the current date, containing the day, month, and year.



**The Statement Terminator** The semicolon character is used to indicate the end of a PHP command. In the previous examples, there is only one command, and the semicolon is not actually required, but it is good practice to always include it to show that a command is complete.

In this book PHP code appears inside tags that look like `<?php ... ?>`. Other tag styles can be used, so you may come across other people's PHP code beginning with tags that look like `<?` (the short tag), `<%` (the ASP tag style) or `<SCRIPT LANGUAGE="php">` (the script tag).

Of the different tag styles that can be used, only the full `<?php` tag and the script tag are always available. The others are turned off or on by using a PHP configuration setting. We will look at the `php.ini` configuration file in Lesson 23, “PHP Configuration.”



**Standard PHP Tags** It is good practice to always use the `<?php` tag style so your code will run on any system that has PHP installed, with no additional configuration needed. If you are tempted to use `<?` as a short-cut, know that any time you move your code to another web server, you need to be sure it will understand this tag style.

Anything that is not enclosed in PHP tags is passed straight through to the browser, exactly as it appears in the script. Therefore, in the previous example, the text `Today is` appears before the generated date when the page is displayed.

## Your First Script

Before you go any further, you need to make sure you can create and run PHP scripts as you go through the examples in this book. This could be on your own machine, and you can find instructions for installing PHP in Appendix A, “Installing PHP.” Also, many web hosting companies include PHP in their packages, and you may already have access to a suitable piece of web space.

Go ahead and create a new file called `time.php` that contains Listing 1.1, in a location that can be accessed by a PHP-enabled web server. This is a slight variation on the date example shown previously.

---

### LISTING 1.1 Displaying the System Date and Time

```
The time is
<?php echo date('H:i:s');?>
and the date is
<?php echo date('j F Y');?>
```

When you enter the URL to this file in your web browser, you should see the current date and time, according to the system clock on your web server, displayed.



**Running PHP Locally** If you are running PHP from your local PC, PHP code in a script will be executed only if it is accessed through a web server that has the PHP module enabled. If you open a local script directly in the web browser—for instance, by double-clicking or dragging and dropping the file into the browser—it will be treated as HTML only.



**Web Document Location** If you were using a default Apache installation in Windows, you would create `time.php` in the folder `C:\Program Files\Apache Group\Apache\htdocs`, and the correct URL would be `http://localhost/time.php`.

If you entered Listing 1.1 exactly as shown, you might notice that the actual output produced could be formatted a little better—there is no space between the time and the word *and*. Any line in a script that only contains code inside PHP tags will not take up a line of output in the generated HTML.

If you use the View Source option in your web browser, you can see the exact output produced by your script, which should look similar to the following:

```
The time is
15:33:09and the date is
13 October 2004
```

If you insert a space character after `?>`, that line now contains non-PHP elements, and the output is spaced correctly.

## The echo Command

While PHP is great for embedding small, dynamic elements inside a web page, in fact the whole page could consist of a set of PHP instructions to generate the output if the entire script were enclosed in PHP tags.

The echo command is used to send output to the browser. Listing 1.1 uses echo to display the result of the date command, which returns a string that contains a formatted version of the current date. Listing 1.2 does the same thing but uses a series of echo commands in a single block of PHP code to display the date and time.

### LISTING 1.2 Using echo to Send Output to the Browser

---

```
<?php
echo "The time is ";
echo date('H:i:s');
echo " and the date is ";
echo date('j F Y');
?>
```

The non-dynamic text elements you want to output are contained in quotation marks. Either double quotes (as used in Listing 1.2) or single quotes (the same character used for an apostrophe) can be used to enclose text strings, although you will see an important difference between the two styles in Lesson 2, “Variables.” The following statements are equally valid:

```
echo "The time is ";
echo 'The time is ';
```

Notice that space characters are used in these statements inside the quotation marks to ensure that the output from date is spaced away from the surrounding text. In fact the output from Listing 1.2 is slightly different from that for Listing 1.1, but in a web browser you will need to use View Source to see the difference. The raw output from Listing 1.2 is as follows:

```
The time is 15:59:50 and the date is 13 October 2004
```

There are no line breaks in the page source produced this time. In a web browser, the output looks just the same as for Listing 1.1 because in



HTML all whitespace, including carriage returns and multiple space or tab characters, is displayed as a single space in a rendered web page.

A newline character inside a PHP code block does not form part of the output. Line breaks can be used to format the code in a readable way, but several short commands could appear on the same line of code, or a long command could span several lines—that's why you use the semicolon to indicate the end of a command.

Listing 1.3 is identical to Listing 1.2 except that the formatting makes this script almost unreadable.

### LISTING 1.3 A Badly Formatted Script That Displays the Date and Time

```
<?php echo "The time is "; echo date('H:i:s'); echo
" and the date is "
; echo date(
'j F Y'
);
?>
```



**Using Newlines** If you wanted to send an explicit newline character to the web browser, you could use the character sequence `\n`. There are several character sequences like this that have special meanings, and you will see more of them in Lesson 6, “Working with Strings.”

## Comments

Another way to make sure your code remains readable is by adding comments to it. A *comment* is a piece of free text that can appear anywhere in a script and is completely ignored by PHP. The different comment styles supported by PHP are shown in Table 1.1.

**TABLE 1.1** Comment Styles in PHP

Comment	Description
// or #	Single-line comment. Everything to the end of the current line is ignored.
/* ... */	Single- or multiple-line comment. Everything between /* and */ is ignored.

Listing 1.4 produces the same formatted date and time as Listings 1.1, 1.2, and 1.3, but it contains an abundance of comments. Because the comments are just ignored by PHP, the output produced consists of only the date and time.

**LISTING 1.4** Using Comments in a Script

```
<?php
/* time.php
 This script prints the current date
 and time in the web browser
*/

echo "The time is ";
echo date('H:i:s'); // Hours, minutes, seconds

echo " and the date is ";
echo date('j F Y'); // Day name, month name, year
?>
```

Listing 1.4 includes a header comment block that contains the filename and a brief description, as well as inline comments that show what each date command will produce.

## Summary

In this lesson you have learned how PHP works in a web environment, and you have seen what a simple PHP script looks like. In the next lesson you will learn how to use variables.

# LESSON 2

## Variables



*In this lesson you will learn how to assign values to variables in PHP and use them in some simple expressions.*

### Understanding Variables

*Variables*—containers in which values can be stored and later retrieved—are a fundamental building block of any programming language.

For instance, you could have a variable called `number` that holds the value 5 or a variable called `name` that holds the value `Chris`. The following PHP code declares variables with those names and values:

```
$number = 5;
$name = "Chris";
```

In PHP, a variable name is always prefixed with a dollar sign. If you remember that, declaring a new variable is very easy: You just use an equals symbol with the variable name on the left and the value you want it to take on the right.



**Declaring Variables** Unlike in some programming languages, in PHP variables do not need to be declared before they can be used. You can assign a value to a new variable name any time you want to start using it.

Variables can be used in place of fixed values throughout the PHP language. The following example uses `echo` to display the value stored in a variable in the same way that you would display a piece of fixed text:

```
$name = "Chris";
echo "Hello, ";
echo $name;
```

The output produced is  
Hello, Chris

## Naming Variables

The more descriptive your variable names are, the more easily you will remember what they are used for when you come back to a script several months after you write it.

It is not usually a good idea to call your variables \$a, \$b, and so on. You probably won't remember what each letter stood for, if anything, for long. Good variable names tell exactly what kind of value you can expect to find stored in them (for example, \$price or \$name).



**Case-Sensitivity** Variable names in PHP are case-sensitive. For example, \$name is a different variable than \$Name, and the two could store different values in the same script.

Variable names can contain only letters, numbers, and the underscore character, and each must begin with a letter or underscore. Table 2.1 shows some examples of valid and invalid variable names.

**TABLE 2.1** Examples of Valid and Invalid Variable Names

Valid Variable Names	Invalid Variable Names
\$percent	\$pct%
\$first_name	\$first-name
\$line_2	\$2nd_line



**Using Underscores** Using the underscore character is a handy way to give a variable a name that is made up of two or more words. For example `$first_name` and `$date_of_birth` are more readable for having underscores in place.

Another popular convention for combining words is to capitalize the first letter of each word—for example, `$FirstName` and `$DateOfBirth`. If you prefer this style, feel free to use it in your scripts but remember that the capitalization does matter.

## Expressions

When a variable assignment takes place, the value given does not have to be a fixed value. It could be an *expression*—two or more values combined using an *operator* to produce a result. It should be fairly obvious how the following example works, but the following text breaks it down into its components:

```
$sum = 16 + 30;
echo $sum;
```

The variable `$sum` takes the value of the expression to the right of the equals sign. The values 16 and 30 are combined using the addition operator—the plus symbol (+)—and the result of adding the two values together is returned. As expected, this piece of code displays the value 46.

To show that variables can be used in place of fixed values, you can perform the same addition operation on two variables:

```
$a = 16;
$b = 30;
$sum = $a + $b;
echo $sum;
```

The values of `$a` and `$b` are added together, and once again, the output produced is 46.

## Variables in Strings

You have already seen that text strings need to be enclosed in quotation marks and learned that there is a difference between single and double quotes.

The difference is that a dollar sign in a double-quoted string indicates that the current value of that variable should become part of the string. In a single-quoted string, on the other hand, the dollar sign is treated as a literal character, and no reference is made to any variables.

The following examples should help explain this. In the following example, the value of variable `$name` is included in the string:

```
$name = "Chris";
echo "Hello, $name";
```

This code displays `Hello, Chris`.

In the following example, this time the dollar sign is treated as a literal, and no variable substitution takes place:

```
$name = 'Chris';
echo 'Hello, $name';
```

This code displays `Hello, $name`.

Sometimes you need to indicate to PHP exactly where a variable starts and ends. You do this by using curly brackets, or *braces* (`{}`). If you wanted to display a weight value with a suffix to indicate pounds or ounces, the statement might look like this:

```
echo "The total weight is {$weight}lb";
```

If you did not use the braces around `$weight`, PHP would try to find the value of `$weightlb`, which probably does not exist in your script.

You could do the same thing by using the *concatenation* operator, the period symbol, which can be used to join two or more strings together, as shown in the following example:

```
echo 'The total weight is ' . $weight . 'lb';
```

The three values—two fixed strings and the variable `$weight`—are simply stuck together in the order in which they appear in the statement. Notice that a space is included at the end of the first string because you want the value of `$weight` to be joined to the word `is`.

If `$weight` has a value of `99`, this statement will produce the following output:

```
The total weight is 99lb
```

## Data Types

Every variable that holds a value also has a data type that defines what kind of value it is holding. The basic data types in PHP are shown in Table 2.2.

**TABLE 2.2** PHP Data Types

Data Type	Description
Boolean	A truth value; can be either <code>TRUE</code> or <code>FALSE</code> .
Integer	A number value; can be a positive or negative whole number.
Double (or float)	A floating-point number value; can be any decimal number.
String	An alphanumeric value; can contain any number of ASCII characters.

When you assign a value to a variable, the data type of the variable is also set. PHP determines the data type automatically, based on the value you assign. If you want to check what data type PHP thinks a value is, you can use the `gettype` function.

Running the following code shows that the data type of a decimal number is `double`:

```
$value = 7.2;
echo gettype($value);
```

The complementary function to `gettype` is `settype`, which allows you to override the data type of a variable. If the stored value is not suitable to be stored in the new type, it will be modified to the closest value possible.

The following code attempts to convert a string value into an integer:

```
$value = "22nd January 2005";
settype($value, "integer");
echo $value;
```

In this case, the string begins with numbers, but the whole string is not an integer. The conversion converts everything up to the first nonnumeric character and discards the rest, so the output produced is just the number 22.



**Analyzing Data Types** In practice, you will not use `settype` and `gettype` very often because you will rarely need to alter the data type of a variable. This book covers this topic early on so that you are aware that PHP does assign a data type to every variable.

## Type Juggling

Sometimes PHP will perform an implicit data type conversion if values are expected to be of a particular type. This is known as *type juggling*.

For example, the addition operator expects to sit between two numbers. String type values are converted to double or integer before the operation is performed, so the following addition produces an integer result:

```
echo 100 + "10 inches";
```

This expression adds 100 and 10, and it displays the result 110.

A similar thing happens when a string operator is used on numeric data. If you perform a string operation on a numeric type, the numeric value is converted to a string first. In fact, you already saw this earlier in this lesson, with the concatenation operator—the value of `$weight` that was displayed was numeric.



The result of a string operation will always be a string data type, even if it looks like a number. The following example produces the result 69, but—as `gettype` shows—`$number` contains a string value:

```
$number = 6 . 9;
echo $number;
echo gettype(6 . 9);
```

We will look at the powerful range of operators that are related to numeric and string data types in PHP in Lessons 5, “Working with Numbers,” and 6, “Working with Strings.”

## Variable Variables

It is possible to use the value stored in a variable as the name of another variable. If this sounds confusing, the following example might help:

```
$my_age = 21;
$varname = "my_age";
echo "The value of $varname is ${$varname}";
```

The output produced is

```
The value of my_age is 21
```

Because this string is enclosed in double quotes, a dollar sign indicates that a variable’s value should become part of the string. The construct `${$varname}` indicates that the value of the variable named in `$varname` should become part of the string and is known as a *variable variable*.

The braces around `$varname` are used to indicate that it should be referenced first; they are required in double-quoted strings but are otherwise optional. The following example produces the same output as the preceding example, using the concatenation operator:

```
echo 'The value of ' . $varname . ' is ' . $$varname;
```

## Summary

In this lesson you have learned how variables work in PHP. In the next lesson you will see how to use conditional and looping statements to control the flow of your script.



## LESSON 3

# Flow Control

*In this lesson you will learn about the conditional and looping constructs that allow you to control the flow of a PHP script.*

In this chapter we'll look at two types of flow control: conditional statements, which tell your script to execute a section of code only if certain criteria are met, and loops, which indicate a block of code that is to be repeated a number of times.

### Conditional Statements

A conditional statement in PHP begins with the keyword `if`, followed by a condition in parentheses. The following example checks whether the value of the variable `$number` is less than 10, and the `echo` statement displays its message only if this is the case:

```
$number = 5;
if ($number < 10) {
 echo "$number is less than ten";
}
```

The condition `$number < 10` is satisfied if the value on the left of the `<` symbol is smaller than the value on the right. If this condition holds true, then the code in the following set of braces will be executed; otherwise, the script jumps to the next statement after the closing brace.



**Boolean Values** Every conditional expression evaluates to a Boolean value, and an `if` statement simply acts on a `TRUE` or `FALSE` value to determine whether the next block of code should be executed. Any zero value in PHP is considered `FALSE`, and any nonzero value is considered `TRUE`.

As it stands, the previous example will be `TRUE` because 5 is less than 10, so the statement in braces is executed, and the corresponding output is displayed. Now, if you change the initial value of `$number` to 10 or higher and rerun the script, the condition fails, and no output is produced.

Braces are used in PHP to group blocks of code together. In a conditional statement, they surround the section of code that is to be executed if the preceding condition is true.



**Brackets and Braces** You will come across three types of brackets when writing PHP scripts. The most commonly used terminology for each type is parentheses (`()`), braces (`{}`), and square brackets (`[]`).

Braces are not required after an `if` statement. If they are omitted, the following single statement is executed if the condition is true. Any subsequent statements are executed, regardless of the status of the conditional.



**Braces and Indentation** Although how your code is indented makes no difference to PHP, it is customary to indent blocks of code inside braces with a few space characters to visually separate that block from other statements.

Even if you only want a condition or loop to apply to one statement, it is still useful to use braces for clarity. It is particularly important in order to keep things readable when you're nesting multiple constructs.

## Conditional Operators

PHP allows you to perform a number of different comparisons, to check for the equality or relative size of two values. PHP's conditional operators are shown in Table 3.1.

**TABLE 3.1** Conditional Operators in PHP

Operator	Description
==	Is equal to
===	Is identical to (is equal and is the same data type)
!=	Is not equal to
!==	Is not identical to
<	Is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to



**= or ==?** Be careful when comparing for equality to use a double equals symbol (==). A single = is always an assignment operator and, unless the value assigned is zero, your condition will always return true—and remember that TRUE is any nonzero value. Always use == when comparing two values to avoid headaches.

## Logical Operators

You can combine multiple expressions to check two or more criteria in a single conditional statement. For example, the following statement checks whether the value of `$number` is between 5 and 10:

```
$number = 8;
if ($number >= 5 and $number <= 10) {
 echo "$number is between five and ten";
}
```

The keyword `and` is a *logical operator*, which signifies that the overall condition will be true only if the expressions on either side are true. That is, `$number` has to be both greater than or equal to 5 and less than or equal to 10.

Table 3.2 shows the logical operators that can be used in PHP.

**TABLE 3.2** Logical Operators in PHP

Operator	Name	Description
<code>! a</code>	NOT	True if <i>a</i> is not true
<code>a &amp;&amp; b</code>	AND	True if both <i>a</i> and <i>b</i> are true
<code>a    b</code>	OR	True if either <i>a</i> or <i>b</i> is true
<code>a and b</code>	AND	True if both <i>a</i> and <i>b</i> are true
<code>a xor b</code>	XOR	True if <i>a</i> or <i>b</i> is true, but not both
<code>a or b</code>	OR	True if either <i>a</i> or <i>b</i> is true

You may have noticed that there are two different ways of performing a logical AND or OR in PHP. The difference between `and` and `&&` (and between `or` and `||`) is the precedence used to evaluate expressions.

Table 3.2 lists the highest-precedence operators first. The following conditions, which appear to do the same thing, are subtly but significantly different:

`a or b and c`  
`a || b and c`

In the former condition, the `and` takes precedence and is evaluated first. The overall condition is true if *a* is true or if both *b* and *c* are true.

In the latter condition, the `||` takes precedence, so *c* must be true, as must either *a* or *b*, to satisfy the condition.



**Operator Symbols** Note that the logical AND and OR operators are the double symbols `&&` and `||`, respectively. These symbols, when used singularly, have a different meaning, as you will see in Lesson 5, “Working with Numbers.”

## Multiple Condition Branches

By using an `else` clause with an `if` statement, you can specify an alternate action to be taken if the condition is not met. The following example tests the value of `$number` and displays a message that says whether it is greater than or less than 10:

```
$number = 16;
if ($number < 10) {
 echo "$number is less than ten";
}
else {
 echo "$number is more than ten";
}
```

The `else` clause provides an either/or mechanism for conditional statements. To add more branches to a conditional statement, the `elseif` keyword can be used to add a further condition that is checked only if the previous condition in the statement fails.

The following example uses the `date` function to find the current time of day—`date("H")` gives a number between 0 and 23 that represents the hour on the clock—and displays an appropriate greeting:

```
$hour = date("H");
if ($hour < 12) {
 echo "Good morning";
}
elseif ($hour < 17) {
 echo "Good afternoon";
}
else {
 echo "Good evening";
}
```

This code displays `Good morning` if the server time is between midnight and 11:59, `Good afternoon` from midday to 4:59 p.m., and `Good evening` from 5 p.m. onward.

Notice that the `elseif` condition only checks that `$hour` is less than 17 (5 p.m.). It does not need to check that the value is between 12 and 17 because the initial `if` condition ensures that PHP will not get as far as the `elseif` if `$hour` is less than 12.

The code in the `else` clause is executed if all else fails. For values of `$hour` that are 17 or higher, neither the `if` nor the `elseif` condition will be true.



**elseif Versus else if** In PHP you can also write `elseif` as two words: `else if`. The way PHP interprets this variation is slightly different, but its behavior is exactly the same.

## The switch Statement

An `if` statement can contain as many `elseif` clauses as you need, but including many of these clauses can often create cumbersome code, and an alternative is available. `switch` is a conditional statement that can have multiple branches in a much more compact format.

The following example uses a `switch` statement to check `$name` against two lists to see whether it belongs to a friend:

```
switch ($name) {
 case "Damon":
 case "Shelley":
 echo "Welcome, $name, you are my friend";
 break;
 case "Adolf":
 case "Saddam":
 echo "You are no friend of mine, $name";
 break;
 default:
 echo "I do not know who you are, $name";
}
```

Each case statement defines a value for which the next block of PHP code will be executed. If you assign your first name to `$name` and run this script, you will be greeted as a friend if your name is Damon or Shelley, and you will be told that you are not a friend if your name is either Adolf or Saddam. If you have any other name, the script will tell you it does not know who you are.

There can be any number of case statements preceding the PHP code to which they relate. If the value that is being tested by the switch statement (in this case `$name`) matches any one of them, any subsequent PHP code will be executed until a break command is reached.



**Breaking Out** The `break` statement is important in a switch statement. When a case statement has been matched, any PHP code that follows will be executed—even if there is another case statement checking for a different value. This behavior can sometimes be useful, but mostly it is not what you want—so remember to put a `break` after every case.

Any other value for `$name` will cause the default code block to be executed. As with an `else` clause, `default` is optional and supplies an action to be taken if nothing else is appropriate.

## Loops

PHP offers three types of loop constructs that all do the same thing—repeat a section of code a number of times—in slightly different ways.

### The while Loop

The `while` keyword takes a condition in parentheses, and the code block that follows is repeated while that condition is true. If the condition is false initially, the code block will not be repeated at all.



**Infinite Loops** The repeating code must perform some action that affects the condition in such a way that the loop condition will eventually no longer be met; otherwise, the loop will repeat forever.



The following example uses a `while` loop to display the square numbers from 1 to 10:

```
$count = 1;
while ($count <= 10) {
 $square = $count * $count;
 echo "$count squared is $square
";
 $count++;
}
```

The counter variable `$count` is initialized with a value of 1. The `while` loop calculates the square of that number and displays it, then adds one to the value of `$count`. The `++` operator adds one to the value of the variable that precedes it.

The loop repeats while the condition `$count <= 10` is true, so the first 10 numbers and their squares are displayed in turn, and then the loop ends.

## The do Loop

The `do` loop is very similar to the `while` loop except that the condition comes after the block of repeating code. Because of this variation, the loop code is always executed at least once—even if the condition is initially false.

The following `do` loop is equivalent to the previous example, displaying the numbers from 1 to 10, with their squares:

```
$count = 1;
do {
 $square = $count * $count;
 echo "$count squared is $square
";
 $count++;
} while ($count <= 10);
```

## The for Loop

The `for` loop provides a compact way to create a loop. The following example performs the same loop as the previous two examples:

```
for ($count = 1; $count <= 10; $count++) {
 $square = $count * $count;
 echo "$count squared is $square
";
}
```

As you can see, using `for` allows you to use much less code to do the same thing as with `while` and `do`.

A `for` statement has three parts, separated by semicolons:

- The first part is an expression that is evaluated once when the loop begins. In the preceding example, you initialized the value of `$count`.
- The second part is the condition. While the condition is true, the loop continues repeating. As with a `while` loop, if the condition is false to start with, the following code block is not executed at all.
- The third part is an expression that is evaluated once at the end of each pass of the loop. In the previous example, `$count` is incremented after each line of the output is displayed.

## Nesting Conditions and Loops

So far you have only seen simple examples of conditions and loops. However, you can nest these constructs within each other to create some quite complex rules to control the flow of a script.



**Remember to Indent** The more complex the flow control in your script is, the more important it becomes to indent your code to make it clear which blocks of code correspond to which constructs.

## Breaking Out of a Loop

You have already learned about using the keyword `break` in a `switch` statement. You can also use `break` in a loop construct to tell PHP to immediately exit the loop and continue with the rest of the script.

The `continue` keyword is used to end the current pass of a loop. However, unlike with `break`, the script jumps back to the top of the same loop and continues execution until the loop condition fails.

## Summary

In this lesson you have learned how to vary the flow of your PHP script by using conditional statements and loops. In the next lesson you will see how to create reusable functions from blocks of PHP code.



# LESSON 4

## Functions

*In this lesson you will learn how frequently used sections of code can be turned into reusable functions.*

### Using Functions

A *function* is used to make a task that might consist of many lines of code into a routine that can be called using a single instruction.

PHP contains many functions that perform a wide range of useful tasks. Some are built in to the PHP language; others are more specialized and are available only if certain extensions are activated when PHP is installed.

The online PHP manual ([www.php.net](http://www.php.net)) is an invaluable reference. As well as documentation for every function in the language, the manual pages are also annotated with user-submitted tips and examples, and you can even submit your own comments if you want.



**Online Reference** To quickly pull up the PHP manual page for any function, use this shortcut:  
[www.php.net/function\\_name](http://www.php.net/function_name).

You have already used the `date` function to generate a string that contains a formatted version of the current date. Let's take a closer look at how that example from Lesson 1, "Getting to Know PHP," works. The example looked like this:

```
echo date('j F Y');
```

The online PHP manual gives the prototype for `date` as follows:

```
string date (string format [, int timestamp])
```

This means that `date` takes a string argument called `format` and, optionally, the integer `timestamp`. It returns a string value. This example sends `Y` to the function as the `format` argument, but `timestamp` is omitted. The `echo` command displays the string that is returned.



**Prototypes** Every function has a prototype that defines how many arguments it takes, the arguments' data types, and what value is returned. Optional arguments are shown in square brackets (`[]`).

## Defining Functions

In addition to the built-in functions, PHP allows you to define your own. There are advantages to using your own function. Not only do you have to type less when the same piece of code has to be executed several times but a custom-defined function also makes your script easier to maintain. If you want to change the way a task is performed, you only need to update the program code once—in the function definition—rather than fix it every place it appears in your script.



**Modular Code** Grouping tasks into functions is the first step toward *modularizing* your code—something that is especially important to keep your scripts manageable as they grow in size and become more complex.

The following is a simple example that shows how a function is defined and used in PHP:

```
function add_tax($amount) {
 $total = $amount * 1.09;
 return $total;
}

$price = 16.00;
echo "Price before tax: $price
";
```

```
echo "Price after tax: ";
echo add_tax($price);
```

The `function` keyword defines a function called `add_tax` that will execute the code block that follows. The code that makes up a function is always contained in braces. Putting `$amount` in parentheses after the function name stipulates that `add_tax` takes a single argument that will be stored in a variable called `$amount` inside the function.

The first line of the function code is a simple calculation that multiplies `$amount` by 1.09—which is equivalent to adding 9% to that value—and assigns the result to `$total`. The `return` keyword is followed by the value that is to be returned when the function is called from within the script.

Running this example produces the following output:

```
Price before tax: 16
Price after tax: 17.44
```

This is an example of a function that you might use in many places in a web page; for instance, on a page that lists all the products available in an online store, you would call this function once for each item that is displayed to show the after-tax price. If the rate of tax changes, you only need to change the formula in `add_tax` to alter every price displayed on that page.

## Arguments and Return Values

Every function call consists of the function name followed by a list of arguments in parentheses. If there is more than one argument, the list items are separated with commas. Some functions do not require any arguments at all, but a pair of parentheses is still required—even if there are no arguments contained in them.

The built-in function `phpinfo` generates a web page that contains a lot of information about the PHP module. This function does not require any arguments, so it can be called from a script that is as simple as

```
<?php phpinfo();?>
```

If you create this script and point a web browser at it, you will see a web page that contains system information and configuration settings.

## Returning Success or Failure

Because `phpinfo` generates its own output, you do not need to prefix it with `echo`, but, for the same reason, you cannot assign the web page it produces to a variable. In fact, the return value from `phpinfo` is the integer value 1.



**Returning True and False** Functions that do not have an explicit return value usually use a return code to indicate whether their operation has completed successfully. A zero value (`FALSE`) indicates failure, and a nonzero value (`TRUE`) indicates success.

The following example uses the `mail` function to attempt to send an email from a PHP script. The first three arguments to `mail` specify the recipient's email address, the message subject, and the message body. The return value of `mail` is used in an `if` condition to check whether the function was successful:

```
if (mail("chris@lightwood.net",
 "Hello", "This is a test email")) {
 echo "Email was sent successfully";
}
else {
 echo "Email could not be sent";
}
```

If the web server that this script is run on is not properly configured to send email, or if there is some other error when trying to send, `mail` will return zero, indicating that the email could not be sent. A nonzero value indicates that the message was handed off to your mail server for sending.



**Return Values** Although you will not always need to test the return value of every function, you should be aware that every function in PHP does return some value.

## Default Argument Values

The `mail` function is an example of a function that takes multiple arguments; the recipient, subject, and message body are all required. The prototype for `mail` also specifies that this function can take an optional fourth argument, which can contain additional mail headers.

Calling `mail` with too few arguments results in a warning. For instance, a script that contains the following:

```
mail("chris@lightwood.net", "Hello");
```

will produce a warning similar to this:

```
Warning: mail() expects at least 3 parameters, 2 given in
/home/chris/mail.php on line 3
```

However, the following two calls to `mail` are both valid:

```
mail("chris@lightwood.net", "Hello", "This is a test email");
```

```
mail("chris@lightwood.net", "Hello", "This is a test email",
 "Cc: editor@sampublishing.com");
```

To have more than one argument in your own function, you simply use a comma-separated list of variable names in the function definition. To make one of these arguments optional, you assign it a default value in the argument list, the same way you would assign a value to a variable.

The following example is a variation of `add_tax` that takes two arguments—the net amount and the tax rate to add on. `$rate` has a default value of 10, so it is an optional argument:

```
function add_tax_rate($amount, $rate=10) {
 $total = $amount * (1 + ($rate / 100));
 return($total);
}
```



Using this function, the following two calls are both valid:

```
add_tax_rate(16);
add_tax_rate(16, 9);
```

The first example uses the default rate of 10%, whereas the second example specifies a rate of 9% to be used—producing the same behavior as the original `add_tax` function example.



**Optional Arguments** All the optional arguments to a function must appear at the end of the argument list, with the required values passed in first. Otherwise, PHP will not know which arguments you are passing to the function.

## Variable Scope

The reason values have to be passed in to functions as arguments has to do with *variable scope*—the rules that determine what sections of script are able to access which variables.

The basic rule is that any variables defined in the main body of the script cannot be used inside a function. Likewise, any variables used inside a function cannot be seen by the main script.



**Scope** Variables available within a function are said to be *local variables* or that their scope is local to that function. Variables that are not local are called *global variables*.

Local and global variables can have the same name and contain different values, although it is best to try to avoid this to make your script easier to read.

When called, `add_tax` calculates `$total`, and this is the value returned. However, even after `add_tax` is called, the local variable `$total` is undefined outside that function.

The following piece of code attempts to display the value of a global variable from inside a function:

```
function display_value() {
 echo $value;
}
```

```
$value = 125;
display_value();
```

If you run this script, you will see that no output is produced because `$value` has not been declared in the local scope.

To access a global variable inside a function, you must use the `global` keyword at the top of the function code. Doing so overrides the scope of that variable so that it can be read and altered within the function. The following code shows an example:

```
function change_value() {
 global $value;
 echo "Before: $value
";
 $value = $value * 2;
}
$value = 100;
display_value();
echo "After: $value
";
```

The value of `$value` can now be accessed inside the function, so the output produced is as follows:

```
Before: 100
After: 200
```

## Using Library Files

After you have created a function that does something useful, you will probably want to use it again in other scripts. Rather than copy the function definition into each script that needs to use it, you can use a library file so that your function needs to be stored and maintained in only one place.

Before you go any further, you should create a library file called `tax.php` that contains both the `add_tax` and `add_tax_rate` functions but no other PHP code.



**Using Library Files** A library file needs to enclose its PHP code inside `<?php` tags just like a regular script; otherwise, the contents will be displayed as HTML when they are included in a script.

## Including Library Files

To incorporate an external library file into another script, you use the `include` keyword. The following includes `tax.php` so that `add_tax` can be called in that script:

```
include "tax.php";
$price = 95;
echo "Price before tax: $price
";
echo "Price after tax: ";
echo add_tax($price);
```



**The `include` path Setting** By default, `include` searches only the current directory and a few system locations for files to be included. If you want to include files from another location, you can use a path to the file.

You can extend the `include` path to include other locations without a path being required by changing the value of the `include_path` setting. Refer to Lesson 23, “PHP,” for more information.

You can use the `include_once` keyword if you want to make sure that a library file is loaded only once. If a script attempts to define the same function a second time, an error will result. Using `include_once` helps to avoid this, particularly when files are being included from other library files. It is often useful to have a library file that includes several other files, each containing a few functions, rather than one huge library file.



**Require** The `require` and `require_once` instructions work in a similar way to `include` and `include_once` but have subtly different behavior. In the event of an error, `include` generates a warning, but the script carries on running as best it can. A failure from a `require` statement causes the script to exit immediately.

## Summary

In this lesson you have learned how to use functions to modularize your code. In the next lesson you will learn about ways to work with numeric data in PHP.

# LESSON 5

## Working with Numbers



*In this lesson you will learn about some of the numeric manipulations you can perform in PHP.*

### Arithmetic

As you would expect, PHP includes all the basic arithmetic operators. If you have not used another programming language, the symbols used might not all be obvious, so we'll quickly run through the basic rules of arithmetic in PHP.

### Arithmetic Operators

Addition is performed with the plus symbol (+). This example adds 6 and 12 together and displays the result:

```
echo 6 + 12;
```

Subtraction is performed with the minus symbol (-), which is also used as a hyphen. This example subtracts 5 from 24:

```
echo 24 - 5;
```

The minus symbol can also be used to negate a number (for example, -20).

Multiplication is performed with the asterisk symbol (\*). This example displays the product of 4 and 9:

```
echo 4 * 9;
```

Division is performed with the forward slash symbol (/). This example divides 48 by 12:

```
echo 48 / 12;
```



**Division** When you divide two integers, the result is an integer if it divides exactly. Otherwise, it is a double. A fractional result is not rounded to an integer.

Modulus is performed by using the percent symbol (%). This example displays 3—the remainder of 21 divided by 6:

```
echo 21 % 6;
```



**Modulus** The modulus operator can be used to test whether a number is odd or even by using `$number % 2`. The result will be 0 for all even numbers and 1 for all odd numbers (because any odd number divided by 2 has a remainder of 1).

## Incrementing and Decrementing

In PHP you can increment or decrement a number by using a double plus (++) or double minus (--) symbol. The following statements both add one to `$number`:

```
$number++;
```

```
++$number;
```

The operator can be placed on either side of a variable, and its position determines at what point the increment takes place.

This statement subtracts one from `$countdown` before displaying the result:

```
echo --$countdown;
```

However, the following statement displays the current value of `$countdown` before decrementing it:

```
echo $countdown--;
```

The increment and decrement operators are commonly used in loops. The following is a typical for loop, using a counter to repeat a section of code 10 times:

```
for ($count=1; $count<=10; $count++) {
 echo "Count = $count
";
}
```

In this case, the code simply outputs the value of `$count` for each pass of the loop.

## Compound Operators

Compound operators provide a handy shortcut when you want to apply an arithmetic operation to an existing variable. The following example uses the compound addition operator to add six to the current value of `$count`:

```
$count += 6;
```

The effect of this is to take the initial value of `$count`, add six to it, and then assign it back to `$count`. In fact, the operation is equivalent to doing the following:

```
$count = $count + 6;
```

All the basic arithmetic operators have corresponding compound operators, as shown in Table 5.1.

**TABLE 5.1** Compound Operators

Operator	Equivalent To
<code>\$a += \$b</code>	<code>\$a = \$a + \$b;</code>
<code>\$a -= \$b</code>	<code>\$a = \$a - \$b;</code>
<code>\$a *= \$b</code>	<code>\$a = \$a * \$b;</code>
<code>\$a /= \$b</code>	<code>\$a = \$a / \$b;</code>
<code>\$a %= \$b</code>	<code>\$a = \$a % \$b;</code>

## Operator Precedence

The rules governing operator precedence specify the order in which expressions are evaluated. For example, the following statement is ambiguous:

```
echo 3 * 4 + 5;
```

Are 3 and 4 multiplied together, and then 5 is added to the result, giving a total of 17? Or are 4 and 5 added together first and multiplied by 3, giving 27? Running this statement in a script will show you that in PHP, the result is 17.

The reason is that multiplication has a higher precedence than addition, so when these operators appear in the same expression, multiplication takes place first, using the values that immediately surround the multiplication operator.

To tell PHP that you explicitly want the addition to take place first, you can use parentheses, as in the following example:

```
echo 3 * (4 + 5);
```

In this case, the result is 27.

In PHP, the precedence of arithmetic operators follows the PEMDAS rule that you may have learned at school: parentheses, exponentiation, multiplication/division, and addition/subtraction.

The full operator precedence list for PHP, including many operators you haven't come across yet, can be found in the online manual at [www.php.net/manual/en/language.operators.php](http://www.php.net/manual/en/language.operators.php).

## Numeric Data Types

You have already seen that PHP assigns a data type to each value and that the numeric data types are integer and double, for whole numbers.

To check whether a value is either of these types, you use the `is_float` and `is_int` functions. Likewise, to check for either numeric data type in one operation, you can use `is_numeric`.



The following example contains a condition that checks whether the value of `$number` is an integer:

```
$number = "28";
if (is_int($number)) {
 echo "$number is an integer";
}
else {
 echo "$number is not an integer";
}
```

Because the actual declaration of that variable assigns a string value—albeit one that contains a number—the condition fails.

Although `$number` in the previous example is a string, PHP is flexible enough to allow this value to be used in numeric operations. The following example shows that a string value that contains a number can be incremented and that the resulting value is an integer:

```
$number = "6";
$number++;
echo "$number has type " . gettype($number);
```

## Understanding NULLs

The value `NULL` is a data type all to itself—a value that actually has no value. It has no numeric value, but comparing to an integer value zero evaluates to true:

```
$number = 0;
$empty=NULL;
if ($number == $empty) {
 echo "The values are the same";
}
```



**Type Comparisons** If you want to check that both the values and data types are the same in a condition, you use the triple equals comparison operator (`===`).

# Numeric Functions

Let's take a look at some of the numeric functions available in PHP.

## Rounding Numbers

There are three different PHP functions for rounding a decimal number to an integer.

You use `ceil` or `floor` to round a number up or down to the nearest integer, respectively. For example, `ceil(1.3)` returns 2, whereas `floor(6.8)` returns 6.



**Negative Rounding** Note the way that negative numbers are rounded. The result of `floor(-1.1)` is -2—the next lowest whole number numerically—not -1. Similarly, `ceil(-2.5)` returns -2.

To round a value to the nearest whole number, you use `round`. A fractional part under .5 will be rounded down, whereas .5 or higher will be rounded up. For example, `round(1.3)` returns 1, whereas `round(1.5)` returns 2.

The `round` function can also take an optional precision argument. The following example displays a value rounded to two decimal places:

```
$score = 0.535;
echo round($score, 2);
```

The value displayed is 0.54; the third decimal place being 5 causes the final digit to be rounded up.

You can also use `round` with a negative precision value to round an integer to a number of significant figures, as in the following example:

```
$distance = 2834;
echo round($distance, -2);
```

## Comparisons

To find the smallest and largest of a group of numbers, you use `min` and `max`, respectively. These functions take two or more arguments and return the numerically lowest or highest element in the list, respectively.

This statement will display the larger of the two variables `$a` and `$b`:

```
echo max($a, $b);
```

There is no limit to the number of arguments that can be compared. The following example finds the lowest value from a larger set of values:

```
echo min(6, 10, 23, 3, 88, 102, 5, 44);
```

Not surprisingly, the result displayed is 3.

## Random Numbers

You use `rand` to generate a random integer, using your system's built-in random number generator. The `rand` function optionally takes two arguments that specify the range of numbers from which the random number will be picked.



**Random Limit** The constant `RAND_MAX` contains the highest random number value that can be generated on your system. This value may vary between different platforms.

The following statement picks a random number between 1 and 10 and displays it:

```
echo rand(1, 10);
```

You can put this command in a script and run it a few times to see that the number changes each time it is run.

There is really no such thing as a computer-generated random number. In fact, numbers are actually picked from a very long sequence that has very

similar properties to true random numbers. To make sure you always start from a different place in this sequence, you have to *seed* the random number generator by calling the `srand` function; no arguments are required.



**Random Algorithms** PHP includes another random number generator, known as Mersenne Twister, that is considered to produce better random results than `rand`. To use this algorithm, you use the functions `mt_rand` and `mt_srand`.

## Mathematical Functions

PHP includes many mathematical functions, including trigonometry, logarithms, and number base conversions. As you will rarely need to use these in a web environment, those functions are not covered in this book.

To find out about a function that performs a specific mathematical purpose, refer to the online manual at [www.php.net/manual/en/ref.math.php](http://www.php.net/manual/en/ref.math.php).

## Summary

In this lesson you have learned how to work with numbers. In the next lesson you will learn all about string handling in PHP.

## LESSON 6

# Working with Strings



*In this lesson you will learn about some of the powerful string functions that are included in the PHP language.*

## Anatomy of a String

A *string* is a collection of characters that is treated as a single entity. In PHP, strings are enclosed in quotation marks, and you can declare a string type variable by assigning it a string that is contained in either single or double quotes.

The following examples are identical; both create a variable called `$phrase` that contains the phrase shown:

```
$phrase = "The sky is falling";
$phrase = 'The sky is falling';
```



**Quote Characters** Quotation marks in PHP do not point in a direction. The same symbol is used to start a string as to indicate the end. You must use two apostrophe characters (') around a single-quoted string—do not use backtick characters (`).

## Escaping Characters with Backslash

Double quotes can be used within single-quoted strings and vice versa. For instance, these string assignments are both valid:

```
$phrase = "It's time to party!";
$phrase = 'So I said, "OK"';
```

However, if you want to use the same character within a quoted string, you must escape that quote by using a backslash. The following examples demonstrate this:

```
$phrase = 'It\'s time to party!';
$phrase = "So I said, \"OK\"";
```

In the previous examples, if the backslash were not used, PHP would mismatch the quotes, and an error would result.

Which style of quoting you use largely depends on personal preference and, hopefully, a desire to create tidy code. Remember, though, as you saw in Lesson 2, “Variables,” that a variable prefixed with a dollar sign inside a double-quoted string is replaced with its values, whereas in a single-quoted string, the dollar sign and variable name appear verbatim.

If you want a dollar sign to form part of a double-quoted string, you can also escape this by using a backslash. For example, the following two statements are equivalent:

```
$offer = 'Save $10 on first purchase';
$offer = "Save \$10 on first purchase";
```

Without the backslash, the second example would attempt to find the value of a variable called \$10, which is, in fact, an illegal variable name.

The backslash character can also be used in a double-quoted string to indicate some special values inside strings. When followed by a three-digit number, it indicates the ASCII character with that octal value.

You can send the common nonprintable ASCII characters by using standard escape characters. A newline is `\n`, tab is `\t`, and so on. Refer to `man ascii` on your system or [www.ascii.cl](http://www.ascii.cl) for a comprehensive list.

## Concatenation

You have already seen how strings can be joined using the period symbol as a concatenation operator. A compound version of this operator, `.=`, can be used to append a string to an existing variable.

The following example builds up a string in stages and then displays the result:

```
$phrase = "I want ";
$phrase .= "to teach ";
$phrase .= "the world ";
$phrase .= "to sing";
echo $phrase;
```

The phrase appears as expected. Note the use of spaces after `teach` and `world` to ensure that the final string is correctly spaced.

## Comparing Strings

You can compare string values simply by using the standard comparison operators. To check whether two strings are equal, you use the double equals (`==`) sign:

```
if ($password == "letmein")
 echo "You have a guessable password";
```

The equality operator, when applied to strings, performs a case-sensitive comparison. In the previous example, any other capitalization of `$password`, such as `LetMeIn`, would not pass this test.

The inequality operators—`<`, `<=`, `>`, and `>=`—perform a comparison based on the ASCII values of the individual characters in the strings. The following condition could be used to divide people into two groups, based on their last name—those with names beginning A–M and those beginning N–Z:

```
if ($last_name < "N")
 echo "You are in group 1";
else
 echo "You are in group 2";
```



**ASCII Values** Because string comparisons are done on their underlying ASCII values, all lowercase letters have higher values than their equivalent uppercase letters. Letters a–z have values 97–122, whereas A–Z occupy values 65–90.

## Formatting Strings

PHP provides a powerful way of creating formatted strings, using the `printf` and `sprintf` functions. If you have used this function in C, these will be quite familiar to you, although the syntax in PHP is a little different.

### Using `printf`

You use `printf` to display a formatted string. At its very simplest, `printf` takes a single string argument and behaves the same as `echo`:

```
printf("Hello, world");
```

The power of `printf`, however, lies in its ability to substitute values into placeholders in a string. Placeholders are identified by the percent character (%), followed by a format specification character.

The following example uses the simple format specifier `%f` to represent a float number.

```
$price = 5.99;
printf("The price is %f", $price);
```

The second argument to `printf` is substituted in place of `%f`, so the following output is produced:

```
The price is 5.99
```

There is actually no limit to the number of substitution arguments in a `printf` statement, as long as there are an equivalent number of placeholders in the string to be displayed. The following example demonstrates this by adding in a string item:

```
$item = "The Origin of Species";
$price = 5.99;
printf("The price of %s is %f", $item, $price);
```

Table 6.1 shows the format characters that can be used with the `printf` function in PHP to indicate different types of values.



**TABLE 6.1** printf Format Characters

Character	Meaning
b	A binary (base 2) number
c	The ASCII character with the numeric value of the argument
d	A signed decimal (base 10) integer
e	A number displayed in scientific notation (for example, 2.6e+3)
u	An unsigned decimal integer
f	A floating-point number
o	An octal (base 8) number
s	A string
x	A hexadecimal (base 16) number with lowercase letters
X	A hexadecimal (base 16) number with uppercase letters

Suppose you use the `%d` format specifier instead of `%f` to display the value of `$price`:

```
$price = 5.99;
printf("As a decimal, the price is %d", $price);
```

In this case, PHP will treat the argument passed as an integer, so only the whole part of the value will be displayed. The output produced is as follows.

As a decimal, the price is 5



**Decimals** The `%d` format string represents a decimal integer, with *decimal* referring to base 10 numbers and not decimal points. There are different format specifiers to display numbers in base 16 (*hex*, `%x`), base 8 (*octal*, `%o`), and base 2 (*binary*, `%b`).

## Format Codes

A format specifier can also include optional elements to specify the padding, alignment, width, and precision of the value to be displayed. This allows you to carry out some very powerful formatting.

The width specifier indicates how many characters the formatted value should occupy in the displayed string and appears between the percent sign and the type specifier. For instance, the following example ensures that the name displayed takes up exactly 10 characters:

```
$name1 = "Tom";
$name2 = "Dick";
$name3 = "Harry";
echo "<PRE>";
printf("%10s \n", $name1);
printf("%10s \n", $name2);
printf("%10s \n", $name3);
echo "</PRE>";
```



**Padding** These examples use `<PRE>` tags to make sure that multiple spaces used for padding are displayed onscreen. Usually a web browser will treat multiple adjacent whitespace characters as a single space.

String padding is not used very often in creating dynamic web pages. However, it is useful when you're producing plain-text output, such as generated email text, in PHP.

If you run this example through a web browser, you will see that each name displayed is indented from the left of the screen by the correct number of characters to make each name right-aligned with the others.

The default behavior is to right-align to the given width. However, you can reverse this by using the minus symbol as an alignment specifier. To left-align the strings in the previous example, you would use the format specifier `%-10s`. Although visibly this would not appear any different from simply using `%s`, the strings would be padded on the right with spaces to a length of 10 characters.

You can change the padding character from a space to any other character by placing that character before the width value, prefixed with a single quotation mark. The following example ensures that a five-digit order number is always displayed padded with zeros if necessary:

```
$order = 201;
printf("Order number: '%05d", $order);
```

The output produced is as follows:

```
Order number: 00201
```

The precision specifier is used with a floating-point number to specify the number of decimal places to display. The most common usage is with currency values, to ensure that the two cent digits always appear, even in a whole dollar amount.

The precision value follows the optional width specifier and is indicated by a period followed by the number of decimal places to display. The following example uses `%.2f` to display a currency value with no width specifier:

```
$price = 6;
printf("The price is %.2f", $price);
```

The price is correctly formatted as follows:

```
The price is 6.00
```



**Float Widths** With floats, the width specifier indicates only the width of the number before the decimal point. For example, `%6.2f` will actually be nine characters long, with the period and two decimal places.

## Using `sprintf`

The `sprintf` function is used to assign formatted strings to variables. The syntax is the same as for `printf`, but rather than being output as the result, the formatted value is returned by the function as a string.

For example, to assign a formatted price value to a new variable, you could do the following:

```
$new_price = sprintf("%.2f", $price);
```

All the format specifier rules that apply to `printf` also apply to `sprintf`.

## String Functions

Let's take a look at some of the other string functions available in PHP. The full list of string functions can be found in the online manual, at [www.php.net/manual/en/ref.strings.php](http://www.php.net/manual/en/ref.strings.php).

### Capitalization

You can switch the capitalization of a string to all uppercase or all lowercase by using `strtoupper` or `strtolower`, respectively.

The following example demonstrates the effect this has on a mixed-case string:

```
$phrase = "I love PHP";
echo strtoupper($phrase) . "
";
echo strtolower($phrase) . "
";
```

The result displayed is as follows:

```
I LOVE PHP
i love php
```

If you wanted to functions capitalize only the first character of a string, you use `ucfirst`:

```
$phrase = "welcome to the jungle";
echo ucfirst($phrase);
```

You can also capitalize the first letter of each word—which is useful for names—by using `ucwords`:

```
$phrase = "green bay packers";
echo ucwords($phrase);
```

Neither `ucfirst` nor `ucwords` affects characters in the string that are already in uppercase, so if you want to make sure that all the other characters are lowercase, you must combine these functions with `strtolower`, as in the following example:

```
$name = "CHRIS NEWMAN";
echo ucwords(strtolower($name));
```

## Dissecting a String

The `substr` function allows you to extract a substring by specifying a start position within the string and a length argument. The following example shows this in action:

```
$phrase = "I love PHP";
echo substr($phrase, 3, 5);
```

This call to `substr` returns the portion of `$phrase` from position 3 with a length of 5 characters. Note that the position value begins at zero, not one, so the actual substring displayed is `ove P`.

If the length argument is omitted, the value returned is the substring from the position given to the end of the string. The following statement produces `love PHP` for `$phrase`:

```
echo substr($phrase, 2);
```

If the position argument is negative, `substr` counts from the end of the string. For example, the following statement displays the last three characters of the string—in this case, `PHP`:

```
echo substr($phrase, -3);
```

If you need to know how long a string is, you use the `strlen` function:

```
echo strlen($phrase);
```

To find the position of a character or a string within another string, you can use `strpos`. The first argument is often known as the *haystack*, and the second as the *needle*, to indicate their relationship.

The following example displays the position of the @ character in an email address:

```
$email = "chris@lightwood.net";
echo strpos($email, "@");
```



**String Positions** Remember that the character positions in a string are numbered from the left, starting from zero. Position 1 is actually the second character in the string. When `strpos` finds a match at the beginning of the string compared, the return value is zero, but when no match is found, the return value is `FALSE`.

You must check the type of the return value to determine this difference. For instance, the condition `strpos($a, $b) === 0` holds true only when `$b` matches `$a` at the first character.

The `strstr` function extracts a portion of a string from the position at which a character or string appears up to the end of the string. This is a convenience function that saves your using a combination of `strpos` and `substr`.

The following two statements are equivalent:

```
$domain = strstr($email, "@");

$domain = strstr($email, strpos($email, "@"));
```

## Summary

In this lesson you have learned how to work with strings in PHP. In the next lesson you will examine how regular expressions are used to perform pattern matching on strings.

# LESSON 7

## Working with Arrays



*In this lesson you will learn how to use arrays in PHP to store and retrieve indexed data.*

### What Is an Array?

An *array* is a variable type that can store and index a set of values. An array is useful when the data you want to store has something in common or is logically grouped into a set.

### Creating and Accessing Arrays

Suppose you wanted to store the average temperature for each month of the year. Using single-value variables—also known as *scalar variables*—you would need 12 different variables—`$temp_jan`, `$temp_feb`, and so on—to store the values. By using an array, you can use a single variable name to group the values together and let an index key indicate which month each value refers to.

The following PHP statement declares an array called `$temps` and assigns it 12 values that represent the temperatures for January through December:

```
$temps = array(38, 40, 49, 60, 70, 79,
 84, 83, 76, 65, 54, 42);
```

The array `$temps` that is created contains 12 values that are indexed with numeric key values from 0 to 11. To reference an indexed value from an array, you suffix the variable name with the index key. To display March's temperature, for example, you would use the following:

```
echo $temps[2];
```



**Index Numbers** Because index values begin at zero by default, the value for March—the third month—is contained in the second element of the array.

The square brackets syntax can also be used to assign values to array elements. To set a new value for November, for instance, you could use the following:

```
$temps[10] = 56;
```



**The array Function** The array function is a shortcut function that quickly builds an array from a supplied list of values, rather than adding each element in turn.

If you omit the index number when assigning an array element, the next highest index number will automatically be used. Starting with an empty array `$temps`, the following code would begin to build the same array as before:

```
$temps[] = 38;
$temps[] = 40;
$temps[] = 49;
...
```

In this example, the value 38 would be assigned to `$temps[0]`, 40 to `$temps[1]`, and so on. If you want to make sure that these assignments begin with `$temps[0]`, it's a good idea to initialize the array first to make sure there is no existing data in that array. You can initialize the `$temps` array with the following command:

```
$temps = array();
```



## Outputting the Contents of an Array

PHP includes a handy function, `print_r`, that can be used to recursively output all the values stored in an array. The following script defines the array of temperature values and then displays its contents onscreen:

```
$temps = array(38, 40, 49, 60, 70, 79,
 84, 83, 76, 65, 54, 42);

print "<PRE>";
print_r($temps);
print "</PRE>";
```

The `<PRE>` tags are needed around `print_r` because the output generated is text formatted with spaces and newlines. The output from this example is as follows:

```
Array
(
 [0] => 38
 [1] => 40
 [2] => 49
 [3] => 60
 [4] => 70
 [5] => 79
 [6] => 84
 [7] => 83
 [8] => 76
 [9] => 65
 [10] => 54
 [11] => 42
)
```



**print\_r** The `print_r` function can be very useful when you're developing scripts, although you will never use it as part of a live website. If you are ever unsure about what is going on in an array, using `print_r` can often shed light on the problem very quickly.

## Looping Through an Array

You can easily replicate the way `print_r` loops through every element in an array by using a loop construct to perform another action for each value in the array.

By using a `while` loop, you can find all the index keys and their values from an array—similar to using the `print_r` function—as follows:

```
while (list($key, $value) = each($temps)) {
 echo "Key $key has value $val
";
}
```

For each element in the array, the index key value will be stored in `$key` and the value in `$value`.

PHP also provides another construct for traversing arrays in a loop, using a `foreach` construct. Whether you use a `while` or `foreach` loop is a matter of preference; you should use whichever you find easiest to read.

The `foreach` loop equivalent to the previous example is as follows:

```
foreach($temps as $key => $value) {
 ...
}
```



**Loops** You may have realized that with the `$temps` example, a `for` loop counting from 0 to 11 could also be used to find the value of every element in the array. However, although that technique would work in this situation, the keys in an array may not always be sequential and, as you will see in the next section, may not even be numeric.

## Associative Arrays

The array examples so far in this chapter have used numeric keys. An *associative* array allows you to use textual keys so that the indexes can be more descriptive.

To assign a value to an array by using an associative key and to reference that value, you simply use a textual key name enclosed in quotes, as in the following examples:

```
$temps["jan"] = 38;
echo $temps["jan"];
```

To define the complete array of average monthly temperatures in this way, you can use the array function as before, but you indicate the key value as well as each element. You use the => symbol to show the relationship between a key and its value:

```
$temps = array("jan" => 38, "feb" => 40, "mar" => 49,
 "apr" => 60, "may" => 70, "jun" => 79,
 "jul" => 84, "aug" => 83, "sep" => 76,
 "oct" => 65, "nov" => 54, "dec" => 42);
```

The elements in an associative array are stored in the order in which they are defined (you will learn about sorting arrays later in this lesson), and traversing this array in a loop will find the elements in the order defined. You can call `print_r` on the array to verify this. The first few lines of output are as follows:

```
Array
(
 [jan] => 38
 [feb] => 40
 [mar] => 49
 ...
```

## Array Functions

You have already seen the array function used to generate an array from a list of values. Now let's take a look at some of the other functions PHP provides for manipulating arrays.

There are many more array functions in PHP than this book can cover. If you need to perform a complex array operation that you have not learned about, refer to the online documentation at [www.php.net/ref.array](http://www.php.net/ref.array).

## Sorting

To sort the values in an array, you use the `sort` function or one of its derivatives, as in the following example:

```
sort($temps);
```



**Sorting Functions** `sort` and other related functions take a single array argument and sort that array. The sorted array is not returned; the return value indicates success or failure.

Sorting the original `$temps` array with `sort` arranges the values into numeric order, but the key values are also renumbered. After you perform the sort, index 0 of the array will contain the lowest value from the array, and there is no way of telling which value corresponds to each month.

You can use `asort` to sort an array while maintaining the key associations, whether it is an associative array or numerically indexed. After you sort `$temps`, index 0 will still contain January's average temperature, but if you loop through the array, the elements will be retrieved in sorted order.

Using the associative array `$temps` as an example, the following code displays the months and their average temperatures, from coldest to hottest:

```
$temps = array("jan" => 38, "feb" => 40, "mar" => 49,
 "apr" => 60, "may" => 70, "jun" => 79,
 "jul" => 84, "aug" => 83, "sep" => 76,
 "oct" => 65, "nov" => 54, "dec" => 42);
asort($temps);
foreach($temps as $month => $temp) {
 print "$month: $temp
\n";
}
```

It is also possible to sort an array on the keys rather than on the element values, by using `ksort`. Using `ksort` on the associative `$temps` array arranges the elements alphabetically on the month name keys. Therefore, when you loop through the sorted array, the first value fetched would be `$temps["apr"]`, followed by `$temps["aug"]`, and so on.

To reverse the sort order for any of these functions, you use `rsort` in place of `sort`. The reverse of `asort` is `arsort`, and the reverse of `ksort` is `krsort`. To reverse the order of an array as it stands without sorting, you simply use `array_reverse`.

## Randomizing an Array

As well as sorting the values of an array into order, PHP provides functions so that you can easily randomize elements in an array.

The `shuffle` function works in a similar way to the sorting functions: It takes a single array argument and shuffles the elements in that array into a random order. As with `sort`, the key associations are lost, and the shuffled values will always be indexed numerically.

## Set Functions

By treating an array as a set of values, you can perform set arithmetic by using PHP's array functions.

To combine the values from different arrays (a union operation), you use the `array_merge` function with two or more array arguments, as in the following example:

```
$union = array_merge($array1, $array2, $array3, ...);
```

A new array is returned that contains all the elements from the listed arrays. In this example, the `$union` array will contain all the elements in `$array1`, followed by all the elements in `$array2`, and so on.

To remove duplicate values from any array, you use `array_unique` so that if two different index keys refer to the same value, only one will be kept.

The `array_intersect` function performs an intersection on two arrays. The following example produces a new array, `$intersect`, that contains all the elements from `$array1` that are also present in `$array2`:

```
$intersect = array_intersect($array1, $array2);
```

To find the difference between two sets, you can use the `array_diff` function. The following example returns the array `$diff`, which contains only elements from `$array1` that are not present in `$array2`:

```
$diff = array_diff($array1, $array2);
```

## Looking Inside Arrays

The `count` function returns the number of elements in an array. It takes a single array argument. For example, the following statement shows that there are 12 values in the `$temps` array:

```
echo count($temps);
```

To find out whether a value exists within an array without having to write a loop to search through every value, you can use `in_array` or `array_search`. The first argument is the value to search for, and the second is the array to look inside:

```
if (in_array("PHP", $languages)) {
 ...
}
```

The difference between these functions is the return value. If the value exists within the array, `array_search` returns the corresponding key, whereas `in_array` returns only a Boolean result.



**Needle in a Haystack** Somewhat confusingly, the order of the *needle* and *haystack* arguments to `in_array` and `array_search` is opposite that of string functions, such as `strpos` and `strstr`.

To check whether a particular key exists in an array, you use `array_key_exists`. The following example determines whether the December value of `$temps` has been set:

```
if (array_key_exists("dec", $temps)) {
 ...
}
```

## Serializing

The `serialize` function creates a textual representation of the data an array holds. This is a powerful feature that gives you the ability to easily write the contents of a PHP array to a database or file.

Lessons 17, “Filesystem Access,” and 19, “Using a MySQL Database,” deal with the specifics of filesystem and database storage. For now let’s just take a look at how serialization of an array works.

Calling `serialize` with an array argument returns a string that represents the keys and values in that array, in a structured format. You can then decode that string by using the `unserialize` function to return the original array.

The serialized string that represents the associative array `$temps` is as follows:

```
a:12:{s:3:"jan";i:38;s:3:"feb";i:40;s:3:"mar";i:49;
s:3:"apr";i:60; s:3:"may";i:70;s:3:"jun";
i:79;s:3:"jul";i:84;s:3:"aug";i:83;s:3:"sep";
si:76;s:3:"oct";i:65;s:3:"nov";i:54;s:3:"dec";i:42;}
```

You can probably figure out how this string is structured, and the only argument you would ever pass to `unserialize` is the result of a `serialize` operation—there is no point in trying to construct it yourself.

## Multidimensional Arrays

It is possible—and often very useful—to use arrays to store two-dimensional or even multidimensional data.

### Accessing Two-Dimensional Data

In fact, a *two-dimensional array* is an array of arrays. Suppose you were to use an array to store the average monthly temperature, by year, using two key dimensions—the month and the year. You might display the average temperature from February 1995 as follows:

```
echo $temps[1995]["feb"];
```

Because `$temps` is an array of arrays, `$temps[1995]` is an array of temperatures, indexed by month, and you can reference its elements by adding the key name in square brackets.

## Defining a Multidimensional Array

Defining a multidimensional array is fairly straightforward, as long as you remember that what you are working with is actually an array that contains more arrays.

You can initialize values by using references to the individual elements, as follows:

```
$temps[1995]["feb"] = 41;
```

You can also define multidimensional arrays by nesting the array function in the appropriate places. The following example defines the first few months for three years (the full array would clearly be much larger than this):

```
$temps = array (
 1995 => array ("jan" => 36, "feb" => 42, "mar" => 51),
 1996 => array ("jan" => 37, "feb" => 42, "mar" => 49),
 1997 => array ("jan" => 34, "feb" => 40, "mar" => 50));
```

The `print_r` function can follow as many dimensions as an array contains, and the formatted output will be indented to make each level of the hierarchy readable. The following is the output from the three-dimensional `$temps` array just defined:

```
Array
(
 [1995] => Array
 (
 [jan] => 36
 [feb] => 42
 [mar] => 51
)
 [1996] => Array
 (
 [jan] => 37
 [feb] => 42
```



```
 [mar] => 49
)
[1997] => Array
(
 [jan] => 34
 [feb] => 40
 [mar] => 50
)
)
```

## Summary

In this lesson you have learned how to create arrays of data and manipulate them. The next lesson examines how regular expressions are used to perform pattern matching on strings.



## LESSON 8

# Regular Expressions

*In this lesson you will learn about advanced string manipulation using regular expressions. You will see how to use regular expressions to validate a string and to perform a search-and-replace operation.*

## Introducing Regular Expressions

Using regular expressions—sometimes known as *regex*—is a powerful and concise way of writing a rule that identifies a particular string format. Because they can express quite complex rules in only a few characters, if you have not come across them before, regular expressions can look very confusing indeed.

At its very simplest, a regular expression can be just a character string, where the expression matches any string that contains those characters in sequence. At a more advanced level, a regular expression can identify detailed patterns of characters within a string and break a string into components based on those patterns.

## Types of Regular Expression

PHP supports two different types of regular expressions: the POSIX-extended syntax—which is examined in this lesson—and the Perl-Compatible Regular Expression (PCRE). Both types perform the same function, using a different syntax, and there is really no need to know how to use both types. If you are already familiar with Perl, you may find it easier to use the PCRE functions than to learn the POSIX syntax.

Documentation for PCRE can be found online at [www.php.net/manual/en/ref.pcre.php](http://www.php.net/manual/en/ref.pcre.php).

## Using ereg

The `ereg` function in PHP is used to test a string against a regular expression. Using a very simple regex, the following example checks whether `$phrase` contains the substring `PHP`:

```
$phrase = "I love PHP";
if (ereg("PHP", $phrase)) {
 echo "The expression matches";
}
```

If you run this script through your web browser, you will see that the expression does indeed match `$phrase`.

Regular expressions are case-sensitive, so if the expression were in lower-case, this example would not find a match. To perform a non-case-sensitive regex comparison, you can use `eregi`:

```
if (eregi("php", $phrase)) {
 echo "The expression matches";
}
```



**Performance** The regular expressions you have seen so far perform basic string matching that can also be performed by the functions you learned about in Lesson 6, “Working with Strings,” such as `strstr`. In general, a script will perform better if you use string functions in place of `ereg` for simple string comparisons.

## Testing Sets of Characters

As well as checking that a sequence of characters appears in a string, you can test for a set of characters by enclosing them in square brackets. You simply list all the characters you want to test, and the expression matches if any one of them occurs.

The following example is actually equivalent to the use of `eregi` shown earlier in this lesson:

```
if (ereg("[Pp][Hh][Pp]", $phrase)) {
 echo "The expression matches";
}
```

This expression checks for either an uppercase or lowercase *P*, followed by an uppercase or lowercase *H*, followed by an uppercase or lowercase *P*.

You can also specify a range of characters by using a hyphen between two letters or numbers. For example, `[A-Z]` would match any uppercase letter, and `[0-4]` would match any number between zero and four.

The following condition is true only if `$phrase` contains at least one uppercase letter:

```
if (ereg("[A-Z]", $phrase)) ...
```

The `^` symbol can be used to negate a set so that the regular expression specifies that the string must not contain a set of characters. The following condition is true only if `$phrase` contains at least one non-numeric character:

```
if (ereg("[^0-9]", $phrase)) ...
```

## Common Character Classes

You can use a number of sets of characters when using regex. To test for all alphanumeric characters, you would need a regular expression that looks like this:

```
[A-Za-z0-9]
```

The character class that represents the same set of characters can be represented in a much clearer fashion:

```
[[:alnum:]]
```

The `[` and `]` characters indicate that the expression contains the name of a character class. The available classes are shown in Table 8.1.

**TABLE 8.1** Character Classes for Use in Regular Expressions

Class Name	Description
alnum	All alphanumeric characters, A–Z, a–z, and 0–9
alpha	All letters, A–Z and a–z
digit	All digits, 0–9
lower	All lowercase characters, a–z
print	All printable characters, including space
punct	All punctuation characters—any printable character that is not a space or alnum
space	All whitespace characters, including tabs and new-lines
upper	All uppercase letters, A–Z

## Testing for Position

All the expressions you have seen so far find a match if that expression appears anywhere within the compared string. You can also test for position within a string in a regular expression.

The `^` character, when not part of a character class, indicates the start of the string, and `$` indicates the end of the string. You could use the following conditions to check whether `$phrase` begins or ends with an alphabetic character, respectively:

```
if (ereg("^[a-z]", $phrase)) ...
```

```
if (ereg("[a-z]$", $phrase)) ...
```

If you want to check that a string contains only a particular pattern, you can sandwich that pattern between `^` and `$`. For example, the following condition checks that `$number` contains only a single numeric digit:

```
if (ereg("^[:digit:]+$", $number)) ...
```



**The Dollar Sign** If you want to look for a literal \$ character in a regular expression, you must delimit the character as `\$` so that it is not treated as the end-of-line indicator.

When your expression is in double quotes, you must use `\\$` to double-delimit the character; otherwise, the \$ sign may be interpreted as the start of a variable identifier.

## Wildcard Matching

The dot or period (.) character in a regular expression is a wildcard—it matches any character at all. For example, the following condition matches any four-letter word that contains a double o:

```
if (ereg("^..oo.$", $word)) ...
```

The ^ and \$ characters indicate the start and end of the string, and each dot can be any character. This expression would match the words *book* and *tool*, but not *buck* or *stool*.



**Wildcards** A regular expression that simply contains a dot matches any string that contains at least one character. You must use the ^ and \$ characters to indicate length limits on the expression.

## Repeating Patterns

You have now seen how to test for a particular character or for a set or class of characters within a string, as well as how to use the wildcard character to define a wide range of patterns in a regular expression. Along with these, you can use another set of characters to indicate where a pattern can or must be repeated a number of times within a string.

You can use an asterisk (\*) to indicate that the preceding item can appear zero or more times in the string, and you can use a plus (+) symbol to ensure that the item appears at least once.

The following examples, which use the \* and + characters, are very similar to one another. They both match a string of any length that contains only alphanumeric characters. However, the first condition also matches an empty string because the asterisk denotes zero or more occurrences of `[[:alnum:]]`:

```
if (ereg("^[[[:alnum:]]*$", $phrase)) ...
```

```
if (ereg("^[[[:alnum:]]+$", $phrase)) ...
```

To denote a group of matching characters that should repeat, you use parentheses around them. For example, the following condition matches a string of any even length that contains alternating letters and numbers:

```
if (ereg("^[[:alpha:]][[:digit:]]+$", $string)) ...
```

This example uses the plus symbol to indicate that the letter/number sequence could repeat one or more times. To specify a fixed number of times to repeat, the number can be given in braces. A single number or a comma-separated range can be given, as in the following example:

```
if (ereg("^[[:alpha:]][[:digit:]]{2,3}$", $string)) ...
```

This expression would match four or six character strings that contain alternating letters and numbers. However, a single letter and number or a longer combination would not match.

The question mark (?) character indicates that the preceding item may appear either once or not at all. The same behavior could be achieved by using `{0,1}` to specify the number of times to repeat a pattern.

## Some Practical Examples

You use regex mostly to validate user input in scripts, to make sure that a value entered is acceptable. The following are some practical examples of using regular expressions.

## Zip Codes

If you have a customer's zip code stored in `$zip`, you might want to check that it has a valid format. A U.S. zip code always consists of five numeric digits, and it can optionally be followed by a hyphen and four more digits. The following condition validates a zip code in this format:

```
if (ereg("^[[:digit:]]{5}(-[[:digit:]]{4})?$", $zip)) ...
```

The first part of this regular expression ensures that `$zip` begins with five numeric digits. The second part is in parentheses and followed by a question mark, indicating that this part is optional. The second part is defined as a hyphen character followed by four digits.

Regardless of whether the second part appears, the `$` symbol indicates the end of the string, so there can be no other characters other than those allowed by the expression if this condition is to be satisfied. Therefore, this condition matches a zip code that looks like either 90210 or 90210-1234.

## Telephone Numbers

You might want to enforce the format of a telephone number to ensure that it looks like (555)555-5555. There are no optional parts to this format. However, because the parentheses characters have a special meaning for regex, they have to be escaped with a backslash.

The following condition validates a telephone number in this format:

```
if (ereg("^([[:digit:]]{3}\)[[:digit:]]{3}-[[:digit:]]{4}$",
 $telephone)) ...
```

## Email Addresses

You need to consider many variables when validating an email address. At the very simplest level, an email address for a .com domain name looks like `somename@somedomain.com`.

However, there are many variations, including top-level domain names that are two characters, such as .ca, or four characters, such as .info.



Some country-specific domains have a two-part extension, such as .co.uk or .com.au.

As you can see, a regular expression rule to validate an email address needs to be quite forgiving. However, by making some general assumptions about the format of an email address, you can still create a rule that rejects many badly formed addresses.

There are two main parts to an email address, and they are separated by an @ symbol. The characters that can appear to the left of the @ symbol—usually the recipient's mailbox name—can be alphanumeric and can contain certain symbols.

Let's assume that the mailbox part of an email address can consist of any characters except for the @ symbol itself and can be any length. Rather than try to list all the acceptable characters you can think of—for instance, should you allow an apostrophe in an email address?—it is usually good enough to enforce that email address can contain only one @ character and that anything up to that character is a valid mailbox name.

For the regex rule, you can define that the domain part of an email address consists of two or more parts, separated by dots. You can also assume that the last part may only be between two and four characters in length, which is sufficient for all top-level domain names currently in use.

The set of characters that can be used in parts of the domain is more restrictive than the mailbox name—only lowercase alphanumeric characters and a hyphen can be used.

Taking these assumptions into consideration, you can come up with the following condition to test the validity of an email address:

```
if (ereg("^[^@]+@([a-z0-9\.-]+\.[a-z]{2,4})$", $email)) ...
```

This regular expression breaks down as follows: any number of characters followed by an @ symbol, followed by one or more parts consisting of only lowercase letters, numbers, or a hyphen. Each of those parts ends with a dot, and the final part must be between two and four letters in length.



**How Far to Go** This expression could be even further refined. For instance, a domain name cannot begin with a hyphen and has a maximum length of 63 characters. However, for the purpose of catching mistyped email addresses, this expression is more than sufficient.

## Breaking a String into Components

You have used parentheses to group together parts of a regular expression to indicate a repeating pattern. You can also use parentheses to indicate subparts of an expression, and `ereg` allows you to break a pattern into components based on the parentheses.

When an optional third argument is passed to `ereg`, that variable is assigned an array of values that correspond to the parts of the pattern identified by the parentheses in the regular expression.

Let's use the email address regular expression as an example. The following code includes three sets of parentheses to isolate the mailbox name, domain name (apart from the extension), and domain extension:

```
$email = "chris@lightwood.net";
if (ereg("^[^@]+)@([a-z\.-]+\.[a-z]{2,4})$",
 $email, $match)) {
 echo "Mailbox: " . $match[1] . "
";
 echo "Domain name: " . $match[2] . "
";
 echo "Domain type: " . $match[3] . "
";
}
else {
 echo "Email address is invalid";
}
```

If you run this script in a web browser, you get output similar to the following:

```
Mailbox: chris
Domain name: lightwood.
Domain type: net
```

Note that the first key of `$match` refers to the first pattern found. The array keys are numbered from zero, as usual; however, `$match[0]` contains the entire matched pattern.

## Searching and Replacing

You can use regular expressions to perform search and replace operations on a string with the `ereg_replace` function. Its three arguments are a regex search pattern, the replacement string, and the string to replace into. The modified string is returned.



**str\_replace** If you want to perform a simple string replace operation that does not require a regular expression, you can use `str_replace` instead of `ereg_replace`. `str_replace` is more efficient because PHP does not even have to consider that you might be looking for a regular expression.

For example, to blank out a telephone number before displaying a string, you could use the following:

```
echo ereg_replace(
 "\([[:digit:]]{3}\)[[:digit:]]{3}-[[:digit:]]{4}$",
 "(XXX)XXX-XXXX", $string);
```

Just like you can use `eregi` in place of `ereg`, to perform a non-case-sensitive search and replace using regex, you can use `eregi_replace`.

## Summary

In this lesson you have learned the basics of regular expressions. If you want to find out more, you can refer to *Sams Teach Yourself Regular Expressions in 10 Minutes* by Ben Forta.

In the next lesson you will learn how to handle date and time values in PHP.



## LESSON 9

# Working with Dates and Times

*In this lesson you will learn how to store, display, and manipulate date and time values in PHP.*

## Date Formats

PHP does not have a native date data type, so in order to store date values in a script, you must first decide on the best way to store these values.

### Do-It-Yourself Date Formats

Although you often see dates written in a structured format, such as 05/03/1974 or 2001-12-31, these are not ideal formats for working with date values. However, the latter of these two is more suitable than the first because the order of its components is from most significant (the year) to the least significant (the day), so values can be compared using the usual PHP operators.

As a string, `2002-01-01` is greater than `2001-12-31`, but because comparisons are performed more efficiently on numbers than on strings, this could be written better as just `20020201`, where the format is `YYYYMMDD`. This format can be extended to include a time portion—again, with the most significant elements first—as `YYYYMMDDHHMMSS`, for example.

However, date arithmetic with this format is nearly impossible. While you can add one to `20040501`, for instance, and find the next day in that month, simply adding one to `20030531` would result in a nonsense date of May 32.

## Unix Timestamp Format

The Unix timestamp format is an integer representation of a date and time. It is a value that counts the number of seconds since midnight on January 1, 1970.



**The Unix Epoch** A timestamp with integer value zero represents precisely midnight, Greenwich Mean Time (GMT), on January 1, 1970. This date is known as the *Unix Epoch*.

Right now, we have a 10-digit date and time timestamp. To find the current timestamp, you use the `time` function:

```
echo time();
```

The Unix timestamp format is useful because it is very easy to perform calculations on because you know that the value always represents a number of seconds. For example, you can just add 3,600 to a timestamp value to increase the time by one hour or add 86,400 to add one day—because there are 3,600 seconds in an hour and 86,400 seconds in a day.

One drawback, however, is that the Unix timestamp format cannot handle dates prior to 1970. Although some systems may be able to use a negative timestamp value to count backward from the Epoch, this behavior cannot be relied on.

Timestamps are good for representing contemporary date values, but they may not always be suitable for handling dates of birth or dates of historical significance. You should consider what values you will be working with when deciding whether a timestamp is the correct format to use.



**Timestamp Limitations** The maximum value of a Unix timestamp depends on the system's architecture. Most systems use a 32-bit integer to store a timestamp, making the latest time it can represent 3:14am on January 19, 2038.

# Working with Timestamps

There are times when using your own date format is beneficial, but in most cases a timestamp is the best choice. Let's look at how PHP interacts with the timestamp date format.

## Formatting Dates

In Lesson 1, "Getting to Know PHP," you used the `date` function to display the current date by passing a format string as the argument, such as in the following example:

```
echo date("j F Y H:i:s");
```

The date displayed looks something like this:

```
12 November 2004 10:23:55
```

The optional second argument to `date` is a timestamp value of the date that you want to display. For example, to display the date when a timestamp first requires a 10-digit number, you could use the following:

```
echo date("j F Y H:I:s", 1000000000);
```

The list of format codes for the `date` function is shown in Table 9.1.

**TABLE 9.1** Format Codes for `date`

Code	Description
a	Lowercase am or pm
A	Uppercase AM or PM
d	Two-digit day of month, 01–31
D	Three-letter day name, Mon–Sun
F	Full month name, January–December
g	12-hour hour with no leading zero, 1–12
G	24-hour hour with no leading zero, 0–23
h	12-hour hour with leading zero, 01–12

Code	Description
H	24-hour hour with leading zero, 00–23
I	Minutes with leading zero, 00–59
j	Day of month with no leading zero, 1–31
l	Full day name, Monday–Sunday
m	Month number with leading zeros, 01–12
M	Three letter month name, Jan–Dec
n	Month number with no leading zeros, 1–12
s	Seconds with leading zero, 00–59
S	Ordinal suffix for day of month, st, nd, rd, or th
w	Number of day of week, 0–6, where 0 is Sunday
W	Week number, 0–53
y	Two-digit year number
Y	Four-digit year number
z	Day of year, 0–365

## Creating Timestamps

Don't worry; you don't have to count from January 1, 1970, each time you want to calculate a timestamp. The PHP function `mktime` returns a timestamp based on given date and time values.

The arguments, in order, are the hour, minute, second, month, day, and year. The following example would assign `$timestamp` the timestamp value for 8 a.m. on December 25, 2001:

```
$timestamp = mktime(8, 0, 0, 12, 25, 2001);
```

The Unix timestamp format counts from January 1, 1970, at midnight GMT. The `mktime` function returns a timestamp relative to the time zone

in which your system operates. For instance, `mktime` would return a timestamp value 3,600 higher when running on a web server in Texas than on a machine in New York with the same arguments.



**Daylight Saving Time** If you are only concerned with the date part of a timestamp, the first three arguments to `mktime` only matter if they are close to midnight at a time of the year when daylight saving time is a factor.

For instance, when the clocks are moved back one hour, that day is only 23 hours long. Adding 86,400 seconds to a timestamp that represents midnight on that day will actually move the day part of the timestamp forward two days. You can use `midday` instead of `midnight` as the time element to avoid these issues.



**Greenwich Mean Time** To obtain timestamp values that are always relative to GMT—the time in London when there is no daylight saving time adjustment—you use `gmkmtime` instead of `mktime`.

The `mktime` function is forgiving if you supply it with nonsense arguments, such as a day of the month that doesn't exist. For instance, if you try to calculate a timestamp for February 29 in a non-leap year, the value returned will actually represent March 1, as the following statement confirms:

```
echo date("d/m/Y", mktime(12, 0, 0, 2, 29, 2003));
```

You can exploit this behavior as a way of performing date and time arithmetic. Consider the following example, which calculates and displays the date and time 37 hours after midday on December 30, 2001:

```
$time = mktime(12 + 37, 0, 0, 12, 30, 2001);
echo date("d/m/Y H:i:s", $time);
```



By simply adding a constant to one of the arguments in `mktime`, you can shift the timestamp value returned by that amount. The date and time display as follows:

```
01/01/2002 01:00:00
```

The value returned in this example has correctly shifted the day, month, year, and hour values, taking into account the number of days in December and that December is the last month of the year.

## Converting Other Date Formats to Timestamps

If you have a date stored in a format like `DD-MM-YYYY`, it's a fairly simple process to convert this to a timestamp by breaking up the string around the hyphen character. The `explode` function takes a delimiter argument and a string and returns an array that contains each part of the string that was separated by the given delimiter.

The following example breaks a date in this format into its components and builds a timestamp from those values:

```
$date = "03-05-1974";
$parts = explode("/", $date);
$timestamp = mktime(12, 0, 0,
 $parts[1], $parts[0], $parts[2]);
```

For many date formats, there is an even easier way to create a timestamp—using the function `strtotime`. The following examples all display the same valid timestamp from a string date value:

```
$timestamp = strtotime("3 May 04");
$timestamp = strtotime("3rd May 2004");
$timestamp = strtotime("May 3, 2004");
$timestamp = strtotime("3-may-04");
$timestamp = strtotime("2004-05-03");
$timestamp = strtotime("05/03/2004");
```

Note that in the last examples, the date format given is `MM/DD/YYYY`, not `DD/MM/YYYY`. You can find the complete list of formats that are acceptable to `strtotime` at [www.gnu.org/software/tar/manual/html\\_chapter/tar\\_7.html](http://www.gnu.org/software/tar/manual/html_chapter/tar_7.html).

## Getting Information About a Timestamp

You can use the date function to return part or all of the date that a timestamp represents as a formatted string, but PHP also provides the `getdate` function, which returns useful values from a timestamp.

Taking a single timestamp argument, `getdate` returns an associative array that contains the indexes shown in Table 9.2.

**TABLE 9.2** Key Elements Returned by `getdate`

Key	Description
seconds	Seconds, 0–59
minutes	Minutes, 0–59
hours	Hours, 0–23
mday	Day of the month, 0–31
wday	Day of the week, 0–6, where 0 is Sunday
yday	Day of the year, 0–365
mon	Month number, 0–12
year	Four-digit year number
weekday	Full day name, Sunday–Saturday
month	Full month name, January–December

The following example uses `getdate` to determine whether the current date falls on a weekday or weekend:

```
$now = getdate();
switch ($now[wday]) {
 case 0: // Sunday
 case 6: // Saturday
 echo "It's the weekend";
 break;
 default: echo "It's a weekday";
}
```

Note that when `getdate` is called without a timestamp argument, it returns an array that contains the elements in Table 9.2 for the current time.

## Summary

In this lesson you have learned how to store and manipulate date and time values in PHP. In the next lesson you will learn about classes in PHP, and you will discover how to use third-party library classes that you download.



## LESSON 10

# Using Classes

*In this lesson you will learn the basics of object-oriented PHP. You will see how a class is defined and how you can access methods and properties from third-party classes.*

## Object-Oriented PHP

PHP can, if you want, be written in an object-oriented (OO) fashion. In PHP5, the OO functionality of the language has been enhanced considerably.

If you are familiar with other OO languages, such as C++ or Java, you may prefer the OO approach to programming PHP, whereas if you are used to other procedural languages, you may not want to use objects at all. There are, after all, many ways to solve the same problem.

If you are new to programming as well as to PHP, you probably have no strong feelings either way just yet. It's certainly true that OO concepts are easier to grasp if you have no programming experience at all than if you have a background in a procedural language, but even so OO methods are not something that can be taught in a ten-minute lesson in this book!

The aim of this lesson is to introduce how a class is created and referenced in PHP so that if you have a preference for using objects, you can begin to develop scripts by using OO methods. Most importantly, however, you will be able to pick up and use some of the many freely available third-party class libraries that are available for PHP from resources such as those at [www.phpclasses.org](http://www.phpclasses.org), and those that are part of PEAR, which you will learn about in Lesson 25, "Using PEAR."

# What Is a Class?

A *class* is the template structure that defines an object. It can contain *functions*—also known as *class methods*—and *variables*—also known as *class properties* or *attributes*.

Each class consists of a set of PHP statements that define how to perform a task or set of tasks that you want to repeat frequently. The class can contain *private* methods, which are only used internally to perform the class's functions, and *public* methods, which you can use to interface with the class.

A good class hides its inner workings and includes only the public methods that are required to provide a simple interface to its functionality. If you bundle complex blocks of programming into a class, any script that uses that class does not need to worry about exactly how a particular operation is performed. All that is required is knowledge of the class's public methods.

Because there are many freely available third-party classes for PHP, in many situations, you need not waste time implementing a feature in PHP that is already freely available.

## When to Use Classes

At first, there may not appear to be any real advantage in using a class over using functions that have been modularized into an include file. OO is not necessarily a better approach to programming; rather, it is a different way of thinking. Whether you choose to develop your own classes is a matter of preference.

One of the advantages of OO programming is that it can allow your code to scale into very large projects easily. In OO programming, a class can inherit the properties of another and extend it; this means that functionality that has already been developed can be reused and adapted to fit a particular situation. This is called *inheritance*, and it is a key feature of OO development.

When you have completed this book, if you are interested in learning more about OO programming, take a look at *Sams Teach Yourself Object-Oriented Programming in 21 Days* by Anthony Sintes.

## What a Class Looks Like

A class is a grouping of various functions and variables—and that is exactly how it looks when written in PHP. A class definition looks very similar to a function definition; it begins with the keyword `class` and an identifier, followed by the class definition, contained in a pair of curly brackets (`{}`).

The following is a trivial example of a class to show how a class looks. This example contains just one property, `myValue`, and one method, `myMethod` (which does nothing):

```
class myClass {
 var $myValue;

 function myMethod() {
 return 0;
 }
}
```

If you are already familiar with OO programming and want to get a head start with OO PHP, you can refer to the online documentation at [www.php.net/manual/en/language.oop5.php](http://www.php.net/manual/en/language.oop5.php).

## Creating and Using Objects

To create an instance of an object from a class, you use the `new` keyword in PHP, as follows:

```
$myObject = new myClass;
```

In this example, `myClass` is the name of a class that must be defined in the script—usually in an include file—and `$myObject` becomes a `myClass` object.



**Multiple Objects** You can use the same class many times in the same script by simply creating new instances from that class but with new object names.

## Methods and Properties

The methods and properties defined in `myClass` can be referenced for `$myObject`. The following are generic examples:

```
$myObject->myValue = "555-1234";
$myObject->myMethod();
```

The arrow symbol (`->`)—made up of a hyphen and greater-than symbol—indicates a method or property of the given object. To reference the current object within the class definition, you use the special name `$this`.

The following example defines `myClass` with a method that references one of the object properties:

```
class myClass {
 var $myValue = "Jelly";

 function myMethod() {
 echo "myValue is " . $this->myValue . "
";
 }
}

$myObject = new myClass;
$myObject->myMethod();
$myObject->myValue = "Custard";
$myObject->myMethod();
```

This example makes two separate calls to `myMethod`. The first time it displays the default value of `myValue`; an assignment within the class specifies a default value for a property. The second call comes after that property has had a new value assigned. The class uses `$this` to reference its own property and does not care, or even know, that in the script its name is `$myObject`.

If the class includes a special method known as a *constructor*, arguments can be supplied in parentheses when an object is created, and those values

are later passed to the constructor function. This is usually done to initialize a set of properties for each object instance, and it looks similar to the following:

```
$myObject = new myClass($var1, $var2);
```

## Using a Third-Party Class

The best way to learn how to work with classes is to use one. Let's take a look at a popular third-party class written by Manuel Lemos, which provides a comprehensive way to validate email addresses. You can download this class from [www.phpclasses.org/browse/file/28.html](http://www.phpclasses.org/browse/file/28.html) and save the file locally as `email_validation.php`.

Manuel's class validates an email address not only by checking that its format is correct but also by performing a domain name lookup to ensure that it can be delivered. It even connects to the remote mail server to make sure the given mailbox actually exists.



**Domain Lookups** If you are following this example on a Windows-based web server, you need to download an additional file, `getmxrr.php`, to add a suitable domain name lookup function to PHP. You can download this file from [www.phpclasses.org/browse/file/2080.html](http://www.phpclasses.org/browse/file/2080.html).

The `email_validation.php` script defines a class called `email_validation_class`, so you first need to create a new instance of a validator object called `$validator`, as follows:

```
$validator = new email_validation_class;
```

You can set a number of properties for your new class. Some are required in order for the class to work properly, and others allow you to change the default behavior.

Each object instance requires you to set the properties that contain the mailbox and domain parts of a real email address, which is the address



that will be given to the remote mail server when checking a mailbox. There are no default values for these properties; they always have to be set as follows:

```
$validator->localuser = "chris";
$validator->localhost = "lightwood.net";
```

The optional `timeout` property defines how many seconds to wait when connected to a remote mail server before giving up. Setting the `debug` property causes the text of the communication with the remote server to be displayed onscreen. You never need to do this, though, unless you are interested in what is going on. The following statements define a timeout of 10 seconds and turn on debug output:

```
$validator->timeout = 10;
$validator->debug = TRUE;
```

The full list of adjustable properties for a validator object is shown in Table 10.1.

**TABLE 10.1** Properties of an `email_validation_class` Object

Property	Description
<code>timeout</code>	Indicates the number of seconds before timing out when connecting to a destination mail server
<code>data_timeout</code>	Indicates the number of seconds before timing out while data is exchanged with the mail server; if zero, takes the value of <code>timeout</code>
<code>localuser</code>	Indicates the user part of the email address of the sending user
<code>localhost</code>	Indicates the domain part of the email address of the sending user
<code>debug</code>	Indicates whether to output the text of the communication with the mail server
<code>html_debug</code>	Indicates whether the debug output should be formatted as an HTML page

The methods in `email_validation_class` are mostly private; you cannot call them directly, but the internal code is made up of a set of functions. If you examine `email_validation.php`, you will see function definitions, including `Tokenize`, `GetLine`, and `VerifyResultLines`, but none of these are useful outside the object itself.

The only public method in a validator object is named `ValidateEmailBox`, and when called, it initiates the email address validation of a string argument. The following example shows how `ValidateEmailBox` is called:

```
$email = "chris@datasnake.co.uk";
if ($validator->ValidateEmailBox($email)) {
 echo "$email is a valid email address";
}
else {
 echo "$email could not be validated";
}
```

The return value from `ValidateEmailBox` indicates whether the validation check is successful. If you have turned on the debug attribute, you will also see output similar to the following, in addition to the output from the script:

```
Resolving host name "mail.datasnake.co.uk"...
Connecting to host address "217.158.68.125"...
Connected.
S 220 mail.datasnake.co.uk ESMTP
C HELO lightwood.net
S 250 mail.datasnake.co.uk
C MAIL FROM: <chris@lightwood.net>
S 250 ok
C RCPT TO: <chris@datasnake.co.uk>
S 250 ok
C DATA
S 354 go ahead
This host states that the address is valid.
Disconnected.
```

## Summary

In this lesson you have learned about OO PHP and seen how to use classes in your own scripts. In the next lesson you will learn how PHP can interact with HTML forms.

# LESSON 11

## Processing HTML Forms



*The reason that PHP came into existence was to provide a simple way of processing user-submitted data in HTML forms. In this lesson you will learn how data entered in each type of form input is made available in a PHP script.*

### Submitting a Form to PHP

In case you are not familiar with HTML forms at all, let's begin by looking over what is involved in creating a web page that can collect information from a user and submit it to a web script.

#### The <FORM> Tag

The HTML <FORM> tag indicates an area of a web page that, when it contains text-entry fields or other form input elements, submits the values entered by a user to a particular URL.

The ACTION attribute in a <FORM> tag indicates the location of the script that the values are to be passed to. It can be a location relative to the current page or a full URL that begins with http://.

The METHOD attribute indicates the way in which the user's web browser will bundle up the data to be sent. Two methods, GET and POST, vary visibly only slightly. Form data submitted using the GET method is tagged on to the end of the URL, whereas the POST method sends the data to the web server without its being visible.



**The GET Method** You have probably seen URLs with GET method data attached, even if you didn't know that's what was going on. If you've ever use the search box at a website and the page address has come back with `?search=yourword`, it submitted the form by using the GET method.

In most situations where you are using an HTML form, the POST method is preferable. It is not only better aesthetically—because the submitted values are not revealed in the script URL—but there is no limit on the amount of data that can be submitted in this way. The amount of data that can be submitted by using the GET method is limited by the maximum URL length that a web browser can handle (the limit in Internet Explorer is 2,048 characters) and the HTTP version on the server (HTTP/1.0 must allow at least 256 characters, whereas HTTP/1.1 must allow at least 2,048).

## The <INPUT> Tag

The <INPUT> tag is used to add one of several types of form input to a web page. The type of input item is specified in the TYPE attribute, and the simplest type is a TEXT input item.

To create a TEXT input item that is suitable for entering a user's email address, you could use the following HTML:

```
<INPUT TYPE="TEXT" NAME="name" SIZE="30" VALUE="">
```

In this HTML, you supply an empty VALUE attribute because you do not want to supply a default value for the input; however, the VALUE attribute can be omitted.



**Field Lengths** The display size of a field does not affect how PHP handles the submitted values. This input has a display size of 30 characters but no MAXLENGTH attribute is set, so users with unusually long names can still type beyond the length of the field.

The CHECKBOX input type creates an input item that has only two possible values: on and off. Check boxes are useful for true/false values, and you could use the following HTML to create a check box with which the user could indicate whether he minds us contacting him by email:

```
<INPUT TYPE="CHECKBOX" NAME="may_contact" VALUE="Y" CHECKED>
```

In this case, the CHECKED attribute indicates that the check box should be checked automatically when the page loads.

The RADIO type is similar to a check box, but instead of a true/false value, a radio button group can contain several values, of which only one can be selected at a time.

To create a radio button group that can be used to gather the user's gender, you could use the following:

```
<INPUT TYPE="radio" NAME="gender" VALUE="m"> Male
<INPUT TYPE="radio" NAME="gender" VALUE="f"> Female
```



**Naming Radio Buttons** The NAME attribute determines the grouping of radio buttons. Only one selection can be made for each radio button group, although you can have several radio button groups on a page if you want. In this example, both buttons have the same name, gender.

To indicate that one of the buttons in a radio button group should be pre-selected, you can use the CHECKED attribute. For instance, if you are creating a website that will appeal primarily to women, you can pre-select the female option, as follows:

```
<INPUT TYPE="radio" NAME="gender" VALUE="m"> Male
<INPUT TYPE="radio" NAME="gender" VALUE="f" CHECKED> Female
```

The final input type that you will learn about is the SUBMIT button. This is the button you click to send the contents of a form to the script specified in the form's METHOD attribute. The label on the button is specified in the

VALUE attribute, so the following HTML would create a submit button labeled Send comments:

```
<INPUT TYPE=SUBMIT VALUE="Send comments">
```

A submit button can also have a NAME attribute, although this is rarely used. You will see later in this lesson how this affects the values sent to PHP.

## The <TEXTAREA> Tag

The <TEXTAREA> tag is used to create a multiple-line text input item. In many respects, it behaves just like a TEXT type input tag, but the way it is formed in HTML is slightly different.

Because the initial value in a text area could span many lines, it is not given in a VALUE attribute. Instead, the starting value appears between a pair of tags, as follows:

```
<TEXTAREA ROWS=4 COLS=50 NAME="comments">
Enter your comments here
</TEXTAREA>
```

PHP is not concerned with what type of input a value comes from; the difference between a text area and text input is an HTML issue only.

## The <SELECT> Tag

The final form item we will look at is the <SELECT> item, correctly known as a *menu* but more commonly called a *drop-down list*.

The most common use of a menu is to prompt for a single selection from a predefined list of values. The following example builds a drop-down list of possible places that visitors may have heard about your website:

```
<SELECT NAME="referrer">
<OPTION VALUE="search">Internet Search Engine</OPTION>
<OPTION VALUE="tv">TV Advertisement</OPTION>
<OPTION VALUE="billboard">Billboard</OPTION>
<OPTION SELECTED VALUE="other">Other</OPTION>
</SELECT>
```

In this case, the `SELECTED` attribute makes “Other” the default selection, even though it appears at the top of the list. If no item has the `SELECTED` attribute, the first option in the list is selected by default.

## Putting It All Together

By putting all these form elements together and adding some label text and a little formatting, you can create a simple comments submission form that you can then process in PHP, as shown in Listing 11.1.

### **LISTING 11.1** A Web Form for Submitting User Comments

---

```
<FORM ACTION="send_comments.php" METHOD=POST>
<TABLE>
<TR>
 <TD>Your name:</TD>
 <TD><INPUT TYPE="TEXT" NAME="name" SIZE=30></TD>
</TR>
<TR>
 <TD>Your email:</TD>
 <TD><INPUT TYPE="TEXT" NAME="email" SIZE=30></TD>
</TR>
<TR>
 <TD>Your gender:</TD>
 <TD><INPUT TYPE="RADIO" NAME="gender" VALUE="m"> Male
 <INPUT TYPE="RADIO" NAME="gender" VALUE="f"> Female
 </TD>
</TR>
<TR>
 <TD>How you found us</TD>
 <TD>
 <SELECT NAME="referrer">
 <OPTION VALUE="search">Internet Search Engine</OPTION>
 <OPTION VALUE="tv">TV Advertisement</OPTION>
 <OPTION VALUE="billboard">Billboard</OPTION>
 <OPTION SELECTED VALUE="other">Other</OPTION>
 </SELECT>
 </TD>
</TR>
<TR>
 <TD>May we email you?</TD>
 <TD><INPUT TYPE="CHECKBOX" NAME="may_contact">
```

*continues*

**LISTING 11.1** Continued

---

```
 VALUE="Y" CHECKED></TD>
</TR>
<TR>
 <TD>Comments</TD>
 <TD><TEXTAREA ROWS=4 COLS=50
 NAME="comments">Enter your comments here
 </TEXTAREA></TD>
</TR>

</TABLE>

<INPUT TYPE="SUBMIT" VALUE="Send comments">
</FORM>
```

## Processing a Form with PHP

Now let's look at how each type of item in a form is handled by PHP after the submit button is clicked.

### Accessing Form Values

Form values are made available in PHP by using some special array structures. The arrays `$_GET` and `$_POST` contain values submitted using the GET and POST methods, respectively. A hybrid array, `$_REQUEST`, contains the contents of both of these arrays, as well as the values from `$_COOKIE`, which you will use in Lesson 14, "Cookies and Sessions."



**Super-globals** The system-generated arrays that have names beginning with an underscore character are known as *super-globals* because they can be referenced from anywhere in a PHP script, regardless of scope. For instance, you do not need to explicitly declare `$_POST` as global to access its elements within a function.



Accessing the values from form items is fairly intuitive: The form item names become the element keys in `$_GET` or `$_POST`, and each value in the array is the value of the corresponding element when it was submitted.

For example, the email address submitted by `comments.html` will be `$_POST["email"]`, and the comments text will be `$_POST["comments"]`.

For `CHECKBOX` and `RADIO` input types, the `VALUE` attribute determines the value seen by PHP. If the check box named `may_contact` is checked, then the array element `$_POST["may_contact"]` has the value `Y`. If it is not checked, this element simply does not exist in the array; you should use `isset` to check whether a check box is checked.



**Default Check Box Values** If you do not specify a `VALUE` attribute for a check box item, its value in PHP when checked is `on`.

The radio group `gender` causes `$_POST["gender"]` to contain the value `m` or `f`, depending on which value is selected and, as with a check box, if no value is selected, the array element does not exist.

The simplest way to see all the submitted data from a form is to use a call to `print_r` to dump out the contents of `$_POST`, as follows:

```
echo "<PRE>";
print_r($_POST);
echo "</PRE>";
```

This is a useful debugging technique if you want to see exactly what data is being passed to a script from a form. If you create `send_comments.php`, containing just these lines, the output shows you the value of each form element in turn. The following is sample output:

```
Array
(
 [name] => Chris Newman
 [email] => chris@lightwood.net
 [gender] => m
```

```
[referrer] => search
[may_contact] => Y
[comments] => This is my favorite website ever
)
```

Even the value of a submit button can be seen by PHP if the button is given a name and the button is clicked when the form is submitted. The following form has two buttons with different names, so that you can use PHP to determine which button was actually clicked:

```
<FORM ACTION="button.php" METHOD=POST>
<INPUT TYPE="SUBMIT" NAME="button1" VALUE="Button 1">
<INPUT TYPE="SUBMIT" NAME="button2" VALUE="Button 2">
</FORM>
```

In `button.php`, you could use a condition similar to the following to see which button is clicked:

```
if (isset($_POST["button1"])) {
 echo "You clicked button 1";
}
elseif (isset($_POST["button2"])) {
 echo "You clicked button 2";
}
else {
 echo "I don't know which button you clicked!";
}
```

The `VALUE` attribute of a submit button determines what label appears on the button itself, but that value is also the value that is passed to PHP when the button is clicked.



**Submit Buttons** Many modern web browsers submit a form when you press the Enter key when focused on any of the input fields. Even if there is only one submit button on a form, its value is not sent to PHP unless it is actually clicked with the mouse.

## Hidden Inputs

One other type of form input is available, and it can be used to pass values between scripts without their being visible on the web page itself.

The `HIDDEN` type takes `NAME` and `VALUE` attributes, as usual, but it simply acts a placeholder for that value.

The following hidden input is passed to the PHP script when the form is submitted, and `$_POST["secret"]` contains the value from the form:

```
<INPUT TYPE="HIDDEN" NAME="secret" VALUE="this is a secret">
```

Be aware, however, that `HIDDEN` attribute inputs are not secure for transmitting passwords and other sensitive data. Although they do not appear on the web page, if you view the page source, you can still see hidden values in the HTML code.

## Creating a Form Mail Script

The desired result from the comments form you've been working with in this lesson is to provide a way of sending user-submitted comments by email to the owner of a website. Now you'll learn how to put together a form handler script to create this component for a website.

### The `mail` Function

PHP's `mail` function sends an email message, using your system's mailer program. On Linux/Unix systems, the `sendmail` utility is used to put a message into the outbound queue. On Windows machines, it usually sends via SMTP, and the name of the relay server must be defined in `php.ini` for this to work properly. Lesson 23, "PHP Configuration," looks at configuration issues in more detail.

The three required arguments to `mail` are the recipient's email address, the message subject, and the message body. An optional fourth argument can contain additional mail headers to be sent; this is useful for setting the `From:` address or adding a `Cc:` recipient.

The script `send_comments.php` in Listing 11.2 takes the data sent from the comments form and sends it on to the owner of the website by email.

This script performs a loop through all the elements in `$_POST` and builds up the string `$body`, which becomes the body text of the email message. Note that `\n` characters are used to separate lines in the output because a plain-text email is created, which means no HTML formatting is required.

**LISTING 11.2**    `send_comments.php`

---

```
<?php

$body = "These comments were sent via the website\n\n";

foreach($_POST as $field => $value) {
 $body .= sprintf("%s = %s\n", $field, $value);
}

mail("owner@website.com", "Comments sent via website", $body,
 'From: "WebComments" <comments@website.com>');

?>
<H1>Thank You</H1>
Your comments have been sent!
```

The email sent to the site owner should look something like the following:

The following comments were submitted via the website

```
name = Chris Newman
email = chris@lightwood.net
gender = m
referrer = search
may_contact = Y
comments = This is my favorite website ever
```

The format of this email is very rough because each line is generated automatically. Of course, if you prefer, you could spend much longer creating a nicely formatted email; for instance, you could replace the coded values of `gender` and `referrer` with their full descriptions.

## Summary

In this lesson you have learned how to process user-submitted data from HTML forms. In the next lesson you will learn how to use PHP to create HTML form items such as menus and radio button groups on-the-fly.

# LESSON 12

## Generating Dynamic HTML



*In this lesson you will learn how to create elements of an HTML form by using PHP. These techniques enable you to specify default values for input items and create dynamic drop-down menus or radio button groups based on data in a script.*

### Setting Default Values

Let's begin with some simple examples that embed PHP within form elements to set the default values of some items when the page is loaded.

#### Default Input Values

The default value of a text input is given in the `VALUE` attribute. This value displays in the field when the page is loaded, and, unless it is overtyped, the same value is sent to the PHP processing script when the form is submitted.

Consider a shopping cart page for an online store, where customers are given the opportunity to change the quantity of each item in their cart before finalizing the order. The current quantity of each line item would be displayed in a small text input box and could be overtyped, and then the user would be able to click a button to refresh the contents of the cart. Listing 12.1 is a very simple example of this, for a store that sells only one product but allows you to choose the quantity to buy.

**LISTING 12.1**    Defaulting the Value of a Text Input Field

```
<?php
if(isset($_POST["quantity"]))
 $quantity = settype($_POST["quantity"], "integer");
else
 $quantity = 1;

$item_price = 5.99;
printf("%d x item = %.2f",
 $quantity, $quantity * $item_price);
?>
<FORM ACTION="buy.php" METHOD=POST>
Update quantity:
<INPUT NAME="quantity" SIZE=2
 VALUE="<?php echo $quantity;?>">
<INPUT TYPE=SUBMIT VALUE="Change quantity">
</FORM>
```

First of all, you set an overall default value for `$quantity` of 1, so that the first time the page is loaded, this is the quantity displayed in the field and used to calculate the total price. Then, inside the `VALUE` tag, you run a single PHP statement to echo the value of `$quantity`. If a quantity value is posted to the form, then that value is used instead.

This script should be called `buy.php` so that the form posts to itself when submitted. If you change the quantity value and press the submit button, the script calculates the new total price. Also, the quantity input field defaults to the value you just entered when the page reloads.

## Checking a Check Box

The `CHECKED` attribute determines whether a check box is on or off by default when a page loads. Using PHP, you can embed a condition within the `<INPUT>` tag to determine whether to include the `CHECKED` attribute on a check box:

```
<INPUT TYPE="CHECKBOX"
 NAME="mybox" <?php if(condition) echo "CHECKED";?>
```

The way this looks can be confusing, particularly because two `>` symbols appear at the end of the tag—one to close the PHP section and one to close the `<INPUT>` tag. In fact, the position of the `CHECKED` attribute is not

important, so depending on your preference, you can move it around for readability:

```
<INPUT <?php if(condition) echo "CHECKED";?>
 TYPE="CHECKBOX" NAME="mybox">
```



**Closing PHP Tags** When embedding small chunks of PHP, you should always try to include the closing `?>` tag as soon as possible. If you miss this closing tag, PHP attempts to interpret the subsequent HTML as PHP and is likely to come up with some interesting error messages!

Spacing can be very important when you're using PHP within HTML. In the previous example, if there is not a space on either side of the PHP statement and the condition is true, the actual HTML produced is as follows:

```
<INPUT CHECKEDTYPE="CHECKBOX" NAME="mybox">
```

Because `CHECKEDTYPE` is not recognized as part of the `<INPUT>` tag, your browser is likely to display this as a text input box, not a check box! It's always better to have too much space around dynamic elements in HTML tags than to risk not having enough.

## Selecting a Radio Button Group Item

The `CHECKED` attribute is also used to specify which item in a radio button group should be selected by default. For example, an online store may offer three shipping options, with each having a different cost. To make sure the customer always chooses a shipping option, one of the selections would be picked by default, with the option to change it if desired (a radio button cannot be deselected except when another button in the same group is selected):

```
<INPUT TYPE="RADIO" CHECKED
 NAME="shipping" VALUE="economy"> Economy
<INPUT TYPE="RADIO" NAME="shipping" VALUE="express"> Standard
<INPUT TYPE="RADIO" NAME="shipping" VALUE="express"> Express
```

To dynamically assign the CHECKED attribute to one of the items in the radio button group, each one must contain a condition that checks the current value of \$shipping against the value that corresponds to that item. Listing 12.2 gives an example.

---

**LISTING 12.2**    Selecting a Default Radio Button Group Item

---

```
<?php
if (!isset($shipping))
 $shipping = "economy";

echo "Your order will be sent via $shipping shipping";
?>

<FORM ACTION="shipping.php" METHOD=POST>

<INPUT TYPE="RADIO" NAME="shipping" VALUE="economy"
 <?php if ($shipping == "economy") echo "CHECKED";?>> Economy

<INPUT TYPE="RADIO" NAME="shipping" VALUE="standard"
 <?php if ($shipping == "standard") echo "CHECKED";?>>
 Standard

<INPUT TYPE="RADIO" NAME="shipping" VALUE="express"
 <?php if ($shipping == "express") echo "CHECKED";?>> Express
<INPUT TYPE="SUBMIT" VALUE="Change shipping option">
</FORM>
```

Notice how cumbersome this is, even for a short radio button group of just three items. Later in this lesson you will learn how to create radio button groups on-the-fly so that larger radio button groups can be managed in a much more elegant way.

## Defaulting a Selection in a Menu

The SELECTED attribute in an <OPTION> tag specifies which item is to be selected by default. If no item has the SELECTED attribute, the first item in the list is shown by default.

Using PHP to display the SELECTED attribute against the appropriate option is just as cumbersome as picking the selected item in a radio button



group, and the same technique applies. Listing 12.3 shown the same example of shipping rates as in Listing 12.2, using a drop-down menu instead of a radio button group.

---

**LISTING 12.3** Selecting a Default Item from a Menu

---

```
<?php
if (!isset($shipping))
 $shipping = "economy";

echo "Your order will be sent via $shipping shipping";
?>
<FORM ACTION="shipping.php" METHOD=POST>
<SELECT NAME="shipping">
<OPTION <?php if ($shipping == "economy") echo "SELECTED";?>
 VALUE="economy">Economy</OPTION>
<OPTION <?php if ($shipping == "standard") echo "SELECTED";?>
 VALUE="standard">Standard</OPTION>
<OPTION <?php if ($shipping == "express") echo "SELECTED";?>
 VALUE="express">Express</OPTION>
<INPUT TYPE="SUBMIT" VALUE="Change shipping option">
</FORM>
```

As with a radio button group, using a function to generate on-the-fly menus allows you to work with much larger option lists and still dynamically select a chosen option.

## Creating Form Elements

Now let's look at how some of the items in an HTML form can be generated by using custom PHP functions. This type of modularization means that you can use a function over and over again whenever you need to include the same type of item on a form.

### Creating a Dynamic Radio Button Group

A modular routine to generate a radio button group requires three pieces of information: the name of the group, a list of values, and a list of labels. You can use an associative array to pass the values and labels to the function in one go.

Say you want to be able to generate the HTML for a radio button group by using simple code similar to the following:

```
$options = array("economy" => "Economy",
 "standard" => "Standard",
 "express" => "Express");
$default = "economy";
$html = generate_radio_group("shipping", $options, $default);
```

As you can see, this is the kind of function you are likely to use again and again when creating HTML forms, and it is very useful to build up a toolbox of similar functions to make it easy to perform common tasks. Here's how the `generate_radio_group` function might be implemented:

```
function generate_radio_group($name, $options, $default="") {
 $name = htmlentities($name);
 foreach($options as $value => $label) {
 $value = htmlentities($value);
 $html .= "<INPUT TYPE=\"RADIO\" ";
 if ($value == $default)
 $html .= "CHECKED ";
 $html .= "NAME=\"{$name}\" VALUE=\"{$value}\">";
 $html .= $label . "
";
 }
 return($html);
}
```

At the heart of the function is a loop through `$options` that generates each `<INPUT>` tag in turn, giving each tag the same `NAME` attribute but a different `VALUE` attribute. The label text is placed next to each button, and in this sample function, the only formatting is to place a `<br>` tag between each button in the group. You could format the options in a table or in any other way you see fit.

At each step of the loop, the script compares the current value of `$value` with the passed-in `$default` value. If they match, the `CHECKED` attribute is included in the generated HTML. Again, spacing is important here; note that the space after `CHECKED` is added to the HTML string.

The `$default` argument is specified as optional. If `generate_radio_group` is called with only two arguments, none of the radio buttons will be selected by default.



**HTML Entities** The `htmlentities` function is used to replace certain characters in a string with corresponding HTML entities. Because the values of `$name` and `$value` are output inside another HTML tag, the `htmlentities` function is important to ensure that there are no characters in those strings that could break the tag.

## Creating a Dynamic Menu

The process for creating a drop-down menu is very similar to that for creating a radio button group. Again, a loop is required—this time to generate an `<OPTION>` tag for each option in turn. The function also needs to include the `<SELECT>` and `</SELECT>` tags around the option list. The function `generate_menu` would look like this:

```
function generate_menu($name, $options, $default="") {

 $html = "<SELECT NAME=\""$name\"">";
 foreach($options as $value => $label) {
 $html .= "<OPTION ";
 if ($value == $default)
 $html .= "SELECTED ";
 $html .= "VALUE=\""$value\"">$label</OPTION>";
 }
 $html .= "</SELECT>";
 return($html);
}
```

The string returned by this function contains the entire HTML code to produce a drop-down menu that contains the supplied options. You might prefer to have the function return only the option tags and place your own `<SELECT>` tags around them; this would allow you to easily add a JavaScript `onChange` event on the menu, for instance.

## Multiple Selection Items

When used with the `MULTIPLE` attribute, the `<SELECT>` form item allows a user to choose multiple options from a menu, usually by holding the `Ctrl`



```

if (!is_array($default))
 $default = array();

foreach($options as $value => $label) {
 $html .= "<INPUT TYPE=CHECKBOX ";
 if (in_array($value, $default))
 $html .= "CHECKED ";
 $html .= "NAME=\"{$name}[]\" VALUE=\"{$value}\">";
 $html .= $label . "
";
}
return($html);
}

$options = array("movies" => "Going to the movies",
 "music" => "Listening to music",
 "sport" => "Playing or watching sports",
 "travel" => "Traveling");

$html = generate_checkboxes("interests",
 $options, $interests);

?>
<H1>Please select your interests</H1>
<FORM ACTION="interests.php" METHOD=POST>
<?php print $html;?>
<INPUT TYPE=SUBMIT VALUE="Continue">
</FORM>

```

In the function `generate_checkboxes`, the `$default` argument is an array rather than a single value; after all, more than one of the options might be selected by default. The array passed in as `$default` can be exactly the same array that is submitted to PHP by the HTML that this function creates.

To find out whether each check box should have the `CHECKED` attribute, `in_array` is called to see whether the current option name is in the list of default values. If `$value` appears anywhere in `$default`, the check box will be checked when the page loads.

Listing 12.4 shows this function in action, using a section of a web page that asks a user about her interests. She can select any number of items from the list, and, in this example, the script submits to itself with the options remaining checked so that the user can change her mind if she wants to.

In the array `$interests` created in PHP, each element is a key name from `$options`. If you want to find the label that corresponds to each selected option, you can reference the corresponding element from `$options`.

## Summary

In this lesson you have learned how to generate HTML components on-the-fly and learned some techniques for creating dynamic form input objects. In the next lesson you will learn how to perform validation on an HTML form.

## LESSON 13

# Form Validation



*In this lesson you will learn some techniques for validating form input in a user-friendly way.*

The principles of validating user-submitted input are fairly straightforward: Just check each item in `$_POST` in turn and make sure it matches a set of criteria. However, making sure the user is able to correct any mistakes and resubmit the form with a minimum of fuss presents a bit more of a challenge.

## Enforcing Required Fields

The most basic type of form validation is to enforce that a particular field must contain a value. In the case of a text input that is submitted with no value entered, the element in `$_POST` is still created, but it contains an empty value. Therefore, you cannot use `isset` to check whether a value was entered; you must check the actual value of the element, either by comparing it to an empty string or by using the following, more compact syntax with the Boolean NOT operator:

```
if (!$_POST["fieldname"]) { ... }
```

Because each field on the form creates an element in `$_POST`, if every field requires a value to be entered, you could use a simple loop to check that there are no empty values in the array:

```
foreach($_POST as $field => $value) {
 if (!$value) {
 $err .= "$field is a required field
";
 }
}
if ($err) {
```

```
 echo $err;
 echo "Press the back button to fix these errors";
}
else {
 // Continue with script
}
```

Rather than exit as soon as an empty field is found, this script builds up an error string, `$err`. After the validation is done, the contents of `$err` are displayed if there are any errors. If there are no errors, `$err` is empty, and script execution continues with the `else` clause.



**Validation Warnings** Always show all the warning messages that relate to the submitted data straight away. You should give your users the opportunity to correct their errors all at one time.

One obvious limitation of this approach is that you cannot pick which fields require a value; every posted field must have been completed. You could improve upon this by supplying a list of required fields in the script, and by using an associative array, you can also provide a label for each field to display in the warning message:

```
$required = array("first_name" => "First name",
 "email" => "Email address",
 "telephone" => "Telephone number");
foreach($required as $field => $label) {
 if (!$_POST[$field]) {
 $err .= "$label is a required field
";
 }
}
```

## Displaying Validation Warnings

Another issue to consider is where to send the user when validation fails. So far we have assumed that a form submits to a processing script, and when one or more validation errors are found, the form prompts the user to use his or her browser's Back button to fix the errors.



Not only does this create one more step for the user to take in order to complete the form—and in an online store, you want as few obstacles between a customer and a completed order as possible—it can also sometimes cause the data in the form fields to be lost when Back is clicked.

Whether the Back button causes data to be lost usually depends on the cache settings, either on the web server, in the user's browser, or at the user's Internet service provider. In many cases there is no problem. However, most notably when a PHP session has been started, no-cache headers are automatically sent to the browser, which causes data in form fields to be reset to their original values when you click the Back button. You will learn more about PHP sessions in Lesson 14, "Cookies and Sessions."

A good technique is to have the form and processing script in the same file and have the form submit to itself. This way, if there are errors, they can be displayed on the same page as the form itself, and the previously entered values can be automatically defaulted into the form.

Listing 13.1 shows a fairly complete example of a registration form, `register.php`. The name and email address fields are required, but the telephone number is optional.

---

**LISTING 13.1** A Sample Registration Form with Required Fields

---

```
<?php

$required = array("name" => "Your Name",
 "email" => "Email Address");

foreach($required as $field => $label) {
 if (!$_POST[$field]) {
 $err .= "$label is a required field
";
 }
}

if ($err) {
 echo $err;
}

?>
<FORM ACTION="register.php" METHOD=POST>
```

**LISTING 13.1**    Continued

---

```

<TABLE BORDER=0>
<TR>
 <TD>Your Name</TD>
 <TD><INPUT TYPE=TEXT SIZE=30 NAME="name"
 VALUE="<?php echo $_POST["name"];?>"></TD>
</TR>
<TR>
 <TD>Email Address</TD>
 <TD><INPUT TYPE=TEXT SIZE=30 NAME="email"
 VALUE="<?php echo $_POST["email"];?>"></TD>
</TR>
<TR>
 <TD>Telephone</TD>
 <TD><INPUT TYPE=TEXT SIZE=12 NAME="telephone"
 VALUE="<?php echo $_POST["telephone"];?>"></TD>
</TR>
</TABLE>
<INPUT TYPE=SUBMIT VALUE="Register">
</FORM>

<?php
}
else {
 echo "Thank you for registering";
}
?>

```

Note that the warning messages in this example appears even if the form has not yet been submitted. This could be improved by also checking for the existence of the `$_POST` array in the script by using `is_array`, but the check for `$err` would also need to look for `$_POST`; otherwise, the form would never be displayed.

The condition that checks `$err` spans the HTML form and, even though the PHP tags are closed around this chunk of HTML, the form is displayed only if that condition is true.

In this example, after the form has been completed successfully, only a simple message is displayed. This is where you would do any processing based on the submitted data, such as storing it to a database, which you will learn about in Lesson 19, “Using a MySQL Database.” Alternatively,

the script could force the browser to redirect the user to another page automatically by using a `Location` HTTP header, as follows:

```
header("Location: newpage.php");
```

## Enforcing Data Rules

You will often want to ensure not only that data is entered into required fields but that the quality of the data is good enough before proceeding. For instance, you might want to check that an email address or a phone number has the right format, using the rules developed in Lesson 8, “Regular Expressions.” You could also enforce a minimum length on a field to make sure a user cannot just enter an *x* in each field to continue to the next page.

Because each field will probably have a different validation rule, you cannot enforce data rules in a loop; you must instead write a rule for each field to be checked. However, when used in conjunction with the check for empty fields in a loop from the previous examples, you should also check that the value has been entered before doing any further validation. Otherwise, the warning message will first tell a user that a field is required and then also that it has been entered in the wrong format!

The following rules could be added to Listing 13.1 after the required fields check to enforce suitable values for email address and telephone number:

```
if ($_POST["email"] &&
 !ereg("^^[^@]+@([a-z0-9-]+\.)+[a-z]{2,4}$",
 $_POST["email"]))
 $err .= "Email address format was incorrect
";

if ($_POST["telephone"] &&
 !ereg("^[[:digit:]]{3}\)[[:digit:]]{3}-[[:digit:]]{4}$",
 $_POST["telephone"]))
 $err .= "Telephone must be in format (555)555-5555
";
```

Because these additional rules add new messages to `$err` if an error is found, the rest of the script remains unchanged.

## Highlighting Fields That Require Attention

Rather than bombard the user with a list of warning messages, it's more user-friendly to highlight the fields in the form that require attention.

The technique to use here is very similar to the previous example, but rather than append each warning message to `$err`, you should give each field its own warning text. If you use an array of warnings, it's simple to see whether the form has been successfully validated by counting the elements in `$warnings`.

You write each rule to add an element to `$warnings` if validation of that field fails, as shown in the following example:

```
if (!ereg("^[^@]+@([a-z\-.]+\.)+[a-z]{2,4}$",
 $_POST["email"]))
 $warnings["email"] = "Invalid Format";
```

Then in the form itself, you can display this warning text next to the corresponding field:

```
<TR>
 <TD>Email Address</TD>
 <TD><INPUT TYPE=TEXT SIZE=30 NAME="email"
 VALUE="<?php echo $_POST["email"];?>"></TD>
 <TD><?php echo $warnings["email"];?></TD>
</TR>
```

Listing 13.2 shows a revised `register.php` file that uses this technique to highlight missing or invalid field values.

---

### LISTING 13.2 Form Validation Using Inline Warnings

---

```
<?php

$required = array("name" => "Your Name",
 "email" => "Email Address");

foreach($required as $field => $label) {
 if (!$_POST[$field]) {
 $warnings[$field] = "Required";
```

```

 }
}

if ($_POST["email"] &&
 !ereg("^^[^@]+@[a-z\-\]+\.[a-z]{2,4}$", $_POST["email"]))
 $warnings["email"] = "Invalid email";

if ($_POST["telephone"] &&
 !ereg("^\[[:digit:]]{3}\[[:digit:]]{3}-\[[:digit:]]{4}$",
 $_POST["telephone"]))
 $warnings["telephone"] = "Must be (555)555-5555";

if (count($warnings) > 0) {

?>
<FORM ACTION="register.php" METHOD=POST>
<TABLE BORDER=0>
<TR>
 <TD>Your Name</TD>
 <TD><INPUT TYPE=TEXT SIZE=30 NAME="name"
 VALUE="<?php echo $_POST["name"];?>"></TD>
 <TD><?php echo $warnings["name"];?></TD>
</TR>
<TR>
 <TD>Email Address</TD>
 <TD><INPUT TYPE=TEXT SIZE=30 NAME="email"
 VALUE="<?php echo $_POST["email"];?>"></TD>
 <TD><?php echo $warnings["email"];?></TD>
</TR>
<TR>
 <TD>Telephone</TD>
 <TD><INPUT TYPE=TEXT SIZE=12 NAME="telephone"
 VALUE="<?php echo $_POST["telephone"];?>"></TD>
 <TD><?php echo $warnings["telephone"];?></TD>
</TR>
</TABLE>
<INPUT TYPE=SUBMIT VALUE="Register">
</FORM>

<?php
}
else {
 echo "Thank you for registering";
}
?>

```

The first loop assigns the warning text “Required” to any required field that is left blank. Each of the individual validation rules has its own warning text.

How you highlight a field that requires attention is up to your imagination and creativity with HTML. For instance, by checking for the presence of an element in \$warnings for each field, you could change the style of the input box to a shaded background, like so:

```
<INPUT TYPE=TEXT SIZE=30 NAME="email"
<?php if ($warnings["email"]) echo "STYLE=\"shaded\"";?>
VALUE="<?php echo $_POST["email"];?>">
```

## Summary

In this lesson you have learned how to validate user input from HTML forms and how to present the form back to the user so that he or she can correct any errors. In the next lesson you will learn about cookies and sessions in PHP.

# LESSON 14

## Cookies and Sessions



*This lesson examines two ways of passing data between pages of a website without requiring a form submission from one page to another: using cookies and using sessions.*

### Cookies

*Cookies* are small pieces of information that are stored in your web browser. They typically contain data that is used to identify you when you look at a website so that site can be customized for each visitor.

Rather than having to pass data to a script by using a form or as values in the query string, cookies are sent back to your scripts automatically by your web browser. Even if you go off and browse to another website, their values are remembered when you return.

For example, if you have to log in to access a particular website, you may be able to let a cookie remember your username so you do not have to type it each time you go back; in this case, you only have to enter your password. Or on a community site, your browser might record the date you last visited in a cookie, so that any forum messages posted since you last visited can be highlighted as new.

### Cookie Ingredients

Each cookie consists of a name and a value, just like regular variables in PHP. The instruction to create a cookie in your web browser is sent as an HTTP header before a web page is transmitted; when your web browser sees this header, it takes the appropriate action.

The HTTP headers that create cookies are the same, regardless of whether they are generated by PHP or any other means of interfacing with your web server. The header used to set a cookie called `email` might look like this:

```
Set-Cookie: email=chris@lightwood.net
```



**HTTP Headers** You will never see an actual HTTP header in your web browser. We will look at how different types of HTTP headers are sent in PHP in Lesson 16, “Communicating with the Web Server.”

A cookie also has an expiration date; some cookies last only as long as your web browser is open and are kept in your computer’s memory, whereas others have a fixed expiration date in the future and are saved to your hard disk. The HTTP header to set the email cookie that will expire at the end of 2005 would look like this:

```
Set-Cookie: email=chris@lightwood.net;
 expires=Sat, 31-Dec-2005 23:59:59 GMT
```

If no `expires` attribute is sent in the `Set-Cookie` header, the cookie will be destroyed when the web browser is closed.

The other attributes that can be set are the domain name and the path by which a browser will send back a cookie. When you make any subsequent visit to a page for which you have a cookie set, its name and value are sent to the web server.

The default behavior is to send a cookie back to any page on the same domain that it was set from. By setting the domain and path, you can tell the cookie to be sent back to other subdomains or only to scripts in a certain part of the site.

The following header creates an email cookie that is sent back to any subdomain of `lightwood.net`, as long as the page requested is in the `/scripts` subdirectory:

```
Set-Cookie: email=chris@lightwood.net; domain=.lightwood.net;
 path=/scripts
```





**Subdomains** You can only set the domain attribute of a cookie to a variant of the domain from which the cookie was originally set, or to `.yourdomain.com` to indicate all subdomains.

This is a security measure to prevent some websites from being able to confuse others. For example, you cannot set a cookie that would be sent back to `www.php.net` from any website that is not hosted at `php.net`.

## Accessing Cookies

The `$_COOKIE` super-global array in PHP contains all the cookies that have been sent to the current script. Cookies are sent back to the web server in an HTTP header, and PHP builds the `$_COOKIE` array based on this information.

You can access cookies in the same way that you reference posted form data. For example, the following statement displays the current value of the email cookie:

```
echo $_COOKIE["email"];
```

If you ever feel that your cookies are getting in a bit of a mess, you can just create a script to dump them all out to screen so you can see what's going on. It is as simple as this:

```
echo "<PRE>";
print_r($_COOKIE);
echo "</PRE>";
```

## Making Cookies with PHP

Although you have now seen how to create cookies by using HTTP headers, you will probably not use this method again because PHP contains a function that makes cookie setting much easier:

```
setcookie("email", "chris@lightwood.net", time() + 3600);
```

Rather than the strictly formatted textual date shown in the header example earlier in this lesson, you specify the expiration date in `setcookie` as a Unix timestamp. This makes it easy to set a cookie that lasts for a fixed amount of time or until a date and time in the future.



**Expiration Times** The expiration argument specifies the latest date and time that a stored cookie will be transmitted. As time comparison is performed on the local computer, the actual expiration of cookies is determined by the local system clock and, if that clock is incorrect, is beyond your control.

The next two optional arguments are used to specify the domain and path for the cookie. If you want to set a domain and path but not an expiration time, you use `NULL` for the third argument:

```
setcookie("email", "chris@lightwood.net", NULL,
 ".lightwood.net", "/scripts");
```

The final optional argument to `setcookie` is a flag that tells the browser to send the cookie back to the server only over an SSL encrypted connection—in other words, only for web pages with addresses that begin `https://`.



**Password Cookies** As handy as it may be to have a password stored in a cookie so that you can be automatically logged in to a website when you revisit it, this is very dangerous, even when the secure flag is set.

Cookies are stored in plain text and can be viewed simply by looking in the correct place on your hard disk. Malicious spyware programs exist that try to steal your passwords by searching through your cookies!

## Deleting Cookies

There is no `unsetcookie` function to tell the web browser to delete a cookie. To stop a cookie value from being sent back to the web server, you use `setcookie` with an empty value and an expiration date that has already passed.

The following example unsets the email cookie by using an expiration value that is one hour ago:

```
setcookie("email", "", time() - 3600);
```



**Overwriting Cookies** When unsetting a cookie or when overwriting an existing cookie with a new value, you must make sure the domain, path, and `ssl-only` arguments are exactly the same as when the cookie was originally created.

## Sessions

Sessions are very similar to cookies in that they can be used for passing values between pages of a website. Rather than storing the values in each web browser, however, the values are stored on the web server, and a single identity cookie is used to tell PHP which set of values corresponds to the current user.

Because much less data is sent back and forth between the web server and browser, sessions are more efficient than cookies when larger amounts of data are stored.

## Creating a Session

To initialize a new session in a PHP script, you use the `session_start` function. You can use an optional argument to specify a session name, but usually this is not required. Every script on your site that starts the same session will be able to access the same set of session variables.

The call to `session_start` to create a new session is as simple as the following:

```
session_start();
```

The `$_SESSION` super-global array is used to store and retrieve session variables. Unlike the other super-globals you have encountered so far, you can assign values directly to `$_SESSION`, after which they are available to any script that shares the session.

Consider the script in Listing 14.1, which maintains two session variables—a count of the number of times you have viewed the page and the timestamp of the last visit.

---

**LISTING 14.1**    Using Session Variables to Track Visits to a Page

---

```
<?php

session_start();

if ($_SESSION["last_visit"]) {
 echo "Date of last visit: ";
 echo date("j F Y, H:i:s", $_SESSION["last_visit"]);
 echo "
";
 echo "Total visits: " . $_SESSION["num_visits"];
}
else
 echo "This is your first visit";

$_SESSION["last_visit"] = time();
$_SESSION["num_visits"]++;
?>
```

Each time the page is loaded, the old values are displayed and the new values set. Notice that if you surf to other websites and then come back, these values are remembered, but if you close your web browser and come back, the values are reset.

## Using Session Variables

One of the advantages of session variables over cookies is their ability to use PHP's data types. Cookie values are always simple text values, but a session variable can take any value that a regular PHP variable can.

For instance, to store a list of items in a cookie, you would have to create an array and pass it to `serialize` to store. By using a session variable, you can create an array directly and store that data structure in the session.

The example in Listing 14.2 uses an array stored in the session to retain a list of values entered through a form. This is a fairly trivial example, but it demonstrates the flexibility you have when using session variables.

---

**LISTING 14.2** Using Arrays as Session Variables

---

```
<?php

session_start();

if (isset($_POST["word"]))
 $_SESSION["words"][] = $_POST["word"];

if (is_array($_SESSION["words"])) {
 foreach($_SESSION["words"] as $word) {
 echo $word . "
";
 }
}

?>
<FORM ACTION="list.php" METHOD=POST>
Enter a word: <INPUT SIZE="10" NAME="word">
<INPUT TYPE=SUBMIT VALUE="Add word to list">
</FORM>
```

## Summary

In this lesson you have learned how to set cookies from PHP and how to use PHP's session management to store values within a browser session. In the next lesson you will use these techniques to create a user authentication system using PHP.



## LESSON 15

# User Authentication

*In this lesson you will build a user authentication process that can be used to protect certain pages of your website by using a password.*

## Types of Authentication

Chances are you have needed to log in to a website in the past, so you should be aware of how the process of authentication works from a user's point of view. Generally speaking, you are asked to enter a username—sometimes your email address—and a password.

There are actually two ways that a website can authenticate a user, though: using basic HTTP authentication and using session-based authentication. The following sections clarify the differences between these two methods.

## Basic HTTP Authentication

Basic HTTP authentication can be performed by web server, without having anything to do with PHP script. The example in this section assumes that you are using Apache web server; for other web servers, you should refer to your documentation.

Authentication is usually done on a per-directory basis but can be set up to apply to individual files if required. By using an `.htaccess` file on your website, you can specify for that directory a custom configuration that instructs the web server to require a login before proceeding. A typical set of configuration directives would look like this:

```
AuthType Basic
AuthName "Protected Website"
```

```
AuthUserFile /home/yourname/htpasswd
require valid-user
```

AuthUserFile points to the location of a password file that is created by using the htpasswd program. To create a new password file, you would run a command like the following:

```
$ htpasswd -c /home/yourname/htpasswd chris
New password:
Re-type new password:
```



**Password Files** You should use the `-c` switch only when you want to create a new file. The htpasswd program does not ask whether you want to overwrite an existing file. Running htpasswd without the `-c` option on an existing password file adds a user.

You have to enter the new password twice, after which an entry is added to the password file given. The entry consists of the username and an encrypted version of the password, separated with a colon character. However, you should never need to work with this file directly. A typical password file entry might look like this:

```
chris:XNiv7qSUTFPU6
damon:ZxxE2PTExeVNU
shelley:SVzAEtxMLEAIs
vanessa:cX/t1Pv2oQfrY
```

When you try to access a file in the protected directory, your web browser pops up a window that asks for a username and password, and the page requested loads only after you have entered the correct information.

The `require valid-user` directive instructs the web server to show the page to any authenticated user. You might want to grant access to only certain users, which you can do with the `require user` directive:

```
require user chris damon shelley
```

Basic HTTP authentication also allows you to set up user groups to give access to particular sections of the site only to certain users. You can then

use the `require group` directive to specify access to one or more user groups.

The following groups file, usually named `htgroups`, divides the users in the password file into two groups:

```
boys: chris damon
girls: shelley vanessa
```

To give access only to the boys group, you could use the following `.htaccess` file:

```
AuthType Basic
AuthName "Boys Only"
AuthUserFile /home/yourname/htpasswd
AuthGroupFile /home/yourname/htgroup
require group boys
```

Although it is fairly easy to set up and reasonably flexible, basic HTTP authentication has some drawbacks. First, you cannot change the look and feel of the pop-up login box. If you want to customize the process at all, you cannot use this method. Furthermore, the password file is stored on the server's filesystem, and updating it from a script may be problematic; you will learn more about these issues when dealing with reading and writing to files in Lesson 17, "Filesystem Access."



**Apache Add-ons** Several third-party modules for the Apache web server—such as `mod_auth_mysql` and `mod_auth_sqlite`—allow you to use basic HTTP authentication with password information stored in a database. Check with your web host to see whether these modules are installed.

## Session-Based Authentication

To provide a completely customizable login process for your website, you must implement it yourself, and doing so in PHP requires using session variables.



In a nutshell, once a user is logged in, the browser's session contains enough information to convince the scripts on the website that you are allowed to view a page. Users log in by using a form on your site where they enter their username and password. You can set up the layout and flow of the login process any way you see fit.

One fairly significant difference from basic HTTP authentication is that the instruction to check the validity of a user's session appears in the script itself, not in a per-directory configuration file.



**Protecting HTML** If your website includes plain HTML files that contain no PHP, you need to add PHP code to them to prevent them from being viewable to an anonymous user. You also need to change their file extension to `.php`.

## Building an Authentication System

The rest of this lesson walks you through building an authentication mechanism using PHP sessions.

### How the System Works

There are two main components of the authentication system you're going to build now. First, you need a login processor that checks the validity of the username and password entered in the form. You also need a piece of code that can be put at the top of each script to check the session and make sure the user is authenticated before continuing.



**Login Forms** You need to make sure you always use the `POST` method for login forms. Submitting a username and password by using the `GET` method causes these values to appear in the URL of the next page for anyone to see!

You should split off the session-checking code into an include file, `auth.inc`, so that it is simple to protect a page by simply putting the following statement at the top of the script:

```
include "auth.inc";
```

You can use a single session variable to store the username of the logged-in user. If the variable contains a username, that user is logged in; logging a user out is as simple as deleting this session variable. As long as nobody else shares the domain on which your website is hosted and could create a conflicting session, this is adequately secure. Knowing this, `auth.inc` can really be as simple as the following:

```
session_start();
if (!isset($_SESSION["auth_username"])) {
 echo "You must be logged in to view this page";
 exit;
}
```

Here you simply display a message and exit the script if the user is not logged in. You will see later on how you can improve this for usability, but you need to create the login process itself first.

## Authenticating a Login

The login form, at its heart, needs to contain just two fields—username and password—and a submit button. As long as these are present, the form's layout is up to you. For now, you can keep it fairly plain, in a simple table layout. Listing 15.1 shows the basic login form.

### **LISTING 15.1**    A Basic Login Form

---

```
<FORM ACTION="login.php" METHOD="POST">
<TABLE BORDER=0>
<TR>
 <TD>Username:</TD>
 <TD><INPUT TYPE="TEXT" SIZE=10 NAME="username"></TD>
</TR>
<TR>
 <TD>Password:</TD>
 <TD><INPUT TYPE="PASSWORD" SIZE=10 NAME="password"></TD>
</TR>
</TABLE>
```

```
<INPUT TYPE=SUBMIT VALUE="Log in">
</FORM>
```



**Password Fields** The PASSWORD type input works exactly the same way as a TEXT type, but the characters entered are obscured as they are typed. The only restriction on a password field is that it cannot be given a VALUE attribute for a default value.

The form handler script, `login.php`, needs to check the submitted username and password values against the list of valid users. In most cases, you would check the values against a user database. You will learn about database access in PHP in Lessons 19, “Using a MySQL Database,” and 20, “Database Abstraction,” and for now you can just use a simple array of users who are permitted to use the site. Listing 15.2 shows how to do this.

---

**LISTING 15.2** A Login Processor Script

---

```
<?php

session_start();

$passwords = array("chris" => "letmein",
 "damon" => "thisisme",
 "shelley" => "mypassword",
 "vanessa" => "opensesame");

if (!$_POST["username"] or !$_POST["password"]) {
 echo "You must enter your username and password";
 exit;
}

if ($_POST["password"] == $passwords[$_POST["username"]]) {
 echo "Login successful";
 $_SESSION["auth_username"] = $_POST["username"];
}
else {
 echo "Login incorrect";
}
?>
```

First, an associative array of passwords is built, using the usernames as keys. The script first checks that both the username and password have been entered and exits immediately if that information is missing.

Then the submitted password is compared to the array element whose key is the submitted username. If the two passwords match, the user is logged in, and the `auth_username` session variable is initialized. Otherwise, a message is displayed that the login failed.

After a user's session has been validated, he or she can view a protected page without `auth.inc` interrupting the script's progress.

## Encrypting Passwords

In the previous example, the passwords are stored in plain text. You probably suspected that this is not particularly secure; anyone who can view the source code of this script can see the passwords for every user.



**Prying Eyes** Even if your server security is airtight, can you be sure that nobody is looking over your shoulder? You should always try to prevent unencrypted passwords from being displayed onscreen.

The `crypt` function in PHP provides a simple but effective one-way encryption algorithm. The same kind of encryption is used by `htpasswd` and even Unix system passwords. To encrypt a password, you pass the password to `crypt`, along with `$salt`—another string around which the encryption is based:

```
$crypt_password = crypt($password, $salt);
```

Although the encrypted string cannot be decoded back to the original password, every time you run `crypt` on the same password with the same `salt`, the result is the same. Knowing this, you can store only the encrypted version of the password and compare it to the freshly encrypted user input.



**Salts** If you do not specify a salt, a random one is chosen so subsequent calls to `crypt` will produce different results. A two-character salt, as used by Unix password files and `htpasswd`, is sufficient.

How a string encoded using `crypt` looks may vary between different web servers as a system-level encryption library is used. Encrypted passwords are not guaranteed to be portable between different systems. The revised `login.php` file is shown in Listing 15.3, but be aware that the encrypted passwords shown may not be valid for your system.

### LISTING 15.3 A Login Processor Script with Encrypted Passwords

```
<?php

session_start();

$passwords = array("chris" => "ZXsDiRf.VB1WQ",
 "damon" => "bQLXBRzdBci7M",
 "shelley" => "KkTH39mVsoc1c",
 "vanessa" => "69SvRIB9QVukk");

if (!$_POST["username"] or !$_POST["password"]) {
 echo "You must enter your username and password";
 exit;
}

$salt = substr($passwords[$_POST["username"]], 0, 2);
if (crypt($_POST["password"], $salt)
 == $passwords[$_POST["username"]]) {
 echo "Login successful";
 $_SESSION["auth_username"] = $_POST["username"];
}
else {
 echo "Login incorrect";
}

?>
```

The salt is always found in the first two characters of the encrypted string, so you assign these two characters to `$salt` to use in the call to `crypt`. Other than this, the process is identical to using plain-text passwords.

## Usability Considerations

The mechanism you have implemented so far is fairly crude. Any login error results in a message being displayed and the script ending. Even when a login is successful, the flow ends, and the user needs to revisit a protected page directly.

The ideal login mechanism interrupts a hit to a protected web page and displays its login form. Then, after successfully authenticating, it forwards the user to the page he or she was originally trying to access.

One way to add this enhancement is to check the name of the script that the user attempted to access in `auth.inc`; the script name and the query string, if there was one, can be found in `$_SERVER["REQUEST_URI"]`. The login form would then be displayed by `auth.inc` itself, rather than being a separate page.

If you add the following hidden input to the login form, the login processor itself will know which script the user came from, and then you can send the user back to the page he or she was trying to access.

```
<INPUT TYPE="HIDDEN" NAME="destination"
 VALUE="<?php print $_SERVER["REQUEST_URI"];?>">
```

When authentication is successful, rather than print a message to screen, you can forward the user to his or her destination by using the following statement:

```
header("Location: $_POST['destination'];");
```

## Summary

In this lesson you have learned ways to protect web pages by using two different authentication methods, including one that is a feature of HTTP. In the next lesson you will see how other HTTP features can be accessed from PHP.

# LESSON 16

## Communicating with the Web Server



*This lesson looks at ways in which PHP can interact with a web server.*

### HTTP Headers

Every page downloaded from a web server is a result of an exchange of HTTP dialogue. The web browser sends a set of instructions to indicate which page it wants to view, and the server responds with a response that indicates the success of the request, along with various other information that is not displayed directly on the web page.

The following HTTP headers show some of the information that is sent along with a typical web page from a PHP-enabled web server:

```
HTTP/1.1 200 OK
Date: Tue, 14 Dec 2004 21:17:28 GMT
Server: Apache/1.3.29 (Unix) mod_gzip/1.3.26.1a PHP/4.3.9
 mod_ssl/2.8.16 OpenSSL/0.9.7c
X-Powered-By: PHP/4.3.9
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

### Sending Custom Headers

The PHP function to send a custom HTTP header is `header()`. Let's start by sending a header that does nothing. Any header that begins with X is considered to be for information only; for example, the X-Powered-By header

shows that PHP is enabled. To stamp your name in the HTTP headers in your script, you could use the following:

```
header("X-PHP-Author: Chris Newman <chris@lightwood.net>");
```

Of course, there is no reason you should want to send a header like this, other than extreme vanity. A regular user browsing the website would never even see this header!

You have already seen how cookies are sent to a web browser by using the `setcookie` function. You have also seen that what happens when this function is called is that a `Set-Cookie` HTTP header is actually sent. The following two PHP statements are therefore equivalent:

```
setcookie("mycookie", "somevalue");
header("Set-Cookie: mycookie=somevalue");
```

## Redirection Headers

The header you will send most often is almost certain to be `Location`, which instructs the web browser to redirect to another URL. You can use this header to change the flow of a website according to events in script. Causing the user's browser to forward to another page is as simple as this:

```
header("Location: anotherpage.php");
```

You can use either a relative or absolute URL in the `Location` header, so you could even forward the user to another domain, like so:

```
header("Location: http://www.somedomain.com/newpage.php");
```

When a `Location` header has been sent, you should halt the script immediately, using `exit`, to make sure that no further output is sent to the browser.

## Checking Whether Headers Have Been Sent

As soon as PHP hits the first piece of non-header output in a script, it makes sure all the necessary headers have been sent to the web browser and begins to work on the page itself. All the HTTP headers must be sent at once and must be sent before any of the web page output.



If the headers have already been sent for a script and you attempt to send another, PHP gives an error like this:

```
Warning: Cannot modify header information - headers already
sent by (output started at /home/chris/ public_html/
header.php:4)in /home/chris/ public_html/header.php on line 5
```

In the case of a Location header, you don't need to display anything on the page because the browser goes straight to the new URL. However, you still need to be careful to avoid any HTML output, and particularly white-space, before the script begins; even a single carriage return before the opening `<?php` tag will prevent you from being able to send custom HTTP headers.

PHP provides the function `headers_sent`, which you can use to detect whether the HTTP headers have already been sent in that script. The function returns `TRUE` if headers have been sent and `FALSE` if it is not too late to send additional custom headers.

The following condition makes sure the headers have not been sent before attempting to perform a redirection:

```
if (!headers_sent()) {
 header("Location: newpage.php");
}
```

Of course, your script would still need to do something else if this condition failed.

Two optional arguments to `headers_sent` allow you to find out the script name and line number where the headers were sent. This is useful if your script is giving an error but you think that the headers have not been sent at that point.

Listing 16.1 attempts to perform a redirect by using a Location header, but if it fails, it displays the reason and an alternative way to get to the destination page. If you run this on your web server, you should add some whitespace or HTML at the top of the script, outside the `<?php` tags, to make sure the headers are sent prematurely.

**LISTING 16.1**    Checking Whether Headers Have Been Sent

---

```
<?php
$destination = "http://www.lightwood.net/";
if (!headers_sent($filename, $line)) {
 header("Location: $location");
}
else {
 echo "Headers were sent in line $line of $filename
";
 echo "Click here to continue";
}
?>
```

## Displaying HTTP Headers

If you want to see which HTTP headers have been or will be sent, you use the `headers_list` function, which is available in PHP version 5 and above. This function returns an array that contains one header per element.

You can perform a loop on the array returned to grab each value in turn. However, in many cases, all you want to do is see the headers that are being output to check them over, and in this case, passing the array to `print_r` does the trick:

```
print_r(headers_list());
```

You need to make sure to put `<PRE>` tags around this for readability. The following is typical output:

```
Array
(
 [0] => X-Powered-By: PHP/5.0.2
 [1] => Set-Cookie: mycookie=somevalue
 [2] => Content-type: text/html
)
```

## Changing Cache Settings

You can use HTTP headers to change the cache settings for a web page, to determine whether a page is completely refreshed each time it is loaded or whether the user's browser—or his or her ISP—will keep a local copy for a period of time to save downloading it from your website again.

You use the `Cache-Control` header to specify what caching scheme to use for a page. The primary control values for this header are shown in Table 16.1.

**TABLE 16.1** Primary Cache-Control Settings

Value	Description
public	May be stored in any web cache.
private	May be saved to the browser's cache but may not be stored in a shared web cache.
no-cache	May not be stored in any cache between the web server and browser.

Usually the reason for overriding the default cache settings is to make sure that a page is fully refreshed every time it is visited.

In most cases, web caches detect that a PHP-generated page with changing content needs to be refreshed frequently, but to make absolutely sure that all your up-to-the-minute content is being displayed correctly around the world, you might want to give it a helping hand.

To make absolutely sure your page will not be cached, using the following statements, which send a number of headers, is generally considered to be the definitive way to prevent caching of any kind:

```
header("Cache-Control: no-store, no-cache, must-revalidate");
header("Cache-Control: post-check=0, pre-check=0", false);
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
header("Last-Modified: ". gmdate("D, d M Y H:i:s") . " GMT");
```

A few different headers are used here. Two `Cache-Control` headers are sent, including a `no-cache` instruction. You can find more information on the other, less common, `Cache-Control` settings at [www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9](http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9).

The `Expires` header tells the browser when a document goes out of date. If you send a historic date in this header, the document will always be considered to be old and need to be refreshed the next time it is viewed.

The Last-Modified header tells the browser how recently the document was modified. When you use the date function, this header always sends the current date, so the browser always thinks it has only just been modified and requests a new copy of the page in full.



**Session Cache Control** When a PHP session is started, no-cache headers are automatically sent, along with the other HTTP headers that establish the session. You can use a different cache setting by using the session\_cache\_limiter function, with one of the values in Table 16.1 as an argument.

## Server Environment Variables

Now let's look at the information that PHP allows you to find out from your web server.

The `$_SERVER` super-global array contains a number of elements that give information about the web server environment during the current page request. To see the full list within the context of a script, you execute this statement at any time:

```
print_r($_SERVER);
```

The examples in this section are common to most web servers. However, some servers may not support all the values shown or may use different names. You can always refer to the output from the previous statement to check which values are available in your script.

## Script Information

The name of the current script can be found in `$_SERVER["SCRIPT_NAME"]`. Knowing this name can be useful if you want to create a form that submits to itself but whose filename you might want to change in the future. You could use the following tag:

```
<FORM ACTION="<?php print $_SERVER["SCRIPT_NAME"];?>"
METHOD=POST>
```

Similar to `SCRIPT_NAME` is the `REQUEST_URI` element, which contains the full uniform resource identifier of the page request. This consists of the full path to the current script, including the question mark and values in the query string, if there are any. The query string is not included as part of the `SCRIPT_NAME` element, but you can access it on its own as `$_SERVER["QUERY_STRING"]`.

If you want to find the domain name under which a script is running, you can look at `$_SERVER["HTTP_HOST"]`. Your web server might be set up with several alias domains, and this provides a way to see which domain name a visitor is viewing your pages on.

## User Information

The `HTTP_USER_AGENT` element contains a string that identifies the user's web browser software and operating system. It might look like one of the following:

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.5)
 Gecko/20041107 Firefox/1.0
Lynx/2.8.5dev.7 libwww-FM/2.14 SSL-MM/1.4.1 OpenSSL/0.9.7a
```

These three examples correspond to Internet Explorer, Mozilla Firefox, and Lynx, respectively. Notice that both Internet Explorer and Firefox report themselves as Mozilla browsers, so to find out specifically which program a user has, you have to look further into the string.

The following condition can be used to produce different output for Internet Explorer than for Firefox:

```
if (strstr($_SERVER["HTTP_USER_AGENT"], "MSIE")) {
 echo "You are using Internet Explorer";
}
elseif (strstr($_SERVER["HTTP_USER_AGENT"], "Firefox")) {
 echo "You are using Firefox";
}
else {
 echo "You are using some other web browser";
}
```

You may need to use this occasionally because of differences in the browsers' implementations of DHTML or JavaScript.

The `REMOTE_ADDR` attribute contains the IP address that the hit to the web server came from. It should either be the IP address of the user's computer or the user's ISP's web cache. You might want to use the remote IP address for logging or security.

If the `REMOTE_ADDR` value is a web cache, the element `HTTP_X_FORWARDED_FOR` is also present, and it contains the IP address of the user's computer.

If the user has logged in by using basic HTTP authentication, you can also find out his or her username by looking at the value in `$_SESSION["REMOTE_USER"]`.

## Server Information

Other elements in `$_SERVER` allow you to access various values related to the web server configuration.

For instance, `$_SERVER["SERVER_NAME"]` corresponds to the `ServerName` Apache directive. This is the primary name of this web server, but not necessarily the domain name the scripts are being accessed from; it might not be the same as `$_SERVER["HTTP_HOST"]`. Similarly, `$_SESSION["SERVER_ADMIN"]` holds the webmaster's email address that is set in the `ServerAdmin` directive.

The `SERVER_ADDR` and `SERVER_PORT` elements contain the IP address and port number of the machine the web server is running on. Checking `$_SERVER["REQUEST_METHOD"]` reveals whether a GET or POST method was used to pass values to the script.

Finally, if you are working on a shared web host or someone else's web server and want to see what web server software version that person is running, you can check `$_SERVER["SERVER_SOFTWARE"]`. This value is the same as the one transmitted in the `Server` header at the beginning of this lesson, and it is similar to the following:

```
Apache/1.3.29 (Unix) mod_gzip/1.3.26.1a PHP/4.3.9
mod_ssl/2.8.16 OpenSSL/0.9.7c
```

## Summary

In this lesson you have learned how to communicate with a web server. In the next lesson you will learn about filesystem access using PHP.



## LESSON 17

# Filesystem Access

*In this lesson you will learn how to access a web server's filesystem by using PHP and how to read and write files from within a script.*

## Managing Files

Let's examine how PHP allows you to work with files stored on a web server's hard disk.

### File Permissions

Before you can perform any filesystem access from PHP, you have to consider the permission settings for the files you want to work with. This section deals primarily with the file permissions system on Unix/Linux systems, but the same considerations apply to all platforms.

Usually your web server will be running under either the apache or nobody username, yet your web documents and PHP scripts will be owned by your actual system user. Unless the web server user has permissions to access your files, any access attempts will fail.

To grant to all users read-only access to a file, you use the `chmod` command, which sets the read flag (`r`) on the file for all users other than the current one (`o`):

```
$ chmod o+r filename
```

You can also set global read/write permissions on a file by using `chmod` as follows:

```
$ chmod o+rw filename
```

Refer to `man chmod` for full details and more examples.



If a file has global permissions, the web server will be able to access it. However, any other user on your system will also have full access to these files. In some situations you might prefer to change the ownership on a file to the apache user rather than grant global write access. The superuser does this by using the `chown` command:

```
chown apache filename
```

The PHP function `chmod` can also be used to alter file permissions. It takes two arguments: a filename and a mode. The mode argument can be either the string type, such as `o+rw`, or the numeric type, such as `0644`, as long as the number is prefixed with a zero.



**Using `chmod`** If you are familiar with the system command `chmod`, you may be used to giving a three-digit number as the mode argument. When run from the shell, `chmod` assumes that an octal value is given, whether or not it is prefixed with a zero; for example, `chmod 644` is the same as `chmod 0644`. In PHP, however, the leading zero is always required in the argument to `chmod`.

## Getting Information About a File

PHP provides a wide range of functions for getting information about a file on your system. The simplest, and one you may use often, is `file_exists`, which simply tells you whether there is a file with the given name argument. The `file_exists` function returns true if any item on the filesystem has the given name, even if it is a directory or a special file type. The argument may contain a path, as shown here:

```
if (file_exists("/home/chris/myfile")) {
 echo "The file exists";
}
else {
 echo "The file does not exist";
}
```

A number of other functions allow you to test certain attributes of a file. These are shown in Table 17.1.

**TABLE 17.1** Functions for Testing Attributes of a File

Function	Description
<code>is_executable</code>	Checks whether the file has the executable attribute.
<code>is_readable</code>	Checks whether the file is readable.
<code>is_writeable</code>	Checks whether the file is writeable.
<code>is_link</code>	Checks for a symbolic link.
<code>is_file</code>	Checks for a real file, not a link.

Yet other functions return information about the file itself. These are shown in Table 17.2.

**TABLE 17.2** Functions That Find Information About a File

Function	Description
<code>filetime</code>	Checks the time of the last file access, as a Unix timestamp.
<code>filectime</code>	Checks the time of file inode creation.
<code>filemtime</code>	Checks the time of the last modification to the file.
<code>fileowner</code>	Checks the user ID of the file owner.
<code>filegroup</code>	Checks the group ID of the file.
<code>fileinode</code>	Checks the inode number of the file.
<code>fileperms</code>	Checks the file's permission settings as an octal value (for example, 0644).
<code>filesize</code>	Checks the size of the file in bytes.
<code>filetype</code>	Checks the type of the file ( <code>fifo</code> , <code>char</code> , <code>dir</code> , <code>block</code> , <code>link</code> , or <code>file</code> ).

## Moving and Copying Files

Assuming that you have permission to do so, you can perform a file copy, move, or delete operation from PHP. The functions for these actions are `copy`, `rename`, and `unlink`, respectively.

The `copy` and `rename` functions take two arguments—the source and destination filenames—whereas `unlink` takes a single filename.



**File Paths** You should be particularly careful when performing file operations from PHP, particularly when deleting. You need to always make sure you know what the current working directory is, or give a full path to the target file.

## Working with Filenames

The functions `basename` and `dirname` provide an easy way to dissect a string into a filename and path, respectively. You might use these functions to find out the base filename when a full path is given or to find the pathname if you want a file created with other files in the same place as a known filename.

The `basename` function returns everything from the last slash character in the string to the end, whereas `dirname` returns the portion of the string before this slash.

The `realpath` function takes a pathname argument and returns its absolute pathname. Any symbolic links in the path are resolved to their actual location on disk, and any references to the current or parent directory using `.` or `..` are removed.

If you want to write to a temporary file, you can use the `tempnam` function to generate a unique temporary filename. It takes two arguments—a directory name and a filename prefix. The prefix argument is required but can be an empty string. The following statement generates a temporary filename in `/tmp` with no prefix:

```
$filename = tempnam("/tmp", "");
```

## Reading and Writing Files

Now let's see how to read and write files from PHP.

### Simple Methods for Reading and Writing Files

PHP provides some simple, high-level functions that can open a file and grab its contents or write data to a file in a single operation.

To read the contents of a file into a string variable, you use `file_get_contents`. The argument is a filename, which can contain a relative or absolute path. The following statement reads a file called `file.txt` into the variable `$data`:

```
$data = file_get_contents("file.txt");
```

An optional second Boolean argument can be set to `true` to search the include path for the given filename. You will see how to configure the include path in Lesson 23, “PHP Configuration.”

The function `file_put_contents` simply dumps the contents of a variable to a local file. Its arguments are the filename to write to and the data to write. The following statement writes the value of `$data` to `file.txt`:

```
file_put_contents("file.txt", $data);
```

### Lower-Level File Access

The functions `file_get_contents` and `file_put_contents` are high-level functions that perform a number of steps that can be done individually with lower-level PHP functions. Although in many cases reading the entire contents of a file or writing data to a file is the task you will want to perform, PHP provides a flexible way to interface with the filesystem.

All file access begins with a file handler, which is established with the `fopen` function. The arguments to `fopen` are a filename and the mode in which to open the file. The available modes are shown in Table 17.3.

**TABLE 17.3** File Mode Arguments to `fopen`

Mode	Description
<code>r</code>	Opens for reading only from the beginning of the file.
<code>r+</code>	Opens for reading and writing from the beginning of the file.
<code>w</code>	Opens for writing only; overwrites the old file if one already exists.
<code>w+</code>	Opens for writing and reading; overwrites the old file if one already exists.
<code>a</code>	Opens for writing only and appends new data to the end of the file.
<code>a+</code>	Opens for reading and writing at the end of the file.

Each file handler points to a position in the file. You can see from Table 17.3 that when a file is opened by using `fopen`, the handler always points to either the beginning or the end of the file. As you read or write using that handler, its pointer location moves, and subsequent actions take place from that point in the file.

Let's look at an example where you read the contents of a file a few bytes at a time. By calling `fopen` with the `r` mode argument, you create a read-only file handler that initially points to the start of that file.

The `fread` function reads a fixed number of bytes from a file handler. Its arguments are the file handler and the number of bytes to read. By performing this action in a loop, you can eventually read the entire contents of a file:

```
$fp = fopen("file.txt", "r");
while ($chunk = fread($fp, 100)) {
 echo $chunk;
}
```

This is a very compact statement that checks that `fread` has succeeded on each pass of the loop. When there is no more data to read, `fread` returns

FALSE. In fact, by using this loop to output the file to screen, you cannot tell from the result that it was actually done in smaller chunks.

An alternative to `fread` is `fgets`, which reads a line of the file at a time. The size argument to `fgets` has been optional since PHP 4.3 but is shown in these examples for completeness. No more data is read after a carriage return is reached in the file or the specified number of bytes has been read, whichever is sooner.

The following example uses `fgets` in a loop, assuming that no line in the file is more than 100 characters wide:

```
$fp = fopen("file.txt", "r");
while ($line = fgets($fp, 100)) {
 echo $line;
}
```



**Chopping Strings** Each line read by `fgets` ends with a newline character. If you want to exclude the newline, you can use the `rtrim` function on the string to remove it along with any trailing whitespace characters.

When you are finished with a file pointer, you should free up its resources by calling the `fclose` function:

```
fclose($fp);
```

## Random Access to Files

The file pointer does not have to be moved sequentially through a file; it can be reassigned to any position while the file handle is still open.

To find the current location of the file pointer, you use `ftell`. An integer is returned—the number of bytes from the start of the file:

```
$filepos = ftell($fp);
```

To send the file pointer to a specific location, you use the `fseek` function. The following statement places the file pointer 100 bytes from the start, using the file handler `$fp`:

```
fseek($fp, 100);
```

Most often you just want to return the file pointer to the beginning of the file. You could set `fseek` to position zero, or you could just use the built-in function `rewind`:

```
rewind($fp);
```

## Writing to a File Pointer

The complementary functions to `fgets` and `fread` to perform write operations on a file pointer are `fputs` and `fwrite`. These functions are actually identical to one another, with the newline character treated just like any other character as they are written to the file.

The following example opens a file and writes to it the current time:

```
$fp = fopen("time.txt", "w");
fwrite($fp, "Data written at ".date("H:i:s"));
fclose($fp);
```

Remember that the `apache` user needs to have write permissions on the directory in order to create a new file.

If you examine the new `time.txt` file, you will see that it does indeed contain the current time.

## Working with Data Files

One of the reasons you might need to access the filesystem from PHP is to load data from a structured file format into your script. One of the easiest file formats to use is comma-separated values (CSV).

Although it would appear to be fairly easy to read a line of the file at a time and call `explode` to break up the line where each comma appears, this would not work where data elements in the CSV file contain commas. If you export data from a spreadsheet, columns containing commas are

usually enclosed in quotes, so you need quite a complex rule to manipulate the data successfully.

Fortunately, PHP includes the function `fgetcsv`. It works in a similar way to `fgets`, except that an array is returned, containing one element for each comma-separated value in the list. The size argument to `fgetcsv` is optional as of PHP 5.

Often the first line of a CSV file contains the column headings. If you know that this is the case, you should discard the file line before processing the data file. The following example reads a comma-separated data file and dumps each record to screen by using `print_r`:

```
$fp = fopen("data.csv", "r");
while ($record = fgetcsv($fp, 1000)) {
 echo $chunk;
}
```



**Reading CSV Files** The `fgetcsv` function requires a line length argument, just like `fgets`. In the previous example, this has an arbitrary value of 1000, but you should ensure that whatever value you use is larger than the longest line in your data file.

You can also write data to a CSV file without having to manually encode the format. The `fputcsv` function takes a file handle and an array argument and writes a comma-separated list of the elements in the array.

The optional third and fourth arguments to `fputcsv` allow you to specify an alternate delimiter and enclosure characters, respectively; the defaults are the comma and double quote characters.

## Working with URLs

A powerful feature of PHP is its ability to deal with remote documents in the same way it deals with local files. It is possible to open a file handle or use the high-level filesystem access functions with a URL argument to read a web page from a PHP script.



The following statements are both valid:

```
$page = file_get_contents("http://www.sampublishing.com/");
$fp = fopen("http://www.sampublishing.com/", "r");
```

You cannot write to an HTTP URL by using `file_put_contents` or `fputs`, however.

## Working with Directories

Similarly to the way that `fopen` generates a file handle for accessing the contents of a file, you can create a directory handle to view the contents of a directory by using the `opendir` function.

There are just three calls that can be performed on a directory handle—`readdir`, `rewinddir`, and `closedir`—each of which takes a single resource argument.

Each call to `readdir` returns the next file from the directory. The order in which files are returned is the order in which they are stored by the filesystem and cannot be changed. The special items `.` and `..` (the current working directory and its parent) are always returned.

You use `rewinddir` to reset the directory handle to the beginning of the file list at any time, and you close the handle with `closedir` when you are finished with it.

To find the name of the current working directory, you use `getcwd`. No arguments are required, and the full path to the current directory is returned. To change directory, you use `chdir` with a relative or absolute path.

## Summary

In this lesson you have learned ways to read and write files on a web server's hard disk. In the next lesson you will learn how to execute local host commands from PHP.



## LESSON 18

# Host Program Execution

*In this lesson you will see how PHP allows you to execute programs on a host system and handle any output that is produced.*

## Executing Host Programs

PHP can call an external program that resides on a web server in a number of different ways. Let's look at them in the following sections.

### The **passthru** Function

The simplest way to run a host command and display the output to screen is by using the `passthru` function. The command passed in as an argument is executed on the web server, and any resulting output is sent to the browser.

The following is a simple example that works on both Unix/Linux and Windows systems:

```
passthru("hostname");
```

The command `hostname` is executed on the host system, and its output is displayed. The `hostname` command finds the system's hostname and displays it.

An optional second argument to `passthru` allows you to find the command's exit code. This is often useful if you want to find out whether a command succeeded—all programs should return an exit code of zero on successful completion—or to perform a test on a command that could have several return values.



**Command Output** Only the standard output stream is displayed in the web browser window, so you must redirect the `stderr` stream if you want to see warnings and errors produced by the host command.

For instance, you can use `passthru("cmd 2>&1")` on a Unix server with the Bourne shell.

The most common nonzero return values are 1 for a nonspecific error and 127, which means that the command you attempted to run could not be found. Other error codes specific to a particular program are usually documented.

The following example makes a system call to the `hostname` command and takes an action, depending on its return code:

```
passthru("hostname", $return);
switch ($return) {
 case 0: echo "Command completed successfully";
 break;
 case 127: echo "Command could not be found";
 break;
 default: echo "Command failed with code $return";
}
}
```

## Using Backticks

The backtick ( ``` ) character is a handy shortcut that can be used to indicate a system command for execution on the web server itself. A string contained between two backticks is executed, and the response produced by the host system is returned.

The following is equivalent to the `passthru` example, but it uses the backtick syntax:

```
echo `hostname`;
```

With backticks you are able to assign the result of a host command to a variable, as shown in the following statement:

```
$hostname = `hostname`;
```

In fact, the backtick characters can be used anywhere in a PHP script. They immediately interrupt program execution to call the host command, with the resulting values replaced into the script. The following example shows that the result of a host command can even be used within a condition:

```
if (chop(`hostname`) == "hal9000") {
 echo "Good evening, Dave";
}
```

Because the result from `hostname` ends with a carriage return—so that the output when run in a command shell looks tidy—the previous example uses `chop` to make sure that only non-whitespace characters are compared.



**Exit Codes** There is no way to obtain an exit code when using backticks. Instead, you should use the `exec` function, which works just like `passthru` but returns the command output as a string. The optional second argument can be used to grab the exit code.

## Building Command Strings

Commands are passed as arguments to `passthru` or `exec` or are simply strings contained in backticks. Therefore, you can build up a command string by using variables or in stages if you want.

Variable substitution takes place within a double-quoted command string, but if the string is enclosed in single quotes, any identifier prefixed with a dollar sign is treated as a shell variable.

Perhaps the strangest looking statement in PHP is one where you execute a command stored in a string by using backticks. This looks as follows:

```
`$cmd`;
```

The variable `$cmd` could contain any system command and, if you really don't care what the output from the command is, this is valid.

Note, however, that the terminating semicolon is required. A closing backtick closes the host command but does not terminate a PHP statement.

# The Host Environment

Now let's look at how PHP interacts with the web server's host environment.

## Detecting the Host Platform

Because different types of systems have different sets of host commands available, if you are writing a script that could potentially be executed on different platforms, it's useful to detect what kind of web server is being used.

The constant `PHP_OS` contains a string that represents the operating system. The most common reason for checking this is to find out whether a script is running on a Windows platform—after all, most Unix-like systems, and even Mac OS, behave in a very similar way.

The value of `PHP_OS` on a Windows web server could be `Windows`, `WINNT`, or `WIN32`, and in the future, other values may come into existence. Therefore, to test for a Windows platform, you should perform a non-case-sensitive comparison on the first three characters of the string. The following condition shows just one of the ways you can do this:

```
if (strtoupper(substr(PHP_OS, 0, 3)) == "WIN") { ... }
```



**Darwin** Be cautious to check only that the value of `PHP_OS` *begins* with `WIN`, as modern versions of Mac OS report themselves as Darwin.

## Environment Variables

The `$_ENV` super-global contains an element for each environment variable present. *Environment variables* are values from the underlying operating system, and those available to PHP are from the environment in which PHP and your web server is running.

The `PATH` environment variable provides your system with a list of locations to search for an executable program. Each location is checked in

turn until the program is found or there are no more locations left to try, when an error occurs.

Finding the current value of the path is as simple as using the following statement:

```
echo $_ENV["PATH"];
```

On a Unix/Linux system it may look like the following:

```
/bin:/usr/bin:/usr/X11R6/bin:/home/chris/bin
```

On a Windows system, however, it may look like this:

```
C:\WINDOWS\system32;C:\WINDOWS
```

Notice that the format is considerably different for the different operating systems. The Unix/Linux version uses colons to separate the locations and forward slashes in pathnames, and the Windows version uses semicolons and backslashes. For this reason, PHP provides the host-specific constants `DIRECTORY_SEPARATOR` and `PATH_SEPARATOR`, which enable you to find the appropriate symbols to use for each of these.

In many cases, resetting the `PATH` value is specific to the underlying platform; for instance, even if you use the correct `PATH_SEPARATOR` constant, `C:/WINDOWS` will not exist on a Linux server. However, this allows you to add the current working directory, or one relative to it, to the path fairly easily.

The following example adds the directory `bin`, relative to the current location, to the start of the system path:

```
$newpath = getcwd() . DIRECTORY_SEPARATOR . "bin" .
 PATH_SEPARATOR . $_ENV["PATH"];
putenv("PATH=$newpath");
```

The `putenv` function takes a single argument in which an environment variable is assigned its new value. This change is not permanent, and the new value is remembered only until the script ends.

## Time Zones

The TZ environment variable contains the server's time zone setting. By overriding this value, you can display the time in another part of the world without needing to know the correct offset or perform any date arithmetic.

Most major cities or regions of the world have a value for TZ that is easy to remember or work out (for instance, Europe/London, US/Pacific). It can also be a value relative to Greenwich Mean Time or some other common time zone, such as GMT-8 or EST. On most systems, you can find the available time zones by looking at the items in /usr/share/zoneinfo.

The script in Listing 18.1 displays the current time in several locations around the globe.

---

**LISTING 18.1** Using the TZ Environment Variable to Change Time Zone

---

```
<?php
$now = time();
$original_tz = $_ENV["TZ"];

echo "The time now is " . date("H:i:s", $now) . "
";

putenv("TZ=US/Pacific");
echo "The time on the US West Coast is " .
 date("H:i:s", $now) . "
";

putenv("TZ=Europe/Paris");
echo "The time in France is " . date("H:i:s", $now) . "
";

putenv("TZ=Australia/Sydney");
echo "The time in Sydney is " . date("H:i:s", $now) . "
";

putenv("TZ=Asia/Tokyo");
echo "The time in Tokyo is " . date("H:i:s", $now) . "
";

putenv("TZ=$original_tz");
?>
```

Note that Listing 18.1 begins by storing the current time zone value so that it can be restored after you are done changing the value.



**Storing the Time** The timestamp is saved to `$now` at the start of Listing 18.1 so that the same value can be passed to each date function. Although the second argument to `date` can be omitted, if it is omitted, it is possible that the script execution could take place as a second ticks over, which would produce confusing output.

## Security Considerations

Hopefully you have realized that having on your web server a script that is able to execute host program commands is not always a good idea. In fact, in Lesson 24, “PHP Security,” you will learn how you can use PHP’s Safe Mode to place restrictions on host program execution.

To end this lesson, you will learn how to make sure that host program execution is always done safely.

## Escaping Shell Commands

Consider the script in Listing 18.2, which creates a web form interface to the `finger` command.

### LISTING 18.2 Calling the `finger` Command from a Web Form

```
<FORM ACTION="finger.php" METHOD="POST">
<INPUT NAME="username" SIZE=10>
<INPUT TYPE="SUBMIT" VALUE="Finger username">
</FORM>
<?php
if ($_POST["username"]) {
 $cmd = "finger {"$_POST['username']}";
 echo "<PRE>" . ` $cmd ` . "</PRE>";
}
?>
```



If you run this script in your browser and enter a username, the `finger` information will be displayed.

However, if you instead enter a semicolon followed by another command—for instance, `;ls`—the `finger` command is run without an argument and then the second command you entered is executed. Similar trickery can be produced using other symbols, depending on your web server platform.

This is clearly not a good thing. You might think that only limited damage could be done through running processes as the same user as the web server; however, many serious exploits can take advantage of this behavior. A malicious user could issue a command such as `wget` or `lynx` to install a hostile program on your server's hard disk and then run it. This could be a rootkit to attempt to take advantage of other server vulnerabilities, or it could be a script to launch a denial-of-service attack by eating up all your system resources. However you look at it, giving anonymous users this kind of access to your web server is bad news.

To protect yourself against this kind of attack, you should use the `escapeshellcmd` function. Any characters that may be used to fool the shell into executing a command other than the one intended are prefixed with a backslash. This way, undesirable characters actually become arguments to the command.

To make Listing 18.2 safe, the statement that builds `$cmd` should be changed to the following:

```
$cmd = escapeshellcmd("finger ${_POST['username']}");
```

Now, entering `;ls` into the form will result in the command executed being `finger \; ls`—actually attempting to find users called `; or ls` on your system.

## Summary

In this lesson you have learned how to safely run host commands on your web server from PHP and deal with the output they produce. In the next lesson you will learn about database access in PHP using MySQL.



## LESSON 19

# Using a MySQL Database

*In this lesson you will learn how to access a MySQL database from PHP. The pairing of PHP and MySQL is so popular and powerful that it is quite rare to find PHP being used without MySQL—or at least some other database back end.*

## Using MySQL

This lesson assumes that you already have MySQL installed on your web server and that PHP has the MySQL module loaded. For information on installing MySQL, see <http://dev.mysql.com/doc/mysql/en/Installing.html>, and to learn how to activate MySQL support in PHP, refer to Lesson 23, “PHP Configuration.”



**Further Reading** To learn about the MySQL database, read *Sams Teach Yourself MySQL in 24 Hours* by Julie Meloni. Or for a quick SQL language guide, refer to *Sams Teach Yourself SQL in 10 Minutes* by Ben Forta.

PHP 5 introduced the `mysqli` extension, which can take advantage of new functionality in MySQL version 4.1 and higher and can also be used in an object-oriented manner. This book concentrates on the classic `mysql`

extension, because it is still the version offered by many web hosting providers and remains available in PHP 5.

Generally speaking, if you want to use `mysqli` instead of the classic `mysql` extension described in this lesson, most function names are prefixed `mysqli` rather than `mysql`, but they behave in a similar way. Refer to the online documentation at [www.php.net/mysqli](http://www.php.net/mysqli) for more information.

## Connecting to a MySQL Database

You can connect to a MySQL database by using the `mysql_connect` function. Three arguments define your connection parameters—the hostname, username, and password. In many cases, the MySQL server will be running on the same machine as PHP, so this value is simply `localhost`. A typical `mysql_connect` statement may look like the following:

```
$db = mysql_connect("localhost", "chris", "mypassword");
```



**Database Hostnames** Because MySQL uses host-based authentication, you must provide the correct hostname—one that allows a connection to be made. For instance, your MySQL server may be running on `www.yourdomain.com` but it might only be configured to accept connections to `localhost`.

Unless you are sure that the MySQL server is running somewhere else, the hostname to use is almost always `localhost`.

The `mysql_connect` function returns a database link identifier, which was assigned to `$db` in the previous example. This resource is used as an argument to the other MySQL functions.

Notice that the connection parameters given to `mysql_connect` do not include a database name. In fact, selecting the database is a separate step after you are connected to a MySQL server; to do it, you use the

`mysql_select_db` function. For example, the following statement selects `mydb` as the current database:

```
mysql_select_db("mydb", $db);
```



**Link Identifiers** The `$db` argument is not actually required in `mysql_select_db` and many other MySQL functions. If it is omitted, PHP assumes that you mean the most recently opened MySQL connection. However, it is good practice to always include the link identifier in MySQL function calls for clarity in your code.

After `mysql_select_db` has been called, every subsequent SQL statement passed to MySQL will be performed on the selected database.

When you are finished using MySQL in a script, you close the connection and free up its resources by using `mysql_close`, like this:

```
mysql_close($db);
```

## Executing SQL Statements

The function to pass a SQL statement to MySQL is `mysql_query`. It takes two arguments—the query itself and an optional link identifier.

The following code executes a `CREATE TABLE SQL` statement on the MySQL database for `$db`:

```
$sql = "CREATE TABLE mytable (col1 INT, col2 VARCHAR(10))";
mysql_query($sql, $conn);
```

If you run a script that contains these statements in your web browser and check your MySQL database, you will find that a new table called `mytable` has been created.

All types of SQL statement can be executed through `mysql_query`, whether they alter the data in some way or fetch a number of rows.

## Commands That Change a Database

Earlier in this lesson you saw an example of a `CREATE TABLE` statement. Other Data Definition Language (DDL) statements can be executed in a similar fashion, and, provided that no errors are encountered, they perform silently. You will learn about error handling later in this lesson.

When executing a `DELETE`, `INSERT`, or `UPDATE` statement—a subset of SQL known as the Database Manipulation Language (DML)—a number of rows in the table may be affected by the query. To find out how many rows are actually affected, you can use the `mysql_affected_rows` function. The following example shows how to do this with a simple `UPDATE` statement:

```
$sql = "UPDATE mytable SET col2 = 'newvalue' WHERE col1 > 5";
mysql_query($sql, $conn);
echo mysql_affected_rows($conn) . " row(s) were updated";
```

The argument to `mysql_affected_rows` is the database link identifier, and a call to this function returns the number of rows affected by the most recent query. The number of rows affected by this `UPDATE` statement is not necessarily the number of rows matching the `WHERE` clause. MySQL does not update a row if the new value is the same as the one already stored.



**Deleting All Rows** If you execute a `DELETE` statement with no `WHERE` clause, the number returned by `mysql_affected_rows` is zero, regardless of the number of rows actually deleted. MySQL simply empties the table rather than delete each row in turn, so no count is available.

## Fetching Queried Data

The `SELECT` statement should return one or more rows from the database, so PHP provides a set of functions to make this data available within a script. In order to work with selected data, you must assign the result from `mysql_query` to a result resource identifier, as follows:

```
$res = mysql_query($sql, $db);
```

You cannot examine the value of `$res` directly. Instead, you pass this value to other functions to retrieve the database records.

You can use the function `mysql_result` to reference a data item from a specific row and column number in the query result. This is most useful when your query will definitely only return a single value—for instance, the result of an aggregate function.

The following example performs a `SUM` operation on the elements in a table column and displays the resulting value onscreen:

```
$sql = "SELECT SUM(col1) FROM mytable";
$res = mysql_query($sql, $conn);
echo mysql_result($res, 0, 0);
```

The three arguments to `mysql_result` are the result resource identifier, a row number, and a column number. Numbering for both rows and columns begins at zero, so this example finds the first row in the first column in the result set. In fact, because of the nature of aggregate functions, you can be sure that there will always be only a single row and column in the result of this query, even if there are no records in the table. An attempt to access a row or column number that does not exist will result in an error.

The function `mysql_num_rows` returns the number of rows found by the query, and you can use this value to create a loop with `mysql_result` to examine every row in the result. The following code shows an example of this:

```
$sql = "SELECT col1, col2 FROM mytable";
$res = mysql_query($sql, $db);
for ($i=0; $i < mysql_num_rows($res); $i++) {
 echo "col1 = " . mysql_result($res, $i, 0);
 echo ", col2 = " . mysql_result($res, $i, 1) . "
";
}
```

With the query used in this example, because the column positions of `col1` and `col2` are known, you can use `mysql_result` with a numeric argument to specify each one in turn.



**Field Names** You can use a string for the column argument to `mysql_result`; in this case, you need to give the column's name. This behavior is particularly useful in `SELECT *` queries, where the order of columns returned may not be known, and in queries where the number of columns returned is not easily manageable.

## Fetching Full Rows of Data

PHP provides a convenient way to work with more than one item from a selected row of data at a time. By using `mysql_fetch_array`, you can create an array from the query result that contains one element for each column in the query.

When you call `mysql_fetch_array` on a result resource handle for the first time, an array is returned that contains one element for each column in the first row of the data set. Subsequent calls to `mysql_fetch_array` cause an array to be returned for each data row in turn. When there is no more data left to be fetched, the function returns `FALSE`.

You can build a very powerful loop structure by using `mysql_fetch_array`, as shown in the following example:

```
$sql = "SELECT col1, col2 FROM mytable";
$res = mysql_query($sql, $conn);
while ($row = mysql_fetch_array($res)) {
 echo "col1 = " . $row["col1"];
 echo ", col2 = " . $row["col2"] . "
";
}
```

Each row of data is fetched in turn, and in each pass of the loop, the entire row of data is available in the array structure, without any further function calls being necessary.

The array contains the row's data, using elements with both numeric and associative indexes. In the previous example, because you know that `col1` is the first column selected, `$row["col1"]` and `$row[0]` contain the same value.

This mechanism provides a method of sequential access to every row returned by a query. Random access is also available, and by using the function `mysql_data_seek`, you can specify a row number to jump to before the next `mysql_fetch_array` is performed.

To jump to the tenth row, you would use the following (remember that the numbering begins at zero, not one):

```
mysql_data_seek($res, 9);
```

It therefore follows that to reset the row position to the start of the data set, you should seek row zero:

```
mysql_data_seek($res, 0);
```

If you attempt to call `mysql_data_seek` with a row number that is higher than the total number of rows available, an error occurs. You should check the row number against the value of `mysql_num_rows` to ensure that it is valid.



**Seeking** To skip to the last row of a data set, you call `mysql_data_seek($res, mysql_num_rows($res) - 1)`. The number of the last row is one less than the total number of rows in the result.

However, the result can usually be achieved more easily by specifying reverse sorting in an `ORDER BY` clause in your SQL and selecting the first row instead.

## Debugging SQL

When a PHP call to the MySQL interface encounters a database error, the warnings displayed are not always as helpful as you might hope. In the following sections you will find out how to make the most of MySQL's error reporting to debug errors at the database level.



## SQL Errors

When there is an error in a SQL statement, it is not reported right away. You should check the return value from `mysql_query` to determine whether there was a problem—it is `NULL` if the query has failed for any reason. This applies to DDL and DML statements as well as to `SELECT` queries.

The following example tries to perform an invalid SQL statement (the table name is missing from the `DELETE` command):

```
$sql = "DELETE FROM";
$res = mysql_query($sql, $db);
if (!$res) {
 echo "There was an SQL error";
 exit;
}
```

If you want to find out why a call to `mysql_query` failed, you must use the `mysql_error` and `mysql_errno` functions to retrieve the underlying MySQL warning text and error code number. A link resource argument can be provided but is required only if you have two or more open MySQL connections in the script:

```
if (!$res) {
 echo "Error " . mysql_errno() . " in SQL ";
 echo "<PRE>$sql</PRE>";
 echo mysql_error();
 exit;
}
```



**Debugging SQL** When you're debugging SQL, it is useful to see the query that was attempted alongside the error message, particularly if your query uses variable substitutions. This is easy to do if the query is stored in a variable—such as `$sql` used throughout this lesson—rather than given directly as an argument to `mysql_query`.

If you do not trap SQL errors in script, PHP will continue to execute until an attempt is made to use the failed result resource. You will see an error message similar to the following if, for instance, `mysql_result` is called with an invalid `$res` value:

```
Warning: mysql_result(): supplied argument is not a valid
MySQL result resource in /home/chris/mysql.php on line 8
```

This error does not give any indication of what the problem was, or even when in the script it occurred. The line number given is the line of the `mysql_result` call, not `mysql_query`, so you have to search upward in the script to find the root of the problem.

## Connection Errors

If an error occurs during connection to a MySQL database, a PHP error is displayed onscreen, similar to the following, which were caused by an invalid password and a mistyped hostname, respectively:

```
Warning: mysql_connect(): Access denied for user
'root'@'localhost'
(using password: YES) in /home/chris/connect.php on line 3
```

```
Warning: mysql_connect(): Unknown MySQL server host
'local-host'
(1) in /home/chris/connect.php on line 3
```

These warnings are generated by PHP and are adequately descriptive. If you want, you can view the actual MySQL error message and error code by using `mysql_error` and `mysql_errno`.

For instance, if you have stopped PHP warnings from being displayed onscreen—you will learn how to do this in Lesson 23—it might be useful to output this information or write it to a log file. You can detect that the connection attempt failed because the link resource is `NULL`.

The following code checks that a connection has been successful before continuing, and it displays the reason for failure, if appropriate:

```
$db = mysql_connect("localhost", "chris", "mypassword");
if (!$db) {
 echo "Connection failed with error " .
 mysql_errno() . "
";
 echo "Warning: " . mysql_error();
 exit;
}
```



**Passwords** Neither the PHP warning nor the message from `mysql_error` contains the password used when the reason for failure is an invalid logon attempt.

## Summary

In this lesson you have learned how to use PHP's interface to the MySQL database system. In the next lesson you will learn how PHP can communicate with different database back ends by using a database abstraction layer.



## LESSON 20

# Database Abstraction

*In this lesson you will learn how to access different databases from PHP, using a single interface. Database abstraction is a very powerful technique; it allows you to write scripts for a nonspecific database back end, which you can then easily port simply by changing the connection parameters.*

## The PEAR DB Class

Many different database abstraction layers are available for PHP, but the one you will learn how to use in this lesson is the PEAR DB class. In Lesson 25, “Using PEAR,” you will find out more about PEAR—the PHP Extension and Application Repository—and some other useful classes it contains.

The DB class implements database abstraction, using PHP’s database extensions, and it currently supports the extensions shown in Table 20.1.

**TABLE 20.1** PHP Database Extensions supported by the PEAR DB Class

Extension	Database
dbase	dBase (.dbf)
fbsql	FrontBase
ibase	Firebird/Interbase
ifx	Informix
mysql	Mini SQL

Extension	Database
mssql	Microsoft SQL Server
mysql	MySQL
mysqli	MySQL 4.1 and higher
oci8	Oracle versions 7, 8, and 9
odbc	ODBC
pgsql	PostgreSQL
sqlite	SQLite
Sybase	Sybase



**DB Class Documentation** The online documentation for the PEAR DB class can be found at <http://pear.php.net/package/DB>.

## Installing the DB Class

To check whether the DB class is installed on your web server, you can run the following command to display a list of installed packages:

```
$ pear list
```

If you need to install the DB class, you run the following command:

```
$ pear install DB
```

Note that you need to be an admin to install a PEAR class, so if you are using a shared web hosting service, you might need to contact your system administrator.

Because the underlying PHP extensions are used, no additional database drivers are needed to communicate with each type of database from the DB class.



**Further Reading** To learn about the MySQL database, read *Sams Teach Yourself MySQL in 24 Hours* by Julie Meloni. Or, for a quick SQL language guide, refer to *Sams Teach Yourself SQL in 10 Minutes* by Ben Forta.

## Data Source Names

To connect to a database through the DB class, you need to construct a valid data source name (DSN), which is a single string that contains all the parameters required to connect and is formed in a similar manner to a URL that you might use to access a protected web page or FTP server.

The following DSN can be used to connect to a MySQL database running on localhost:

```
mysql://chris:mypassword@localhost/mydb
```

The components of this DSN are the database back-end type (mysql), username (chris), password (mypassword), host (localhost), and database name (mydb).

The full syntax definition for a DSN is as follows, and the components that it can be constructed from are given in Table 20.2.

```
phptype(dbsyntax)://username:password@protocol+hostspec/
database?option=value
```

**TABLE 20.2** Components of a DSN

Component	Description
phptype	Database back-end protocol to use (for example, mysql, oci8)
dbsyntax	Optional parameters related to SQL syntax; for ODBC, should contain the database type (for example, access, mssql)
username	Username for database login

Component	Description
password	Password for database login
protocol	Connection protocol (for example, tcp, unix)
hostspec	Host specification, either <i>hostname</i> or <i>hostname:port</i>
database	Database name
option	Additional connection options; multiple options are separated by &

As shown in the first example of connecting to MySQL, not every component of the DSN is required. The exact syntax depends on what information your database back end needs.

For instance, a connection to SQLite—which requires no username, password, or hostspec—would look like the following:

```
sqlite:///path/to/dbfile
```

On the other hand, a connection to a PostgreSQL server that is not running on a standard port number would require something more complex like this:

```
pgsql://username:password@tcp(hostname:port)/dbname
```



**Database Types** The database type values for the `phptype` argument are the values shown in the first column of Table 20.1.

## Using the DB Class

To begin using the DB class in scripts, you simply include it by using the following statement:

```
include "DB.php";
```

To make a connection to a database, you call the `connect` method on the `DB` class, giving your DSN as the argument:

```
$db = DB::connect($dsn);
```

The `$db` return value is an object on which the `DB` class methods can be invoked to perform different types of database operation.



**Database Objects** Note that you cannot create a new instance of a `DB` object by using the `new` keyword. You must call `DB::connect` to begin a new database session.

If the database connection fails, the return value is a `DB_Error` object, which you can analyze by using the `isError` and `getMessage` methods. The following code shows a database connection attempt with error checking:

```
$db = DB::connect($dsn);
if (DB::isError($db)) {
 echo "Connection error: " . $db->getMessage();
 exit;
}
```

The function `isError` returns `true` only if the argument passed is a `DB_Error` object, which indicates a problem of some kind with the database connection. You can then call the `getMessage` method on the `DB_Error` object to retrieve the actual error message from the database server.



**Connection Errors** `$db` is assigned an object value of some kind, whether or not the connection is successful. Its value will never be `NULL` or `FALSE`.



## Performing a Query

To execute a SQL query through the DB class, you use the query method. The return value depends on the type of query being executed, but in the event of any error, a DB\_Error object is returned, and the error can be detected and diagnosed in the same way as can connection errors.

The following example executes the query stored in \$sql with error checking:

```
$res = $db->query($sql);
if (DB::isError($res)) {
 echo "Query error " . $res->getMessage();
 exit;
}
```

If the query submitted is an INSERT, UPDATE, or DELETE statement, the return value is the constant DB\_OK. You can find out the number of rows affected by the statement by calling the affectedRows method on the database object itself, as shown in the following example:

```
$sql = "UPDATE mytable SET col2 = 'newvalue' WHERE col1 > 5";
$res = $db->query($sql);
echo $db->affectedRows(). " row(s) were affected";
```

## Retrieving Selected Data

If you issue a SELECT statement, the return value from the query is a DB\_Result object, which can then be used to access records from the result data set.

To view the number of rows and columns in the data set, you use the numRows and numCols methods, respectively, as in this example:

```
$sql = "SELECT * FROM mytable";
$res = $db->query($sql);
echo "Query found " . $res->numRows . " row(s) " .
 "and " . $res->numCols . " column(s)";
```

You can use the fetchRow method on a DB\_Result object to return a row of data at a time in an array structure. The result pointer is then increased

so that each subsequent call to `fetchRow` returns the next row of data, in order. The following code shows how you can fetch all the rows returned by a query by using `fetchRow` in a loop:

```
$sql = "SELECT col1, col2 FROM mytable";
$res = $db->query($sql);
while ($row = $res->fetchRow()) {
 echo "col1 = " . $row[0] . ", ";
 echo "col2 = " . $row[1] . "
";
}
```

In this example, elements of `$row` are numerically indexed, beginning at zero. Because the selected columns are specified in the `SELECT` statement, the order is known and you can be sure that `$row[0]` contains the value of `col1`.

You can give an optional argument to `fetchRow` to change the array indexing. The default, which causes a numerically indexed array to be created, is `DB_FETCHMODE_ORDERED`. By specifying `DB_FETCHMODE_ASSOC`, you cause an associative array to be created, using the column names as keys.

You could use the following loop to reproduce the previous example, instead using an associative array of the fetched values:

```
while ($row = $res->fetchRow(DB_FETCHMODE_ASSOC)) {
 echo "col1 = " . $row["col1"] . ", ";
 echo "col2 = " . $row["col2"] . "
";
}
```

If you prefer, you can use the `fetchRow` method to create an object structure rather than an array, by passing the argument `DB_FETCHMODE_OBJECT`. The following loop is equivalent to the previous two examples, but it uses the object method:

```
while ($row = $res->fetchRow(DB_FETCHMODE_OBJECT)) {
 echo "col1 = " . $row->col1 . ", ";
 echo "col2 = " . $row->col2 . "
";
}
```



**Fetch Modes** Which fetch mode you use usually depends on your preference. The associative array and object structures usually create mode-readable code. However, where optimal performance is essential, you should try to use `DB_FETCHMODE_ORDERED`.

## Query Shortcuts

If a query will return only a single row and column—for instance, the result of a single aggregate function—you can use the `getOne` method to quickly execute the query and return the result. A string query argument is supplied, and the database result is returned:

```
$sum = $db->getOne("SELECT sum(col1) FROM mytable");
```

Other shortcut methods are available, including `getRow`, to execute a query and return a whole row, and `getAll`, to execute a query and return the entire dataset as an array. Refer to the documentation for a full list of functions.

## Database Portability Issues

By using database abstraction, you can write database-driven code that should be able to work with a multitude of back ends, simply by changing the DSN used to connect to the database.

However, not all database systems are the same, so you need to consider the design of database tables and SQL statements in order to make sure that your code is as widely supported as possible.

The most important consideration is to make sure that your SQL is written for the lowest common subset of the SQL language available to all the database back ends you want to be compatible with. For example, SQL that contains subqueries will not work with MySQL 4.0 or earlier. Similarly, you should avoid SQL commands that are specific to certain database systems, such as `LIMIT` or `CREATE SEQUENCE`.

## Portability Modes

The DB class includes some portability mode settings that can ease the transition from one database back end to another. These modes are indicated by a series of constants, shown in Table 20.3, that you can set by using the `setOption` method with the required options, combined with a logical OR operator. The following statement shows an example:

```
$db->setOption('portability',
 DB_PORTABILITY_ERRORS | DB_PORTABILITY_NUMROWS);
```

**TABLE 20.3** Portability Mode Constants

Constant	Mode
DB_PORTABILITY_ALL	Turns on all portability features
DB_PORTABILITY_NONE	Turns off all portability features
DB_PORTABILITY_DELETE_COUNT	Forces a count to take place in a DELETE statement with no WHERE clause, with WHERE 1=1 appended to the statement
DB_PORTABILITY_ERRORS	Increases consistency of error reporting between different database systems
DB_PORTABILITY_LOWERCASE	Forces conversion of names of tables and columns to lowercase
DB_PORTABILITY_NULL_TO_EMPTY	Converts fetched NULL values to empty strings; some databases do not distinguish these
DB_PORTABILITY_NUMROWS	Enables the <code>numRows</code> method to work correctly in Oracle
DB_PORTABILITY_RTRIM	Forces trailing whitespace to be trimmed from fetched data

## Working with Quotes

You can use the DB method `quoteSmart` to enclose a value in quotation marks so that it can be safely inserted into a column. String values are enclosed in quotes, and any characters that need to be delimited are automatically taken care of.

The following example builds a SQL statement by using `quoteSmart` to ensure that the apostrophe in the string does not interfere:

```
$sql = "INSERT INTO phrases (phrase) " .
 "VALUES (" . $db->quoteSmart($text) . ")";
```

The following is the value of `$sql` when the previous statement is executed, using the MySQL driver:

```
INSERT INTO phrases (phrase)
VALUES ('Let\'s get ready to rumble')
```

The output and the delimiting rules used depend on the database you are connected to.

## Sequences

The way sequences are implemented in different database engines varies considerably. In MySQL, for instance, you use the `AUTO_INCREMENT` attribute on a table column, and in SQL Server it is called an `IDENTITY` field. In Oracle you use `CREATE SEQUENCE` to create a database object that tracks the sequence value independently of any table.

The DB class uses its own set of functions to manage sequences so that using any kind of auto-incrementing field does not tie your code to one particular database back end.



**Sequences** If your back-end database supports `CREATE SEQUENCE`, that functionality will be used. Otherwise, the DB class emulates the sequence by using a table that holds the sequence value, and it performs an increment each time the sequence is accessed.

To create a new sequence, you use the `createSequence` method on a database object, with a unique sequence identifier. After the sequence has been created, the `nextId` method can be called with that identifier to return the next sequential value.

The following example creates a sequence called `order_number` and displays the first sequence value:

```
$db->createSequence("order_number");
echo $db->nextId("order_number");
```

Subsequent calls to `nextId` for this sequence return incremental values.

To drop a sequence when you no longer have a use for it, you call the `dropSequence` method.

## Query Limits

MySQL implements the `LIMIT` keyword in SQL statements, which you can use to restrict the number of rows returned by a query. This is non-standard SQL, and other database systems do not include this feature.

The `DB` class includes the `limitQuery` method, which you can use to emulate the `LIMIT` clause in a SQL statement for maximum compatibility. This method is called in the same way as a query, but it takes two additional arguments: to specify the starting row and number of rows to be returned.

The following example returns five rows from the query's data set, beginning at row 11 (where row numbering begins at zero):

```
$res = $db->limitQuery("SELECT * FROM mytable", 10, 5);
```

## Summary

In this lesson you have learned how to write database-driven PHP scripts by using a database abstraction layer. In the next lesson you will learn how to write and run command scripts by using PHP.

## LESSON 21

# Running PHP on the Command Line



*Although PHP was conceived as a tool for creating dynamic web pages, because the PHP language is very powerful, it has also become popular for writing command scripts and even desktop programs.*

*In this lesson you will learn how to write PHP for use from the command line and create your own command scripts.*

## The Command-Line Environment

In order to use PHP from the command line, you need to have a PHP executable installed on your system. When running in a web environment, PHP is usually installed as an Apache module, but it is also possible to build a standalone program called `php` that can be used as a command-line interface (CLI).

## Differences Between CLI and CGI Binaries

Beginning in version 4.2, PHP started to differentiate between binary programs intended for CGI and those for CLI use. Both executables provide the same language interpreter, but the CLI version includes the following changes to make it more suitable for command-line use:

- No HTTP headers are written in the output.
- Error messages do not contain HTML formatting.
- The `max_execution_time` value is set to zero, meaning that the script can run for an unlimited amount of time.

To find out whether a php binary is a CGI or CLI version, you can run it with the `-v` switch to see its version information. For instance, the following output is from the CLI version PHP 5.0.3:

```
PHP 5.0.3 (cli) (built: Dec 15 2004 08:07:57)
Copyright (c) 1997-2004 The PHP Group
Zend Engine v2.0.3, Copyright (c) 1998-2004 Zend Technologies
```

The value in parentheses after the version number indicates the Server Application Programming Interface (SAPI) that is in use. You can also find this value dynamically in a script by looking at the return value from the function `php_sapi_name`.



**Windows Distributions** The Windows distributions of PHP 4.2 included two binaries—the CGI version was called `php.exe`, and the CLI binary was `php-cli.exe`. For PHP 4.3, both were called `php.exe`, but they were found in folders called `cli` and `cgi`, respectively.

For PHP 5 and higher, `php.exe` is the CLI version, and now the CGI binary is named `php-cgi.exe`. A new `php-win.exe` CLI binary is also included that runs silently—that is, the user doesn't need to open a command prompt window.

## PHP Shell Scripts on Linux/Unix

On a Linux/Unix platform, a *shell script* is simply a text file that contains a series of instructions that are to be processed by a specific language interpreter. The simplest shell interpreter is the Bourne Shell, `sh`, although these days it has been superseded by the Bourne Again Shell, `bash`, which is fully compatible with `sh` but also includes other useful features.

Because the command language available in most command shells is very restrictive and often requires calls to external programs, PHP is not only a more powerful language, suitable for many tasks, but its built-in features also usually give better performance than the standard system tools.





**PHP Location** The `php` executable is usually installed to `/usr/local/bin` or `/usr/bin`, depending on whether it was installed from source or a binary package, but your actual location may vary. Try typing `which php` to find the location if you do not know it.

All shell scripts must begin with the characters `#!/`, followed by the path to the command interpreter that is to be used. For a traditional shell script, this would look like the following:

```
#!/bin/sh
```

However, for a PHP script, the first line would be

```
#!/usr/local/bin/php
```



**Hash Bang** The most widely used pronunciation for the character sequence `#!/`, found at the start of a shell script, is “hash bang,” although sometimes it is also referred to as “shebang.”

The file permissions on a shell script must allow the file to be executed. To set execute permission for the owner of the file, you use the following command:

```
$ chmod u+x myscript.php
```

If your script is to be run by any system user, the command to set global execute permission is as follows:

```
$ chmod a+x myscript.php
```

If the execute bit is not set, you can still run a file that contains a series of PHP commands through the PHP interpreter by invoking `php` with a file-name argument. The following two commands are identical to one another (the `-f` switch can be used for clarity but is not required):

```
$ php myscript.php
$ php -f myscript.php
```



**Script Names** There are no naming requirements for any type of shell script. However, it is useful to retain the `.php` extension so that the filename indicates a PHP script. Bourne shell scripts sometimes have the file extension `.sh` but often are command names with no file extension at all.

## PHP Command Scripts on Windows

Windows does not allow an alternate command interpreter to be used in a batch script, so to execute a PHP script under Windows, you have to pass a filename argument to `php.exe`. The `-f` switch is optional, so the following two commands are identical to one another:

```
> php.exe myscript.php
> php.exe -f myscript.php
```



**Batch Scripts** If you want, you can create a simple batch script to invoke `php.exe` with the correct filename argument so that you can run your script by using a single command.

To do so, you create a file named `myscript.bat` that contains the command `php.exe`, followed by your script name. You can then run that script by simply entering `myscript` at the command prompt.

## Embedding PHP Code

Just as when it is used in the web environment, PHP code in a command script needs to be embedded. Any text that does not appear inside `<?php` tags is sent straight to the output.

Because you usually want to create a script that is entirely made up of PHP code, you must remember to begin every PHP shell script with a

<?php tag. However, the embedded nature of PHP means you could create a PHP script that generates only certain elements within a largely static text file.

## Writing Scripts for the Command Line

The PHP language provides certain functionality that is particularly useful for writing command-line scripts. You will rarely, if ever, use these features in the web environment, but they are described in the following sections.

### Character Mode Output

When you're producing web output, you use the `<br>` tag to produce a simple line break in the output. When it is sent to a web page, the newline character, `\n`, causes a line break in the HTML source, but it is not visible in the rendered web page.

Command-line scripts, however, produce text-only output, so you must use the newline character to format your output. If your script produces any output, you should always include `\n` after the last item has been displayed.

You can also take advantage of the fixed-width character mode when running command-line scripts—for instance, by spacing output into columns. The `printf` function allows you to use width and alignment format characters, which have no effect on HTML output unless they're contained in `<PRE>` tags. For more information, refer to Lesson 6, “Working with Strings.”

### Command-Line Arguments

You can pass arguments to a shell script by simply appending them after the script name itself. The number of arguments passed can be found in the variable `$argc`, and the arguments themselves are stored in a numerically indexed array named `$argv`.



**Arguments** The identifier names `argc` and `argv` are used for historic reasons. They originated in C and are now widely used in many programming languages.

In PHP `$argc` is assigned for convenience only; you could, of course, perform `count($argv)` to find out how many arguments were passed to the script.

The `$argv` array will always contain at least one element. Even if no additional arguments are passed to the script, `$argv[0]` will contain the name of the script itself, and `$argc` will be 1.

The script in Listing 21.1 requires exactly two arguments to be passed. Otherwise, an error message appears, and the script terminates. The output produced shows which of the two arguments is greater.

### LISTING 21.1 Using Command-Line Arguments

```
#!/usr/local/bin/php
<?php
```

```
if ($argc != 3) {
 echo $argv[0].": Must provide exactly two arguments\n";
 exit;
}

if ($argv[1] < $argv[2]) {
 echo $argv[1] . " is less than ". $argv[2] . "\n";
}
elseif ($argv[1] > $argv[2]) {
 echo $argv[1] . " is greater than ". $argv[2] . "\n";
}
else {
 echo $argv[1] . " is equal to ". $argv[2] . "\n";
}
?>
```

Notice that the initial condition in Listing 21.1 checks that the value of `$argc` is 3; there must be two arguments, plus the script name itself in `$argv[0]`. In fact, `$argv[0]` is output as part of the error message. This is a useful technique for ensuring that the actual script name is shown, whatever its name happens to be.

## Input/Output Streams

Although it is possible to read and write directly to the standard input, output, and error streams in the web environment, doing so is much more useful in command-line scripts.

Stream access is performed using the same set of functions as for file access: You simply open a file pointer to the appropriate stream and manipulate it in the same way.

The stream identifiers look like URLs—remember that PHP also allows you to open URLs by using the file access functions—constructed of `php://` followed by the name of the stream. For instance, to open the standard input stream for reading, you use the following command:

```
$fp = fopen("php://stdin", "r");
```

However, because stream access is common in command-line scripts, PHP provides a shortcut. The constants `STDIN`, `STDOUT`, and `STDERR` provide instant access to an opened stream without requiring a call to `fopen`.

The script in Listing 21.2 uses all three standard streams. It reads data from standard input and capitalizes the letters it contains by using `strtoupper`. If the input data contains non-alphanumeric characters, a warning is sent to the standard error stream as well.

---

### LISTING 21.2 Reading and Writing Standard Streams

---

```
#!/usr/local/bin/php
<?php
```

```
while (!feof(STDIN)) {
 $line++;
 $data = trim(fgets(STDIN));

 fputs(STDOUT, strtoupper($data)."\n");
 if (!ereg("^[[:alnum:]]+$", $data)) {
 fputs(STDERR,
 "Warning: Invalid characters on line $line\n");
 }
}
?>
```

If you run this script from the command line, it waits for you to type data a line at a time, and it returns the uppercase version after each line is entered. The advantage of using the standard input stream is that you can redirect input from another source.

To pass the contents of the file `myfile` into a script named `myscript` and have the output written to `outfile`, for instance, you would run the following command:

```
$ myscript < myfile > outfile
```

With the example in Listing 21.2, `outfile` would contain only the uppercase data. The warning messages produced would continue to be displayed to screen, unless you also redirected standard error.

To summarize, the constants and stream identifiers available in command-line PHP are shown in Table 21.1.

**TABLE 21.1** Stream Access for CLI PHP

| Constant | Identifier   | Stream          |
|----------|--------------|-----------------|
| STDIN    | php://stdin  | Standard input  |
| STDOUT   | php://stdout | Standard output |
| STDERR   | php://stderr | Standard error  |

## Creating Desktop Applications

PHP is such a powerful language that you can even use it to create desktop applications. Furthermore, because PHP is an interpreted language, any such applications are likely to be highly portable.

The PHP-GTK extension implements an interface to the GIMP window toolkit, GTK+. This provides PHP developers with the ability to create applications with a graphical front end that includes windows, menus, buttons, and even drag-and-drop functionality.

Creating a complex graphical application is beyond the scope of this book. If you are interested in learning more about PHP-GTK, however, see <http://gtk.php.net>.

## Summary

In this lesson you have learned how to write PHP scripts by using the CLI. In the next lesson you will learn techniques for error handling and debugging in PHP.



## LESSON 22

# Error Handling

*In this lesson you will learn how to deal with errors in PHP scripts effectively and how to debug code that does not work as you expect them to.*

## Error Reporting

PHP has a configurable error reporting system that you can set to be just as pedantic you want it to be about code. By default, the strictest mode is not enabled, and, in most cases, you get a warning message only when an imperfection has a good chance of affecting the intended purpose of your script.

## Changing Error Levels

To change the error reporting level, you use the `error_reporting` function with a value that is made up of the constants shown in Table 22.1.

**TABLE 22.1** Error Reporting Constants

| Constant               | Description                                                     |
|------------------------|-----------------------------------------------------------------|
| <code>E_ERROR</code>   | Indicates a fatal runtime error. Script execution is halted.    |
| <code>E_WARNING</code> | Issues runtime warnings. Non-fatal; script execution continues. |
| <code>E_PARSE</code>   | Indicates compile-time parsing errors.                          |
| <code>E_NOTICE</code>  | Issues runtime notices, which may or not indicate errors.       |



| Constant                       | Description                                                                                   |
|--------------------------------|-----------------------------------------------------------------------------------------------|
| <code>E_CORE_ERROR</code>      | Issues a fatal error generated internally by PHP.                                             |
| <code>E_CORE_WARNING</code>    | Issues a warning generated internally by PHP.                                                 |
| <code>E_COMPILE_ERROR</code>   | Issues a fatal error generated by the Zend engine.                                            |
| <code>E_COMPILE_WARNING</code> | Issues a warning generated by the Zend engine.                                                |
| <code>E_USER_ERROR</code>      | Issues a user-generated error message, triggered by <code>trigger_error</code> .              |
| <code>E_USER_WARNING</code>    | Issues a user-generated warning, triggered by <code>trigger_error</code> .                    |
| <code>E_USER_NOTICE</code>     | Issues a user-generated notice, triggered by <code>trigger_error</code> .                     |
| <code>E_ALL</code>             | Issues all errors and warnings except <code>E_STRICT</code> .                                 |
| <code>E_STRICT</code>          | Issues all errors and warnings, plus PHP suggests code changes to improve code compatibility. |

You combine these constants by using bitwise operators to create a bit-mask that represents the desired level. The default value is `E_ALL & ~E_NOTICE`, which means that all errors and warnings are displayed except for `E_STRICT`, which is not covered by `E_ALL`, and `E_NOTICE`.

To set the error reporting level so that all warnings and notices are displayed, you use the following command:

```
error_reporting(E_ALL);
```

The type of notices that are not displayed by default are not life-threatening and do not affect the normal execution of a script.

The `E_NOTICE` error level can be very useful during script development because it alerts you to the use of undefined variables. Although using `E_NOTICE` does not cause an error in your script, seeing these warnings can often alert you to an identifier name that was mistyped and should be referencing a previously declared value.

When you use `E_NOTICE`, you are also warned about certain points of coding style. For instance, array key identifiers should be enclosed in quotation marks, but a sloppy programmer might use `$array[key]`. PHP first assumes that `key` is a constant, but if no constant with that name is defined, it also tries to use it as a string key name. With `E_NOTICE` enabled, you are advised of this ambiguity.

The `E_STRICT` level is new in PHP 5, and it is useful if you want to make sure your code is up-to-date. It warns you if you use deprecated functions that have been left in the PHP language for backward compatibility.

You can also set the error reporting level in the `php.ini` file, or per directory, by using `.htaccess`, using the `error_reporting` directive. Lesson 23, “PHP Configuration,” describes how to use these features.



**Displaying Errors** The `log_errors` and `display_errors` configuration directives allow you to choose whether errors and warnings are displayed to screen or written to a log file.

On a production web site, you should consider whether displaying error messages onscreen is a security risk because it could convey information about your system to an intruder or a competitor.

## Custom Error Handlers

PHP allows you to define a custom function that is called whenever an error is encountered. This replaces the default action of displaying the error message to screen or logging to a file, depending on your configuration.

You use the `set_error_handler` function to declare which function should be used as the custom error handler. Its first argument is the function name, and you can give an optional second argument that contains a bitmask that specifies which error levels should be handled by that function.

For example, to use the function `myhandler` to trap all `E_WARNING` and `E_NOTICE` errors, you use the following command:

```
set_error_handler("myhandler", E_WARNING & E_NOTICE);
```

The user-defined error handler function requires two parameters: an error code number and a string error message. The error code value can be compared to the constants in Table 22.1 to find out what type of error occurred. You can include three more optional parameters if you want to process the information they pass to the function: the filename, line number, and context when the error occurred.

The example in Listing 22.1 declares a custom error handler function that logs all errors to a MySQL database table.

---

**LISTING 22.1** Writing a Custom Error Handler

---

```
<?php

function log_errors($errno, $errstr, $errfile, $errline) {

 $db = mysql_connect("localhost", "loguser", "logpassword");
 mysql_select_db("test", $db);
 $errstr = mysql_escape_string($errstr);
 $sql = "insert into php_log
 (errno, errstr, errfile, errline)
 values
 ('$errno', '$errstr', '$errfile', '$errline')";

 $res = mysql_query($sql, $db);
}

set_error_handler("log_errors");

// Assigning an undefined variable will raise a warning
$a = $b;
?>
```

You create the database table required to log these errors by using the following SQL statement:

```
CREATE TABLE php_log (
 error_timestamp timestamp,
 errno int,
 errstr text,
 errfile text,
 errline int
);
```

Note that this example does not use the optional fifth parameter that passes in the context. The context is passed as an array that contains the contents of every variable—both local values and system super-globals—in the script at the time the error occurred.

Although this information can sometimes be useful when you're debugging, it is a lot of information to store to a log file or table. Furthermore, because the value passed is an array, you need to pass `$errcontext` through the `serialize` function in order to store it to a database or text file.



**Context Data** Most likely you are interested in only a small part of the context data passed to your error handler function, so you can have the function extract as much information as is necessary and discard the rest.

## Raising User Errors

You can use the `trigger_error` function to raise an error on demand. If a custom error handler has been defined, it handles this user error. Otherwise, the default PHP error handler takes the appropriate action.

You should pass an error message string to the `trigger_error` function, and you can optionally give an error type constant. If no error type is given, `E_USER_NOTICE` is used.

For example, to raise a user notice type, you use this statement:

```
trigger_error("Some kind of error happened");
```

The error message displayed will look similar to the following:

```
Notice: Some kind of error happened in /home/chris/error.php
on line 3
```

To raise an error with the same message text but as a fatal error, you use this statement instead:

```
trigger_error("Some kind of error happened", E_USER_ERROR);
```



**Error Types** You have to use the `E_USER_ERROR` and `E_USER_WARNING` types, not `E_ERROR` and `E_WARNING`, for user errors. An `E_USER_ERROR` type error is still treated as a fatal error, however, and script execution ends immediately when it occurs.

## Logging Errors

You can accomplish most simple error logging requirements by using the `error_log` function. You can use this function to write an error message to the web server log file or some other local file, send it via email, or transmit it to a remote debugging service.

The `error_log` function takes the following arguments:

```
error_log($message, $message_type, $destination,
 $extra_headers);
```

Only the message argument is required, and the default action is to write this message text to the usual PHP log file. In most cases, this is your web server's log file.

The `message_type` argument specifies a number that determines what type of destination is supplied. Its possible values are shown in Table 22.2.

**TABLE 22.2** message\_type Argument Values in error\_log

| Value | Description                                                                                                                                                       |
|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | The message will be written to the default web server log file.                                                                                                   |
| 1     | The message will be sent via email; destination contains the address to send to, and extra_headers contains optional email headers.                               |
| 2     | The message will be sent to a remote debugging service; destination contains the remote hostname. Note that remote debugging is not available in PHP 4 and later. |
| 3     | The message will be appended to a local file; destination contains the filename and path.                                                                         |

For example, to send an error message via email, you might use the following statement:

```
error_log("An error occurred in your script", 1,
 "chris@lightwood.net",
 "From: PHP Script Error <errors@yoursite.com>");
```

## Suppressing Errors and Warnings

PHP allows you to suppress warning messages in your script. You can either turn off warnings completely or select individual commands for which any errors will not be displayed.

## The Error Suppression Operator

If you want to stop a warning message from appearing for a particular statement only, you can use the @ symbol to silence it. You might want to do this so that your custom error messages are displayed.

For example, when you connect to a MySQL database, PHP raises its own error if the connection fails. You can also detect the failure by checking whether a valid database resource handle was returned. The following

code uses the @ symbol to suppress the PHP error message so that only your message is displayed onscreen:

```
$db = @ mysql_connect("localhost", "username", "password");
if (!$db) {
 echo "Database connection failed";
 exit;
}
```

You can place the @ symbol before any expression in PHP. It causes any error messages generated as a result of that expression being evaluated to be ignored.

In the preceding example, the expression being silenced is the database connection attempt. As a general rule, if something in PHP has a value, it can be prefixed with the @ symbol. You cannot prepend an @ symbol to a language construct such as a function definition or conditional statement.



**Parsing Errors** The @ operator does not hide error messages caused by parsing errors in script.

The following statement is also valid, although it does not make it clear that you are expecting the error message to originate with mysql\_connect:

```
@ $db = mysql_connect("localhost", "username", "password");
```



**Error Suppression** If you have used set\_error\_handler to specify a custom error handler in your script, the @ operator will have no effect.

## Preventing Error Display

The configuration directive display\_errors can be set to Off in php.ini to prevent any errors from being displayed onscreen. You will learn how to change the value of php.ini settings in Lesson 23.

If you choose to prevent error display for your whole website this way, you should turn on the `log_errors` setting so that errors and warnings are written to a file; otherwise, you will have no way of knowing about potential problems. You should not consider turning `display_errors` off while a website is in development.

## Summary

In this lesson you have learned how to detect and handle errors in PHP scripts. In the next lesson you will learn about the various PHP settings that you can configure to suit your particular needs.



# LESSON 23

## PHP Configuration



*In this lesson you will learn how to configure global PHP settings at run-time, using the `php.ini` file, and per-directory settings, using `.htaccess`.*

### Configuration Settings

PHP allows you to tune many aspects of its behavior by using a set of configuration directives. These directives can be global for your entire web server, or you can make local changes that apply only to certain scripts.

#### Using `php.ini`

PHP's configuration file is named `php.ini`. Its location is set at compile time; by default, it is located in `/usr/local/lib/php.ini` on Linux/Unix servers and `C:\WINDOWS\php.ini` on Windows systems.

The `php.ini` file contains a list of configuration directives and their values, separated by equals signs. The default `php.ini` file distributed with PHP is well documented, with plenty of comments. Any line that begins with a semicolon is considered a comment, and sections of the file are broken up using headings in square brackets, which the compiler also ignores.

Listing 23.1 shows an extract from an unchanged `php.ini` file for PHP 5 that contains the log settings. As you can see, for many setting changes, you do not even need to refer to the online documentation.

**LISTING 23.1** An Extract from `php.ini`

```
; Print out errors (as a part of the output). For
; production web sites,
; you're strongly encouraged to turn this feature off,
; and use error logging
; instead (see below). Keeping display_errors enabled
; on a production web site
; may reveal security information to end users, such as
; file paths on your Web
; server, your database schema or other information.
display_errors = On

; Even when display_errors is on, errors that occur
; during PHP's startup
; sequence are not displayed. It's strongly recommended
; to keep
; display_startup_errors off, except for when debugging.
display_startup_errors = Off

; Log errors into a log file (server-specific log, stderr,
; or error_log (below))
; As stated above, you're strongly advised to use error
; logging in place of
; error displaying on production web sites.
log_errors = Off

; Set maximum length of log_errors. In error_log information
; about the source is
; added. The default is 1024 and 0 allows to not apply any
; maximum length at all
.
log_errors_max_len = 1024
```



**True or False** Boolean values in `php.ini` can be set to true (that is, on or yes) or false (that is, off, no, or none). These values are not case-sensitive.

When it runs as a web server module, `php.ini` is read when the web server process starts, and changes made to the configuration file do not take place until the web server is restarted.

If your web server runs PHP as a CGI binary, the `php.ini` settings are loaded each time a script is run because a new php process is started. Similarly, command-line PHP loads the settings from `php.ini` each time a script is run.

## Alternate `php.ini` Files

You can create separate `php.ini` files to apply for the different ways PHP can be run. If you create a file named `php-SAPI.ini` (replacing *SAPI* with the a valid SAPI name), that file is read instead of the global `php.ini`.

For instance, to provide a different set of directives only for command-line PHP, you would use a configuration file named `php-cli.ini`. For the Apache web server module, the filename would be `php-apache.ini`.

On a Windows system, a `php.ini` file in the Apache installation directory is used before one in `C:\WINDOWS`. This allows you to maintain different PHP settings for multiple web servers on the same machine.

To force the use of a particular configuration file, you must invoke php with the `-c` option. In a shell script, you might change the first line to the following to force a custom configuration file to be used only for that script:

```
#!/usr/local/bin/php -c /path/to/php.ini
```

## Per-Directory Configuration

Apache web server allows you to use a per-directory configuration file named `.htaccess` to supply custom web server directives. PHP supports the use of `.htaccess` to override the global settings from `php.ini`.

To give a new value for a PHP setting, you use `php_value` followed by the directive from `php.ini` and the new value. The following line in an `.htaccess` file gives a new value for `max_execution_time` of 60 seconds:

```
php_value max_execution_time 60
```



**Using .htaccess** Be aware of the syntax difference when changing configuration settings in `php.ini` and `.htaccess`. In `php.ini` there must be an equals sign between the directive name and the value. In `.htaccess` the value follows the directive name, with no equals sign.

Changes made in `.htaccess` apply only to the directory in which it resides and its subdirectories. Any settings in `.htaccess` override the global `php.ini` as well as any settings made in an `.htaccess` file in a parent directory.

## Dynamic Configuration

You can alter values of directives set in `php.ini` on-the-fly by using the `ini_set` function. It takes two arguments: the directive name and the new value. When you change a setting by using `ini_set`, the return value is the previous setting for that directive.

The following example changes the `memory_limit` setting for the current script to run a section of code that may require more resources than usual:

```
$limit = ini_set("memory_limit", "128M");
// Execute code that requires this setting
ini_set("memory_limit", $limit);
```

The previous value is saved to a variable and then restored when the intensive code has completed.

To find the current value of any `php.ini` setting without changing it, you use the `ini_get` function.

## Configuration Directives

This lesson cannot cover every configuration directive in `php.ini` in detail—there are simply too many. However, in the following sections you will learn how some of the most commonly used settings work. For a full reference, refer to [www.php.net/manual/en/ini.php](http://www.php.net/manual/en/ini.php).

## Configuring the PHP Environment

The following sections list some of the common configuration directives that affect the environment in which PHP runs. Each directive listed in the following sections is shown with its default entry from the `php.ini` file that is distributed with PHP 5, where the default is set.

### PHP Tag Styles

These directives allow you to select which tag styles can be used in a PHP script:

- **short\_open\_tag = On**—The `short_open_tag` directive enables or disables the use of the `<?` opening tag. If this setting is turned off, your scripts must use the full `<?php` tag.

Because `<?` can have other meanings when embedded in a web page, you should try to avoid using `short_open_tag`, and in future releases of PHP, it may be disabled by default.

- **asp\_tags = Off**—The `asp_tags` style of PHP tag begins with `<%` and ends with `%>`. You must enable this style in `php.ini` if you want to use it.

### System Resource Limits

The following directives allow you to manage the system resources available to a PHP script:

- **max\_execution\_time = 30;**—The `max_execution_time` directive specifies the maximum total number of seconds that a script can run. After this time is exceeded, an error occurs, and script execution stops.

Unless you have a specific need for a higher value in order to run slow scripts, you should not change this value. An accidental infinite loop in your script would eat up a lot of system resources, and `max_execution_time` is a safeguard against this kind of problem.

If a web page takes 30 seconds or more to load, visitors will probably not wait for it to finish, unless they have requested

some specific information that they understand may take some time to generate.

- **memory\_limit = 8M**—Each PHP script has a memory usage limit to make sure that the work it is doing does not get out of control and affect the system in a negative way. Most scripts use only a very small amount of memory; to find out just how much, you can call the `memory_get_usage` function.

The **M** suffix indicates a value in megabytes; the **K** or **G** suffix could also be used, to indicate kilobytes or gigabytes, respectively. If you are absolutely sure you want to remove the memory limit completely, you can set `memory_limit` to `-1`.

## Form Processing

You can use these directives to change the way PHP interacts with web forms:

- **magic\_quotes**—The `magic_quotes` settings instruct PHP to automatically delimit quotes so that they are safe to use as string values. These are the defaults:

```
magic_quotes_gpc = On
magic_quotes_runtime = Off
magic_quotes_sybase = Off
```

The `magic_quotes_gpc` setting applies to data posted from a form and data from cookie values. (gpc stands for GET, POST, and COOKIE data.) The `magic_quotes_runtime` directive tells PHP to delimit quotes in data generated by the script, such as from a database query or host command.

Usually, quotes are delimited with a backslash character, but some databases, notably Sybase, use another quote character. When the `magic_quotes_sybase` setting is enabled, delimited quotes appear as `' '` instead of `\ '`.

- **register\_globals = Off**—The `register_globals` setting has been disabled in PHP by default since version 4.2. When it is enabled, this option causes PHP to create global variables that

contain the same information as the super-globals `$_ENV`, `$_GET`, `$_POST`, `$_COOKIE`, and `$_SERVER`. The variable names correspond to the key names in each of the super-global arrays.

- **`variables_order = "EGPCS"`**—The `variables_order` directive determines the order in which global variables are registered from the super-globals. With `register_globals` enabled and the default ordering, a cookie named `email` is registered more recently than a posted form value with the same name, so `$email` in the script contains the cookie's value.

Because `register_globals` creates values that are not distinguished by their source, it is strongly recommended that you use the super-global arrays; when you do so, you can be confident that `$_POST["email"]` was a form-submitted value, but `$email` could have come from one of several sources.

- **`register_long_arrays = On`**—Older PHP versions use arrays named `$HTTP_GET_VARS`, `$HTTP_POST_VARS`, `$HTTP_SERVER_VARS`, and so on instead of the newer super-global arrays. The `register_long_arrays` directive determines whether arrays with these names are created. This feature remains enabled by default for backward compatibility.

## Include Files

You can use the `include_path` directive to give a list of locations in which to search for a file referenced in an `include` or `require` statement. The locations are separated by colons on Linux/Unix systems and by semicolons on Windows systems.

Often you need to ensure that include files are kept in a directory that is not directly accessible by a web server. The following example defines an include path that contains a directory parallel to the web root of `/home/chris/public_html`:

```
php_value include_path ../home/chris/include
```

The period character (`.`) is used to indicate the current working directory, and in this example, it is given higher priority than the defined include

directory. In this case, if an `include` statement finds a matching file in both locations, the one in the working directory will be used. This type of configuration allows you to use shared library files across your server but override them for some scripts when necessary.

The `auto_prepend_file` and `auto_append_file` directives allow you to specify files that are automatically added at the start and end of each PHP script. The filename given is found in `include_path`, or a full path to the file can be given.

A common use for `auto_prepend_file` is to automatically include part of the HTML layout before the output from your script so that all your pages look the same. Because `auto_prepend_file` is a PHP feature, only files parsed by PHP have the file prepended; static HTML pages do not.



**HTTP Headers** After any output has been sent to the browser, you cannot use the `header` function to send HTTP headers or use any other PHP functions that require headers to be sent, such as session control functions or cookies. Therefore, any script included by `auto_prepend_file` must produce no output if you want to send custom HTTP headers.

## Error Logging

As you learned in Lesson 22, “Error Handling,” PHP allows you to configure the strictness of error reporting and the means by which it is reported.

The value of the `error_reporting` directive is a bitmask comprised of the values found in Table 22.1 in Lesson 22. You can use logical operators to combine values as follows:

```
error_reporting = E_ALL & ~E_NOTICE & ~E_STRICT
```

The `display_errors` and `log_errors` directives determine whether an error is written to the screen display and web server log file, respectively.



The default settings are as follows, with errors displayed to screen and not written to a file:

```
display_errors = On
log_errors = Off
```

You can use the `error_log` directive to specify an alternate filename, as in the following example:

```
error_log = /tmp/php_log
```

## Configuring PHP Extensions

Some PHP extensions have their own directives that can be configured in `php.ini` to adjust the behavior of that extension.

For clarity in the configuration file, section headings are used to separate extension-specific settings. For instance, all the settings that affect the MySQL extension are found in a section of `php.ini` that begins `[MySQL]`. Each directive name also has a prefix that indicates the extension to which it belongs (for example, `mysql.connect_timeout` or `session.cookie_path`).

You can find documentation for extension-specific configuration directives in the online manual pages for each extension.

## Configuring System Security

Some of the directives in `php.ini` that are not covered in this lesson—most notably the `safe_mode` directive and its related settings—concern server security. These configuration options allow you to restrict certain types of functionality on the web server, and you will learn about them in Lesson 24, “PHP Security.”

## Loadable Modules

PHP allows you to load certain extensions at runtime. This means that you can extend the functionality of PHP without needing to recompile from source.

## Loading Extensions on Demand

You use the `d1` function to dynamically load an extension module. You build extensions as dynamically loadable objects when PHP is compiled, by using the `--with-EXTENSION=shared` switch. For instance, running the following `configure` statement causes PHP to be compiled with MySQL support linked in but with socket support as a loadable extension:

```
./configure --with-mysql --with-sockets=shared
```

The argument given to `d1` is the filename of the extension. In the case of the sockets extension, it would be called `sockets.so` on Linux/Unix but `php_sockets.dll` on Windows systems.



**Loadable Extensions** Whether the `d1` function is available is governed by the `enable_d1` directive in `php.ini`. You may find that on a shared web hosting service, this feature is not available to you.

To check whether an extension is loaded into PHP, you use the `extension_loaded` function. Given an extension name argument, this function returns `TRUE` or `FALSE`, depending on the presence of that extension. Note that PHP cannot tell whether an extension was loaded by using `d1` or is compiled in.

## Loading Modules on Startup

If you have extensions as loadable modules and want them to be loaded into PHP without needing to run `d1` in every script, you can use the `extension` directive in `php.ini` to provide a list of extensions to load at startup.

Each extension is given on a separate line, and there is no limit to the number of extensions you can load in this way. The following lines from

php.ini ensure that the sockets and imap extensions are loaded automatically on a Linux/Unix server:

```
extension=imap.so
extension=sockets.so
```

On a Windows web server, the configuration lines need to look like this, to reflect the difference in filenames between the two platforms:

```
extension=php_imap.dll
extension=php_sockets.dll
```

## Summary

In this lesson you have learned how to configure PHP at runtime. In the next lesson you will learn about PHP's Safe Mode and how to minimize security threats to your website.



## LESSON 24

# PHP Security

*PHP is undoubtedly a very powerful server-side scripting language, but with great power comes great responsibility. In this lesson you will learn how to use PHP's Safe Mode to make sure that some of the potentially dangerous features of PHP are locked down.*

## Safe Mode

PHP's Safe Mode attempts to provide a degree of basic security in a shared environment, where multiple user accounts exist on a PHP-enabled web server.

When a web server is running PHP in Safe Mode, some functions are disabled completely, and others are available with limited functionality.

## Restrictions Enforced by Safe Mode

Functions that attempt to access the filesystem have restricted functionality in Safe Mode. The web server process runs under the same user ID for all web space accounts and must have the appropriate read or write permission to access a file. This is a requirement of the underlying operating system and has nothing to do with PHP itself.

When Safe Mode is enabled and an attempt is made to read or write a local file, PHP checks whether file ownership of the script is the same as that of the target file. If the owner differs, the operation is prohibited.



**Write Permission** Although Safe Mode implements measures to prevent you from opening another user's files through PHP, the operating system's file permissions may still allow read or even write access to those files at a lower level. Be aware that a user who has shell access to the web server will be able to read any files that are accessible by the web server and write to any file that has global write permission.

The following core filesystem functions are restricted by this rule:

|                             |                                 |
|-----------------------------|---------------------------------|
| <code>chdir</code>          | <code>move_uploaded_file</code> |
| <code>chgrp</code>          | <code>parse_ini_file</code>     |
| <code>chown</code>          | <code>rmdir</code>              |
| <code>copy</code>           | <code>rename</code>             |
| <code>fopen</code>          | <code>require</code>            |
| <code>highlight_file</code> | <code>show_source</code>        |
| <code>include</code>        | <code>symlink</code>            |
| <code>link</code>           | <code>touch</code>              |
| <code>mkdir</code>          | <code>unlink</code>             |

Functions that are part of PHP extensions that also access the filesystem are similarly affected.



**Loadable Modules** The `d1` function is disabled in Safe Mode, regardless of the owner of the extension file. Extensions must be loaded into PHP at startup, using the extension directive in `php.ini`.

Functions that execute host programs are disabled unless they are run from the directory given in the `safe_mode_exec_dir` directive, which you will learn about in the next section. Even if execution is allowed, arguments to the commands are automatically passed to the `escapeshellcmd` function.

The following program execution functions are affected by this rule:

|                       |                         |
|-----------------------|-------------------------|
| <code>exec</code>     | <code>shell_exec</code> |
| <code>passthru</code> | <code>system</code>     |
| <code>popen</code>    |                         |

In addition, the backtick operator (```) is disabled.

The `putenv` function has no effect when run in Safe Mode, although no error is produced. Similarly, other functions that attempt to change the PHP environment, such as `set_time_limit` and `set_include_path`, are ignored.

## Enabling Safe Mode

You turn Safe Mode on or off by using the `safe_mode` directive in `php.ini`. To activate Safe Mode for all users on a shared web server, you use the following directive:

```
safe_mode = On
```

As you learned in the previous section, functions that access the filesystem perform a check on the owner of the file. By default, the check is performed on the file owner's user ID, but you can relax this to check the owner's group ID (GID) instead by turning on the `safe_mode_gid` directive.

If you have shared library files on your system, you can use the `safe_mode_include_dir` directive to get a list of locations for which the UID/GID check will not be performed when an `include` or `require` statement is encountered.



**Include Directories** If you want to list more than one location in the `safe_mode_include_dir` directive, you can separate them using colons on Linux/Unix or semi-colons on Windows systems—just as you do with the `include_path` setting.

To allow inclusion of files in `/usr/local/include/php` for any user in Safe Mode, you would use the following directive:

```
safe_mode_include_dir = /usr/local/include/php
```

To provide a location from which the system can be executed, you use the `safe_mode_exec_dir` directive.

To allow programs in `/usr/local/php-bin` to be executed in Safe Mode, you would use the following directive:

```
safe_mode_exec_dir = /usr/local/php-bin
```



**Executables** Rather than allow execution of all programs from `/usr/bin` or some other system location, you should create a new directory and copy or link only selected binaries into it.

To allow setting of certain environment variables, you use the `safe_mode_allowed_env_vars` directive. The value given is a prefix, and by default it allows only environment variables that begin with `PHP_` to be changed. If more than one value is given, the list should be separated by commas.

The following directive also allows the time zone environment variable, `TZ`, to be changed:

```
safe_mode_allowed_env_vars = PHP_,TZ
```

## Other Security Features

In addition to Safe Mode, PHP provides a number of functions that allow you to place restrictions on the features available to PHP.

### Hiding PHP

You can use the `expose_php` directive in `php.ini` to prevent the presence of PHP being reported by the web server, as follows:

```
expose_php = On
```

By using this setting, you can discourage automated scripts from trying to attack your web server. Usually, the HTTP headers contain a line that looks like the following:

```
Server: Apache/1.3.33 (Unix) PHP/5.0.3 mod_ssl/2.8.16
OpenSSL/0.9.7c
```

With the `expose_php` directive enabled, the PHP version is not included in this header.

Of course, the `.php` file extension is a giveaway to visitors that PHP is in use on a website. If you want to use a totally different file extension, you need to first find the following line in `httpd.conf`:

```
AddType application/x-httpd .php
```

Then you need to change `.php` to any file extension you like. You can specify any number of file extensions, separated by spaces. To have PHP parse `.html` and `.htm` files so there is no indication that a server-side language is being used at all, you can use the following directive:

```
AddType application/x-httpd .html .htm
```





**Parsing HTML** Configuring your web server to parse all HTML files with PHP may be convenient, but a small performance hit is involved because the PHP parser needs to fire up even if there is no server-side code in a web page.

By using a different file extension for static pages, you can eliminate the need for PHP to be involved where it is not necessary.

## Filesystem Security

Safe Mode restricts filesystem access only to files owned by the script owner, and you can use the `open_basedir` directive to specify the directory in which a file must reside. If you specify a directory, PHP will refuse any attempt to access a file that is not in that directory or its subdirectory tree. The `open_basedir` directive works independently of Safe Mode.

To restrict filesystem access on your web server to only the `/tmp` directory, you use the following directive:

```
open_basedir = /tmp
```

## Function Access Control

You can use the `disable_functions` directive to specify a comma-delimited list of function names that will be disabled in the PHP language. This setting works independently of Safe Mode.

To disable the `d1` function without turning on Safe Mode, you use the following directive:

```
disable_functions = d1
```

You can also disable access to classes by using the `disable_classes` directive in the same way.

## Database Security

You learned in Lesson 18, “Host Program Execution,” how a malicious user might try to run an arbitrary host command on your system, and that you can use the `escapeshellcmd` function to prevent this kind of abuse.

A similar situation applies to database use through PHP. Suppose your script contains the following lines to execute a MySQL query based on a form value:

```
$sql = "UPDATE mytable SET col1 = " . $_POST["value"] . "
 WHERE col2 = 'somevalue'";
$res = mysql_query($sql, $db);
```

You are expecting `$_POST["value"]` to contain an integer value to update the value of column `col1`. However, a malicious user could enter a semi-colon in the form input field, followed by any SQL statement he or she wants to execute.

For instance, suppose the following is the value of `$_POST["value"]`:

```
0; INSERT INTO admin_users (username, password)
VALUES ('me', 'mypassword');
```

The SQL executed would then look like the following (the statements are shown here on separate lines for clarity):

```
UPDATE mytable SET col1 = 0;
INSERT INTO admin_users (username, password)
VALUES ('me', 'mypassword');
WHERE col2 = 'somevalue';
```

This is clearly a bad situation! The first statement updates the value of `col1` for all rows in `mytable`. This will be an inconvenience, but the second statement creates a more serious problem—the user has been able to execute an `INSERT` statement that creates a new administrator login. The third statement is rubbish, but by the time the SQL parser reaches that statement and throws an error, the damage has been done. This type of attack is known as *SQL injection*.

Of course, for SQL injection to be a serious threat, the user must understand a little about your database structure. In this example, the attacker is aware that you have a table called `admin_users`, that it contains fields

named username and password, and that the password is stored unencrypted.

A visitor to your website would not generally know such information about a database you built yourself. However, if your website includes open-source components—perhaps you have used a freeware discussion board program—the table definitions for at least some of your database are accessible to users.

Furthermore, if your script produces output whenever a query fails, this could reveal important details about your database structure. On a production website, you should consider setting `display_errors` to `off` and using `log_errors` to write warnings and error messages to a file instead.



**Database Permissions** It is vital that the database connection from your script be made by a database user who has only just enough access rights to perform the job.

You should certainly never connect as an administrator from a script. If you did, an attacker would be able to gain full access to your database and others on the same server. Attackers will also be able to run the `GRANT` or `CREATE USER` command to give themselves full access outside the confines of your script!

To prevent the possibility of a SQL injection attack, you must ensure that user-submitted data that forms part of a query cannot be used to interrupt the SQL statement that you intend to execute.

The previous example shows an integer value being updated. If this were a string value enclosed in single quotes, the attacker would need to submit a closing quote before the semicolon and then the SQL statement. However, when `magic_quotes_gpc` is turned on, a quotation mark submitted via a web form will be automatically delimited.

To be absolutely sure that form-submitted values are not vulnerable to SQL injection attacks, you should always ensure that the data received is

appropriate. If your query expects a numeric value, you should test the form value with `is_numeric` or use `settype` to convert it to a number, removing any characters that are designed to fool SQL.

If you are working with several user-submitted values in one SQL statement, you can use the `sprintf` function to build a SQL statement string, using format characters that indicate the data type of each value. The following is an example:

```
$sql = sprintf("UPDATE mytable SET col1 = %d
 WHERE col2 = '%s'",
 $_POST["number"],
 mysql_escape_string($_POST["string"]));
```

The preceding example assumes that a MySQL database is being used, so the string value is passed to `mysql_escape_string`. For other databases, you should ensure that quote characters are adequately delimited, by using `addslashes` or another suitable method.

## Summary

In this lesson you have learned about the security considerations involved in building a dynamic website using PHP. In the next lesson you will learn about PEAR—PHP's primary resource for third-party add-ons.

# LESSON 25

## Using PEAR



*In this lesson you will learn about the PHP Extension and Application Repository (PEAR).*

### Introducing PEAR

PEAR is a framework and distribution system for reusable PHP packages. PEAR is made up of the following:

- A structured library of open-source code for PHP developers
- A system for distributing and maintaining code in packages
- The PEAR Coding Standards (PCS)
- The PHP Foundation Classes (PFC)
- Online support for the PEAR community through a website and mailing list

### The PEAR Code Library

PEAR brings together many different open-source projects, each of which is bundled into its own package. Each PEAR package has its own maintainers and developers, who determine the changes and release cycle for their own packages, but the package structure is consistent for all PEAR projects.

You use the PEAR installer, which is shipped with PHP, to automatically download and install a PEAR package by simply giving its name. You will learn how to use the PEAR installer later in this lesson.

Each package may have dependencies from other PEAR packages, and this is explicitly noted in the documentation, even if packages appear to be related because of their names.

A package tree structure exists within PEAR, and an underscore character (`_`) separates nodes in the hierarchy. For instance, the `HTTP` package contains various `HTTP` utilities, whereas `HTTP_Header` deals specifically with `HTTP` header requests.

## Package Distribution and Maintenance

PEAR packages are registered in a central database at <http://pear.php.net>. The PEAR website provides a searchable interface to the database by package name, category, and release date.

Maintainers of PEAR packages use the PEAR website to manage their projects. A CVS server allows developers to collaborate on source code and, once a release has been agreed upon, it can be made available from this central location immediately.

## PEAR Coding Standards

The PCS documents were created because many different teams are developing open-source packages that might be of use to the PHP community.

The documents in PCS outline a structured way in which code should be written in order for a package to be accepted as part of the PEAR project. The standards are quite detailed and contain mostly points of style, such as identifier naming conventions and a consistent style to use when declaring functions and classes.

This may sound a little daunting, but as your scripts become more complicated, you will realize how important it is to write readable code, and you will begin to develop a clear coding style. The PCS documentation simply formalizes a set of guidelines for writing readable PHP.

You can find the PCS documents online at <http://pear.php.net/manual/en/standards.php>.

## PHP Foundation Classes

PFC is a subset of PEAR packages, and these classes have a strict set of entrance criteria:

- **Quality**—Packages must be in a stable state.
- **Generality**—Packages should not be excessively specific to any particular type of environment.
- **Interoperability**—Packages should work well with other packages and in different environments, and they should have a standardized API.
- **Compatibility**—Packages must be designed to be backward compatible when new features are added.

At the present time, only the PEAR installer is shipped with PHP. However, at a later date, certain classes may be included as standard. The PFC would be those classes.

## Online Support for PEAR

The PEAR website, at <http://pear.php.net>, includes comprehensive online documentation for the PEAR project. The package database can be searched via the website, and package maintainers can log in to update their project details.

There are a number of mailing lists for PEAR users, maintainers, core developers, and webmasters. You can join any or all of these lists by using the form at <http://pear.php.net/support/lists.php>.

## Using PEAR

In the following sections you will learn how to use PEAR to find and install packages on a system, and you'll learn how to submit your own projects for consideration as PEAR packages.

### Finding a PEAR Package

On every page of the PEAR website is a search box that you can use to search the package database. You simply enter a name, or part of a name, and all matching packages are displayed.



**Searching Packages** To perform a detailed search on a package name, maintainer, or release date, you can use the form at <http://pear.php.net/package-search.php>.

You can click the name of the package you are interested in from the search results. The page that is then displayed should give some key information about that package, including a summary of its features, the current release version, and status and information about its dependencies—that is, any other PEAR packages that are required for this package to work.

The tabs at the top of the package details page allow you to view the documentation. If you are unsure from the summary information about exactly what you can achieve by using a particular package, you can browse through the documentation pages.

If you simply want to browse all the available PEAR packages, you can go to the categorized list at <http://pear.php.net/packages.php>.

## Using the PEAR Installer

When you have decided that a package will be useful, you can download it from the web by using the tab at the top of its package information page. However, using the PEAR installer program is a quick and easy way to manage packages within a PHP installation. The installer is able to find and download the latest version of a package and can also install it for you automatically.

The PEAR installer is named `pear`. To run the installer, you run the `pear` command followed by a command option. To see all the packages currently installed on a system, you can use the `list` command option:

```
$ pear list
```





**Command Options** Running pear with no arguments brings up a list of all the available command options.

The output produced should be similar to the following:

Installed packages:

=====

| Package    | Version | State  |
|------------|---------|--------|
| DB         | 1.6.2   | stable |
| HTTP       | 1.2.2   | stable |
| Net_DNS    | 1.00b2  | beta   |
| Net_Smtp   | 1.2.6   | stable |
| Net_Socket | 1.0.1   | stable |
| PEAR       | 1.3.2   | stable |
| SQLite     | 1.0.2   | stable |

Each package name, the version installed, and its release status are shown. The actual packages installed on your system may differ from the ones shown here.

You can use the search command option to search the PEAR package database. To search for all packages that contain the string mail, you run the following command:

```
pear search mail
```

The output produced displays all matching packages, their latest version numbers, and a brief summary. The search performed is not case-sensitive.

To view all the available stable PEAR packages, you use the list-all command:

```
pear list-all
```

This produces a long list!

To download and install a package, you use the install command option followed by the name of the package. To install the Mail package, you issue the following command:

```
pear install Mail_Queue
```

Some packages cannot be installed unless others are already installed on your system, and installation will fail if the required packages are not found. The following output shows an attempt to install the Mail\_Queue package before the Mail package has been installed:

```
pear install Mail_Queue
downloading Mail_Queue-1.1.3.tar ...
Starting to download Mail_Queue-1.1.3.tar (-1 bytes)
.....done: 98,816 bytes
requires package `Mail'
Mail_Queue: Dependencies failed
```

Some dependencies are optional. When you install the Mail package to fix the dependency reported in the previous error message, PEAR advises you that the functionality of the Mail package can be enhanced if you also install Net\_SMTP:

```
pear install Mail
downloading Mail-1.1.4.tar ...
Starting to download Mail-1.1.4.tar (-1 bytes)
.....done: 73,728 bytes
Optional dependencies:
package `Net_SMTP' version >= 1.1.0 is recommended to
utilize some features.
install ok: Mail 1.1.4
```

You can use the upgrade command option to download and install a later version of an installed package. To check whether a new version of the Mail package is released, you use the following command:

```
pear upgrade Mail
```

If a later version than the one installed is found, it is upgraded automatically.



**Upgrading Packages** You can use the `upgrade-all` command to check for newer versions of all your installed PEAR packages at once.

If you want to remove a PEAR package completely, you use the `uninstall` command.

## Contributing Your Own PEAR Project

If you have written a PHP project that you think will be useful to other developers, you might consider submitting a proposal to have it included in PEAR.

The online documentation (at <http://pear.php.net/manual/en/guide-newmaint.php>) includes a guide for project maintainers that details the process of first making sure that your project is suitable for submission to PEAR and then ensuring that your code is of a suitable standard. You should read that guide if you intend to write a package suitable for use by other developers. Even if your project is not suitable for PEAR, these guidelines will make you think about your software design and coding standards and will help you produce a much higher-quality package.

## Summary

In this lesson you have learned how to use PEAR. Now that you have completed this book, you can take advantage of the many freely available PEAR classes so that you can create PHP scripts that perform a wide variety of functions. Happy coding!



# APPENDIX A

## Installing PHP

*If you need to install PHP for yourself, this appendix is for you: It takes you through the process step-by-step, on both Linux/Unix and Windows platforms.*

### Linux/Unix Installation

These instructions take you through installing PHP from source, using an Apache web server. You should become the root user to perform the installation by issuing the `su` command and entering the superuser password.

### Compiling Apache from Source

If you already have installed an Apache web server that supports dynamic shared objects (DSO), you can skip this section. To check whether your web server includes this feature, run the following command:

```
$ httpd -l
```

If the output includes `mod_so.c`, then DSO support is included. Note that you may have to supply the full path to `httpd` (for instance, `/usr/local/apache/bin/httpd`).

You begin by downloading the latest Apache source code from <http://httpd.apache.org>. At the time of this writing, the latest version is 2.0.52, so the file to download is called `httpd-2.0.52.tar.bz2`. If a later version is available, you should be sure to substitute the appropriate version number wherever it appears in a filename.

You need to save this file to your filesystem in `/usr/local/src` or some other place where you keep source code. Uncompress the archive using `bunzip2`, as follows:

```
bunzip2 httpd-2.0.52.tar.bz2
```

When the file has been uncompressed, it loses the .bz2 file extension. You extract this archive file by using tar:

```
tar xvf httpd-2.0.52.tar
```

Files are extracted to a directory called httpd-2.0.52. You should change to this new directory before continuing:

```
cd httpd-2.0.52
```

Next, you should issue the configure command with any configuration switches that are appropriate. For instance, to change the base installation directory, you should use the --prefix switch, followed by the desired location. You can enter configure --help to see a list of the possible configure switches.

You need to include at least the --enable-module=so switch to ensure that DSO support is available for loading the PHP module later on. You should enter the following command, adding any other configuration switches that you need to include:

```
./configure --enable-module=so
```

The configure command produces several screens full of output as it tries to detect the best compilation settings for your system. When it is done, you are returned to a shell prompt and can continue the installation.

To begin compiling, you issue the make command:

```
make
```

Again, a lot of output is produced, and the time required for compilation depends on the speed of your system. When the build is done, you see the following line and are returned to a shell prompt:

```
make[1]: Leaving directory `/usr/local/src/httpd-2.0.52'
```

The final step is to install the newly built software. To do this, you simply enter make install, and the files are automatically copied to their correct system locations:

```
make install
```

You issue the `apachectl start` command to start the Apache web server and enter your server's IP address in a web browser to test that the installation is successful. You use the following command if you have not changed the default installation location:

```
/usr/local/apache/bin/apachectl start
```

## Compiling and Installing PHP

You can download the latest version of PHP from [www.php.net/downloads.php](http://www.php.net/downloads.php). At the time of this writing, the latest version is 5.0.3, so the file to download is called `php-5.0.3.tar.bz2`. If a later version is available, you should be sure to substitute the appropriate version number wherever it appears in a filename.

You need to save this file to your filesystem in `/usr/local/src` or some other place where you keep source code. You uncompress the archive by using `bunzip2`, as follows:

```
bunzip2 php-5.0.3.tar.bz2
```



**Uncompressing** If your system does not include the `bunzip2` utility, you should download the file called `httpd-2.0.52.tar.gz` instead. This archive is slightly larger but is compressed using `gzip`, which is more widely available.

When the file has been uncompressed, it loses the `.bz2` file extension. Extract this uncompressed archive file by using `tar`:

```
tar xvf php-5.0.3.tar
```

Files are extracted to a directory called `php-5.0.3`. You should change to this new directory before continuing:

```
cd php-5.0.3
```

Next, you should issue the `configure` command with any configuration switches that are appropriate. For example, to include database support

through the MySQLi extension, you would use the `--with-mysqli` switch, followed by the path to the `mysql_config` utility. To see the full list of configure switches, you can run `configure --help`.

You need to include either the `--with-apxs` or `--with-apxs2` switch—the latter is for Apache 2.0—followed by the location of the `apxs` utility on your system. You would use one of the following statements with a default Apache installation:

```
./configure --with-apxs2=/usr/local/apache2/bin/apxs
./configure --with-apxs2=/usr/local/apache/bin/apxs
```

The `configure` command produces several screens full of output as it tries to detect the best compilation settings for your system. When it is done, you are returned to a shell prompt and can continue the installation.

To begin compiling, you issue the `make` command:

```
make
```

Again, a lot of output is produced, and the time required for compilation depends on the speed of your system. When the build is done, you see the following text and are returned to a shell prompt:

```
Build complete.
(It is safe to ignore warnings about tempnam and tmpnam).
```

The final step is to install the newly built PHP module into your web server. To do this, you enter `make install`, and the files are automatically copied to their correct system locations:

```
make install
```

To complete the installation, you need to make a change to the web server configuration file to tell it that `.php` files should be passed to the PHP module. You should edit the `httpd.conf` file to add the following line:

```
AddType application/x-httpd-php .php
```

You can include other file extensions besides `.php` if you want.

When you next restart your web server by using the `apachectl restart` command, the PHP extension will be loaded. To test PHP, you can create

a simple script, `/usr/local/apache2/htdocs/index.php`, that looks like this:

```
<?php
phpinfo();
?>
```

In your web browser, you can visit `index.php` on the IP address of your web server, and you should see a page that gives lots of information about the PHP configuration.

## Windows Installation

The instructions in this section take you through installing PHP into an Apache web server on a Windows system.

### Installing Apache

If you already have an Apache web server installed on your system, you can skip this section.

Download the latest version of Apache from <http://httpd.apache.org>. The file to get is the MSI Installer package, named `apache_2.0.52-win32-x86-no_ssl.msi` for the current Apache 2.0.52 release. Save this file to your desktop and double-click to begin the installation process.

The installation process is done through a wizard and is mostly self-explanatory. You must accept the license terms to continue with the installation, after which you are shown some release notes. Click Next after you have read these, and you are asked to enter your server information.

Enter your server's domain name and hostname and your email address. If you are installing on a personal workstation, you should use `localhost` and `localhost` for your server information. You should leave the recommended option to install Apache on port 80 selected.

When asked to choose a setup type, you should select the typical setup. Then you are given the opportunity to select the destination folder for the Apache files. By default, this is `C:\Program Files\Apache Group`. Finally, Apache is ready to install, and clicking the Install button causes your system to start copying and setting up files on your system.



When the installation is complete, the Apache server and monitor program start up, and you see a new icon in your system tray. You can double-click this icon to bring up the Apache Service Monitor, which you can use to start and stop the web server process. A green light indicates a running server.

## Installing PHP

You can download the latest version of PHP from the Windows Binaries section of [www.php.net/downloads.php](http://www.php.net/downloads.php). You should choose the zip file rather than the installer package; it is named `php-5.0.3-Win32.zip` for the latest version of PHP, which is 5.0.3 at this writing. If a later version is available, be sure to substitute the appropriate version number wherever it appears in a filename.

You need to save the zip file to your desktop and double-click it to extract it to `C:\php`. You can choose another location, as long as you also change the other instructions in this section to reflect it.

Next, you need to add the PHP module to Apache. Using the file explorer, you need to open the Apache configuration directory (if you used the default location, it should be `C:\Program Files\Apache Group\Apache2\conf`) and edit `httpd.conf`. Then you need to add the following lines to the end of the file:

```
LoadModule php5_module c:/php/php5apache2.dll
AddType application/x-httpd-php .php
```

When you next restart your web server from the Apache monitor, the PHP extension will be loaded. To test PHP, you can create in the `htdocs` folder under your Apache installation location a simple script that looks like this:

```
<?php
phpinfo();
?>
```

In your web browser, if you visit `http://localhost/index.php`, you should see a page that gives lots of information on the PHP configuration.

## Troubleshooting

If you experience installation problems, first you should check that you have followed the steps in this chapter exactly. If you continue to have difficulties, try the following websites, which may be able to provide assistance:

- <http://httpd.apache.org/docs-2.0/faq/support.html>
- [www.php.net/manual/en/faq.build.php](http://www.php.net/manual/en/faq.build.php)

# INDEX



## SYMBOLS

---

# (use in single-line comments), 11-12  
#! (hash bang character), shell script, 187  
\$\_SERVER super-global array (web server environmental information), 142  
    scripts, 142-143  
    servers, 144  
    users, 143-144  
% (modulus operator), 40  
' (backticks), host program execution, 157-158  
\* (multiplication operator), 39  
+ (addition operator), 39  
++ (increment operator), 40-41  
- (subtraction operator), 39  
— (decrement operator), 40-41  
/ (division operator), 39  
/\*, \*/ (use in multiple-line comments), 11-12  
// (use in single-line comments), 11-12

## A

---

accessing  
    arrays, 57-58  
    cookies, 123  
    filesystems via file permissions, 146-147  
    form values  
        *GET method*, 98-100  
        *POST method*, 98-100  
activating Safe Mode, 216-217  
addition (+) operator, 39  
alignment specifiers, string formatting, 52

alnum class, characters in regular expressions, 71  
alpha class, characters in regular expressions, 71  
Apache web servers  
    add-on HTTP authentication modules, 130  
    dynamic shared objects (DSOs), 230-232  
    PHP installations  
        *on Linux/Unix platforms*, 230-233  
        *on Windows platforms*, 234-235  
    source code  
        *apachectl start command*, 232  
        *compiling*, 230-232  
        *configure command*, 231  
        *downloading*, 230  
        *make command*, 231  
        *make install command*, 231  
        *troubleshooting installation of*, 236  
Apache.org website, installation resources, 236  
apachectl start command, starting Apache web servers, 232  
arguments, functions, 32-33  
arithmetic operators  
    addition (+), 39  
    compound operations, 41  
    division (/), 39  
    modulus (%), 40  
    multiplication (\*), 39  
    precedence rules, 42  
    subtraction (-), 39  
array function, 58  
arrays, 5

- accessing, 57-58
- associative, textual key names, 60
- contents
  - outputting, 59
  - searching, 64
- creating, 57-58
- declaring, 57-58
- functions, 61
  - array\_diff*, 64
  - array\_intersect*, 63
  - array\_key\_exists*, 64
  - array\_merge*, 63
  - array\_search*, 64
  - array\_unique*, 63
  - asort*, 62
  - count*, 64
  - in\_array*, 64
  - ksort*, 62
  - rsort*, 63
  - serialize*, 65
  - shuffle*, 63
  - sort*, 62
  - unserialize*, 65
- index values, assigning, 58
- looping through
  - for* loop, 60
  - foreach* loop, 60
  - while* loop, 60-61
- merging, 63-64
- multidimensional, defining, 66-67
- randomizing, 63
- session variables, storing, 127
- two-dimensional, 65-66
- array\_diff* function, 64
- array\_intersect* function, 63
- array\_key\_exists* function, 64
- array\_merge* function, 63
- array\_search* function, 64
- array\_unique* function, 63
- ASCII values, strings, comparing, 49
- asort function, array manipulation, 62
- associative arrays, textual key names, 60
- attributes of files, retrieving (*file\_exists* function), 147-148
- authentication (websites), 128
  - basic HTTP, 129
    - Apache add-on modules*, 130
    - drawbacks*, 130
    - htaccess* file, 128
    - htpasswd* program, 129-130
- code listings
  - Basic Login Form (15.1)*, 132-133
  - Login Processor Script (15.2)*, 133-134

- Login Processor Script with Encrypted Passwords (15.3)*, 135-136
- session-based, 130-131
  - building*, 131-136
  - login forms*, 131-134
  - login usability*, 136
  - password encryption*, 134-136
- auto\_append* directive (php.ini file), 210
- auto\_prepend* directive (php.ini file), 210

---

## B

- backslash (\) character, escaping strings, 47-48
- backticks (`), host program execution, 157-158
- base 2 numbers (strings), 51
- basename function (filenames), 149
- basic HTTP authentication, 129
  - Apache add-on modules, 130
  - drawbacks, 130
  - htaccess file, configuration directives, 128
  - htpasswd program, 129-130
- Basic Login Form (Listing 15.1), 132-133
- binaries
  - CLI versus CGI, differentiating, 185
  - PHP, CLI/CGI binary support, 186
  - versions
    - SAPI numbering*, 186
    - v switch*, 186
- Boolean data types, 17-18
  - values in conditional statements, 20
- Bourne Again Shell (bash) interpreter, 186
- Bourne Shell (sh) interpreter, 186
- braces ({} )
  - code indentation rules, 21
  - use in conditional statements, 21
  - use in variables, 16
- brackets ([]), use in conditional statements, 21
- breaking strings into components in regular expressions, 76-77
- breaking loops, 28
- bunzip2 utility, uncompressing Apache source code downloads, 230

---

## C

- cache settings, HTTP headers, 140-142
- Cache-Control header (HTTP)
  - no-cache value, 140-142
  - private value, 140-142
  - public value, 140-142

- capitalization of strings
  - strtolower function, 54-55
  - strtoupper function, 54-55
- case-sensitive variables, 14
- ceil function, rounding number functions, 44
- CGI binaries
  - PHP version support, 186
  - versus CLI binaries, differentiating, 185
- character classes for regular expressions, 70
  - alnum, 71
  - alpha, 71
  - digit, 71
  - lower, 71
  - print, 71
  - punct, 71
  - space, 71
  - upper, 71
- character mode output in command-line scripts, 189
- characters
  - regular expressions
    - testing for repeat patterns*, 72-73
    - wildcard matching*, 72
  - sets, testing (ereg function), 69-70
- check boxes in forms, checking (dynamic HTML), 104-105
- chmod command, file permissions, setting, 146-147
- classes (OO programming), 87
  - appearance of, 88
  - constructors, 89
  - definitions, 88
  - disabling (disable\_classes directive), 219
  - functions, 87
  - inheritance, 87
  - methods, 87-89
  - object instances, creating, 88-89
  - private methods, 87
  - public methods, 87
  - third-party, 87, 90-92
  - when to use, 87
- CLI (command-line interface), 185
  - binaries
    - PHP version support*, 186
    - SAPI version numbering*, 186
    - version type*, 186
    - versus CGI binaries*, 185
  - stream access constants, 192
- closing
  - files via fclose function, 152
  - MySQL databases, 166
- code
  - braces ({}), indentation rules, 21
  - comments
    - multiple-line*, 11-12
    - single-line*, 11-12
  - functions, uses for, 30
  - modular, 31
- code listings
  - A Badly Formatted Script That Displays the Date and Time (1.3), 11
  - A Sample Registration Form with Required Fields (13.1), 115-117
  - A Web Form for Submitting User Comments (11.1), 97-98
  - An Extract from php.ini (23.1), 203-205
  - Basic Login Form (15.1), 132-133
  - Calling the finger Command from a Web Form (18.2), 162-163
  - Checking Whether Headers Have Been Sent (16.1), 139
  - Creating a Multiple-Option Selection Using Check Boxes (12.4), 110-112
  - Defaulting the Value of a Text Input Field (12.1), 103-104
  - Displaying the System Date and Time (1.1), 8-9
  - error reporting, Writing a Custom Error Handler (22.1), 197-198
  - Form Validation Using Inline Warnings (13.2), 118
  - Login Processor Script (15.2), 133-134
  - Login Processor Script with Encrypted Passwords (15.3), 135-136
  - Reading and Writing Standard Streams (21.2), 191-192
  - Selecting a Default Item from a Menu (12.3), 106-107
  - Selecting a Default Radio Button Group Item (12.2), 105-106
  - Using Arrays as Session Variables (14.2), 127
  - Using Command-Line Arguments (21.1), 190
  - Using Comments in a Script (1.4), 12
  - Using echo to Send Output to the Browser (1.2), 10-11
  - Using Session Variables to Track Visits to a Page (14.1), 126
  - Using the TZ Environment Variable to Change Time (18.1), 161-162

- combining words, use of underscore characters, 15
- comma-separated values (CSV), date files, reading, 153
- command strings, host program execution, 158
- command-line arguments
  - shell scripts, passing to, 189-190
  - Using Command-Line Arguments (Listing 21.1), 190
- command-line interface. *See* CLI
- command-line scripts
  - character mode output, 189
  - executing in Windows, 188
  - input/output streams, 191-192
  - PHP code, embedding, 188
  - streams, Reading and Writing Standard Streams (Listing 21.2), 191-192
- comments
  - code
    - multiple-line, 11-12*
    - single-line, 11-12*
  - Using Comments in a Script (Listing 1.4), 12
- comparing strings, ASCII values, 49
- comparison numeric functions
  - max, 45
  - min, 45
- comparison operators
  - equality, 49
  - inequality, 49
- compiling
  - Apache source code, 230-232
  - PHP (make command), 233
- compound operators, 41
- concatenation operators (strings), 48-49
  - joining, 16
- conditional statements, 20-21
  - Boolean values, 20
  - braces ({}), 21
  - brackets ([]), 21
  - logical operators, 22-23
  - multiple condition branches, 24-25
  - operators, 21-22
  - switch statement, 25-26
- configure command, Apache source code
  - switch settings, 231-233
- configuring PHP
  - .htaccess file, 205-206
  - php.ini file, 203, 205-211
  - switches via configure command, 232-233
- connecting MySQL databases, 165-166
  - DB class (PEAR), 178
  - errors, 172-173
- constants, error reporting, 194-196
- constructors, class methods, 89
- converting
  - date formats to timestamps, 83
  - implicit data types via type juggling, 18-19
- cookies, 121
  - accessing, 123
  - components, 121
  - domain paths, 122
  - expiration dates, 122
  - expiration times, 124
  - overwriting, 125
  - passwords, 124
  - setcookie function, creating, 123-124
  - stored values, 121
  - subdomains, 123
  - transmission of, 121
  - unsetcookie function, deleting, 125
  - versus sessions, 125
  - viewing, 123
- copy function, 149
- count function, array manipulation, 64
- custom error handlers, declaring, 196-198

---

## D

- data files
  - comma-separated values (CSV), reading, 153
  - retrieving via fgetcsv function, 154
  - writing to via fputcsv function, 154
- Data Manipulation Language (DML)
  - DELETE statement, 167
  - INSERT statement, 167
  - UPDATE statement, 167
- data rules, forms validation requirements, 117
- data source names (DSN)
  - components, 176-177
  - full syntax definition, 176
  - MySQL databases, connecting, 176-177
- data types
  - Boolean, 17-18
  - double, 17-18
  - gettype function, 17
  - integer, 17-18
  - juggling, 18-19
  - NULL values, 43
  - numeric, 42-43
  - querying, 17
  - settype function, 18
  - string, 17-18
    - converting to integer, 18*

- database abstraction (PEAR DB class)
  - installing, 175-176
  - supported extensions, 174
- databases
  - abstraction, PEAR DB class, 174-176
  - data source names (DSN)
    - components*, 176-177
    - connecting*, 176-177
    - full syntax definition*, 176
  - DB class (PEAR), portability modes, 181-182
  - MySQL
    - changing*, 167
    - closing connections*, 166
    - connecting*, 165-166
    - connection errors*, 172-173
    - full rows of data, fetching*, 169-170
    - hostnames*, 165
    - link identifiers*, 166
    - mysql extension versus mysqli extension*, 164
    - queried data, fetching*, 167-169
    - selecting*, 165
    - SQL statement execution*, 166-170
  - queries, limiting (DB class), 184
  - security
    - escapeshellcmd function*, 220
    - SQL injections*, 220-222
  - sequences
    - creating (DB class)*, 183-184
    - dropping (DB class)*, 184
  - SQL errors, debugging, 171-172
- date command, 7
- date formats
  - converting to timestamps, 83
  - listing of, 83
  - storage overview, 78
  - Unix timestamp
    - best uses*, 79
    - drawbacks*, 79
- date function, 30-31, 80
  - format codes, 80-81
- DB class (PEAR)
  - connect method, 178
  - createSequence method, 184
  - data source names (DSN), 176-177
    - components*, 176-177
    - full syntax definition*, 176
  - dropSequence method, 184
  - fetch modes, 181
  - getMessage method, 178
  - installing, 175-176
  - isError method, 178
  - limitQuery method, 184
  - online documentation, 175
  - portability modes, 181-182
  - queries, limiting, 184
  - quotation marks, use on string values, 183
  - selected data, retrieving, 179-180
  - sequences, 183
    - creating*, 184
    - dropping*, 184
  - SQL queries
    - executing*, 179
    - shortcut methods*, 181
  - supported database extensions, 174
  - using in scripts, 177
- DB\_PORTABILITY\_ALL constant, 182
- DB\_PORTABILITY\_DELETE\_COUNT constant, 182
- DB\_PORTABILITY\_ERRORS constant, 182
- DB\_PORTABILITY\_LOWERCASE constant, 182
- DB\_PORTABILITY\_NONE constant, 182
- DB\_PORTABILITY\_NULL\_TO\_EMPTY Y constant, 182
- DB\_PORTABILITY\_NUMROWS constant, 182
- DB\_PORTABILITY\_RTRIM constant, 182
- DB\_Result object
  - fetchRow method, 179-180
  - getAll method, 181
  - getOne method, 181
  - getRow method, 181
  - numCols method, 179-180
  - numRows method, 179-180
- debugging SQL errors in databases, 171-172
- decimal numbers (strings), 51
- declaring
  - arrays, 57-58
  - custom error handlers, 196-198
  - variables, 13
- decrement (—) operator, 40-41
- default argument values in functions, 34-35
- default input values in forms, setting (dynamic HTML), 103-104
- defining functions, 31-32
- DELETE statement (DML), 167
- desktop applications, creating (PHP-GTK extension), 192
- detecting host platforms, 159
- digit class, characters in regular expressions, 71

- directories
  - changing, 155
  - viewing contents, 155
- disable\_classes directive, 219
- disable\_functions directive, 219
- disabling
  - classes, 219
  - functions, 219
- displaying
  - HTTP headers, 140
  - validation warnings for forms, 114-117
- Displaying the System Date and Time (Listing 1.1), 8-9
- display\_errors directive, 201
- dissecting strings
  - sublen function, 55
  - subpos function, 55-56
  - substr function, 55-56
- division (/) operator, 39
- do loops, 27
- dollar sign prefix (\$), 13
- double data types, 17-18
- double-quoted strings, 16-17, 47
- downloading
  - Apache source code, 230
  - PHP, latest version of, 232
- dynamic HTML (DHTML), forms
  - check boxes, checking, 104-105
  - default input values, setting, 103-104
  - menus
    - creating, 109
    - default selections, 106-107
    - selecting multiple items, 109-112
  - radio buttons
    - creating, 107-109
    - selecting, 105-106
- dynamic shared objects (DSOs), Apache web server support, 230-232

---

## E

- echo command, 7
  - listings
    - A Badly Formatted Script That Displays the Date and Time (1.3), 11*
    - Using echo to Send Output to the Browser (1.2), 10-11*
  - outputting to browser, 10-11
- else clause, multiple condition branches, 24-25
- elseif keyword, multiple condition branches, 24-25

- email
  - return values, mail function example, 33
  - scripts, creating for user comments in forms, 101-102
- email address validation example, regular expressions, 74-76
- email\_validation\_class (third-party), 90
  - methods, 92
  - properties, 91-92
- embedding PHP code in command scripts, 188
- enable\_dl function, loading extensions on demand, 212
- enabling Safe Mode, 216-217
- environment variables
  - code listings, Using the TZ Environment Variable to Change Time (18.1), 161-162
  - PATH, 159-160
  - putenv function, 160
  - TZ, 161-162
  - web servers, \$\_SERVER super-global array, 142-144
- equality operator, string comparisons, 49
- ereg function (PCRE), 69
  - character sets, testing, 69-70
- ereg\_replace function, 77
- error handlers
  - custom, declaring, 196-198
  - Writing a Custom Error Handler (Listing 22.1), 197-198
- errors
  - reporting, 194
    - constants, 194-196
    - custom error handlers, 196-198
    - logs, 199-200
    - php.ini file configuration directives, 210-211
    - preventing display of, 201
    - user errors, raising, 198-199
    - suppressing, 200-202
  - error\_log function, message\_type argument, 199-200
- escape characters in strings, 47-48
- escapeshellcmd function, 163, 220
- escaping shell commands, host program execution, 162-163
- expiration times of cookies, 124
- explicit newline characters, 11
- expose\_php directive, hiding presence of PHP, 218-219
- expressions, use in variables, 15
- extensions
  - configuration directives, 211
  - loading on demand, 212
  - loading on startup, 212-213



external host programs, executing  
   via backticks (```), 157-158  
   via command strings, 158  
   via passthru function, 156-157

E\_ALL constant, 195  
 E\_COMPILE\_ERROR constant, 195  
 E\_COMPILE\_WARNING constant, 195  
 E\_CORE\_ERROR constant, 195  
 E\_CORE\_WARNING constant, 195  
 E\_ERROR constant, 194  
 E\_NOTICE constant, 194  
 E\_PARSE constant, 194  
 E\_STRICT constant, 195  
 E\_USER\_ERROR constant, 195  
 E\_USER\_NOTICE constant, 195  
 E\_USER\_WARNING constant, 195  
 E\_WARNING constant, 194

## F

fclose function, files, closing, 152  
 fetch modes, DB class (PEAR), 181  
 fetching  
   full rows of data in databases  
     (MySQL), 169-170  
   queried data in databases (MySQL),  
     167-169  
 fgets function, data file retrieval, 154  
 fgets function, files, reading, 152  
 fields in forms  
   highlighting, 118  
   validation requirements, 113-114  
 file directories  
   changing, 155  
   contents, viewing, 155  
 file extensions, shell scripts, 188  
 file permissions  
   Safe Mode restrictions, 214-216  
   shell scripts, executing, 187  
 fileatime function, 148  
 filectime function, 148  
 filegroup function, 148  
 fileinode function, 148  
 filemtime function, 148  
 fileowner function, 148  
 fileperms function, 148  
 files  
   attributes, testing (file\_exists  
     function), 147-148  
   closing via fclose function, 152  
   opening via fopen function,  
     150-151  
   pointers  
     moving (*fseek function*), 153  
     moving (*ftell function*), 152  
     writing to, 153

reading  
   via *fgets function*, 152  
   via *file\_get\_contents function*,  
     150  
   via *fread function*, 151  
 remote handling via URLs, 154  
 writing via file\_put\_contents  
   function, 150

filesize function, 148  
 filesystems, access management  
   file copying, 149  
   file information, 147-148  
   file movement, 149  
   file names, 149  
   file permissions, 146-147  
   restricting, 219  
 filetype function, 148  
 file\_exists function, file information  
   retrieval, 147-148  
 file\_get\_contents function, files, reading,  
   150  
 file\_put\_contents function, files, writing,  
   150  
 finger command, Calling the finger  
   Command from a Web Form (Listing  
   18.2), 162-163  
 float widths, string formatting, 53  
 floor function, rounding numbers, 44  
 flow control  
   conditional statements, 20-21  
     *Boolean values*, 20  
     *logical operators*, 22-23  
     *multiple condition branches*,  
       24-25  
     *operators*, 21-22  
     *switch statement*, 25-26  
   loops, 26  
     *breaking out of*, 28  
     *do*, 27  
     *for*, 27-28  
     *nested*, 28  
     *while*, 26-27

fopen function  
   files, opening, 150-151  
   mode arguments, 150-151  
 for loop, 27-28  
   arrays, looping through, 60  
 foreach loop, looping through arrays, 60  
 <FORM> tag (HTML)  
   ACTION attribute, 93  
   GET method, 94  
   METHOD attribute, 93  
   POST method, 94  
 format codes, string formatting, 52-53  
 format specifiers, string formatting, 52

- formatting strings
  - format codes, 52-53
  - printf function, 50-51
  - sprintf function, 53-54
- forms
  - check boxes, checking (dynamic HTML), 104-105
  - code listings
    - A Sample Registration Form with Required Fields (Listing 13.1), 115-117*
    - Defaulting the Value of a Text Input Field (12.1), 103-104*
    - Form Validation Using Inline Warnings (Listing 13.2), 118*
    - Selecting a Default Item from a Menu (12.3), 106-107*
    - Selecting a Default Radio Button Group Item (12.2), 105-106*
  - default input values, setting (dynamic HTML), 103-104
  - email scripts, creating, 101-102
  - hidden inputs, 100-101
  - HTML
    - `<FORM>` tag, 93-94
    - `<INPUT>` tag, 94-96
    - `<SELECT>` tag, 96-97
    - submitting to PHP, 93-97
    - submitting user comments, 97-98
  - `<TEXTAREA>` tag, 96
  - menus
    - creating (dynamic HTML), 109
    - default selections (dynamic HTML), 106-107
    - selecting multiple items (dynamic HTML), 109-112
  - processing (php.ini file configuration directives), 208-209
  - radio buttons
    - creating (dynamic HTML), 107-109
    - selecting (dynamic HTML), 105-106
  - validation
    - data rules enforcement, 117
    - highlighting fields, 118
    - required fields enforcement, 113-114
    - warning displays, 114-117
    - values, accessing, 98-100
- fputcsv function, data files, writing to, 154
- fputs function, file pointer writing, 153
- fread function, files, reading, 151
- fseek function, file pointer movement, 153
- ftell function, file pointer movement, 152
- functions
  - arguments, 32-33
  - array manipulation, 61
    - `array_diff`, 64
    - `array_intersect`, 63
    - `array_key_exists`, 63
    - `array_merge`, 63
    - `array_search`, 64
    - `array_unique`, 63
    - `asort`, 62
    - `count`, 64
    - `in_array`, 64
    - `ksort`, 62
    - `rsort`, 63
    - `serialize`, 65
    - `shuffle`, 63
    - `sort`, 62
    - `unserialize`, 65
  - basename, 149
  - copy, 149
  - date, 30-31
    - format codes, 80-81
  - default argument values, 34-35
  - defining, 31-32
  - disabling (disable\_functions directive), 219
  - `ereg_replace`, 77
  - `error_log`, 199-200
  - escapeshellcmd, 163, 220
  - `fgetcsv`, 154
  - `fileatime`, 148
  - `filectime`, 148
  - `filegroup`, 148
  - `fileinode`, 148
  - `filemtime`, 148
  - `fileowner`, 148
  - `fileperms`, 148
  - `filesize`, 148
  - `filetype`, 148
  - `file_exists`, 147-148
  - `fputcsv`, 154
  - `fputs`, 153
  - `fwrite`, 153
  - `generate_checkboxes`, 110-112
  - `generate_menu`, 109
  - `generate_radio_button_group`, 107-109
  - `getcwd`, 155
  - `getdate`, 84-85
  - `headers_list`, 140
  - `headers_sent`, 138-139
  - `htmlentities`, 109
  - `ini_set`, 206
  - library files, creating, 36-37

- mail
    - form handling functions, 101-102*
    - return values, 33*
  - mathematical, 46
  - mktime, 81-83
  - mysql\_affected\_rows, 167
  - mysql\_close, 166
  - mysql\_connect, 165-166
  - mysql\_errno, 171-173
  - mysql\_error, 171-173
  - mysql\_fetch\_array, 169-170
  - mysql\_num\_rows, 168
  - mysql\_query, 166-167
  - mysql\_result, 168-169
  - mysql\_select\_db, 165
  - numeric, rounding numbers, 44
  - passthru, 156-157
  - phpinfo, 32-33
  - php\_sapi\_name, 186
  - printf, 50-51
  - print\_r, 59
  - prototype, 30-31
  - putenv, 160
  - readdir, 155
  - realpath, 149
  - rename, 149
  - restricted use of in Safe Mode, 215-216
  - return codes, failure/success, 33-34
  - return values, 32-33
  - rewinddir, 155
  - session\_start, 125-126
  - setcookie, cookie creation, 123-124
  - set\_error\_handler, 196-198
  - sprintf, 53-54
  - strptime, 83
  - trigger\_error, 198-199
  - unlink, 149
  - unsetcookie, cookie deletion, 125
  - uses for, 30
  - variable scope, 35-36
  - fwrite function, file pointer writing, 153
- ## G - H
- 
- generate\_checkboxes function, 110-112
  - generate\_menu function, dynamic menu creation (dynamic HTML), 109
  - generate\_radio\_group function, radio button group creation (dynamic HTML), 107-109
  - GET method, form values, accessing, 98-100
  - getcwd function, directories, changing, 155
  - getdate function, timestamp information values, 84-85
  - gettype function, 17
  - global variables, 35-36
  - GMT (Greenwich Mean Time), timestamps, 82
  - GNU.org website, date formats listing, 83
  - header function, custom HTTP headers, sending, 137-138
  - headers (HTTP)
    - cache settings, 140-142
    - checking on send condition, 138-139
    - code listings, Checking Whether Headers Have Been Sent (16.1), 139
    - cookies
      - accessing, 123*
      - domain paths, 122*
      - expiration dates, 122*
      - expiration times, 124*
      - subdomains, 123*
      - transmission, 121*
      - viewing, 123*
    - custom, sending, 137-138
    - displaying, 140
    - redirection, 138
    - sample server information, 137
  - headers\_list function, HTTP header display, 140
  - headers\_sent function, HTTP header transmissions, checking, 138-139
  - hidden inputs in forms, 100-101
  - hiding PHP (expose\_php directive), 218-219
  - highlighting form fields, 118
  - host environments
    - environment variables
      - PATH, 159-160*
      - putenv function, 160*
      - TZ, 161-162*
    - platforms, detecting, 159
  - host platforms, detecting, 159
  - host programs
    - executing
      - via backticks ('), 157-158*
      - via command strings, 158*
      - via passthru function, 156-157*
    - shell commands, escaping, 162-163
  - hostnames in MySQL databases, 165
  - htaccess file, per-directory configuration, 205-206

HTML (Hypertext Markup Language)  
 forms  
   submitting to PHP, 93-97  
   submitting user comments, 97-98  
 htmlentities function, 109  
 htpasswd program, 129-130  
 HTTP (Hypertext Transfer Protocol)  
   authentication, 129  
     *Apache add-on modules, 130*  
     *drawbacks, 130*  
     *htaccess file, 128*  
     *htpasswd program, 129-130*  
   code listings, Checking Whether  
     Headers Have Been Sent (16.1),  
     139  
   headers  
     *cache settings, 140, 142*  
     *checking on send condition,*  
       138-139  
     *cookie transmission, 121*  
     *custom, sending, 137-138*  
     *displaying, 140*  
     *redirection, 138*  
     *sample server information,*  
       137  
 HTTP\_USER\_AGENT element, 143-144

## I - J - K

include files, php.ini file configuration  
 directives, 209-210  
 include keyword, library function files,  
 37  
 include once keyword, library function  
 files, 37  
 include\_path directive (php.ini file), 209  
 increment (++) operator, 40-41  
 indenting code with braces, 21  
 index values in arrays, assigning, 58  
 inequality operator, string comparisons,  
 49  
 infinite loops, 26  
 inheritance in classes, 87  
 ini\_get function (php.ini file), 206  
 ini\_set function (php.ini file), 206  
 INPUT CHECKBOX input type, 95  
 INPUT tag (PHP)  
   CHECKED attribute, 95, 104-106  
   generate\_radio\_group function,  
     107-109  
   MAXLENGTH attribute, 94  
   NAME attribute, 95  
   RADIO input type, 95  
   TEXT input items, 94  
   VALUE attribute, 94  
 input values in forms, setting default  
 values (dynamic HTML), 103-104

INSERT statement (DML), 167  
 install command, PEAR package  
 installations, 227-228  
 installing  
   DB class (PEAR), 175-176  
   MySQL, website resources, 164  
   PEAR  
     *packages, 227-228*  
     *via PEAR installer, 226-227*  
   PHP to Apache Web servers  
     *Linux/Unix platforms, 230-233*  
     *Windows platforms, 234-235*  
 instances, class objects, creating, 88-89  
 integer data types, 17-18  
 in\_array function, array manipulation, 64  
 ksort function, array manipulation, 62

## L

library files, functions  
   creating, 36-37  
   including in other scripts, 37  
 link identifiers, MySQL databases, 166  
 Linux platforms, PHP installations on  
   Apache web servers, 230-233  
 list-all command, viewing PEAR  
 packages, 227  
 listings. *See* code listings  
 loadable modules  
   loading on demand, 212  
   loading on startup, 212-213  
 local variables, 35-36  
 logging errors (error\_log function),  
 199-200  
 logical operators, conditional statements,  
 22-23  
 login forms, session-based authentication,  
 131-134  
   usability of, 136  
 Login Processor Script (Listing 15.2),  
 133-134  
 looping through arrays  
   for loop, 60  
   foreach loop, 60  
   while loop, 60-61  
 loops, 26  
   breaking out of, 28  
   do, 27  
   for, 27-28  
   infinite, 26  
   nested, 28  
   while, 26-27  
 lower class, characters in regular  
 expressions, 71

## M

mail function  
   default argument values, 34-35  
   email scripts, form handlers,  
     101-102  
   return values, 33  
 Mail package (PEAR), 228  
 Mail\_Queue package (PEAR), 228  
 make command  
   Apache source code, compile  
     process, 231  
   PHP, compiling process, 233  
 make install command  
   Apache source code, 231  
   PHP installation process, 233  
 mathematical functions, 46  
 max function, numeric comparisons, 45  
 menus  
   code listings, Creating a Multiple-  
     Option Selection Using Check  
     Boxes (12.4), 110-112  
   forms  
     *creating (dynamic HTML),*  
       109  
     *default selections (dynamic*  
       *HTML), 106-107*  
     *selecting multiple items*  
       *(dynamic HTML), 109-112*  
 merging arrays, 63-64  
 Mersenne Twister, random number  
   generator, 46  
 message\_type argument (error\_log  
   function), 199-200  
 methods  
   classes, 89  
   constructors, 89  
 min function, numeric comparisons, 45  
 mktime function, timestamp creation,  
   81-83  
 modular code, 31  
 modulus (%) operator, 40  
 moving  
   file pointers  
     *via fseek function, 153*  
     *via ftell function, 152*  
   files  
     *copy function, 149*  
     *unlink function, 149*  
 multidimensional arrays, defining, 66-67  
 multiple condition branches in condi-  
   tional statements  
     else clause, 24-25  
     elseif keyword, 24-25  
 multiplication (\*) operator, 39

## MySQL

databases  
   *changing, 167*  
   *closing connections, 166*  
   *connecting, 165-166*  
   *connection errors, 172-173*  
   *full rows of data, fetching,*  
     *169-170*  
   *hostnames, 165*  
   *link identifiers, 166*  
   *queried data, fetching,*  
     *167-169*  
   *selecting, 165*  
   *SQL errors, debugging,*  
     *170-172*  
   installation resources, 164  
   mysql extension versus mysqli  
     extension, 164  
   online documentation, 165  
   SQL statements, executing, 166-170  
 MySQL Development website, 164  
 mysql\_affected\_rows function, 167  
 mysql\_close function, 166  
 mysql\_connect function, 165-166  
 mysql\_errno function, 171-173  
 mysql\_error function, 171-173  
 mysql\_fetch\_array function, 169-170  
 mysql\_num\_rows function, 168  
 mysql\_query function, 166-167  
 mysql\_result function, 168-169  
 mysql\_select\_db function, 165

## N

naming  
   shell scripts, 188  
   variables, 13-14  
     *case-sensitivity, 14*  
     *conventions, 14*  
 negative numbers, rounding, 44  
 nested loops, 28  
 Net\_Smtp package (PEAR), 228  
 newline characters, 11  
 NULL value, 43  
 numeric data types, 42-43  
 numeric functions  
   comparison  
     max, 45  
     min, 45  
 random  
   *rand, 45-46*  
   *srand, 46*  
 rounding numbers  
   *ceil, 44*  
   *floor, 44*  
   *round, 44*

## O

object-oriented programming. *See* OO programming

objects, instances, creating (OO programming), 88-89

OO (object-oriented) programming, 86  
advantages, 87  
classes

*appearance of*, 88

*constructors*, 89

*definitions*, 88

*functions*, 87

*inheritance*, 87

*methods*, 87, 89

*objects, instance creation*, 88-89

*private methods*, 87

*public methods*, 87

*third-party*, 87, 90, 92

PHP Classes website, 86

PHP functionality, 86

PHP.net website resources, 88

*Sams Teach Yourself Object-Oriented Programming in 21 Days*, 88

when to use, 87

opening files via `fopen` function, 150-151

`open_basedir` directive, fileaccess restrictions, 219

operators

arithmetic

*addition (+)*, 39

*division (/)*, 39

*modulus (%)*, 40

*multiplication (\*)*, 39

*subtraction (-)*, 39

compound, 41

conditional statements, 21-22

decrement (`—`), 40-41

increment (`++`), 40-41

precedence rules, 42

OPTION tag (PHP)

`generate_menu` function, 109

SELECTED attribute, 106-107

outputting array contents

(`print_r` function), 59

overwriting cookies, 125

## P

packages (PEAR)

dependencies, 223-224

distribution of, 224

downloading, 223-224

installing, 223-224, 227-228

PEAR Foundation Classes (PFC), 224-225

Mail, 228

Mail\_Queue, 228

maintenance of, 224

Net\_SMTP, 228

searching, 225-226

tree structure, 223-224

uninstalling, 228

upgrading, 228

viewing, 227

passing command-line arguments to shell scripts, 189-190

passthru function, host programs,

executing, 156-157

passwords

cookies, danger in, 124

encryption, session-based

authentication, 134-136

PATH environment variable, host

environments, 159-160

PCRE (Perl-Compatible Regular

Expression), 68

`ereq` function, testing sets of

characters, 69-70

PHP.net website documentation, 68

PCS (PEAR Coding Standards),

document styles, 224

PEAR (PHP Extension and Application

Repository)

code library, package dependencies, 223-224

coding standards, document styles, 224

components overview, 223

Foundation Classes package criteria,

224-225

installer, launching, 226-227

packages

*dependencies*, 223-224

*distribution of*, 224

*downloading*, 223-224

*installing*, 223-224, 227-228

*Mail*, 228

*Mail\_Queue*, 228

*maintenance of*, 224

*Net\_SMTP*, 228

*searching*, 225-226

*tree structure*, 223-224

*uninstalling*, 228

*upgrading*, 228

*viewing*, 227

projects, submitting proposals, 229

website

*online support and resources*, 225

*package locator*, 225-226

- pear command, launching PEAR installer, 226-227
- PEAR DB class (database abstraction), 174
  - connect method, 178
  - fetch modes, 181
  - getMessage method, 178
  - installing, 175-176
  - isError method, 178
  - online documentation, 175
  - portability modes, 181-182
  - queries, limiting, 184
  - quotation marks, use on string values, 183
  - selected data, retrieving, 179-180
  - sequences, 183
    - creating, 184
    - dropping, 184
  - SQL queries
    - executing, 179
    - shortcut methods, 181
  - supported database extensions, 174
  - using in scripts, 177
- per-directory configuration (.htaccess file), 205-206
- Perl-Compatible Regular Expression. *See* PCRE
- permissions (files), 146
  - chmod command, 146-147
  - read-only, 146-147
- PFC (PEAR Foundation Classes), package criteria, 224-225
- PHP (Hypertext Preprocessor), 5
  - Apache web servers
    - installations to Linux/Unix systems, 230-233
    - installations to Windows systems, 234-235
  - features overview, 5-6
  - latest version
    - compiling (*make command*), 233
    - downloading, 232
    - installing (*make install command*), 233
  - programming overview, 5-6
  - running locally from PC, 9
  - switches, configuring (*configure command*), 232-233
  - versions, binaries support, 186
- PHP Classes website, 86
- third-party class downloads, 90
- PHP code, command scripts, embedding, 188
- PHP Extension and Application Repository. *See* PEAR
- php-cli.ini file, 205
- PHP-GTK extension, creating desktop applications, 192
- PHP-GTK.net website, 192
- php-SAPI.ini file, 205
- php.ini file
  - alternates, 205
  - code listing, An Extract from php.ini (23.1), 203-205
  - configuration directives, 203-205
    - error logging*, 210-211
    - forms processing*, 208-209
    - include files*, 209-210
    - system resource limits*, 207-208
    - system security*, 211
    - tag styles*, 207
  - enable\_dl function, 212
  - extensions, configuring, 211
  - ini\_get function, 206
  - ini\_set function, 206
  - online resources, PHP.net website, 206
- PHP.net website
  - array functions, 61
  - installation resources, 236
  - mathematical function resources, 46
  - MySQL documentation, 165
  - online manual documentation, 30
  - OO programming resources, 88
  - operator precedence, 42
  - PCRE documentation, 68
  - PHP downloads, 232
  - php.ini file online resources, 206
  - string functions listing, 54
- phpinfo function, 32-33
- php\_sapi\_name function (CLI binaries), 186
- platforms, hosts, detecting, 159
- portability modes, DB class (PEAR), 181
  - constants, 182
- POSIX-extended syntax in regular expressions, 68
- POST method, form values, accessing, 98-100
- precedence of arithmetic operators, 42
- precision specifiers, string formatting, 53
- preventing error displays, 201
- print class, characters in regular expressions, 71
- printf function, string formatting, 50-51
- print\_r function, array contents, outputting, 59
- private methods (classes), 87
- projects, submitting proposals to PEAR, 229

- prototypes, functions, 30-31
- public methods (classes), 87
- punct class, characters in regular expressions, 71
- putenv function, host environment variables, 160

## Q - R

---

- queries (SQL)
  - executing via DB class (PEAR), 179-181
  - limiting (DB class), 184
- quotation marks, string values, 183
- radio buttons in forms
  - creating (dynamic HTML), 107-109
  - selecting (dynamic HTML), 105-106
- rand function, random number generation, 45-46
- random number generator (Mersenne Twister), 46
- random numeric functions
  - rand, 45-46
  - srand, 46
- randomizing arrays, 63
- read-only access permission (files), 146-147
- readdir function, directory navigation function, 155
- reading files
  - via fgetc function, 152
  - via file\_get\_contents function, 150
  - via fread function, 151
- realpath function (filenames), 149
- records, retrieving (PEAR DB class), 179-180
- redirection headers (HTTP), 138
- regular expressions (regex), 68
  - character classes, 70
    - alnum*, 71
    - alpha*, 71
    - digit*, 71
    - lower*, 71
    - print*, 71
    - punct*, 71
    - space*, 71
    - upper*, 71
  - characters, testing for repeat patterns, 72-73
- Perl-Compatible Regular Expression (PCRE), 68
  - ereq* function, 69-70
- POSIX-extended syntax, 68

- strings
  - breaking into components*, 76-77
  - searching and replacing*, 77
  - testing position*, 71-72
  - wildcard matching*, 72
- user input validation examples
  - email addresses*, 74-76
  - telephone numbers*, 74
  - zip codes*, 74
- remote files, handling via URLs, 154
- rename function, 149
- renaming files (rename function), 149
- repeat patterns, testing characters in regular expressions, 72-73
- replacing strings in regular expressions, 77
- reporting errors, 194
  - constants, 194-196
  - logs, 199-200
  - php.ini file configuration directives, 210-211
  - preventing display of, 201
  - user errors, raising, 198-199
- require keyword, library function files, 38
- require once keyword, library function files, 38
- retrieving
  - data files via fgetcsv function, 154
  - file information (file\_exists function), 147-148
  - records (PEAR DB class), 179-180
- return values
  - functions, 32-33
    - failure*, 33-34
    - success*, 33-34
  - mail function example, 33
- rewinddir function, directory navigation function, 155
- rounding number functions
  - ceil, 44
  - floor, 44
  - round, 44
- rsort function, array manipulation, 63

## S

---

- Safe Mode, 214
  - enabling, 216-217
  - file permissions, restrictions, 214-216
  - functions, restricted use of, 215-216
- Sams Teach Yourself MySQL in 24 Hours*, 164
- Sams Teach Yourself Object-Oriented Programming in 21 Days*, 88



- scalar variables, 57
- scripts
  - A Badly Formatted Script That Displays the Date and Time (Listing 1.3), 11
  - Displaying the System Date and Time (Listing 1.1), 8-9
  - flow control
    - conditional statements*, 20-21
    - conditional statements, logical operators*, 22-23
    - conditional statements, multiple condition branches*, 24-25
    - conditional statements, operators*, 21-22
    - conditional statements, switch statement*, 25-26
    - loops*, 26-28
  - library functions, including, 37
  - Using Comments in a Script (Listing 1.4), 12
  - Using echo to Send Output to the Browser (Listing 1.2), 10-11
  - web server information, web page requests, 142-143
- searching
  - PEAR packages, 225-226
  - strings in regular expressions, 77
- security
  - authentication
    - basic HTTP*, 128-130
    - session-based*, 130-131
    - session-based, building*, 131-136
    - session-based, login forms*, 131-134
    - session-based, login usability*, 136
    - session-based, password encryption*, 134-136
  - classes, disabling, 219
  - databases
    - escapeshellcmd function*, 220
    - SQL injections*, 220-222
  - filesystem access, restricting, 219
  - functions, disabling, 219
  - hiding presence of PHP (*expose\_php* directive), 218-219
  - host programs, shell commands, escaping, 162-163
  - php.ini file configuration directives, 211
  - Safe Mode, 214
    - enabling*, 216-217
    - permission restrictions*, 214-216
    - websites, password cookies, 124
  - SELECT tag (PHP)
    - generate\_menu* function, 109
    - MULTIPLE attribute, 109-112
  - semicolon character (;), statement termination, 7
  - sending HTTP headers
    - checking on send condition, 138-139
    - customized, 137-138
  - sequences in databases
    - creating (DB class), 183-184
    - dropping (DB class), 184
  - serialize function, array manipulation, 65
  - server-side scripting, page processing, 6-7
  - SERVER\_ADDR element, 144
  - SERVER\_PORT element, 144
  - session-based authentication, 130-131
    - building, 131-136
    - login forms, 131-134
    - login usability, 136
    - password encryption, 134-136
  - sessions
    - code listings
      - Using Arrays as Session Variables (14.2)*, 127
      - Using Session Variables to Track Visits to a Page (14.1)*, 126
    - creating via *session\_start* function, 125-126
    - variables, array storage, 127
    - versus cookies, 125
  - session\_start* function, session creation, 125-126
  - setcookie function, cookies, creating, 123-124
  - settype function, 18
  - set\_error\_handler function, 196-198
  - shell commands, host programs, escaping, 162-163
  - shell scripts
    - #! (hash bang) character, 187
    - arguments, passing, 189-190
    - Bourne Again Shell (bash), 186
    - Bourne Shell (sh), 186
    - file extensions, 188
    - file permissions, executing, 187
    - naming requirements, 188
  - shuffle function, array manipulation, 63
  - single-line comments, 11-12
  - single-quoted strings, 47
    - variables, 16-17

- sort function, array manipulation, 62
- source code (Apache)
  - apachectl start command, 232
  - compiling, 230-232
  - configure command, 231
  - downloading, 230
  - make command, 231
  - make install command, 231
- space class, characters in regular expressions, 71
- sprintf function, string formatting, 53-54
- SQL (Structured Query Language)
  - errors, debugging, 171-172
  - injections, database security, 220-222
  - queries, executing via DB class (PEAR), 179-181
  - statements
    - DML subset, 167
    - executing (MySQL), 166-170
- srand function, random number generation, 46
- statement termination, semicolon (;)
  - character, 7
- storing date formats, 78
  - Unix timestamp, 79
- stream identifiers, command-line scripts, 191-192
- string data types, 17-18
  - converting to integer data type, 18
- strings
  - capitalization
    - strtolower function, 54-55
    - strtoupper function, 54-55
    - ucfirst function, 54
    - ucwords function, 54
  - comparing ASCII values, 49
  - concatenation operator, 48-49
  - dissecting
    - sublen function, 55
    - substrpos function, 55-56
    - substr function, 55-56
  - escape characters, backslash (\), 47-48
  - formatting
    - format codes, 52-53
    - printf function, 50-51
    - sprintf function, 53-54
  - function of, 47
  - joining via concatenation operator, 16
  - numbers
    - base 2 format (%b), 51
    - decimal format (%d), 51
  - padding, 52
  - quotation marks
    - double, 47
    - single, 47
  - regular expressions
    - breaking into components, 76-77
    - position, testing, 71-72
    - searching and replacing, 77
  - variables
    - double-quoted, 16-17
    - single-quoted, 16-17
- strtolower function, string capitalization, 54-55
- strtotime function, 83
- strtoupper function, string capitalization, 54-55
- sublen function, dissection of strings, 55
- submitting
  - HTML forms, 93-97
    - <FORM> tag, 93-94
    - <INPUT> tag, 94-96
    - <SELECT> tag, 96-97
    - <TEXTAREA> tag, 96
  - project proposals (PEAR), 229
  - user comments in HTML forms, 97-98
- substrpos function, strings, dissection of, 55-56
- substr function, strings, dissection of, 55-56
- subtraction (-) operator, 39
- suppressing errors, 200-202
- switch statement, use in conditional statements, 25-26
- system resource limits, php.ini file
  - configuration directives, 207-208

---

## T

- tag styles, php.ini file configuration directives, 207
- tags (HTML)
  - FORM, 93-94
  - INPUT, 94-96
  - SELECT, 96-97
  - TEXTAREA, 96
- tags (PHP), processing instructions, 7-8
- telephone number validation example, regular expressions, 74
- testing string position in regular expressions, 71-72
- third-party classes, 90-92
  - email\_validation\_class, 90
  - methods, 92
  - properties, 91-92
- time function, timestamp, locating, 79
- time zones, setting web servers, 161-162

time.php script, date and time display, 8-9  
 timestamps, 80  
   converting date formats to, 83  
   creating (mktime function), 81-83  
   date function, format codes, 80-81  
   GMT (Greenwich Mean Time), 82  
   information values, obtaining (getdate function), 84-85  
   time function, 79  
 trigger\_error function, raising user errors, 198-199  
 troubleshooting  
   Apache source code installations, 236  
   PHP installations, 236  
 two-dimensional arrays, 65-66  
 type juggling (data types), implicit conversion, 18-19  
 TZ environment variable, host environments, 161-162

## U - V

ucfirst function, string capitalization, 54  
 ucwords function, string capitalization, 54  
 uncompressing Apache source code archive (bunzip2), 230  
 underscore characters, combining words, 15  
 uninstalling PEAR packages, 228  
 Unix timestamp format  
   best uses, 79  
   drawbacks, 79  
   ease of use, 79  
   maximum value, 79  
   mktime function, 81-83  
   starting value, 79  
 unlink function, files, moving, 149  
 unserialize function, array manipulation, 65  
 unsetcookie function, cookies, deleting, 125  
 UPDATE statement (DML), 167  
 upgrade-all command, 228  
 upgrading PEAR packages, 228  
 upper class, characters in regular expressions, 71  
 user comments  
   forms, email script creation, 101-102  
   Web forms, submitting, 97-98  
 user errors, raising (trigger\_error function), 198-199

users, web server information, web page requests, 143-144  
 Using Command-Line Arguments (Listing 21.1), 190  
 Using echo to Send Output to the Browser (Listing 1.2), 10-11  
 validating  
   email addresses with regular expressions, 74-76  
   forms  
   *data rules enforcement*, 117  
   *highlighting fields*, 118  
   *required fields enforcement*, 113-114  
   *warning displays*, 114-117  
   telephone numbers with regular expressions, 74  
   zip codes with regular expressions, 74  
 values in forms, accessing, 98-100  
 variable scope  
   global, 35-36  
   local, 35-36  
 variables  
   braces, 16  
   declaring, 13  
   dollar sign prefix (\$), 13  
   expressions, use in, 15  
   fixed values, 13  
   function of, 13  
   in strings  
   *double-quoted*, 16-17  
   *single-quoted*, 16-17  
   invalid names, 14  
   naming, 13-14  
   *case-sensitivity*, 14  
   *conventions*, 14  
   scalar, 57  
   underscore characters, word combinations, 15  
   valid names, 14  
   values, 13  
 viewing  
   cookies, 123  
   PEAR packages, 227

## W - Z

warnings in forms validation, displaying, 114-117  
 web browsers  
   cookies  
   *acceptance of*, 121  
   *accessing*, 123

- creating via *setcookie* function, 123-124
  - deleting via *unsetcookie* function, 125
  - domain paths, 122
  - expiration dates, 122
  - expiration times, 124
  - function of, 121
  - overwriting, 125
  - passwords, 124
  - stored values, 121
  - subdomains, 123
  - versus sessions, 125
  - viewing, 123
- HTTP\_USER\_AGENT element, 143-144
- sessions
  - creating via *session\_start* function, 125-126
  - storing variables in arrays, 127
- web pages
  - HTTP headers, cache control settings, 140-142
  - server-side scripting, processing, 6-7
- web servers
  - environment variables, *\$\_SERVER* super-global array, 142-144
  - host environments
    - environment variables, 159-160
    - time zone settings, 161-162
  - host platforms, detecting, 159
  - host programs
    - executing via backticks (```), 157-158
    - executing via command strings, 158
    - executing via *passthru* function, 156-157
  - HTTP headers
    - cache settings, 140-142
    - checking on send condition, 138-139
    - custom, sending, 137-138
    - displaying, 140
    - redirection, 138
    - sample information, 137
  - server-side scripting of web pages, 6-7
- websites
  - authentication, 128
    - basic HTTP, 128-130
    - session-based, 130-131
    - session-based, building, 131-136
    - session-based, login forms, 131-134
    - session-based, login usability, 136
    - session-based, password encryption, 134-136
  - cookies
    - accessing, 123
    - creating via *setcookie* function, 123-124
    - deleting via *unsetcookie* function, 125
    - domain paths, 122
    - expiration dates, 122
    - expiration times, 124
    - overwriting, 125
    - passwords, 124
    - stored values, 121
    - subdomains, 123
    - transmission of, 121
    - viewing, 123
  - GNU.org, date formats, 83
  - MySQL, 164
  - PEAR, online support and resources, 225
  - PHP Classes, 86, 90
  - PHP-GTK.net, 192
  - PHP.net, 30, 42, 46, 54, 61
    - MySQL documentation, 165
    - OO programming resources, 88
    - php.ini* file resources, 206
  - sessions
    - creating via *session\_start* function, 125-126
    - variables, 127
    - versus cookies, 125
- while loops, 26-27
  - arrays
    - looping through, 60-61
- width specifiers, string formatting, 52
- wildcards, matching characters in regular expressions, 72
- Windows
  - php command scripts, executing, 188
  - platforms, PHP installations on Apache web servers, 234-235
- writing
  - to data files via *fputcsv* function, 154
  - to file pointers, 153
  - to files via *file\_put\_contents* function, 150
- zip code validation example (regular expressions), 74

**SAMS**  
**Teach**  
**Yourself**



# CSS

Russ Weakley

*in* **10**  
**Minutes**

**SAMS**  
**Teach**  
**Yourself**

**CSS**

*in* **10**  
**Minutes**

Russ Weakley

**SAMS**

800 East 96th Street, Indianapolis, Indiana 46240 USA

# Sams Teach Yourself CSS in 10 Minutes

## Copyright © 2006 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32745-7

Library of Congress Catalog Card Number: 2004097471

Printed in the United States of America

First Printing: November 2005

08 07 06 05 4 3 2 1

## Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

## Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**

**1-800-382-3419**

**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**

**international@pearsoned.com**

## SENIOR ACQUISITIONS EDITOR

Linda Bump Harrison

## DEVELOPMENT EDITOR

Jon Steever

## MANAGING EDITOR

Charlotte Clapp

## PROJECT EDITOR

Mandie Frank

## COPY EDITOR

Jessica McCarty

## INDEXER

Aaron Black

## PROOFREADER

Brad Engels

## TECHNICAL EDITOR

Kevin Ruse

## PUBLISHING COORDINATOR

Vanessa Evans

## MULTIMEDIA DEVELOPER

Dan Scherf

## DESIGNER

Gary Adair

## PAGE LAYOUT

Nonie Ratcliff

# Table of Contents

|                                           |               |
|-------------------------------------------|---------------|
| <b>Introduction</b>                       | <b>1</b>      |
| Who Is This Book For? .....               | 1             |
| How This Book Works .....                 | 1             |
| Online Support Files .....                | 2             |
| Conventions Used in This Book .....       | 2             |
| <br><b>1 Understanding CSS</b>            | <br><b>3</b>  |
| What Is CSS? .....                        | 3             |
| What Does Cascading Mean? .....           | 3             |
| Why Use CSS? .....                        | 5             |
| <br><b>2 Using CSS Rules</b>              | <br><b>6</b>  |
| Setting Up the HTML Code .....            | 6             |
| Creating a Rule Set .....                 | 6             |
| Using Multiple Declarations .....         | 9             |
| Combining Selectors .....                 | 10            |
| Using Shorthand Properties .....          | 11            |
| Using Shorthand Borders .....             | 14            |
| Using Shorthand Margins and Padding ..... | 16            |
| Other Shorthand Properties .....          | 18            |
| <br><b>3 Selectors in Action</b>          | <br><b>20</b> |
| Setting Up the HTML Code .....            | 20            |
| Type Selectors .....                      | 21            |
| Class Selectors .....                     | 22            |
| ID Selectors .....                        | 22            |
| Descendant Selectors .....                | 23            |
| Universal Selectors .....                 | 24            |
| Advanced Selectors .....                  | 25            |



|          |                                                     |           |
|----------|-----------------------------------------------------|-----------|
| <b>4</b> | <b>Applying Styles</b>                              | <b>28</b> |
|          | Setting Up the HTML Code .....                      | 28        |
|          | Applying Inline Styles .....                        | 29        |
|          | Using Header Styles .....                           | 29        |
|          | Using External Style Sheets .....                   | 30        |
|          | @import Styles .....                                | 32        |
|          | Hiding Styles from Older Browsers .....             | 32        |
| <b>5</b> | <b>Getting to Know the CSS Box Model</b>            | <b>35</b> |
|          | Understanding Inline and Block Level Elements ..... | 35        |
|          | Setting Box Width .....                             | 36        |
|          | Margins .....                                       | 37        |
|          | Background Color and Background Image .....         | 39        |
|          | Padding .....                                       | 39        |
|          | Border .....                                        | 40        |
|          | Content Area .....                                  | 40        |
| <b>6</b> | <b>Adding Background Images</b>                     | <b>43</b> |
|          | Setting Up the HTML Code .....                      | 43        |
|          | Creating Selectors to Style the Header .....        | 44        |
|          | Adding background-image .....                       | 44        |
|          | Setting background-repeat .....                     | 45        |
|          | Adding background-position .....                    | 46        |
|          | Using the background Shortcut .....                 | 48        |
|          | Adding padding .....                                | 48        |
| <b>7</b> | <b>Formatting Text</b>                              | <b>50</b> |
|          | Setting Up the HTML Code .....                      | 50        |
|          | Removing Font Elements .....                        | 51        |
|          | Creating the Selectors .....                        | 52        |
|          | Styling the <p> Element .....                       | 53        |
|          | Styling the First Paragraph .....                   | 54        |
|          | Converting to Shorthand .....                       | 55        |
| <b>8</b> | <b>Styling a Flexible Heading</b>                   | <b>57</b> |
|          | Styling the Heading .....                           | 57        |
|          | Adding Color, Font Size, and Weight .....           | 58        |

|                                                                     |           |
|---------------------------------------------------------------------|-----------|
| Setting Text Options .....                                          | 60        |
| Applying Padding and Borders .....                                  | 60        |
| Adding a Background Image .....                                     | 62        |
| <b>9 Styling a Round-Cornered Heading</b> .....                     | <b>63</b> |
| Styling the Heading .....                                           | 63        |
| Styling the <h2> Element .....                                      | 64        |
| Adding a Background Image .....                                     | 65        |
| Styling the <em> Element .....                                      | 66        |
| <b>10 Styling Links</b> .....                                       | <b>68</b> |
| Links and Pseudo-Classes .....                                      | 68        |
| Setting Pseudo-Class Order .....                                    | 69        |
| Using Classes with Pseudo-Classes .....                             | 70        |
| Styling Links with Background Images .....                          | 71        |
| Removing Underlines and Applying Borders .....                      | 73        |
| Increasing the Active Area of Links .....                           | 74        |
| <b>11 Positioning an Image and Its Caption</b> .....                | <b>77</b> |
| Wrapping the Image and Caption .....                                | 77        |
| Floating the Container .....                                        | 78        |
| Applying Padding, Background Color, and a<br>Background Image ..... | 79        |
| Styling the Caption .....                                           | 81        |
| Styling the Image .....                                             | 82        |
| Creating a Side-By-Side Variation .....                             | 83        |
| Creating a Photo Frame Variation .....                              | 84        |
| <b>12 Creating a Photo Gallery</b> .....                            | <b>87</b> |
| Creating a Thumbnail Gallery .....                                  | 87        |
| Positioning the <div> Elements .....                                | 88        |
| Styling the Image .....                                             | 90        |
| Styling the Paragraph Element .....                                 | 91        |
| Forcing a New Line .....                                            | 93        |
| Creating a Side-By-Side Variation .....                             | 95        |

|           |                                                                                |            |
|-----------|--------------------------------------------------------------------------------|------------|
| <b>13</b> | <b>Styling a Block Quote</b>                                                   | <b>98</b>  |
|           | Applying the <code>&lt;blockquote&gt;</code> Element .....                     | 98         |
|           | Styling the <code>&lt;blockquote&gt;</code> Element .....                      | 100        |
|           | Styling the Paragraph .....                                                    | 101        |
|           | Styling the source Class .....                                                 | 102        |
|           | Creating a Variation .....                                                     | 104        |
| <b>14</b> | <b>Styling a Data Table</b>                                                    | <b>107</b> |
|           | Starting with a Basic Table .....                                              | 107        |
|           | Adding Accessibility Features to a Data Table .....                            | 108        |
|           | Creating Selectors to Style a Table .....                                      | 111        |
|           | Styling the Caption .....                                                      | 112        |
|           | Styling the <code>&lt;table&gt;</code> Element .....                           | 113        |
|           | Styling the <code>&lt;th&gt;</code> and <code>&lt;td&gt;</code> Elements ..... | 113        |
|           | Styling the <code>&lt;tr&gt;</code> Element .....                              | 114        |
|           | Targeting Instances of the <code>&lt;th&gt;</code> Element .....               | 115        |
|           | Creating Alternate Row Colors .....                                            | 117        |
| <b>15</b> | <b>Creating Vertical Navigation</b>                                            | <b>120</b> |
|           | Why Use a List? .....                                                          | 120        |
|           | Styling the List .....                                                         | 121        |
|           | Styling the <code>&lt;ul&gt;</code> Element .....                              | 121        |
|           | Styling the <code>&lt;a&gt;</code> Element .....                               | 122        |
|           | Adding a Hover Effect .....                                                    | 127        |
|           | Styling the <code>&lt;li&gt;</code> Element .....                              | 128        |
| <b>16</b> | <b>Creating Horizontal Navigation</b>                                          | <b>130</b> |
|           | Styling the List .....                                                         | 130        |
|           | Styling the <code>&lt;ul&gt;</code> Element .....                              | 131        |
|           | Styling the <code>&lt;li&gt;</code> Element .....                              | 132        |
|           | Styling the <code>&lt;a&gt;</code> Element .....                               | 133        |
|           | Styling the <code>:hover</code> Pseudo Class .....                             | 137        |
|           | Summary .....                                                                  | 138        |
| <b>17</b> | <b>Styling a Round-Cornered Box</b>                                            | <b>139</b> |
|           | Setting Up the HTML Code .....                                                 | 139        |
|           | Creating the Illusion of Round Corners .....                                   | 139        |

|                                                                            |            |
|----------------------------------------------------------------------------|------------|
| Creating Selectors to Style the Round-Cornered Box .....                   | 140        |
| Preparing the Images .....                                                 | 141        |
| Styling the <div> Element .....                                            | 141        |
| Styling the <h2> Element .....                                             | 142        |
| Styling the <p> Element .....                                              | 143        |
| Styling the .furtherinfo Class .....                                       | 144        |
| Styling the <a> Element .....                                              | 146        |
| Creating a Fixed-Width Variation .....                                     | 147        |
| Creating a Top-Only Flexible Variation .....                               | 149        |
| <br><b>18 Creating a Site Header</b> .....                                 | <b>152</b> |
| Setting Up the HTML Code .....                                             | 152        |
| Creating Selectors to Style the Header .....                               | 153        |
| Styling the <body> Element .....                                           | 153        |
| Styling the Container .....                                                | 154        |
| Styling the <h1> Element .....                                             | 156        |
| Styling the <image> Element .....                                          | 157        |
| Styling the <ul> Element .....                                             | 158        |
| Styling the <li> Element .....                                             | 160        |
| Styling the <a> Element .....                                              | 162        |
| <br><b>19 Positioning Two Columns with a Header<br/>and a Footer</b> ..... | <b>165</b> |
| Setting Up the HTML Code .....                                             | 165        |
| Creating Selectors to Style the Two-Column Layout .....                    | 166        |
| Styling the <body> Element .....                                           | 167        |
| Styling the Container .....                                                | 168        |
| Styling the <h1> Element .....                                             | 169        |
| Styling the #nav Container .....                                           | 171        |
| Styling the <ul> Element .....                                             | 173        |
| Styling the <li> Element .....                                             | 175        |
| Styling the #content Container .....                                       | 177        |
| Styling the #footer Container .....                                        | 179        |
| Styling the <h2> Element .....                                             | 182        |
| Styling the <a> Element .....                                              | 184        |

|           |                                                             |            |
|-----------|-------------------------------------------------------------|------------|
| <b>20</b> | <b>Styling a Page for Print</b>                             | <b>188</b> |
|           | Setting Up the Print CSS .....                              | 188        |
|           | Starting with Existing HTML and CSS Code .....              | 191        |
|           | Creating Selectors to Style for Print .....                 | 192        |
|           | Styling the <body> Element .....                            | 193        |
|           | Styling the <h1> Element .....                              | 194        |
|           | Styling the #nav Container .....                            | 196        |
|           | Styling the #footer Container .....                         | 197        |
|           | Styling the <a> Element .....                               | 199        |
| <b>21</b> | <b>Positioning Three Columns with a Header and a Footer</b> | <b>201</b> |
|           | Setting Up the HTML Code .....                              | 201        |
|           | Creating Selectors to Style the Three-Column Layout .....   | 202        |
|           | Creating a Liquid-Layout Grid .....                         | 204        |
|           | Creating the Background Images .....                        | 204        |
|           | Styling the <body> Element .....                            | 206        |
|           | Styling the <h1> Element .....                              | 207        |
|           | Styling the <h2> and <h3> Elements .....                    | 209        |
|           | Styling the First Container .....                           | 210        |
|           | Styling the Second Container .....                          | 212        |
|           | Styling the #content Column .....                           | 213        |
|           | Styling the #news Column .....                              | 215        |
|           | Styling the #nav Column .....                               | 217        |
|           | Styling the <u1> Element .....                              | 219        |
|           | Styling the #footer Element .....                           | 221        |
| <b>22</b> | <b>Troubleshooting CSS</b>                                  | <b>225</b> |
|           | Setting Up the CSS Code .....                               | 225        |
|           | Fixing the Problems .....                                   | 227        |
|           | Some Tips for Troubleshooting CSS Problems .....            | 234        |
|           | <b>Index</b>                                                | <b>237</b> |

---

## About the Author

**Russ Weakley** has worked in the design field for more than 18 years. During the last nine years, he has focused on web design through his own business, Max Design. He is also the web designer for the Australian Museum.

He co-chairs the Web Standards Group with Peter Firminger. The role of this group is to assist web developers in learning about new technologies and accessibility issues. He also co-founded Web Essentials, which organizes web development conferences and workshops that attract speakers and delegates from all over the world.

Internationally recognized for his presentations and workshops on web development, standards, and accessibility, Weakley has also produced a series of widely acclaimed CSS-based tutorials including Listamatic, Listamatic2, Listutorial, Floatutorial, and Selectutorial, which can all be found on his website at <http://www.maxdesign.com.au>.

# Acknowledgments

Thanks to the Sams team, in particular Linda Harrison, for giving me the opportunity to write this book.

Thanks to everyone who has given me comments, criticism, and positive feedback on my online tutorials. This feedback has given me confidence and insight into the problems other designers and developers face when learning CSS.

Thanks to Lisa Miller, who willingly proofread and user-tested every lesson.

Finally, thanks to my partner, Anna, for her patience, support, and encouragement throughout the writing of this book.

## We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

*Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.*

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:     webdev@sampublishing.com

Mail:       Mark Taber  
              Associate Publisher  
              Sams Publishing  
              800 East 96th Street  
              Indianapolis, IN 46240 USA

## Reader Services

For more information about this book or another Sams Publishing title, visit our website at [www.sampublishing.com](http://www.sampublishing.com). Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.





# Introduction

*Cascading Style Sheets (CSS)* is a simple and powerful language for adding style to web documents. Whether you are a web designer, developer, or anywhere in between, CSS is an important part of developing websites.

Many web developers still use tables for layout and do not understand the benefits of CSS. Although there are many good CSS resources and books available, people are often overwhelmed by the sheer volume of information. It is hard to decide the best place to start.

*Sams Teach Yourself CSS in 10 Minutes* is designed to help you get a handle on CSS quickly and easily through a series of step-by-step lessons.

## Who Is This Book For?

This book is for you if any (or all) of the following apply:

- You're new to CSS
- You want a simple, hands-on guide to using CSS
- You want to quickly learn how to get the most out of CSS
- You want to learn new ways to use CSS

## How This Book Works

*Sams Teach Yourself CSS in 10 Minutes* is divided into 22 lessons that gradually build on one another. By the end of the book, you should have a solid understanding of CSS and how to apply it in a variety of real-world situations.

Each lesson is written in simple steps so that you can quickly grasp the overall concept and put it into practice. The lessons are also designed to stand alone so that you can jump directly to particular topics as needed.

## Online Support Files

Each lesson from *Sams Teach Yourself CSS in 10 Minutes* has support files that can be downloaded from the Sams Publishing website. The files can either be downloaded as a single file for all lessons, or individually for each lesson.

The address is <http://www.sampublishing.com/>.

## Conventions Used in This Book

This book uses different typefaces to differentiate between HTML/CSS code and other content.

HTML and CSS code are presented using monospace type. Bold text indicates a change in code from the previous step.



**Note** A Note presents pertinent pieces of information related to the surrounding discussion.



**Caution** A Caution advises you about potential problems having to do with CSS or its implementation in specific browsers.



**Tip** Tip offers advice or demonstrates an easier way to do something.

# LESSON 1

# Understanding CSS



*In this lesson, you will learn about Cascading Style Sheets and why you should use them.*

## What Is CSS?

*Cascading Style Sheets (CSS)* is a language that works with HTML documents to define the way content is presented. The presentation is specified with *styles* that are placed directly into HTML elements, the head of the HTML document, or separate style sheets.

Style sheets contain a number of CSS rules. Each rule selects elements in an HTML document. These rules then define how the elements will be styled.

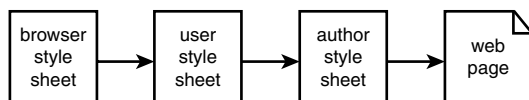
Any number of HTML files can be linked to a single CSS file.

## What Does Cascading Mean?

There are three types of style sheets that can influence the presentation of an HTML document in a browser. These are

- **Browser style sheets**—Browsers apply style sheets to all web documents. Although these style sheets vary from browser to browser, they all have common characteristics, including black text, blue links, and purple visited links. These are referred to as *default* browser style sheets.
- **User style sheets**—A user is anyone who looks at your website. Most modern browsers allow users to set their own style sheets within their browser. These style sheets will override the browser's default style sheets—for that user only.

- **Author style sheets**—The author is the person who develops the website—you! As soon as you apply a basic style sheet to a page, you have added an author style sheet. Author styles generally override user styles, which override browser styles. The cascade is shown in Figure 1.1.



**FIGURE 1.1** The three types of style sheets that influence the presentation of a web page.

*Cascading* means that styles can fall (or cascade) from one style sheet to another. The cascade is used to determine which rules will take precedence and be applied to a document.

For example, rules in author style sheets will generally take precedence over rules in user style sheets. Rules in user and author style sheets will take precedence over rules in the browser's default style sheet.



**Where Does CSS Come From?** CSS is a recommendation of the *World Wide Web Consortium (W3C)*. The W3C is an industry consortium of web stakeholders including universities; companies such as Microsoft, Netscape, and Macromedia; and experts in web-related fields.

One of the W3C's roles is to produce recommendations for a range of aspects of the Web. CSS1 became a recommendation in late 1996, CSS2 became a recommendation in May 1998, and CSS2.1 became a recommendation in January 2003.

## Why Use CSS?

Some of the benefits of using CSS for authors include

- **Easy to maintain**—The power of CSS is that a single CSS file can be used to control the appearance of multiple HTML documents. Changing the appearance of an entire site can be done by editing one CSS file rather than multiple HTML documents.
- **Smaller file sizes**—CSS allows authors to remove all presentation from HTML documents, including layout tables, spacer images, decorative images, fonts, colors, widths, heights, and background images. Presentation can then be controlled by CSS files. This can dramatically reduce the file sizes of HTML documents.
- **Increased accessibility**—CSS, combined with well-structured HTML documents, can aid devices such as screen readers. With presentational markup removed, the only thing that a screen reader encounters is structural content. CSS also can be used to increase the clickable area of links, as well as control line height and text line lengths for users with motor skill or cognitive difficulties.
- **Different media**—CSS can be styled specifically for different media, including browsers, printers, handheld devices, and projectors—without changing the content or document structure in any way.
- **More control over typography**—CSS allows authors to control the presentation of content with properties such as capitalize, uppercase, lowercase, text-decoration, letter-spacing, word-spacing, text-indent, and line-height. CSS can also be used to add margins, borders, padding, background color, and background images to any HTML element.

## Summary

In this lesson, you learned about Cascading Style Sheets and why you should use them. You also learned where style sheets come from and the three types of style sheets that can affect a web page.



## LESSON 2

# Using CSS Rules

*In this lesson, you will learn the syntax and rules of the Cascading Style Sheet (CSS) language. You will learn the components of CSS rules, including selectors, declarations, properties, and values. You will learn how to style a series of simple HTML elements. You will also learn how to use shorthand properties.*

## Setting Up the HTML Code

The HTML code for this lesson will be comprised of three elements—`<h1>`, `<h2>`, and `<p>`, as shown in Listing 2.1.

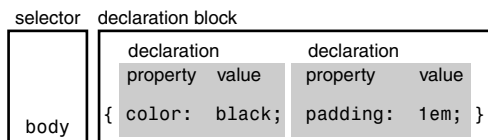
### **LISTING 2.1** HTML Code Containing the Markup for Lesson 2

---

```
<h1>
 Level 1 heading
</h1>
<h2>
 Level 2 heading
</h2>
<p>
 Lorem ipsum dolor sit amet, consectetur...
</p>
```

## Creating a Rule Set

A *rule*, or *rule set*, is a statement that tells browsers how to render particular elements on an HTML page. A rule set consists of a selector followed by a declaration block. Inside the declaration block, there can be one or more declarations. Each declaration contains a property and a value as shown in Figure 2.1.



**FIGURE 2.1** Diagram of rule set structure.

The first step in creating a rule set is to decide on a *selector*. The selector “selects” the elements on an HTML page that are affected by the rule set. The selector consists of everything up to (but not including) the first left curly bracket. The selectors used in this lesson are shown in Listing 2.2. Selectors are discussed in more detail in Lesson 3, “Selectors in Action.”

## LISTING 2.2 CSS Code Showing Selectors

---

```
h1
h2
p
```

Next, the *declaration block* must be created. A declaration block is a container that consists of everything between (and including) the curly brackets. The declaration blocks used in this lesson are highlighted in Listing 2.3.

## LISTING 2.3 CSS Code Showing Declaration Blocks

---

```
h1 {...}
h2 {...}
p {...}
```

Inside the declaration block, there are one or more declarations. *Declarations* tell a browser how to draw any element on a page that is selected. A declaration consists of a property and one or more values, separated by a colon. The end of each declaration is indicated with a semicolon.

The declarations used in this lesson are highlighted in Listing 2.4.

## LISTING 2.4 CSS Code Showing Declarations

---

```
h1 { text-align: center; }
h2 { font-style: italic; }
p { color: maroon; }
```





**Using Whitespace** Whitespace (spaces, tabs, line feeds, and carriage returns) is allowed around rule sets, as well as inside declaration blocks.

Rule sets can be laid out to suit your needs. Some developers prefer all declarations within a single line to conserve space as shown in Listing 2.5. Others prefer to place each declaration on a new line to make the rule sets easier to read as shown in Listing 2.6.

---

**LISTING 2.5** CSS Code Showing Single-Line Rule Set

---

```
h2 { font-style: italic; text-align: center; color: navy; }
```

---

**LISTING 2.6** CSS Code Showing Multiple-Line Rule Set

---

```
h2
{
 font-style: italic;
 text-align: center;
 color: navy;
}
```

The *property* is an aspect of the element that you are choosing to style. There can be only one property within each declaration unless a shorthand property is used (see “Using Shorthand Properties,” later in this lesson). The properties used in this lesson are highlighted in bold in Listing 2.7.

---

**LISTING 2.7** CSS Code Showing Properties

---

```
h1 { text-align: center; }
h2 { font-style: italic; }
p { color: maroon; }
```

The *value* is the exact style you want to set for the property. Values can include length, percentage, color, url, keyword, and shape. The values used in this lesson are highlighted (in bold) in Listing 2.8.

---

**LISTING 2.8** CSS Code Showing Values

---

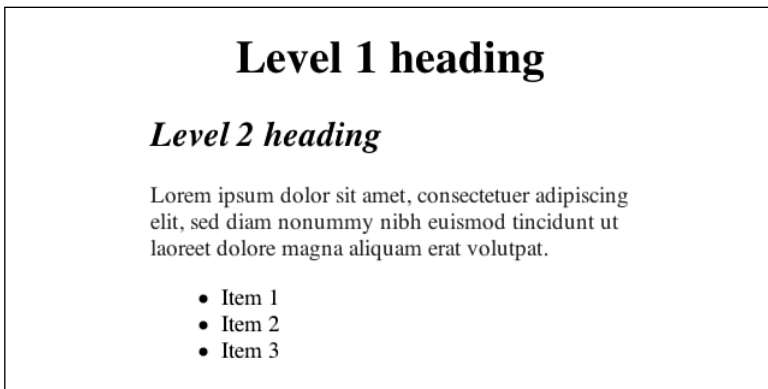
```
h1 { text-align: center; }
h2 { font-style: italic; }
p { color: maroon; }
```

The first rule set will target the `<h1>` element and align it in the center of the browser window.

The second rule set will target the `<h2>` element and render it in italics.

The third selector will target the `<p>` element and color all the text inside the element maroon.

The results of this styling applied to the HTML code in Listing 2.1 are shown in Figure 2.2.



**FIGURE 2.2** Screenshot of styled elements.

## Using Multiple Declarations

More than one declaration can be used within a declaration block. Each declaration must be separated with a semicolon.

In this example, the `<h1>` and `<h2>` elements will be styled with a new declaration—`color: navy;`. The `<h2>` element also will be styled with `text-align: center;`, which will align it in the center of the browser window. The new declarations are highlighted in Listing 2.9. The results are shown in Figure 2.3.

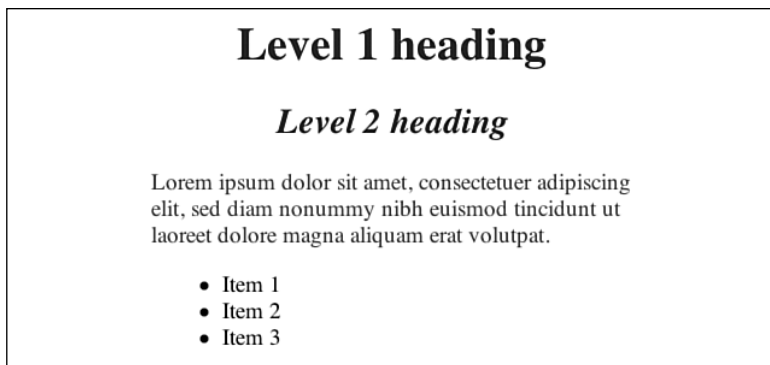
**LISTING 2.9** CSS Code Showing Multiple Declarations

---

```
h1
{
 text-align: center;
 color: navy;
}

h2
{
 font-style: italic;
 text-align: center;
 color: navy;
}

p
{
 color: maroon;
}
```



**FIGURE 2.3** Screenshot of elements styled with multiple declarations.

## Combining Selectors

When several selectors share the same declarations, they may be grouped together to prevent the need to write the same rule more than once. Each selector must be separated by a comma.

The `<h1>` and `<h2>` elements share two declarations, so parts of the two rule sets can be combined to be more efficient as shown in Listing 2.10.

**LISTING 2.10** CSS Code Showing Combined Selectors

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h2
{
 font-style: italic;
}

p
{
 color: maroon;
}
```



**Adding CSS Comments** CSS comments can be added to CSS to explain your code. Like HTML comments, CSS comments will be ignored by the browser. A CSS comment begins with `/*` and ends with `*/`. Comments can appear before or within rule sets as well as across multiple lines. They also can be used to comment out entire rules or individual declarations.

## Using Shorthand Properties

Shorthand properties allow the values of several properties to be specified within a single property. Shorthand properties are easier to write and maintain. They also make CSS files more concise.

For example, the `<h2>` element can be styled with `font-style`, `font-variant`, `font-weight`, `font-size`, `line-height`, and `font-family` as shown in Listing 2.11, or with a single `font` property as shown in Listing 2.12 and Figure 2.4.

Most shorthand properties do not require the values to be placed in a set order. However, when using the `font` property, it is safer to set values in the order specified by the W3C, which is `font-style`, `font-variant`, `font-weight`, `font-size`, `line-height`, and `font-family`.

When `font-size` and `line-height` are used within the `font` property, they must be specified with `font-size` first, followed by a forward slash (/), followed by `line-height`, as shown in Listing 2.12.

---

**LISTING 2.11** CSS Code Highlighting All font Properties

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h2
{
 font-style: italic;
 font-variant: small-caps;
 font-weight: bold;
 font-size: 100%;
 line-height: 120%;
 font-family: arial, helvetica, sans-serif;
}

p
{
 color: maroon;
}
```

---

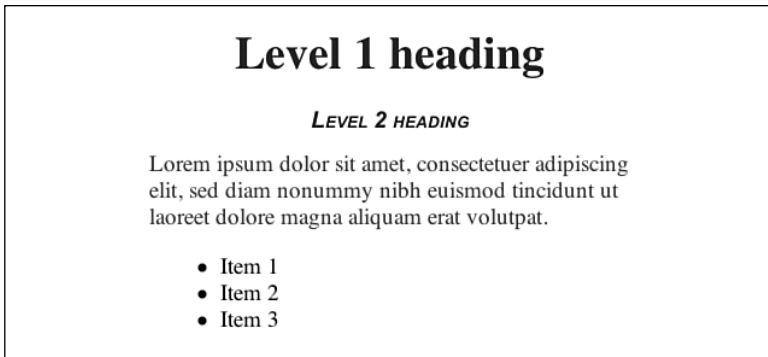
**LISTING 2.12** CSS Code Highlighting Shorthand font Property

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
 sans-serif;
}

p
{
 color: maroon;
}
```



**FIGURE 2.4** Screenshot of styled `<h2>` element.

Values for the shorthand font property include `font-style`, `font-variant`, `font-weight`, `font-size`, `line-height`, and `font-family`. However, all of these values do not need to be included in a shorthand declaration. For example, for the `<p>` element, you might want to only specify values for `font-size` and `font-family` as shown in Listing 2.13.

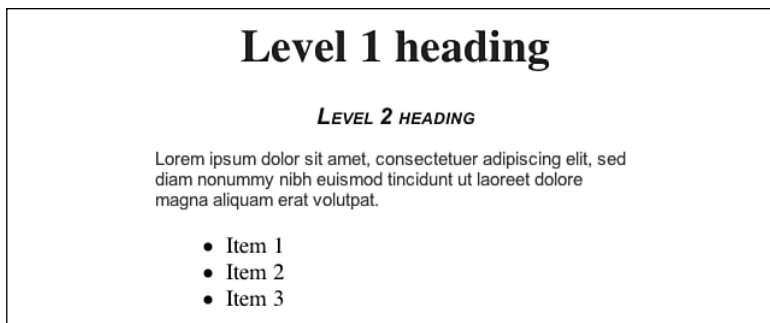
In this case, `font-style`, `font-variant`, `font-weight`, and `line-height` are not included in the shorthand property, so they will be assigned their default value. The results can be seen in Figure 2.5.

### **LISTING 2.13** CSS Code Highlighting All font Properties

```
h1, h2
{
 text-align: center;
 color: navy;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
}
```



**FIGURE 2.5** Screenshot of styled <p> element.

## Using Shorthand Borders

Border properties also can be converted to the shorthand border property. The <h1> element can be styled with `border-width`, `border-style`, and `border-color` as shown in Listing 2.14, or with a single border property as shown in Listing 2.15. The results can be seen in Figure 2.6.

### **LISTING 2.14** CSS Code Highlighting All border Properties

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border-width: 1px;
 border-style: solid;
 border-color: gray;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
 sans-serif;
}
```

*continues*

```
p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
}
```

### LISTING 2.15 CSS Code Highlighting the Shorthand border Property

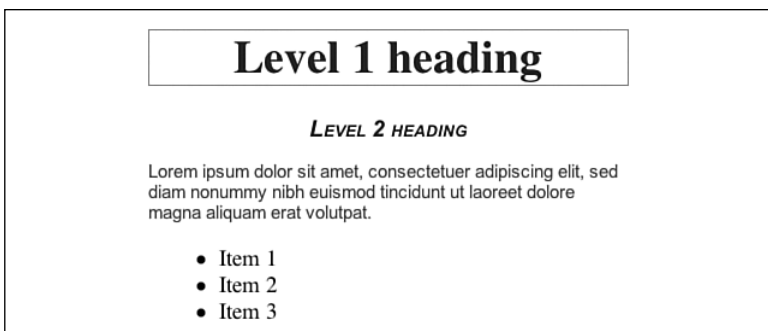
---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border: 1px solid gray;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
}
```



**FIGURE 2.6** Screenshot of styled `<h1>` elements.



## Using Shorthand Margins and Padding

Margins create space between the edge of an element and the edge of any adjacent elements. Padding creates the space between the edge of the element and its content (see Lesson 5, “Getting to Know the CSS Box Model,” for more information). The margin and padding shorthand properties also can be used to make CSS code more concise.

The margin property can combine margin-top, margin-right, margin-bottom, and margin-left. The padding property can combine padding-top, padding-right, padding-bottom, and padding-left.

The margin and padding properties also can be used to style different values for each side of an element. Values are applied in the following order: top, right, bottom, and left—clockwise, starting at the top.

The <p> element can be styled with padding-top, padding-right, padding-bottom, and padding-left as shown in Listing 2.16, or with a single padding property as shown in Listing 2.17.

### **LISTING 2.16** CSS Code Highlighting All padding Properties

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border: 1px solid gray;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
```

*continues*

```
font: 80% arial, helvetica, sans-serif;
padding-top: 1em;
padding-right: 2em;
padding-bottom: 3em;
padding-left: 4em;
}
```

### LISTING 2.17 CSS Code Highlighting a Shorthand padding Property

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border: 1px solid gray;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
 padding: 1em 2em 3em 4em;
}
```

You can use one, two, three, and four values within a shorthand declaration.

The declaration `p { padding: 1em; }` will apply 1em of padding to all sides of an element.

The declaration `p { padding: 1em 2em; }` will apply 1em of padding to the top and bottom, and 2em of padding to the left and right of an element.

The declaration `p { padding: 1em 2em 3em; }` will apply 1em of padding to the top, 2em of padding to the left and right, and 3em to the bottom of an element.

The declaration `p { padding: 1em 2em 3em 4em; }` will apply 1em of padding to the top, 2em of padding to the right, 3em of padding to the bottom, and 4em of padding to the left of an element.

## Other Shorthand Properties

The background property combines background-color, background-image, background-repeat, background-attachment, and background-position as shown in Listing 2.18.

### LISTING 2.18 CSS Code Highlighting a Shorthand background Property

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border: 1px solid gray;
 background: yellow url(tint.jpg) repeat-y 100% 0;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
 padding: 1em 2em 3em 4em;
}
```

The list-style property combines list-style-type, list-style-position, and list-style-image as shown in Listing 2.19.

**LISTING 2.19** CSS Code Highlighting a Shorthand list Property

---

```
h1, h2
{
 text-align: center;
 color: navy;
}

h1
{
 border: 1px solid gray;
 background: yellow url(tint.jpg) repeat-y 100% 0;
}

h2
{
 font: italic small-caps bold 100%/120% arial, helvetica,
sans-serif;
}

p
{
 color: maroon;
 font: 80% arial, helvetica, sans-serif;
 padding: 1em 2em 3em 4em;
}

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

## Summary

In this lesson, you learned how to use selectors, declarations, properties, shorthand properties, and values to style a series of simple HTML elements. In the next lesson, you will learn about the different types of selectors and how to use them.



# LESSON 3

## Selectors in Action

*In this lesson, you will learn about the different types of selectors and how to use them.*

### Setting Up the HTML Code

Selectors are one of the most important aspects of CSS because they are used to “select” elements on an HTML page so that they can be styled. The HTML code for this lesson is shown in Listing 3.1.

#### **LISTING 3.1** HTML Code Containing Markup for Lesson 3

---

```
<body>
<div id="content">
 <h1>
 Heading here
 </h1>
 <p class="intro">
 Lorem ipsum dolor sit amet.
 </p>
 <p>
 Lorem ipsum dolor sit amet.
 </p>
</div>
<div id="nav">

 item 1
 item 2
 item 3

</div>
<div id="footer">
 Lorem ipsum dolor sit amet.
</div>
</body>
```



**What Is a `<div>`?** The `<div>` element is a generic container that can be used to add structure to an HTML document. Although it is a block level element, it does not add any other presentation to the content.

For this lesson, the `<div>` element has been used to contain logical divisions of content, such as navigation and footer information.

These divisions of content can then be styled to suit your needs using descendant selectors, which are covered later in this lesson.

## Type Selectors

Type selectors will select any HTML element on a page that matches the selector.

In the HTML sample shown in Listing 3.1, there are seven HTML elements that could be used as type selectors, including `<body>`, `<div>`, `<h1>`, `<p>`, `<ul>`, `<li>`, and `<a>`.

For example, to select all `<li>` elements on the page, the `<li>` selector is used as shown in Listing 3.2.

### **LISTING 3.2** CSS Code Containing Styling for the `<li>` Element

---

```
li
{
 color: blue;
}
```

## Class Selectors

Class selectors can be used to select any HTML element that has been given a class attribute.

In the HTML sample shown in Listing 3.1, there are two HTML elements with class attributes—`<p class="intro">` and `<a href="#" class="intro">`.

For example, to select all instances of the `intro` class, the `.intro` selector is used as shown in Listing 3.3.

### LISTING 3.3 CSS Code Containing Styling for the `.intro` Class

---

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

You also can select specific instances of a class by combining type and class selectors. For example, you might want to select the `<p class="intro">` and the `<a href="#" class="intro">` separately. This is achieved using `p.intro` and `a.intro` as shown in Listing 3.4.

### LISTING 3.4 CSS Code Containing Two Different Stylings of the `.intro` Class

---

```
p.intro
{
 color: red;
}

a.intro
{
 color: green;
}
```

## ID Selectors

ID selectors are similar to class selectors. They can be used to select any HTML element that has an ID attribute. However, each ID can be used

only once within a document, whereas classes can be used as often as needed.

In this lesson, IDs are used to identify unique parts of the document structure, such as the content, navigation, and footer.

In the HTML sample shown in Listing 3.1, there are three IDs:

`<div id="content">`, `<div id="nav">`, and `<div id="footer">`. To select `<div id="nav">`, the `#nav` selector is used as shown in Listing 3.5.

### LISTING 3.5 CSS Code Containing the `#nav` ID Selector

```
#nav
{
 color: blue;
}
```



**Should You Use ID or Class?** Classes can be used as many times as needed within a document. IDs can be applied only once within a document. If you need to use the same selector more than once, classes are a better choice.

However, IDs have more weight than classes. If a class selector and ID selector apply the same property to one element, the ID selector's value would be chosen. For example, `h2#intro { color: red; }` will override `h2.intro { color: blue; }`.

## Descendant Selectors

Descendant selectors are used to select elements that are descendants of another element.

For example, in the HTML sample shown in Listing 3.1, three `<a>` elements are descendants of the `<li>` elements. To target these three `<a>` elements only, and not all other `<a>` elements, a descendant selector can be used as shown in Listing 3.6. This selector targets any `<a>` element that is nested inside an `<li>` element.



**LISTING 3.6** CSS Code Containing Descendant Selector

---

```
li a
{
 color: green;
}
```

Descendant selectors do not have to use direct descendant elements. For example, the `<a>` element is a descendant of `<div id="nav">` as well as the `<li>` element. This means that `#nav a` can be used as a selector as well (see Listing 3.7).

**LISTING 3.7** CSS Code Containing Descendant Selector

---

```
#nav a
{
 color: red;
}
```

Descendant selectors also can include multiple levels of descendants to be more specific as shown in Listing 3.8.

**LISTING 3.8** CSS Code Containing Descendant Selector

---

```
#nav ul li a
{
 color: green;
}
```

## Universal Selectors

Universal selectors are used to select any element. For example, to set the margins and padding on every element to 0, `*` can be used as shown in Listing 3.9.

**Listing 3.9** CSS Code Containing the Universal Selector

---

```
*
{
 margin: 0;
 padding: 0;
}
```

Universal selectors also can be used to select all elements within another element as shown in Listing 3.10. This will select any element inside the `<p>` element.

---

**LISTING 3.10** CSS Code Containing the Universal Selector Within the `<p>` Element

---

```
p *
{
 color: red;
}
```

## Advanced Selectors

Child selectors are used to select an element that is a direct child of another element (parent). Child selectors will not select all descendants, only direct children. For example, you might want to target an `<em>` that is a direct child of a `<div>`, but not other `<em>` elements that are descendants of the `<div>`. The selector is shown in Listing 3.11.

Child selectors are not supported by Windows Internet Explorer 5, 5.5, and 6, but are supported by most other standards-compliant browsers.

---

**LISTING 3.11** CSS Code Containing the Child Selector

---

```
div > em
{
 color: blue;
}
```

Adjacent sibling selectors will select the sibling immediately following an element. For example, you might want to target an `<h3>` element, but only `<h3>` elements that immediately follow an `<h2>` element. This is a commonly used example because it has a real-world application. There is often too much space between `<h2>` and `<h3>` elements when they appear immediately after each other. The selector is shown in Listing 3.12.

Adjacent sibling selectors are not supported by Windows Internet Explorer 5, 5.5, and 6, but are supported by most other standards-compliant browsers.

**LISTING 3.12** CSS Code Containing the Adjacent Sibling Selector

---

```
h2 + h3
{
 margin: -1em;
}
```

Attribute selectors are used to select elements based on their attributes or attribute value. For example, you might want to select any image on an HTML page that is called "small.gif" as shown in Listing 3.13.

Attribute selectors are not supported by Windows Internet Explorer 5, 5.5, and 6, or Macintosh Internet Explorer 5. They are also not supported by earlier versions of Opera.

**LISTING 3.13** CSS Code Containing the Attribute Selector

---

```
img[src="small.gif"]
{
 border: 1px solid #000;
}
```

Pseudo-elements enable you to style information that is not available in the document tree. For instance, using standard selectors, there is no way to style the first letter or first line of an element's content. However, the content can be styled using pseudo-elements as shown in Listing 3.14.

Pseudo-elements `:before` and `:after` are not supported by Windows Internet Explorer 5, 5.5, and 6, or Macintosh Internet Explorer 5. They are also not supported by earlier versions of Opera.

**LISTING 3.14** CSS Code Containing the Psuedo-Element Selector

---

```
p:first-line
{
 font-weight: bold;
}

p:first-letter
{
 font-size: 200%; font-weight: bold;}
```

Pseudo-classes enable you to format items that are not in the document tree. They include `:first-child`, `:link`, `:visited`, `:hover`, `:active`, `:focus`, and `:lang(n)`. Pseudo-classes are covered in Lesson 10, “Styling Links.”

## Summary

In this lesson, you learned how to use a range of selectors, including type, class, ID, descendant, and universal. You also learned about the difference between ID and class selectors. In the next lesson, you will learn how to apply inline styles, header styles, and styles within external style sheets.



## LESSON 4

# Applying Styles

*In this lesson, you will learn the three different locations where you can place CSS code, including inline, header, and external style sheets. You will also learn how to target a style sheet to a specific device such as a cell phone, television, or PDA by using media types. You will also learn methods that can be used to hide advanced styles from older browsers using media types.*

## Setting Up the HTML Code

The HTML code for this lesson that contains a single paragraph of text is shown in Listing 4.1.

### **LISTING 4.1** HTML Code Containing the Markup for Lesson 4

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
 <meta http-equiv="content-type" content="text/html;
 charset=utf-8">
 <title>Lesson 4</title>
</head>
<body>
<p>
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
</body>
</html>
```

In this lesson, the `<p>` element will be styled with font-family, width, background-color, margin, and padding.

These styles can be applied to `<p>` elements using three methods: *inline* styles, *header* styles, and *external* style sheets. Although each method will be explained, external style sheets are the preferred option because they do not add CSS to the HTML markup.

## Applying Inline Styles

Inline styles can be applied directly to elements in the HTML code using the `style` attribute. However, inline styles should be avoided wherever possible because the styles are added to the HTML markup. This defeats the main purpose of CSS, which is to apply the same styles to as many pages as possible across your website using external style sheets. Styles that are applied inline can cause additional maintenance across a website because multiple pages might need changing rather than one CSS file.

An example of an inline style is shown in Listing 4.2.

### LISTING 4.2 HTML Code Containing an Inline Style for the `<p>` Element

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
 <meta http-equiv="content-type" content="text/html;
 charset=utf-8">
 <title>Lesson 4</title>
</head>
<body>
<p style="font-family: arial, helvetica, sans-serif; margin:
 1em;padding: 1em; background-color: gray; width: 10em;">
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
</body>
</html>
```

## Using Header Styles

Header styles also can be used to style the `<p>` element. The CSS rules can be placed in the head of the document using the `style` element. Like

inline styles, header styles should be avoided where possible because the styles are added to the HTML markup rather than in external CSS files.

There are cases where header styles might be the preferred option in specific instances, such as a CSS rule that is specific to one page within a large website. Rather than add this rule to an overall CSS file, a header style may be used.

An example of a header style is shown in Listing 4.3. The `type="text/css"` attribute must be specified within the `style` element in order for browsers to recognize the file type.

---

**LISTING 4.3** HTML Code Containing Header Styles

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
 <meta http-equiv="content-type" content="text/html;
 charset=utf-8">
 <title>Lesson 4 - listing 2</title>
 <style type="text/css" media="screen">
 p
 {
 font-family: arial, helvetica, sans-serif;
 margin: 1em;
 padding: 1em;
 background-color: gray;
 width: 10em;
 }
 </style>
</head>
<body>
<p>
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
</body>
</html>
```

## Using External Style Sheets

The third method of applying styles to a document involves linking to external style sheets. External style sheets are the most appropriate

method for styling documents. If styles need to be changed, the modifications can take place in one CSS file rather than all HTML pages.

To change the header style to an external style, move the rule set to a new CSS file as shown in Listing 4.4.

Next, link to this style sheet from your HTML file using the `link` element as shown in Listing 4.5.

---

**LISTING 4.4** CSS Code Containing an External Style Sheet with Styles for the `<p>` Element

---

```
p
{
 font-family: arial, helvetica, sans-serif;
 margin: 1em;
 padding: 1em;
 background-color: gray;
 width: 10em;
}
```

---

**LISTING 4.5** HTML Code Containing the `link` Element

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
 <meta http-equiv="content-type" content="text/html;
 charset=utf-8">
 <title>Lesson 4</title>
 <link rel="stylesheet" href="style.css" type="text/css"
 media="screen">
</head>
<body>
<p>
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
</body>
</html>
```



## @import Styles

Header and external style sheets also can import other style sheets using the @import rule as shown in Listing 4.6. The @import rule must be placed before all other rules in the header or external style sheet.

### LISTING 4.6 CSS Code Containing an Imported Style Sheet

---

```
@import "advanced.css";
```

```
p
{
 font-family: arial, helvetica, sans-serif;
 margin: 1em;
 padding: 1em;
 background-color: gray;
 width: 10em;
}
```

Imported styles can be used to link to multiple CSS files as well as to hide styles from older browsers.

## Hiding Styles from Older Browsers

Some older browsers, such as Netscape Navigator 4 and Internet Explorer 4, have poor support for CSS. It is possible to hide styles from these browsers using specific media types and @import rules.

All styles will be hidden from Netscape Navigator 4 by changing the link element's media type from screen to screen, projection as shown in Listing 4.7. Netscape Navigator 4 does not support multiple media types.

### LISTING 4.7 HTML Code Containing a link Element with a screen, projection Media Type

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
<head>
 <meta http-equiv="content-type" content="text/html;
 charset=utf-8">
 <title>Lesson 4</title>
```

*continues*

```
<link rel="stylesheet" href="style.css" type="text/css"
 media="screen, projection">
</head>
<body>
<p>
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
</body>
</html>
```

The remaining styles will be hidden from Internet Explorer 4 and several other older browsers by moving the <p> element rule set out of the original style sheet and into the imported style sheet as shown in Listings 4.8 and 4.9. Internet Explorer 4 is not able to read imported files.

---

**LISTING 4.8** CSS Code Inside the Original Style Sheet Called `style.css`

```
@import "advanced.css";
```

---

**LISTING 4.9** CSS Code Inside the Import Style Sheet Called `advanced.css`

```
p
{
 font-family: arial, helvetica, sans-serif;
 margin: 1em;
 padding: 1em;
 background-color: gray;
 width: 10em;
}
```

All modern browsers will read the multiple media type `screen, projection`, as well as the imported style, so they will display the fully styled <p> element.

Header styles also can be hidden from older browsers by enclosing the contents of the `style` element inside a comment as shown in Listing 4.10.

**LISTING 4.10** HTML Code Containing Header Styles Within a Comment

---

```
<style type="text/css" media="screen">
<!--
 p
 {
 font-family: arial, helvetica, sans-serif;
 margin: 1em;
 padding: 1em;
 background-color: gray;
 width: 10em;
 }
-->
</style>
```

## Summary

In this lesson, you learned how to apply inline, header, and external styles to a document. You also learned what a media type is and how to hide advanced styles from older browsers using multiple media types and `@import`. In the next lesson, you will learn about the CSS box model including margin, background color, background image, padding, and border.

## LESSON 5

# Getting to Know the CSS Box Model

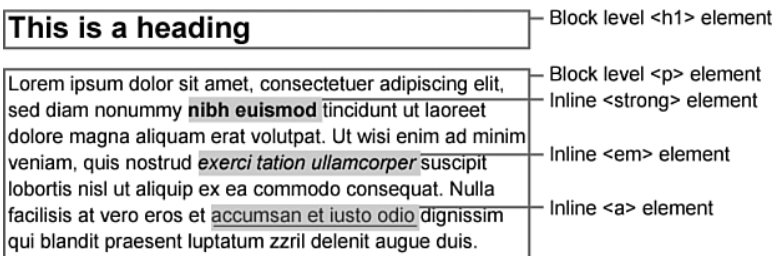


*In this lesson, you will learn about the CSS box model—the rectangular boxes that are generated for all HTML elements. You will learn about aspects that make up the box model, including margin, background color, background image, padding, and border. You also will learn the difference between inline and block level elements.*

## Understanding Inline and Block Level Elements

Block level elements are normally displayed as blocks with line breaks before and after. Examples of block level elements include paragraphs, headings, divs, and block quotes.

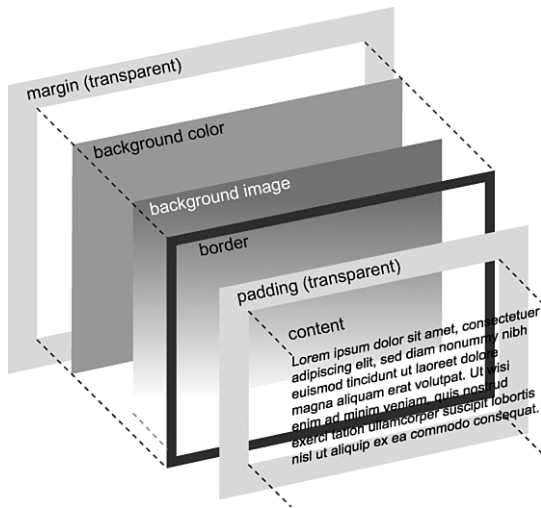
Inline elements are not displayed as blocks. The content is displayed in lines and there are no line breaks before and after. Examples of inline elements include emphasized text, strong text, and links. Examples of both block and inline elements are shown in Figure 5.1.



**FIGURE 5.1** Examples of block level and inline elements.

All block level and inline elements are boxes that use the box model. Both types of elements can be styled with box model properties such as margin, background-color, background-image, padding, and border as shown in Figure 5.2.

Some box model properties, such as height and width, do not apply to inline elements. Also, margin and padding applied to an inline element will affect content on either side, but not content above or below.



**FIGURE 5.2** Three-dimensional diagram of the CSS box model.

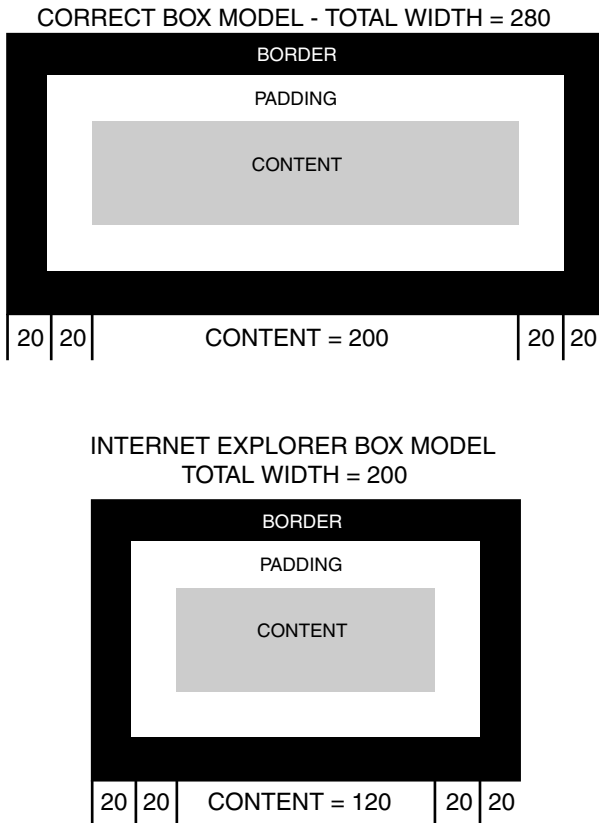
## Setting Box Width

The width of an element is applied to the content area. Other measurements, such as padding, border, and margins, are added to this width.

For example, if an element is specified with `width: 200px;`, the content area is 200px wide. If padding, border, or margin are applied to the same element, their measurements are added to the overall width.

However, Internet Explorer 5 for Windows (and Internet Explorer 6 for Windows in some circumstances) will use a different method to set widths

for boxes. If padding and border are applied to an element, their measurements are subtracted from the overall width. This is shown in Figure 5.3.



**FIGURE 5.3** CSS box model showing Internet Explorer width problem.

## Margins

Margins can be applied to the outside of any block level or inline element. They create space between the edge of an element and the edge of any adjacent elements.

Margins can be applied to individual sides of a box as shown in Listing 5.1.

---

**LISTING 5.1** CSS Code Containing Various margin Properties

---

```
p { margin-top: 0; }
p { margin-right: 2em; }
h2 { margin-bottom: 3em; }
h3 { margin-left: 1em; }
```

Margins also can be applied using a single shorthand property. If one margin value is specified, it applies to all sides of an element as shown in Listing 5.2.

---

**LISTING 5.2** CSS Code Containing the margin Shorthand Property with One Value Specified

---

```
p { margin: 1em; }
```

If two values are specified, the top and bottom margins are set to the first value and the right and left margins are set to the second as shown in Listing 5.3.

---

**LISTING 5.3** CSS Code Containing the Shorthand margin Property with Two Values Specified

---

```
p { margin: 1em 0; }
```

If three values are specified, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third as shown in Listing 5.4.

---

**LISTING 5.4** CSS Code Containing the Shorthand margin Property with Three Values Specified

---

```
p { margin: 1em 0 2em; }
```

If four values are specified, they apply to the top, right, bottom, and left as shown in Listing 5.5.

---

**LISTING 5.5** CSS Code Containing the Shorthand margin Property with Four Values Specified

---

```
p { margin: 1em 2em 2em 1em; }
```

# Background Color and Background Image

The `background-color` property sets the background color of an element.

The `background-image` property applies a background image to an element, which will appear on top of any `background-color`. Background images are covered in more detail in Lesson 6, “Adding Background Images.”

## Padding

Padding can be applied to the outside edges of the content area of any block level or inline element. Padding creates the space between the edge of the element and its content.

Like margins, padding can be applied to individual sides of a box as shown in Listing 5.6.

### **LISTING 5.6** CSS Code Containing Various padding Properties

---

```
p { padding: 1em; }
h1 { padding-top: 0; }
h2 { padding-right: 2em; }
h2 { padding-bottom: 3em; }
h3 { padding-left: 1em; }
```

Padding also can be applied using a single shorthand property. If one padding value is specified, it applies to all sides of an element as shown in Listing 5.7.

### **LISTING 5.7** CSS Code Containing the Shorthand padding Property with One Value Specified

---

```
p { padding: 1em; }
```

If two values are specified, the top and bottom margins are set to the first value and the right and left margins are set to the second as shown in Listing 5.8.



**LISTING 5.8** CSS Code Containing the Shorthand padding Property with Two Values Specified

---

```
p { padding: 1em 0; }
```

If three values are specified, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third as shown in Listing 5.9.

**LISTING 5.9** CSS Code Containing the Shorthand margin Property with Three Values Specified

---

```
p { padding: 1em 0 2em; }
```

If four values are specified, they apply to the top, right, bottom, and left as shown in Listing 5.10.

**LISTING 5.10** CSS Code Containing the Shorthand padding Property with Four Values Specified

---

```
p { padding: 1em 2em 2em 1em; }
```

## Border

The border properties specify the width, color, and style of the border of an element. Shorthand border properties include `border-top`, `border-bottom`, `border-right`, `border-left`, and `border` as shown in Listing 5.11.

**LISTING 5.11** CSS Code Containing Various Shorthand border Properties

---

```
p { border-top: 1px solid red; }
p { border-right 1px solid red; }
p { border-bottom: 1px solid red; }
p { border-left: 1px solid red; }
p { border: 1px solid red; }
```

## Content Area

The content area of a box can be given width, height, and color. Width and height can be specified in points (equal to 1/72 of an inch), picas

(equal to 12 points), pixels, ems, exes, millimeters, centimeters, inches, or percents as shown in Listing 5.12.

---

**LISTING 5.12** CSS Code Containing Various width and height Values

---

```
p { width: 100pt; }
p { height: 20pc; }
p { width: 300px; }
p { height: 40em; }
p { width: 50ex; }
p { height: 600mm; }
p { width: 70cm; }
p { height: 8in; }
p { width: 50%; }
```

The color property can be used to style the text color. Color can be specified in a number of ways, including keywords, hexadecimal RGB, and functional notation RGB.

Keywords for color include aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. Although other keywords might work in some browsers, they are not part of the specification and should not be used.

Hexadecimal colors can be specified using only three or six hexadecimal characters as shown in Listing 5.13. When a color has three pairs of hexadecimal digits (such as #ff0000), it can be shortened by removing one digit from each pair (#f00). RGB colors can be specified using three comma-separated integer or percentage values. For example, the color red can be specified using either rgb(255, 0, 0) or rgb(100%, 0%, 0%) as shown in Listing 5.13.

---

**LISTING 5.13** CSS Code Containing Various color Values

---

```
p { color: red }
p { color: #f00 }
p { color: #ff0000 }
p { color: rgb(255,0,0) }
p { color: rgb(100%, 0%, 0%) }
```

## Summary

In this lesson, you learned about the CSS box model including margin, background color, background image, padding, and border. You also learned the difference between inline and block level elements and how some versions of Internet Explorer misinterpret the box model. In the next lesson, you will learn how to apply a background image to the <body> element. You also will learn how to apply the background-repeat and background-position properties.

## LESSON 6

# Adding Background Images



*In this lesson, you will learn how to apply a background image to the <body> element. You will also learn how to apply the background-repeat and background-position properties. The aim is to create a gradient image that repeats down the right edge of the page.*

## Setting Up the HTML Code

The HTML code for this lesson will be comprised of three paragraphs of text as shown in Listing 6.1.

### **LISTING 6.1** HTML Code Containing the Markup for Lesson 6

---

```
<p>
 Lorem I
psum dolor sit amet...
</p>
<p>
 Ut wisi enim ad minim veniam...
</p>
<p>
 Duis autem vel eum iriure dolor...
</p>
```

## Creating Selectors to Style the Header

To style the `<body>` element and its content, you will only need a single type selector as shown in Listing 6.2.

### LISTING 6.2 CSS Code Showing the Selector to Style the Body

---

```
body { ... }
```

## Adding background-image

The `background-image` property is used to add a background image to the `<body>` element.

Values for the `background-image` property are either a `url` (to specify the image) or `none` (when no image is used).

For this lesson, you will use `url(chapter6.jpg)`. The image path can be written with or without quotation marks. The `background-image` code is shown in Listing 6.3. The results can be seen in Figure 6.1.

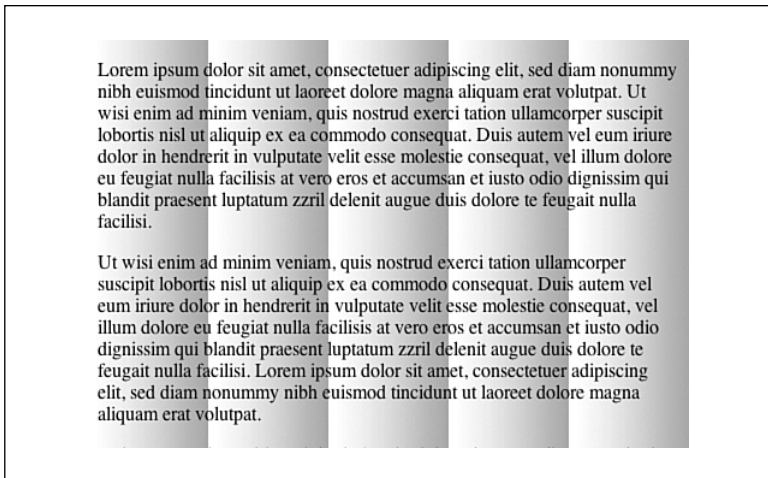
### LISTING 6.3 CSS Code Styling the `<body>` Element with a Background Image

---

```
body
{
 background-image: url(chapter6.jpg);
}
```



**Background Images and Internet Explorer 5 for Macintosh** Internet Explorer 5 for Macintosh will not render background-images if quotations are used around image paths. Because quotation marks are not needed, it is simpler and safer to leave them out.



**FIGURE 6.1** Screenshot of styled `<body>`.

## Setting background-repeat

The background image in this lesson is now repeating across the screen. This can be controlled using `background-repeat`.

Values for the `background-repeat` property (see Figure 6.2) include `repeat` (where the image is repeated both horizontally and vertically), `repeat-x` (where the image is repeated horizontally only), `repeat-y` (where the image is repeated vertically only), and `no-repeat` (where the image is not repeated).

In this lesson, you will use `repeat-y`, as shown in Listing 6.4, to force the image to repeat vertically down the page.

---

### **LISTING 6.4** CSS Code Setting background-repeat

```
body
{
 background-image: url(chapter6.jpg);
 background-repeat: repeat-y;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 6.2** Screenshot of `<body>` styled with `background-repeat`.

## Adding background-position

Now that the background image is repeating correctly, it must be positioned down the right edge of the `<body>` element. This is achieved using `background-position`.

Values for the `background-position` property include percentage (such as `0 100%`), length (such as `2px 20px`), and keywords (such as `left top`). In each case, the horizontal position is specified first, and then the vertical position. The values `0% 0%` will position the upper-left corner of the image in the upper-left corner of the box's padding edge. Values of `0 100%` will position the bottom-left corner of the image in the bottom-left corner of the box's padding edge. Values of `2px 20px` will position the top-left corner of the image 2px in from the left edge of the box and 20px down from the top of the box.

If only one percentage or length value is given, it sets the horizontal position only and the vertical position will be `50%`. If two values are given, the horizontal position comes first. Combinations of length and percentage values are allowed (such as `50% 2cm`). Negative positions are also allowed (such as `-20px 10px`).

For this lesson, you will use percentage values of `100% 0`, which will place the image in the right and top of the element. The code is shown in Listing 6.5.

The image will now repeat down the right edge of the `<body>` element (see Figure 6.3).



**Background Position Issues** Some browsers do not recognize the `background-position` keyword value `right`. However, all modern browsers support the percentage value of `100%`, so this value can be used instead.

#### LISTING 6.5 CSS Code Styling the `<body>` Element with `background-position`

```
body
{
 background-image: url(chapter6.jpg);
 background-repeat: repeat-y;
 background-position: 100% 0;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 6.3** Screenshot of `<body>` element styled with `background-position`.



## Using the background Shortcut

As discussed in Lesson 2, “Using CSS Rules,” shorthand properties allow the values of several properties to be specified within a single property. The background property can be used to combine background-color, background-image, background-repeat, background-attachment, and background-position.

When sorting shorthand properties, browsers will first set all the individual properties to their initial values, and then override these with values specified by the author.

A default background rule would be set to background: transparent none repeat scroll 0 0;. If the declarations used in this lesson are combined into the shorthand rule, they will override the default values for background-image, background-repeat, and background-position. The result will be background: transparent url(chapter6.jpg) repeat-y scroll 100% 0;.

However, the rule can be shortened to include only the values that are needed, so the final declaration will be background: url(chapter6.jpg) repeat-y 100% 0; (see Listing 6.6).

### LISTING 6.6 CSS Code Styling the <body> Element with a Shorthand background Property

---

```
body
{
 background: url(chapter6.jpg) repeat-y 100% 0;
}
```

## Adding padding

The final step will be to add padding to the <body> element to push the text away from the background-image. This can be achieved using the shorthand padding declaration padding: 1em 80px 1em 1em; as shown in Listing 6.7. This will place 1em of padding on the top, bottom, and left of the <body> and 80px on the right edge. The results can be seen in Figure 6.4.

**LISTING 6.7** CSS Code Adding padding to the <body> Element

```
body
{
 background: url(chapter6.jpg) repeat-y 100% 0;
 margin: 0;
 padding: 1em 80px 1em 1em;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 6.4** Screenshot of <body> element with padding applied.

## Summary

In this lesson, you learned how to apply a background image to the <body> element. You also learned how to apply the background-repeat, background-position, and shorthand background properties. In the next lesson, you will learn how to style text using the font, size, color, and alignment properties.



# LESSON 7

## Formatting Text

*In this lesson, you will learn how to style text using font, size, alignment, and color properties instead of the <font> element.*

### Setting Up the HTML Code

The HTML code for this lesson contains three paragraphs of text as shown in Listing 7.1. The contents of these paragraphs are wrapped inside <font> elements. The first paragraph has been set to a larger font size. It also has been colored and styled in bold and italic. The results are shown in Figure 7.1.

#### **LISTING 7.1** HTML Code Containing Markup for Lesson 7

---

```
<p align="center">
 <font size="4" color="#990000" face="times, times new
 roman">
 <i>Lorem ipsum dolor sit amet, consectetuer adipiscing
 elit...</i>
</p>
<p>

 Ut wisi enim ad minim
 veniam, quis nostrud exerci...
</p>
<p>

 Duis autem vel eum iriure dolor in hendrerit vulputate...

</p>
<p>
 <p> nostrud
 exerci...

</p>
```

*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo.*

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit culputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis ero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 7.1** Screenshot of <font>-styled paragraphs.

## Removing Font Elements

Instead of using <font> elements throughout a document, you should use CSS to style the content. This reduces the overall file size and makes future maintenance easier. All font-styling information can be stored in one external file, rather than scattered throughout every document in a website.

The <font> elements will be removed from the HTML markup as shown in Listing 7.2. The first paragraph will be styled with an introduction class because it will need additional styling (see Figure 7.2).

**LISTING 7.2** HTML Code Containing the Markup Without `<font>` Elements

---

```
<p class="introduction">
 Lorem ipsum dolor sit amet, consectetur adipiscing
 elit...
</p>
<p>
 Ut wisi enim ad minim veniam, quis nostrud exerci...
</p>
<p>
 Duis autem vel eum iriure dolor in hendrerit vulputate...
</p>
 Ut wisi enim ad minim veniam, quis nostrud exerci...
</p>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis ero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 7.2** Screenshot of paragraphs with `<font>` elements removed.

## Creating the Selectors

To style the paragraphs, two selectors will be used as shown in Listing 7.3.

**LISTING 7.3** CSS Code Showing the Selectors to Style the Paragraphs

---

```
p {...}
p.introduction {...}
```

## Styling the <p> Element

The font family is set using the `font-family` property. A range of fonts should always be included, separated by commas. A generic font family must be included at the end of the list. If a user does not have the initial font family, his or her browser will look for the second font family. If no font family matches are found, the browser will fall back to the generic font family.



**Generic Font Families** Generic font families are a fallback mechanism to provide some basic font styling if none of the specified font families are available. The five generic font families are serif, sans-serif, cursive, fantasy, and monospace.

The `font-size` property will be set to 80%, which will make it 80% of the user's default browser style. Using percentages will allow the user to control the overall size of fonts (see Figure 7.3).



**Ems and Percents** In theory, there is no difference between using ems or percents for font sizing. However, Internet Explorer 5 for Windows will misread `em` measurements below 100% and change the unit from ems to pixels. For example, a value of `.8em` will be displayed at 8px.

To avoid this problem, `font-size` should be set using percentage units for any value below 100%.

Finally, a `line-height` of 140% will be included to provide space between each line and make the text more readable. The default `line-height` for most browsers is 120%. Setting a value of 140% will add 20% additional space between each line. The rule set is shown in Listing 7.4.

### Listing 7.4 CSS Code Containing Styles for the <p> Element

```
p
{
 font-family: arial, helvetica, sans-serif;
 font-size: 80%;
 line-height: 140%;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis eros eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 7.3** Screenshot of styled paragraphs.

## Styling the First Paragraph

The first paragraph in this example will use different fonts than the other paragraphs. In this case, it will be styled with `times`, `"times new roman"`, `serif`. Fonts such as Times New Roman, which have spaces in their names, should always be wrapped in quotation marks.

The next step is to style the text italic and bold. This is achieved using `font-style: italic;` and `font-weight: bold;`.

To align the text in the center of the screen, use `text-align: center;`.

The font size can be increased using `font-size: 110%;` and the font color can be set using `color: #900;` as shown in Listing 7.5 (see Figure 7.4).

**LISTING 7.5** CSS Code Containing Styles for the First Paragraph

---

```
p
{
 font-family: arial, helvetica, sans-serif;
 font-size: 80%;
 line-height: 1.4;
}

p.introduction
{
 font-family: times, "times new roman", serif;
 font-style: italic;
 font-weight: bold;
 text-align: center;
 font-size: 110%;
 color: #900;
}
```

*Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  
diam nonummy nibh euismod tincidunt ut laoreet dolore magna  
aliquam erat volutpat. Ut wisi enim ad minim veniam quis  
nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip  
ex ea commodo.*

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit  
lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor  
in hendrerit vulputate velit esse molestie consequat, vel illum dolore eu feugiat  
nulla facilisis eros eros et accumsan et iusto odio dignissim qui blandit praesent  
luptatum zzril.

Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie  
consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et  
iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te  
feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  
diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat  
volutpat.

**FIGURE 7.4** Screenshot of styled first paragraph.

## Converting to Shorthand

As discussed in Lesson 2, “Using CSS Rules,” shorthand properties are easier to write and maintain than longhand properties. They also make CSS files more concise.



The `<p>` element can be styled so that `font-size`, `line-height`, and `font-family` are declared as a single font property.

The `introduction` class can be styled so that `font-style`, `font-weight`, `font-size`, `line-height`, and `font-family` are declared as a single font property as shown in Listing 7.6.

---

**LISTING 7.6** CSS Code Containing the Shorthand Styles

---

```
p
{
 font: 80%/1.4 arial, helvetica, sans-serif;
}

p.introduction
{
 font: bold italic 110%/1.4 times, "times new roman",
 serif;
 text-align: center;
 color: #900;
}
```

## Summary

In this lesson, you learned how to style text using `font-family`, `font-size`, `line-height`, `font-style`, `font-weight`, `text-align`, and `color`. You also learned how to use the shorthand font property. In the next lesson, you will learn how to style a heading using `border`, `background-images`, and `text-transform` properties.

# LESSON 8

## Styling a Flexible Heading



*In this lesson, you will learn how to create a border above and below a heading, add a continuous gradient background image, and style the text with `text-transform` and letter spacing.*

### Styling the Heading

To style this heading, you will need a selector that targets the `<h1>` element. To make sure you don't target every `<h1>` on the page, you should also include a unique identifier, such as `header`, within the selector (see Listing 8.1).

#### **LISTING 8.1** CSS Code Showing the Selector to Style the Heading

---

```
h1#header { ... }
```

The HTML code used for this heading is shown in Listing 8.2.

#### **LISTING 8.2** HTML Code Containing the Markup for a Heading

---

```
<h1 id="header">
 Page Heading
</h1>
```



**Heading Levels and Document Structure** Web documents should use semantically correct markup to add meaning to the content. For example, headings should be placed inside heading elements, paragraphs of text should be placed inside paragraph elements, and lists should be placed inside list elements.

When the semantically correct HTML markup is in place, CSS can be used to visually style the content.

Heading levels are an important part of this markup. Ideally, web pages should start with a single `<h1>` element for the most significant information on the page, such as the page title or the site name.

Headings should never be faked using `<font>` or `<strong>` elements because they do not provide meaning for devices such as screen readers or text-based browsers.

## Adding Color, Font Size, and Weight

To add a color to the heading, use the `color` property. The color can be changed to suit your needs.

For this heading, you will set the `font-size` to `120%` and the `font-weight` to `normal`, as shown in Listing 8.3.

### **LISTING 8.3** CSS Code Setting font-size and font-weight

```
h1#header
{
 color: #036;

 font-size: 120%;
 font-weight: normal;
}
```



**Overriding Standard Heading Settings** Any HTML document may have three or more style sheets associated with it, including a *browser* style sheet, a *user* style sheet, and one or more *author* style sheets.

Browsers apply default style sheets to all web documents. Although these browser style sheets vary from browser to browser, they have common characteristics, such as black text and blue links.

Most modern browsers allow users to set their own style sheets within their browser. These user style sheets will override any browser default style sheets—for that user only.

As soon as you apply a basic style sheet or an inline style to a page, you have added an author style sheet. Author style sheets will generally override both browser and user style sheets.

Most browser style sheets will display an `<h1>` element in bold text at 200% of the default font size. If you style the `<h1>` element to 120%, this measurement will be used instead of 200% because your style sheet will override the browser style sheet.



**Using Shorthand Hexadecimal Colors** Hexadecimal colors can be specified using the `#` symbol immediately followed by three or six hexadecimal characters.

Three-digit hexadecimal values are converted to six-digit form by replicating digits. So, `#f00` is the same as `#ff0000` and `#f2a` is the same as `#ff22aa`.

## Setting Text Options

The next step is to center the heading, make it uppercase, and add some letter spacing. This can be achieved using the `text-transform`, `text-align`, and `letter-spacing` properties as shown in Listing 8.4 and illustrated in Figure 8.1. These options can be changed to suit your needs.

### LISTING 8.4 CSS Code Transforming Text

```
h1#header
{
 color: #036;
 font-size: 120%;
 font-weight: normal;
 text-transform: uppercase;
 text-align: center;
 letter-spacing: .5em;
}
```

P A G E H E A D I N G

**FIGURE 8.1** Screenshot of uppercase, centered heading.

## Applying Padding and Borders


Later in this lesson, you will be adding borders to the top and bottom of the heading. To avoid placing the text hard against the borders, you will need to add some top and bottom padding. You can use the shorthand padding property, setting top and bottom padding to `.4em`, and left and right padding to `0`.

To apply borders to the top and bottom of the heading, use the `border-top` and `border-bottom` properties as shown in Listing 8.5.

The results can be seen in Figure 8.2. The borders can be removed or changed to suit your needs.

### LISTING 8.5 CSS Code Adding Borders

```
h1#header
{
 color: #036;
 font-size: 120%;
 font-weight: normal;
 text-transform: uppercase;
 text-align: center;
 letter-spacing: .5em;
 padding: .4em 0;
 border-top: 1px solid #069;
 border-bottom: 1px solid #069;
}
```



P A G E H E A D I N G

FIGURE 8.2 Screenshot of bordered heading.



**CSS Borders** CSS Border properties define the borders around an element.

`border-color` sets the color of the border (for example, red, transparent, none, #036, #003366).

`border-width` sets the thickness of the border (for example, thin, medium, thick, 1px, .5em, 1ex).

`border-style` sets the appearance of the border (for example, none, hidden, dotted, dashed, solid, double, groove, ridge, inset, outset).

Border properties and values can be specified in many ways. The simplest option is to use the shorthand `border` property like this:

```
border: (width) (style) (color);
```

Some border styles, such as `dotted`, are not supported by Internet Explorer 5 or 5.5.

## Adding a Background Image

To add a background image to the heading, use the `background` property. You can then specify the `url` and the `repeat` value as shown in Listing 8.6. In this case, the image is set to `repeat-x`, so it will repeat across the `x` axis only.

The results can be seen in Figure 8.3. The background image can be removed or changed to suit your needs.

### LISTING 8.6 CSS Code Adding a Background Image

---

```
h1#header
{
 color: #036;
 font-size: 120%;
 font-weight: normal;
 text-transform: uppercase;
 text-align: center;
 letter-spacing: .5em;
 padding: .4em 0;
 border-top: 1px solid #069;
 border-bottom: 1px solid #069;
 background: url(chapter8.jpg) repeat-x;
}
```



**FIGURE 8.3** Screenshot of finished heading.

## Summary

In this lesson, you have learned how to style a flexible heading using `font-size`, `font-weight`, `borders`, `padding`, and `background` images. In the next lesson, you will learn how to style a round-cornered heading.

# LESSON 9

## Styling a Round-Cornered Heading



*In this lesson, you will learn how to wrap a round-cornered box around a heading. The box is made from two background images that adjust to suit any size heading.*

### Styling the Heading

To style this heading, you will need a selector that targets the `<h2>` element. To make sure you don't target every `<h2>` on the page, you should also include a class within the selector. A class is used in this case instead of an ID because you might want to include more than one of these fixed-width headings on a single page (see Listing 9.1).

#### **LISTING 9.1** CSS Code Showing the Selectors to Style the Heading

---

```
h2.decorative {...}
h2.decorative em {...}
```

The HTML code used for this heading is shown in Listing 9.2. Notice that the heading is wrapped inside an `<em>` (emphasis) element. This additional element will be important later in the lesson.



**LISTING 9.2** HTML Code Containing the Markup for a Heading

---

```
<h2 class="decorative">
 Section Heading
</h2>
```



**Creating the Scaleable Background Image** The heading in this lesson will eventually be wrapped inside a round-cornered box.

This box must be able to grow downward if the heading text is long, or if the user has chosen to use larger font sizes within her browser.

For this reason, the round-cornered box is made up of two background images. The first image is the top section of the box, and the second image is the bottom section of the box.

The first image must be very long, in order to grow downward as needed.

If the first background image is applied to the `<h2>` element, the second background image must be applied to another element.

One simple option is to wrap the heading text in an `<em>` element and apply the second image to this. As long as the second image is positioned at the bottom of the `<em>` element, the content can grow as needed.

## Styling the `<h2>` Element

To add a color to the `<h2>` element, use the `color` property. The color can be changed to suit your needs.

The font weight, size, and family are set using the `font` property, and the heading is centered using the `text-align` property as shown in Listing 9.3 and illustrated in Figure 9.1.

**LISTING 9.3** CSS Code Setting Text Alignment

```
h1#header
{
 color: #036;
 font: bold 100% arial, helvetica, sans-serif;
 text-align: center;
}
```

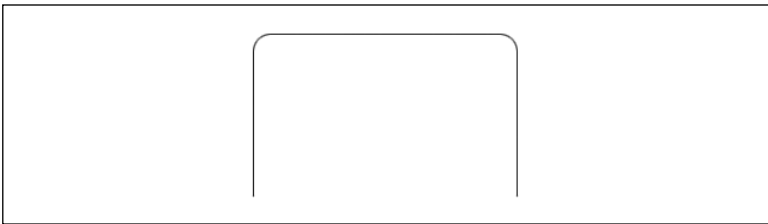


*Section Heading*

**FIGURE 9.1** Screenshot of styled heading.

## Adding a Background Image

To add a background image to the `<h2>` element, use the `background` property. The image should be set to `no-repeat` so it doesn't reappear in the middle of a long heading. The image is shown in Figure 9.2.



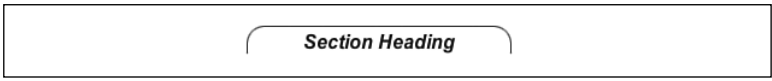
**FIGURE 9.2** Screenshot of the first image—which is applied to the `<h2>` element.

You will also need to set a width for this element because the background image is 220 pixels wide. If the width is left undefined, the heading will poke out the side of the round-cornered box.

You should also apply 5 pixels of padding on the top of the element as shown in Listing 9.4 and illustrated in Figure 9.3. This top padding will move the text down slightly so it doesn't sit hard against the background image.

**LISTING 9.4** CSS Code Setting Background Image, Width, and Padding

```
h1#header
{
 color: #036;
 font: bold 100% arial, helvetica, sans-serif;
 text-align: center;
 background: url(chapter9.gif) no-repeat;
 width: 220px;
 padding: 5px 0 0 0;
}
```

Section Heading**FIGURE 9.3** Screenshot of heading with background image.

## Styling the <em> Element

The <em> element will be used to house the second background image—the bottom of the round-cornered box. This means it must be given the same width as the <h2> element in order for the two images to line up properly.

However, because the <em> is an inline element, it will ignore any width that is specified. The solution is to set it to `display: block`; before applying a width.

Next, `padding-bottom` is used to move the text up slightly so it doesn't sit hard against the background image.

You can also override the <em> element's default italic style using `font-style: normal`;

Finally, you need to apply the background image using the background property. The image should be set to `no-repeat` so it does not reappear under the heading. The background position should be set to `0 100%` so the lower-left corner of the image will align with the lower-left corner of the element's edge.

The image is shown in Figure 9.4 and the CSS code is shown in Listing 9.5. The completed heading is shown in Figure 9.5.

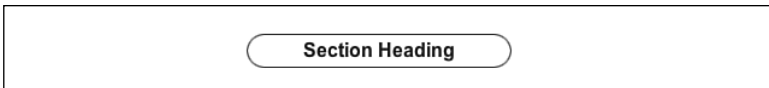


**FIGURE 9.4** Screenshot of the second image—which is applied to the `<em>` element.

### LISTING 9.5 CSS Code Styling the `<em>` Element

```
h1#header
{
 color: #036;
 font: bold 100% arial, helvetica, sans-serif;
 text-align: center;
 background: url(chapter9.gif) no-repeat;
 width: 220px;
 padding: 5px 0 0 0;
}

h2.decorative em
{
 display: block;
 width: 220px;
 padding: 0 0 5px 0;
 font-style: normal;
 background: url(chapter9a.gif) no-repeat 0 100%;
}
```



**FIGURE 9.5** Screenshot of the final heading.

## Summary

In this lesson, you have learned how to apply color, font, and text alignment to a heading. You have also learned how to set background images for two elements in order to achieve a round-corner box. In the next lesson you will learn about styling links.



# LESSON 10

## Styling Links

*In this lesson, you will learn how to style links. You also will learn how to add background images to links, turn off link underlines, use borders for underlines, and increase the active link area.*

### Links and Pseudo-Classes

You have already learned how to style `<a>` or link elements in Lesson 3, “Selectors in Action.” Now you will learn how to style the five different link states.

Links can be in the following states:

- **Normal**—The standard unvisited link state
- **Visited**—The link points to a URI that has already been visited
- **Hover**—The cursor is over the active area of the link
- **Active**—The moment the link is selected or clicked
- **Focus**—The link is in focus and ready to accept input, such as a click or mouse down

Some link states cannot occur at the same time. For example, a link can either be visited or unvisited—it cannot be both. However, visited and unvisited links can also be in the hover, active, and focus states.

Each of these states can be styled individually using `link` pseudo-classes (classes that do not exist in the document structure). The five link pseudo-classes are

- **`a:link`**—Styles unvisited link elements
- **`a:visited`**—Styles visited link elements

- **a:focus**—Styles the state during focus
- **a:hover**—Styles the state when the cursor moves over a link
- **a:active**—Styles the state when a link is activated

The five pseudo-classes are shown in Listing 10.1.

### LISTING 10.1 CSS Code Containing the Five Link Pseudo-Classes

---

```
a:link {...}
a:visited {...}
a:focus {...}
a:hover {...}
a:active {...}
```



**Focus and Active States** The `a:focus` pseudo-class highlights the tab position for people who use a keyboard to navigate.

Unfortunately, Internet Explorer for Windows does not support the `a:focus` pseudo-class. Instead, it uses the `a:active` pseudo-class for tab highlighting.

As an additional problem, Internet Explorer for Windows incorrectly applies the `a:active` pseudo-class. The `a:active` state remains visible until another action takes place.

## Setting Pseudo-Class Order

The five pseudo-classes have the same weight, so the order in which they are placed within a CSS file is important. Pseudo-class declarations that appear later in a CSS file will override those that appear earlier. The correct order is shown in Listing 10.2.

---

**LISTING 10.2** CSS Code Containing Correct Order of <a> Pseudo-Classes

---

```
a {...}
a:link {...}
a:visited {...}
a:focus {...}
a:hover {...}
a:active {...}
```

## Using Classes with Pseudo-Classes

Class selectors can be combined with pseudo-classes to create links for different purposes. For example, you might want to style links depending on whether they are internal or external.

A class could be added to all external links, and then these links could be styled using a combined class and pseudo-class selector as shown in Listing 10.3. The results can be seen in Figure 10.1.

---

**LISTING 10.3** CSS Code Demonstrating Combined Class and Pseudo-Class Selectors

---

```
a:link
{
 color: blue;
}

a:visited
{
 color: purple;
}

a.external:link
{
 color: red;
 font-weight: bold;
}

a.external:visited
{
 color: black;
 font-weight: bold;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 10.1** Screenshot showing difference between `a:link` and `a:external:link`.

## Styling Links with Background Images

Following on from the preceding example, it is possible to display a small icon beside every external link.

The first step is to create basic rules for the required link states. In this case, you can use `a:link`, `a:visited`, and `a:hover`.

Three new states are then added, specifically for links styled with an “external” class. These are `a.external:link`, `a.external:visited`, and `a.external:hover`.

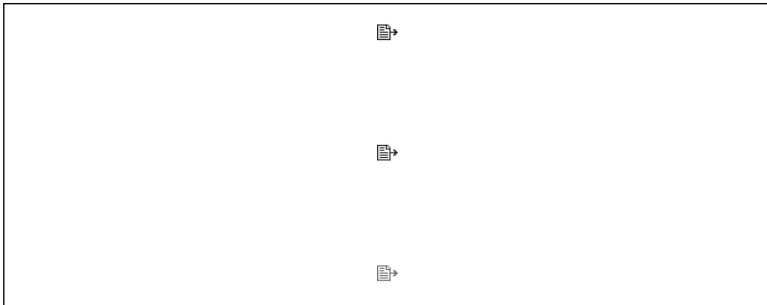
In each case, a background image is added to the link. The image is set to `no-repeat` so that it doesn’t tile across the background of the entire link. The position is set to `100%`, which will place the right edge of the image against the right edge of the link.

A single background image is used for all three states. The vertical `background-position` needs to change for each state. This means that a single image is loaded and cached, so there is no lag when the rollover image is required.

The `a:link` state has been set to `0`. The `a:visited` state has been set to `-100px`. The `a:hover` state has been set to `-200px`. The background image is shown in Figure 10.2.

Finally, padding has been used to push the link content away from the background image as shown in Listing 10.4 (see Figure 10.3).





**FIGURE 10.2** Screenshot of background image used to style the .external link.

**LISTING 10.4** CSS Code Containing the Markup to Style the .external Link

---

```
a:link
{
 color: blue;
}

a:visited
{
 color: purple;
}

a:hover
{
 color: red;
}

a.external:link
{
 background: url(chapter10.gif) no-repeat 100% 0;
 padding-right: 20px;
}

a.external:visited
{
 background: url(chapter10.gif) no-repeat 100% -100px;
 padding-right: 20px;
}

a.external:hover
```

*continues*

```
{
 background: url(chapter10.gif) no-repeat 100% -200px;
 padding-right: 20px;
}
```

Lorem ipsum dolor sit amet, [consectetuer](#) adipiscing elit, sed diam [nonummy nibh](#) euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 10.3** Screenshot showing styled `.external` link with a background image.

## Removing Underlines and Applying Borders

Some users, particularly those with poor eyesight, might find standard link underlines hard to read. This is particularly true for links that contain italic text.

One solution is to turn off text underlines and use borders.

The first step is to set the `text-decoration` to `none`. This will turn off link underlines.

Next, the required states need to be colored. In this case, `a:link` is set to blue, `a:visited` is set to purple, and `a:hover` is set to red.

Finally, borders are added to each state using `border-bottom` as shown in Listing 10.5. `padding-bottom` can be added to control the distance between the underline and the content, if required (see Figure 10.4).

### LISTING 10.5 CSS Code to Style Links with Borders

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

a:link
{
```

*continues*

**LISTING 10.5** Continued

```
 color: blue;
 border-bottom: 1px solid blue;
}

a:visited
{
 color: purple;
 border-bottom: 1px solid purple;
}

a:hover
{
 color: red;
 border-bottom: 1px solid red;
}
```

**Standard underlined links**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**Border-bottom underlined links**

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

**FIGURE 10.4** Screenshot showing difference between links with underlines and links with border-bottom.

## Increasing the Active Area of Links

For some users, particularly those with motor-skill difficulties, clicking on links can be difficult. Using CSS, the active area of links can be increased.

The first step is to add .5em of padding above and below the <a> element to increase the active area of a link. This is achieved using padding:

```
.5em 0;.
```

Next, the `<a>` element should be set to `position: relative`, which will stop the padding from affecting surrounding text, as shown in Listing 10.6 (see Figure 10.5).

To see the increased link area in action, you can apply a background color to the `<a>` element. This background color can be removed before it is applied in a real situation.

A more detailed explanation of this technique is available on David Benton's website at <http://www.dbenton.com/go/chronicles/2004/08/22/fitts-law-and-text-links/>.

### LISTING 10.6 CSS Code Containing Styles to Increase the Active Area of Links

---

```
a
{
 padding: .4em 0;
 position: relative;
 z-index: 1;
 background: yellow;
}
```

#### Standard links

Lorem ipsum dolor sit amet, [consectetur](#) adipiscing elit, sed  
diam [nonummy nibh](#) euismod tincidunt ut laoreet dolore magna  
aliquam erat volutpat.

#### Links with increased active area

Lorem ipsum dolor sit amet, [consectetur](#) adipiscing elit, sed  
diam [nonummy nibh](#) euismod tincidunt ut laoreet dolore magna  
aliquam erat volutpat.

**FIGURE 10.5** Screenshot showing difference between standard link area and links with increased active area.

## Summary

In this lesson, you learned how to style links and pseudo-classes. You learned how to apply background images and borders to the `<a>` element. You also learned how to increase the active area of links to make them more accessible. In the next lesson, you will learn how to position and style an image and its caption.

# LESSON 11

## Positioning an Image and Its Caption



*In this lesson, you will learn how to position and style an image and its caption.*

### Wrapping the Image and Caption

The first step is to wrap a container around the image and caption so they can be floated together. There are many elements that can be used such as paragraphs, lists, or even definition lists. However, for this lesson, you will use a `<div>` as shown in Listing 11.1.

#### **LISTING 11.1** HTML Code Containing the Markup for a Container, an Image, and Its Caption

---

```
<div class="imagecaption">

 A flower from my garden.
</div>
<p>
 Lorem ipsum dolor sit amet...
</p>
```

To make sure you don't target every `<div>` on the page, you should include a class within the selector as shown in Listing 11.2.

A class is used here instead of an ID because you might want to include more than one floated image and caption on a page.

**LISTING 11.2** CSS Code Showing the Selectors for Styling the Container

---

```
div.imagecaption {...}
div.imagecaption img {...}
```

## Floating the Container

To move the container across to the right edge of the browser window, use `float: right`.

When the container is floated, it must be given a width. In this case, you will use `width: 182px`.

Why the strange width? The image inside of this container is 180px wide. Later in this lesson, the image will be given a 1px-wide border. The right border width (1px), left border width (1px), and image width (180px) add up to 182px.

Next, you might want to create some space around the container so that text and other elements don't butt up against it (see Figure 11.1). You can achieve this by applying margins to the right, bottom, and left of the container as shown in Listing 11.3.

**LISTING 11.3** CSS Code Floating the Container

---

```
div.imagecaption
{
 float: right;
 width: 182px;
 margin: 0 1em 1em 1em;
 display: inline;
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.



A flower from my garden.

**FIGURE 11.1** Screenshot of floated container.



**Floats, Margins, and Internet Explorer 5** If you view samples from this lesson in Internet Explorer 5 and 5.5 for Windows, you will notice that the right margin is actually much wider than in other browsers. In fact, it is 2em wide—double the width it is supposed to be. This is Internet Explorer’s Double Margin Float Bug.

This bug occurs when you apply a right margin to a right floated element and it sits directly against the right edge of the parent container.

The opposite is also true. The bug will occur when you apply a left margin to a left floated element and it sits against the left edge of the parent container.

Luckily, there is a solution. Simply add `display: inline` to the rule set. All other browsers will ignore this declaration, but Internet Explorer 5 and 5.5 for Windows will then apply the correct margin width.

## Applying Padding, Background Color, and a Background Image

Now that the container is positioned, you can style its appearance using `padding`, `background-image`, and `background-color`.

The first step is to apply padding to the container to create space around the image and caption. You can use the shorthand padding rule to specify values for top, right, bottom, and left. All sides should be given a value of 10px except the bottom, which should be given a value of 70px. This will provide some space for the background image.

The background image can be applied using `background-image: url(chapter11.gif)` as shown in Figure 11.2.

You must set a repeat value to avoid the image repeating across the entire background area of the container. In this case, you will need the image to repeat across the x-axis. This is achieved using `background-repeat: repeat-x`.





**FIGURE 11.2** Screenshot of background image.

The background image needs to be aligned with the bottom of the container. One way to achieve this is to set the vertical value to 100%, which will place the bottom of the image against the bottom of the padding area. The declaration would be `background-position: 0 100%`.

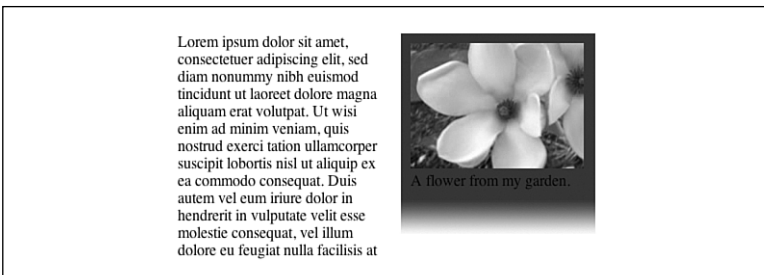
The background color can be specified using `background-color: #036`.

All of these background declarations can be condensed into one shorthand declaration as shown in Listing 11.4. The results are illustrated in Figure 11.3.

#### **LISTING 11.4** CSS Code Setting Padding, Background Color, and Background Image

---

```
div.imagecaption
{
 float: right;
 width: 182px;
 margin: 0 1em 1em 1em;
 display: inline;
 padding: 10px 10px 70px 10px;
 background: #036 url(chapter11.gif) repeat-x 0 100%;
}
```



**FIGURE 11.3** Screenshot of container with padding and background.



**Background Position Keywords** When you position a background image, you can use measurements (for example, 1em, 20%, and 5px) or you can use keywords (left, right, center, top, and bottom).

Unfortunately, some versions of Internet Explorer and Opera for Windows ignore keywords.

Luckily, these can be replaced with percentage values that have exactly the same effect.

If you want to position a background image at the right of a container, you can use a horizontal value of 100%. This will align the right edge of the image with the right edge of the container.

If you want to position a background image at the bottom of a container, you can use a vertical value of 100%. This will align the bottom edge of the image with the bottom edge of the container.

## Styling the Caption

The next step is to apply some basic styles to the caption text, starting with color.

There are many ways that the color white can be specified, including white, #fff, #ffffff, rgb(255,255,255), and rgb(100%,100%,100%). In this case, the three-digit hexadecimal option will be used—color: #fff.

Text can be aligned to the left, right, center, or justify. This caption will be centered using text-align: center as shown in Listing 11.5. The results can be seen in Figure 11.4.

### LISTING 11.5 CSS Code Styling the Caption

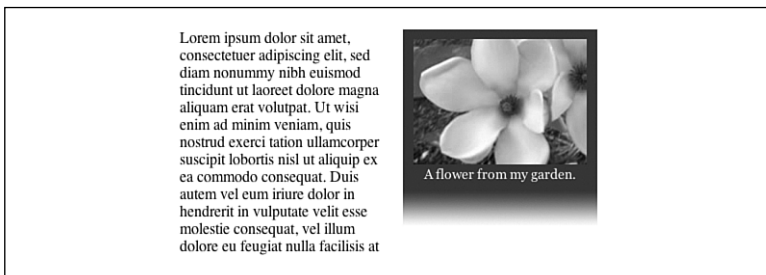
```
div.imagecaption
{
 float: right;
 width: 182px;
```

*continues*

**LISTING 11.5** Continued

---

```
margin: 0 1em 1em 1em;
display: inline;
padding: 10px 10px 70px 10px;
background: #036 url(chapter11.gif) repeat-x 0 100%;
color: #fff;
text-align: center;
}
```



**FIGURE 11.4** Screenshot of styled caption.

## Styling the Image

Finally, the image can be styled with a border.

There are many ways to specify a border around an image. The simplest method is the shorthand border property shown in Listing 11.6 and illustrated in Figure 11.5.

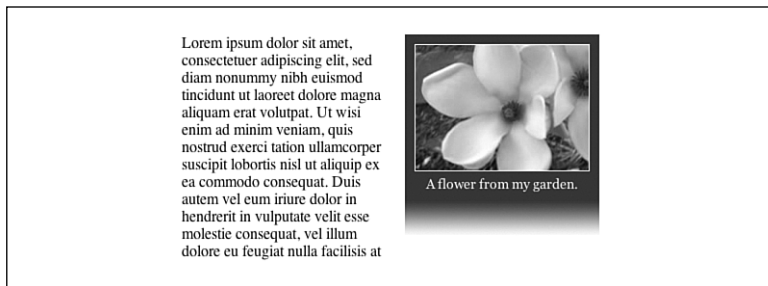
**LISTING 11.6** CSS Code Styling the Image

---

```
div.imagecaption
{
 float: right;
 width: 182px;
 margin: 0 1em 1em 1em;
 display: inline;
 padding: 10px 10px 70px 10px;
 background: #036 url(chapter11.gif) repeat-x 0 100%;
 color: #fff;
 text-align: center;
}
```

*continues*

```
div.imagecaption img
{
 border: 1px solid #fff;
}
```



**FIGURE 11.5** Screenshot of the final result.

## Creating a Side-By-Side Variation

Using the same selectors and HTML code, it is possible to change the layout to display the image and caption side by side.

First, the width of the container will need to be increased to accommodate the new caption and image locations. The declaration can be changed to `width: 302px`. This width can be changed to suit your needs.

Next, padding can be set to `10px` for all sides because you do not need any space for a background image.

The `background-image`, `background-repeat`, and `background-position` properties can also be removed, leaving a simple declaration—`background: #036`.

The image must be floated to the right so that the caption can sit beside it. Width does not need to be defined in this case because the image has its own intrinsic width.

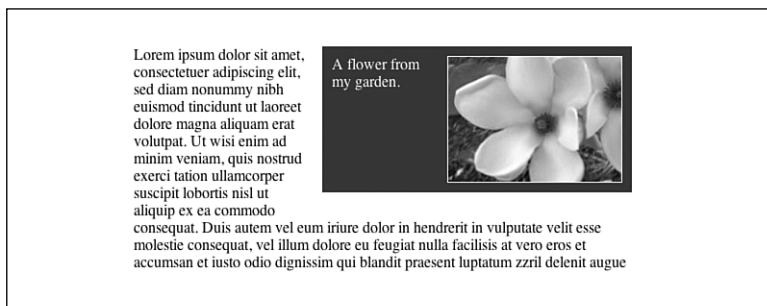
Finally, the image will need to be given some margin so that the caption doesn't butt up against it. You can use `margin-right: 1em` as shown in Listing 11.7 and illustrated in Figure 11.6.

**LISTING 11.7** CSS Code for the Side-By-Side Variation

---

```
div.imagecaption
{
 float: right;
 width: 302px;
 margin: 0 1em 1em 1em;
 display: inline;
 padding: 10px;
 background: #036;
 color: #fff;
}

div.imagecaption img
{
 float: right;
 margin-left: 1em;
 border: 1px solid #fff;
}
```



**FIGURE 11.6** Screenshot of the side-by-side variation.

## Creating a Photo Frame Variation

Another variation is to create a simple photo frame using the container and some borders.

The first step is to change the padding to 15px for top, right, and left edges, and 20px for the bottom. Like the original example, this additional space will be used for the background image.

The borders will be used to create a three-dimensional illusion. The top and left edges will have light-colored thin borders, whereas the right and bottom edges will have darker and thicker borders. This can be achieved using three border declarations as shown in Listing 11.8.

---

**LISTING 11.8** CSS Code for the Photo Frame Variation

---

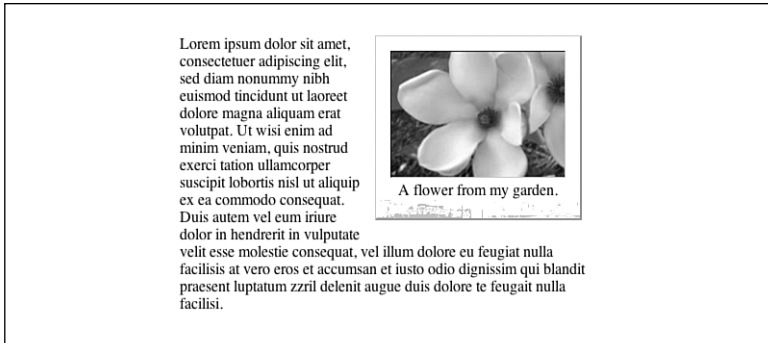
```
div.imagecaption
{
 float: right;
 width: 182px;
 margin: 0 1em 1em 1em;
 padding: 15px 15px 20px 15px;
 display: inline;
 text-align: center;
 border-color: #CCC #999 #999 #CCC;
 border-width: 1px 2px 2px 1px;
 border-style: solid;
 background: url(chapter11c.gif) repeat-x 0 100%;
}

div.imagecaption img
{
 border-color: #000 #ccc #ccc #000;
 border-width: 1px 1px 1px 1px;
 border-style: solid;
}
}
```

A new background image can be used with the same settings as the original version.

Finally, borders need to be added to the image. These borders will be set in the opposite colors to the container, with darker borders on the top and left edges of the image, and lighter borders on the right and bottom edges.

The final results can be seen in Figure 11.7.



**FIGURE 11.7** Screenshot of the photo frame variation.

## Summary

In this lesson, you have learned how to wrap a container around an image and its caption. You then learned how to float the container and style it with width, margin, padding, background image, color, and text-align. You also learned how to apply borders to the image. In the next lesson, you will learn how to create a photo gallery.

# LESSON 12

## Creating a Photo Gallery



*In this lesson, you will learn how to create a photo gallery using a series of floated <div> elements. You will also learn how to use two background images to create a flexible container.*

### Creating a Thumbnail Gallery

First of all, you will need a series of thumbnail images and captions. Each thumbnail image and caption will be placed inside a <div> element. The caption will then be placed inside a <p> element as shown in Listing 12.1.

To make sure you don't target every <div> on the page, you should apply the same classname to each one.

#### **LISTING 12.1** HTML Code Containing the Markup for a Thumbnail Gallery

---

```
<div class="thumbnail">

 <p>A flower from my garden</p>
</div>
<div class="thumbnail">

 <p>White and pinkflower in Spring</p>
</div>
<div class="thumbnail">

 <p>Flower in morning light</p>
</div>
<div class="thumbnail">

 <p>A close-up of flower petals</p>
</div>
```

*continues*



**LISTING 12.1** Continued

---

```
<div class="thumbnail">

 <p>A timeless flower </p>
</div>
```

You will be using three selectors in this lesson. The first selector will target any `<div>` that contains a "thumbnail" class.

The second selector will target any image inside a `<div>` that contains a "thumbnail" class.

The third selector will target any `<p>` element inside a `<div>` that contains a "thumbnail" class. The selectors are shown in Listing 12.2.

**LISTING 12.2** CSS Code Showing the Selectors for Styling the Container

---

```
div.thumbnail {...}
div.thumbnail img {...}
div.thumbnail p {...}
```

## Positioning the `<div>` Elements

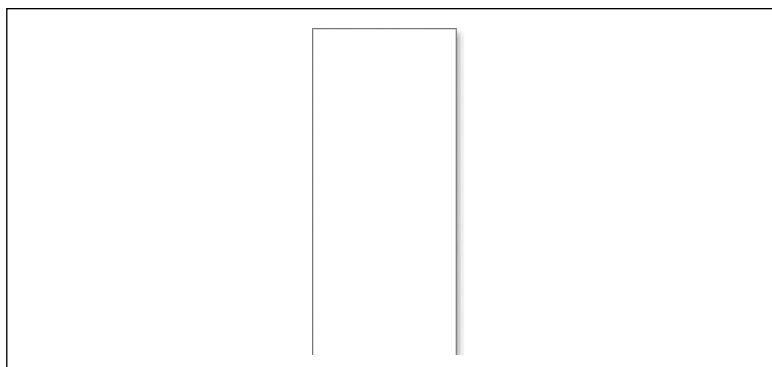
Because the images and captions will sit beside each other in rows, you will need to float the `<div>`. This can be achieved using `float: left`.

When the `<div>` is floated, it must be given a width. In this case, you will use `width: 130px`. This width can be changed to suit your needs.

Next, you might want to create some space around the `<div>` elements so that they don't butt up against each other. You can achieve this by applying margins to the right and bottom of each `<div>`.

Finally, a background image (shown in Figure 12.1) will be added to the `<div>`. As you can see in Figure 12.1, the image is very long so that it is able to grow to accommodate long captions.

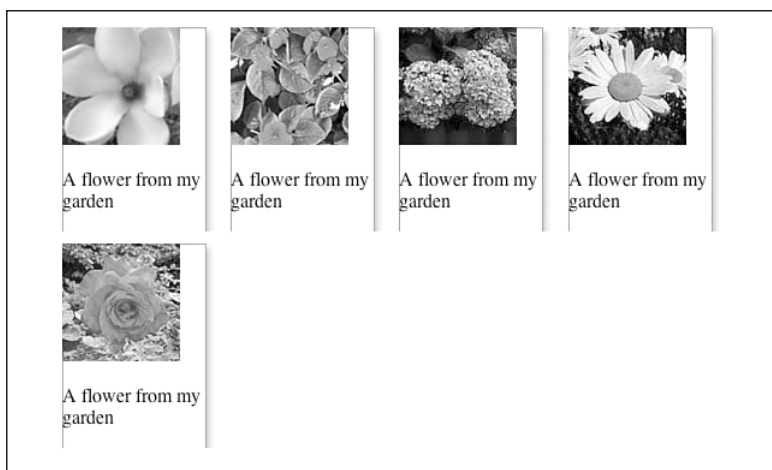
This background image must be set to `no-repeat` because you don't want it to reappear under the caption. The CSS code is shown in Listing 12.3 and illustrated in Figure 12.2.



**FIGURE 12.1** Screenshot of background image used by <div> element.

### LISTING 12.3 CSS Code Floating the Container

```
div.thumbnail
{
 width: 130px;
 float: left;
 margin: 0 10px 10px 0;
 background: url(chapter12a.gif) no-repeat;
}
```



**FIGURE 12.2** Screenshot of positioned containers.



**Liquid and Fixed-Width Layouts** The floated `<div>` elements you have created will sit in a line beside each other, depending on the width of your browser window.

If you decrease the width of your browser, one or more `<div>` elements may drop to a new line below.

This happens because the `<div>` elements have no container apart from the browser window. This type of layout is referred to as a *liquid layout*.

However, you can stop the `<div>` elements from dropping to new lines. If you place them inside a fixed-width container, they will remain in position—no matter how narrow the browser window. This type of layout is referred to as a *fixed-width layout*.

## Styling the Image

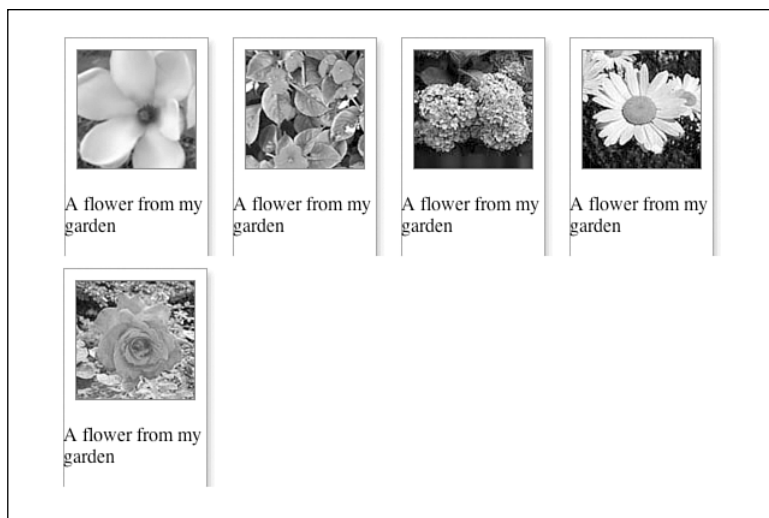
At present, the image sits hard against the edge of its container, the `<div>` element. To give it some space, you can set margins on the top and left using `margin: 10px 0 0 10px`.

You can also add a border to the image using `border: 1px solid #777` as shown in Listing 12.4. The results can be seen in Figure 12.3.

### LISTING 12.4 CSS Code for Styling the Image

```
div.thumbnail
{
 width: 130px;
 float: left;
 margin: 0 10px 10px 0;
 background: url(chapter12a.gif) no-repeat;
}

div.thumbnail img
{
 margin: 10px 0 0 10px;
 border: 1px solid #777;
}
```



**FIGURE 12.3** Screenshot of styled image.

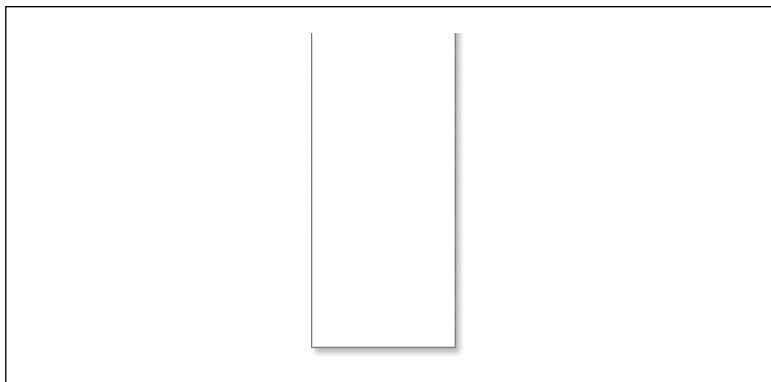
## Styling the Paragraph Element

Most browsers render standard `<p>` elements with margins of 1em above and below. You can override these margins using `margin: 0`.

Now the paragraph will sit under the image, but it is still sitting against the edge of its container. You will need to give it some space using padding. In this case, apply 20px of padding to the right, 30px below, and 10px to the left.

Next, you need to add a background image to finish off the illusion of the drop-shadow box. This image (shown in Figure 12.4) will sit at the bottom of the `<p>` element.

Set the image to `no-repeat` so it doesn't reappear in a long caption. The vertical background position should be set to `100%`, which will make the bottom of the image sit against the bottom of the `<p>` element. The code is shown in Listing 12.5 and illustrated in Figure 12.5.



**FIGURE 12.4** Screenshot of background image used by `<p>` element.

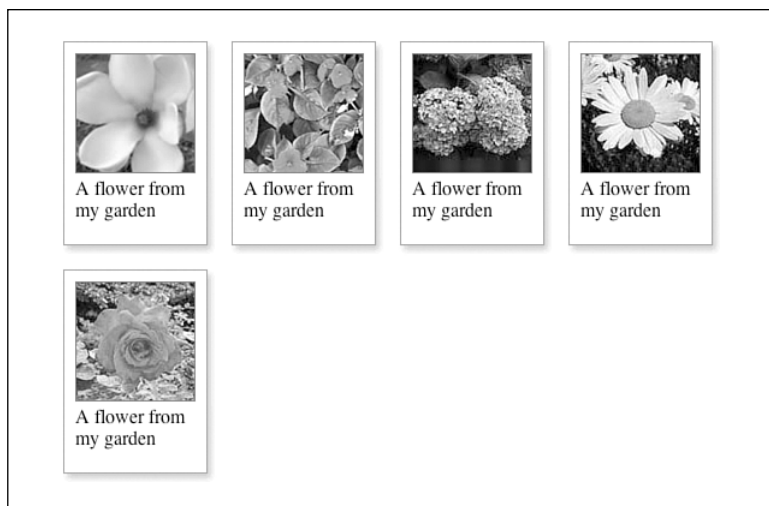
#### **LISTING 12.5** CSS Code for Styling the `<p>` Element

---

```
div.thumbnail
{
 width: 130px;
 float: left;
 margin: 0 10px 10px 0;
 background: url(chapter12a.gif) no-repeat;
}

div.thumbnail img
{
 border: 1px solid #777;
 margin: 10px 0 0 10px;
}

div.thumbnail p
{
 margin: 0;
 padding: 0 20px 30px 10px;
 background: url(chapter12b.gif) no-repeat 0 100%;
}
```



**FIGURE 12.5** Screenshot of final thumbnail gallery.

## Forcing a New Line

There may be situations when you want to set a number of thumbnails on each line. To do this, you need to create a new class and then apply this class to specific `<div>` elements. The new class will have one declaration—`clear: left`. This will move the `<div>` down to a new line, below the bottom edge of any previous left-floating `<div>` elements. The CSS code is shown in Listing 12.6.

---

### LISTING 12.6 CSS Code Forcing a New Line

```
div.thumbnail
{
 width: 130px;
 float: left;
 margin: 0 10px 10px 0;
 background: url(chapter12a.gif) no-repeat;
}

div.thumbnail img
{
```

*continues*

**LISTING 12.6** Continued

---

```
 border: 1px solid #777;
 margin: 10px 0 0 10px;
 }

 div.thumbnail p
 {
 margin: 0;
 padding: 0 20px 30px 10px;
 background: url(chapter12b.gif) no-repeat 0 100%;
 }

 .clear
 {
 clear: left;
 }
```

This new class will need to be added to any <div> that must start on a new line. The simplest way to add this new class is to include it in the existing class attribute. So, `class="thumbnail"` can be changed to `class="thumbnail clear"` as shown in Listing 12.7. The final results can be seen in Figure 12.6.

**LISTING 12.7** HTML Code Showing New Class

---

```
<div class="thumbnail">

 <p>A flower from my garden</p>
</div>
<div class="thumbnail">

 <p>A flower from my garden</p>
</div>
<div class="thumbnail">

 <p>A flower from my garden</p>
</div>
<div class="thumbnail clear">

 <p>A flower from my garden</p>
</div>
<div class="thumbnail">
```

*continues*

```

<p>A flower from my garden</p>
</div>
```



**FIGURE 12.6** Screenshot of thumbnail gallery.

## Creating a Side-By-Side Variation

Using the same selectors and HTML code, it is possible to change the layout so that the images and their captions are displayed side by side.

First, the width of the `<div>` will need to be increased to accommodate the caption beside the image. The declaration can be changed to `width: 250px`.

Next, some `padding-bottom` and a `border` can be added to the `<div>`.

The `background` declaration can be removed completely.

The image must be floated to the left so that the caption can sit beside it. Width does not need to be defined in this case because the image has its own intrinsic width.

Finally, the image will need to be given some `margin` so that the caption doesn't butt up against it. You can use `10px` for all sides except the bottom as shown in Listing 12.8. The results can be seen in Figure 12.7.



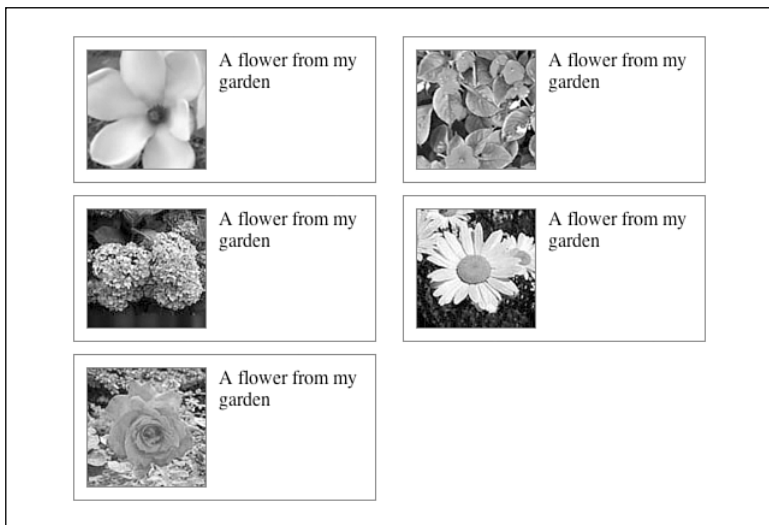
**LISTING 12.8** CSS Code for the Side-By-Side Variation

---

```
div.thumbnail
{
 float: left;
 width: 250px;
 margin: 0 10px 10px 0;
 padding-bottom: 10px;
 border: 1px solid #777;
}

div.thumbnail img
{
 float: left;
 border: 1px solid #777;
 margin: 10px 10px 0 10px;
}

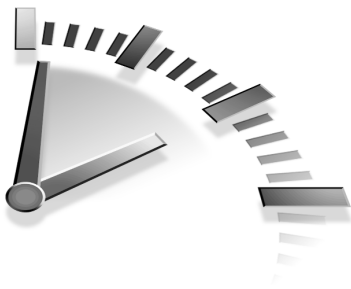
div.thumbnail p
{
 margin: 0;
 padding: 10px;
}
```

**FIGURE 12.7** Screenshot of the side-by-side variation.

## Summary

In this lesson, you have learned to float a series of `<div>` elements, and then style them. You also learned how to force a line break using `clear`.

In the next lesson, you will learn how to style a blockquote.



## LESSON 13

# Styling a Block Quote

*In this lesson, you will learn how to style a block quote using two background images. The aim is to create a quotation that sits inside two large graphic quotation marks.*

## Applying the <blockquote> Element

The <blockquote> element is often used to indent text. However, it should not be used for this purpose. It should only be used to mark up long quotations that consist of block level content.

For this lesson, you will need a quotation and a source or author for the quotation. These two items will be marked up as paragraphs and then placed inside a <blockquote>. The paragraph that contains the source information will be given a source class. A class is used in this case because you might want to have more than one <blockquote> on a page.

Additional information about the source, such as a web address, can be added to the <blockquote> using the cite attribute as shown in Listing 13.1.

### LISTING 13.1 HTML Code Containing the Markup for a Quotation

---

```
<blockquote cite="http://www.sitename.com/quote/smith.htm">
 <p>
 Lorem ipsum dolor ...
 </p>
 <p class="source">
 John Smith
 </p>
</blockquote>
```



### **Semantically Correct Markup and Block Quotes**

Semantically correct markup is about understanding HTML elements and what they mean. It is also about using these elements to give meaning to the content they contain.

If you use semantically correct markup, your content will have meaning in a wide range of devices, including text browsers, screen readers, and hand-held devices.

If you use poor semantic markup, however, your content will either have no meaning or incorrect meaning in a wide range of devices.

An example of poor semantic markup is using a `<blockquote>` to indent text. The `<blockquote>` will change the content presentation rather than add meaning. Even worse, the indented content is given incorrect meaning.

It would be far better to indent content using CSS and save the `<blockquote>` for its intended purpose—long quotations.

To style the `<blockquote>` and its content, you will use three selectors:

- The first is a type selector, used to target any instance of the `<blockquote>` on the page.
- The second is a descendant selector, used to target any `<p>` element inside a `<blockquote>`.
- The third is another descendant selector, used to target any `<p>` element that has been styled with the source class.

The selectors are shown in Listing 13.2.

**LISTING 13.2** CSS Code Showing the Selectors for Styling the `<blockquote>`

```
blockquote {...}
blockquote p {...}
blockquote p.source {...}
```

## Styling the `<blockquote>` Element

Unstyled `<blockquote>` elements are indented on the left and right sides. You can change this default behavior by resetting the margins. In this case, you will set the top and bottom margins to 1em and the left and right margins to 0. This can be achieved using a shorthand margin declaration—`margin: 1em 0`.

Next, you can apply a border to the `<blockquote>` to separate it from other content on the page. You can use a 1-pixel-wide border set to light gray, `border: 1px solid #ddd`.

The first background image will be applied directly to the `<blockquote>` element. The declaration will be `background-image: url(lesson13.gif)`.

The image will be positioned in the top left corner of the `<blockquote>`. This is achieved by setting the x and y axis to 5px using a declaration of `background-position: 5px 5px`.

To stop the image from repeating across the entire `<blockquote>`, add `background-repeat: no-repeat`.

These three background declarations can be shortened into a single declaration using `background: url(lesson13.gif) 5px 5px no-repeat`. The background image is shown in Figure 13.1.



**FIGURE 13.1** Background image applied to `<blockquote>`.

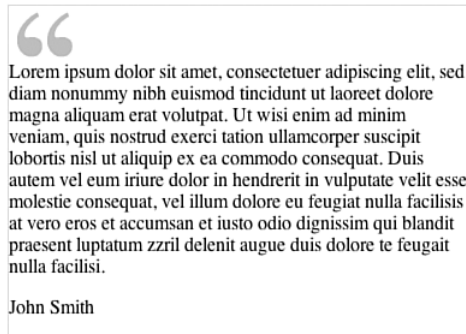
Now the image is in position, but the text is sitting over the top of it. This can be fixed by applying some padding to the top of the `<blockquote>`

using `padding-top: 30px` as shown in Listing 13.3. The results can be seen in Figure 13.2.

---

**LISTING 13.3** CSS Code Styling the `<blockquote>`

```
blockquote
{
 margin: 1em 0;
 border: 1px solid #ddd;
 background: url(lesson13.gif) 5px 5px no-repeat;
 padding-top: 30px;
}
```



**FIGURE 13.2** Screenshot of styled `<blockquote>`.

## Styling the Paragraph

Next, you will need to add padding to the left and right of any paragraphs inside the `<blockquote>`. This will push the content away from the borders. You can use `padding: 0 70px` as shown in Listing 13.4. The results can be seen in Figure 13.3.

---

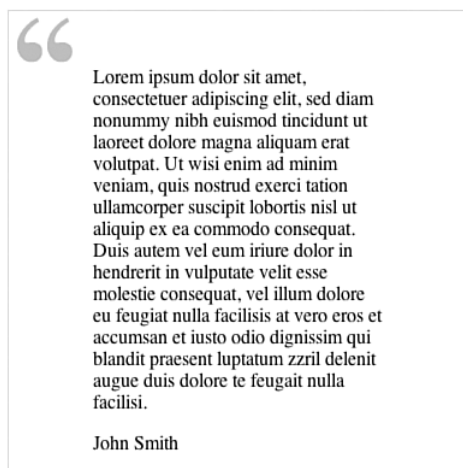
**LISTING 13.4** CSS Code Styling the Paragraph

```
blockquote
{
 margin: 1em 0;
 border: 1px solid #ddd;
 padding: 0 70px;
}
```

**LISTING 13.4** Continued

```
background: url(lesson13.gif) 5px 5px no-repeat;
padding-top: 30px;
}

blockquote p
{
 padding: 0 70px;
}
```

**FIGURE 13.3** Screenshot of styled paragraph.

## Styling the **source** Class

Now that the `<blockquote>` and paragraph elements are styled, you can focus on the paragraph classed with `source`. This paragraph will be used to place the second background image in the bottom-right corner of the block quote.

A shorthand declaration can be used to set the image, repeat, and position—`background: url(lesson13a.gif) no-repeat 100% 100%`. The x and y axis must be set to `100%` to place the bottom-right edge of the

image in the bottom-right corner of the paragraph. `no-repeat` should be used to stop the image from repeating under the text. The background image is shown in Figure 13.4.



**FIGURE 13.4** Background image applied to paragraph classed with `source`.

You will need to apply `padding-bottom: 30px` to the paragraph to push the text up 30 pixels and stop it from sitting on top of the background image.

The top image sits 5px in from the top and left edges of the `<blockquote>`. To achieve the same result on the bottom, apply a 5px margin to the right and bottom of the source class paragraph using `margin: 0 5px 5px 0`.

Finally, the source text can be differentiated from other block quote text by aligning it to the right and making it italic. This can be achieved with `text-align: right` and `font-style: italic` as shown in Listing 13.5. The results can be seen in Figure 13.5.

---

**LISTING 13.5** CSS Code Styling the source Class Paragraph

```
blockquote
{
 margin: 1em 0;
 border: 1px solid #ddd;
 background: url(lesson13.gif) 5px 5px no-repeat;
 padding-top: 30px;
}

 blockquote p
{
 padding: 0 70px;
}

blockquote p.source
{

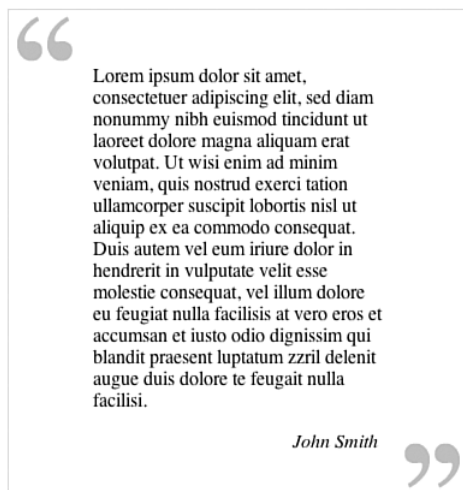
```

*continues*



**LISTING 13.5** Continued

```
background: url(lesson13a.gif) no-repeat 100% 100%;
padding-bottom: 30px;
margin: 0 5px 5px 0;
text-align: right;
font-style: italic;
}
```

**FIGURE 13.5** Screenshot of styled paragraph classed with source.

## Creating a Variation

Using the same selectors and HTML code, it is possible to change the layout so that the `<blockquote>` and its content looks entirely different.

For this example, you will create a fixed-width `<blockquote>` with one large background image as shown in Figure 13.6.

**FIGURE 13.6** Screenshot of new background image.

In the `blockquote` selector, the background image and its position need to change. In this case, both the x and y axes will be set to 0, which means the image will sit in the top-left corner of the `<blockquote>`.

The `padding-top` declaration changes from 30px to 1px to trap paragraph margins.



**Trapping Margins** A standard paragraph has predefined top and bottom margins.

When a paragraph is placed inside of another container, its top margin can cause problems. Some browsers will display the paragraph and top margin inside of the container. Other browsers, however, will display the paragraph only, and allow the margin to poke out the top of the container.

You can stop this from occurring by applying either `border-top` or `padding-top` to the container. The amount can be as tiny as 1px, as long as it is present.

This is referred to as trapping margins.

The `blockquote.p` selector changes from `padding: 0 70px` to `padding: 0 1em 0 80px`. This new padding will move the text away from the left edge of the `<blockquote>` to allow room for the new background image.

The `blockquote.p.source` selector has a range of changes. The background image is removed completely. Margins are set to 0. A 5px white border is added to the top of the paragraph to separate the source from the quotation. Padding is changed from `padding-bottom: 30px` to `padding: .5em .5em .5em 80px`.

Finally, `text-align: right` is removed and `background: #336` is added. The results are shown in Listing 13.6. The results can be seen in Figure 13.7.

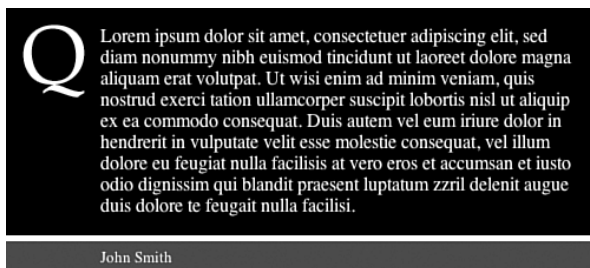
**LISTING 13.6** CSS Code for the <blockquote> Variation

---

```
blockquote
{
 margin: 1em 0;
 border: 1px solid #000;
 background: #000 url(lesson13b.gif) no-repeat 0 0;
 padding-top: 1px;
 color: #fff;
 width: 500px;
}

blockquote p
{
 padding: 0 1em 0 80px;
}

blockquote p.source
{
 margin: 0;
 border-top: 5px solid #fff;
 padding: .5em .5em .5em 80px;
 background: #336;
 font-style: italic;
}
```

**FIGURE 13.7** Screenshot of the <blockquote> variation.

## Summary

In this lesson, you learned how to wrap the <blockquote> element around a long quotation and source, and then style it in different ways. In the next lesson, you will learn how to mark up and then style accessible data tables.

# LESSON 14

## Styling a Data Table



*This lesson is divided into two sections. First, you will learn how to add accessibility features to a data table to make the content more accessible for screen readers. Second, you will learn how to style various elements within the table.*

### Starting with a Basic Table

As CSS increases in popularity, there is a growing trend to move away from using tables to mark up content. However, there are times when tables are the best markup option, especially for tabular data. A basic data table is shown in Listing 14.1.

#### **LISTING 14.1** HTML Code Containing the Markup for a Data Table

---

```
<table>
 <tr>
 <td>Item</td>
 <td>Threaded screws </td>
 <td>Flat nails</td>
 <td>Dyna-bolts </td>
 <td>Spring washers</td>
 </tr>
 <tr>
 <td>1 kg</td>
 <td>$2.50</td>
 <td>$3.50</td>
 <td>$4.50</td>
 <td>$2.50</td>
 </tr>
</table>
```

*continues*

**LISTING 14.1** Continued

---

```

 <td>2kg</td>
 <td>$3.00</td>
 <td>$4.00</td>
 <td>$5.00</td>
 <td>$3.00</td>
 </tr>
 <tr>
 <td>3kg</td>
 <td>$3.50</td>
 <td>$4.50</td>
 <td>$5.50</td>
 <td>$3.50</td>
 </tr>
 <tr>
 <td>4kg</td>
 <td>$4.00</td>
 <td>$5.00</td>
 <td>$6.00</td>
 <td>$4.00</td>
 </tr>
</table>

```

## Adding Accessibility Features to a Data Table

There are a range of features that can be added to data tables to make them more accessible.

The summary attribute shown in Listing 14.2 should be used on complex data tables because it provides a clear description of what the table presents. It does not display on screens of current (standards-compliant) web browsers, but it can display on other web-browsing devices such as handhelds, cell phones, and so forth. The summary attribute is used as an orientation for people who use nonvisual devices.

**LISTING 14.2** HTML Code Showing summary

---

```

<table summary="Table of screws, Flat nails, Dyna-bolts and
Spring washers, in kilos">

```

A caption should be included with any data table. It provides a brief description of the table's contents. Unlike the summary, the caption is displayed on the screen—usually centered above the table. The caption should appear directly after the opening table tag as shown in Listing 14.3.

---

**LISTING 14.3** HTML Code Showing caption

```
<caption>
 Pricing for threaded screws, flat nails, dyna-bolts and
 spring washers
</caption>
```

The `<th>` element, shown in Listing 14.4, should be used to define any row or column heading within a data table. It is used to create a relationship between `<th>` and `<td>` elements, which is important for nonvisual devices.

---

**LISTING 14.4** HTML Code Showing `<th>` Elements

```
<th>Item</th>
<th>Threaded screws </th>
<th>Flat nails</th>
<th>Dyna-bolts </th>
<th>Spring washers</th>
```

The `<thead>`, `<tbody>`, and `<tfoot>` elements shown in Listing 14.5 are used to group rows in tables. The `<thead>` and `<tfoot>` should contain information about the table's columns and the `<tbody>` should contain the table data.

---

**LISTING 14.5** HTML Showing `<thead>` and `<tbody>` Elements

```
<thead>
 <tr>
 <th>Item</th>
 <th>Threaded screws </th>
 <th>Flat nails</th>
 <th>Dyna-bolts </th>
 <th>Spring washers</th>
 </tr>
</thead>
<tbody>
```

*continues*

**LISTING 14.5**    Continued

---

```

 <tr>
 <th>1 kg</th>
 <td>$2.50</td>
 <td>$3.50</td>
 <td>$4.50</td>
 <td>$2.50</td>
 </tr>
</tbody>

```

The `abbr` attribute, shown in Listing 14.6, is used to provide an abbreviated form of the relevant cell's contents. The `abbr` attribute is important for people who use screen readers and may have to hear a cell's content read out loud repeatedly.

**LISTING 14.6**    HTML Code Showing `abbr` Attributes

---

```

<tr>
 <th>Item</th>
 <th abbr="screws">Threaded screws</th>
 <th abbr="nails">Flat nails</th>
 <th abbr="bolts">Dyna-bolts</th>
 <th abbr="washers">Spring washers</th>
</tr>

```

`headers` and `ids` are used to tie a table's data cells with their appropriate header. Each header must be given a unique `id`. The `headers` attribute is then added to each `<td>` element as shown in Listing 14.7.

**LISTING 14.7**    HTML Code Showing `headers` and `ids`

---

```

<table summary="Table of screws, Flat nails, Dyna-bolts and
Spring washers, in kilos">
 <caption>
 Pricing for threaded screws, flat nails, dyna-bolts
and spring washers
 </caption>
 <thead>
 <tr>
 <th>Item</th>
 <th id="screws" abbr="screws">Threaded
screws</th>

```

*continues*

```

 <th id="nails" abbr="nails">Flat nails</th>
 <th id="bolts" abbr="bolts">Dyna-bolts</th>
 <th id="washers" abbr="washers">Spring
washers</th>
 </tr>
</thead>
<tbody>
 <tr>
 <th id="one">1 kg</th>
 <td headers="screws one">$2.50</td>
 <td headers="nails one">$3.50</td>
 <td headers="bolts one">$4.50</td>
 <td headers="washers one">$2.50</td>
 </tr>
 <tr>
 <th id="two">2kg</th>
 <td headers="screws two">$3.00</td>
 <td headers="nails two">$4.00</td>
 <td headers="bolts two">$5.00</td>
 <td headers="washers two">$3.00</td>
 </tr>
 <tr>
 <th id="three">3kg</th>
 <td headers="screws three">$3.50</td>
 <td headers="nails three">$4.50</td>
 <td headers="bolts three">$5.50</td>
 <td headers="washers three">$3.50</td>
 </tr>
 <tr>
 <th id="four">4kg</th>
 <td headers="screws four">$4.00</td>
 <td headers="nails four">$5.00</td>
 <td headers="bolts four">$6.00</td>
 <td headers="washers four">$4.00</td>
 </tr>
</tbody>
</table>

```

## Creating Selectors to Style a Table

To style this table and its content, you will use eight selectors as shown in Listing 14.8.



**LISTING 14.8**    CSS Code Showing the Selectors for Styling the Table

---

```
caption {...}
table {...}
th, td {...}
tr {...}
thead th {...}
tbody th {...}
tr.alternate {...}
tr.alternate th ... }
```

## Styling the Caption

An unstyled table caption will be displayed above the table. On most modern browsers the caption will be center aligned, but you can change the default caption alignment using `text-align: left`.

To increase the space between the caption and its table, `margin-bottom` can be set to `.5em`.

The caption also can be given more weight to make it stand out from the table content. This is achieved using `font-weight:bold` as shown in Listing 14.9. The results can be seen in Figure 14.1.

**LISTING 14.9**    CSS Code for Styling the Caption

---

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}
```

Pricing for threaded screws, flat nails, dyna-bolts and spring washers				
Item	Threaded screws	Flat nails	Dyna-bolts	Spring washers
1 kg	\$2.50	\$3.50	\$4.50	\$2.50
2kg	\$3.00	\$4.00	\$5.00	\$3.00
3kg	\$3.50	\$4.50	\$5.50	\$3.50
4kg	\$4.00	\$5.00	\$6.00	\$4.00

Item	Threaded screws	Flat nails	Dyna-bolts	Spring washers
1 kg	\$2.50	\$3.50	\$4.50	\$2.50
2kg	\$3.00	\$4.00	\$5.00	\$3.00
3kg	\$3.50	\$4.50	\$5.50	\$3.50
4kg	\$4.00	\$5.00	\$6.00	\$4.00

**FIGURE 14.1**    Screenshot of styled caption.

## Styling the <table> Element

Apply `border-collapse: collapse` to the <table> element to remove cellspacing as shown in Listing 14.10.

### LISTING 14.10 CSS Code for Styling the <table> Element

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}

table
{
 border-collapse: collapse;
}
```



**Tables and cellspacing** A standard table will have about 2px of cellspacing between cells. This can be removed using two methods.

The first method is to apply `cellspacing="0"` as an attribute inside the <table> element. This is not ideal because a presentation attribute has been added to the table. If you were to change the presentation at a later date, you would need to adjust the HTML as well as the CSS.

The second method is to apply `border-collapse: collapse` to the <table> element using CSS. This method is preferred because the appearance of the table can be changed at any time without affecting the HTML.

## Styling the <th> and <td> Elements

Now the <th> and <td> elements need to be styled with a right and bottom border. Because the border will appear on all <td> and <th> elements, you can group both elements into one selector.

Applying a border is more powerful than using cellspacing because you can change the color or width of these borders at any time to suit your needs.

To apply padding to all cells, use `padding: .5em` as shown in Listing 14.11.

---

**LISTING 14.11**    CSS Code for Styling the `<th>` and `<td>` Elements

---

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}

table
{
 border-collapse: collapse;
}

th, td
{
 border-right: 1px solid #fff;
 border-bottom: 1px solid #fff;
 padding: .5em;
}
```

## Styling the `<tr>` Element

The `<tr>` element should be styled with a `background-color` as shown in Listing 14.12. This color can be changed to suit your needs. The results can be seen in Figure 14.2.

---

**LISTING 14.12**    CSS Code for Styling the `<tr>` Element

---

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}
```

```
table
{
 border-collapse: collapse;
}

th, td
{
 border-right: 1px solid #fff;
 border-bottom: 1px solid #fff;
 padding: .5em;
}

tr
{
 background: #B0C4D7;
}
```

Pricing for threaded screws, flat nails, dyna-bolts and spring washers				
Item	Threaded screws	Flat nails	Dyna-bolts	Spring washers
1 kg	\$2.50	\$3.50	\$4.50	\$2.50
2kg	\$3.00	\$4.00	\$5.00	\$3.00
3kg	\$3.50	\$4.50	\$5.50	\$3.50
4kg	\$4.00	\$5.00	\$6.00	\$4.00

**FIGURE 14.2** Screenshot of styled `<tr>` element.

## Targeting Instances of the `<th>` Element

The next step is to create background colors for the `<th>` element. Using descendant selectors, it is possible to apply different colors to the `<th>` elements on the top and left side of the table.

The `<th>` elements across the top of the table are styled with `thead th { ... }` because they appear inside the `<thead>` element.

The `<th>` elements down the side of the table are styled with `tbody th { ... }` because they appear inside the `<tbody>` element.

The <th> elements down the side also can be set to `font-weight: normal` to differentiate them from the headers across the top as shown in Listing 14.13. The results can be seen in Figure 14.3.

---

**LISTING 14.13**    CSS Code for Styling the <th> Elements

---

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}

table
{
 border-collapse: collapse;
}

th, td
{
 border-right: 1px solid #fff;
 border-bottom: 1px solid #fff;
 padding: .5em;
}

tr
{
 background: #B0C4D7;
}

thead th
{
 background: #036;
 color: #fff;
}

tbody th
{
 font-weight: normal;
 background: #658CB1;
}
```

Item	Threaded screws	Flat nails	Dyna-bolts	Spring washers
1 kg	\$2.50	\$3.50	\$4.50	\$2.50
2kg	\$3.00	\$4.00	\$5.00	\$3.00
3kg	\$3.50	\$4.50	\$5.50	\$3.50
4kg	\$4.00	\$5.00	\$6.00	\$4.00

**Figure 14.3** Screenshot of styled `<th>` elements.

## Creating Alternate Row Colors

It is possible to style alternate table rows so that they have different background colors. This aids readability, especially on a long table.

One method is to add a class to every second `<tr>` element. In this case, the class is `alternate`. The `<td>` and `<th>` elements within these rows can be given a slightly different background color. The selectors will need to be `tr.alternate td {...}` and `tr.alternate th {...}` as shown in Listing 14.14. The results can be seen in Figure 14.4.

### **LISTING 14.14** CSS Code for Styling Alternate Rows

```
caption
{
 text-align: left;
 margin: 0 0 .5em 0;
 font-weight: bold;
}

table
{
 border-collapse: collapse;
}

th, td
{
 border-right: 1px solid #fff;
 border-bottom: 1px solid #fff;
```

**LISTING 14.14** Continued

```

padding: .5em;
}

tr
{
 background: #B0C4D7;
}

thead th
{
 background: #036;
 color: #fff;
}

tbody th
{
 font-weight: normal;
 background: #658CB1;
}

tr.alternate
{
 background: #D7E0EA;
}

tr.alternate th
{
 background: #8AA9C7;
}

```

**Pricing for threaded screws, flat nails, dyna-bolts and spring washers**

Item	Threaded screws	Flat nails	Dyna-bolts	Spring washers
1 kg	\$2.50	\$3.50	\$4.50	\$2.50
2kg	\$3.00	\$4.00	\$5.00	\$3.00
3kg	\$3.50	\$4.50	\$5.50	\$3.50
4kg	\$4.00	\$5.00	\$6.00	\$4.00

**FIGURE 14.4** Screenshot of styled alternate rows.

## Summary

In this lesson, you learned how to add accessibility features to a data table, including `summary`, `caption`, `<thead>`, `<th>`, `headers`, and `ids`. You also learned how to style the table and its elements. These accessibility features not only enhance the website via other devices, but help create CSS suitable for these devices. In the next lesson, you will learn how to create vertical navigation.





# LESSON 15

## Creating Vertical Navigation

*In this lesson, you will learn how to create vertical navigation. You will also learn how to apply background images and hover effects.*

### Why Use a List?

At its most basic level, site navigation is simply a list of links to other pages in the site. So, a standard HTML list is the ideal starting point (see Listing 15.1). The resulting list is shown in Figure 15.1.

#### **LISTING 15.1** HTML Code Containing the Markup for a List

```
<ul id="navigation">
 Home
 About
 Services
 Staff
 Portfolio
 Contact
 Sitemap

```

- [Home](#)
- [About](#)
- [Services](#)
- [Staff](#)
- [Portfolio](#)
- [Contact](#)
- [Sitemap](#)

**FIGURE 15.1** Screenshot of unstyled list.

## Styling the List

To style this list, you will need to use selectors that target the `<ul>`, `<li>`, and `<a>` elements. To make sure you do not target every instance of these elements on the page, you will need to include the unique identifier, `navigation`, within each selector. The four selectors that you will use are shown in Listing 15.2.

### LISTING 15.2 CSS Code Showing the Selectors for Styling the List

---

```
ul#navigation {...}
ul#navigation a {...}
ul#navigation a:hover {...}
ul#navigation li {...}
```



**What Are Selectors?** Selectors are used to “select” elements on an HTML page so that they can be styled.

For more information, see Lesson 3, “Selectors in Action.”

## Styling the `<ul>` Element

Most browsers display HTML lists with left indentation. To set this indentation, some browsers use padding (Firefox, Netscape, and Safari), and others use margins (Internet Explorer and Opera).

To remove this left indentation consistently across all browsers, set both `padding-left` and `margin-left` to `0` on the `<ul>` element as shown in Listing 15.3.

### LISTING 15.3 CSS Code for Zeroing Margins and Padding

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
}
```

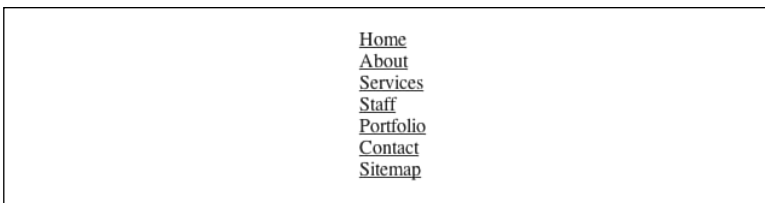
To remove the list bullets, set the `list-style-type` to `none` as in Listing 15.4. The results of the CSS style rules are shown in Figure 15.2.

---

**LISTING 15.4**    CSS Code for Removing List Bullets

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}
```



**FIGURE 15.2**    Screenshot of list with the `<ul>` element styled.

## Styling the `<a>` Element

Text links are generally only active when you mouse over the actual text area. You can increase this active area by applying `display: block;` to the `<a>` element. This will change it from inline to block level, and the active area will extend to the full width of the list item.

When the `<a>` element is block level, users do not have to click on the text; they can click on any area of the list item.

Style the `<a>` elements with `display: block;` as shown in Listing 15.5.

---

**LISTING 15.5**    CSS Code for Setting `display: block`

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}
```

*continues*

```
ul#navigation a
{
 display: block;
}
```

To remove the underlines on the links, use `text-decoration: none;` (see Listing 15.6).

### LISTING 15.6 CSS Code for Removing Link Underlining

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
}
```



**Changing Link Behavior** Changing standard hyper-link behavior (such as removing underlines) can be confusing for some users who might not realize that the item is a link.

For this reason, it is generally not a good idea to remove underlines on links unless you provide some other means to allow users to distinguish links.

To set the background color, you can use the shorthand rule `background: #036;` as shown in Listing 15.7. This color can be changed to suit your needs.

**LISTING 15.7**    CSS Code for Setting Background Color

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
}
```

Next, the text color should be set to #fff (the hex color for white). See Listing 15.8. Like the background color, text color can be changed to suit your needs.

**LISTING 15.8**    CSS Code for Setting Text Color

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
 color: #fff;
}
```

You will need .2em padding on the top and bottom of the <a> element, and .5em padding on both sides. Rather than specify these amounts in separate declarations, you can use one shorthand declaration to define them all. In this case you will use `padding: .2em .5em`, which will apply .2em of padding on the top and bottom of the <a> element, and .5em on both sides as shown in Listing 15.9.

**LISTING 15.9** CSS Code for Setting Padding

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
 color: #fff;
 padding: .2em .5em;
}
```



**Why Use Ems?** You can use either pixels or ems to specify measurement units for padding, margins, and widths. Ems are more flexible because they scale up or down to match the user's font size settings.

To provide some space between the list items, you can add a border on the bottom of each list item. In this case you will use `border-bottom: #fff` as shown in Listing 15.10.

**LISTING 15.10** CSS Code for Setting Borders

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
```

*continues*

**LISTING 15.10** Continued

---

```
color: #fff;
padding: .2em .5em;
border-bottom: 1px solid #fff;
}
```



**Customizing the Border Bottom** In this lesson, the border-bottom is set to #fff, assuming that the page background is white. However, the color of the border-bottom should be set in the same color as the page or container background.

If more space is required between list items, the width of the border can be increased.

Set the width of the <a> element using width: 7em; as shown in Listing 15.11 and illustrated in Figure 15.3. This width can be changed to suit your needs.

**LISTING 15.11** CSS Code for Setting Width

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
 color: #fff;
 padding: .2em .5em;
 border-bottom: 1px solid #fff;
 width: 7em;
}
```



**List Width** As discussed in Lesson 5, “Getting to Know the CSS Box Model,” padding is added to the content area to give a final width. In this case, .5em of padding is applied to the left and right sides of the list. When added to the content width of 7em, the list items are now 8em wide.



**FIGURE 15.3** Screenshot of list with `<a>` element styled.



**Fixing Odd Borders** Certain browsers, such as Netscape, Mozilla, and Firefox, will render the border-bottom incorrectly for some list items—generally in the middle of a list. This can be fixed by changing the border thickness to 1em instead of 1px.

## Adding a Hover Effect

The `:hover` pseudo-class can be used to change the style of links when they are rolled over. In this case, you will set the background to #69C and the color to #000 as shown in Listing 15.12. The results can be seen in Figure 15.4. These colors can be changed to suit your needs.

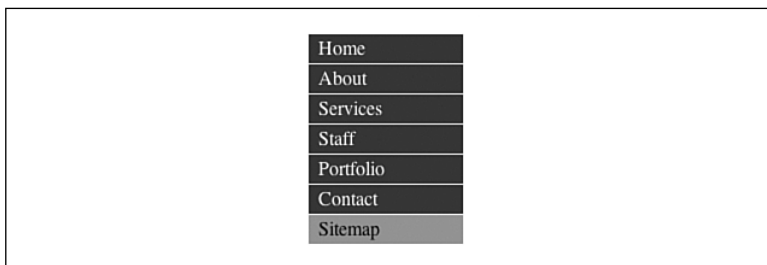


**LISTING 15.12** CSS Code for Setting Hover

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
 color: #fff;
 padding: .2em .5em;
 border-bottom: 1px solid #fff;
 width: 7em;
}

ul#navigation a:hover
{
 background: #69C;
 color: #000;
}
```

**FIGURE 15.4** Screenshot of list showing hover.

## Styling the `<li>` Element

You might notice that there are slight gaps between list items in some versions of Internet Explorer for Windows or Opera. This can be overcome by setting the `<li>` element to `display: inline` (see Listing 15.13).

**LISTING 15.13** CSS Code for Setting display: Inline on the <li> Element

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}

ul#navigation a
{
 display: block;
 text-decoration: none;
 background: #036;
 color: #fff;
 padding: .2em .5em;
 border-bottom: 1px solid #fff;
 width: 7em;
}

ul#navigation a:hover
{
 background: #69C;
 color: #000;
}

ul#navigation li
{
 display: inline;
}
```

## Summary

In this lesson, you have learned that lists can be styled to look like a vertical navigation panel, how to set declarations on the <a> element, and how to use the :hover pseudo-class. In the next lesson, you will learn how to create horizontal navigation.



## LESSON 16

# Creating Horizontal Navigation

*In this lesson, you will learn how to create horizontal navigation from a standard HTML list. You will also learn how to float the `<ul>` element to create a navigation bar and float the `<a>` element to create a series of square buttons—each with a thin dividing line down its right edge.*

## Styling the List

To style this list, you will need to use selectors that target the `<ul>`, `<li>`, and `<a>` elements. You will also need to include the unique identifier, `navigation`, within each selector. The four selectors that you will use are shown in Listing 16.1. The HTML code is shown in Listing 16.2.

### LISTING 16.1 CSS Code Showing the Selectors for Styling the List

---

```
ul#navigation {...}
ul#navigation li {...}
ul#navigation a {...}
ul#navigation a:hover {...}
```

### LISTING 16.2 HTML Code Containing the Markup for a List

---

```
<ul id="navigation">
 Home
 About
 Services
 Staff
 Portfolio
 Contact

```

## Styling the <ul> Element

As discussed in Lesson 15, “Creating Vertical Navigation,” most browsers display HTML lists with left indentation. To remove this left indentation, set both `padding-left` and `margin-left` to `0` on the `<ul>` element as shown in Listing 16.3.

### LISTING 16.3 CSS Code Zeroing Margins and Padding

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
}
```

To remove the list bullets, set the `list-style-type` to `none` as in Listing 16.4.

### LISTING 16.4 CSS Code Removing List Bullets

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
}
```

Next, add a background color using the shorthand `background: #036`; as shown in Listing 16.5. This color can be changed to suit your needs.

### LISTING 16.5 CSS Code Setting Background Color

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
}
```

To float the `<ul>`, use `float: left`. You will also need to set a width. In this case, we will use `100%` because we want the list to spread across the full width of the page. The results are shown in Listing 16.6 and illustrated in Figure 16.1. At this stage, the text is almost illegible. This will be addressed when the `<a>` element is styled.



**Why Float the `<ul>` and `<a>` Elements?** In this lesson, both the `<ul>` and `<a>` elements need to be floated.

The `<a>` element is floated so that the list items sit in a horizontal line, butting up against each other.

The `<ul>` must be floated so that it wraps around the `<a>` elements. Otherwise, it will have no height and will not be visible.

### LISTING 16.6 CSS Code Setting Float and Width

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}
```



**FIGURE 16.1** Screenshot of list with the `<ul>` element styled.

## Styling the `<li>` Element

To make sure the list items are displayed in a single line, the `<li>` element must be set to `display: inline` as shown in Listing 16.7. The results can be seen in Figure 16.2.

**Listing 16.7** CSS Code Setting `display: inline;`

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}

ul#navigation li
{
 display: inline;
}
```

**FIGURE 16.2** Screenshot of list with the `<li>` element styled.

## Styling the `<a>` Element

You can increase the active area of text links by applying `display: block;` to the `<a>` element. This will change it from inline to block level and allow you to apply padding to all sides of the element.

Set the `<a>` element to `display: block;` so that padding can be applied to all sides. This will give the element additional width and height, increasing the clickable area.

The `<a>` element should then be floated, so that each list item moves into a single line butting against the previous item (see Listing 16.8).

**Listing 16.8** CSS Code Setting `display: block;`

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
}
```

*continues*

**Listing 16.8** Continued

---

```
float: left;
width: 100%;
}

ul#navigation li
{
 display: inline;
}

ul#navigation a
{
 display: block;
 float: left;
}
```



**Floats and Width** For this lesson, you will not be setting a width on the floated `<a>` elements. This will allow each list item to have its own width based on the number of characters and the surrounding padding.

However, this is not generally considered to be a good practice. It is best to set a width on all floated items (except if applied directly to an image, which has implicit width).

If no width is set, the results can be unpredictable. Theoretically, a floated element with an undefined width should shrink to the widest element within it. This could be a word, a sentence, or even a single character—and results can vary from browser to browser.

In this case, the results are acceptable because the styled list displays well in almost all modern browsers (including Internet Explore 5+, Netscape 6+, Opera 6+, Firefox, and Safari).

Next, add some padding using the padding declaration. You can use .2em for top and bottom padding, and 1em for left and right padding as shown in Listing 16.9.

---

**LISTING 16.9** CSS Code Setting Padding

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}

ul#navigation li
{
 display: inline;
}

ul#navigation a
{
 display: block;
 float: left;
 padding: .2em 1em;
}
```

To remove the underlines on the links, use `text-decoration: none;`. To set the text color and background color, use `color: #fff;` (white) and `background: #036;` as shown in Listing 16.10. These colors can be changed to suit your needs.

---

**LISTING 16.10** CSS Code Setting Text Decoration, Color, and Background Color

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}
```

*continues*



**LISTING 16.10** Continued

---

```
ul#navigation li
{
 display: inline;
}

ul#navigation a
{
 display: block;
 float: left;
 padding: .2em 1em;
 text-decoration: none;
 color: #fff;
 background: #036;
}
```

To separate each list item, a white line divider will be added to the end of each item. This is achieved by adding a white border to the right side of each list item, using `border-right: 1px solid #fff;` as shown in Listing 16.11 and illustrated in Figure 16.3.

**LISTING 16.11** CSS Code Setting a Border

---

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}

ul#navigation li
{
 display: inline;
}

ul#navigation a
{
 display: block;
 float: left;
 padding: .2em 1em;
 text-decoration: none;
```

*continues*

```
color: #fff;
background: #036;
border-right: 1px solid #fff;
}
```



**FIGURE 16.3** Screenshot of list with the `<a>` element styled.

## Styling the `:hover` Pseudo Class

Finally, the `:hover` pseudo class is used to change the style of links when they are rolled over. In this case, you will set the background to `#69C` and the color to `#000` as shown in Listing 16.12 and illustrated in Figure 16.4. These colors can be changed to suit your needs.

### LISTING 16.12 CSS Code Setting a Hover

```
ul#navigation
{
 margin-left: 0;
 padding-left: 0;
 list-style-type: none;
 background: #036;
 float: left;
 width: 100%;
}

ul#navigation li
{
 display: inline;
}

ul#navigation a
{
 display: block;
 float: left;
 padding: .2em 1em;
 text-decoration: none;
 color: #fff;
 background: #036;
```

*continues*

**LISTING 16.12** Continued

---

```
border-right: 1px solid #fff;
}

ul#navigation a:hover
{
 color: #000;
 background: #69C;
}
```



**FIGURE 16.4** Screenshot of finished list.

## Summary

In this lesson you have learned that lists can be styled to look like a horizontal navigation panel, how to float the `<ul>` and `<a>` elements, and how to use the `:hover` pseudo class. In the next lesson, you will learn how to style a round-cornered box.

# LESSON 17

## Styling a Round-Cornered Box



*In this lesson, you will learn how to create a flexible-width, round-cornered box using four corner images.*

### Setting Up the HTML Code

The HTML code for this lesson is comprised of an overall `<div>` container, a heading, and two paragraphs. The `<div>` is styled with a `pullquote` id, and the second paragraph is styled with a `furtherinfo` class as shown in Listing 17.1.

#### **LISTING 17.1** HTML Code Containing the Markup for a Round-Cornered Box

---

```
<div id="pullquote">
 <h2>Heading here</h2>
 <p>
 Lorem ipsum dolor sit amet....
 </p>
 <p class="furtherinfo">
 More information
 </p>
</div>
```

### Creating the Illusion of Round Corners

There are many methods that can be used to create a flexible-width, round-cornered box. The method described in this lesson uses four individual corner images.

These images should not be placed in the HTML code because they are purely presentational. Ideally, they should be applied as background images using CSS.

The top-left image will be applied as a background image to the `<div>` container and the top-right image will be applied to the `<h2>` element.

The bottom-left image will be applied to the last `<p>` element inside the box. This `<p>` element will be given a `.furtherinfo` class to differentiate it from other paragraphs in the box.

The bottom-right image will be applied to a specific instance of the `<a>` element.



**Adding Elements to Achieve Round Corners** There are many ways to create a flexible-width, round-cornered box.

One commonly used method involves nesting four levels of `<div>` elements that are then used to position the background images.

Where possible, it is better to use existing HTML elements or instances of elements rather than add new elements. Additional elements create unnecessary markup, which can increase page size and make maintenance more difficult.

## Creating Selectors to Style the Round-Cornered Box

To style the round-cornered box, you will need to use the five selectors shown in Listing 17.2.

---

### **LISTING 17.2** CSS Code Showing the Selectors for Styling the Two-Column Layout

---

```
div#pullquote
div#pullquote h2
```

```
div#pullquote p
div#pullquote p.furtherinfo
div#pullquote p.furtherinfo a
```

## Preparing the Images

The four images used in this lesson are shown in Figures 17.1 through 17.4. The style and color of these images can be changed to suit your needs.



**FIGURE 17.1** Image 1, which will be applied to the `<div>` element. The image should be made over 2,000 pixels long so that it will grow to the width of the widest monitors.



**FIGURE 17.2** Image 2, which will be applied to the `<h2>` element.



**FIGURE 17.3** Image 3, which will be applied to the `<p>` element styled with `.furtherinfo`.



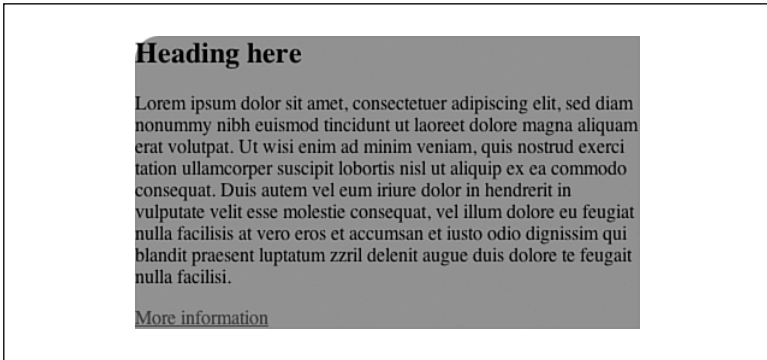
**FIGURE 17.4** Image 4, which will be applied to the `<a>` element.

## Styling the `<div>` Element

The first step in creating a round-cornered box is to style the `<div>`, which needs to be given some space on all sides. This can be achieved using `margin: 2em`. Next, the `<div>` needs to have a background image applied to the top-left corner. Use `background: #09f url(lesson17a.gif) no-repeat`; as shown in Listing 17.3. The results can be seen in Figure 17.5.

**LISTING 17.3** CSS Code for Styling the <div> Element

```
div#pullquote
{
 margin: 2em;
 background: #09f url(lesson17a.gif) no-repeat;
}
```



**FIGURE 17.5** Screenshot of styled <div> element.

## Styling the <h2> Element

Now that the <div> has been styled, the <h2> element will be used to position the second background image in the top-right corner.

The <h2> element's margins have to be turned off using `margin: 0`.

Next, the <h2> needs some padding on all edges except the bottom. This is achieved using `padding: 20px 20px 0 20px`.

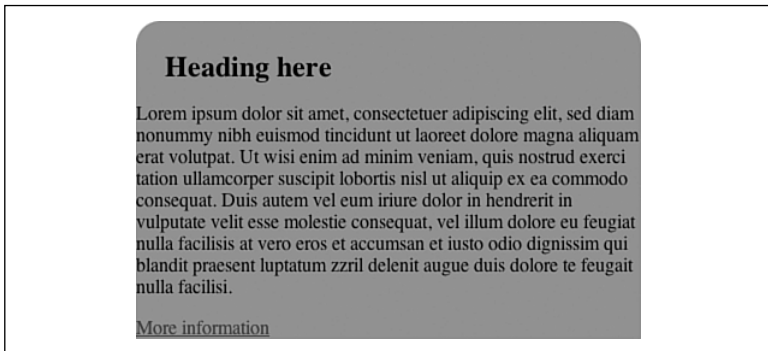
Finally, the background image is added using `background: url(lesson17b.gif) no-repeat 100% 0`. The horizontal background position is set to 100%, so the right edge of the image will sit against the right edge of the <h2> element. The image is also set to no-repeat so that it does not repeat across the background of the <h2> element as shown in Listing 17.4. The results can be seen in Figure 17.6.

**Listing 17.4** CSS Code for Styling the <div> Element

---

```
div#pullquote
{
 margin: 2em;
 background: #09f url(lesson17a.gif) no-repeat;
}

div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17b.gif) no-repeat 100% 0;
}
```



**FIGURE 17.6** Screenshot of styled <h2> element.

## Styling the <p> Element

The <p> element must be padded on both sides to keep it away from the edges of its container. This can be achieved using `padding: 0 20px` as shown in Listing 17.5. The results can be seen in Figure 17.7.

**Listing 17.5** CSS Code for Styling the <p> Element

---

```
div#pullquote
{
 margin: 2em;
 background: #09f url(lesson17a.gif) no-repeat;
}
```

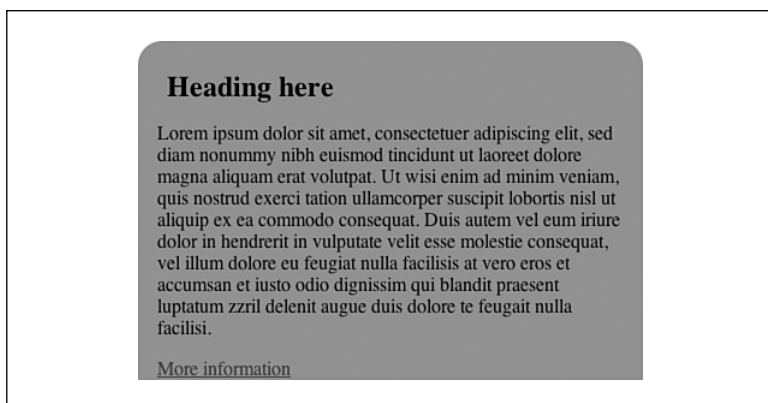
*continues*



**Listing 17.5** Continued

```
div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17b.gif) no-repeat 100% 0;
}

div#pullquote p
{
 padding: 0 20px;
}
```

**FIGURE 17.7** Screenshot of styled `<p>` element.

## Styling the `.furtherinfo` Class

The last paragraph inside the round-cornered box is styled with a `.furtherinfo` class. This paragraph will be used to position the third image in the bottom-left corner.

The paragraph must be given some padding, but it will only be padded on the left edge. This can be achieved using `padding: 0 0 0 20px`.

The background image can be set using `background:`

`url(lesson17c.gif) no-repeat 0 100%;`. The vertical background position is set to `100%`, so the bottom edge of the image will sit against the bottom of the `<p>` element. The image is also set to `no-repeat` so that it

does not repeat across the background of the <p> element as shown in Listing 17.6. The results can be seen in Figure 17.8.

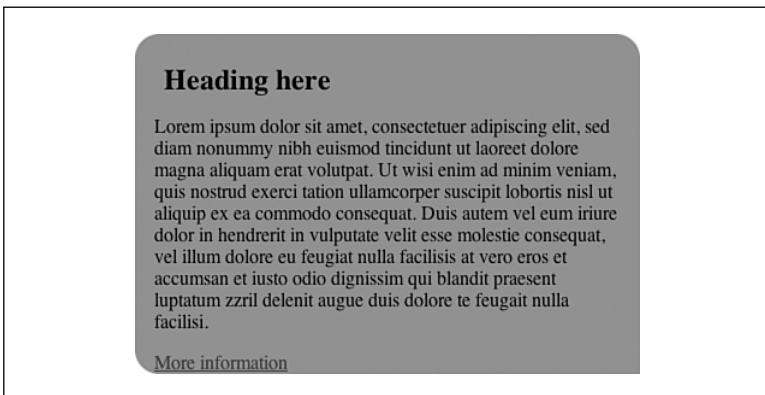
#### **LISTING 17.6** CSS Code for Styling the .furtherinfo Class

```
div#pullquote
{
 margin: 2em;
 background: #09f url(lesson17a.gif) no-repeat;
}

div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17b.gif) no-repeat 100% 0;
}

div#pullquote p
{
 padding: 0 20px;
}

div#pullquote p.furtherinfo
{
 padding: 0 0 0 20px;
 background: url(lesson17c.gif) no-repeat 0 100%;
}
```



**Figure 17.8** Screenshot of styled .furtherinfo class.

## Styling the <a> Element

The bottom-right image is applied to the <a> element.

In order for the image to display correctly, the <a> element must first be converted to block level using `display: block`.

The next step is to add padding using `padding: 0 20px 20px 0`; This will apply padding to the right and bottom of the element.

The content of the <a> element can be aligned to the right by using `text-align: right`.

Finally, the background image is applied using `background: url(lesson17d.gif) no-repeat 100% 100%`; as shown in Listing 17.7. This will apply the image to the bottom-right edge of the <a> element. The results can be seen in Figure 17.9.

### LISTING 17.7 CSS Code for Styling the <a> Element

---

```
div#pullquote
{
 margin: 2em;
 background: #09f url(lesson17a.gif) no-repeat;
}

div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17b.gif) no-repeat 100% 0;
}

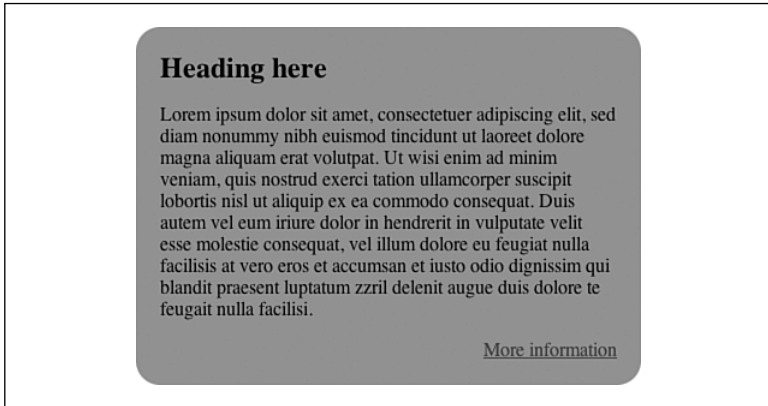
div#pullquote p
{
 padding: 0 20px;
}

div#pullquote p.furtherinfo
{
 padding: 0 0 0 20px;
 background: url(lesson17c.gif) no-repeat 0 100%;
}

div#pullquote p.furtherinfo a
```

*continues*

```
{
 display: block;
 padding: 0 20px 20px 0;
 text-align: right;
 background: url(lesson17d.gif) no-repeat 100% 100%;
}
```



**FIGURE 17.9** Screenshot of styled `<a>` element.

## Creating a Fixed-Width Variation

Creating a fixed-width version of the round-cornered box requires only two background images.

The first image can be applied to the `<div>` container using `background: #09f url(lesson17f.gif) no-repeat 0 100%;`. This will place the image in the bottom corner of the box. The image used is shown in Figure 17.10.



**FIGURE 17.10** Background image used to style the `<div>` element.

The `<div>` element needs to be given a width to match the width of the two images. `400px` has been used here. This measurement can be changed to suit your needs.

The `<div>` must also be given some bottom padding so that the text doesn't sit over the top of the image. This can be achieved using `padding-bottom: 20px`.

The `<p>` element must be padded on both sides to keep it away from the edges of its container. This can be achieved using `padding: 0 20px`.

The `<h2>` element must have the margins set to 0 so that there are no gaps at the top of the container. It must also be padded on the top right and left using `padding: 20px 20px 0 20px`.

A background image is applied to the `<h2>` element using `background: url(lesson17e.gif) no-repeat 100% 0`; . This will position the image at the top left of the element. The image is shown in Figure 17.11.



**FIGURE 17.11** Background image used to style the `<h2>` element.

Finally, the `.furtherinfo` paragraph content needs to be aligned right. This can be achieved using `text-align: right` as shown in Listing 17.8. The results can be seen in Figure 17.12.

### **LISTING 17.8** CSS Code for Styling the Fixed-Width, Round-Cornered Box

---

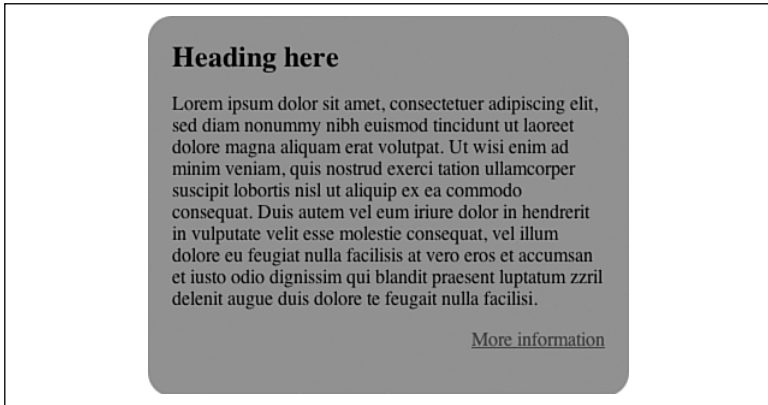
```
div#pullquote
{
 background: #09f url(lesson17f.gif) no-repeat 0 100%;
 width: 400px;
 padding-bottom: 20px;
}

div#pullquote p
{
 padding: 0 20px;
}

div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17e.gif) no-repeat 100% 0;
}
```

*continues*

```
div#pullquote p.furtherinfo
{
 text-align: right;
}
```

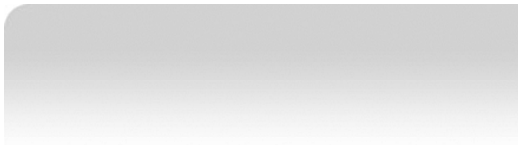


**FIGURE 17.12** Screenshot of fixed-width, round-cornered box.

## Creating a Top-Only Flexible Variation

To create a top-only flexible variation of the round-cornered box, two background images are needed.

The first image can be applied to the `<div>` container using `background: #fff url(lesson17g.jpg) no-repeat;`. This will place the image in the top-left corner of the box. The image used is shown in Figure 17.13.



**Figure 17.13** Background image used to style the `<div>` element. The image should be made over 2,000 pixels long so that it will grow to the width of the widest monitors.

The <p> element must be padded on both sides to keep it away from the edges of its container. This can be achieved using padding: 0 20px.

The <h2> element must have the margins set to 0 so that there are no gaps at the top of the container. It must also be padded on the top right and left using padding: 20px 20px 0 20px;.

A background image is applied to the <h2> element using background: url(lesson17h.jpg) no-repeat 100% 0;. This will position the image at the right of the element. The image is shown in Figure 17.14.



**FIGURE 17.14** Background image used to style the <h2> element.

Finally, the .furtherinfo paragraph content needs to be aligned right. This can be achieved using text-align: right as shown in Listing 17.9. The results can be seen in Figure 17.15.

### **LISTING 17.9** CSS Code for Styling the Top-Only Variation

---

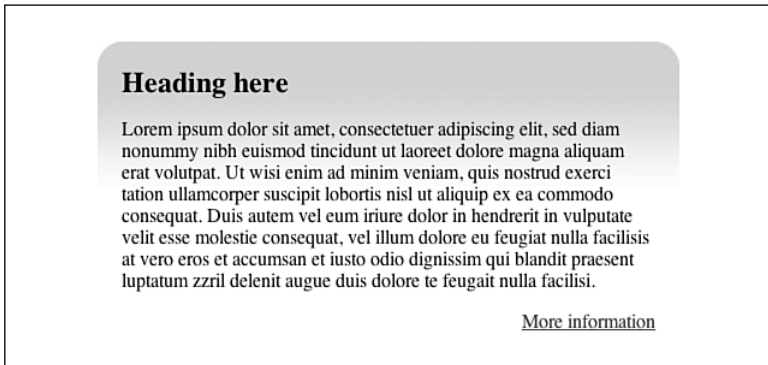
```
div#pullquote
{
 background: #fff url(lesson17g.jpg) no-repeat;
}

div#pullquote p
{
 padding: 0 20px;
}

div#pullquote h2
{
 margin: 0;
 padding: 20px 20px 0 20px;
 background: url(lesson17h.jpg) no-repeat 100% 0;
}
```

*continues*

```
div#pullquote p.furtherinfo
{
 text-align: right;
}
```



**FIGURE 17.15** Screenshot of styled top-only variation.

## Summary

In this lesson, you learned how to apply images to four different elements to create the illusion of a flexible, round-cornered box. You also learned how to create a fixed-width, round-cornered box and a top-only, round-cornered box. In the next lesson, you will learn how to set up and style a site header using an `<h1>` element, an image, and a list.





# LESSON 18

## Creating a Site Header

*In this lesson, you will learn how to set up and style a site header using an `<h1>` element, an image, and a list.*

### Setting Up the HTML Code

The HTML code for this lesson is comprised of three main components: a `<div>` element, which helps define the header section semantically; an `<h1>` element; and a `<ul>` element for navigation as shown in Listing 18.1.

#### **LISTING 18.1** HTML Code Containing the Markup for the Site Header

---

```
<div id="container">
 <h1>

 </h1>
 <ul id="topnav">
 Skip to content
 Home
 About
 Services
 Staff
 Portfolio
 Contact

</div>
```



**Skip to content Link** What does the Skip to content link mean?

This link, called a *skip link*, enables users to skip over navigation links to get to the main content of the page.

Skip links are beneficial for blind, visually impaired, and mobility impaired users, as well as people who use text browsers, mobile phones, and PDAs.

Some developers use hidden skip links, which are specifically designed for blind users. However, skip links should be visible for people who cannot use the mouse and must use tabbing to navigate websites.

## Creating Selectors to Style the Header

To style the site header and its content, you will use 10 selectors as shown in Listing 18.2.

### LISTING 18.2 CSS Code Showing the Selectors for Styling the Header

---

```
body {...}
#container {...}
h1 {...}
h1 img {...}
ul#topnav {...}
ul#topnav li {...}
ul#topnav li a:link {...}
ul#topnav li a:visited {...}
ul#topnav li a:hover {...}
ul#topnav li a:active {...}
```

## Styling the <body> Element

Some browsers use margins and others use padding on the <body> element to indent content from the edges of the browser window. Because

this site header sits against the top edge of the browser window, you will need to set both margins and padding to 0.

Theoretically, you should be able to apply auto margins to the left and right of a container to center it on the page. However, some browsers won't center the container using this method because they ignore the auto margins. This problem can be overcome by adding two simple declarations.

The first declaration, `text-align: center`, is applied to the `<body>` element. The second declaration, `text-align: left`, is added to the container rule set (see "Styling the Container" later in this lesson).

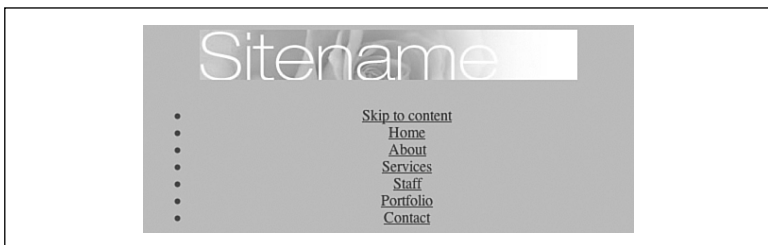
A `background-color` and `color` must also be set on the `<body>` element. In this case, you can use a background color of `#B0BFC2` and a color of `#444` as shown in Listing 18.3. The results can be seen in Figure 18.1.

---

**LISTING 18.3** CSS Code for Styling the Body

---

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 background: #B0BFC2;
 color: #444;
}
```



**FIGURE 18.1** Screenshot of styled `<body>`.

## Styling the Container

Now that the `<body>` element has been styled, all content will be centered on the page. This can be overcome by setting the container to `text-align: left`.

For browsers that support auto margins, the correct centering method is then applied: `margin: 0 auto`.

The container can be set to a width of 700px. This width can be changed to suit your needs.

Finally, the container must be set with a white background using `background: #fff` as shown in Listing 18.4. The results can be seen in Figure 18.2.

#### LISTING 18.4 CSS Code for Styling the Container

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF;
}
```



FIGURE 18.2 Screenshot of styled container.

## Styling the <h1> Element

As you saw in Listing 18.1, the <h1> element contains an image. This site header graphic is placed inside the <h1> element to give it greater semantic meaning. Screen readers and text-based browsers will read the alt attribute "Sitename" as if it were a text heading.

You will need to set margins and padding to 0 so that the image can sit against the top edge of the browser window.

You also can add a white border to the bottom of the <h1> element using the shorthand border: 1px solid #fff; as shown in Listing 18.5. The results can be seen in Figure 18.3.

### LISTING 18.5 CSS Code for Styling the <h1> Element

---

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF;
}

h1
{
 margin: 0;
 padding: 0;
 border-bottom: 1px solid #fff;
}
```



**FIGURE 18.3** Screenshot of styled `<h1>` element.

## Styling the `<image>` Element

The image should be set to `display: block`. This will remove any gaps that appear below it and force the `<h1>` `border-bottom` to sit up against it.

The image is nested inside an `<a>` element. In some browsers this will cause the image to display with a 2-pixel-wide border. To avoid this, the image should be styled with `border: 0` as shown in Listing 18.6. The results can be seen in Figure 18.4.

### **LISTING 18.6** CSS Code for Styling the `<image>` Element

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 font: 85% arial, helvetica, sans-serif;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF;
}

h1
```

*continues*

**LISTING 18.6** Continued

```

{
 margin: 0;
 padding: 0;
 border-bottom: 1px solid #fff;
}

h1 img
{
 display: block;
 border: 0;
}

```

**FIGURE 18.4** Screenshot of styled `<img>` element.

## Styling the `<ul>` Element

As discussed in Lesson 15, “Creating Vertical Navigation,” most browsers display HTML lists with left indentation. To remove this left indentation consistently across all browsers, you must override both margins and padding.

Here, you will set the margin to 0 so that the list sits up against the `<h1>` element.

The list items will need to be padded with 5px above and below, and 10px to the left and right. This can be achieved with a shorthand padding declaration of `padding: 5px 10px`.

The HTML bullets must be removed using `list-style-type: none`.

The `<ul>` can be styled with `background-color: #387A9B`.

Because you do not want these rules to be applied to all `<ul>` elements on the page, you should use a descendant selector that targets the topnav list only. This is achieved by using `ul#topnav` as shown in Listing 18.7. The results can be seen in Figure 18.5.

---

**LISTING 18.7** CSS Code for Styling the `<ul>` Element

---

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 font: 85% arial, helvetica, sans-serif;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF;
}

h1
{
 margin: 0;
 padding: 0;
 border-bottom: 1px solid #fff;
}

h1 img
{
 display: block;
 border: 0;
}

ul#topnav
{
 margin: 0;
 padding: 5px 10px;
 list-style-type: none;
 background: #387A9B;
```





**FIGURE 18.5** Screenshot of styled `<ul>` element.

## Styling the `<li>` Element

The list must be displayed across the screen rather than down. This is achieved by setting the `<li>` to `display: inline`.

Next, a graphic bullet needs to be added to the `<li>`. The best way to do this is by using a background image set with `background: url(header-bullet.gif) no-repeat 0 50%;`. This will place one image vertically centered beside each list item.

Padding will need to be added to move the text away from the background image. In this case, `padding: 0 10px 0 8px;` will be used as shown in Listing 18.8. This will apply 10px of padding to the right and 8px to the left of each list item. The results can be seen in Figure 18.6.

### **LISTING 18.8** CSS Code for Styling the `<li>` Element

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 font: 85% arial, helvetica, sans-serif;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
```

*continues*

```
 width: 700px;
 background: #FFF;
 }

 h1
 {
 margin: 0;
 padding: 0;
 border-bottom: 1px solid #fff;
 }

 h1 img
 {
 display: block;
 border: 0;
 }

 ul#topnav
 {
 margin: 0;
 padding: 5px 10px;
 list-style-type: none;
 background: #387A9B;
 }

 ul#topnav li
 {
 display: inline;
 background: url(header-bullet.gif) no-repeat 0 50%;
 padding: 0 10px 0 8px;
 }
```



**FIGURE 18.6** Screenshot of styled `<li>` element.

## Styling the <a> Element

To avoid targeting all links on the page, a specific selector should be used. Here, you will use `ul#topnav li a:link`, `ul#topnav li a:visited`, `ul#topnav li a:hover`, and `ul#topnav li a:active`.

The link and visited pseudo-classes can be set with `text-decoration: none` (which will turn off link underlines) and `color: #FFF` as shown in Listing 18.9.

The hover and active pseudo-classes also will be set with `text-decoration: none` as well as `color: #387A9B`; and `background: #FFF`; . The results can be seen in Figure 18.7.

### **LISTING 18.9**    CSS Code for Styling the <a> Element

---

```
body
{
 margin: 0;
 padding: 0;
 text-align: center;
 font: 85% arial, helvetica, sans-serif;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF;
}

h1
{
 margin: 0;
 padding: 0;
 border-bottom: 1px solid #fff;
}

h1 img
```

*continues*

```
{
 display: block;
 border: 0;
}

ul#topnav
{
 margin: 0;
 padding: 5px 10px;
 list-style-type: none;
 background: #387A9B;
}

ul#topnav li
{
 display: inline;
 background: url(header-bullet.gif) no-repeat 0 50%;
 padding: 0 10px 0 8px;
}

ul#topnav li a:link, ul#topnav li a:visited
{
 text-decoration: none;
 color: #fff;
}

ul#topnav li a:hover, ul#topnav li a:active
{
 text-decoration: none;
 color: #387A9B;
 background: #fff;
}
```



**FIGURE 18.7** Screenshot of styled <a> element.

## Summary

In this lesson, you have learned how to center content within a browser window and style an `<h1>` element, an `<img>` element, and a `<ul>` element as links. In the next lesson, you will learn how to position a two-column page layout with a header and a footer.

# LESSON 19

## Positioning Two Columns with a Header and a Footer



*In this lesson, you will learn how to position a two-column page layout with a header and a footer. There are many ways to position these two columns. This method involves floating them both because it is the most reliable method across most modern browsers.*

### Setting Up the HTML Code

The HTML code for this lesson is comprised of five main containers: an `<h1>` element, and three `<div>` elements inside an overall `<div>` container as shown in Listing 19.1.

#### **LISTING 19.1** HTML Code Containing the Markup for a Two-Column Layout

---

```
<div id="container">
 <h1>
 Sitename
 </h1>
 <div id="nav">

 Home
 About us
 Services
 Staff
 Portfolio
 Contact us

 </div>
```

*continues*

**LISTING 19.1** Continued

---

```
</div>
<div id="content">
 <h2>
 About Sitename
 </h2>
 <p>
 Lorem ipsum dolor...
 </p>
 <p>
 Ut wisi enim ad...
 </p>
</div>
<div id="footer">
 Copyright © Sitename 2005
</div>
</div>
```

## Creating Selectors to Style the Two-Column Layout

To style the two-column layout and its content, you will use 12 selectors as shown in Listing 19.2.

**LISTING 19.2** CSS Code Showing the Selectors for Styling a Two-Column Layout

---

```
body {...}
#container {...}
h1 {...}
#nav {...}
#nav ul {...}
#nav li {...}
#content {...}
#footer {...}
h2 {...}
a:link {...}
a:visited {...}
a:hover, a:active {...}
```

## Styling the <body> Element

As discussed in Lesson 18, “Creating a Site Header,” to center a container on the page, you must find ways to work around browsers that don’t support auto margins. The first work-around involves applying `text-align: center` to the `<body>` element as shown in Listing 19.3. The results can be seen in Figure 19.1.

A `background-color` and `color` also must be set on the `<body>` element. You can use a background color of `#B0BFC2` and a color of `#444`.

### LISTING 19.3 CSS Code for Styling the Body

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}
```

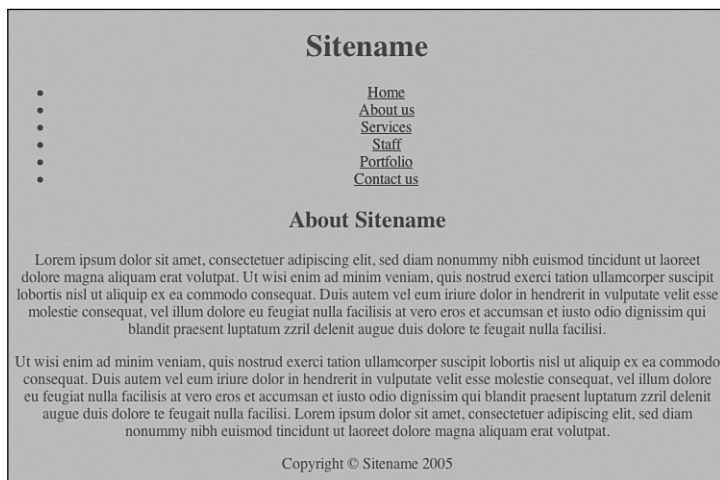


FIGURE 19.1 Screenshot of styled `<body>`.



## Styling the Container

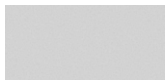
Now that the `<body>` element has been styled, all content will be centered on the page. This can be overcome by setting the container to `text-align: left`.

For browsers that support auto margins, the correct centering method is then applied: `margin: 0 auto`.

The container can be set to a width of 700px. This width can be changed to suit your needs.

Finally, the container must be set with a background image using `background: #FFF url(header-base.gif) repeat-y`; as shown in Listing 19.4.

The background image can be seen in Figure 19.2 and the results are shown in Figure 19.3.



**FIGURE 19.2** Background image used for container.



**Creating the Illusion of Column Colors** One problem with floating containers is that they will generally only extend to the depth of their content. If one column is much shorter than another, it can be very hard to create columns using background colors alone.

So, how do you get the shorter column's background color to extend to the bottom of the page?

One simple solution is to use a background image that gives the illusion of column colors. This image can be added to the overall container as a background image and repeated down the y axis.

The floated containers then sit over the top of this repeated image, and the colors will extend to the bottom of the page, no matter which column is longer.

**LISTING 19.4** CSS Code for Styling the Container

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}
```

**FIGURE 19.3** Screenshot of styled container.

## Styling the <h1> Element

The first step in styling the heading is to set a background color. You can use `background: #D36832`. The color can then be set to `#fff`.

Next, `padding: 20px` can be applied to create some space around the `<h1>` content.

You will then need to set `margin: 0` to remove the default top and bottom margins.

You also can add a border to the bottom of the `<h1>` element using the shorthand `border: 5px solid #387A9B`; as shown in Listing 19.5. The results can be seen in Figure 19.4.

---

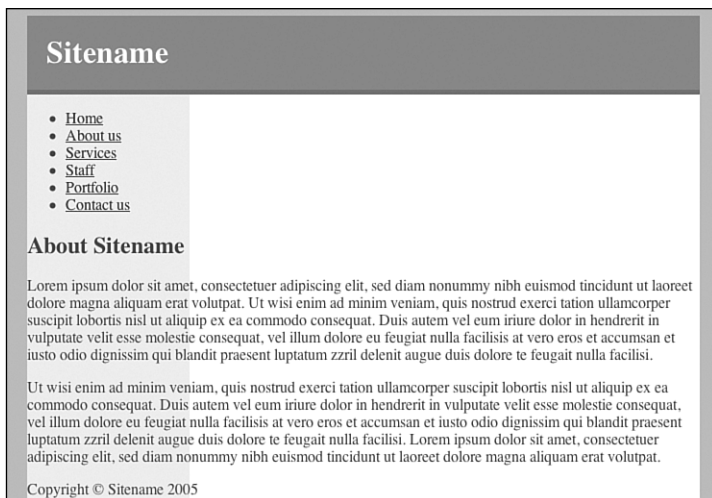
**LISTING 19.5**    CSS Code for Styling the `<h1>` Element

---

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}
```



**FIGURE 19.4** Screenshot of styled `<h1>` element.

## Styling the #nav Container

To position the `#nav` container and `#content` container beside each other, they will both need to be floated.

To float the `#nav` container, use `float: left`. You also will need to set a width, which in this case will be `130px`.

Internet Explorer 5 and 5.5 for Windows will sometimes display margins at double the specified width in certain circumstances. This *Double Margin Float Bug* is explained in Lesson 11, “Positioning an Image and Its Caption.” The bug can be fixed by setting the floated item to `display: inline`. All other browsers will ignore this declaration, but Internet Explorer 5 and 5.5 for Windows will then apply the correct margin width.

Now that the margins will display correctly in all recent browsers, you can apply `margin-left: 20px`.

Finally, padding needs to be applied to the top and bottom of the container. Here, you will need to use `padding: 15px 0` as shown in Listing 19.6. The results can be seen in Figure 19.5.

---

**LISTING 19.6**    CSS Code for Styling the #nav Container

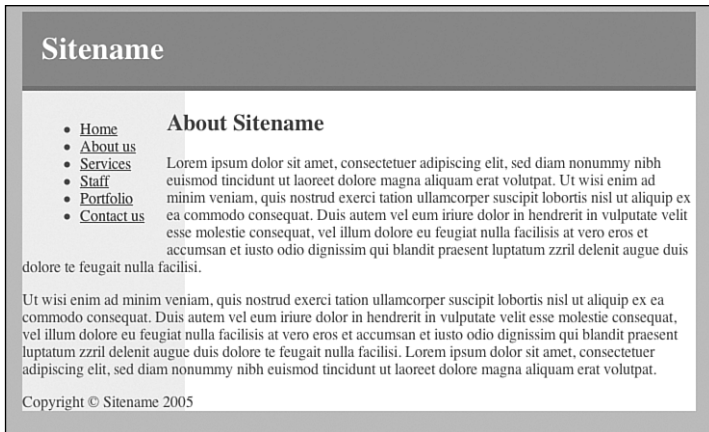
---

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}
```



**Figure 19.5** Screenshot of styled #nav container.

## Styling the <u1> Element

Both the margins and padding of the <u1> need to be set to 0 to remove any browser default styling.

To remove list bullets, use `list-style-type: none`.

The list items will need to be aligned against the right edge of the #nav container. This can be achieved using `text-align: right` as shown in Listing 19.7. The results can be seen in Figure 19.6.

### LISTING 19.7 CSS Code for Styling the <u1> Element

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}
```

*continues*

**LISTING 19.7**    Continued

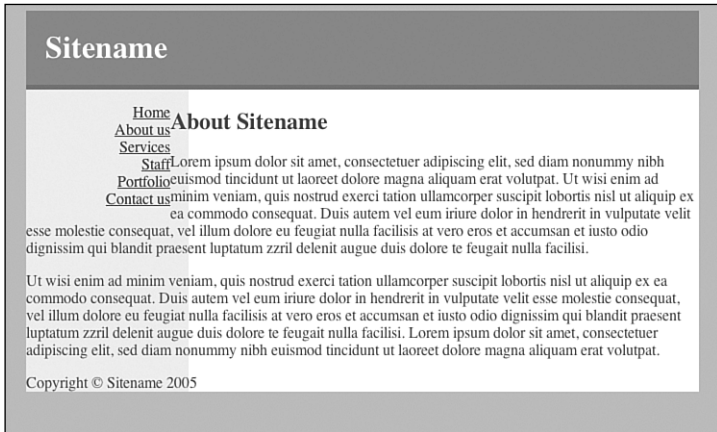
---

```
#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}
```



**FIGURE 19.6** Screenshot of styled `<ul>` element.

## Styling the `<li>` Element

The list items will now have a background image applied to them to act as a bullet. You will use `background: url(header-bullet.gif) no-repeat 100% .4em;` to place the background image against the right edge of the `<li>` element, and `.4em` from the top. The image is also set to `no-repeat`, so it does not repeat across the entire `<li>` element.

Padding can then be applied to the right edge and bottom of the `<li>`. The right padding will move the list content away from the edge so that it does not appear over the top of the background image. The bottom padding is used to provide some space between list items as shown in Listing 19.8. The results can be seen in Figure 19.7.

### LISTING 19.8 CSS Code for Styling the `<li>` Element

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}
```

*continues*



**LISTING 19.8**    Continued

---

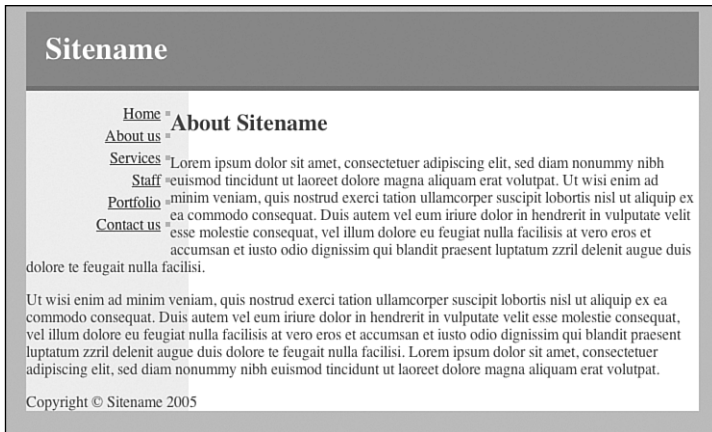
```
#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}

#nav li
{
 background: url(header-bullet.gif) no-repeat 100% .4em;
 padding: 0 10px 5px 0;
}
```



**FIGURE 19.7** Screenshot of styled `<li>` element.

## Styling the #content Container

The #content container needs to be set to `float: left` so that it sits beside the #nav container. You also will need to set a width, which in this case will be 475px.

To create a gutter between the two columns, use `margin-left: 45px`.

Finally, padding needs to be applied to the top and bottom of the container. You will need to use `padding: 15px 0` as shown in Listing 19.9. The results can be seen in Figure 19.8.

### LISTING 19.9 CSS Code for Styling the #content Container

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
```

*continues*

**LISTING 19.9**    Continued

---

```
margin: 0 auto;
width: 700px;
background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}

#nav li
{
 background: url(header-bullet.gif) no-repeat 100% .4em;
 padding: 0 10px 5px 0;
}

#content
{
 float: left;
 width: 475px;
 margin-left: 45px;
 padding: 15px 0;
}
```



**FIGURE 19.8** Screenshot of styled #content container.

## Styling the #footer Container

The #footer container is displayed after the #nav and #content containers. Because these two containers are floated, there is a possibility that the #footer container might try to sit beside them. This can be fixed using the clear property on the #footer container. The four options are clear: left, clear: right, clear: both, and clear: none.

Here you will use clear: both, which will force the #footer container to sit below the two floated containers.

To set a background color, use background: #387A9B. The color can then be set to #fff.

Next, padding can be used to create some space around the content. You can apply 5px to the top and bottom, and 10px to the left and right edges using padding: 5px 10px as shown in Listing 19.10.

To align the footer content to the right, use text-align: right.

Finally, the font size of the footer can be reduced because it is less important information. You can use font-size: 80% as shown in Listing 19.10. The results can be seen in Figure 19.9.

---

**LISTING 19.10**    CSS Code for Styling the #footer Container

---

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}

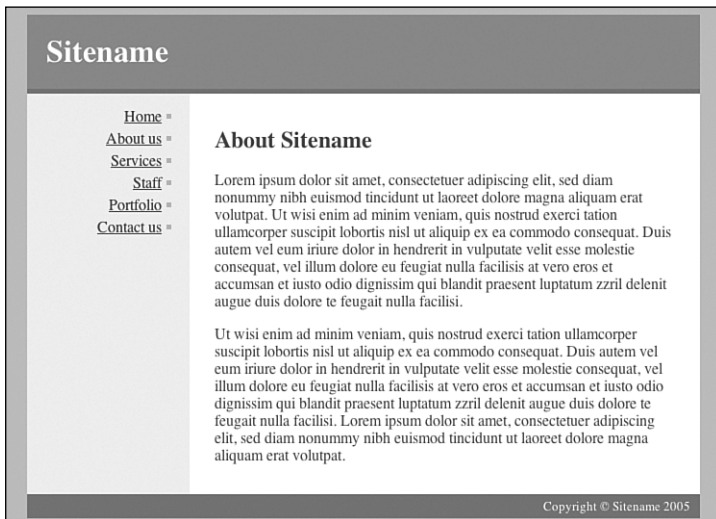
#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}
```

*continues*

```
#nav li
{
 background: url(header-bullet.gif) no-repeat 100% .4em;
 padding: 0 10px 5px 0;
}

#content
{
 float: left;
 width: 475px;
 margin-left: 45px;
 padding: 15px 0;
}

#footer
{
 clear: both;
 background: #387A9B;
 color: #fff;
 padding: 5px 10px;
 text-align: right;
 font-size: 80%;
}
```



**FIGURE 19.9** Screenshot of styled #footer container.

## Styling the <h2> Element

The <h2> element is used for the main heading on the page. Its top margin needs to be removed so that the <h2> element lines up with the content in the #nav container. This is achieved using `margin-top: 0`.

Next, the color can be changed using `color: #B23B00`.

Standard headings generally are displayed in bold text. You can override this default behavior using `font-weight: normal` as shown in Listing 19.11. The results can be seen in Figure 19.10.

---

**LISTING 19.11**    CSS Code for Styling the <h2> Element

---

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
```

*continues*

```
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
 }

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}

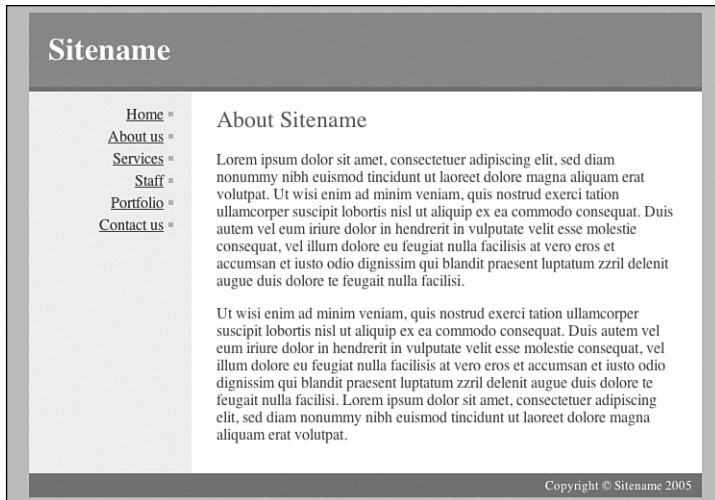
#nav li
{
 background: url(header-bullet.gif) no-repeat 100% .4em;
 padding: 0 10px 5px 0;
}

#content
{
 float: left;
 width: 475px;
 margin-left: 45px;
 padding: 15px 0;
}

#footer
{
 clear: both;
 background: #387A9B;
 color: #fff;
 padding: 5px 10px;
 text-align: right;
 font-size: 80%;
}

h2
{
 margin-top: 0;
 color: #B23B00;
 font-weight: normal;
}
```





**FIGURE 19.10** Screenshot of styled `<h2>` element.

## Styling the `<a>` Element

The final step in this lesson involves setting the link colors. You will work on four pseudo-classes.

The `a:link` pseudo-class can be set to `color: #175B7D`, and the `a:visited` pseudo-class can be set to `color: #600`.

The `a:hover` and `a:active` pseudo-classes also can be set with `color: #fff`; and `background: #175B7D` as shown in Listing 19.12. The results can be seen in Figure 19.11.

### **LISTING 19.12** CSS Code for Styling the `<h2>` Element

```
body
{
 text-align: center;
 background: #B0BFC2;
 color: #444;
}

#container
```

*continues*

```
{
 text-align: left;
 margin: 0 auto;
 width: 700px;
 background: #FFF url(header-base.gif) repeat-y;
}

h1
{
 background: #D36832;
 color: #FFF;
 padding: 20px;
 margin: 0;
 border-bottom: 5px solid #387A9B;
}

#nav
{
 float: left;
 width: 130px;
 display: inline;
 margin-left: 20px;
 padding: 15px 0;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 text-align: right;
}

#nav li
{
 background: url(header-bullet.gif) no-repeat 100% .4em;
 padding: 0 10px 5px 0;
}

#content
{
 float: left;
 width: 475px;
 margin-left: 45px;
```

*continues*

**LISTING 19.12** Continued

---

```
padding: 15px 0;
}

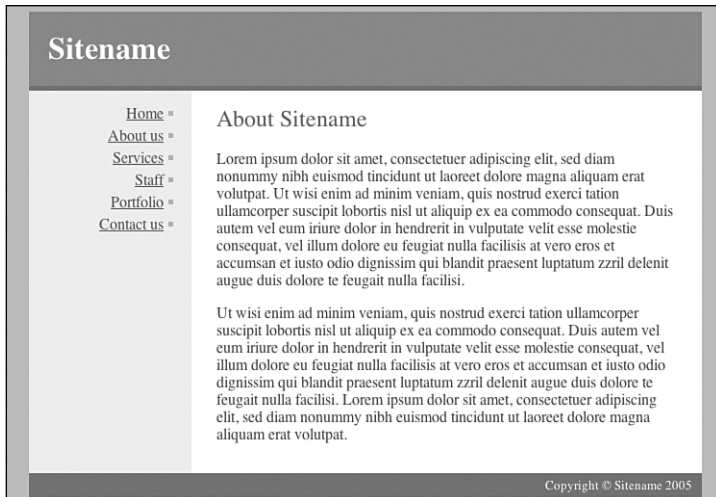
#footer
{
 clear: both;
 background: #387A9B;
 color: #fff;
 padding: 5px 10px;
 text-align: right;
 font-size: 80%;
}

h2
{
 margin-top: 0;
 color: #B23B00;
 font-weight: normal;
}

a:link
{
 color: #175B7D;
}

a:visited
{
 color: #600;
}

a:hover, a:active
{
 color: #fff;
 background: #175B7D;
}
```



**FIGURE 19.11** Screenshot of styled `<a>` element.

## Summary

In this lesson, you learned how to center content within a browser window, style an `<h1>` element, and position and style two columns and a footer. In the next lesson, you will learn how to style a page for print using CSS.



## LESSON 20

# Styling a Page for Print

*In this lesson, you will learn how to style a page for print using CSS. You will also learn about the media type and how it is used.*

## Setting Up the Print CSS

In the past, many sites provided two versions of each page—one designed for the screen and the other designed to be printed. The two versions generally used the same content, but were presented in different ways.

One advantage of CSS is that you can style a page for print without the need for additional pages. This is achieved using the `media` attribute, which can be applied to external style sheet links (as shown in Listing 20.1) or to the `style` element for embedded styles (as shown in Listing 20.2).

### **LISTING 20.1** HTML Code Containing a Link to an External Style Sheet with a Media Value of screen

---

```
<link rel="stylesheet" href="lesson20.css" type="text/css"
media="screen">
```

### **LISTING 20.2** HTML Code Containing an Embedded Style Sheet with a Media Value of screen

---

```
<style type="text/css" media="screen">
<!--
... Screen style rules go here ...
-->
</style>
```

If your layout needs to be identical for screen and print, the media attribute value can be set to `all` as shown in Listing 20.3 (for external style sheets) and Listing 20.4 (for embedded styles).

### **LISTING 20.3** HTML Code Containing a Link to an External Style Sheet with a Media Value of `all`

---

```
<link rel="stylesheet" href="lesson20.css" type="text/css"
media="all">
```

### **LISTING 20.4** HTML Code Containing an Embedded Style Sheet with a Media Value of `all`

---

```
<style type="text/css" media="all">
<!--
... All style rules go here ...
-->
</style>
```

If the layout needs to be different for screen and print, the media attribute values can be set to `screen` and `print` as shown in Listings 20.5 and 20.6.

### **LISTING 20.5** HTML Code Containing Links to External Style Sheets with Media Values Set to `screen` and `print`

---

```
<link rel="stylesheet" href="lesson20.css" type="text/css"
media="screen">
<link rel="stylesheet" href="lesson20-print.css"
type="text/css" media="print">
```

### **LISTING 20.6** HTML Code Containing Embedded Style Sheets with Media Values Set to `screen` and `print`

---

```
<style type="text/css" media="screen">
<!--
... Screen style rules go here ...
-->
</style>

<style type="text/css" media="print">
<!--
... Print style rules go here ...
-->
</style>
```



**Media Types and Older Browsers** There are currently 10 media types within the CSS2 Specification. They are

- **all**—Suitable for all devices
- **aural**—For speech synthesizers
- **braille**—For Braille tactile-feedback devices
- **embossed**—For paged Braille printers
- **handheld**—For handheld devices such as mobile phones and PDAs
- **print**—For print material and for documents viewed in Print Preview mode
- **projection**—For projected presentations
- **screen**—For color computer screens
- **tty**—For media using fixed-pitch character grids such as teletypes and terminals
- **tv**—For television-type devices

Multiple media types can be provided within the same attribute as long as they are separated by commas. For example, you could use `media="print, projection"` to target these media types only.

If no media type is specified, `screen` will be applied because it is the default value.

Some early browsers, such as Netscape Navigator 4, do not understand the `all` media type or comma-separated media types. If you intend to support these browsers, it might be best to specify `screen` or no media type at all.

# Starting with Existing HTML and CSS Code

For this lesson, you will use the HTML code from Lesson 19, “Positioning Two Columns with a Header and a Footer,” as shown in Listing 20.7 and rework it for print.

## LISTING 20.7 HTML Code Containing the Markup for a Two-Column Layout

---

```
<div id="container">
 <h1>
 Sitename
 </h1>
 <div id="nav">

 Home
 About us
 Services
 Staff
 Portfolio
 Contact us

 </div>
 <div id="content">
 <h2>
 About Sitename
 </h2>
 <p>
 Lorem ipsum dolor...
 </p>
 <p>
 Ut wisi enim ad...
 </p>
 </div>
 <div id="footer">
 Copyright © Sitename 2005
 </div>
</div>
```



## Creating Selectors to Style for Print

To style the two-column layout for print, you will use five selectors as shown in Listing 20.8.

### LISTING 20.8 CSS Code Showing the Selectors for Styling the Two-Column Layout for Print

---

```
body {...}
h1 {...}
#nav {...}
#footer {...}
a {...}
```



**Some Warnings About Styling for Print** When styling pages for print, you should be aware that floated containers and complex absolute positioning can cause problems in some browsers.

*Long floated containers* can cause problems in some versions of Mozilla and Netscape. These browsers sometimes have trouble calculating the length of floated containers and will only print the first page of the container's content.

*Complex absolute positioning* (also layouts that need to be pixel perfect) can cause problems for Internet Explorer 6. This browser has been known to crash when using Print Preview to view absolutely positioned layouts.

Although some floated and absolutely positioned content can be used, it is best to keep the overall layout as simple as possible when styling for print.

Many browsers do not print *background images* as a default. For this reason, background images should not be used to display information that is critical to the reader.

## Styling the <body> Element

The <body> element will be styled with font, color, and background declarations as shown in Listing 20.9.

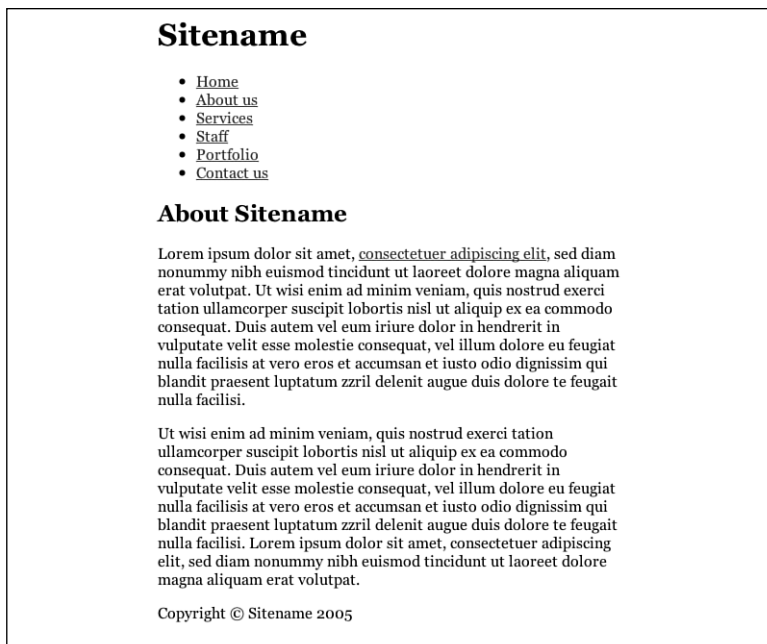
The font property will be used to determine the base font size and family for all elements on the page. In this example, 100% has been used because it enables the user to control the font size. A serif font has been chosen because it is more readable in print, whereas a sans-serif font is more readable on a screen or monitor (small cell phone screens in particular make serif type difficult to read). These can be changed to suit your needs.

No margins or padding have been specified for the <body> element; these will be determined by the printer. The results can be seen in Figure 20.1.

---

### LISTING 20.9 CSS Code for Styling the Body

```
body
{
 font: 100% georgia, times, serif;
 background: #fff;
 color: #000;
}
```



**FIGURE 20.1** Screenshot of styled `<body>`.

## Styling the `<h1>` Element

The `<h1>` element will be styled very simply with `border-bottom` and `margin-bottom` as shown in Listing 20.10 and Figure 20.2.

### **Listing 20.10** CSS Code for Styling the `<h1>` Element

---

```
body
{
 font: 100% georgia, times, serif;
 background: #fff;
 color: #000;
}

h1
{
 border-bottom: 1px solid #999;
 margin-bottom: 1em;
}
```



**What Is an Em?** In traditional typesetting, an *em* space is defined as the width of an uppercase *M* in the current face and point size. An *em dash* is the width of a capital *M*, an *en dash* is half the width of a capital *M*, and an *em quad* (a unit of spacing material typically used for paragraph indentation) is the square of a capital *M*.

In CSS, an *em* is a relative measure of length that inherits size information from parent elements. If the parent element is the `<body>`, the size of the element is actually determined by the user's browser font settings. So in most browsers, where the default font size is 16px, 1em will be 16px.

## Sitename

---

- [Home](#)
- [About us](#)
- [Services](#)
- [Staff](#)
- [Portfolio](#)
- [Contact us](#)

### About Sitename

Lorem ipsum dolor sit amet, [consectetuer adipiscing elit](#), sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, [consectetuer adipiscing elit](#), sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

Copyright © Sitename 2005

**FIGURE 20.2** Screenshot of styled `<h1>`.



**Hiding Content from the Printer** Some website content has no purpose on a printed page, such as site-based navigation or some advertising.

These areas of content can be hidden from the printer using `display: none;`.

When this declaration is applied to an element, the elements and all descendants will not be displayed. You cannot override this behavior by setting a different display property on the descendant elements.

## Styling the #nav Container

To remove the navigation from the printed page, use `display: none;` as shown in Listing 20.11. The results can be seen in Figure 20.3.

---

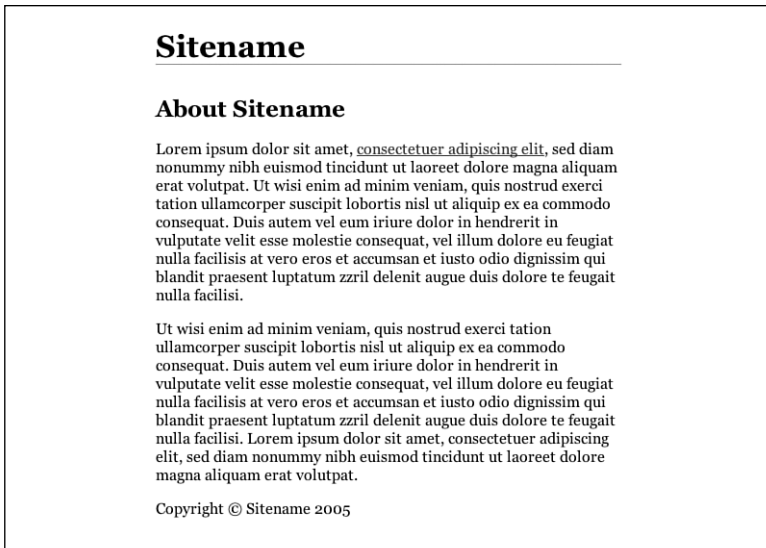
### LISTING 20.11 CSS Code for Styling the #nav Container

---

```
body
{
 font: 100% georgia, times, serif;
 background: #fff;
 color: #000;
}

h1
{
 border-bottom: 1px solid #999;
 margin-bottom: 1em;
}

#nav
{
 display: none;
}
```



**FIGURE 20.3** Screenshot of styled #nav container.

## Styling the #footer Container

The footer can be separated from other content using `border-top` and `margin-top` properties.

The footer content also can be right-aligned using the `text-align` property.

To add space between the border and footer content, `padding-top` can be used as shown in Listing 20.12. The results can be seen in Figure 20.4.

### **LISTING 20.12** CSS Code for Styling the #footer Container

```
body
{
 font: 100% georgia, times, serif;
 background: #fff;
 color: #000;
}

h1
```

*continues*

**LISTING 20.12** Continued

```
{
 border-bottom: 1px solid #999;
 margin-bottom: 1em;
}

#nav
{
 display: none;
}

#footer
{
 border-top: 1px solid #999;
 text-align: right;
 margin-top: 3em;
 padding-top: 1em;
}
```

## Sitename

---

### About Sitename

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi.

Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat.

---

Copyright © Sitename 2005

**FIGURE 20.4** Screenshot of styled #footer container.

## Styling the <a> Element

Hyperlinks have no real value on a printed page. To make links appear the same as all other content, you could set the color to #000 and then turn off underlines with `text-decoration: none`; as shown in Listing 20.13. The final result can be seen in Figure 20.5.

---

**LISTING 20.13** CSS Code for Styling the <a> Element

---

```
body
{
 font: 100% georgia, times, serif;
 background: #fff;
 color: #000;
}

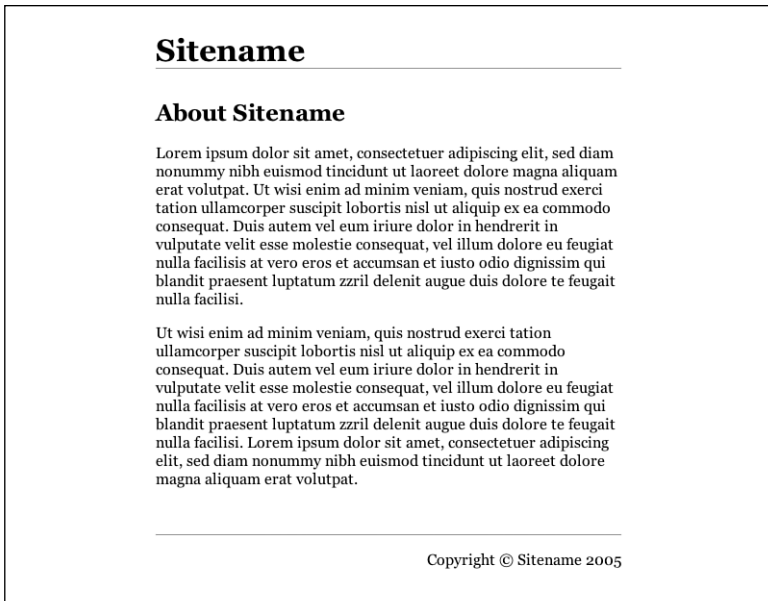
h1
{
 border-bottom: 1px solid #999;
 margin-bottom: 1em;
}

#nav
{
 display: none;
}

#footer
{
 border-top: 1px solid #999;
 text-align: right;
 margin-top: 3em;
 padding-top: 1em;
}

a
{
 color: #000;
 text-decoration: none;
}
```





**FIGURE 20.5** Screenshot of styled `<a>` element.

## Summary

In this lesson, you have learned how to style a layout specifically for print using the media attribute. In the next lesson, you will learn how to position a three-column liquid layout with a header and a footer.

# LESSON 21

## Positioning Three Columns with a Header and a Footer



*In this lesson, you will learn how to position a three-column liquid layout with a header and a footer. This method involves placing background images inside two containers to give the illusion of column colors, and then floating all three columns.*

### Setting Up the HTML Code

Although tables can be used to create HTML page layouts, they are not ideal. Pages laid out with tables are often much larger in size due to the additional markup that is required. They are also less flexible, so it is harder to move sections of the layout without restructuring the markup completely.

CSS-based layouts, on the other hand, are generally smaller in file size and much more flexible.

The HTML code for this lesson is comprised of seven main containers—an `<h1>` element and six `<div>` elements as shown in Listing 21.1.

#### **LISTING 21.1** HTML Code Containing the Markup for a Three-Column Layout

---

```
<h1>
 Sitename
</h1>
<div id="container">
```

*continues*

**LISTING 21.1** Continued

---

```
<div id="container2">
 <div id="content">
 <h2>
 Page heading
 </h2>
 <p>
 Lorem ipsum dolor sit amet...
 </p>
 </div>
 <div id="news">
 <h3>
 News
 </h3>
 <p>
 Lorem ipsum dolor sit amet...
 </p>
 </div>
 <div id="nav">
 <h3>
 Sections
 </h3>

 Home
 About us
 Services
 Staff
 Portfolio
 Contact us

 </div>
 <div id="footer">
 Copyright © Sitenam 2005
 </div>
</div>
</div>
```

## Creating Selectors to Style the Three-Column Layout

To style the three-column layout and its content, you will use 10 selectors as shown in Listing 21.2.

**LISTING 21.2** CSS Code Showing the Selectors for Styling the Three-Column Layout

```
body {...}
h1 {...}
h2, h3, p {...}
#container {...}
#container2 {...}
#content {...}
#news {...}
#nav {...}
#nav ul {...}
#footer {...}
```



**What Is a Liquid Layout?** There are four main methods used to lay out web pages. They are

- **Liquid layout**—All containers on the page have their widths defined in percents. A liquid layout will resize when you resize your browser window.
- **Combination liquid and fixed layout**—Similar to liquid layouts, except one or more of the containers on the page have fixed widths.
- **Fixed-width layout**—All containers on the page have their widths defined in pixels or other fixed units. A fixed layout will not move in and out when you resize your browser window.
- **Em-driven layout**—All containers on the page have their widths defined in `ems`. The containers will scale according to the user's current browser font size. These layouts will not move when the browser window is resized.

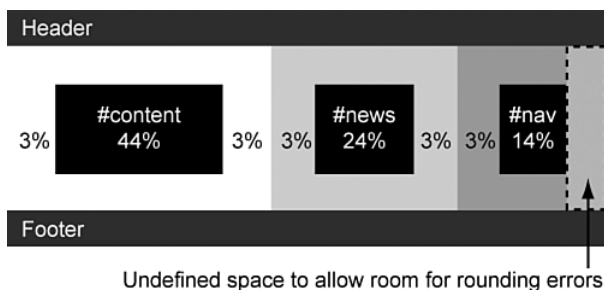
You also can use combinations of the preceding list.

## Creating a Liquid-Layout Grid

When creating a liquid layout, it is important to include vertical gutters so that the content columns do not butt up against each other. All widths should be set in percentages so that the entire page can be resized as a single unit, depending on the size of the browser window.

Percentage widths are calculated by the browser, so there will be some degree of rounding up or down of the measurements. For this reason, you should leave some undefined space so that there is room for possible rounding errors.

For this layout, the total width of the containers and their padding adds up to 100%. However, the padding on the right side of the #nav has not been included, so there is 3% of undefined space within the overall layout. The measurements are shown in Figure 21.1.



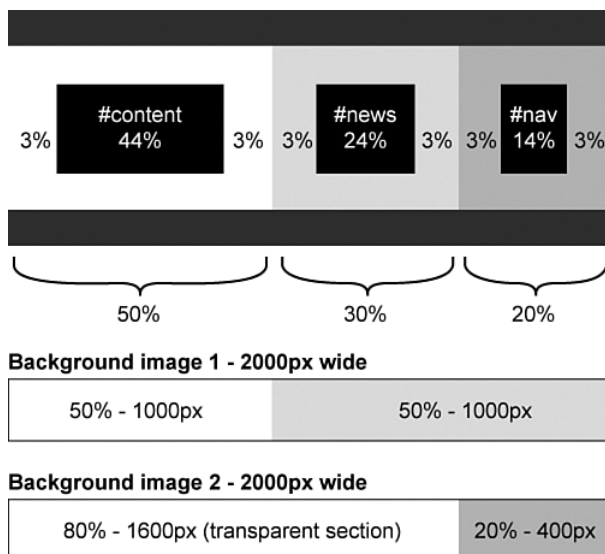
**FIGURE 21.1** Diagram of liquid-layout grid.

## Creating the Background Images

This layout uses two background images to give the illusion of column colors. The images must be wide enough to ensure they fill the largest screen. Here, they are 2000px wide.

The first image will be used to create the #content and #news column background colors.

The second image will be used to create the #nav column background color. Because this image sits over the top of the first image, it must have a transparent background as shown in Figure 21.2.



**FIGURE 21.2** Diagram showing the background images.

## Styling the <body> Element

Now that the layout grid and images are finished, you can begin coding the page.

The first step is to force the contents of the page up against the edges of the browser window. To do this, you must set both margin and padding to 0.

You can set both font-size and font-family for all elements on the page by applying these properties to the <body> element using `font: 90% arial, helvetica, sans-serif;`.



### Setting margin and padding on the <body> Element

Browsers use different methods to set their default indentation on the <body> element.

If `padding: 0` is used, Opera will set the content against the edges of the browser window.

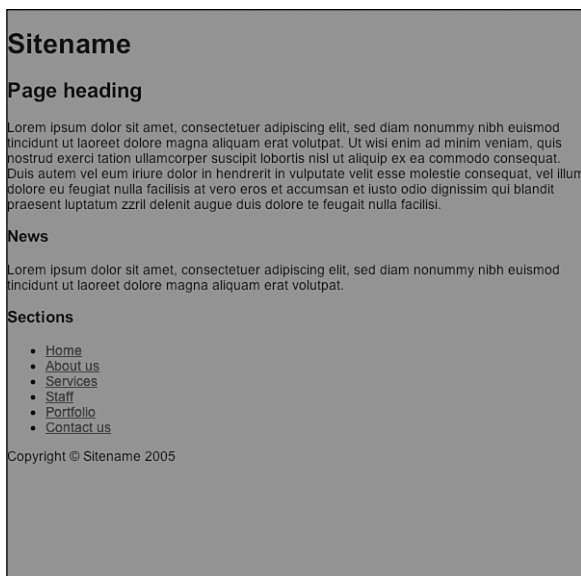
If `margin: 0` is used, all other standards-compliant browsers will set the content against the edges of the browser window.

The only way to force all browsers to work the same way is to set both margin and padding to 0.

The background-color and color properties also must be set on the <body> element. You can use a background color of #387A9B and a color of #333 as shown in Listing 21.3. The results can be seen in Figure 21.3.

### LISTING 21.3    CSS Code for Styling the <body> Element

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}
```



**FIGURE 21.3** Screenshot of the styled <body> element.

## Styling the <h1> Element

The <h1> element will be used to create the top banner.

First, the background-color and color properties must be set. In this example, you will use a background color of #D36832 and a color of #fff.

Standard <h1> elements have predefined top and bottom margins. To force the <h1> element into the top corner of the browser window, these margins must be set to 0.

To create space around the <h1> content, padding: .5em 3%; is used. This will put .5em of padding on the top and bottom of the content, and 3% on the left and right edges.

Finally, a border is applied to the bottom of the element using border-bottom: 5px solid #387A9B; as shown in Listing 21.4 and Figure 21.4.

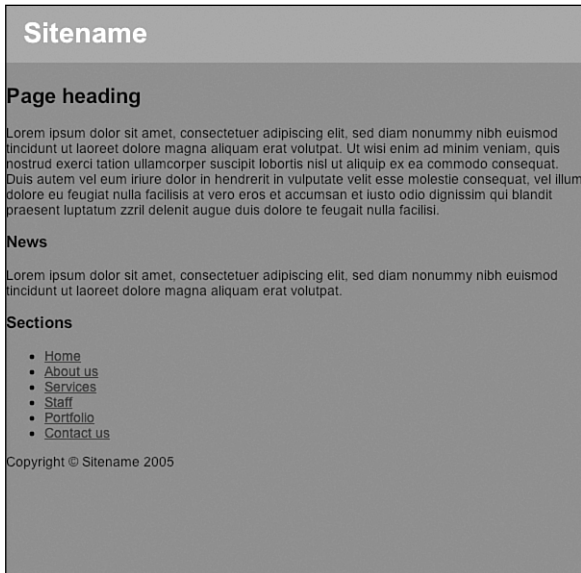


**LISTING 21.4** CSS Code for Styling the <h1> Element

---

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}

h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}
```

**FIGURE 21.4** Screenshot of the styled <h1> element.

## Styling the <h2> and <h3> Elements

The <h2> and <h3> elements sit inside the #container and #container2 elements. Some browsers will display these heading elements and their top margin inside the container. Other browsers will display the headings only and allow the margin to poke out the top of the container. This is explained in more detail in the “Trapping Margins” Note in Lesson 13, “Styling a Block Quote.” There are many ways to overcome this problem. In this lesson, the top margins on the <h2> and <h3> elements will be removed using `margin-top: 0` (see Listing 21.5 and Figure 21.5).

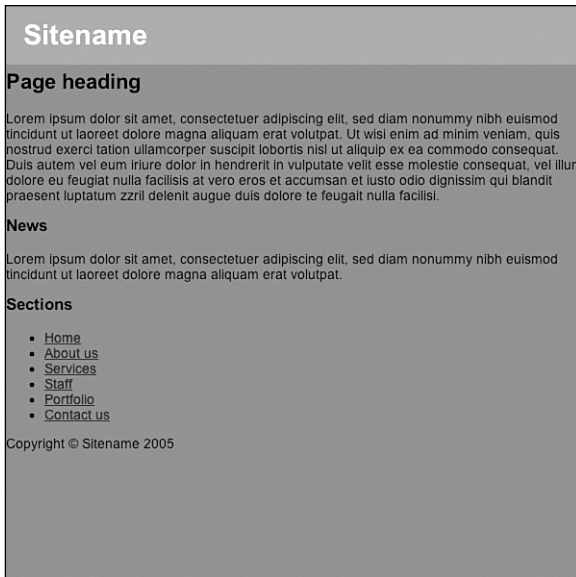
### Listing 21.5 CSS Code for Styling the <h2> and <h3> Elements

---

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}

h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}
```



**FIGURE 21.5** Screenshot of styled `<h2>` and `<h3>` elements.

## Styling the First Container

The first background image will be applied to the `#container` element using `background: url(back01.gif) repeat-y 50% 0;` as shown in Listing 21.6. This will repeat the background image—`back01.gif`—down the y axis. The image will be positioned so that 50% of the image will sit exactly 50% of the way across the background of the `#container` element (see Figure 21.6).

### LISTING 21.6 CSS Code for Styling the First Container

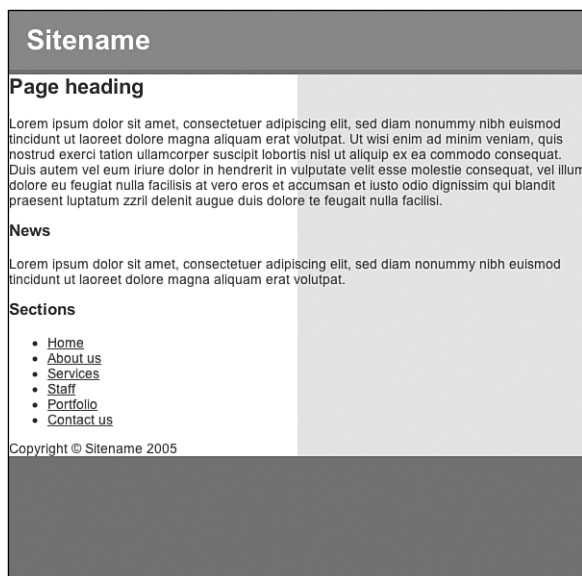
```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}
```

*continues*

```
h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

#container
{
 background: url(back01.gif) repeat-y 50% 0;
}
```



**FIGURE 21.6** Screenshot of the styled first container.

## Styling the Second Container

The second background image will be applied to the `#container2` element using `background: url(back02.gif) repeat-y 80% 0;` as shown in Listing 21.7 and Figure 21.7. Like the preceding `#container` rules, this will place 80% of the image 80% of the way across the browser window. The image is repeated down the y axis.

---

### LISTING 21.7 CSS Code for Styling the Second Container

---

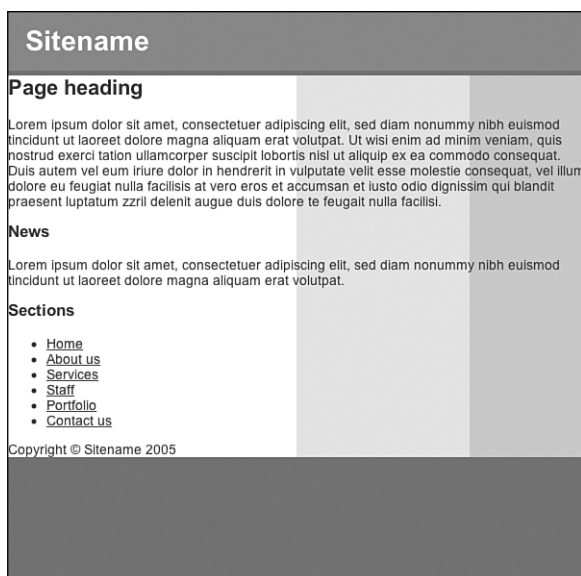
```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}

h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

#container
{
 background: url(back01.gif) repeat-y 50% 0;
}

#container2
{
 background: url(back02.gif) repeat-y 80% 0; }
}
```



**Figure 21.7** Screenshot of the styled second container.

## Styling the #content Column

The first column must be floated and set with a width of 44%. It also should be given some margin using `margin: 1em 3%;` as shown in Listing 21.8. The results can be seen in Figure 21.8. This will provide 1em of space above the container and 3% margin on either side, as a gutter between other columns.

### **LISTING 21.8** CSS Code for Styling the #content Column

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}
```

*continues*

**LISTING 21.8**    Continued

---

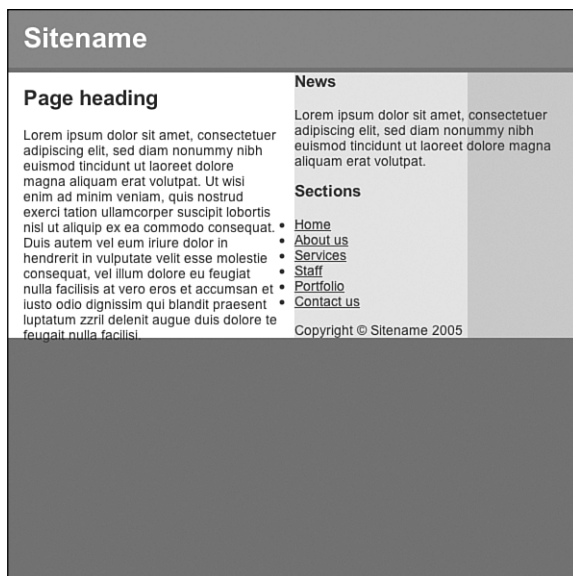
```
h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

#container
{
 background: url(back01.gif) repeat-y 50% 0;
}

#container2
{
 background: url(back02.gif) repeat-y 80% 0;
}

#content
{
 width: 44%;
 float: left;
 margin: 1em 3%;
}
```



**Figure 21.8** Screenshot of the styled #content column.

## Styling the #news Column

The second column must be floated and set with a width of 24%. Like the first column, it also should be given a margin of 1em 3%; as shown in Listing 21.9 and Figure 21.9. As with the #content column, this will provide 1em of space above the container and 3% margin on either side.

### LISTING 21.9 CSS Code for Styling the #news Column

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}
```

*continues*



**LISTING 21.9**    Continued

---

```
h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

#container
{
 background: url(back01.gif) repeat-y 50% 0;
}

#container2
{
 background: url(back02.gif) repeat-y 80% 0;
}

#content
{
 width: 44%;
 float: left;
 margin: 1em 3%;
}

#news
{
 width: 24%;
 float: left;
 margin: 1em 3%;
}
```



**FIGURE 21.9** Screenshot of the styled #news column.

# Styling the #nav Column

The third column must be floated and set with a width of 14%. Like the other columns, it also should be padded. However, the padding for this element is padding: 1em 0 1em 3%; as shown in Listing 21.10. The right-edge padding is not defined because this space is left undefined for rounding errors. The results can be seen in Figure 21.10.

## LISTING 21.10 CSS Code for Styling the #nav Column

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}
```

*continues*

**LISTING 21.10**    Continued

---

```
h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

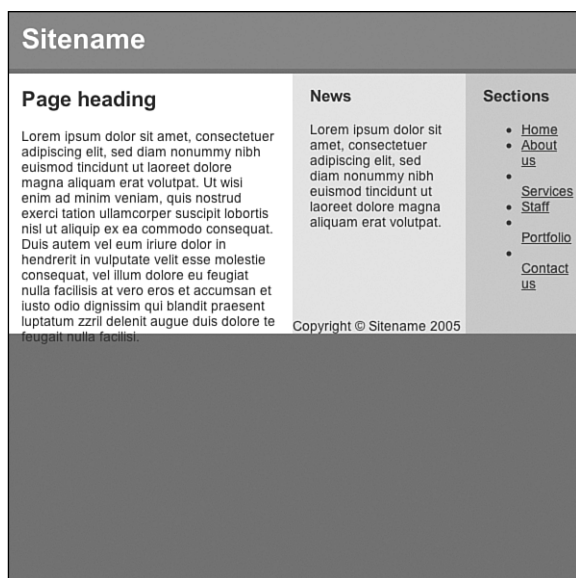
#container
{
 background: url(back01.gif) repeat-y 50% 0;
}

#container2
{
 background: url(back02.gif) repeat-y 80% 0;
}

#content
{
 width: 44%;
 float: left;
 margin: 1em 3%;
}

#news
{
 width: 24%;
 float: left;
 margin: 1em 3%;
}

#nav
{
 width: 14%;
 float: left;
 margin: 1em 0 1em 3%;
}
```



**FIGURE 21.10** Screenshot of the styled #nav column.

## Styling the <ul> Element

The third column contains a navigation list. In this lesson, the list items will line up with the edge of the container without bullets. This is achieved using three declarations—`margin: 0;`, `padding: 0;`, and `list-style-type: none;`.

To add space between each list item, the `line-height` can be increased using `line-height: 150%;` as shown in Listing 21.11 and Figure 21.11.

### LISTING 21.11 CSS Code for Styling the <ul> Element

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
```

*continues*

**LISTING 21.11** Continued

---

```
 color: #333;
 }

 h1
 {
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
 }

 h2, h3
 {
 margin-top: 0;
 }

 #container
 {
 background: url(back01.gif) repeat-y 50% 0;
 }

 #container2
 {
 background: url(back02.gif) repeat-y 80% 0;
 }

 #content
 {
 width: 44%;
 float: left;
 margin: 1em 3%;
 }

 #news
 {
 width: 24%;
 float: left;
 margin: 1em 3%;
 }

 #nav
 {
 width: 14%;
```

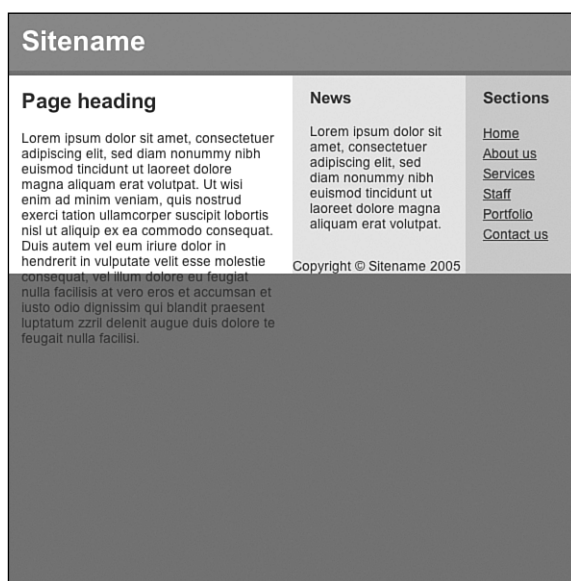
*continues*

```

float: left;
margin: 1em 0 1em 3%;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 line-height: 150%;
}

```



**FIGURE 21.11** Screenshot of the styled `<ul>` element.

## Styling the #footer Element

The `#footer` must sit under the three floated columns. This is achieved using `clear: both;`. The background-color and color properties can be applied using `background: #387A9B;` and `color: #fff;`.

Next, the `#footer` element should be padded to provide some space around the content. This is achieved using `padding: 5px 3%;`.

Finally, the footer content can be aligned to the right using `text-align: right;` as shown in Listing 21.12. The results can be seen in Figure 21.12.

---

**LISTING 21.12**    CSS Code for Styling the `#footer` Element

---

```
body
{
 margin: 0;
 padding: 0;
 font: 90% arial, helvetica, sans-serif;
 background: #387A9B;
 color: #333;
}

h1
{
 background: #D36832;
 color: #FFF;
 margin: 0;
 padding: .5em 3%;
 border-bottom: 5px solid #387A9B;
}

h2, h3
{
 margin-top: 0;
}

#container
{
 background: url(back01.gif) repeat-y 50% 0;
}

#container2
{
 background: url(back02.gif) repeat-y 80% 0;
}

#content
{
 width: 44%;
 float: left;
 margin: 1em 3%;
```

*continues*

```
}

#news
{
 width: 24%;
 float: left;
 margin: 1em 3%;
}

#nav
{
 width: 14%;
 float: left;
 margin: 1em 0 1em 3%;
}

#nav ul
{
 margin: 0;
 padding: 0;
 list-style-type: none;
 line-height: 150%;
}

#footer
{
 clear: both;
 background: #387A9B;
 color: #fff;
 padding: 5px 3%;
 text-align: right;
}
```

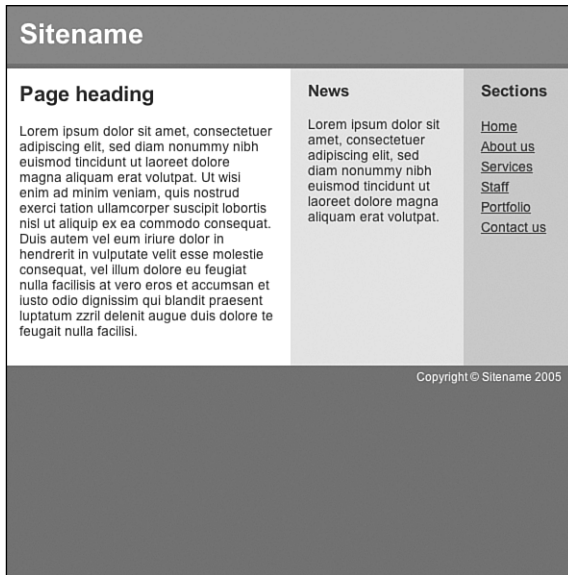


**Collapsing Liquid Layouts** Three-column liquid layouts will generally expand and contract to the width of the browser window.

When a browser window is reduced in width, one or more columns might drop below the first column on the page.

Why does this happen? If there isn't enough horizontal room on the current line for the floated column, it will move down, line by line, until there is room for it.





**FIGURE 21.12** Screenshot of the finished layout.

## Summary

In this lesson, you learned how to create a three-column liquid layout with column colors. You also learned how to style an `<h1>` element to make a banner. In the next lesson, you will learn how to fix some common CSS errors as well as learn some tips for troubleshooting CSS.

# LESSON 22



## Troubleshooting CSS

*In this lesson, you will learn how to fix some common CSS errors. You also will learn some tips for troubleshooting CSS.*

### Setting Up the CSS Code

The CSS code for this lesson is shown in Listing 22.1. The code contains 12 common CSS problems that will be corrected during the lesson.

#### **LISTING 22.1** CSS Code Showing All the Rules with 12 Common Problems

---

```
body
{
 font-family: times, times new roman, serif;
}

#container
{
 border: 1px gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attach: fixed;
 width: 700;
}
```

*continues*

**LISTING 22.1** Continued

---

```
h1
{
 font-size: 200%;
 color: none;
}

.introductionText
{
 font-weight: bold;
}

h2
{
 font-size: 120%
 font-weight: normal;
 color: #34a32;
}

p, ul,
{
 font-size: 80%;
 color: 333;
}

a:visited
{
 color: purple;
}

a:link
{
 color: blue;
}

a:hover
{
 color: red;
}

a:active
{
 color: black;
}
```

*continues*

```
#container p
{
 color: #000;
}

p.intro
{
 color: #900;
}
```

## Fixing the Problems

**Problem 1—In the body rule set, the font family name Times New Roman contains whitespace.** Any font with whitespace should be wrapped in quotation marks. Listing 22.2 shows the problem and Listing 22.3 shows the corrected code.

### LISTING 22.2 CSS Code Showing an Unquoted font-family Name

---

```
body
{
 font-family: times, times new roman, serif;
}
```

### LISTING 22.3 CSS Code Showing a Quoted font-family Name

---

```
body
{
 font-family: times, "times new roman", serif;
}
```

**Problem 2—The border property in the #container rule set does not have border-style specified.** This border will not be displayed because the default border-style is none. A border-style dotted, dashed, solid, double, grooved, ridged, inset, or outset should be specified. Listing 22.4 shows the problem and Listing 22.5 shows the corrected code.

**LISTING 22.4** CSS Code Showing an Incorrect border Declaration

---

```
#container
{
 border: 1px gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attach: fixed;
 width: 700;
}
```

**LISTING 22.5** CSS Code Showing a Correct border Declaration

---

```
#container
{
 border: 1px solid gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attach: fixed;
 width: 700;
}
```

**Problem 3—The background-attach property does not exist.** The correct property is background-attachment. Listing 22.6 shows the problem and Listing 22.7 shows the corrected code.

**LISTING 22.6** CSS Code Showing an Incorrect Property

---

```
#container
{
 border: 1px solid gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attach: fixed;
 width: 700;
}
```

**LISTING 22.7** CSS Code Showing a Correct Property

---

```
#container
{
 border: 1px solid gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attachment: fixed;
 width: 700;
}
```

**Problem 4—The width value in the #container rule set does not contain a unit of measurement.** Browsers will have to guess whether the author requires the width to be rendered in points, picas, pixels, ems, exs, millimeters, centimeters, inches, or percents. Listing 22.8 shows the problem and Listing 22.9 shows the corrected code.

**LISTING 22.8** CSS Code Showing a Width Measurement Without a Specified Unit Value

---

```
#container
{
 border: 1px solid gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attachment: fixed;
 width: 700;
}
```

**LISTING 22.9** CSS Code Showing a Width Measurement in Pixels

---

```
#container
{
 border: 1px solid gray;
 background-image: url("background.jpg");
 background-repeat: repeat-x;
 background-attachment: fixed;
 width: 700px;
}
```

**Problem 5—In the <h1> rule set, the color value is specified as none.** This is an invalid value. The value should be specified in hexadecimal RGB, keywords, user interface keywords, or decimal RGB. The author

might also prefer the element to inherit its color from the parent, in which case `inherit` should be used. Listing 22.10 shows the problem and Listing 22.11 shows the corrected code.

---

**LISTING 22.10** CSS Code Showing an Incorrect Color Value

---

```
h1
{
 font-size: 200%;
 color: none;
}
```

---

**LISTING 22.11** CSS Code Showing a Correct Color Value

---

```
h1
{
 font-size: 200%;
 color: inherit;
}
```

**Problem 6—Some authors prefer to use upper- and lowercase class names.** For the class to be applied, the uppercase and lowercase letters must be exactly the same within the HTML code and within the class selector.

Let's assume the HTML code contains an `IntroductionText` class. In this lesson, the author has specified `.introductionText`. The selector does not match the HTML classname, so the rule will not be applied. Listing 22.12 shows the problem and Listing 22.13 shows the corrected code.

---

**LISTING 22.12** CSS Code Showing an Incorrectly Spelled Classname

---

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

---

**LISTING 22.13** CSS Code Showing the Correctly Spelled Classname

---

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

**Problem 7—In the <h2> rule set, the font-size declaration is missing a semicolon.** Browsers will read the next declaration, font-weight: normal;, as part of the font-size declaration. This combined declaration is invalid, so both declarations will be ignored. Listing 22.14 shows the problem and Listing 22.15 shows the corrected code.

---

**LISTING 22.14** CSS Code Showing a Declaration with a Missing Semicolon

---

```
h2
{
 font-size: 120%
 font-weight: normal;
 color: #34a32;
}
```

---

**LISTING 22.15** CSS Code Showing the Corrected Declaration

---

```
h2
{
 font-size: 120%;
 font-weight: normal;
 color: #34a32;
}
```

**Problem 8—The hexadecimal number within the <h2> rule set is missing a digit.** Hexadecimal numbers must be three or six digits. Listing 22.16 shows the problem and Listing 22.17 shows the corrected code.

---

**LISTING 22.16** CSS Code Showing an Incorrect Hexadecimal Number

---

```
h2
{
 font-size: 120%;
 font-weight: normal;
 color: #34a32;
}
```



**LISTING 22.17** CSS Code Showing a Correct Hexadecimal Number

---

```
h2
{
 font-size: 120%;
 font-weight: normal;
 color: #34a323;
}
```

**Problem 9—There is an additional comma at the end of the multiple selectors p, ul,.** This will cause the entire rule set to be ignored. Listing 22.18 shows the problem and Listing 22.19 shows the corrected code.

**LISTING 22.18** CSS Code Showing a Comma at the End of the Selectors

---

```
p, ul,
{
 font-size: 80%;
 color: 333;
}
```

**LISTING 22.19** CSS Code Showing the Corrected Multiple Selector

---

```
p, ul
{
 font-size: 80%;
 color: 333;
}
```

**Problem 10—The color #333 is missing a # symbol for hexadecimal values.** Although some browsers will apply this incorrect declaration, others will not. Listing 22.20 shows the problem and Listing 22.21 shows the corrected code.

**LISTING 22.20** CSS Code Showing a Color Value Without a # Symbol

---

```
p, ul
{
 font-size: 80%;
 color: 333;
}
```

### LISTING 22.21 CSS Code Showing the Corrected Declaration

---

```
p, ul
{
 font-size: 80%;
 color: #333;
}
```

**Problem 11—The `a:hover` pseudo-class will not be applied because it comes before the `a:link` pseudo-class.** The order must be swapped.

Listing 22.22 shows the problem and Listing 22.23 shows the corrected code.

### LISTING 22.22 CSS Code Showing the `a:hover` Pseudo-class Before the `a:link` Pseudo-class

---

```
a:hover
{
 color: red;
}

a:link
{
 color: blue;
}
```

### LISTING 22.23 CSS Code Showing the Pseudo-classes in Correct Order

---

```
a:link
{
 color: blue;
}

a:hover
{
 color: red;
}
```

**Problem 12—The `#container p { color: black; }` rule set will style all paragraphs within the `#container` element to the black color. The next rule set, `p.intro { color: #900; }`, is designed to style the first paragraph in the `#container` element to the `#900` color.**

However, the `p.intro` rule set will not be applied because the `#container` `p` rule set has more weight. Selectors that contain IDs have more weight than selectors with classes. All paragraphs inside the `#container` will still be styled to the black color.

For the `p.intro` rule set to be applied, the selector must be changed to `#container p.intro`, which gives it more weight. Listing 22.24 shows the problem and Listing 22.25 shows the corrected code.

---

**LISTING 22.24**    CSS Code Showing a Rule Set Without ID

---

```
#container p
{
 color: #000;
}

p.intro
{
 color: #900;
}
```

---

**LISTING 22.25**    CSS Code Showing a Rule Set with ID

---

```
#container p
{
 color: #000;
}

#container p.intro
{
 color: #900;
}
```

## Some Tips for Troubleshooting CSS Problems

**Tip 1—Make sure you validate your HTML and CSS files.** Seven of the twelve problems listed previously would be immediately picked up by the CSS validator.

You can find the W3C HTML validator at <http://validator.w3.org/>. The W3C CSS validator is at <http://jigsaw.w3.org/css-validator/>.

**Tip 2—The best way to avoid problems, especially when you are new to CSS, is to build your layouts in stages and test each stage across a range of browsers.** Start with the overall framework, position these elements, and test across browsers. When you feel confident that the framework is stable, you can start styling more detailed elements.

**Tip 3—If there is a specific problem on a page, it often helps to turn on borders so that you can identify the elements and see how they interact.** An example of turning on borders is shown in Listing 22.26.

**LISTING 22.26** CSS Code Showing a Rule Set to Turn On Borders

---

```
li a { border: 1px solid red; }
```

**Tip 4—A quick technique for finding major errors in the CSS is to comment out one rule set at a time (as shown in Listing 22.27) and observe the results.** When you have found the offending rule sets, you can begin commenting out declarations within these sets to find the culprit.

**LISTING 22.27** CSS Code Showing a Commented-Out Rule Set

---

```
h2
{
 font-size: 120%;
 font-weight: normal;
 color: #34a323;
}

/*
p, ul,
{
 font-size: 80%;
 color: #333;
}
*/
```

**Tip 5—Whenever possible, use a full and complete doctype at the top of your (X)HTML document.** All (X)HTML documents must have a doctype declaration to be valid. The doctype states the version of (X)HTML being used in the document. Web browsers use doctypes to

determine which rendering mode to use. If a correct and full doctype is present in a document, many web browsers will switch to Standards mode, which means they will follow the CSS specification more closely.

The main doctypes are shown in Listings 22.28 to 22.32.

---

**LISTING 22.28** HTML Code Showing HTML 4.01 Strict Doctype

---

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

---

**LISTING 22.29** HTML Code Showing HTML 4.01 Transitional Doctype

---

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

---

**LISTING 22.30** HTML Code Showing XHTML 1.0 Strict Doctype

---

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

---

**LISTING 22.31** HTML Code Showing XHTML 1.0 Transitional Doctype

---

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

---

**LISTING 22.32** HTML Code Showing XHTML 1.1 Doctype

---

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

## Summary

In this lesson, you learned how to fix some common CSS errors, as well as some tips for troubleshooting CSS.

# INDEX



## A - B

---

- `<a>` element, 74. *See also* links
  - headers, styling for, 162
  - printing, styling for, 199
  - round-cornered boxes, creating, 146-147
  - styling, 122-126, 133-137
  - two-column layouts, styling for, 184
- a:, 71. *See also* pseudo-classes
- abbr attribute (data tables), 110
- active links, 68-69. *See also* links
- adjacent sibling selectors, 25
- attribute selectors, 26
- author style sheets, 4
- background images
  - headings, adding to, 62
  - links, styling, 71-73
  - liquid layouts, 204-205
  - printing, 192
  - scalable images, 64-65
- background property shortcuts, 48
- background-positions, adding, 46-47
- background-repeat, setting, 45
- backgrounds
  - `<body>` element, adding to, 44
  - boxes, 39
  - Macintosh Internet Explorer, 44
  - shorthand properties, 18
- block level element, 35-36. *See also* boxes
- block quotes
  - paragraphs, styling, 101-102
  - source class, styling, 102-104
  - trapping margins, 105
  - variations, creating, 104-106
- `<blockquote>` element
  - applying, 98-99
  - styling, 100-101
- `<body>` element
  - background images, adding, 44
  - background shortcuts, utilizing, 48
  - background-position, adding, 46-47
  - background-repeat, 45
  - headers, styling for, 153-154
  - liquid layouts, styling for, 206-207
  - padding, 48
  - printing, styling for, 193

- selectors for styling, 44
- two-column layouts, styling
  - for, 167
- bold styles, applying, 54
- border-style, specifying, 227

## borders

- boxes, 40
- headings, applying to, 61
- images, styling, 82
- links, applying, 73-74
- lists, fixing, 127
- shorthand properties, 14-15
- turning on, 234

## boxes

- backgrounds, 39
- block level element, 35-36
- borders, 40
- content area, formatting, 40-41
- inline level element, 35-36
- margins, 37-38
- padding, 39-40
- round-cornered boxes
  - <a> element*, 146-147
  - <div> element*, 140-141
  - fixed-widths*, 147-149
  - furtherinfo class*, 144
  - <h2> element*, 142-143
  - overview*, 139-140
  - <p> element*, 143
  - selectors*, 140
  - top-only flexible variation*, 149-151
- width, setting, 36-37

- browser style sheets, 3-4

- browsers, hiding styles, 32-34

# C

---

- caption attribute (data tables), 109

## captions

- containers
  - floating*, 78
  - styling*, 79-80

- images, displaying side-by-side, 83

- styling, 81

- tables, styling, 112

- wrapping, 77

- cascading, 4

- case-sensitive class names, reconciling, 230

- cellspacing, removing from tables, 113

- child selectors, 25

- cite attribute, 98

- class selectors, 22-23

## colors

- background boxes, 39
- column colors, creating, 168
- h1 rule set, fixing, 229
- <h2> element*, adding to, 64
- headings, adding to, 58
- hexadecimal colors, headings, 60
- table rows, alternating, 117-118
- text, styling, 41

- comments (CSS), adding, 11

- complex absolute positioning, 192

## containers

- border-style, specifying, 227
- headers, styling for, 154
- images, 78-80
- liquid layouts, styling for, 210-212
- long floated containers, 192
- two-column layouts, 168
- width, fixing, 229

## contents

- boxes, formatting area, 40-41
- liquid layouts, styling for, 213
- two-column layouts, styling for, 177

---

**D - E - F**

---

data table accessibility features, 108-111

declarations

- description of, 7
- multiple declarations, utilizing, 9

descendant selectors, 23-24

`<div>` element

- description of, 21
- lines, moving down, 93
- positioning, 88
- round-cornered boxes, creating, 140-141

doctype, 235

Double Margin Float Bug, 79

em-driven layouts, 203

`<em>` element, 66

ems, 195

- advantages of, 125
- versus percents, 53

external style sheets, 30-31

fixed-width layouts, 90, 203

flexible headings, 57

focus links, 68. *See also* links

`<font>` element, removing, 51-52

font families, setting, 53

fonts

- bold styles, applying, 54
- headings, styling, 58-60
- italic styles, applying, 54
- whitespace, fixing, 227

footers

- liquid layouts, styling for, 221
- printing, styling for, 197
- two-column layouts, styling for, 179-180

.furtherinfo class, 144

---

**G - H**

---

generic font families, 53

`<h1>` element

- headers, styling for, 156
- liquid layouts, styling for, 207
- printing, styling for, 194
- two-column layouts, styling for, 169

h1 rule set color values, fixing, 229

`<h2>` element, 142-143

- background images, adding, 65
- colors, adding, 64
- liquid layouts, styling for, 209
- targeting, 63
- two-column layouts, styling for, 182

`<h3>` element, 209

headings, 44

- `<a>` element, styling, 162
- background images, adding, 62
- `<body>` element, styling, 153-154
- borders, applying, 61
- color, adding, 58
- containers, styling, 154
- data table headers, 110
- flexible headings, 57
- fonts, styling, 58-60
- `<h1>` element, styling, 156
- hexadecimal colors, utilizing, 60
- `<image>` element, styling, 157
- @import styles, 32
- `<li>` element, styling, 160
- padding, adding, 60
- round-cornered headings, 66. *See also* `<h2>` element



- standard settings, overriding, 59
- styles, applying, 29-30
- text options, setting, 60
- `<ul>` element, styling, 158-159
- hexadecimal
  - colors
    - headings, adding to, 60*
    - specifying, 41*
  - numbers, 231
- horizontal navigation
  - `<a>` element, styling, 133-137
  - `:hover` pseudo-class, styling, 137
  - HTML lists, styling, 130
  - `<li>` element, styling, 132
  - `<ul>` element, styling, 131-132
- hover effects, 68, 127. *See also* links
- hover pseudo-class, styling, 137

## I

---

- icons, 71-73
- ID selectors, 22-23
- ids (data tables), 110
- `<image>` element, 157
- images
  - backgrounds
    - background-repeat, 45*
    - <body> element, adding to, 44*
    - boxes, 39*
    - <h2> element, 65*
    - headings, adding to, 62*
    - links, styling, 71-73*
    - liquid layouts, 204-205*
    - Macintosh Internet Explorer 5, 44*
    - printing, 192*
    - scalable images, 64-65*

- borders, styling, 82
- captions, displaying
  - side-by side, 83
- containers, 78-80
- photo frame variations,
  - creating, 84-85
- scalable background images,
  - creating, 64
  - wrapping, 77
- `@import` rule, 32-34
- inline level element, 35-36. *See also* boxes
- inline styles, applying, 29
- italic styles, applying, 54

## J - K - L

---

- layouts
  - em-driven layouts, 203
  - fixed-width layouts, 90, 203
  - liquid layouts
    - background images, 204-205*
    - <body> element, styling, 206-207*
    - collapsing, 223*
    - containers, styling, 210-212*
    - content columns, styling, 213*
    - creating, 204*
    - description of, 90, 203*
    - footers, styling, 221*
    - <h1> element, styling, 207*
    - <h2> element, styling, 209*
    - <h3> element, styling, 209*
    - nav, styling, 217*
    - news columns, styling, 215*
    - <ul> element, styling, 219*

<li> element  
   headers, styling for, 160  
   styling, 128-132  
   two-column layouts, styling for, 175

line-height, setting, 53

links  
   active areas, increasing, 74-75  
   a:active versus a:focus, 69  
   borders, applying, 73-74  
   pseudo-classes  
     *class selector*  
       *combinations*, 70  
       *order, setting*, 69  
       *types of*, 68  
   states of, 68  
   styling, with background images, 71-73  
   underlines, removing, 73-74

liquid layouts  
   background images, 204-205  
   <body> element, styling, 206-207  
   collapsing, 223  
   containers, styling, 210-212  
   content columns, styling, 213  
   creating, 204  
   description of, 90, 203  
   footers, styling, 221  
   <h1> element, styling, 207  
   <h2> element, styling, 209  
   <h3> element, styling, 209  
   nav, styling, 217  
   news columns, styling, 215  
   <ul> element, styling, 219

list-style shorthand properties, 18

lists  
   <a> element, styling, 122-126  
   borders, fixing, 127  
   hover effects, adding, 127  
   <li> element, styling, 128-129  
   purpose of, 120

  styling, 121  
   <ul> element, styling, 121  
 long floated containers, 192

## M - N - O

---

Macintosh Internet Explorer  
   background images, 44

margins  
   <body> element, setting, 206  
   boxes, 37-38  
   shorthand properties, 16-18  
   trapping, 105

media attribute  
   media types, 190  
   styling for print, 188-189

nav container  
   liquid layouts, styling for, 217  
   printing, styling for, 196  
   two-column layouts, styling for, 171-172

Netscape Navigator 4, hiding styles, 32-34

news columns, styling, 215

none background images, 44

normal links, 68. *See also* links

## P

---

<p> element  
   header styles, applying, 29-30  
   fonts, styling, 53  
   round-cornered boxes, creating, 143

padding  
   block quotes, 101  
   <body> element, 48, 206  
   boxes, 39-40  
   headings, adding to, 60  
   shorthand properties, 16-18

paragraphs  
 selectors, styling for, 52  
 thumbnail galleries, styling, 91  
 trapping margins, 105  
 percents versus ems, 53  
 photo frame variations, creating, 84-85  
 photo galleries. *See* thumbnail galleries  
 printing  
   <a> element, styling for, 199  
   background images, 192  
   <body> element, styling for, 193  
   complex absolute positioning, 192  
   CSS, setting up, 188-190  
   footer container, styling for, 197  
   <h1> element, styling for, 194  
   hiding content from, 196  
   long floated containers, 192  
   nav container, styling for, 196  
 properties (rule sets)  
   description of, 8  
   shorthand properties  
     *background*, 18  
     *borders*, 14-15  
     *list-style*, 18  
     *margins/padding*, 16-18  
     *utilizing*, 11-13  
 pseudo-classes. *See also* individual pseudo-classes  
   links  
     *class selector combinations*, 70  
     *order, setting*, 69  
     *styling with*, 71-73  
     *types of*, 68  
   organizing, 233  
   selectors, 26-27

## Q - R

---

quotes. *See* block quotes  
 repeating images, 45  
 round-cornered boxes  
   <a> element, 146-147  
   <div> element, 140-141  
   fixed-widths, 147-149  
   .furtherinfo class, 144  
   <h2> element, 142-143  
   overview, 139-140  
   <p> element, 143  
   selectors, 140  
   top-only flexible variation, 149-151  
 round-cornered headings, 66.  
   *See also* <h2> element  
 rule sets  
   background shorthand properties, 18  
   border shorthand properties, 14-15  
   creating, 7-9  
   declarations  
     *description of*, 7  
     *multiple declarations, utilizing*, 9  
   definition of, 6  
   list-style shorthand properties, 18  
   margins/padding shorthand properties, 16-18  
   selectors. *See also* selectors  
     *combining*, 10  
     *description of*, 7  
   values, 8  
   whitespace, utilizing, 8

## S

---

scalable background images, creating, 64

selectors. *See also individual selectors*

- adjacent sibling selectors, 25
- attribute selectors, 26
- body styling, 44
- child selectors, 25
- class selector/pseudo-class combinations, 70
- class selectors, 22-23
- combining, 10
- descendant selectors, 23-24
- description of, 7, 121
- ID selectors, 22-23
- paragraphs, styling, 52
- prioritizing, 233
- pseudo-class selectors, 27
- pseudo-element selectors, 26
- type selectors, 21
- universal selectors, 24

shorthand properties

- backgrounds, 18
- borders, 14-15
- list-style, 18
- margins/padding, 16-18
- utilizing, 11-13

side-by-side variations (thumbnail galleries), creating, 95

Skip to content links, 153

source class block quotes, styling, 102-104

style sheets, 3-4

styles

- external style sheets, applying with, 30-31
- header, applying, 29-30
- hiding from old browsers, 32-34

@import rule, utilizing, 32

inline, applying, 29

summary attribute (data tables), 108

## T

---

<table> element, styling, 113

tables

- basic table example, 107-108
- captions, styling, 112
- cellspacing, removing, 113
- data table accessibility features, 108-111
- row colors, alternating, 117-118
- selectors for styling, 111
- <td> element, styling, 113-114
- <th> element
  - styling, 113-114
  - targeting, 115-116
- <tr> element, styling, 114-115

<tbody> element, 109

<td> element, styling, 113-114

text

- bold styles, applying, 54
- color property, 41
- headings, styling, 58-60
- italic styles, applying, 54
- whitespace, fixing, 227

<tfoot> element, 109

<th> element

- purpose of, 109
- styling, 113-114
- targeting, 115-116

<thead> element, 109

thumbnail galleries

- creating, 87
- <div> element, positioning, 88
- images, styling, 90

- paragraph element, styling, 91
- side-by-side variations, creating, 95
- thumbnail galleries, creating, 87
- top-only flexible variation round-cornered box, 149-151
- `<tr>` element
  - styling, 114-115
  - table row colors, alternating, 117
- trapping margins, 105
- two-column layouts
  - `<a>` element, styling, 184
  - `<body>` element, styling, 167
  - column colors, creating, 168
  - containers, styling, 168
  - content containers, styling, 177
  - footer containers, styling, 179-180
  - `<h1>` element, styling, 169
  - `<h2>` element, styling, 182
  - `<li>` element, styling, 175
  - nav containers, styling, 171-172
  - `<ul>` element, styling, 173
- type selectors, 21

## U - V

---

- `<ul>` element
  - headers, styling for, 158-159
  - liquid layouts, styling for, 219
  - styling, 121, 131-132
  - two-column layouts, styling for, 173
- underlined links, 73-74
- universal selectors, 24
- url background images, 44
- user style sheets, 3-4

- values (rule sets), 8
- vertical navigation, 120. *See also* lists
- visited links, 68. *See also* links

## W - X - Y - Z

---

- W3C, 4
- whitespace
  - fixing, 227
  - utilizing in rule sets, 8
- widths
  - boxes, setting, 36-37
  - container, fixing, 229
  - lists, 127
  - round-cornered boxes, 147-149
- wrapping images and captions, 77
- XHTML doctype, 235