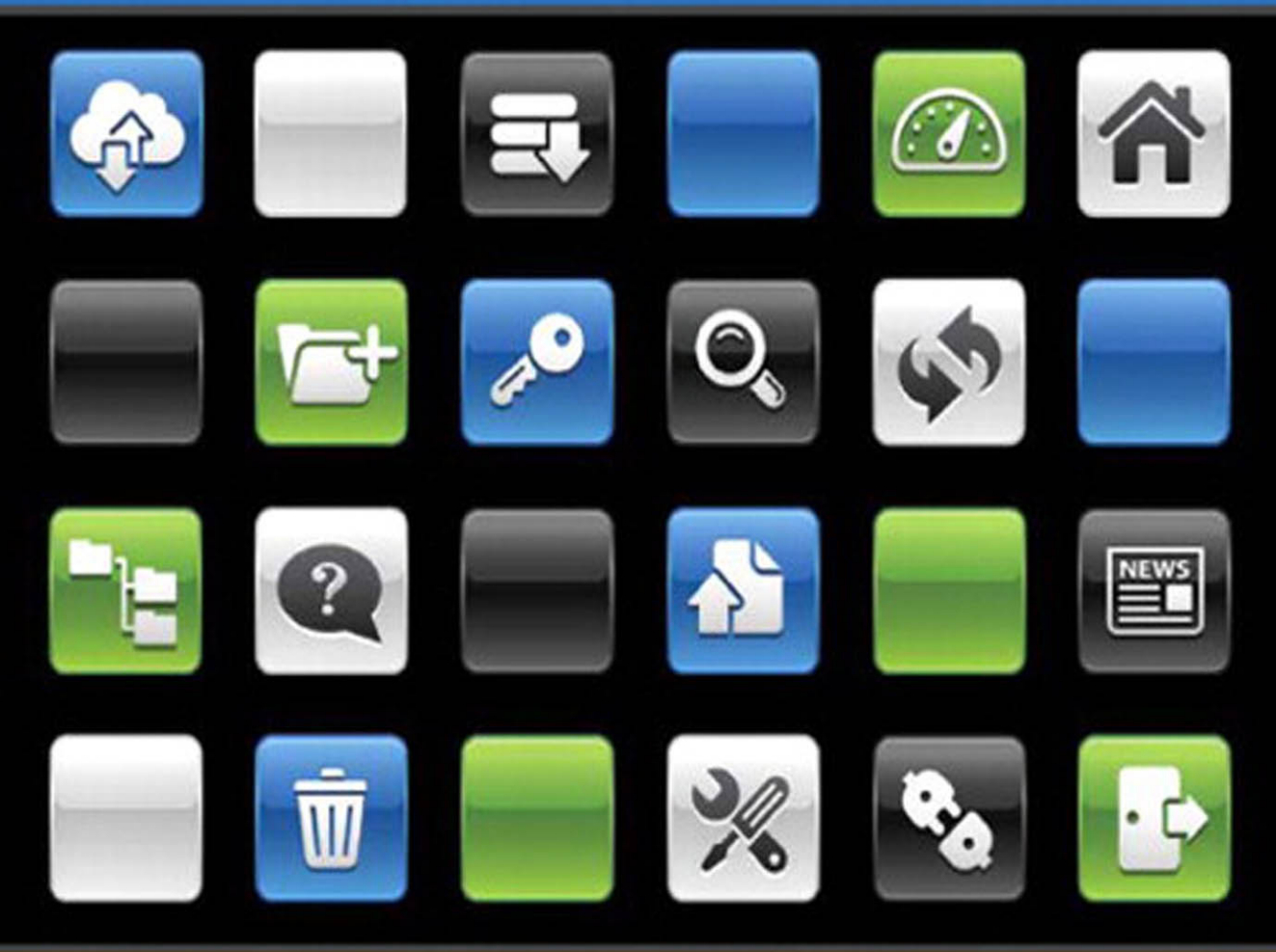
PROGRAMMING THE World Wide Web

EIGHTH EDITION



ROBERT W. SEBESTA

PROGRAMMING THE

WORLD WIDE WEB

EIGHTH EDITION

ROBERT W. SEBESTA

University of Colorado at Colorado Springs

PEARSON

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Vice President/Editorial Director: Marcia Horton
Executive Editor: Matt Goldstein
Editorial Assistant: Kelsey Loanes
Senior Managing Editor: Scott Disanno
Program Manager: Kayla Smith-Tarbox
Project Manager: Irwin Zucker
Art Director: Jayne Conte
Cover Designer: Bruce Kenselaar
Cover Art: © Palsur/Shutterstock
Full-Service Project Management: Vasundhara Sawhney/Cenveo® Publisher Services
Composition: Cenveo Publisher Services
Printer/Binder: R.R. Donnelley—Harrisonburg
Cover Printer: R.R. Donnelley—Harrisonburg
Text Font: JansonText

Trademarks

Credits for illustrations appear on page xx.

Library of Congress Cataloging-in-Publication Data
Sebesta, Robert W., author.
Programming the World Wide Web / Robert W. Sebesta, University of Colorado at Colorado Springs. -- Eighth edition.
pages cm
Includes index.
ISBN 978-0-13-377598-3 (alk. paper)
1. Internet programming. 2. World Wide Web. I. Title.
QA76.625.S42 2014
006.7'6--dc 3

2014000161

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN-10: 0-13-377598-4
ISBN-13: 978-0-13-377598-3

To Aidan

This page intentionally left blank

Preface

It is difficult to overestimate the effect the World Wide Web has had on the day-to-day lives of people, at least those in the developed countries. In just 20 years, we have learned to use the Web for a myriad of disparate tasks, ranging from the mundane task of shopping for airline tickets to the crucial early-morning gathering of business news for a high-stakes day trader.

The speed at which millions of Web sites appeared in the last two decades would seem to indicate that the technologies used to build them were sitting on the shelf, fully developed and ready to use, even before the Web appeared. Also, one might guess that the tens of thousands of people who built those sites were sitting around unemployed, waiting for an opportunity and already possessing the knowledge and abilities required to carry out this mammoth construction task when it appeared. Neither of these was true. The need for new technologies was quickly filled by a large number of entrepreneurs, some at existing companies and some who started new companies. A large part of the programmer need was filled, at least to the extent to which it was filled, by new programmers, some straight from high school. Many, however, were previously employed by other sectors of the software development industry. All of them had to learn to use new languages and technologies.

A visit to a bookstore, either a bricks-and-mortar store or a Web site, will turn up a variety of books on Web technologies aimed at the practicing professional. One difficulty encountered by those teaching courses in Web programming technologies in colleges is the lack of textbooks that are targeted to their needs. Most of the books that discuss Web programming were written for professionals, rather than college students. Such books are written to fulfill the needs of professionals, which are quite different from those of college students. One major difference between an academic book and a professional book lies in the assumptions made by the author about the prior knowledge and experience of the audience. On the one hand, the backgrounds of professionals vary widely, making it difficult to assume much of anything. On the other hand, a book written for junior computer science majors can make some definite assumptions about the background of the reader.

This book is aimed at college students, not necessarily only computer science majors, but anyone who has taken at least two courses in programming. Although students are the primary target, the book is also useful for professional programmers who wish to learn Web programming.

The goal of the book is to provide the reader with a comprehensive introduction to the programming tools and skills required to build and maintain server sites on the Web. A wide variety of technologies are used in the construction of a Web site. There are now many books available for professionals that focus on these technologies. For example, there are dozens of books that specifically address only HTML. The same is true for at least a half-dozen other Web technologies. This book provides descriptions of many of the most widely used Web technologies, as well as an overview of how the Web works.

The first seven editions of the book were used to teach a junior-level Web programming course at the University of Colorado at Colorado Springs. The challenge for students in the course is to learn to use several different programming languages and technologies in one semester. A heavy load of programming exercises is essential to the success of the course. Students in the course build a basic, static Web site, using only HTML as the first assignment. Throughout the remainder of the semester, they add features to their site as the new technologies are introduced in the course. Our students' prior course work in Java and data structures, as well as C and assembly language, is helpful, as is the fact that many of them have learned some HTML on their own before taking the course.

The most important prerequisite to the material of this book is a solid background in programming in some language that supports object-oriented programming. It is helpful to have some knowledge of a second programming language and a bit of UNIX, particularly if a UNIX-based Web server is used for the course. Familiarity with a second language makes learning the new languages easier.

New to the Eighth Edition

- **Chapter 2** Added descriptions of three new type attribute values for the `input` element, `url`, email, and `range` to Section 2.9.2.
- **Chapter 3** Added descriptions of four new selectors, `first-child`, `last-child`, `only-child`, and `empty`, to Section 3.4.5.
- **Chapter 5** Expanded Section 5.9, titled **The `canvas` Element**, from thirteen lines to three and one-half pages, adding three new figures.
- **Chapter 7** Added the new section, 7.2, titled **Uses of XML**, which briefly describes some of the many areas in which XML has been used. Deleted Section 7.4, titled **Document Type Definitions**, in its entirety.
- **Chapter 12** Added Section 12.2.7, titled **Attributes**.
- **Chapter 14** Added a completely new chapter, now Chapter 14, titled **Android Software Development**.

Table of Contents

Chapter 1 lays the groundwork for the rest of the book. A few fundamentals are introduced, including the history and nature of the Internet, the World Wide Web, browsers, servers, URLs, MIME types, and HTTP. Also included in Chapter 1 are brief overviews of the most important topics of the rest of the book.

Chapter 2 provides an introduction to HTML, including images, links, lists, tables, forms, the audio and video elements, the organizational elements, and the time element. Small examples are used to illustrate many of the HTML elements that are discussed in this chapter.

The topic of Chapter 3 is cascading style sheets, which provide the standard way of imposing style on the content specified in HTML tags. Because of the size and complexity of the topic, the chapter does not cover all of the aspects of style sheets. The topics discussed are levels of style sheets, style specification formats, selector formats, property values, and color. Among the properties covered are those for fonts, lists, and margins. Small examples are used to illustrate the subjects that are discussed.

Chapter 4 introduces the core of JavaScript, a powerful language that could be used for a variety of different applications. Our interest, of course, is its use in Web programming. Although JavaScript has become a large and complex language, we use the student's knowledge of programming in other languages to leverage the discussion, thereby providing a useful introduction to the language in a manageably small number of pages. Topics covered are the object model of JavaScript, its control statements, objects, arrays, functions, constructors, and pattern matching.

Chapter 5 discusses some of the features of JavaScript that are related to HTML documents. Included is the use of the basic and DOM 2 event and event-handling model, which can be used in conjunction with some of the elements of HTML documents. The HTML `canvas` element also is described.

One of the interesting applications of JavaScript is building dynamic HTML documents with the Document Object Model (DOM). Chapter 6 provides descriptions of a collection of some of the changes that can be made to documents with the use of JavaScript and the DOM. Included are positioning elements; moving elements; changing the visibility of elements; changing the color, style, and size of text; changing the content of tags; changing the stacking order of overlapped elements; moving elements slowly; and dragging and dropping elements.

Chapter 7 presents an introduction to XML, which provides the means to design topic-specific markup languages that can be shared among users with common interests. Included are the syntax and document structure used by XML, namespaces, XML schemas, and the display of XML documents with both cascading style sheets and XML transformations. Also included is an introduction to Web services and XML processors.

Chapter 8 introduces the Flash authoring environment, which is used to create a wide variety of visual and audio presentations—in particular, those that include animation. A series of examples is used to illustrate the development processes, including drawing figures, creating text, using color, creating motion

and shape animations, adding sound tracks to presentations, and designing components that allow the user to control the Flash movie.

Chapter 9 introduces PHP, a server-side scripting language that enjoys wide popularity, especially as a database access language for Web applications. The basics of the language are discussed, as well as the use of cookies and session tracking. The use of PHP as a Web database access language is covered in Chapter 13.

Chapter 10 introduces Ajax, the relatively recent technology that is used to build Web applications with extensive user interactions that are more efficient than those same applications if they do not use Ajax. In addition to a thorough introduction to the concept and implementation of Ajax interactions, the chapter includes discussions of return document forms, Ajax toolkits, and Ajax security. Several examples are used to illustrate approaches to using Ajax.

Java Web software is discussed in Chapter 11. The chapter introduces the mechanisms for building Java servlets and gives several examples of how servlets can be used to present interactive Web documents. The NetBeans framework is introduced and used throughout the chapter. Support for cookies in servlets is presented and illustrated with an example. Then JSP is introduced through a series of examples, including the use of code-behind files. This discussion is followed by an examination of JavaBeans and JavaServer Faces, along with examples to illustrate their use.

Chapter 12 is an introduction to ASP.NET, although it begins with a brief introduction to the .NET Framework and C#. ASP.NET Web controls and some of the events they can raise and how those events can be handled are among the topics discussed in this chapter. ASP.NET AJAX is also discussed. Finally, constructing Web services with ASP.NET is introduced. Visual Studio is introduced and used to develop all ASP.NET examples.

Chapter 13 provides an introduction to database access through the Web. This chapter includes a brief discussion of the nature of relational databases, architectures for database access, the structured query language (SQL), and the free database system MySQL. Then, three approaches to Web access to databases are discussed: using PHP, using Java JDBC, and using ASP.NET. All three are illustrated with complete examples. All of the program examples in the chapter use MySQL.

Chapter 14 introduces the development of Android applications. The basics of view documents, which are written in an XML-based markup language, and activities, which are written in a form of Java, are introduced. Several relatively simple examples are used to illustrate this new approach to building Web applications for mobile devices.

Chapter 15 introduces the Ruby programming language. Included are the scalar types and their operations, control statements, arrays, hashes, methods, classes, code blocks and iterators, and pattern matching. There is, of course, much more to Ruby, but the chapter includes sufficient material to allow the student to use Ruby for building simple programs and Rails applications.

Chapter 16 introduces the Rails framework, designed to make the construction of Web applications relatively quick and easy. Covered are simple document requests, both static and dynamic, and applications that use databases, including the use of scaffolding.

Appendix A introduces Java to those who have experience with C++ and object-oriented programming, but who do not know Java. Such students can learn enough of the language from this appendix to allow them to understand the Java applets, servlets, JSP, and JDBC that appear in this book.

Appendix B is a list of 140 named colors, along with their hexadecimal codings.

Support Materials

Supplements for the book are available at the Pearson Web site www.pearsonhighered.com/sebesta. Support materials available to all readers of this book include

- A set of lecture notes in the form of PowerPoint files. The notes were developed to be the basis for class lectures on the book material.
- Source code for examples

Additional support material, including solutions to selected exercises and figures from the book, are available only to instructors adopting this textbook for classroom use. Contact your school's Pearson Education representative for information on obtaining access to this material, or visit pearsonhighered.com.

Software Availability

Most of the software systems described in this book are available free to students. These systems include browsers that provide interpreters for JavaScript and parsers for XML. Also, PHP, Ruby, and Java language processors, the Rails framework, the Java class libraries to support servlets, the Java JDBC, and the Android Development system, are available and free. ASP.NET is supported by the .NET software available from Microsoft. The Visual Web Developer 2013, a noncommercial version of Visual Studio, is available free from Microsoft. A free 30-day trial version of the Flash development environment is available from Adobe.

Differences between the Seventh Edition and the Eighth Edition

The eighth edition of this book differs from the seventh in the following ways:

Descriptions of the `url`, `email`, and `range` attributes of the `input` element were added to Chapter 2.

Descriptions of four new selectors, `first-child`, `last-child`, `only-child`, and `empty`, were added to Chapter 3.

The description of the `canvas` element was increased from a paragraph to three and one-half pages and three new figures were added to Chapter 5.

A new section was added to Chapter 7, titled **Uses of XML**, which briefly describes some of the many areas in which XML has been used. Section 7.4, titled **Document Type Definitions**, was deleted in its entirety.

A new section, titled **Attributes**, which describes the attributes of C# was added to Chapter 12.

A completely new chapter was added to the book, Chapter 14, titled **Android Software Development**, which introduces the structure of Android applications and the process of developing them. The use of intents to call other activities and data persistence are also discussed.

Throughout the book, numerous small revisions, additions, and deletions were made to improve the correctness and clarity of the material.

Acknowledgments

The quality of this book was significantly improved as a result of the extensive suggestions, corrections, and comments provided by its reviewers. It was reviewed by the following individuals:

Lynn Beighley

R. Blank
CTO, Almer/Blank; Training Director, The Rich Media Institute; Faculty, USC Viterbi School of Engineering

Stephen Brinton
Gordon College

David Brown
Pellissippi State Technical Community College

Barry Burd
Drew University

William Cantor
Pennsylvania State University

Dunren Che
Southern Illinois University Carbondale

Brian Chess
Fortify Software

Randy Connolly
Mount Royal University

Mark DeLuca
Pennsylvania State University

Sanjay Dhamankar
President, OMNIMA Systems, Inc.

Marty Hall

Peter S. Kimble
University of Illinois

Mark Llewellyn
University of Central Florida

Chris Love
ProfessionalASPNET.com

Gabriele Meiselwitz
Towson University

Eugene A. "Mojo" Modjeski
Rose State College

Najib Nadi
Villanova University

Russ Olsen

Jamel Schiller
University of Wisconsin—Green Bay

Stephanie Smullen
University of Tennessee at Chattanooga

Marjan Trutschl
Louisiana State University—Shreveport

J. Reuben Wetherbee
University of Pennsylvania

Christopher C. Whitehead
Columbus State University

Matt Goldstein, Executive Editor; Kelsey Loanes, Editorial Assistant, and Kayla Smith-Tarbox, Program Manager, all deserve my gratitude for their encouragement and help in completing the manuscript.

Brief Contents

Contents

Credits

Figures 2.10, 2.14, and 3.7 Courtesy of Robert W. Sebesta
Figures 2.11 and 2.12 © Total Validator
Figures 3.12, 6.9, and 6.12 © Colin Underhill / Alamy
Figures 4.13, 5.1, 12.5–12.8, 12.21–12.23 © Microsoft Corporation
Figures 4.14 and 5.2 © Mozilla
Figure 4.15 © Google, Inc.
Figure 6.10 © Chris Mattison / Alamy
Figure 6.11 © Charles Polidano / Touch The Skies / Alamy
Figures 8.1–8.5, 8.8, 8.10, 8.13, 8.14, 8.16, 8.19, 8.20 8.23, 8.26–8.33 © Adobe Systems Inc. All rights reserved. Adobe and Flash is/are either [a] registered trademark[s] or a trademark[s] of Adobe Systems Incorporated in the United States and/or other countries.
Figures 11.4–11.9, 11.1711.18 © Oracle and/or its affiliates. All rights reserved.

CHAPTER

1

Fundamentals

The lives of most inhabitants of the industrialized countries, as well as many in the unindustrialized countries, have been changed forever by the advent of the World Wide Web. Although this transformation has had some downsides—for example, easier access to pornography and gambling and the ease with which people with destructive ideas can propagate those ideas to others—on balance, the changes have been enormously positive. Many use the Internet and the World Wide Web daily, communicating with friends, relatives, and business associates through electronic mail and social networking sites, shopping for virtually anything that can be purchased anywhere, and digging up a limitless variety and amount of information, from movie theater show times, to hotel room prices in cities halfway around the world, to the history and characteristics of the culture of some small and obscure society. In recent years, social networking has been used effectively to organize social and political demonstrations, and even revolutions. Constructing the software and data that provide access to all this information requires knowledge of several different technologies, such as markup

languages and meta-markup languages, as well as programming skills in a myriad of different programming languages, some specific to the World Wide Web and some designed for general-purpose computing. This book is meant to provide the required background and a basis for acquiring the knowledge and skills necessary to build the World Wide Web sites that provide both the information users want and the advertising that pays for its presentation.

This chapter lays the groundwork for the remainder of the book. It begins with introductions to, and some history of, the Internet and the World Wide Web. Then, it discusses the purposes and some of the characteristics of Web browsers and servers. Next, it describes Uniform Resource Locators (URLs), which specify addresses of resources available on the Web. Following this, it introduces Multipurpose Internet Mail Extensions (MIMEs), which define types and file name extensions for files with different kinds of contents. Next, it discusses the Hypertext Transfer Protocol (HTTP), which provides the communication interface for connections between browsers and Web servers. Finally, the chapter gives brief overviews of some of the tools commonly used by Web programmers, including HTML, XML, JavaScript, Flash, Servlets, JSP, JSF, ASP.NET, PHP, Ruby, Rails, and Ajax. They are discussed in far more detail in the remainder of the book (HTML in Chapters 2 and 3; JavaScript in Chapters 4, 5, and 6; XML in Chapter 7; Flash in Chapter 8; PHP in Chapter 9; Ajax in Chapter 10; Servlets, JSP, and JSF in Chapter 11; Ruby in Chapters 14 and 15; and Rails in Chapter 15).

1.1 A Brief Introduction to the Internet

Virtually every topic discussed in this book is related to the Internet. Therefore, we begin with a quick introduction to the Internet itself.

1.1.1 Origins

In the 1960s, the U.S. Department of Defense (DoD) became interested in developing a new large-scale computer network. The purposes of this network were communications, program sharing, and remote computer access for researchers working on defense-related contracts. One fundamental requirement was that the network be sufficiently robust so that even if some network nodes were lost to sabotage, war, or some more benign cause, the network would continue to function. The DoD's Advanced Research Projects Agency (ARPA)[1] funded the construction of the first such network, which connected about a dozen ARPA-funded research laboratories and universities. The first node of this network was established at UCLA in 1969.

Because it was funded by ARPA, the network was named ARPAnet. Despite the initial intentions, the primary early use of ARPAnet was simple text-based communications through electronic mail. Because ARPAnet was available only

1. ARPA was renamed Defense Advanced Research Projects Agency (DARPA) in 1972.

to laboratories and universities that conducted ARPA-funded research, the great majority of educational institutions were not connected. As a result, several other networks were developed during the late 1970s and early 1980s, with BITNET and CSNET among them. BITNET, which is an acronym for *B*ecause *I*t's *T*ime *Net*work, began at the City University of New York. It was built initially to provide electronic mail and file transfers. CSNET, which is an acronym for *C*omputer *S*cience *Net*work, connected the University of Delaware, Purdue University, the University of Wisconsin, the RAND Corporation, and Bolt, Beranek, and Newman (a research company in Cambridge, Massachusetts). Its initial purpose was to provide electronic mail. For a variety of reasons, neither BITNET nor CSNET became a widely used national network.

A new national network, NSFnet, was created in 1986. It was sponsored, of course, by the National Science Foundation (NSF). NSFnet initially connected the NSF-funded supercomputer centers that were at five universities. Soon after being established, it became available to other academic institutions and research laboratories. By 1990, NSFnet had replaced ARPAnet for most nonmilitary uses, and a wide variety of organizations had established nodes on the new network—by 1992, NSFnet connected more than one million computers around the world. In 1995, a small part of NSFnet returned to being a research network. The rest became known as the Internet, although this term was used much earlier for both ARPAnet and NSFnet.

1.1.2 What Is the Internet?

The Internet is a huge collection of computers connected in a communications network. These computers are of every imaginable size, configuration, and manufacturer. In fact, some of the devices connected to the Internet—such as plotters and printers—are not computers at all. The innovation that allows all these diverse devices to communicate with each other is a single, low-level protocol named Transmission Control Protocol/Internet Protocol (TCP/IP). TCP/IP became the standard for computer network connections in 1982. It can be used directly to allow a program on one computer to communicate with a program on another computer via the Internet. In most cases, however, a higher-level protocol runs on top of TCP/IP. Nevertheless, it is TCP/IP that provides the low-level interface that allows most computers (and other devices) connected to the Internet to appear exactly the same.[2]

Rather than connecting every computer on the Internet directly to every other computer on the Internet, normally the individual computers in an organization are connected to each other in a local network. One node on this local network is physically connected to the Internet. So, the Internet is actually a network of networks, rather than a network of computers.

Obviously, all devices connected to the Internet must be uniquely identifiable.

2. TCP/IP is not the only communication protocol used by the Internet—User Datagram Protocol/Internet Protocol (UDP/IP) is an alternative that is used in some situations.

1.1.3 Internet Protocol Addresses

For people, Internet nodes are identified by names; for computers, they are identified by numeric addresses. This relationship exactly parallels the one between a variable name in a program, which is for people, and the variable's numeric memory address, which is for the machine.

The Internet Protocol (IP) address of a machine connected to the Internet is a unique 32-bit number. IP addresses usually are written (and thought of) as four 8-bit numbers, separated by periods. The four parts are separately used by Internet-routing computers to decide where a message must go next to get to its destination.

Organizations are assigned blocks of IPs, which they in turn assign to their machines that need Internet access—which now include virtually all computers. For example, a small organization may be assigned 256 IP addresses, such as `191.57.126.0` to `191.57.126.255`. Very large organizations, such as the Department of Defense, may be assigned 16 million IP addresses, which include IP addresses with one particular first 8-bit number, such as `12.0.0.0` to `12.255.255.255`.

Although people nearly always type domain names into their browsers, the IP works just as well. For example, the IP for United Airlines (`www.ual.com`) is `209.87.113.93`. So, if a browser is pointed at `http://209.87.113.93`, it will be connected to the United Airlines Web site.

In late 1998, a new IP standard, IPv6, was approved, although it still is not widely used. The most significant change was to expand the address size from 32 bits to 128 bits. This is a change that will soon be essential because the number of remaining unused IP addresses is diminishing rapidly.

1.1.4 Domain Names

Because people have difficulty dealing with and remembering numbers, machines on the Internet also have textual names. These names begin with the name of the host machine, followed by progressively larger enclosing collections of machines, called *domains*. There may be two, three, or more domain names. The first domain name, which appears immediately to the right of the host name, is the domain of which the host is a part. The second domain name gives the domain of which the first domain is a part. The last domain name identifies the type of organization in which the host resides, which is the largest domain in the site's name. For organizations in the United States, `edu` is the extension for educational institutions, `com` specifies a company, `gov` is used for the U.S. government, and `org` is used for many other kinds of organizations. In other countries, the largest domain is often an abbreviation for the country—for example, `se` is used for Sweden, and `kz` is used for Kazakhstan.

Consider this sample address:

```
movies.marxbros.comedy.com
```

Here, `movies` is the host name and `marxbros` is `movies`'s local domain, which is a part of `comedy`'s domain, which is a part of the `com` domain. The host name and all the domain names are together called a *fully qualified domain name*.

Because IP addresses are the addresses used internally by the Internet, the fully qualified domain name of the destination for a message, which is what is given by a browser user, must be converted to an IP address before the message can be transmitted over the Internet to the destination. These conversions are done by software systems called *name servers*, which implement the Domain Name System (DNS). Name servers serve a collection of machines on the Internet and are operated by organizations that are responsible for the part of the Internet to which those machines are connected. All document requests from browsers are routed to the nearest name server. If the name server can convert the fully qualified domain name to an IP address, it does so. If it cannot, the name server sends the fully qualified domain name to another name server for conversion. Like IP addresses, fully qualified domain names must be unique. Figure 1.1 shows how fully qualified domain names requested by a browser are translated into IPs before they are routed to the appropriate Web server.

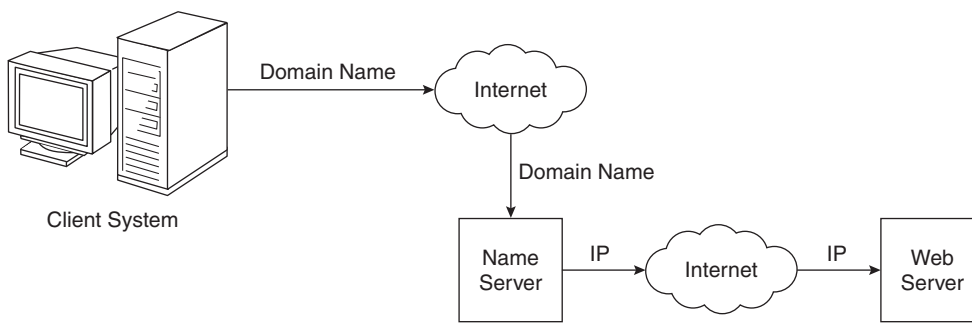


Figure 1.1 Domain name conversion

One way to determine the IP address of a Web site is by using `telnet` on the fully qualified domain name. This approach is illustrated in Section 1.7.1.

By the mid-1980s, a collection of different protocols that run on top of TCP/IP had been developed to support a variety of Internet uses. Among these protocols, the most common were `telnet`, which was developed to allow a user on one computer on the Internet to log onto and use another computer on the Internet; File Transfer Protocol (`ftp`), which was developed to transfer files among computers on the Internet; `Usenet`, which was developed to serve as an electronic bulletin board; and `mailto`, which was developed to allow messages to be sent from the user of one computer on the Internet to other users of other computers on the Internet.

This variety of protocols, each having its own user interface and useful only for the purpose for which it was designed, restricted the growth of the Internet. Users were required to learn all the different interfaces to gain all

the advantages of the Internet. Before long, however, a better approach was developed: the World Wide Web.

1.2 The World Wide Web

This section provides a brief introduction to the evolution of the World Wide Web.

1.2.1 Origins

In 1989, a small group of people led by Tim Berners-Lee at Conseil Européen pour la Recherche Nucléaire (CERN) or European Organization for Particle Physics proposed a new protocol for the Internet, as well as a system of document access to use it.[3] The intent of this new system, which the group named the World Wide Web, was to allow scientists around the world to use the Internet to exchange documents describing their work.

The proposed new system was designed to allow a user anywhere on the Internet to search for and retrieve documents from databases on any number of different document-serving computers connected to the Internet. By late 1990, the basic ideas for the new system had been fully developed and implemented on a NeXT computer at CERN. In 1991, the system was ported to other computer platforms and released to the rest of the world.

For the form of its documents, the new system used *hypertext*, which is text with embedded links to text, either in the same document or in another document, to allow nonsequential browsing of textual material. The idea of hypertext had been developed earlier and had appeared in Xerox's NoteCards and Apple's HyperCard in the mid-1980s.

From here on, we will refer to the World Wide Web simply as the *Web*. The units of information on the Web have been referred to by several different names; among them, the most common are *pages*, *documents*, and *resources*. Perhaps the best of these is *documents*, although that seems to imply only text. *Pages* is widely used, but it is misleading in that Web units of information often have more than one of the kind of pages that make up printed media. There is some merit to calling these units *resources*, because that covers the possibility of nontextual information. This book will use *documents* and *pages* more or less interchangeably, but we prefer *documents* in most situations.

Documents are sometimes just text, usually with embedded links to other documents, but they often also include images, sound recordings, or other kinds of media. When a document contains nontextual information, it is called *hypermedia*.

In an abstract sense, the Web is a vast collection of documents, some of which are connected by links. These documents are accessed by Web browsers, introduced in Section 1.3, and are provided by Web servers, introduced in Section 1.4.

3. Although Berners-Lee's college degree (from Oxford) was in physics, his first stint at CERN was as a consulting software engineer. Berners-Lee was born and raised in London.

1.2.2 Web or Internet?

It is important to understand that the Internet and the Web are not the same thing. The *Internet* is a collection of computers and other devices connected by equipment that allows them to communicate with each other. The *Web* is a collection of software and protocols that has been installed on most, if not all, of the computers on the Internet. Some of these computers run Web servers, which provide documents, but most run Web clients, or browsers, which request documents from servers and display them to users. The Internet was quite useful before the Web was developed, and it is still useful without it. However, most users of the Internet now use it through the Web.

1.3 Web Browsers

When two computers communicate over some network, in many cases one acts as a client and the other as a server. The client initiates the communication, which is often a request for information stored on the server, which then sends that information back to the client. The Web, as well as many other systems, operates in this client-server configuration.

Documents provided by servers on the Web are requested by *browsers*, which are programs running on client machines. They are called browsers because they allow the user to browse the resources available on servers. The first browsers were text based—they were not capable of displaying graphic information, nor did they have a graphical user interface. This limitation effectively constrained the growth of the Web. In early 1993, things changed with the release of Mosaic, the first browser with a graphical user interface. Mosaic was developed at the National Center for Supercomputer Applications (NCSA) at the University of Illinois. Mosaic's interface provided convenient access to the Web for users who were neither scientists nor software developers. The first release of Mosaic ran on UNIX systems using the X Window system. By late 1993, versions of Mosaic for Apple Macintosh and Microsoft Windows systems had been released. Finally, users of the computers connected to the Internet around the world had a powerful way to access anything on the Web anywhere in the world. The result of this power and convenience was explosive growth in Web usage.

A browser is a client on the Web because it initiates the communication with a server, which waits for a request from the client before doing anything. In the simplest case, a browser requests a static document from a server. The server locates the document among its servable documents and sends it to the browser, which displays it for the user. However, more complicated situations are common. For example, the server may provide a document that requests input from the user through the browser. After the user supplies the requested input, it is transmitted from the browser to the server, which may use the input to perform some computation and then return a new document to the browser to inform the user of the results of the computation. Sometimes a browser directly requests the execution of a program stored on the server. The output of the program is then returned to the browser.

Although the Web supports a variety of protocols, the most common one is the HTTP. HTTP provides a standard form of communication between browsers and Web servers. Section 1.7 provides an introduction to HTTP.

The most commonly used browsers are Microsoft Internet Explorer (IE), which runs only on PCs that use one of the Microsoft Windows operating systems,[4] Firefox, and Chrome. The latter two are available in versions for several different computing platforms, including Windows, Mac OS, and Linux. Several other browsers are available, including Opera and Apple's Safari. However, because the great majority of browsers now in use are Chrome, IE, or Firefox, in this book we focus on them.

1.4 Web Servers

Web servers are programs that provide documents to requesting browsers. Servers are slave programs: They act only when requests are made to them by browsers running on other computers on the Internet.

The most commonly used Web servers are Apache, which has been implemented for a variety of computer platforms, and Microsoft's Internet Information Server (IIS), which runs under Windows operating systems. As of October 2013, there were over 150 million active Web hosts in operation,[5] about 65 percent of which were Apache, about 16 percent were IIS, and about 14 percent were `nginx` (pronounced "engine-x"), a product produced in Russia.[6]

1.4.1 Web Server Operation

Although having clients and servers is a natural consequence of information distribution, this configuration offers some additional benefits for the Web. While serving information does not take a great deal of time, displaying information on client screens is time consuming. Because Web servers need not be involved in this display process, they can handle many clients. So, it is both a natural and efficient division of labor to have a small number of servers provide documents to a large number of clients.

Web browsers initiate network communications with servers by sending them URLs (discussed in Section 1.5). A URL can specify one of two different things: the address of a data file stored on the server that is to be sent to the client, or a program stored on the server that the client wants executed and the output of the program returned to the client.

4. Actually, versions 4 and 5 of IE (IE4 and IE5) were also available for Macintosh computers, and IE4 was available for UNIX systems. However, later versions are available for Windows platforms only.

5. There were well more than 500 million sites on line.

6. These statistics are from `www.netcraft.com` and `W3techs.com`.

All the communications between a Web client and a Web server use the standard Web protocol, HTTP, which is discussed in Section 1.7.[7]

When a Web server begins execution, it informs the operating system under which it is running that it is now ready to accept incoming network connections through a specific port on the machine. While in this running state, the server runs as a background process in the operating system environment. A Web client, or browser, opens a network connection to a Web server, sends information requests and possibly data to the server, receives information from the server, and closes the connection. Of course, other machines exist between browsers and servers on the network—specifically, network routers and domain name servers. This section, however, focuses on just one part of Web communication: the server.

Simply put, the primary task of a Web server is to monitor a communications port on its host machine, accept HTTP commands through that port, and perform the operations specified by those commands. All HTTP commands include a URL, which includes the specification of a host server machine. When the URL is received, it is translated into either a file name (in which case the file is returned to the requesting client) or a program name (in which case the program is run and its output is sent to the requesting client). This process sounds pretty simple, but, as is the case in many other simple-sounding processes, there are a large number of complicating details.

All current Web servers have a common ancestry: the first two servers, developed at CERN in Europe and NCSA at the University of Illinois. Currently, the two most common server configurations are Apache running on Linux and Microsoft's IIS running on Windows.

1.4.2 General Server Characteristics

Most of the available servers share common characteristics, regardless of their origin or the platform on which they run. This section provides brief descriptions of some of these characteristics.

The file structure of a Web server has two separate directories. The root of one of these is the *document root*. The file hierarchy that grows from the document root stores the Web documents to which the server has direct access and normally serves to clients. The root of the other directory is the *server root*. This directory, along with its descendant directories, stores the server and its support software.

The files stored directly in the document root are those available to clients through top-level URLs. Typically, clients do not access the document root directly in URLs; rather, the server maps requested URLs to the document root, whose location is not known to clients. For example, suppose that the site name is `www.tunias.com` (not a real site—at least, not yet), which we will assume to be a UNIX-based system. Suppose further that the document root is named `topdocs` and is stored in the `/admin/web` directory, making its address `/admin/web/topdocs`. A request for a file from a client with the URL `http://www.tunias.com/petunias.html` will cause the server to search for the file with the file

7. Some of these communications use HTTPS, the secure version of HTTP.

path `/admin/web/topdocs/petunias.html`. Likewise, the URL `http://www.tunias.com/bulbs/tulips.html` will cause the server to search for the file with the address `/admin/web/topdocs/bulbs/tulips.html`.

Many servers allow part of the servable document collection to be stored outside the directory at the document root. The secondary areas from which documents can be served are called *virtual document trees*. For example, the original configuration of a server might have the server store all its servable documents from the primary system disk on the server machine. Later, the collection of servable documents might outgrow that disk, in which case part of the collection could be stored on a secondary disk. This secondary disk might reside on the server machine or on some other machine on a local area network. To support this arrangement, the server is configured to direct-request URLs with a particular file path to a storage area separate from the document-root directory. Sometimes files with different types of content, such as images, are stored outside the document root.

Early servers provided few services other than the basic process of returning requested files or the output of programs whose execution had been requested. The list of additional services has grown steadily over the years. Contemporary servers are large and complex systems that provide a wide variety of client services. Many servers can support more than one site on a computer, potentially reducing the cost of each site and making their maintenance more convenient. Such secondary hosts are called *virtual hosts*.

Some servers can serve documents that are in the document root of other machines on the Web; these are called *proxy servers*.

Although Web servers were originally designed to support only the HTTP protocol, many now support `ftp`, `gopher`, `news`, and `mailto`. In addition, nearly all Web servers can interact with database systems through server-side scripts.

1.4.3 Apache

Apache began as the NCSA server, `httpd`, with some added features. The name *Apache* has nothing to do with the Native American tribe of the same name. Rather, it came from the nature of its first version, which was *a patch*y version of the `httpd` server. The primary reasons for the popularity of Apache are that it is both fast and reliable. Furthermore, it is open-source software, which means that it is free and is managed by a large team of volunteers, a process that efficiently and effectively maintains the system. Finally, it is one of the best available servers for Linux systems, which are the most popular for Web servers.

Apache is capable of providing a long list of services beyond the basic process of serving documents to clients. When Apache begins execution, it reads its configuration information from a file and sets its parameters to operate accordingly. A new copy of Apache includes default configuration information for a *typical* operation. The site manager modifies this configuration information to fit his or her particular needs and tastes.

For historical reasons, there are three configuration files in an Apache server: `httpd.conf`, `srm.conf`, and `access.conf`. Only one of these, `httpd.conf`,

actually stores the directives that control an Apache server's behavior. The other two point to `httpd.conf`, which is the file that contains the list of directives that specify the server's operation. These directives are described at `http://httpd.apache.org/docs/2.2/mod/quickreference.html`.

1.4.4 IIS

Although Apache has been ported to the Windows platforms, it is not the most popular server on those systems. Because the Microsoft IIS server is supplied as part of Windows—and because it is a reasonably good server—most Windows-based Web servers use IIS. Apache and IIS provide similar varieties of services.

From the point of view of the site manager, the most important difference between Apache and IIS is that Apache is controlled by a configuration file that is edited by the manager to change Apache's behavior. With IIS, server behavior is modified by changes made through a window-based management program, named the IIS snap-in, which controls both IIS and `ftp`. This program allows the site manager to set parameters for the server.

1.5 Uniform Resource Locators

Uniform (or universal)[8] Resource Identifiers (URIs) are used to identify resources (often documents) on the Internet. URIs are used for two different purposes, to name a resource, in which case they are often called URIs, even though they could be more accurately called Uniform Resource Names (URNs). The more commonly used form of URIs is to provide a path to, or location of, a resource, in which case they are called Uniform Resource Locators (URLs). The general forms of URIs and URLs are similar, and URIs are often confused with URLs. We will use URIs in Chapter 8 to name namespaces for use with XML Schema. In this chapter, we only deal with URLs.

1.5.1 URL Formats

All URLs have the same general format:

scheme:object-address

The scheme is often a communications protocol. Common schemes include `http`, `ftp`, `gopher`, `telnet`, `file`, `mailto`, and `news`. Different schemes use object addresses that have different forms. Our interest here is in the HTTP protocol, which supports the Web. This protocol is used to request and send Hypertext Markup Language (HTML) documents. In the case of HTTP, the form of the object address of a URL is as follows:

//fully-qualified-domain-name/path-to-document

8. Fortunately, resource addresses are usually referred to as URIs, so whether it is *uniform* or *universal* is usually irrelevant.

Another scheme of interest to us is `file`. The `file` protocol means that the document resides on the machine running the browser. This approach is useful for testing documents to be made available on the Web without making them visible to any other browser. When `file` is the protocol, the fully qualified domain name is omitted, making the form of such URLs as follows:

`file:` *//path-to-document*

Because the focus of this book is HTML documents, the remainder of the discussion of URLs is limited to the HTTP protocol.

The *host name* is the name of the server computer that stores the document (or provides access to it, although it is stored on some other computer). Messages to a host machine must be directed to the appropriate process running on the host for handling. Such processes are identified by their associated port numbers. The default port number of Web server processes is 80. If a server has been configured to use some other port number, it is necessary to attach that port number to the host name in the URL. For example, if the Web server is configured to use port 800, the host name must have `:800` attached.

URLs can never have embedded spaces.[9] Also, there is a collection of special characters, including semicolons, colons, and ampersands (`;`, `:`, `&`), that cannot appear in a URL. To include a space or one of the disallowed special characters, the character must be coded as a percent sign (`%`) followed by the two-digit hexadecimal ASCII code for the character. For example, if `San Jose` is a domain name, it must be typed as `San%20Jose` (`20` is the hexadecimal ASCII code for a space). All the details characterizing URLs can be found at `http://www.w3.org/Addressing/URL/URI_Overview.html`.

1.5.2 URL Paths

The path to the document for the HTTP protocol is similar to a path to a file or directory in the file system of an operating system and is given by a sequence of directory names and a file name, all separated by whatever separator character the operating system uses. For UNIX servers, the path is specified with forward slashes; for Windows servers, it is specified with backward slashes. Most browsers allow the user to specify the separators incorrectly—for example, using forward slashes in a path to a document file on a Windows server, as in the following:

```
http://www.gumboco.com/files/f99/storefront.html
```

The path in a URL can differ from a path to a file because a URL need not include all directories on the path. A path that includes all directories along the way is called a *complete path*. In most cases, the path to the document is relative to some base path that is specified in the configuration files of the server. Such

9. Actually, some browsers incorrectly accept spaces in URLs, although doing so is nonstandard behavior.

paths are called *partial paths*. For example, if the server's configuration specifies that the root directory for files it can serve is `files/f99`, the previous URL is specified as follows:

```
http://www.gumboco.com/storefront.html
```

If the specified document is a directory rather than a single document, the directory's name is followed immediately by a slash, as in the following:

```
http://www.gumboco.com/departments/
```

Sometimes a directory is specified (with the trailing slash) but its name is not given, as in the following example:

```
http://www.gumboco.com/
```

The server then searches at the top level of the directory in which servable documents are normally stored for something it recognizes as a home page. By convention, this page is often a file named `index.html`. The home page usually includes links that allow the user to find the other related servable files on the server.

If the directory does not have a file that the server recognizes as being a home page, a directory listing is constructed and returned to the browser.

1.6 Multipurpose Internet Mail Extensions

A browser needs some way to determine the format of a document it receives from a Web server. Without knowing the form of the document, the browser would not be able to render it, because different document formats require different rendering software. The forms of these documents are specified with Multipurpose Internet Mail Extensions (MIMEs).

1.6.1 Type Specifications

MIME was developed to specify the format of different kinds of documents to be sent via Internet mail. These documents could contain various kinds of text, video data, or sound data. Because the Web has needs similar to those of Internet mail, MIME was adopted as the way to specify document types transmitted over the Web. A Web server attaches an MIME format specification to the beginning of the document that it is about to provide to a browser. When the browser receives the document from a Web server, it uses the included MIME format specification to determine what to do with the document. If the content is text, for example, the MIME code tells the browser that it is text and also indicates the particular kind of text it is. If the content is sound, the MIME code tells the browser that it is sound and then gives the particular representation of sound so the browser can choose a program to which it has access to produce the transmitted sound.

MIME specifications have the following form:

type/subtype

The most common MIME types are `text`, `image`, and `video`. The most common text subtypes are `plain` and `html`. Some common image subtypes are `gif` and `jpeg`. Some common video subtypes are `mpeg` and `quicktime`. A list of MIME specifications is stored in the configuration files of every Web server. In the remainder of this book, when we say *document type*, we mean both the type and subtype of the document.

Servers determine the type of a document by using the file name extension as the key into a table of types. For example, the extension `.html` tells the server that it should attach `text/html` to the document before sending it to the requesting browser.[10]

Browsers also maintain a conversion table for looking up the type of a document by its file name extension. However, this table is used only when the server does not specify an MIME type, which may be the case with some older servers. In all other cases, the browser gets the document type from the MIME header provided by the server.

1.6.2 Experimental Document Types

Experimental subtypes are sometimes used. The name of an experimental subtype begins with `x-`, as in `video/x-msvideo`. Any Web provider can add an experimental subtype by having its name added to the list of MIME specifications stored in the Web provider's server. For example, a Web provider might have a handcrafted database whose contents he or she wants to make available to others through the Web. Of course, this raises the issue of how the browser can display the database. As might be expected, the Web provider must supply a program that the browser can call when it needs to display the contents of the database. These programs either are external to the browser, in which case they are called *helper applications*, or are code modules that are inserted into the browser, in which case they are called *plug-ins*.

Every browser has a set of MIME specifications (file types) it can handle. All can deal with `text/plain` (unformatted text) and `text/html` (HTML files), among others. Sometimes a particular browser cannot handle a specific document type, even though the type is widely used. These cases are handled in the same way as the experimental types described previously. The browser determines the helper application or plug-in it needs by examining the browser configuration file, which provides an association between file types and their required helpers or plug-ins. If the browser does not have an application or a plug-in that it needs to render a document, an error message is displayed.

A browser can indicate to the server the document types it prefers to receive, as discussed in Section 1.7.

10. This is not necessarily correct. HTML documents also use the `.html` file extension, but, strictly speaking, they should use a different MIME type.

1.7 The Hypertext Transfer Protocol

All Web communications transactions use the same protocol: the Hypertext Transfer Protocol (HTTP). The current version of HTTP is 1.1, formally defined as RFC 2616, which was approved in June 1999. RFC 2616 is available at the Web site for the World Wide Web Consortium (W3C), `http://www.w3.org`. This section provides a brief introduction to HTTP.

HTTP consists of two phases: the request and the response. Each HTTP communication (request or response) between a browser and a Web server consists of two parts: a header and a body. The header contains information about the communication; the body contains the data of the communication if there is any.

1.7.1 The Request Phase

The general form of an HTTP request is as follows:

1. HTTP method Domain part of the URL HTTP version
2. Header fields
3. Blank line
4. Message body

The following is an example of the first line of an HTTP request:

```
GET /storefront.html HTTP/1.1
```

Only a few request methods are defined by HTTP, and even a smaller number of these are typically used. Table 1.1 lists the most commonly used methods.

Table 1.1 HTTP request methods

Method	Description
`GET`	Returns the contents of a specified document
`HEAD`	Returns the header information for a specified document
`POST`	Executes a specified document, using the enclosed data
`PUT`	Replaces a specified document with the enclosed data
`DELETE`	Deletes a specified document

Among the methods given in Table 1.1, `GET` and `POST` are the most frequently used. `POST` was originally designed for tasks such as posting a news article to a newsgroup. Its most common use now is to send form data from a browser to a server, along with a request to execute a server-resident program on the server that will process the data.

Following the first line of an HTTP communication is any number of header fields, most of which are optional. The format of a header field is the field name

followed by a colon and the value of the field. There are four categories of header fields:

1. *General:* For general information, such as the date
2. *Request:* Included in request headers
3. *Response:* For response headers
4. *Entity:* Used in both request and response headers

One common request field is the `Accept` field, which specifies a preference of the browser for the MIME type of the requested document. More than one `Accept` field can be specified if the browser is willing to accept documents in more than one format. For example, we might have any of the following:

```
Accept: text/plain
Accept: text/html
Accept: image/gif
```

A wildcard character, the asterisk (*), can be used in place of a part of a MIME type, in which case that part can be anything. For example, if any kind of text is acceptable, the `Accept` field could be as follows:

```
Accept: text/*
```

The `Host:` *`host name`* request field gives the name of the host. The `Host` field is required for HTTP 1.1. The `If-Modified-Since: date` request field specifies that the requested file should be sent only if it has been modified since the given date.

If the request has a body, the length of that body must be given with a `Content-length` field, which gives the length of the response body in bytes. The `POST` method requests require this field because they send data to the server.

The header of a request must be followed by a blank line, which is used to separate the header from the body of the request. Requests that use the `GET`, `HEAD`, and `DELETE` methods do not have bodies. In these cases, the blank line signals the end of the request.

A browser is not necessary to communicate with a Web server; `telnet` can be used instead. Consider the following command, given at the command line of any widely used operating system:

```
>telnet blanca.uccs.edu http
```

This command creates a connection to the `http` port on the `blanca.uccs.edu` server. The server responds with the following:[11]

```
Trying 128.198.162.60 ...
Connected to blanca
Escape character is '^]'.
```

11. Notice that this `telnet` request returns to the IP of the server.

The connection to the server is now complete, and HTTP commands such as the following can be given:

```
GET /~user1/respond.html HTTP/1.1
Host: blanca.uccs.edu
```

The header of the response to this request is given in Section 1.7.2.

1.7.2 The Response Phase

The general form of an HTTP response is as follows:

1. Status line
2. Response header fields
3. Blank line
4. Response body

The status line includes the HTTP version used, a three-digit status code for the response, and a short textual explanation of the status code. For example, most responses begin with the following:

```
HTTP/1.1  200  OK
```

The status codes begin with `1`, `2`, `3`, `4`, or `5`. The general meanings of the five categories specified by these first digits are shown in Table 1.2.

Table 1.2 First digits of HTTP status codes

First Digit	Category
1	Informational
2	Success
3	Redirection
4	Client error
5	Server error

One of the more common status codes is one users never want to see: `404 Not Found`, which means the requested file could not be found. Of course, `200 OK` is what users want to see, because it means that the request was handled without error. The `500` code means that the server has encountered a problem and was not able to fulfill the request.

After the status line, the server sends a response header, which can contain several lines of information about the response, each in the form of a field. The only essential field of the header is `Content-type`.

The following is the response header for the request given near the end of Section 1.7.1:

```
HTTP/1.1 200 OK
Date: Sat, 25 July 2009 22:15:11 GMT
Server: Apache/2.2.3 (CentOS)
Last-modified: Tues, 18 May 2004 16:38:38 GMT
ETag: "1b48098-16c-3dab592dc9f80"
Accept-ranges: bytes
Content-length: 364
Connection: close
Content-type: text/html, charset=UTF-8
```

The response header must be followed by a blank line, as is the case for request headers. The response data follows the blank line. In the preceding example, the response body would be the HTML file, `respond.html`.

The default operation of HTTP 1.1 is that the connection is kept open for a time so that the client can make several requests over a short span of time without needing to reestablish the communications connection with the server.

1.8 Security

The Internet and the Web are fertile grounds for security problems. On the Web server side, anyone on the planet with a computer, a browser, and an Internet connection can request the execution of software on any server computer. He or she can also access data and databases stored on the server computer. On the browser end, the problem is similar: Any server to which the browser points can download software to be executed on the browser host machine. Such software can access parts of the memory and memory devices attached to that machine that are not related to the needs of the original browser request. In effect, on both ends, it is like allowing any number of total strangers into your house and trying to prevent them from leaving anything in the house, taking anything from the house, or altering anything in the house. The larger and more complex the design of the house, the more difficult it will be to prevent any of those activities. The same is true for Web servers and browsers: The more complex they are, the more difficult it is to prevent security breaches. Today's browsers and Web servers are indeed large and complex software systems, so security is a significant problem in Web applications.

The subject of Internet and Web security is extensive and complicated, so much so that numerous books have been written on the topic. Therefore, this one section of one chapter of one book can give no more than a brief sketch of some of the subtopics of security.

One aspect of Web security is the matter of getting one's data from the browser to the server and having the server deliver data back to the browser without anyone or any device intercepting or corrupting those data along the way. Consider a simple case of transmitting a credit card number to a company

from which a purchase is being made. The security issues for this transaction are as follows:

1. *Privacy:* It must not be possible for the credit card number to be stolen on its way to the company's server.
2. *Integrity:* It must not be possible for the credit card number to be modified on its way to the company's server.
3. *Authentication:* It must be possible for both the purchaser and the seller to be certain of each other's identity.
4. *Nonrepudiation:* It must be possible to prove legally that the message was actually sent and received.

The basic tool to support privacy and integrity is encryption. Data to be transmitted is converted into a different form, or encrypted, such that someone (or some computer) who is not supposed to access the data cannot decrypt it. So, if data is intercepted while en route between Internet nodes, the interceptor cannot use the data because he or she cannot decrypt it. Both encryption and decryption are done with a key and a process (applying the key to the data). Encryption was developed long before the Internet existed. Julius Caesar crudely encrypted the messages he sent to his field generals while at war. Until the middle 1970s, the same key was used for both encryption and decryption, so the initial problem was how to transmit the key from the sender to the receiver.

This problem was solved in 1976 by Whitfield Diffie and Martin Hellman of Stanford University, who developed *public-key encryption*, a process in which a public key and a private key are used, respectively, to encrypt and decrypt messages. A communicator—say, Joe—has an inversely related pair of keys, one public and one private. The public key can be distributed to all organizations that might send messages to Joe. All of them can use the public key to encrypt messages to Joe, who can decrypt the messages with his matching private key. This arrangement works because the private key need never be transmitted and also because it is virtually impossible to compute the private key from its corresponding public key. The technical wording for this situation is that it is *computationally infeasible* to determine the private key from its public key.

The most widely used public-key algorithm is named RSA, developed in 1977 by three MIT professors—Ron Rivest, Adi Shamir, and Leonard Adleman—the first letters of whose last names were used to name the algorithm. Most large companies now use RSA for e-commerce.

Another, completely different security problem for the Web is the intentional and malicious destruction of data on computers attached to the Internet. The number of different ways this can be done has increased steadily over the life span of the Web. The sheer number of such attacks has also grown rapidly. There is now a continuous stream of new and increasingly devious Denial-of-Service (DoS) attacks, viruses, and worms being discovered, which have caused billions of dollars of damage, primarily to businesses that use the Web heavily. Of course, huge damage also has been done to home computer systems through Web intrusions.

DoS attacks can be created simply by flooding a Web server with requests, overwhelming its ability to operate effectively. Most DoS attacks are conducted

with the use of networks of virally infected *zombie* computers, whose owners are unaware of their sinister use. So, DoS and viruses are often related.

Viruses are programs that often arrive in a system in attachments to electronic mail messages or attached to free downloaded programs. Then they attach to other programs. When executed, they replicate and can overwrite memory and attached memory devices, destroying programs and data alike.

Worms damage memory, like viruses, but spread on their own, rather than being attached to other files. Perhaps the most famous worm so far has been the Blaster worm, spawned in 2003.

DoS, virus, and worm attacks are created by malicious people referred to as *hackers*. The incentive for these people apparently is simply the feeling of pride and accomplishment they derive from being able to cause huge amounts of damage by outwitting the designers of Web software systems.

Protection against viruses and worms is provided by antivirus software, which must be updated frequently so that it can detect and protect against the continuous stream of new viruses and worms.

1.9 The Web Programmer's Toolbox

This section provides an overview of the most common tools used in Web programming—some are programming languages, some are not. The tools discussed are HTML, a markup language, along with a few high-level markup document-editing systems; XML, a meta-markup language; JavaScript, PHP, and Ruby, which are programming languages; JSF, ASP.NET, and Rails, which are development frameworks for Web-based systems; Flash, a technology for creating and displaying graphics and animation in HTML documents; and Ajax, a Web technology that uses JavaScript and XML.

Web programs and scripts are divided into two categories—client side and server side—according to where they are interpreted or executed. HTML and XML are client-side languages; PHP and Ruby are server-side languages; JavaScript is most often a client-side language, although it can be used for both.

We begin with the most basic tool: HTML.

1.9.1 Overview of HTML

At the onset, it is important to realize that HTML is not a programming language—it cannot be used to describe computations. Its purpose is to describe the general form and layout of documents to be displayed by a browser.

The word *markup* comes from the publishing world, where it is used to describe what production people do with a manuscript to specify to a printer how the text, graphics, and other elements in the book should appear in printed form. HTML is not the first markup language used with computers. TeX and LaTeX are older markup languages for use with text; they are now used primarily to specify how mathematical expressions and formulas should appear in print.

An HTML document is a mixture of content and controls. The controls are specified by the tags of HTML. The name of a tag specifies the category of its

content. Most HTML tags consist of a pair of syntactic markers that are used to delimit particular kinds of content. The pair of tags and their content together are called an *element*. For example, a paragraph element specifies that its content, which appears between its opening tag, `<p>`, and its closing tag, `</p>`, is a paragraph. A browser has a default style (font, font style, font size, and so forth) for paragraphs, which is used to display the content of a paragraph element.

Some tags include attribute specifications that provide some additional information for the browser. In the following example, the `src` attribute specifies the location of the `img` tag's image content:

```
<img src = "redhead.jpg"/>
```

In this case, the image document stored in the file `redhead.jpg` is to be displayed at the position in the document in which the tag appears.

A brief history of HTML appears in Chapter 2.

1.9.2 Tools for Creating HTML Documents

HTML documents can be created with a general-purpose text editor. There are two kinds of tools that can simplify this task: HTML editors and What-You-See-Is-What-You-Get (WYSIWYG, pronounced *wizzy-wig*) HTML editors.

HTML editors provide shortcuts for producing repetitious tags such as those used to create the rows of a table. They also may provide a spell-checker and a syntax-checker, and they may color code the HTML in the display to make it easier to read and edit.

A more powerful tool for creating HTML documents is a WYSIWYG HTML editor. Using a WYSIWYG HTML editor, the writer can see the formatted document that the HTML describes while he or she is writing the HTML code. WYSIWYG HTML editors are very useful for beginners who want to create simple documents without learning HTML and for users who want to prototype the appearance of a document. Still, these editors sometimes produce poor-quality HTML. In some cases, they create proprietary tags that some browsers will not recognize.

Two examples of WYSIWYG HTML editors are Microsoft Expression Web and Adobe Dreamweaver. Both allow the user to create HTML-described documents without requiring the user to know HTML. They cannot handle all the tags of HTML, but they are very useful for creating many of the common features of documents. Information on Dreamweaver is available at `http://www.adobe.com/`; information on Expression Web is available at `http://www.microsoft.com/`.

1.9.3 Plug-ins and Filters

Two different kinds of converter tools can be used to create HTML documents. *Plug-ins*[12] are programs that can be integrated with a word processor. Plug-ins add new capabilities to the word processor, such as toolbar buttons and menu

12. The word *plug-in* applies to many different software systems that can be added to or embedded in other software systems. For example, many different plug-ins can be added to Web browsers.

elements that provide convenient ways to insert HTML into the document being created or edited. The plug-in makes the word processor appear to be an HTML editor that provides WYSIWYG HTML document development. The end result of this process is an HTML document. The plug-in also makes available all the tools that are inherent in the word processor during HTML document creation, such as a spell-checker and a thesaurus.

The second kind of tool is a *filter*, which converts an existing document in some form, such as LaTeX or Microsoft Word, to HTML. Filters are never part of the editor or word processor that created the document—an advantage because the filter can then be platform independent. For example, a WordPerfect user working on a Macintosh computer can use a filter running on a UNIX platform to produce HTML documents with the same content on that machine. The disadvantage of filters is that creating HTML documents with a filter is a two-step process: First you create the document, and then you use a filter to convert it to HTML.

Neither plugs-ins nor filters produce HTML documents that, when displayed by browsers, have the identical appearance of that produced by the word processor.

The two advantages of both plug-ins and filters, however, are that existing documents produced with word processors can be easily converted to HTML and that users can use a word processor with which they are familiar to produce HTML documents. This obviates the need to learn to format text by using HTML directly. For example, once you learn to create tables with your word processor, it is easier to use that process than to learn to define tables directly in HTML.

The HTML output produced by either filters or plug-ins often must be modified, usually with a simple text editor, to perfect the appearance of the displayed document in the browser. Because this new HTML file cannot be converted to its original form (regardless of how it was created), you will have two different source files for a document, inevitably leading to version problems during maintenance of the document. This is clearly a disadvantage of using converters.

1.9.4 Overview of XML

HTML is defined with the use of the Standard Generalized Markup Language (SGML), which is a language for defining markup languages. (Languages such as SGML are called meta-markup languages.) XML (eXtensible Markup Language) is a simplified version of SGML, designed to allow users to easily create markup languages that fit their own needs. Whereas HTML users must use the predefined set of tags and attributes, when a user creates his or her own markup language with XML, the set of tags and attributes is designed for the application at hand. For example, if a group of users wants a markup language to describe data about weather phenomena, that language could have tags for cloud forms, thunderstorms, and low-pressure centers. The content of these tags would be restricted to relevant data. If such data is described with HTML, cloud forms could be put in generic tags, but then they could not be distinguished from thunderstorm elements, which would also be in the same generic tags.

Whereas HTML describes the overall layout and gives some presentation hints for general information, XML-based markup languages describe data and its meaning through their individualized tags and attributes. XML does not specify any presentation details.

The great advantage of XML is that application programs can be written to use the meanings of the tags in the given markup language to find specific kinds of data and process it accordingly. The syntax rules of XML, along with the syntax rules for a specific XML-based markup language, allow documents to be validated before any application attempts to process their data. This means that all documents that use a specific markup language can be checked to determine whether they are in the standard form for such documents. Such an approach greatly simplifies the development of application programs that process the data in XML documents.

XML is discussed in Chapter 7.

1.9.5 Overview of JavaScript

JavaScript is a client-side scripting language whose primary uses in Web programming are to validate form data, to build Ajax-enabled HTML documents and to create dynamic HTML documents.

The name *JavaScript* is misleading because the relationship between Java and JavaScript is tenuous, except for some of the syntax. One of the most important differences between JavaScript and most common programming languages is that JavaScript is dynamically typed. This type design is virtually the opposite of that of strongly typed languages such as C++ and Java.

JavaScript "programs" are usually embedded in HTML documents,[13] which are downloaded from a Web server when they are requested by browsers. The JavaScript code in an HTML document is interpreted by an interpreter embedded in the browser on the client.

One of the most important applications of JavaScript is to create and modify documents dynamically. JavaScript defines an object hierarchy that matches a hierarchical model of an HTML document. Elements of an HTML document are accessed through these objects, providing the basis for dynamic documents.

Chapter 4 provides a more detailed look at JavaScript. Chapters 5 and 6 discuss the use of JavaScript to provide access to and dynamic modification of HTML documents.

1.9.6 Overview of Flash

There are two components of Flash: the authoring environment, which is a development framework, and the player. Developers use the authoring environment to create static graphics, animated graphics, text, sound, and interactivity

13. We quote the word *programs* to indicate that these are not programs in the general sense of the self-contained collections of C++ or C code we normally call programs.

to be a part of stand-alone HTML documents or to be a part of other HTML documents. These documents are served by Web servers to browsers, which use the Flash player plug-in to display the documents. Much of this development is done by clicking buttons, choosing menu items, and dragging and dropping graphic elements.

Flash makes animation very easy. For example, for motion animation, the developer needs only to supply the beginning and ending positions of the figure to be animated—Flash builds the intervening frames. The interactivity of a Flash application is implemented with ActionScript, a dialect of JavaScript.

Flash is now the leading technology for delivering graphics and animation on the Web. It has been estimated that nearly 99 percent of the world's computers used to access the Internet have a version of the Flash player installed as a plug-in in their browsers.

Flash is introduced in Chapter 8.

1.9.7 Overview of PHP

PHP is a server-side scripting language specifically designed for Web applications. PHP code is embedded in HTML documents, as is the case with JavaScript. With PHP, however, the code is interpreted on the server before the HTML document is delivered to the requesting client. A requested document that includes PHP code is preprocessed to interpret the PHP code and insert its output into the HTML document. The browser never sees the embedded PHP code and is not aware that a requested document originally included such code.

PHP is similar to JavaScript, both in terms of its syntactic appearance and in terms of the dynamic nature of its strings and arrays. Both JavaScript and PHP use dynamic data typing, meaning that the type of a variable is controlled by the most recent assignment to it. PHP's arrays are a combination of dynamic arrays and hashes (associative arrays). The language includes a large number of predefined functions for manipulating arrays.

PHP allows simple access to HTML form data, so form processing is easy with PHP. PHP also provides support for many different database management systems. This versatility makes it an excellent language for building programs that need Web access to databases.

A subset of PHP is described in Chapter 9.

1.9.8 Overview of Ajax

Ajax, shorthand for *A*synchronous *Ja*vaScript + *X*ML, had been around for a few years in the early 2000s, but did not acquire its catchy name until 2005.[14] The idea of Ajax is relatively simple, but it results in a different way of viewing and building Web interactions. This new approach produces an enriched Web experience for those using a certain category of Web interactions.

14. Ajax was named by Jesse James Garrett, who has stated on numerous occasions that *Ajax* is shorthand, not an acronym. Thus, we spell it Ajax, not AJAX.

In a traditional (as opposed to Ajax) Web interaction, the user sends messages to the server either by clicking a link or a form's *Submit* button. After the link has been clicked or the form has been submitted, the client waits until the server responds with a new document. The entire browser display is then replaced by that of the new document. Complicated documents take a significant amount of time to be transmitted from the server to the client and more time to be rendered by the browser. In Web applications that require frequent interactions with the client and remain active for a significant amount of time, the delay in receiving and rendering a complete response document can be disruptive to the user.

In an Ajax Web application, there are two variations from the traditional Web interaction. First, the communication from the browser to the server is asynchronous; that is, the browser need not wait for the server to respond. Instead, the browser user can continue whatever he or she was doing while the server finds and transmits the requested document and the browser renders the new document. Second, the document provided by the server usually is only a relatively small part of the displayed document, and therefore it takes less time to be transmitted and rendered. These two changes can result in much faster interactions between the browser and the server.

The *x* in *Ajax*, from *XML*, is there because in some cases the data supplied by the server is in the form of an XML document, which provides the new data to be placed in the displayed document. However, in some cases the data is plain text, which even may be JavaScript code. It also can be HTML.

The goal of Ajax is to have Web-based applications become closer to desktop (client-resident) applications, in terms of the speed of interactions and the quality of the user experience. Wouldn't we all like our Web-based applications to be as responsive as our word processors?

Ajax is discussed in more depth in Chapter 10.

1.9.9 Overview of Servlets, JavaServer Pages, and JavaServer Faces

There are many computational tasks in a Web interaction that must occur on the server, such as processing order forms and accessing server-resident databases. A Java class called a *servlet* can be used for these applications. A servlet is a compiled Java class, an object of which is executed on the server system when requested by the HTML document being displayed by the browser. A servlet produces an HTML document as a response, some parts of which are static and are generated by simple output statements, while other parts are created dynamically when the servlet is called.

When an HTTP request is received by a Web server, it examines the request. If a servlet must be called, the Web server passes the request to the servlet processor, called a *servlet container*. The servlet container determines which servlet must be executed, makes sure that it is loaded, and calls it. As the servlet handles the request, it generates an HTML document as its response, which is returned to the server through the response object parameter.

Java can also be used as a server-side scripting language. An HTML document with embedded Java code is one form of JavaServer Pages (JSP). Built on top of servlets, JSP provides alternative ways of constructing dynamic Web

documents. JSP takes an opposite approach to that of servlets: Instead of embedding HTML in Java code that provides dynamic documents, code of some form is embedded in HTML documents to provide the dynamic parts of a document. These different forms of code make up the different approaches used by JSP. The basic capabilities of servlets and JSP are the same.

When requested by a browser, a JSP document is processed by a software system called a *JSP container*. Some JSP containers compile the document when it is loaded on the server; others compile it only when requested. The compilation process translates a JSP document into a servlet and then compiles the servlet. So, JSP is actually a simplified approach to writing servlets.

JavaServer Faces (JSF) adds another layer to the JSP technology. The most important contribution of JSF is an event-driven user interface model for Web applications. Client-generated events can be handled by server-side code with JSF.

Servlets, JSP, and JSF are discussed in Chapter 11.

1.9.10 Overview of Active Server Pages .NET

Active Server Pages .NET (ASP.NET) is a Microsoft framework for building server-side dynamic documents. ASP.NET documents are supported by programming code executed on the Web server. As discussed in Section 1.9.9, JSF uses Java to describe the dynamic generation of HTML documents, as well as to describe computations associated with user interactions with documents. ASP.NET provides an alternative to JSF, with two major differences: First, ASP.NET allows the server-side programming code to be written in any of the .NET languages.[15] Second, in ASP.NET all programming code is compiled, which allows it to execute much faster than interpreted code.

Every ASP.NET document is compiled into a class. From a programmer's point of view, developing dynamic Web documents (and the supporting code) in ASP.NET is similar to developing non-Web applications. Both involve defining classes based on library classes, implementing interfaces from a library, and calling methods defined in library classes. An application class uses, and interacts with, existing classes. In ASP.NET, this is exactly the same for Web applications: Web documents are designed by designing classes.

ASP.NET is discussed in Chapter 12.

1.9.11 Overview of Ruby

Ruby is an object-oriented interpreted scripting language designed by Yukihiro Matsumoto (a.k.a. Matz) in the early 1990s and released in 1996. Since then, it has continually evolved and its level of usage has grown. The original motivation for Ruby was the dissatisfaction of its designer with the earlier languages Perl and Python.

The primary characterizing feature of Ruby is that it is a pure object-oriented language, just as is Smalltalk. Every data value is an object and all operations are via method calls. The operators in Ruby are only syntactic mechanisms to

15. In most cases, it is done in either Visual Basic .NET or C#.

specify method calls for the corresponding operations. Because they are methods, many of the operators can be redefined by user programs. All classes, whether predefined or user defined, can have subclasses.

Both classes and objects in Ruby are dynamic in the sense that methods can be dynamically modified. This means that classes and objects can have different sets of methods at different times during execution. So, different instantiations of the same class can behave differently.

The syntax of Ruby is related to that of Eiffel and Ada. There is no need to declare variables, because dynamic typing is used. In fact, all variables are references and do not have types, although the objects they reference do.

Our interest in Ruby is based on Ruby's use with the Web development framework Rails (see Section 1.9.12). Rails was designed for use with Ruby, and it is Ruby's primary use in Web programming. Programming in Ruby is introduced in Chapter 15.

Ruby is culturally interesting because it is the first programming language designed in Japan that has achieved relatively widespread use outside that country.

1.9.12 Overview of Rails

Rails is a development framework for Web-based applications that access databases. A framework is a system in which much of the more-or-less standard software parts are furnished by the framework, so they need not be written by the applications developer. ASP.NET and JSF are also development frameworks for Web-based applications. Rails, whose more official name is *Ruby on Rails*, was developed by David Heinemeier Hansson in the early 2000s and was released to the public in July 2004. Since then, it has rapidly gained widespread interest and usage. Rails is based on the Model-View-Controller (MVC) architecture for applications, which clearly separates applications into three parts: presentation, data model, and program logic.

Rails applications are tightly bound to relational databases. Many Web applications are closely integrated with database access, so the Rails relational database framework is a widely applicable architecture.

Rails can be, and often is, used in conjunction with Ajax. Rails uses the JavaScript framework Prototype to support Ajax and interactions with the JavaScript model of the document being displayed by the browser. Rails also provides other support for developing Ajax, including producing visual effects.

Rails was designed to be used with Ruby and makes use of the strengths of that language. Furthermore, Rails is written in Ruby. Rails is discussed in Chapter 16.

Summary

The Internet began in the late 1960s as the ARPAnet, which was eventually replaced by NSFnet for nonmilitary users. NSFnet later became known as the Internet. There are now many millions of computers around the world that are connected to the Internet. Although much of the network control equipment is different and many kinds of computers are connected, all these connections are

made through the TCP/IP protocol, making them all appear, at least at the lowest level, the same to the network.

Two kinds of addresses are used on the Internet: IP addresses, which are four-part numbers, for computers; and fully qualified domain names, which are words separated by periods, for people. Fully qualified domain names are translated to IP addresses by name servers running DNS. A number of different information interchange protocols have been created, including `telnet`, `ftp`, and `mailto`.

The Web began in the late 1980s at CERN as a means for physicists to share the results of their work efficiently with colleagues at other locations. The fundamental idea of the Web is to transfer hypertext documents among computers by means of the HTTP protocol on the Internet.

Browsers request HTML documents from Web servers and display them for users. Web servers find and send requested documents to browsers. URLs are used to address all documents on the Internet; the specific protocol to be used is the first field of the URL. URLs also include the fully qualified domain name and a file path to the specific document on the server. The type of a document that is delivered by a Web server appears as an MIME specification in the first line of the document. Web sites can create their own experimental MIME types, provided that they also furnish a program that allows the browser to present the document's contents to the user.

HTTP is the standard protocol for Web communications. HTTP requests are sent over the Internet from browsers to Web servers; HTTP responses are sent from Web servers to browsers to fulfill those requests. The most commonly used HTTP requests are `GET` and `POST`.

Web programmers use several languages to create the documents that servers can provide to browsers. The most basic of these is HTML, the standard markup language for describing the content to be presented by browsers. Tools that can be used without specific knowledge of HTML are available to create HTML documents. A plug-in is a program that can be integrated with a word processor to make it possible to use the word processor to create HTML. A filter converts a document written in some other format to HTML. XML is a meta-markup language that provides a standard way to define new markup languages.

JavaScript is a client-side scripting language that can be embedded in an HTML document to describe simple computations. JavaScript code is interpreted by the browser on the client machine; it provides access to the elements of an HTML document, as well as the ability to change those elements dynamically.

Flash is a framework for building graphics, sound, and animation into HTML documents.

Ajax is an approach to building Web applications in which partial document requests are handled asynchronously. Ajax can significantly increase the speed of user interactions, so it is most useful for building systems that have frequent interactions.

PHP is the server-side equivalent of JavaScript. It is an interpreted language whose code is embedded in HTML documents. PHP is used primarily for form processing and database access from browsers.

Servlets are server-side Java programs that are used for form processing, database access, or building dynamic documents. JSP documents, which are

translated into servlets, are an alternative approach to building these applications. JSF is a development framework for specifying forms and their processing in JSP documents.

ASP.NET is a Web development framework. The code used in ASP.NET documents, which is executed on the server, can be written in any .NET programming language.

Ruby is an object-oriented scripting language that is introduced here primarily because of its use in Rails, a Web applications framework. Rails provides a significant part of the code required to build Web applications that access databases, allowing the developer to spend his or her time on the specifics of the application without dealing with the drudgery of the housekeeping details.

Review Questions

1.1 What was one of the fundamental requirements for the new national computer network proposed by the DoD in the 1960s?

1.2 What protocol is used by all computer connections to the Internet?

1.3 What is the form of an IP address?

1.4 Describe a fully qualified domain name.

1.5 What is the task of a DNS name server?

1.6 What is the purpose of `telnet`?

1.7 In the first proposal for the Web, what form of information was to be interchanged?

1.8 What is hypertext?

1.9 What kind of browser, introduced in 1993, led to a huge expansion of Web usage?

1.10 In what common situation is the document returned by a Web server created after the request is received?

1.11 What is the document root of a Web server?

1.12 What is a virtual document tree?

1.13 What is the server root of a Web server?

1.14 What is a virtual host?

1.15 What is a proxy server?

1.16 What does the `file` protocol specify?

1.17 How do partial paths to documents work in Web servers?

1.18 When a browser requests a directory without giving its name, what is the name of the file that is normally returned by the Web server?

1.19 What is the purpose of an MIME type specification in a request-response transaction between a browser and a server?

1.20 With what must a Web server furnish the browser when it returns a document with an experimental MIME type?

1.21 Describe the purposes of the five most commonly used HTTP methods.

1.22 What is the purpose of the `Accept` field in an HTTP request?

1.23 What response header field is most often required?

1.24 What important capability is lacking in a markup language?

1.25 What problem is addressed by using a public-key approach to encryption?

1.26 Is it practically possible to compute the private key associated with a given public key?

1.27 What is the difference between a virus and a worm?

1.28 What appears to motivate a hacker to create and disseminate a virus?

1.29 What is a plug-in?

1.30 What is a filter HTML converter?

1.31 Why must code generated by a filter often be modified manually before use?

1.32 What is the great advantage of XML over HTML for describing data?

1.33 How many different tags are predefined in an XML-based markup language?

1.34 What is the relationship between Java and JavaScript?

1.35 What are the most common applications of JavaScript?

1.36 Where is JavaScript most often interpreted, on the server or on the browser?

1.37 What is the primary use of Flash?

1.38 Where are Flash movies interpreted, on the server or on the browser?

1.39 Where are servlets executed, on the server or on the browser?

1.40 In what language are servlets written?

1.41 In what way are JSP documents the opposite of servlets?

1.42 What is the purpose of JSF?

1.43 What is the purpose of ASP.NET?

1.44 In what language is the code in an ASP.NET document usually written?

1.45 Where is PHP code interpreted, on the server or on the browser?

1.46 In what ways is PHP similar to JavaScript?

1.47 In what ways is Ruby more object oriented than Java?

1.48 In what country was Ruby developed?

1.49 What is the purpose of Rails?

1.50 For what particular kind of Web application was Rails designed?

1.51 Which programming languages are used in Ajax applications?

1.52 In what fundamental way does an Ajax Web application differ from a traditional Web application?

Exercises

1.1 For the following products, to what brand do you have access, what is its version number, and what is the latest available version?

a. Browser
b. Web server
c. JavaScript
d. PHP
e. Servlets
f. ASP.NET
g. Ruby
h. Rails

1.2 Search the Web for information on the history of the following technologies, and write a brief overview of those histories:

a. TCP/IP
b. SGML
c. HTML
d. ARPAnet
e. BITNET
f. XML
g. JavaScript
h. Flash
i. Servlets
j. JSP
k. JSF
l. Rails
m. Ajax

This page intentionally left blank

CHAPTER

2

Introduction to HTML/XHTML

This chapter introduces the most commonly used subset of the Hypertext Markup Language (HTML). Because of the simplicity of HTML, the discussion moves quickly. Although the eXtensible Hypertext Markup Language (XHTML) is no longer in the evolutionary line of HTML, the strictness of its syntax rules are valuable and their use is acceptable in HTML documents, so we describe and use the XHTML form of HTML. The chapter begins with a brief history of the evolution of HTML and XHTML, followed by a description of the form of tags

and the structure of an HTML document. Then, tags used to specify the presentation of text are discussed, including those for line breaks, paragraph breaks, headings, and block quotations, as well as tags for specifying the style and relative size of fonts. This discussion is followed by a description of the formats and uses of images in Web documents. Next, hypertext links are introduced. Three kinds of lists—ordered, unordered, and definition—are then covered. After that, the HTML tags and attributes used to specify tables are discussed. The next section of the chapter introduces forms, which provide the means to collect information from Web clients. Following this are brief sections that describe and illustrate the `audio`, the `video`, the organization elements, and the `time` element. Finally, the last section summarizes the syntactic differences between HTML and XHTML.

One good reference for information about HTML and XHTML is `http://www.w3schools.com`.

2.1 Origins and Evolution of HTML and XHTML

HTML is a markup language, which means it is used to mark parts of documents to indicate how they should appear, in print or on a display.[1] HTML is defined with the meta-markup language,[2] Standard Generalized Markup Language (SGML), which is an International Standards Organization (ISO) standard notation for describing information-formatting languages.[3] The original intent of HTML was different from those of other such languages, which dictate all the presentation details of text, such as font style, size, and color. Rather, HTML was designed to specify document structure at a higher and more abstract level, necessary because HTML-specified documents had to be displayable on a variety of computer systems using different browsers.

The appearance of style sheets that could be used with HTML in the late 1990s advanced its capabilities closer to those of other information-formatting languages by providing ways to include the specification of presentation details. These specifications are introduced in Chapter 3.

2.1.1 Versions of HTML and XHTML

The original version of HTML was designed in conjunction with the structure of the Web and the first browser at Conseil Européen pour la Recherche Nucléaire (CERN), or European Laboratory for Particle Physics. Use of the Web began its meteoric rise in 1993 with the release of Mosaic, the first graphical Web browser. Not long after Mosaic was commercialized and marketed by Netscape, the company founded by the designers of Mosaic, Microsoft began developing its

1. The term markup comes from the publishing industry, where in the past documents were marked up by hand to indicate to a typesetter how the document should appear in print.

2. A meta-markup language is a language for defining markup languages.

3. Not all information-formatting languages are based on SGML; for example, PostScript and LaTeX are not.

browser, Internet Explorer (IE). The release of IE in 1995 marked the beginning of a four-year marketing competition between Netscape and Microsoft. During this time, both companies worked feverously to develop their own extensions to HTML in an attempt to gain market advantage. Naturally, this competition led to incompatible versions of HTML, both between the two developers and also between older and newer releases from the same company. These differences made it a serious challenge to Web content providers to design single HTML documents that could be viewed by the different browsers.

In late 1994, Tim Berners-Lee, who developed the initial version of HTML, started the World Wide Web Consortium (W3C), whose primary purpose was to develop and distribute standards for Web technologies, starting with HTML. The first HTML standard, HTML 2.0, was released in late 1995. It was followed by HTML 3.2 in early 1997. Up to that point, W3C was trying to catch up with the browser makers, and HTML 3.2 was really just a reflection of the then-current features that had been developed by Netscape and Microsoft. Fortunately, since 1998 the evolution of HTML has been dominated by W3C, in part because Netscape gradually withdrew from its browser competition with Microsoft. There are now several different organizations that produce and distribute browsers, all of which except Microsoft, at least until recently, followed relatively closely the HTML standards produced by W3C. The HTML 4.0 specification was published in late 1997. The 4.01 version of HTML, which is the latest completed standard version, was approved by W3C in late 1999.

The appearance of style sheets (in 1997) that could be used with HTML made some features of earlier versions of HTML obsolete. These features, as well as some others, have been *deprecated*, meaning that they will be dropped from HTML in the future. Deprecating a feature is a warning to users to stop using it because it will not be supported forever. Although even the latest releases of browsers still support the deprecated parts of HTML, we do not include descriptions of them in this book.

There are two fundamental problems with HTML 4.01. First, it specifies loose syntax rules. These permit many variations of document forms which may be interpreted differently by different browsers. As a result of the lax syntax rules, HTML documents naturally have been haphazardly written. By some estimates, 99 percent of the HTML documents served on the Web contain errors. The second problem with HTML 4.01 is that its specification does not define how a user agent (an HTML processor, most often a browser) is to recover when erroneous code is encountered. Consequently, every browser uses its own variation of error recovery.

eXtensible Markup Language (XML)[4] is an alternative to SGML. It was designed to describe data and it has strict syntax rules. The XML specification requires that XML processors not accept XML documents with any errors.

XHTML 1.0, which was approved in early 2000, is a redefinition of HTML 4.01 using XML. XHTML 1.0 is actually three standards: Strict, Transitional, and Frameset. The Strict standard requires all the syntax rules of XHTML 1.0 be followed. The Transitional standard allows deprecated features of XHTML 1.0 to be included. The Frameset standard allows the collection of frame elements

4. XML (eXtensible Markup Language) is the topic of Chapter 7.

and attributes to be included, although they have been deprecated. The XHTML 1.1 standard was recommended by W3C in May 2001. This standard, primarily a modularization of XHTML 1.0, drops some of the features of its predecessor—most notably, frames.

The XHTML 1.0 specification addresses one of the problems of HTML 4.01 by providing complete rules stating what is and what is not syntactically acceptable. Furthermore, it specifies that XHTML documents must be served with the `application/xhmtl+xml` MIME type. This means that user agents were required to halt interpretation of an XHTML document when the first syntactic error was found, as is the case with XML documents. Because this was a drastic response to the error-ignoring nature of HTML 4.01, the XHTML 1.0 specification included Appendix C, which allowed XHTML documents to be served as HTML, that is, with the `text/html` MIME type, which allows the continuation of the practice of user agents ignoring syntax errors.

The XHTML 1.1 specification, published in 2001, included only relatively minor additions to XHTML 1.0, but eliminated the Appendix C loophole. This meant that W3C officially specified that XHTML 1.1 documents had to be served as `application/xhtml+xml` MIME type and that user agents were required to reject all syntactically incorrect documents. This is oft termed "draconian error handling."

Although the value of the consistent and coherent syntax rules of XHTML were widely recognized and accepted, the draconian error handling was not. It was generally agreed by developers and providers of markup documents that browser users should not be given error messages due to the syntax errors found in documents they were attempting to view. The result was that XHTML 1.1 documents were still served with the `text/html` MIME type and browsers continued to use forgiving HTML parsers.

W3C apparently ignored these issues and forged ahead with the development of the next version of XHTML, 2.0. The design of XHTML 2.0 further promoted its ultimate demise by not requiring it to be backward compatible with either HTML 4.01 or XHTML 1.1.

In reaction to the XHTML 1.1 specification and the development of XHTML 2.0, a new organization was formed in 2004 by browser vendors, Web development companies, and some W3C members. This group, which became known as the Web Hypertext Application Technology (WHAT) Working Group, began working on the next version of HTML, which was to be based on HTML 4.01, rather than XHTML 1.1. Among the goals of the new version of HTML were the following: backward compatibility with HTML 4.01, error handling that is clearly defined in the specification, and users would not be exposed to document syntax errors. Initially, W3C had no interest in being involved in this new project, which would compete with XHTML 2.0.

The first results of the WHAT Working Group were WebForm 2.0, which extended HTML forms, Web Applications 1.0, which was a new version of HTML, and an algorithm for user agent error handling.

After several years of separate work, W3C on XHTML 2.0 and the WHAT Working Group on a new version of HTML, the head of W3C, Berners-Lee, made the momentous decision in 2006 that W3C would begin working with the

WHAT Working Group. In 2009, W3C decided to adopt the HTML development and drop the XHTML 2.0 development effort. The first action of W3C was to rename Web Applications 1.0 as HTML5.

2.1.2 HTML versus XHTML

Until 2010, many Web developers used XHTML to gain the advantages of stricter syntax rules, standard formats, and validation, but their documents were served as `text/html` and browsers used HTML parsers. Other developers stubbornly clung to HTML. They are now enthusiastically climbing on the HTML5 bandwagon. Meanwhile, the XHTML crowd is disappointed and confused at the realization that the W3C effort to coerce developers into using the more strict syntactic rules of XHTML to produce documents less prone to errors is over—W3C had capitulated, apparently willing to accept life in a world populated by syntactically sloppy documents.

In previous editions of this book, we followed the W3C lead in strict pursuance of clear and syntactically correct documents by presenting XHTML 1.0 Strict example documents and using and encouraging validation to ensure adherence to that standard. Now, the major browser vendors have all implemented at least some of the more important new features of HTML5. This makes use of the W3C XHTML 1.0 Strict validator impossible.

There are strong reasons that one should use XHTML. One of the most compelling is that quality and consistency in any endeavor, be it electrical wiring, software development, or Web document development, rely on standards. HTML has few syntactic rules, and HTML processors (e.g., browsers) do not enforce the rules it does have. Therefore, HTML authors have a high degree of freedom to use their own syntactic preferences to create documents. Because of this freedom, HTML documents lack consistency, both in low-level syntax and in overall structure. By contrast, XHTML has strict syntactic rules that impose a consistent structure on all XHTML documents. Furthermore, the fact that there are a large number of poorly structured HTML documents on the Web is a poor excuse for generating more.

There are two issues in choosing between HTML and XHTML: First, one must decide whether the additional discipline required to use XHTML is worth the gain in document clarity and uniformity in display across a variety of browsers. Second, one must decide whether the possibility of validation afforded by authoring XHTML documents is worth the trouble.

In this edition, we follow a compromise approach. Of course we must discuss some of the new exciting features of HTML5. At some point in the future they will be widely used. However, we also are firm believers in standards and strict syntax rules. So, our compromise is to write and promote HTML5, but also the syntax rules of XHTML 1.0 Strict, all of which are legal in HTML5. In Section 2.5.3, we describe how to validate that documents follow the XHTML 1.0 Strict syntax rules.

The remainder of this chapter provides an introduction to the most commonly used tags and attributes of HTML, as well as some of the new tags and attributes of HTML5. We present this material using the XHTML 1.0 Strict syntax, but point out the HTML form when it is different.

2.2 Basic Syntax

The fundamental syntactic units of HTML are called *tags*. In general, tags are used to specify categories of content. For each kind of tag, a browser has default presentation specifications for the specified content. The syntax of a tag is the tag's name surrounded by angle brackets (< and >). Tag names must be written in all lowercase letters.[5] Most tags appear in pairs: an opening tag and a closing tag. The name of the closing tag, when one is required, is the name of its corresponding opening tag with a slash attached to the beginning. For example, if the tag name is `p`, its closing tag is `</p>`. Whatever appears between a tag and its closing tag is the *content* of the tag. A browser display of an HTML document shows the content of all the document's tags; it is the information the document is meant to portray. Not all tags can have content.

The opening tag and its closing tag together specify a container for the content they enclose. The container and its content together are called an *element*. For example, consider the following element:

```
<p> This is simple stuff. </p>
```

Attributes, which are used to specify alternative meanings of a tag, are written between the opening tag name and its right-angle bracket. They are specified in keyword form, which means that the attribute's name is followed by an equals sign and the attribute's value. Attribute names, like tag names, are written in lowercase letters. Attribute values must be delimited by double quotes.[6] There will be numerous examples of attributes in the remainder of this chapter.

Comments in programs increase the readability of those programs. Comments in HTML serve the same purpose. They are written in HTML in the following form:

```
<!--  anything except two adjacent dashes -->
```

Browsers ignore HTML comments—they are for people only. Comments can be spread over as many lines as are needed. For example, you could have the following comment:

```
<!--  PetesHome.html
      This document describes the home document of
      Pete's Pickles
      -->
```

Informational comments are as important in HTML documents as they are in programs. Documents sometimes have lengthy sequences of lines of markup that together produce some part of the display. If such a sequence is not preceded by a comment that states its purpose, a document reader may have difficulty determining why the sequence is there. As is the case in programs, commenting every line is both tedious and counterproductive. However, comments that precede logical collections of lines of markup are essential to making a document (or a program) more understandable.

---

5. In HTML, tag names and attribute names can be written in any mixture of uppercase and lowercase letters.

6. In HTML, some attribute values, for example, numbers, need not be quoted.

Besides comments, several other kinds of text that are ignored by browsers may appear in an HTML document. Browsers ignore all unrecognized tags. They also ignore line breaks. Line breaks that show up in the displayed content can be specified, but only with tags designed for that purpose. The same is true for multiple spaces and tabs.

When introduced to HTML, programmers find it a bit frustrating. In a program, the statements specify exactly what the computer must do. HTML tags are treated more like suggestions to the browser. If a reserved word is misspelled in a program, the error is usually detected by the language implementation system and the program is not executed. However, a misspelled tag name usually results in the tag being ignored by the browser, with no indication to the user that anything has been left out. Browsers are even allowed to ignore tags that they recognize. Furthermore, the user can configure his or her browser to react to specific tags in different ways.

2.3 Standard HTML Document Structure

The first line of every HTML document is a `DOCTYPE` command, which specifies the particular SGML Document-Type Definition (DTD) with which the document complies. For HTML, this declaration is simply the following:

```
<!DOCTYPE html>
```

An HTML document must include the four tags `<html>`, `<head>`, `<title>`, and `<body>`.[7] The `<html>` tag identifies the root element of the document. So, HTML documents always have an `<html>` tag following the `DOCTYPE` command and they always end with the closing `html` tag, `</html>`. The `html` element includes an attribute, `lang`, which specifies the language in which the document is written, as shown in the following element:

```
<html lang = "en">
```

In this example, the language is specified as `"en"`, which means English.

An HTML document consists of two parts, the *head* and the *body*. The head element provides information about the document but does not provide its content. The head element always contains two simple elements, a title element and a meta element. The meta element is used to provide additional information about a document. It has no content; rather, all the information provided is specified with attributes. At a minimum, the meta tag specifies the character set used to write the document. The most popular international character set used for the Web is the 8-bit Unicode Transformation Format (UTF-8). This character set uses from 1 byte to 6 bytes to represent a character, but is backward compatible with the ASCII character set. This compatibility is accomplished by having all the single-byte characters in UTF-8 correspond to the ASCII characters. Following is the necessary meta element:[8]

```
<meta charset = "utf-8" />
```

7. This is another XHTML rule. Documents that do not include all of these are acceptable in HTML.

8. This meta element is required by HTML, but not for XHTML.

The slash at the end of this element indicates that it has no closing tag—it is a combined opening and closing tag.

The content of the title element is displayed by the browser at the top of its display window, usually in the browser window's title bar.

The body of a document provides its content.

Following is a skeletal document that illustrates the basic structure:

```
<!DOCTYPE html>
<!-- File name and document purpose -->
<html lang = "en">
  <head>
    <title> A title for the document </title>
    <meta charset = "utf-8" />
    ...
  </head>
  <body>
  ...
  </body>
</html>
```

Notice that we have used a simple formatting pattern for the document, similar to what is often used for programs. Whenever an element is nested inside a preceding element, the nested element is indented. In this book, we will indent nested elements two spaces, although there is nothing special about that number. As is the case with programs, the indentation pattern is used to enhance readability.

2.4 Basic Text Markup

This section describes how the text content of an HTML document body can be formatted with HTML tags. By *formatting*, we mean layout and some presentation details. For now, we will ignore the other kinds of content that can appear in an HTML document.

2.4.1 Paragraphs

Text is often organized into paragraphs in the body of a document.[9] The XHTML standard does not allow text to be placed directly in a document body.[10] Instead, text is often placed in the content of a paragraph element, the name of which is `p`. In displaying text, the browser puts as many words as will fit on the lines in the browser window. The browser supplies a line break at the end of each line. As stated in Section 2.2, line breaks embedded in text are ignored by the browser.

9. In Section 2.12, several elements are introduced that provide more detailed ways of organizing text (and other information) in a document.

10. HTML allows text to appear virtually anywhere in a document.

For example, the following paragraph might[11] be displayed by a browser as shown in Figure 2.1:

```
<p>
   Mary had
a
   little lamb, its fleece was white as snow. And
 everywhere that
  Mary went, the lamb
 was sure to go.
</p>
```

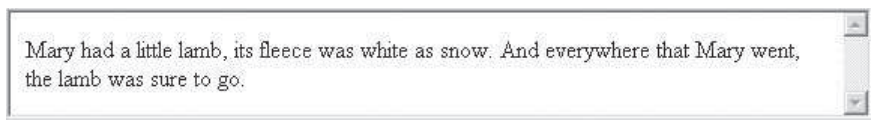


Figure 2.1 Filling lines

Notice that multiple spaces in the source paragraph element are replaced by single spaces in Figure 2.1.

The following is our first example of a complete HTML document:

```
<!DOCTYPE html>
<!-- greet.html
     A trivial document
     -->
<html lang = "en">
  <head>
    <title> Our first document </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <p>
      Greetings from your Webmaster!
    </p>
  </body>
</html>
```

Figure 2.2 shows a browser display of `greet.html`.

11. We say "might" because the width of the display that the browser uses determines how many words will fit on a line.

Figure 2.2 Display of `greet.html`

If a paragraph tag appears at a position other than the beginning of the line, the browser breaks the current line and inserts a blank line. For example, the following line would be displayed as shown in Figure 2.3:

```
<p> Mary had a little lamb, </p> <p> its fleece was white
      as snow. </p>
```

Figure 2.3 The paragraph element

2.4.2 Line Breaks

Sometimes text requires an explicit line break without the preceding blank line. This is exactly what the break tag does. The break tag, which is named `br`, differs syntactically from the paragraph tag in that it can have no content and therefore has no closing tag (because a closing tag would serve no purpose). In XHTML, the break tag is specified with the following:

```
<br />
```

The slash indicates that the tag is both an opening and closing tag. The space before the slash represents the absent content.[12]

Consider the following markup:

```
<p>
Mary had a little lamb, <br />
   its fleece was white as snow.
</p>
```

This markup would be displayed as shown in Figure 2.4.

Figure 2.4 Line breaks

12. Some older browsers have trouble with the tag `<br/>` but not with `<br />`. In HTML, the break tag can be written as <br>, without a closing tag or slash.

2.4.3 Preserving White Space

Sometimes it is desirable to preserve the white space in text—that is, to prevent the browser from eliminating multiple spaces and ignoring embedded line breaks. This can be specified with the `pre` tag—for example,

```
<pre>
Mary
    had a
        little
            lamb
</pre>
```

This markup would be displayed as shown in Figure 2.5. Notice that the content of the `pre` element is shown in monospace, rather than in the default font. The `pre` element not only keeps line breaks from the source, it also preserves the character and line spacing.

Figure 2.5 The `pre` element

A `pre` element can contain virtually any other tags, except those that cause a paragraph break, such as paragraph elements. Because other markups can appear in a `pre` element, special characters in text content, such as `<`, must be avoided. In Section 2.4.7, we will describe how to safely include such characters as character entities.

2.4.4 Headings

Text is often separated into sections in documents by beginning each section with a heading. Larger sections sometimes have headings that appear more prominent than headings for sections nested inside them. In HTML, there are six levels of headings, specified by the tags `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, and `<h6>`, where `<h1>` specifies the highest-level heading. Headings are usually displayed in a boldface font whose default font size depends on the number in the heading tag. On most browsers, `<h1>`, `<h2>`, and `<h3>` use font sizes that are larger than that of the default size of text, `<h4>` uses the default size, and `<h5>` and `<h6>` use smaller sizes. The heading tags always break the current line, so their content always appears on a new line. Browsers usually insert some vertical space before and after all the headings.

The following example illustrates the use of headings:

```
<!DOCTYPE html>
<!-- headings.html
     An example to illustrate headings
     -->
<html lang = "en">
  <head>
    <title> Headings </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h1> Aidan's Airplanes (h1) </h1>
    <h2> The best in used airplanes (h2) </h2>
    <h3> "We've got them by the hangarful" (h3) </h3>
    <h4> We're the guys to see for a good used airplane (h4) </h4>
    <h5> We offer great prices on great planes (h5) </h5>
    <h6> No returns, no guarantees, no refunds,
         all sales are final! (h6) </h6>
  </body>
</html>
```

Figure 2.6 shows a browser display of `headings.html`.

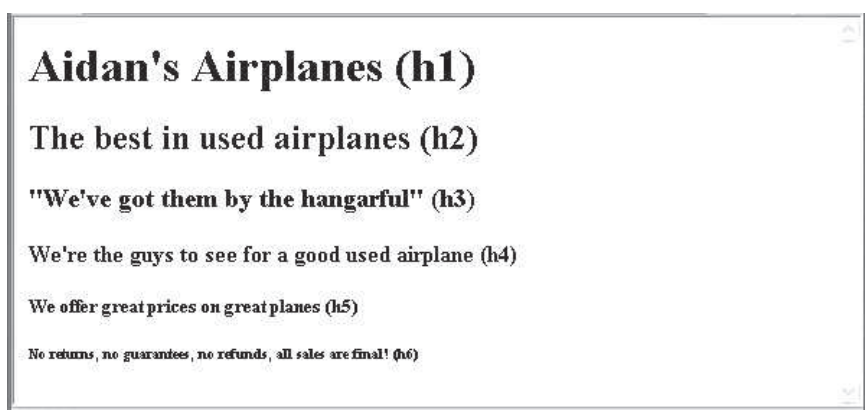


Figure 2.6 Display of `headings.html`

2.4.5 Block Quotations

Sometimes we want a block of text to be set off from the normal flow of text in a document. In many cases, such a block is a long quotation. The `<blockquote>` tag is designed for this situation. Browser designers are allowed to determine how the content of `<blockquote>` can be made to look different from the surrounding text. However, in most cases the block of text simply is indented on both sides. Consider the following example document:

```
<!DOCTYPE html>
<!-- blockquote.html
     An example to illustrate a blockquote
     -->
<html lang = "en">
  <head>
    <title> Blockquotes </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <p>
      Abraham Lincoln is generally regarded as one of the greatest
      presidents of the United States. His most famous speech was
      delivered in Gettysburg, Pennsylvania, during the Civil War.
      This speech began with
    </p>
    <blockquote>
      <p>
        "Fourscore and seven years ago our fathers brought forth on
        this continent, a new nation, conceived in Liberty, and
        dedicated to the proposition that all men are created equal.
      </p>
      <p>
        Now we are engaged in a great civil war, testing whether
        that nation or any nation so conceived and so dedicated,
        can long endure."
      </p>
    </blockquote>
    <p>
      Whatever one's opinion of Lincoln, no one can deny the
      enormous and lasting effect he had on the United States.
    </p>
  </body>
</html>
```

Figure 2.7 shows a browser display of `blockquote.html`.

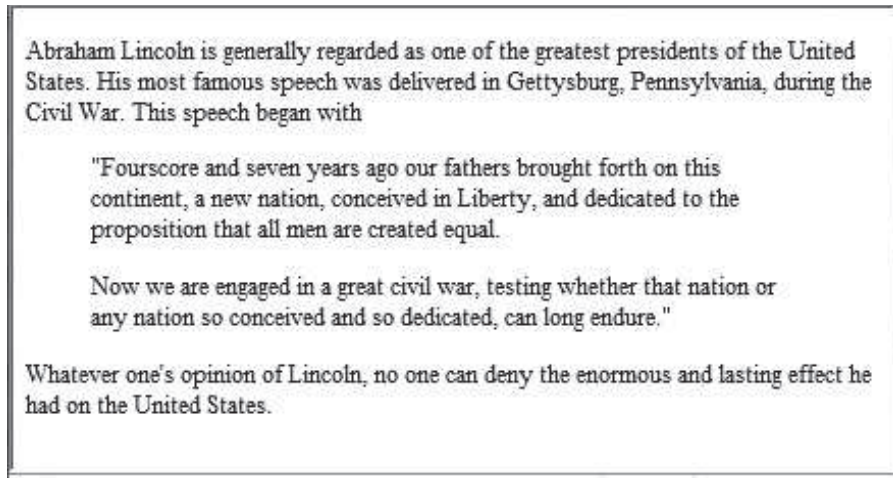


Figure 2.7 Display of `blockquote.html`

2.4.6 Font Styles and Sizes

Early Web designers used a collection of tags to set font styles and sizes. For example, `<i>` specified italics and `<b>` specified bold. Since the advent of cascading style sheets (see Chapter 3), the use of these tags has become passé. There are a few tags for fonts that are still in widespread use, called *content-based style tags*. These tags are called content based because they indicate the style of the text that appears in their content. Three of the most commonly used content-based tags are the emphasis tag, the strong tag, and the code tag.

The emphasis element, `em`, specifies that its textual content is special and should be displayed in some way that indicates this distinctiveness. Most browsers use italics for such content.

The strong element, `strong`, is like the emphasis tag, but more so. Browsers often set the content of strong elements in bold.

The code element, `code`, is used to specify a monospace font, usually for program code. For example, consider the following element:

```
<code> cost = quantity * price </code>
```

This markup would be displayed as shown in Figure 2.8.

Figure 2.8 The `code` element

Subscript and superscript characters can be specified by the `sub` and `sup` elements, respectively. These are not content-based tags. For example,

```
X<sub>2</sub><sup>3</sup> + y<sub>1</sub><sup>2</sup>
```

would be displayed as shown in Figure 2.9.

Figure 2.9 The `sub` and `sup` elements

Content-based style tags are not affected by `<blockquote>`, except when there is a conflict. For example, if the text content of `<blockquote>` is normally set in italics by the browser and a part of that text is made the content of an `em` element, the `em` element would have no effect.

Elements are categorized as being either block or inline. The content of an *inline* element appears on the current line (if it fits). So, an inline element does not implicitly include a line break. One exception is `br`, which is an inline element, but its entire purpose is to insert a line break in the content. A *block* element breaks the current line so that its content appears on a new line. The heading and block quote elements are block elements, whereas `em` and `strong` are inline elements. In XHTML, block elements cannot appear in the content of inline elements. Therefore, a block element can never be nested directly in an inline element. Also, inline elements and text cannot be directly nested in body or form elements. Only block elements can be so nested. That is why the example `greet.html` has the text content of its body nested in a paragraph element.

2.4.7 Character Entities

HTML provides a collection of special characters that are sometimes needed in a document but cannot be typed as themselves. In some cases, these characters are used in HTML in some special way—for example, > and < are used to delimit element names. In other cases, the characters do not appear on keyboards, such as the small raised circle that represents *degrees* in a reference to temperature. Finally, there is the nonbreaking space, which browsers regard as a hard space—they do not squeeze them out, as they do other multiple spaces. These special characters are defined as *entities*, which are codes for the characters. The browser replaces an entity in a document by its associated character. Table 2.1 lists some of the most commonly used entities.

Table 2.1 Some commonly used entities

Character	Entity	Meaning
&	`amp;`	Ampersand
<	`<`	Is less than
>	`>`	Is greater than
"	`&quot;`	Double quote
'	`&apos;`	Single quote (apostrophe)
¼	`&frac14;`	One-quarter
½	`&frac12;`	One-half
¾	`&frac34;`	Three-quarters
°	`&deg;`	Degree
(space)	` `	Nonbreaking space
©	`&copy;`	Copyright
€	`&euro;`	Euro

For example, the following text:

```
The price is < 10 Euros
```

could be placed in the content of a document as given:

```
The price is &lt; 10 &euro;
```

2.4.8 Horizontal Rules

Two parts of a document can be separated from each other by placing a horizontal line between them, thereby making the document easier to read. Such lines are called *horizontal rules*, and the block element that creates them is `hr`. The `hr` element causes a line break (ending the current line) and places a line across the screen. The browser chooses the thickness, length, and horizontal placement of the line. Typically, browsers display lines that are three pixels thick.

Because `hr` has no content, it is specified with `<hr />`.[13]

2.4.9 Other Uses of the `meta` Element

The `meta` element, which we have been using to specify the character set used in documents, is often used to provide information about the document, primarily for search engines. The two attributes that are used for this are `name` and

13. The horizontal rule tag can be written in HTML as `<hr>`.

content. The user makes up a name as the value of the name attribute and specifies information through the content attribute. One commonly chosen name is keywords; the value of the content attribute associated with the keywords is that which the author of a document believes characterizes his or her document. An example is as follows:

```
<meta name = "keywords" content = "binary trees,
linked lists, stacks" />
```

Web search engines use the information provided with the meta element to categorize Web documents in their indices. So, if the author of a document seeks widespread exposure for the document, one or more meta elements are included to ensure that it will be found by Web search engines. For example, if an entire book were published as a Web document, it might have the following meta elements:

```
<meta name = "Title" content = "Don Quixote" />
<meta name = "Author" content = "Miguel Cervantes" />
<meta  name = "keywords" content = "novel,
 Spanish literature, groundbreaking work" />
```

2.5 Images

The inclusion of images in a document can dramatically enhance its appearance, although images slow the document-download process. The file in which the image is stored is specified in an element. The image is inserted into the display of the document by the browser.

2.5.1 Image Formats

The two most common methods of representing images are the Graphic Interchange Format (GIF, pronounced like the first syllable of *jiffy*) and the Joint Photographic Experts Group (JPEG, pronounced *jay-peg*) format. Most contemporary browsers can render images in either of these two formats. Files in both formats are compressed to reduce storage needs and allow faster transfer over the Internet.

The GIF was developed by the CompuServe network service provider for the specific purpose of transmitting images. It uses 8-bit color representations for pixels, allowing a pixel to have 256 different colors. If you are not familiar with color representations, this format may seem to be entirely adequate. However, the color displays on most contemporary computers can display a much larger number of colors. Files containing GIF images use the .gif (or .GIF) extension on their names. GIF images can be made to appear transparent.

The JPEG format uses 24-bit color representations for pixels, which allows JPEG images to include more than 16 million different colors. Files that store JPEG images use the .jpg (or .JPG or .jpeg) extension on their names. The compression algorithm used by JPEG is better at shrinking an image than the one used by GIF. This compression process actually loses some of the color accuracy of the image, but because there is so much to begin with, the loss is

rarely discernible by the user. Because of this powerful compression process, even though a JPEG image has much more color information than a GIF image of the same subject, the JPEG image can be smaller than the GIF image. Hence, JPEG images are often preferred to GIF images. One disadvantage of JPEG is that it does not support transparency.

The third image format is now gaining popularity: Portable Network Graphics (PNG, pronounced *ping*). PNG was designed in 1996 as a free replacement for GIF after the patent owner for GIF, Unisys, suggested that the company might begin charging royalties for documents that included GIF images.[14] Actually, PNG is a good replacement for both GIF and JPEG because it has the best characteristics of each (the possibility of transparency, as provided by GIF, and the same large number of colors as JPEG). One drawback of PNG is that, because its compression algorithm does not sacrifice picture clarity, its images require more space than comparable JPEG images.[15] Support for PNG in the earlier IE browsers was unacceptably poor, which kept many developers from using PNG. However, the IE9+ browsers have adequate support for it. Information on PNG can be found at `www.w3.org/Graphics/PNG`.

2.5.2 The Image Element

The `img` element is an inline element that specifies an image that is to appear in a document. This element never has content, so it includes a slash in its opening tag and has no closing tag. In its simplest form, the image element includes two attributes: `src`, which specifies the file containing the image; and `alt`, which specifies text to be displayed when it is not possible to display the image. If the file is in the same directory as the HTML file of the document, the value of `src` is just the image's file name. In many cases, image files are stored in a subdirectory of the directory where the HTML files are stored. For example, the image files might be stored in a subdirectory named `images`. If the image file's name is `stars.jpg` and the image file is stored in the `images` subdirectory, the value of `src` would be as follows:

```
"images/stars.jpg"
```

Some seriously aged browsers are not capable of displaying images. When such a browser finds an `<img />` tag, it simply ignores it, possibly leaving the user confused by the text in the neighborhood of where the image was supposed to be. Also, graphical browsers, which *are* capable of displaying images, may have image downloading disabled by the browser user. This is done when the Internet connection is slow and the user chooses not to wait for images to download. It is also done by visually impaired users. In any case, it is helpful to have some text displayed in place of the ignored image. For these reasons, the `alt` attribute is required by XHTML. The value of `alt` is displayed whenever the browser cannot or has been instructed not to display the image.

14. The patent expired in the United States in 2003.

15. Space is not the direct issue; download time, which depends on file size, is the real issue.

Two optional attributes of `img`—`width` and `height`—can be included to specify (in pixels) the size of the rectangle for the image. These attributes can be used to scale the size of the image (i.e., to make it larger or smaller). Care must be taken to ensure that the image is not distorted in the resizing. For example, if the image is square, the `width` and `height` attribute values must be kept equal when they are changed. For example, if the image in the file `C210.jpg` is square and we want it to fit in a 200-pixel square, we could use the following:

```
<img src = "c210.jpg"  height = "200"  width = "200"
     alt = "Picture of a Cessna 210" />
```

A percentage value can be given for the width of an image. This specifies the percentage of the width of the display that will be occupied by the image. For example, `width = "50%"` will result in the image filling half of the width of the display. If no height is given and a percentage value is given for the width, the browser will scale the height to the width, maintaining the original shape of the image.

The following example extends the airplane ad document to include information about a specific airplane and its image:

```
<!DOCTYPE html>
<!-- image.html
     An example to illustrate an image
     -->
<html lang = "en">
  <head>
    <title> Images </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h1> Aidan's Airplanes </h1>
    <h2> The best in used airplanes </h2>
    <h3> "We've got them by the hangarful" </h3>
    <h2> Special of the month </h2>
    <p>
      1960 Cessna 210 <br />
      577 hours since major engine overhaul<br />
      1022 hours since prop overhaul <br /><br />
      <img src =  "c210new.jpg"  alt = "Picture of a Cessna 210" />
      <br />
      Buy this fine airplane today at a remarkably low price
      <br />
      Call 999-555-1111 today!
    </p>
  </body>
</html>
```

Figure 2.10 shows a browser display of `image.html`.

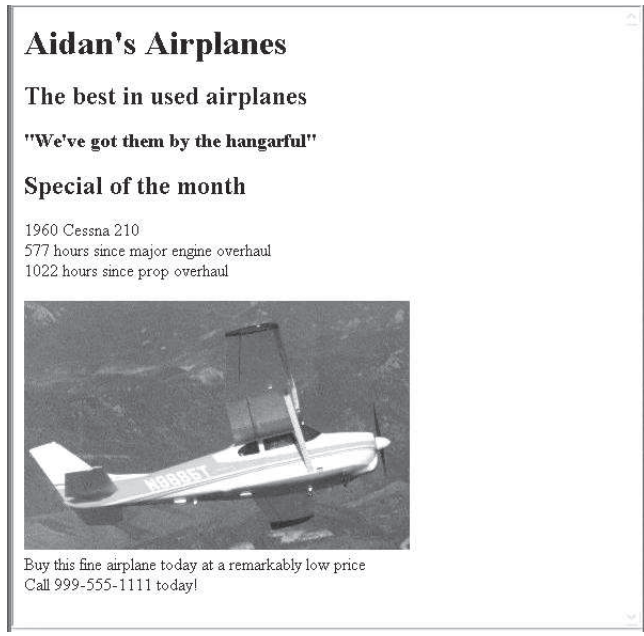


Figure 2.10 Display of `image.html`

2.5.3 XHTML Document Validation

For validation of the XHTML syntax of a document, we must make three temporary changes to the document. First, we replace the HTML document type with the following XHTML 1.0 Strict standard document type:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml1-strict.dtd">
```

A complete explanation of this `DOCTYPE` command requires more effort, both to write and to read, than is justified at this stage of our introduction to HTML.

Second, we must add the following attribute to the `html` element:

```
xmlns = "http://www.w3.org/1999/xhtml"
```

This specifies the namespace used for the document, XHTML.

Third, we must comment out the `meta` element, as in the following:

```
<!-- <meta charset = "utf-8" /> -->
```

The modified version of `image.html` is as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml1-strict.dtd">
```

```
<!-- image.html
     An example to illustrate an image
     -->
<html lang = "en"  xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Images </title>
 <!--   <meta charset = "utf-8" /> -->
  </head>
  <body>
    <h1> Aidan's Airplanes </h1>
    <h2> The best in used airplanes </h2>
    <h3> "We've got them by the hangarful" </h3>
    <hr />
    <h2> Special of the month </h2>
    <p>
      1960 Cessna 210 <br />
      577 hours since major engine overhaul<br />
      1022 hours since prop overhaul <br /><br />
      <img src =  "c210new.jpg"  alt = "Picture of a Cessna
                   210" />
      <br />
       Buy this fine airplane today at a remarkably low price
      <br />
      Call 999-555-1111 today!
    </p>
  </body>
</html>
```

After making these three temporary changes to a document, it can be validated with the Total Validator Tool, which provides a convenient way to validate documents against XHTML 1.0 Strict standard.[16] The validator program is available at `http://totalvalidator.com`. Figure 2.11 shows a browser display of the Total Validator Tool, after it has been downloaded and installed.

The file name of the document to be validated is entered (including the path name) in the *Starting web page* text box, or found by browsing. When the *Validate* button is clicked, the validation system is run on the specified file.

Figure 2.12 shows a browser display of the document returned by the validation system for `image.html`.

One of the most common errors made in crafting HTML documents that use the XHTML syntax is putting text or elements where they do not belong. For example, putting text directly into the body element is invalid.

16. W3C also has an XHTML validator, available at `http://validator.w3.org`. This validator will produce errors for all HTML5 markups that are not valid in XHTML 1.0.

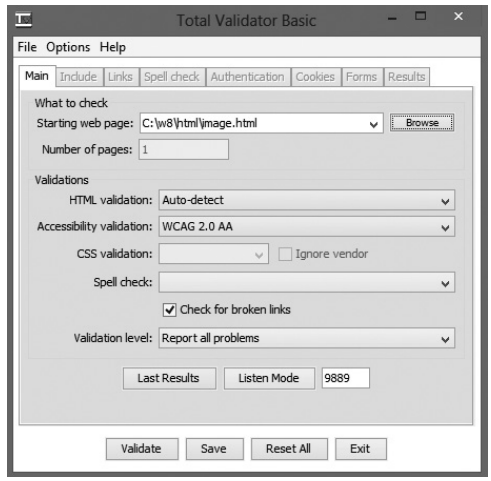


Figure 2.11 Display of the Total Validator Tool screen

The Total Validator Tool is a valuable tool for producing documents that adhere to XHTML 1.0 Strict standards. Validation ensures that a document is syntactically correct and more likely to display similarly on a variety of browsers. After validation, the document easily can be returned to its HTML form.

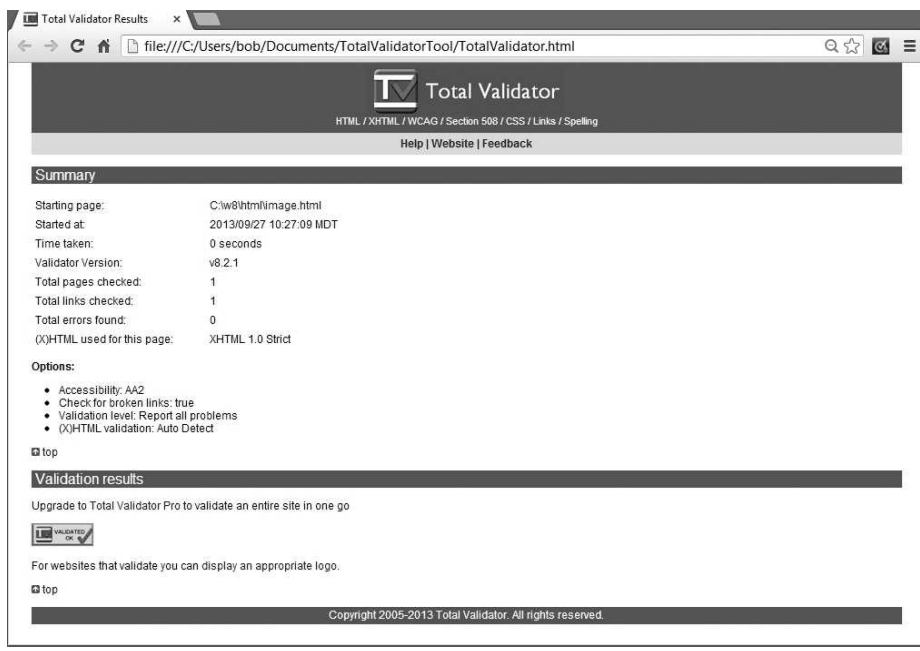


Figure 2.12 Total Validation Tool output for `image.html`

2.6 Hypertext Links

A hypertext link in an HTML document, which we simply call a *link* here, acts as a pointer to some particular place in some Web resource. That resource can be an HTML document anywhere on the Web, or it may be the document currently being displayed. Without links, Web documents would be boring and tedious to read. There would be no convenient way for the browser user to get from one document to a logically related document. Most Web sites consist of many different documents, all logically linked together. Therefore, links are essential to building an interesting Web site.

2.6.1 Links

A link that points to a different resource specifies the address of that resource. Such an address might be a file name, a directory path and a file name, or a complete URL. If a link points to a specific place in any document other than its beginning, that place somehow must be marked. Specifying such places is discussed in Section 2.6.2.

Links are specified in an attribute of an anchor element, `a`, which is an inline element. The anchor element that specifies a link is called the *source* of that link. The document whose address is specified in a link is called the *target* of that link.

As is the case with many elements, the anchor element can include many different attributes. However, for creating links, only one attribute is required: `href` (an acronym for *h*ypertext *ref*erence). The value assigned to `href` specifies the target of the link. If the target is in another document in the same directory, it is just the document's file name. If the target document is in some other directory, the UNIX path name conventions are used. For example, suppose we have a document in the `public_html` directory (which stores servable documents) that is linked to a document named `C210data.html`, which is stored in the `airplanes` subdirectory of `public_html`. The value of the `href` attribute of the anchor element would be `"airplanes/c210data.html"`. This is the relative method of document addressing, which means the address is relative to the address of the document currently being displayed. Absolute file addresses could be used in which the entire path name for the linked-to file is given. However, relative links are easier to maintain, especially if a hierarchy of HTML files must be moved. If the document is on some other machine (not the server providing the document that includes the link), obviously relative addressing cannot be used.

The content of an anchor element, which becomes the clickable link the user sees, is usually text or an image, and cannot be another anchor element. Links are usually implicitly rendered in a different color than that of the surrounding text. Sometimes they are also underlined. When the mouse cursor is placed over the content of the anchor element and the left mouse button is pressed, the link is taken by the browser. If the target is in a different document, that document is loaded and displayed, replacing the currently displayed document. If the target is in the current document, the document is scrolled by the browser to display the part of the document in which the target of the link is defined.

As an example of a link to the top of a different document, consider the following document, which adds a link to the document displayed in Figure 2.10:

```
<!DOCTYPE html>
<!-- link.html
     An example to illustrate a link
     -->
<html lang = "en">
  <head>
    <title> A link </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h1> Aidan's Airplanes </h1>
    <h2> The best in used airplanes </h2>
    <h3> "We've got them by the hangarful" </h3>
    <h2> Special of the month </h2>
    <p>
      1960 Cessna 210 <br />
      <a href = "C210data.html"> Information on the Cessna 210 </a>
    </p>
  </body>
</html>
```

In this case, the target is a complete document that is stored in the same directory as the HTML document. Figure 2.13 shows a browser display of `link.html`. When the link shown in Figure 2.13 is clicked, the browser displays the screen shown in Figure 2.14.

Figure 2.13 Display of `link.html`

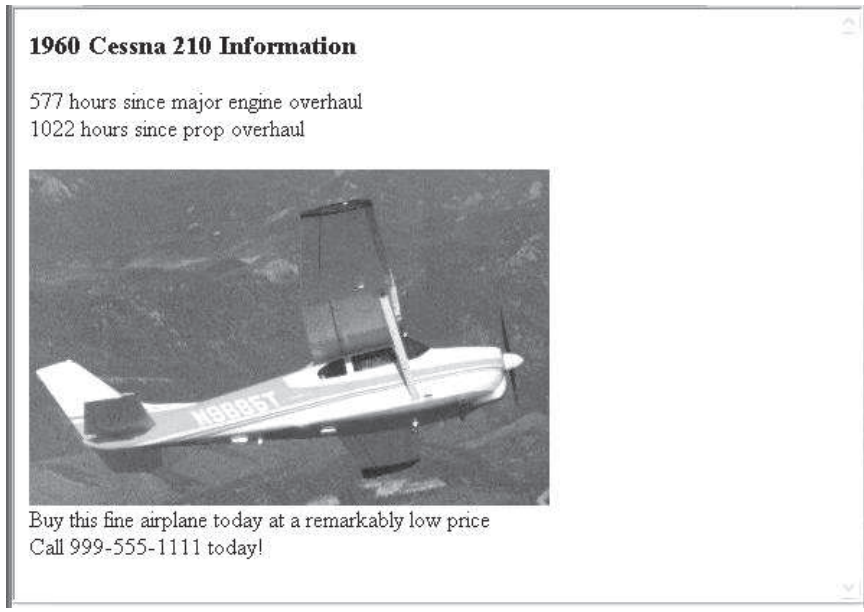


Figure 2.14 Following the link from `link.html`

Links can include images in their content, in which case the browser displays the image together with the textual link:

```
<a href = "c210data.html" >
  <img src = "small-airplane.jpg"
       alt = "An image of a small airplane" />
    Information on the Cessna 210
</a>
```

An image itself can be an effective link (the content of an anchor element). For example, an image of a small house can be used for the link to the home document of a site. The content of the anchor element for such a link is just the image element.

2.6.2 Targets within Documents

If the target of a link is not at the beginning of a document, it must be some element within the document, in which case there must be some means of specifying that target element. If the target element has an `id` attribute, that value can be used to specify the target. Consider the following example:

```
<h2 id = "avionics"> Avionics </h2>
```

Nearly all elements can include an `id` attribute. The value of an `id` attribute must be unique within the document.

If the target is in the same document as the link, the target is specified in the `href` attribute value by preceding the `id` value with a pound sign (#), as in the following example:

```
<a href = "#avionics"> What about avionics? </a>
```

When the `What about avionics?` link is taken, the browser moves the display so that the element whose `id` is `avionics` is at the top.

When the target is an element in another document, the value of that element's `id` is specified at the end of the URL, separated by a pound sign (#), as in the following example:

```
<a href = "aidan1.html#avionics"> Avionics </a>
```

2.6.3 Using Links

One common use of links to parts of the same document is to provide a table of contents in which each entry has a link. This technique provides a convenient way for the user to get to the various parts of the document simply and quickly. Such a table of contents is implemented as a stylized list of links by using the list specification capabilities of HTML, which are discussed in Section 2.7.

Links exemplify the true spirit of hypertext. The reader can click on links to learn more about a particular subtopic of interest and then return to the location of the link. Designing links requires some care because they can be annoying if the designer tries too hard to convince the user to take them. For example, making them stand out too much from the surrounding text can be distracting. A link should blend into the surrounding text as much as possible so that reading the document without clicking any of the links is easy and natural.

2.7 Lists

We frequently make and use lists in daily life—for example, to-do lists and grocery lists. Likewise, both printed and displayed information is littered with lists. HTML provides simple and effective ways to specify lists in documents. The primary list types supported are those with which most people are already familiar: unordered lists such as grocery lists and ordered lists such as the assembly instructions for a new desktop computer. Definition lists can also be defined. The tags used to specify unordered, ordered, and definition lists are described in this section.

2.7.1 Unordered Lists

The `ul` element, which is a block element, creates an unordered list. Each item in a list is specified with an `li` element (li is an acronym for *l*ist *i*tem). Any elements can appear in a list item, including nested lists. When displayed, each list item

is implicitly preceded by a bullet. The following document, `unordered.html`, illustrates an unordered element:

```
<!DOCTYPE html>
<!-- unordered.html
     An example to illustrate an unordered list
     -->
<html lang = "en">
  <head>
    <title> Unordered list </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h3> Some Common Single-Engine Aircraft </h3>
    <ul>
      <li> Cessna Skyhawk </li>
      <li> Beechcraft Bonanza </li>
      <li> Piper Cherokee </li>
    </ul>
  </body>
</html>
```

Figure 2.15 shows a browser display of `unordered.html`.

Figure 2.15 Display of `unordered.html`

2.7.2 Ordered Lists

Ordered lists are lists in which the order of items is important. This orderedness of a list is shown in its display by the implicit attachment of a sequential value to the beginning of each item. The default sequential values are Arabic numerals, beginning with `1`.

An ordered list is created within the block element `ol`. The items are specified and displayed just as are those in unordered lists, except that the items in an

ordered list are preceded by sequential values instead of bullets. Consider the following example of an ordered list:

```
<!DOCTYPE html>
<!-- ordered.html
     An example to illustrate an ordered list
     -->
<html lang = "en">
  <head>
    <title> Ordered list </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h3> Cessna 210 Engine Starting Instructions </h3>
    <ol>
      <li> Set mixture to rich </li>
      <li> Set propeller to high RPM </li>
      <li> Set ignition switch to "BOTH" </li>
      <li> Set auxiliary fuel pump switch to "LOW PRIME" </li>
      <li> When fuel pressure reaches 2 to 2.5 PSI, push
           starter button </li>
    </ol>
  </body>
</html>
```

Figure 2.16 shows a browser display of `ordered.html`.

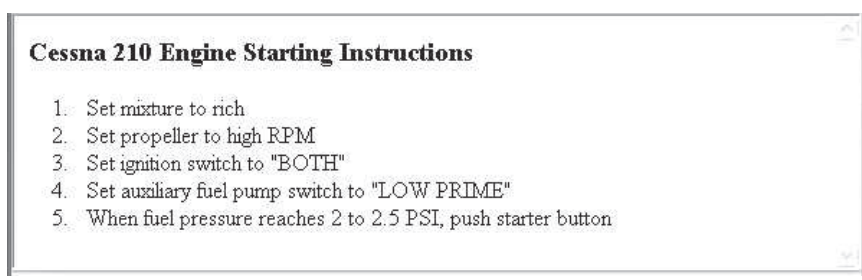


Figure 2.16 Display of `ordered.html`

As noted previously, lists can be nested. However, a list cannot be directly nested; that is, an `<ol>` tag cannot immediately follow an `<ol>` tag. Rather, the nested list must be the content of an `li` element. The following example illustrates nested ordered lists:

```
<!DOCTYPE html>
<!-- nested_lists.html
     An example to illustrate nested lists
     -->
<html lang = "en">
  <head>
    <title> Nested lists </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h3> Aircraft Types </h3>
    <ol>
      <li> General Aviation (piston-driven engines)
        <ol>
          <li> Single-Engine Aircraft
            <ol>
              <li> Tail wheel </li>
              <li> Tricycle </li>
            </ol> <br />
          </li>
          <li> Dual-Engine Aircraft
            <ol>
              <li> Wing-mounted engines </li>
              <li> Push-pull fuselage-mounted engines </li>
            </ol>
          </li>
        </ol> <br />
      </li>
      <li> Commercial Aviation (jet engines)
        <ol>
          <li> Dual-Engine
            <ol>
              <li> Wing-mounted engines </li>
              <li> Fuselage-mounted engines </li>
            </ol> <br />
          </li>
          <li> Tri-Engine
            <ol>
              <li> Third engine in vertical stabilizer </li>
              <li> Third engine in fuselage </li>
            </ol>
          </li>
        </ol>
      </li>
    </ol>
  </body>
</html>
```

Figure 2.17 shows a browser display of `nested_lists.html`.

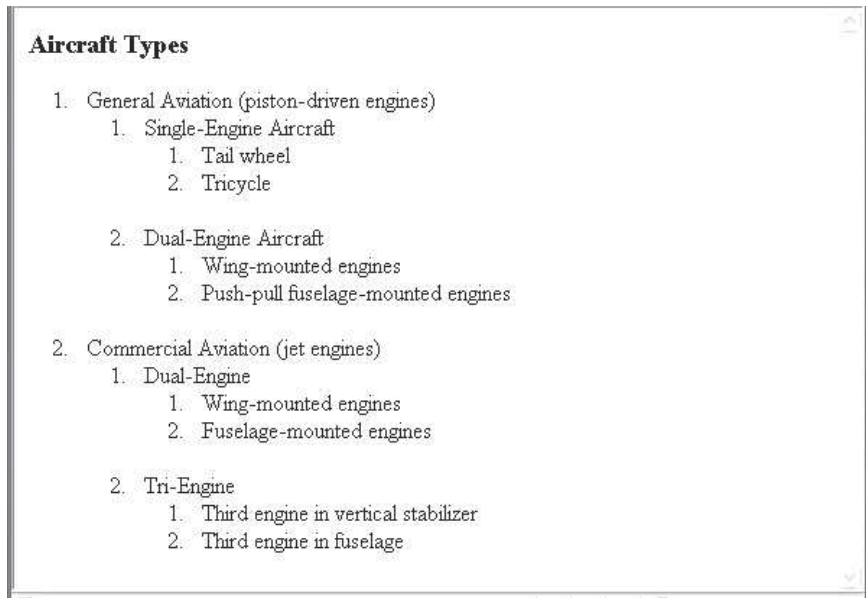


Figure 2.17 Display of `nested_lists.html`

One problem with the nested lists shown in Figure 2.17 is that all three levels use the same sequence values. Chapter 3 describes how style sheets can be used to specify different kinds of sequence values for different lists.

The `nested_lists.html` example uses nested ordered lists. There are no restrictions on list nesting, provided that the nesting is not direct. For example, ordered lists can be nested in unordered lists and vice versa.

2.7.3 Definition Lists

As the name implies, definition lists are used to specify lists of terms and their definitions, as in glossaries. A definition list is given as the content of a `dl` element, which is a block element. Each term to be defined in the definition list is given as the content of a `dt` element. The definitions themselves are specified as the content of `dd` elements. The defined terms of a definition list are usually displayed in the left margin; the definitions are usually shown indented on the line or lines following the terms, as in the following example:

```
<!DOCTYPE html>
<!-- definition.html
     An example to illustrate definition lists
     -->
<html lang = "en">
```

```
  <head>
    <title> Definition lists </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h3> Single-Engine Cessna Airplanes </h3>
    <dl>
      <dt> 152 </dt>
      <dd> Two-place trainer </dd>
      <dt> 172 </dt>
      <dd> Smaller four-place airplane </dd>
      <dt> 182 </dt>
      <dd> Larger four-place airplane </dd>
      <dt> 210 </dt>
      <dd> Six-place airplane - high performance </dd>
    </dl>
  </body>
</html>
```

Figure 2.18 shows a browser display of `definition.html`.

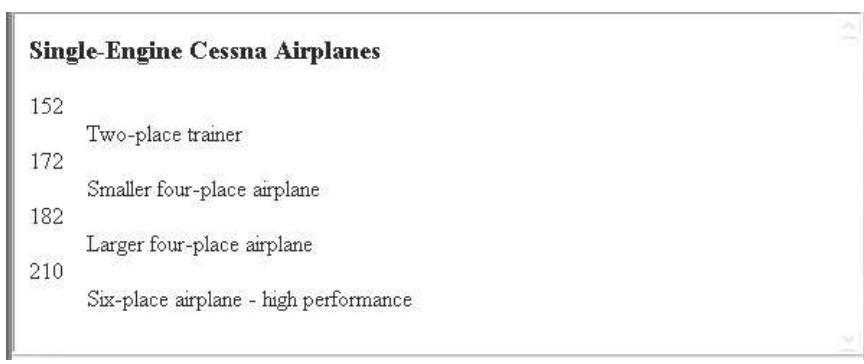


Figure 2.18 Display of `definition.html`

2.8 Tables

Tables are common fixtures in printed documents, books, and, of course, Web documents. They provide an effective way of presenting many kinds of information.

A table is a matrix of cells. The cells in the top row often contain column labels, those in the leftmost column often contain row labels, and most of the remaining cells contain the data of the table. The content of a cell can be almost any document element, including text, a heading, a horizontal rule, an image, or a nested table.

2.8.1 Basic Table Tags

In HTML 4.01 and XHTML 1.0, the `table` element has an attribute, `border`, that specifies the styles of the border around the outside of the table and that of the rules, or lines that separate the cells of a table. This attribute is not included in HTML5. The styles of the border and rules in a table are specified in HTML5 with style sheets, as we will describe in Chapter 3. So, the tables in this chapter will have neither borders nor rules.

In most cases, a displayed table is preceded by a title, given as the content of a `caption` element, which can immediately follow the `<table>` tag. The cells of a table are specified one row at a time. Each row of the table is specified with a row element, `tr`. Within each row, the row label is specified by the table heading element, `th`. Although the `th` element has *heading* in its name, we call these elements *labels* to avoid confusion with headings created with the `hx` elements. Each data cell of a row is specified with a table data element, `td`. The first row of a table usually has the its column labels. For example, if a table has three data columns and their column labels are, respectively, `Apple`, `Orange`, and `Screwdriver`, the first row can be specified by the following:

```
<tr>
  <th> Apple </th>
  <th> Orange </th>
  <th> Screwdriver </th>
</tr>
```

Each data row of a table is specified with a heading tag and one data tag for each data column. For example, the first data row for our work-in-progress table might be as follows:

```
<tr>
  <th> Breakfast </th>
  <td> 0 </td>
  <td> 1 </td>
  <td> 0 </td>
</tr>
```

In tables that have both row and column labels, the upper-left corner cell is often empty. This empty cell is specified with a table header tag that includes no content (either `<th></th>` or just `<th />`).

The following document describes the whole table:

```
<!DOCTYPE html>
<!-- table.html
     An example of a simple table
     -->
<html lang = "en">
```

```
  <head>
    <title> A simple table </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <table>
      <caption> Fruit Juice Drinks </caption>
      <tr>
        <th> </th>
        <th> Apple </th>
        <th> Orange </th>
        <th> Screwdriver </th>
      </tr>
      <tr>
        <th> Breakfast </th>
        <td> 0 </td>
        <td> 1 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Lunch </th>
        <td> 1 </td>
        <td> 0 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Dinner </th>
        <td> 0 </td>
        <td> 0 </td>
        <td> 1 </td>
      </tr>
    </table>
  </body>
</html>
```

Figure 2.19 shows a browser display of this table.

Fruit Juice Drinks

	Apple	Orange	Screwdriver
Breakfast	0	1	0
Lunch	1	0	0
Dinner	0	0	1

Figure 2.19 Display of `table.html`

2.8.2 The `rowspan` and `colspan` Attributes

In many cases, tables have multiple levels of row or column labels in which one label covers two or more secondary labels. For example, consider the display of a partial table shown in Figure 2.20. In this table, the upper-level label **Fruit Juice Drinks** spans the three lower-level label cells. Multiple-level labels can be specified with the `rowspan` and `colspan` attributes.

Figure 2.20 Two levels of column labels

The `colspan` attribute specification in a table header or table data tag tells the browser to make the cell as wide as the specified number of rows below it in the table. For the previous example, the following markup could be used:

```
<tr>
  <th colspan = "3"> Fruit Juice Drinks </th>
</tr>
<tr>
  <th> Apple </th>
  <th> Orange </th>
  <th> Screwdriver </th>
</tr>
```

If there are fewer cells in the rows above or below the spanning cell than the `colspan` attribute specifies, the browser stretches the spanning cell over the number of cells that populate the column in the table.[17] The `rowspan` attribute of the table heading and table data tags does for rows what `colspan` does for columns.

A table that has two levels of column labels and also has row labels must have an empty upper-left corner cell that spans both the multiple rows of column labels and the multiple columns. Such a cell is specified by including both `rowspan` and `colspan` attributes. Consider the following table specification, which is a minor modification of the previous table:

```
<!DOCTYPE html>
<!-- cell_span.html
     An example to illustrate rowspan and colspan
     -->
<html lang = "en">
```

17. Some browsers add empty row cells to allow the specified span to occur.

```
  <head>
    <title> Rowspan and colspan </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <table>
      <caption> Fruit Juice Drinks and Meals </caption>
      <tr>
        <td rowspan = "2"> </td>
        <th colspan = "3"> Fruit Juice Drinks </th>
      </tr>
      <tr>
        <th> Apple </th>
        <th> Orange </th>
        <th> Screwdriver </th>
      </tr>
      <tr>
        <th> Breakfast </th>
        <td> 0 </td>
        <td> 1 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Lunch </th>
        <td> 1 </td>
        <td> 0 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Dinner </th>
        <td> 0 </td>
        <td> 0 </td>
        <td> 1 </td>
      </tr>
    </table>
  </body>
</html>
```

Figure 2.21 shows a browser display of `cell_span.html`.

2.8.3 Table Sections

Tables naturally occur in two and sometimes in three parts: header, body, and footer (not all tables have a natural footer). These three parts can be respectively denoted in HTML with the `thead`, `tbody`, and `tfoot` elements. The header

Fruit Juice Drinks and Meals

	Fruit Juice Drinks		
	Apple	**Orange**	**Screwdriver**
Breakfast	0	1	0
Lunch	1	0	0
Dinner	0	0	1

Figure 2.21 Display of `cell_span.html`: multiple-labeled columns and labeled rows

includes the column labels, regardless of the number of levels in those labels. The body includes the data of the table, including the row labels. The footer, when it appears, sometimes has the column labels repeated after the body. In some tables, the footer contains totals for the columns of data above. A table can have multiple body sections, in which case the browser may separate them with horizontal lines that are thicker than the rule lines within a body section.

2.8.4 Uses of Tables

During the late 1990s a widespread trend evolved among Web document developers to use tables for whole document layout. There were several reasons behind this trend, including the following: At that time, Cascading Style Sheets (CSS) was not uniformly supported by the major browsers. Web document developers were not yet familiar with CSS. There was no widespread understanding of the advantages of the division of document design into the use of HTML for semantics and CSS for presentation. There was an explosion of demand for Web document designers during the dotcom boom of the late 1990s that led to a significant infusion of designers with little background in Web design. Many of them found it easier to use tables than CSS for document layout. Finally, Web design tools of the time encouraged the use of tables for general layout (by using tables themselves).

When the dotcom expansion collapsed in 2001, many of those drawn into the business of Web document design in the late 1990s departed for other areas of endeavor. One result of this was a rise in the average skill level of those who remained in the business of developing Web documents. Also, knowledge of CSS and its advantages grew. This environment produced a new trend in the use of tables—tableless layout. The disadvantages of the widespread use of tables for general layout were recognized. Among these is the large proliferation of tags in documents, many of them meaningless table cell tags. In many cases, table cells consisted of single-pixel transparent GIF images with explicit width and height used to align elements in a document. One result of this proliferation of tags was many unnecessarily large documents, which took large amounts of time to download, which in turn slowed the overall operation of the Internet. This unnecessary complexity of documents also required additional effort for maintenance.

Using CSS, rather than tables, to design document layout results in smaller and less complex documents. The smaller size of documents speeds up their download and raises the overall performance of the Internet. The use of CSS also lowers the cost of maintaining documents.

Even when tableless design is embraced, there are still many situations that can make good use of tables. Any time a natural table of information must be part of a document, an HTML table should be used for it. However, the use of tables for general layout of elements in a document should be avoided.

2.9 Forms

The most common way for a user to communicate information from a Web browser to the server is through a form. Modeled on the paper forms that people frequently are required to fill out, forms can be described in HTML and displayed by the browser. HTML provides elements to generate the commonly used objects on a screen form. These objects are called *controls* or *widgets* or *components*. There are controls for single-line and multiple-line text collection, checkboxes, radio buttons, and menus, among others. All control elements are inline elements. Most controls are used to gather information from the user in the form of either text or button selections. Each control can have a value, usually given through user input. Together, the values of all the controls (that have values) in a form are called the *form data*. Every form whose data is to be processed on the server requires a *Submit* button (see Section 2.9.5). When the user clicks the *Submit* button, the form data is encoded and sent to the Web server. Form processing is discussed in three subsequent chapters (Chapters 9, 11, and 12).

2.9.1 The `form` Element

All the controls of a form appear in the content of a `form` element, which is a block element, can have several different attributes, only one of which, `action`, is required.[18] The `action` attribute specifies the URL of the application on the Web server that is to be called when the user clicks the *Submit* button. In this chapter, our examples of form elements will not have corresponding application programs, so the value of their `action` attributes will be the empty string (`""`).

The `method` attribute of `form` specifies which technique, `get` or `post`, will be used to pass the form data to the server. The default is `get`, so if no `method` attribute is given in the `<form>` tag, `get` will be used. The alternative technique is `post`. In both techniques, the form data is coded into a text string when the user clicks the *Submit* button. This text string is often called the *query string*.[19]

18. Actually, the action attribute is not required by HTML. However, it is required by XHTML, so we will include it in our examples.

19. The query string has an assignment statement for each control that has a data value, with the name of the control as its left side and its value as its right side. These assignment statements are separated by ampersands (&).

When the `get` method is used, the browser attaches the query string to the URL of the HTTP request, so the form data is transmitted to the server together with the URL. The browser inserts a question mark at the end of the actual URL just before the first character of the query string so that the server can easily find the beginning of the query string. The `get` method can also be used to pass parameters to the server when forms are not involved. (This cannot be done with `post`.) One last advantage of `get` is that a site bookmark can include specific form values, which makes it more specific than one with only the URL. One disadvantage of the `get` method is that some servers place a limit on the length of the URL string and truncate any characters past the limit. So, if the form has more than a few controls, `get` is not a good choice. Because the form values are displayed by the browser, `get` should not be used if sensitive information such as passwords or credit card numbers is included in the form data.

When the `post` method is used, the query string is passed by some other method to the form-processing program. There is no length limitation for the query string with the `post` method, so, obviously, it is the better choice when there are more than a few controls in the form. There are also some security concerns with `get` that are not a problem with `post`.

2.9.2 The `input` Element

Many of the commonly used controls are specified with the inline element `input`, including those for text, passwords, checkboxes, radio buttons, plain buttons, ranges of numbers, URLs, electronic mail addresses, reset, submit, and image. There is also a hidden control, which is used in Chapter 12. The text, password, checkboxes, and radio controls are discussed in this section. The action buttons are discussed in Section 2.9.5.

The `type` attribute, which specifies the particular kind of control, is required in the `input` element. All the previously listed controls except reset and submit also require a `name` attribute. The values of the name attributes are included in the form data that is sent to the server. They are used by the server form processing software to find the specific component values in the form data. The controls for checkboxes and radio buttons require a `value` attribute, which initializes the value of the control. The values of these controls are placed in the form data that is sent to the server when the *Submit* button is clicked.

Because forms are processed on the server, which requires the form components to have name attributes, we will always include them in the components in our examples. In many cases, controls are also referenced in code on the client, primarily for client-side validation. Client code often references controls through their `id` attribute values. Therefore, it is common to include both `name` and `id` attributes on form control elements.

A text control, which we usually refer to as a text box, creates a horizontal box into which the user can type text. Text boxes are used to gather information from the user, such as the user's name and address. The default size of a text box is often

20 characters. Because the default size can vary among browsers, it is a good idea to include a size on each text box. This is done with the `size` attribute of `input`. If the user types more characters than will fit in the box, the box is scrolled. If you do not want the box to be scrolled, you can include the `maxlength` attribute to specify the maximum number of characters that the browser will accept in the box. Any additional characters are ignored. As an example of a text box, consider the following:

```
<form action = "">
  <p>
    <input type = "text"  name = "theName"  size = "25" />
    ...
  </p>
</form>
```

Suppose the user typed the following line:

```
Alfred Paul von Frickenburger
```

The text box would collect the whole string, but the string would be scrolled to the right, leaving the following shown in the box:

```
ed Paul von Frickenburger
```

The left end of the line would be part of the value of `theName`, even though it does not appear in the box. The ends of the line can be viewed in the box by moving the cursor off the ends of the box.

Notice that controls cannot appear directly in the form content—they must be placed in some block container, such as a paragraph. This is because neither text nor inline tags can appear directly in a form element and `input` is an inline element.[20]

Now consider a similar text box that includes a `maxlength` attribute:

```
<form action = "">
  <p>
    <input type = "text"  name = "theName"  size = "25"
           maxlength = "25" />
    ...
  </p>
</form>
```

If the user typed the same name as in the previous example, the resulting value of the `theName` text box would be as follows:

```
Alfred Paul von Frickenbu
```

No matter what was typed after the `u` in that person's last name, the value of `theName` would be as shown.

20. This restriction is for XHTML, not HTML.

If the contents of a text box should not be displayed when they are entered by the user, a password control can be used as follows:

```
<input type = "password"  name = "myPassword"
       size = "10" maxlength = "10" />
```

In this case, regardless of what characters are typed into the password control, only bullets or asterisks are displayed by the browser.

There are no restrictions on the characters that can be typed into a text box. So, the string `"?!34,:"` could be entered into a text box meant for names. Therefore, the entered contents of text boxes nearly always must be validated, either on the browser or on the server to which the form data is passed for processing, or on both. Validation is done on the client to avoid wasting time by sending invalid data to the server. It is also done on the server because client-side validation can be subverted by unscrupulous users.

Text boxes, as well as most other control elements, should be labeled. Labeling could be done simply by inserting text into the appropriate places in the form:

```
Phone: <input type = "text"  name = "thePhone" />
```

This markup effectively labels the text box, but there are several ways the labeling could be better. For one thing, there is no connection between the label and the control. Therefore, they could become separated in maintenance changes to the document. A control and its label can be connected by putting both of them in the content of a label element, as in the following:

```
<label> Phone: <input type = "text"  name = "thePhone" />
</label>
```

Now the text box and its label are encapsulated together. There are several other benefits of this approach to labeling controls. First, browsers often render the text content of a label element differently to make it stand out. Second, if the text content of a label element is selected, the cursor is implicitly moved to the control in the content of the label. This feature is an aid to new Web users. Third, the text content of a label element can be rendered by a speech synthesizer on the client machine when the content of the label element is selected. This feature can be a great aid to a user with a visual disability.

Checkbox and radio controls are used to collect multiple-choice input from the user. A checkbox control is a single button that is either on or off (checked or not). If a checkbox button is on, the value associated with the name of the button is the string assigned to its `value` attribute. A checkbox button does not contribute to the form data if it is not checked. Every checkbox button requires a `name` attribute and a `value` attribute in its `<input>` tag. For form processing on the server, the name identifies the button and the value is its value (if the button is checked). The attribute `checked`, which is assigned the value `checked`, specifies that the checkbox button is initially on. In many cases, checkboxes appear in lists, with each one in the list having the same name, thereby forming a checkbox group. Checkbox elements should appear in label elements, for the same reasons that text boxes should. The following example illustrates a checkbox:

```
<!DOCTYPE html>
<!-- checkbox.html
     An example to illustrate a checkbox
     -->
<html lang = "en">
  <head>
    <title> Checkboxes </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <p>
      Grocery Checklist
    </p>
    <form action = "">
      <p>
        <label> <input type = "checkbox"  name = "groceries"
               value = "milk"  checked = "checked" /> Milk </label>
        <label> <input type = "checkbox"  name = "groceries"
               value = "bread" /> Bread </label>
        <label> <input type = "checkbox"  name = "groceries"
               value = "eggs" /> Eggs </label>
      </p>
    </form>
  </body>
</html>
```

Figure 2.22 shows a browser display of `checkbox.html`.

Figure 2.22 Display of `checkbox.html`

If the user does not turn on any of the checkbox buttons in our example, `milk` will be the value for `groceries` in the form data. If the `milk` checkbox is left on and the `eggs` checkbox is also turned on by the user, the values of `groceries` in the form data would be `milk` and `eggs`.

Radio buttons are closely related to checkbox buttons. The difference between a group of radio buttons and a group of checkboxes is that only one radio button can be on or pressed at any time. Every time a radio button is pressed, the button

in the group that was previously on is turned off. Radio buttons are named after the mechanical push buttons on the radios of cars of the 1950s—when you pushed one button on such a radio, the previously pushed button was mechanically forced out. The `type` value for radio buttons is "`radio`". All radio buttons in a group must have the `name` attribute set in the `<input>` tag, and all radio buttons in a group must have the same name value. A radio button definition may specify which button is to be initially in the pressed, or on, state. This specification is indicated by including the `checked` attribute, set to the value `checked`, in the `<input>` tag of the button's definition. The following example illustrates radio buttons:

```
<!DOCTYPE html>
<!-- radio.html
     An example to illustrate radio buttons
     -->
<html lang = "en">
  <head>
    <title> Radio </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <p>
      Age Category
    </p>
    <form action = "">
      <p>
        <label><input type = "radio"  name = "age"
               value = "under20" checked = "checked" />
               0-19 </label>
        <label><input type = "radio"  name = "age"
               value = "20-35" /> 20-35 </label>
        <label><input type = "radio"  name = "age"
               value = "36-50" /> 36-50 </label>
        <label><input type = "radio"  name = "age"
               value = "over50" /> Over 50 </label>
      </p>
    </form>
  </body>
</html>
```

Figure 2.23 shows a browser display of `radio.html`.

A plain button has the type `button`. Plain buttons are used to cause an action, which is written in JavaScript, similar to an event handler, as described in Chapter 5.

The `url` and `email` values for the type attribute are different than other features of HTML described in this book in the sense that they are not yet fully

Figure 2.23 Display of `radio.html`

supported by the three most popular browsers. However, they also do not cause any problems if used.

The `url` value for the type attribute of an `input` element is used when the value of the input is a URL. For example, we could have the following:

```
<input type = "url" id = "myUrl"  name = "myUrl" >
```

The only difference between this element and a text type element is that the browser is supposed to check to determine whether the input could possibly be a valid URL. This check is to make sure the input includes a colon and that the colon is both preceded and followed by at least one character.

Current FX browsers require that one or more characters be followed by a colon. If not, it colors the text box borders red. If the cursor is placed in an invalid `url` box, it displays: "Please enter a URL."

The `email` value for the type attribute of an `input` element is used when the value of the input is an electronic mail address. For example, we could have the following:

```
<input type = "email"  id = "myEmail"  name = "myEmail" >
```

The only difference between this element and a text type element is that the browser is supposed to determine whether the input could possibly be a valid electronic mail address. This check is to make sure the input includes an at-sign (@) and that it is both preceded and followed by at least one character.

Current FX browsers require one or more characters followed by an at-sign, followed by one or more characters. If the input is not considered valid, it colors the text box red. When the cursor is placed in an invalid `email` box, it displays: "Please enter an email address."

Neither the current IE browsers nor the current Chrome browsers check the content of either a URL text box or an electronic mail text box.

The `range` value for the type attribute of an `input` element is used when the value of the input is a number and there are constraints on the range of values that are acceptable. The acceptable range is specified with the `max` and `min` attributes. The default values for `min` and `max` are `0` and `100`, respectively. For example, we could have the following:

```
<input type = "range" id = "myAge"  name = "myAge"
       min = "18"  max = "110" >
```

Current FX and Chrome browsers display a slider for a range type input element, but it is of little value, because it is not labeled with numeric values and the chosen value is not displayed. The current IE browsers display a slider and display the chosen value when the slider is moved.

The `placeholder` attribute can be included in the `text`, `url`, `email`, and `password` input types. It is used to provide initial values to those elements, which serve as hints to the user as to what the input should be. For example, we could have the following:

```
<input type = "text"  id = "name"  name = "name"  size =
"30"  placeholder = "Your name" >
```

2.9.3 The `select` Element

Checkboxes and radio buttons are effective methods for collecting multiple-choice data from a user. However, if the number of choices is large, the form may become too long to display. In these cases, a menu should be used. A menu is specified with a `select` element (rather than with the `input` element). There are two kinds of menus: those in which only one menu item can be selected at a time (which are related to radio buttons) and those in which multiple menu items can be selected at a time (which are related to checkboxes). The default option is the one related to radio buttons. The other option can be specified by adding the `multiple` attribute, set to the value `"multiple"`.[21] When only one menu item is selected, the value sent in the form data is the value of the `name` attribute of the `<select>` tag and the chosen menu item. When multiple menu items are selected, the value for the menu in the form data includes all selected menu items. If no menu item is selected, no value for the menu is included in the form data. The `name` attribute, of course, is required in the `<select>` tag.

The `size` attribute, specifying the number of menu items that initially are to be displayed for the user, can be included in the `<select>` tag. If no `size` attribute is specified, the value `1` is used. If the value for the `size` attribute is `1` and `multiple` is not specified, just one menu item is displayed, with a downward scroll arrow. If the scroll arrow is clicked, the menu is displayed as a pop-up menu. If either `multiple` is specified or the `size` attribute is set to a number larger than `1`, the menu is usually displayed as a scrolled list.

Each of the items in a menu is specified with an `option` element, nested in the select element. The content of an `option` element is the value of the menu item, which is just text. (No tags may be included.) The `<option>` tag can include the `selected` attribute, which specifies that the item is preselected. The value assigned to `selected` is `"selected"`, which can be overridden by the user. The following document describes a menu with the default value (`1`) for `size`:

```
<!DOCTYPE html>
<!-- menu.html
     An example to illustrate menus
     -->
<html lang = "en">
```

21. XHTML requires a value for the `multiple` attribute. However, HTML does not.

```
<head>
  <title> Menu </title>
  <meta charset = "utf-8" />
</head>
<body>
  <p>
    Grocery Menu - milk, bread, eggs, cheese
  </p>
  <form action = "">
    <p>
      With size = 1 (the default)
      <select name = "groceries">
        <option> milk </option>
        <option> bread </option>
        <option> eggs </option>
        <option> cheese </option>
      </select>
    </p>
  </form>
</body>
</html>
```

Figure 2.24 shows a browser display of `menu.html`. Figure 2.25 shows a browser display of `menu.html` after clicking the scroll arrow. Figure 2.26 shows a browser display of `menu.html` after modification to set `size` to `"2."`

Figure 2.24 Display of `menu.html` (default `size` of `1`)

Figure 2.25 Display of `menu.html` after the scroll arrow is clicked

Figure 2.26 Display of `menu.html` with `size` set to `2`

When the `multiple` attribute of the `<select>` tag is set, adjacent options can be chosen by dragging the mouse cursor over them while the left mouse button is held down. Nonadjacent options can be selected by clicking them while holding down the keyboard *Control* key.

2.9.4 The `textarea` Element

In some situations, a multiline text area is needed. The `textarea` element is used to create such a control. The text typed into the area created by `textarea` is not limited in length, and there is implicit scrolling when needed, both vertically and horizontally. The default size of the visible part of the text in a text area is often quite small, so the `rows` and `cols` attributes should usually be included and set to reasonable sizes. If some default text is to be included in the text area, it can be included as the content of the text area element. The following document describes a text area whose window is 40 columns wide and three lines tall:

```
<!DOCTYPE html>
<!-- textarea.html
     An example to illustrate a textarea
     -->
<html lang = "en" >
  <head>
    <title> Textarea </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <p>
      Please provide your employment aspirations
    </p>
    <form action = "handler">
      <p>
        <textarea name = "aspirations"  rows = "3"  cols = "40">
          (Be brief and concise)
        </textarea>
      </p>
    </form>
  </body>
</html>
```

Figure 2.27 shows a browser display of `textarea.html` after some text has been typed into the area.

Figure 2.27 Display of `textarea.html` after some text entry

2.9.5 The Action Buttons

The *Reset* button clears all the controls in the form to their initial states. The *Submit* button has two actions: First, the form data is encoded and sent to the server; second, the server is requested to execute the server-resident program specified in the `action` attribute of the `<form>` tag. The purpose of such a server-resident program is to process the form data and return some response to the user. Neither *Submit* nor *Reset* button requires name or `id` attributes. The *Submit* and *Reset* buttons are created with `input` elements, as shown in the following example:

```
<form action = "">
  <p>
    <input type = "submit"  value = "Submit Form" />
    <input type = "reset"  value = "Reset Form" />
  </p>
</form>
```

Figure 2.28 shows a browser display of *Submit* and *Reset* buttons.

Figure 2.28 *Submit* and *Reset* buttons

The image button is an alternative *Submit* button. The difference is that an image is the clickable area, rather than a button.

2.9.6 Example of a Complete Form

The document that follows describes a form for taking sales orders for popcorn. Three text boxes are used at the top of the form to collect the buyer's name and address. A table is used to collect the actual order. Each row of this table names

a product with the content of a `td` element, displays the price with another `td` element, and uses a text box with `size` set to `2` to collect the quantity ordered. The payment method is input by the user through one of four radio buttons.

Notice that none of the input controls in the order table are embedded in label elements. This is because table elements cannot be labeled, except by using the row and column labels.

Tables present special problems for the visually impaired. The best solution is to use style sheets (see Chapter 3) instead of tables to lay out tabular information.

```
<!DOCTYPE html>
<!-- popcorn.html
     This describes a popcorn sales form document>
     -->
<html lang = "en">
  <head>
    <title> Popcorn Sales Form </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <h2> Welcome to Millennium Gymnastics Booster Club Popcorn
         Sales
    </h2>
    <form action = "">
      <p>
<!-- Text boxes for name and address -->
        <label> Buyer's Name:
          <input type = "text"  name = "name"
                 size = "30" /> </label>
        <br />
        <label> Street Address:
          <input type = "text"  name = "street"
                 size = "30" /> </label>
          <br />
        <label> City, State, Zip:
          <input type = "text"  name = "city"
                 size = "30" /> </label>
      <p />
<!-- A table for item orders -->
      <table>
<!-- First, the column headings -->
        <tr>
          <th> Product Name </th>
          <th> Price </th>
          <th> Quantity </th>
        </tr>
```

```
<!-- Now, the table data entries -->
        <tr>
          <td> Unpopped Popcorn (1 lb.) </td>
          <td> $3.00 </td>
          <td> <input type = "text"  name = "unpop"
                      size = "2" />
          </td>
        </tr>
        <tr>
          <td> Caramel Popcorn (2 lb. canister) </td>
          <td> $3.50 </td>
          <td> <input type = "text"  name = "caramel"
                      size = "2" />
          </td>
        </tr>
        <tr>
          <td> Caramel Nut Popcorn (2 lb. canister) </td>
          <td> $4.50 </td>
          <td> <input type = "text"  name = "caramelnut"
                      size = "2" />
          </td>
        </tr>
        <tr>
          <td> Toffey Nut Popcorn (2 lb. canister) </td>
          <td> $5.00 </td>
          <td> <input type = "text"  name = "toffeynut"
                      size = "2" />
          </td>
        </tr>
      </table>
      <p />
<!-- The radio buttons for the payment method -->
      <h3> Payment Method: </h3>
      <p>
        <label> <input type = "radio"  name = "payment"
                       value = "visa"  checked = "checked" />
                       Visa
        </label>
        <br />
        <label> <input type = "radio"  name = "payment"
                       value = "mc" /> Master Card
        </label>
        <br />
        <label> <input type = "radio"  name = "payment"
                       value = "discover" /> Discover
        </label>
```

```
        <br />
        <label> <input type = "radio"  name = "payment"
                       value = "check" /> Check
        </label>
        <br />
      </p>
<!-- The submit and reset buttons -->
      <p>
        <input type = "submit"  value = "Submit Order" />
        <input type = "reset"  value = "Clear Order Form" />
      </p>
    </form>
  </body>
</html>
```

Figure 2.29 shows a browser display of `popcorn.html`.

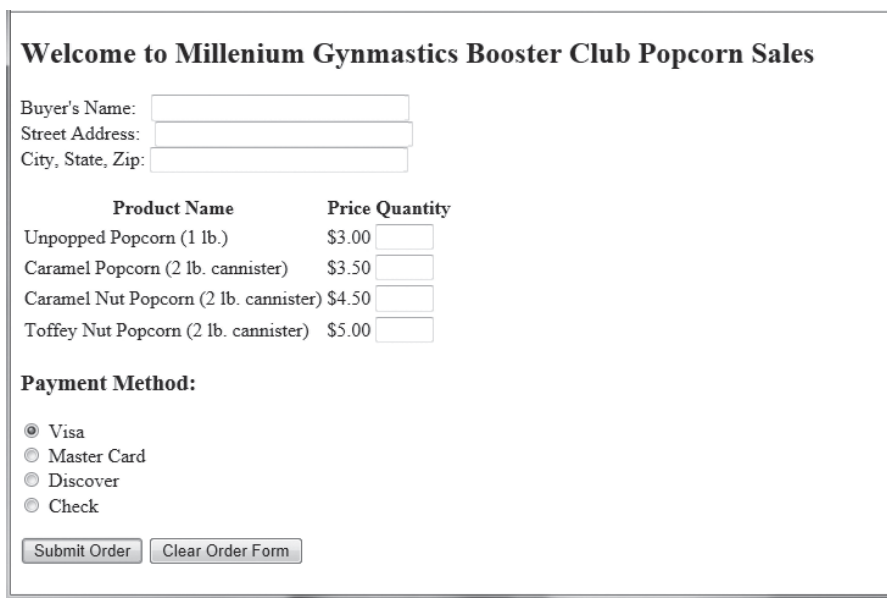


Figure 2.29 Display of `popcorn.html`

Chapter 9 has a PHP script for processing the data from the form in `popcorn.html`.

2.10 The audio Element

Although it has been long recognized that the inclusion of sound during the display of a Web document can enhance its effect, until the arrival of HTML5 there was no standard way of doing that without a plug-in, such as Flash or Microsoft's Media Player. The audio element of HTML5 changes that.

Audio information is coded into digital streams with encoding algorithms called *audio codecs*. There are a large number of different audio codecs. Among these the most commonly used on the Web are MPEG-3 (MP3), Vorbis, and Wav.

Coded audio data is packaged in containers. A container can be thought of as a zip file; it is a way to pack data into a file, but the encoding of the data in the file is irrelevant to the container. A zip file may contain textual data coded in ASCII or it might contain floating-point numbers coded in binary. Likewise, an audio container may contain MP3 or Vorbis coded audio. There are three different audio container types: Ogg, MP3, and Wav. The type of container is indicated by the file name extension. For example, an Ogg container file has the `.ogg` file name extension; Vorbis codec audio data is stored in Ogg containers; MP3 codec audio data is stored in MP3 containers; and Wav codec audio data is stored in Wav containers.

The only commonly used attribute of the audio element is `controls`, which we always set to `"controls"`. This attribute, when present, creates a display of a start/stop button, a clock, a slider of the progress of the play, the total time of the file, and a slider for volume control. The general syntax of an audio element is as follows:

```
<audio attributes>
  <source src = "filename1">
  ...
  <source src = "filenamen">
  Your browser does not support the audio element
</audio>
```

A browser chooses the first audio file it can play and skips the content of the audio element. If it cannot play any of the audio files that appear in the source elements, it does nothing other than displaying its content. Unfortunately, different browsers are capable of playing different audio container/codec combinations. The Firefox 3.5+ browsers support the Ogg/Vorbis and Wav/Wav container/codec audio files. The Chrome 3.0+ browsers support the Ogg/Vorbis and MP3/MP3 container/codec audio files. IE9+ browsers support the MP3/MP3 container/codec audio files. Safari 3.0+ browsers support the Wav/Wav container/codec audio files.

Following is a simple document that illustrates the use of the audio element:

```
<!DOCTYPE html>
<!-- audio.html
     test the audio element
     -->
<html lang = "en">
```

```
  <head>
    <title> test audio element </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    This is a test of the audio element
    <audio controls = "controls" >
      <source src = "nineoneone.ogg" />
      <source src = "nineoneone.wav" />
      <source src = "nineoneone.mp3" />
      Your browser does not support the audio element
    </audio>
  </body>
</html>
```

Note that `audio.html` includes three elements that specify three different audio container files. This allows IE9+, Firefox 3.5+, Chrome 3.0+, and Safari 3.0+ browsers to play the sound clip. A chosen sound file can be converted to the other audio container/codec combinations with software available on the Web.

2.11 The `video` Element

Prior to HTML5, there was no standard way of including video clips in a Web document. The most common approach to video on the Web was the use of the Flash plug-in. The appearance of the video element in HTML5 changes that.

Video information, like audio information, must be digitized into data files before it can be played by a browser, this time by algorithms called *video codecs*. As is the case with audio, video data is stored in containers. There are many different video containers and many different video codecs. Further complicating the situation is the fact that not all video codecs can be stored in all video containers. The most common video containers used on the Web are MPEG-4 (`.mp4` files), Flash Video (`.flv` files), Ogg (`.ogv` files), WebM (`.webm` files), and Audio Video Interleave (`.avi` files).

The most common video codecs used on the Web are H.264 (also known as MPEG-4 Advance Video Coding, or MPEG-4 AVC), which can be embedded in MP4 containers, Theora, which can be embedded in any container, and VP8, which can be embedded in WebM containers. In addition to video data, video containers also store audio data, because most video is accompanied by audio. The three most common container/video codec/audio codec combinations used on the Web are the Ogg container with Theora video codec and Vorbis audio codec, MPEG-4 container with H.264 video codec and AAC audio codec, and WebM container with VP8 video codec and Vorbis audio codec.

IE9+ browsers support the MPEG-4 video containers, Firefox 3.5+ browsers support the Ogg video containers, Firefox 4.0+ browsers support Ogg and WebM video containers, Chrome 6.0+ browsers support all three of the most common video containers, and Safari 3.0+ browsers support the MPEG-4 video containers.

The video element can have several attributes and, like the audio element, can include several nested source elements. The width and height attributes set the size of the screen for the video in pixels. The autoplay attribute specifies that the video plays automatically as soon as it is ready. The preload attribute tells the browser to load the video file or files as soon as the document is loaded. This is not a good thing if not all users will play the video. The controls attribute specifies that play, pause, and volume controls be included in the display. The loop attribute specifies that the video is to be played continuously.

The syntax of the video element is similar to that of the audio element. The general form is as follows:

```
<video attributes>
  <source src = "filename1">
  ...
  <source src = "filenamen">
  Your browser does not support the video element
</video>
```

The semantics of the video element is similar to that of the audio element. Following is an example of a document that includes a video element:

```
<!DOCTYPE html>
<!-- testvideo.html
     test the video element
     -->
<html lang = "en">
  <head>
    <meta charset = "utf-8" />
    <title> test video element </title>
  </head>
  <body>
    This is a test of the video element.....
    <video width = "600" height = "500" autoplay = "autoplay"
           controls = "controls"  preload = "preload">
      <source src = "NorskTippingKebab.mp4" />
      <source src = "NorskTippingKebab.ogv" />
      <source src = "NorskTippingKebab.webm" />
      Your browser does not support the video element
    </video>
  </body>
</html>
```

Older browsers, probably most common among those is IE8, do not recognize the video element. One way to allow such browsers is to nest an object element in the video element that plays the video with Flash. This process is described in Pilgrim.[22]

22. Mark Pilgrim, *HTML5 Up and Running*, O'Reilly (2010): pp. 114–115.

2.12 Organization Elements

One of the deficiencies of HTML 4.01 (and XHTML 1.0) is that it is difficult to organize displayed information in meaningful ways. The primary elements for this in those languages were division (`div`) and paragraph (`p`). Headers were the only way to implement an outline, but it was logical to use just one `h1` header in a document. Furthermore, the `h2`, `h3`, and other header elements had to be nested according to their numbers (e.g., `h3` headers inside `h2` headers). HTML now has a collection of new elements that assist in organizing documents and outlines of documents.

The first part of many documents is a header. If the header consists of just a single phrase, it can be an `h1` element. However, headers of documents often include more information, in many cases a second phrase or sentence called a *tagline*. The header element was designed to encapsulate the whole header of a document. This makes clear what is in the header. For example, one might have the following header:

```
<header>
  <h1> The Podunk Press </h1>
  <h2> "All the news we can fit" </h2>
</header>
```

The beginning part of a document may contain further information that precedes the body of the document, for example, a table of contents. For situations such as this, the `hgroup` element can be used to enclose the header and the other information that precedes the body. Following is an example of this:

```
<hgroup>
  <header>
    <h1> The Podunk Press </h1>
    <h2> "All the news we can fit" </h2>
  </header>
  -- table of contents --
</hgroup>
```

The `footer` element is designed to enclose footer content in a document, such as author and copyright data. For example, consider the following footer element:

```
<footer>
  &copy; The Podunk Press, 2012
  <br />
  Editor in Chief: Squeak Martin
</footer>
```

The following document, `organized.html`, illustrates the `header`, `hgroup`, and `footer` elements:

```
<!DOCTYPE html>
<!-- organized.html
     An example to illustrate organization elements of HTML5
     -->
```

```
<html lang = "en">
  <head>
    <title> Organization elements </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <hgroup>
      <header>
        <h1> The Podunk Press </h1>
        <h3> "All the news we can fit" </h2>
      </header>
       <ol>
        <li> Local news </li>
        <li> National news </li>
        <li> Sports </li>
        <li> Entertainment </li>
      </ol>
    </hgroup>
    <p>
      -- Put the paper's content here --
    </p>
    <footer>
      &copy; The Podunk Press, 2012
      <br />
      Editor in Chief: Squeak Martin
    </footer>
  </body>
</html>
```

Figure 2.30 shows a display of `organized.html`.

Figure 2.30 Display of `organized.html`

The section element is for encapsulating the sections of a document, for example the chapters of a book or separate parts of a paper. A footer element may include one or more sections.

The article element is used to encapsulate a self-contained part of a document that comes from some external source, such as a post from a forum or a newspaper article. An article element can include a header, a footer, and sections. article elements are convenient when a document is put together from several separately written parts.

The aside element is for content that is tangential to the main information of the document. In print, such content is often placed in a sidebar.

The nav element is for encapsulating navigation sections; that is, lists of links to different parts of the document. The nav elements clearly mark the parts of a document that are used to get to other documents. They are especially useful for visually impaired users who use text-to-speech readers to "view" documents.

2.13 The time Element

The time element is used to time stamp an article or a document. This element includes both a textual part, in which the time and/or date information can be in any format, and a machine-readable part, which of course has a strict format. The machine-readable part is given as the value of the datetime attribute of the time element, which is optional. The date part of datetime is given as a four-digit year, a dash, the two-digit month, a dash, and the two-digit day of the month, for example, "2011-02-14". If a time is included with the machine-readable data, it is added to the date with an uppercase T, followed by the hour, a colon, the minute, a colon, and the second. The second is optional if its value is zero. The hour, minute, and second values must be in two-digit form. For example, we could have "2010-02-14T08:00". There is another optional attribute of the time element, pubdate. If the time element is not nested in an article element, the pubdate attribute specifies that the time stamp is the publication date of the document. If the time element is nested inside an article element, it is the publication date of the article. An example of a complete time element is as follows:

```
<time datetime = "2011-02-14T08:00" pubdate = "pubdate">
  February 14, 2011 8:00am MDT
</time>
```

Note that the information in the content of a time element is not necessarily related to the information in the datetime attribute.

The time part of the value of datetime can have a time zone offset attached. The time zone value is an offset in the range of –12:00 to +14:00 (from Coordinated Universal Time). The sign on the time zone value separates it from the time value. For example, we could have the following datetime value:

```
"2011-02-14T08:00-06:00"
```

There are two deficiencies with the time element. First, no years before the beginning of the Christian era can be represented, because negative years are

not acceptable. The second problem is that no approximations are possible—you cannot specify "circa 1900."

2.14 Syntactic Differences between HTML and XHTML

The discussion that follows points out some significant differences between the syntactic rules of HTML (or lack thereof) and those of XHTML.

Case sensitivity. In HTML, tag and attribute names are case insensitive, meaning that `FORM`, `form`, and `Form` are equivalent. In XHTML, tag and attribute names must be all lowercase.

Closing tags. In HTML, closing tags may be omitted if the processing agent (usually a browser) can infer their presence. For example, in HTML, paragraph elements often do not have closing tags. The appearance of another opening paragraph tag is used to infer the closing tag on the previous paragraph. Thus, in HTML we could have

```
<p>
During Spring, flowers are born. ...
<p>
During Fall, flowers die. ...
```

In XHTML, all elements must have closing tags. For elements that do not include content, in which the closing tag appears to serve no purpose, a slash can be included at the end of the opening tag as an abbreviation for the closing tag. For example, the following two lines are equivalent in XHTML:

```
<input type = "text"  name = "address" > </input>
```

and

```
<input type = "text"  name = "address" />
```

Recall that some browsers can become confused if the slash at the end is not preceded by a space.

Quoted attribute values. In HTML, attribute values must be quoted only if there are embedded special characters or white-space characters. Numeric attribute values are rarely quoted in HTML. In XHTML, all attribute values must be double quoted, regardless of what characters are included in the value.

Explicit attribute values. In HTML, some attribute values are implicit; that is, they need not be explicitly stated. For example, if the `multiple` attribute appears in a select tag without a value, it specifies that multiple items can be selected. The following is valid in HTML:

```
<select multiple>
```

This select tag is invalid in XHTML, in which such an attribute must be assigned a string of the name of the attribute. For example, the following is valid in XHTML:

```
<select multiple = "multiple">
```

Other such attributes are `checked` and `selected`.

`id` *and* `name` *attributes*. HTML markup often uses the `name` attribute for elements. This attribute was deprecated for some elements in HTML 4.0, which added the `id` attribute to nearly all elements. In XHTML, the use of `id` is encouraged and the use of `name` is discouraged. However, form control elements must still use the `name` attribute because it is employed in processing form data on the server.

Element nesting. Although HTML has rules against improper nesting of elements, they are not enforced. Examples of nesting rules are (1) an anchor element cannot contain another anchor element, and a form element cannot contain another form element; (2) if an element appears inside another element, the closing tag of the inner element must appear before the closing tag of the outer element; (3) block elements cannot be nested in inline elements; (4) text cannot be directly nested in body or form elements; and (5) list elements cannot be directly nested in list elements. In XHTML, these nesting rules are strictly enforced.

All the XHTML syntactic rules are checked by the Total Validator Tool software.

Summary

Without the style sheets to be described in Chapter 3, HTML is capable of specifying only the general layout of documents, with few presentation details. The current version of HTML is still 4.01, although the HTML5 specification has been distributed via the Web. Although the XHTML development process has stopped, the strict syntactic rules of XHTML are still valuable and can be used with both HTML 4.01 and HTML5.

The elements of HTML specify how content is to be arranged in a display by a browser (or some other HTML processor). Most elements consist of opening and closing tags to encapsulate the content that is to be affected by the tag. HTML documents have two parts: the head and the body. The head describes some things about the document, but does not include any content. The body has the content, as well as the tags and attributes that describe the layout of that content.

Line breaks in text are ignored by browsers. The browser fills lines in its display window and provides line breaks when needed. Line breaks can be specified with the `br` element. Paragraph breaks can be specified with `p`. Headings can be created with the `hx` elements, where `x` can be any number from 1 to 6. The `blockquote` element is used to set off a section of text. The `sub` and `sup` elements are used to create subscripts and superscripts, respectively. Horizontal lines can be specified with the `hr` element.

Images in JPEG, PNF format, or in GIF can be inserted into documents with the `img` element. The `alt` attribute of `img` is used to present a message to the user when his or her browser is unable (or unwilling) to present the associated image.

Links support hypertext by allowing a document to define links that reference positions in either the current document or other documents. These links can be taken by the user viewing the document on a browser.

HTML supports unordered lists, using the `ul` element, and ordered lists, using the `ol` element. Both these kinds of lists use the `li` element to define list elements. The `dl` element is used to describe definition lists. The `dt` and `dd` elements are used to specify the terms and their definitions, respectively.

Tables are easy to create with HTML, through a collection of tags designed for that purpose. The `table` element is used to create a table, `tr` to create table rows, `th` to create label cells, and `td` to create data cells in the table. The `colspan` and `rowspan` attributes, which can appear in both `<th>` and `<td>` tags, provide the means of creating multiple levels of column and row labels, respectively.

HTML forms are sections of documents that contain controls used to collect input from the user. The data specified in a form can be sent to a server-resident program in either of two methods: `get` or `post`. The most commonly used controls (text boxes, checkboxes, passwords, radio buttons, and the action buttons submit, reset, and button) are specified with the `<input>` tag. The *Submit* button is used to indicate that the form data is to be sent to the server for processing. The *Reset* button is used to clear all the controls in a form. The text box control is used to collect one line of input from the user. Checkboxes are one or more buttons used to select one or more elements of a list. Radio buttons are like checkboxes, except that, within a group, only one button can be on at a time. A password is a text box whose content is never displayed by the browser.

Menus allow the user to select items from a list when the list is too long to use checkboxes or radio buttons. Menu controls are created with the `select` element. A text area control, which is created with the `textarea` element, creates a multiple-line text-gathering box with implicit scrolling in both directions.

The audio element allows a document to specify the playing of audio files, the video element allows a document to specify the playing of video files, and the time element provides a way to specify a date and time stamp in machine-readable form in a document.

Review Questions

2.1 What does it mean for a tag or an attribute of HTML to be deprecated?

2.2 What is the form of an HTML comment?

2.3 How does a browser treat line breaks in text that is to be displayed?

2.4 What is the difference between the effect of a paragraph element and a break element?

2.5 Which heading elements use fonts that are smaller than the normal text font size?

2.6 How do browsers usually set block quotations differently from normal text?

2.7 What does the `code` element specify for its content?

2.8 What are the differences between the JPEG and GIF image formats?

2.9 What are the two required attributes of an `img` element?

2.10 What is the purpose of the `alt` attribute of `img`?

2.11 What tag is used to define a link?

2.12 What attribute is required in all anchor tags?

2.13 Does HTML allow nested links?

2.14 How is the target of a link usually identified in a case where the target is in the currently displayed document but not at its beginning?

2.15 What is the form of the value of the `href` attribute in an anchor tag when the target is a fragment of a document other than the one in which the link appears?

2.16 What is the default bullet form for the items in an unordered list?

2.17 What are the default sequence values for the items in an ordered list?

2.18 What tags are used to define the terms and their definitions in a definition list?

2.19 What is the purpose of the `colspan` attribute of the `th` element?

2.20 What is the purpose of the `rowspan` attribute of the `td` element?

2.21 What are controls?

2.22 Which controls discussed in this chapter are created with the `input` element?

2.23 What is the default size of a text control's text box?

2.24 What is the difference between the `size` and `maxlength` attributes of `input` for text controls?

2.25 What is the difference in behavior between a group of checkbox buttons and a group of radio buttons?

2.26 Under what circumstances is a menu used instead of a radio button group?

2.27 How are scroll bars specified for `textarea` controls?

2.28 Explain the behavior of an input element with the `url` type.

2.29 Explain the behavior of an input element with the `email` type.

2.30 What is the purpose of the `placeholder` attribute of an input element?

2.31 Before HTML5, how were sound clips played while a browser displayed a Web document?

2.32 What is an audio codec?

2.33 What is an audio container?

2.34 Why would an audio element include more than one source element?

2.35 Before HTML5, what was the most common way to play video clips when a Web document was displayed?

2.36 What does the `autoplay` attribute of the video element do?

2.37 Before HTML5, what HTML elements were used to organize documents?

2.38 What is the purpose of the article element?

2.39 What is the format of the date part of the value of the `datetime` attribute?

Exercises

2.1 Create and test an HTML document for yourself, including your name, address, and electronic mail address. If you are a student, you must include your major and your grade level. If you work, you must include your employer, your employer's address, and your job title. This document must use several headings and `<em>`, `<strong>`, `<hr />`, `<p>`, and `<br />` tags.

2.2 Add pictures of yourself and at least one other image (of your friend, spouse, or pet) to the document created for Exercise 2.1.

2.3 Add a second document to the document created for Exercise 2.1 that describes part of your background, using `background` as the link content. This document should have a few paragraphs of your personal or professional history.

2.4 Create and test an HTML document that describes an unordered list equivalent to your typical grocery shopping list. (If you've never written a grocery list, use your imagination.)

2.5 Create and test an HTML document that describes an unordered list of at least four states. Each element of the list must have a nested list of at least three cities in the state.

2.6 Create and test an HTML document that describes an ordered list of your five favorite movies.

2.7 Modify the list of Exercise 2.6 to add nested, unordered lists of at least two actors and/or actresses in your favorite movies.

2.8 Create and test an HTML document that describes an ordered list with the following contents: The highest level should be the names of your parents, with your mother first. Under each parent, you must have a nested, ordered list of the brothers and sisters of your parents (your aunts and uncles) in order by age, eldest first. Each of the nested lists in turn must have nested lists of the children of your aunts and uncles (your cousins)—under the proper parents, of course. Regardless of how many aunts, uncles, and cousins you actually have, there must be at least three list items in each sublist below each of your parents and below each of your aunts and uncles.

2.9 Create and test an HTML document that describes a table with the following contents: The columns of the table must have the headings "Pine," "Maple," "Oak," and "Fir." The rows must have the labels "Average Height," "Average Width," "Typical Life Span," and "Leaf Type." You can make up the data cell values.

2.10 Modify and test an HTML document from Exercise 2.9 that adds a second-level column label, "Tree," and a second-level row label, "Characteristics."

2.11 Create and test an HTML document that defines a table with columns for state, state bird, state flower, and state tree. There must be at least five rows for states in the table.

2.12 Create and test an HTML document that defines a table with two levels of column labels: an overall label, "Meals," and three secondary labels, "Breakfast," "Lunch," and "Dinner." There must be two levels of row labels: an overall label, "Foods," and four secondary labels, "Bread," "Main Course," "Vegetable," and "Dessert." The cells of the table must contain a number of grams for each of the food categories.

2.13 Create and test an HTML document that is the home page of a business, Tree Branches, Unlimited, which sells tree branches. This document must include images and descriptions of at least three different kinds of tree branches. There must be at least one unordered list, one ordered list, and one table. Detailed descriptions of the different branches must be stored in separate documents that are accessible through links from the home document. You must invent several practical uses for tree branches and include sales pitches for them.

2.14 Create and test an HTML document that has a form with the following controls:

a. A text box to collect the user's name
b. Four checkboxes, one each for the following items:
    i. Four 25-watt light bulbs for $2.39
    ii. Eight 25-watt light bulbs for $4.29
    iii. Four 25-watt long-life light bulbs for $3.95
    iv. Eight 25-watt long-life light bulbs for $7.49
c. A collection of three radio buttons that are labeled as follows:
    i. Visa
    ii. Master Card
    iii. Discover

2.15 Modify the document from one of the earlier exercises to add a sound track that plays continuously while the document is displayed. The audio must be in all three of the common container/codec forms.

2.16 Modify the document from one of the earlier exercises to add a video that plays continuously while the document is displayed. The video should be related to the information displayed by the document. The video must be in all three of the common container/codec forms.

2.17 Modify the document from one of the earlier exercises to add both header and footer elements. An article element that contains information relevant to the document but which is from some external source must also be included.

CHAPTER

3

Cascading Style Sheets

This chapter introduces the concept of style sheets and describes how they are used to override the default styles of the elements of HTML documents. To begin, the three levels of style sheets and the format of style specifications are introduced. Next, selector forms are discussed. Then, the many varieties of property-value forms are described. Next, specific properties for fonts and lists are introduced and illustrated. A discussion of the properties for specifying colors, background images, and text alignment follows. The box model of document

elements is then discussed, along with borders and the associated padding and margin properties. The chapter's next section describes two elements, `span` and `div`, that are used to delimit the scope of style sheet specifications. Finally, the last section of the chapter provides an overview of the resolution process for conflicting style specifications.

There are several CSS properties that are used to specify the position of elements in the display of a document. Because these are used to build dynamic documents, they are discussed in Chapter 6, rather than in this chapter.

3.1 Introduction

We have said that HTML is concerned primarily with content rather than the details of how that content is presented by browsers. That is not entirely true, however, even with the elements discussed in Chapter 2. Some of those elements—for example, `code`—specify presentation details or style. However, these presentation specifications can be more precisely and more consistently described with style sheets. Furthermore, many of the elements and attributes that can be used for describing presentation details have been deprecated in favor of style sheets.

Most HTML elements have associated properties that store presentation information for browsers. Browsers use default values for these properties when the document does not specify values. For example, the `h2` element has the `font-size` property, for which a browser would have the default value of a particular size. A document could specify that the `font-size` property for `h2` be set to a larger size, which would override the default value. The new value could apply to one occurrence of an `h2` element, some subset of the occurrences, or all such occurrences in the document, depending on how the property value is set.

A style sheet is a syntactic mechanism for specifying style information. The idea of a style sheet is not new: Word processors and desktop publishing systems have long used style sheets to impose a particular style on documents. The first style-sheet specification for use in HTML documents, dubbed Cascading Style Sheets (CSS1), was developed in 1996 by the World Wide Web Consortium (W3C). In mid-1998, the second standard, CSS2, was released. CSS2 added many properties and property values to CSS1. It also extended presentation control to media other than Web browsers, such as printers. As a result of the incomplete implementation of (and perhaps a lack of interest in) parts of CSS2 by browser implementers, W3C developed a new standard, CSS2.1, which reflected the level of acceptance of CSS2. Internet Explorer 8 and later (IE8+), Chrome 5 and later (C5+), and Firefox 3 and later (FX3+) fully support CSS2.1, which was at the *working draft* stage as of spring 2011. CSS3 has been in development since the late 1990s. Current versions of browsers already have implemented some parts of CSS3. This chapter covers most of CSS2.1.

Perhaps the most important benefit of style sheets is their capability of imposing consistency on the style of Web documents. For example, they allow the author to specify that all paragraphs of a document have the same presentation style and therefore the same appearance.

CSS style sheets are called *cascading* style sheets because they can be defined at three different levels to specify the style of a document. Lower-level style sheets can override higher- level style sheets, so the style of the content of an element is determined, in effect, through a cascade of style-sheet applications.

3.2 Levels of Style Sheets

The three levels of style sheets, in order from lowest level to highest level, are *inline*, *document level*, and *external*. Inline style sheets apply to the content of a single HTML element, document-level style sheets apply to the whole body of a document, and external style sheets can apply to the bodies of any number of documents. Inline style sheets have precedence over document style sheets, which have precedence over external style sheets. For example, if an external style sheet specifies a value for a particular property of a particular element, that value is used until a different value is specified in either a document style sheet or an inline style sheet. Likewise, document style sheet property values can be overridden by different property values in an inline style sheet. In effect, the properties of a specific element are those that result from a merge of all applicable style sheets, with lower-level style sheets having precedence in cases of conflicting specifications. There are other ways style specification conflicts can occur. These ways and their resolution are discussed in Section 3.13.

If no style sheet provides a value for a particular style property, the browser default property value is used. Because none of the example HTML documents in Chapter 2 includes style sheets, every element in those documents uses the browser default value for its properties.[1]

As is the case with elements and attributes, a particular browser may not be capable of using the property values specified in a style sheet. For example, if the value of the `font-size` property of a paragraph is set to a particular size, but the browser cannot display the particular font being used in that size, the browser obviously cannot fulfill the property specification. In this case, the browser either would substitute an alternative value or would simply ignore the given font-size value and use its default font size.

Inline style specifications appear within the opening tag and apply only to the content of that element. This fine-grain application of style defeats one of the primary advantages of style sheets—that of imposing a uniform style on the elements of at least one whole document. Another disadvantage of inline style sheets is that they result in style information, which is expressed in a language distinct from HTML markup, being embedded in various places in documents. It is better to keep style specifications separate from HTML markup. For this reason, among others, W3C deprecated inline style sheets in XHTML 1.1.[2] Therefore, inline style specifications should be used sparingly. This chapter discusses inline style sheets, but we follow our own advice and make little use of them in our examples.

---

1. This is not precisely true; some property values could come from browser user overrides of the browser default values.

2. A feature being placed on the list of deprecated features is a warning to users to restrict their use of that feature, because sometime in the future it will be discontinued.

Document-level style specifications appear in the document head section and apply to the entire body of the document. This is obviously an effective way to impose a uniform style on the presentation of all the content of a single document.

In many cases, it is desirable to have a style sheet apply to more than one document. That is the purpose of external style sheets, which are not part of any of the documents to which they apply. They are stored separately and are referenced in all documents that use them. Another advantage of external style sheets is that their use cleanly separates CSS from HTML. External style sheets are written as text files with the MIME type `text/css`. They can be stored on any computer on the Web. The browser fetches external style sheets just as it fetches HTML documents. The `<link>` tag is used to specify external style sheets.[3] Within `<link>`, the `rel` attribute is used to specify the relationship of the linked-to document to the document in which the link appears. The `href` attribute of `<link>` is used to specify the URL of the style sheet document, as in the following example:

```
<link rel = "stylesheet"  type = "text/css"
      href = "http://www.cs.usc.edu/styles/wbook.css" />
```

The link to an external style sheet must appear in the head of the document. If the external style sheet resides on the Web server computer, only its path address must be given as the value of `href`. An example of an external style sheet appears in Section 3.6.

Because it is good to separate CSS from markup, it is preferable to use external style sheets. However, because the example documents in this book are all relatively small and we want to keep the CSS near where it is used so it is easy to reference, we nearly always use document-level style sheets in our examples.

External style sheets can be validated with the service provided at `http://jigsaw.w3.org/css-validator/`.

3.3 Style Specification Formats

The format of a style specification depends on the level of style sheet. Inline style specifications appear as values of the `style` attribute of a tag,[4] the general form of which is as follows:

```
style = "property_1 : value_1 ; property_2 : value_2 ; . . . ;
         property_n : value_n ;"
```

Although it is not required, it is recommended that the last property-value pair be followed by a semicolon.

---

3. There is an alternative to using the `<link>` tag, `@import`. However, `@import` is slower so there is no good reason to use it.

4. The `style` attribute is deprecated in the XHTML 1.1 recommendation. However, it is still part of HTML5.

Document style specifications appear as the content of a `style` element within the header of a document, although the format of the specification is quite different from that of inline style sheets. The general form of the content of a style element is as follows:[5]

```
<style type = "text/css">
  rule_list
</style>
```

The `type` attribute of the `<style>` tag tells the browser the type of style specification, which is `text/css` for CSS.

Each style rule in a rule list has two parts: a selector, which specifies the element or elements affected by the rule, and a list of property-value pairs. The list has the same form as the quoted list for inline style sheets, except that it is delimited by braces rather than double quotes. So, the form of a style rule is as follows:

selector { *property_1* : *value_1* ; *property_2* : *value_2* ; . . . ;
property_n : *value_n* ; }

If a property is given more than one value, those values usually are separated with spaces. For some properties, however, multiple values are separated with commas.

Like all other kinds of coding, complicated CSS rule lists should be documented with comments. Of course, HTML comments cannot be used here, because CSS is not HTML. Therefore, a different form of comment is needed. CSS comments are introduced with `/*` and terminated with `*/`,[6] as in the following element:

```
<style type = "text/css">
  /* Styles for the initial paragraph */
  ...
  /* Styles for other paragraphs */
  ...
</style>
```

An external style sheet consists of a list of style rules of the same form as in document style sheets. The `<style>` tag is not included. An example of an external style sheet appears in Section 3.6.

3.4 Selector Forms

A selector specifies the elements to which the following style information applies. The selector can have a variety of forms.

---

5. Browsers so old that they do not recognize the `<style>` tag may display the content of the style element at the top of the document. There are now so few such browsers in use that we ignore the issue here. Those who are concerned put the rule list in an HTML comment.

6. This form of comment is adopted from the C programming language and some of its descendants.

3.4.1 Simple Selector Forms

The simplest selector form is a single element name, such as `h1`. In this case, the property values in the rule apply to all occurrences of the named element. The selector could be a list of element names separated by commas, in which case the property values apply to all occurrences of all the named elements. Consider the following examples:

`h1` {*property-value list*}
`h2, h3` {*property-value list*}

The first of these selector forms specifies that the following property-value list applies to all `h1` elements. The second specifies that the following property-value list applies to all `h2` and `h3` elements.

3.4.2 Class Selectors

Class selectors are used to allow different occurrences of the same element to use different style specifications. A style class is defined in a style element by giving the style class a name, which is attached to the element's name with a period. For example, if you want two paragraph styles in a document—say, `normal` and `warning`—you could define these two classes in the content of a style element as follows:

`p.normal` {*property-value list*}
`p.warning` {*property-value list*}

Within the document body, the particular style class that you want is specified with the `class` attribute of the affected element—in the preceding example, the paragraph element. For example, you might have the following markup:

```
<p class = "normal">
A paragraph of text that we want to be presented in
'normal' presentation style
</p>
<p class = "warning">
A paragraph of text that is a warning to the reader, which
should be presented in an especially noticeable style
</p>
```

3.4.3 Generic Selectors

Sometimes it is convenient to have a class of style specifications that applies to the content of more than one kind of element. This is done by using a generic class, which is defined without an element name in its selector. Without the element name, the name of the generic class begins with a period, as in the following generic style class:

`.sale` {*property-value list*}

Now, in the body of a document, you could have the following markup:

```
<h3 class = "sale"> Weekend Sale </h3>
...
<p class = "sale">
...
</p>
```

3.4.4 `id` Selectors

An `id` selector allows the application of a style to one specific element. The general form of an `id` selector is as follows:[7]

#*specific-id* {*property-value list*}

As you would probably guess, the style specified in the `id` selector applies to the element with the given `id`. For example, consider the following selector:

`#section14` {*property-value list*}

Following is the `h2` element to which this style applies:

```
<h2 id = "section14"> 1.4 Calico Cats </h2>
```

3.4.5 Contextual Selectors

Selectors can specify, in several different ways, that the style should apply only to elements in certain positions in the document. The simplest form of contextual selector is the descendant selector. Element B is a *descendant* of element A if it appears in the content of A. In this situation, A is the *ancestor* of B. A particular element can be selected by listing one or more of the ancestors of the element in the selector, with only white space separating the element names. For example, the following rule applies its style only to the content of ordered list elements that are descendants of unordered list elements in the document:

`ul ol` {*property-value list*}

An element is a *child* of another element if it is a descendant and it is nested directly in that element. Element B is directly nested in element A if there are no opening tags between the opening tag of A and that of B that do not have corresponding closing tags. Also, if B is the child of A, we call A the *parent* of B. For example, in the following, the first and third `li` elements are children of the `ol` element. The second `li` element is a descendant of the `ol` element, but is not a child of it, because its parent is the `ul` element.

7. For the oddly curious reader, the Bell Labs name for the # symbol is *octothorpe*. It was named that when it was first put on the telephone dial. The name comes from the eight points of intersection of the figure with the circumference of its circumscribed circle.

```
<ol>
  <li> ... </li>
    <ul>
      <li> ... </li>
      ...
    </ul>
  <li> ... </li>
  ...
</ol>
```

CSS includes a child selector. For example, the following selector applies to `li` elements only if they are children of `ol` elements:

`ol > li` {*property-value list*}

Child selectors can be specified over any number of elements in a family hierarchy (not just one generation). For example, the following selector selects `em` elements that are children of `h1` elements that are children of paragraph elements:

`p > h1 > em` {*property-value list*}

The `first-child` selector specifies the first child of the element to whose name it is attached. For example, consider the following:

`p:first-child` {*property-value list*}

The properties in the list are applied to the first child element of each `p` element.

The `last-child` selector specifies the last child of the element to whose name it is attached. For example, consider the following:

`p:last-child` {*property-value list*}

The properties in the list are applied to the last child element of each `p` element.

The `only-child` selector specifies the child of the element to whose name it is attached, but only if that child is the only child of the element. For example, consider the following:

`p:only-child` {*property-value list*}

The properties in the list are applied to the child element of each `p` element that has just one child element.

The `empty` selector specifies the element to which it is attached when that element has no child elements. For example, consider the following:

`p:empty` {*property-value list*}

The properties in the list are applied to each `p` element that has no child elements.

3.4.6 Pseudo Classes

Pseudo classes specify that certain styles apply when something happens, rather than because the target element simply exists. In this section, we describe and illustrate four pseudo classes, two that are used exclusively to style hypertext links, and two that can be used to style any element.

The two pseudo classes for styling links are `link` and `visited`. The `link` pseudo class is used to style a link that has not been selected; `visited` is used to style a link that previously has been selected.

The style of the `hover` pseudo class applies when it is associated element has the mouse cursor over it. The style of the `focus` pseudo class applies when it is associated element has focus.[8]

Browsers often use blue as the default color for unvisited links and red or purple for visited links. They usually also underline links. If the background color is white, this is fine. However, if the background color is not white, it is better to make the unvisited links some bright color that contrasts with the background color of the document. Contrasting colors can be found with a color sphere, such as can be found at `http://www.colorjack.com/sphere`. In such a sphere, contrasting colors are 180 degrees apart on the sphere. Visited links can then be a muted version of the chosen contrasting color.

When using the `hover` pseudo class, changing the size of an element from its initial size, for example by enlarging or shrinking the font size, can lead to problems. For example, if the font size is made larger, the larger size could cause the element to overflow the area reserved for it in the display, causing the whole document to be rearranged. That would likely annoy the user.

Whereas the names of style classes and generic classes begin with a period, the names of pseudo classes begin with a colon. For example, the selector for the `hover` pseudo class applied to an `h2` element is as follows:

`h2:hover` {*property-value list*}

Any time the mouse cursor is positioned over an `h2` element, the styles defined in the given property-value list are applied to the content of the `h2` element.

3.4.7 The Universal Selector

The universal selector, denoted by an asterisk (`*`), applies its style to all elements in a document. For example, if we wanted every element in a document to have a particular set of properties, we could include the following:

`*` {*property-value list*}

3.5 Property-Value Forms

CSS includes a large number of different properties, arranged in categories. The most commonly used categories are: fonts, lists, alignment of text, margins, colors, backgrounds, and borders. As you probably would guess, only a fraction of the properties are discussed here. The complete details of all properties and property values can be found at `http://www.w3.org/TR/2011/REC-CSS2-20110607/propidx.html`.

8. One way an element acquires focus is when the user places the mouse cursor over it and clicks the left mouse button.

Property values can appear in a variety of forms. Keyword property values are used when there are only a few possible values and they are predefined—for example, `large`, `medium`, and `small`. Keyword values are not case sensitive, so `Small`, `SmAlL`, and `SMALL` are all the same as `small`.

Number values are used when no meaningful units can be attached to a numeric property value. A number value can be either an integer or a sequence of digits with a decimal point and can be preceded by a sign (`+` or `-`).

Length values are specified as number values that are followed immediately by a two-character abbreviation of a unit name. There can be no space between the number and the unit name. The possible unit names are `px`, for pixels; `in`, for inches; `cm`, for centimeters; `mm`, for millimeters; `pt`, for points (a point is 1/72 inch); and `pc`, for picas, which are 12 points. Note that on a display, `in`, `cm`, `mm`, `pt`, and `pc` are approximate measures. Their actual values depend on the screen resolution. There are also two relative length values: `em`, which is the value of the current font size in pixels, and `ex`, which is the height of the letter *x*.

Percentage values are used to provide a measure that is relative to the previously used measure for a property value. Percentage values are numbers that are followed immediately by a percent sign (`%`). For example, if the font size were set to `75%` with a style sheet, it would make the new current size for the font 75 percent of its previous value. Font size would stay at the new value until changed again. Percentage values can be signed. If preceded by a plus sign, the percentage is added to the previous value; if negative, the percentage is subtracted.

URL property values use a form that is slightly different from references to URLs in links. The actual URL, which can be either absolute or relative, is placed in parentheses and preceded by `url`, as in the following property:

```
url(tetons.jpg)
```

There can be no space between `url` and the left parenthesis. If there is one, the property and its value will be ignored by the browser.

Color property values can be specified as color names, as six-digit hexadecimal numbers, or in RGB form. RGB form is just the word `rgb` followed by a parenthesized list of three decimal numbers in the range of `0` to `255` or three percentage values. These numbers or percentages specify the levels of red, green, and blue, respectively. For example, a value of `0` or `0%` as the first of the three values would specify that no red be included in the color. A value of `255` or `100%` would specify the maximum amount of red. Hexadecimal numbers must be preceded with pound sign (`#`), as in `#43AF00`. For example, fuchsia (a mixture of red and blue) could be specified with

```
fuchsia
```

or

```
rgb(255, 0, 255)
```

or

```
#FF00FF
```

As is the case with `url`, there can be no space between `rgb` and the left parenthesis. If there is, the value will be ignored by the browser.

CSS also includes properties for counters and strings, but they are not covered here.

As has been hinted previously, many property values are inherited by descendent elements. For example, because `font-size` is an inherited property, setting it to a value on the `<body>` tag effectively sets that value as the new default property value for all the elements for the whole body of the document (because all elements in the body of a document are descendants of the body element).

Not all properties are inherited, although those cases are somewhat intuitive. For example, `background-color` and the margin properties are not inherited.[9]

Color values in hexadecimal and RGB can be converted between the two forms with calculators `at http://www.javascripter.net/faz/hextorgb.htm`.

3.6 Font Properties

The font properties are among the most commonly used of the style-sheet properties. Virtually all HTML documents include text, which is often used in a variety of different situations. This creates a need for text in many different fonts, font styles, and sizes. The font properties allow us to specify these different forms.

3.6.1 Font Families

The `font-family` property is used to specify a list of font names. The browser uses the first font in the list that it supports.[10] For example, consider the following property:

```
font-family: Arial, Helvetica, Futura
```

This tells the browser to use Arial if it supports that font. If it does not support Arial, it should use Helvetica if it supports it. If the browser supports neither Arial nor Helvetica, it should use Futura if it supports it. If the browser does not support any of the specified fonts, it will use an alternative of its choosing.

A generic font can be specified as a `font-family` value. The possible generic fonts and examples of each are shown in Table 3.1. Every browser has a font defined for each of these generic names. A good approach to specifying fonts is to use a generic font as the last font in the value of a `font-family` property.

---

9. `inherit` is CSS keyword. By default, most properties are set to `inherit`. If a property is set to `inherit`, its value will be inherited. Some of the properties that are not `inherit` by default can be set to `inherit`, as in the following: `div {background-color: inherit;}`.

10. Typically, browsers support all fonts that are installed on the browser's host computer.

For example, because Arial, Helvetica, and Futura are sans-serif fonts,[11] the previous example would be better as follows:

```
font-family: Arial, Helvetica, Futura, sans-serif
```

Now, if the browser does not support any of the named fonts, it will use a font from the same category, in this case, sans serif.

Table 3.1 Generic fonts

Generic Name	Examples
`Serif`	`Times New Roman, Garamond`
`Sans-serif`	`Arial, Helvetica`
`cursive`	`Caflisch Script, Zapf-Chancery`
`fantasy`	`Critter, Cottonwood`
`monospace`	`Courier, Prestige`

If a font name has more than one word, the whole name should be delimited by single quotes,[12] as in the following example:

```
font-family: 'Times New Roman'
```

In practice, the quotes may not be required, but their use is recommended because they may be necessary in the future.

3.6.2 Font Sizes

The `font-size` property does what its name implies; its value specifies the size of the font. Unfortunately, it is not as simple as we wish it were.

There are two categories of `font-size` values: absolute and relative. In the absolute category, the size value could be given as a length value in points, picas, or pixels, or as a keyword from the list: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, and `xx-large`. One problem with the keyword sizes is that the size relationship between adjacent keywords is not exactly the same on different browsers.

The relative size values are `smaller` and `larger`, which adjust the font size relative to the font size of the parent element. Once again, however, the amount of change that results from these is not the same among browsers. Percent values can also be used to adjust the font size relative to the font size of the parent element. But in this case, the property value is a uniform size adjustment. Finally, a number with the unit `em` can be used. For example,

```
font-size: 1.2em
```

11. Serifs are nonstructural decorations that may appear at the ends of strokes in a character. Sans-serif fonts do not have serifs.

12. Single quotes are used here, because in the case of inline style sheets, the whole property list is delimited by double quotes.

This sets the font size to 1.2 times the font size of the parent element. So, percentages and the use of `em` are equivalent. `1.2em` and `120%` are exactly the same.

One problem with using points and picas for font sizes is that they do not display in the same size on different computers. Points and picas were designed for printed media—that is where they should be used. Furthermore, if the user changes the default font size, on some browsers these will not change. If a relative size is given, the font size will be scaled relative to a new default set by the user. Although the keywords are in the absolute category, they are set relative to the default font size of the browser.

Considering all these issues, percentages and `em` are good choices for setting font sizes. We use `em` in the example documents in the remainder of the book.

3.6.3 Font Variants

The default value of the `font-variant` property is `normal`, which specifies the usual character font. This property can be set to `small-caps` to specify small capital letters. These are all uppercase, but the letters that are normally uppercase are a bit larger than those that are normally lowercase.

3.6.4 Font Styles

The `font-style` property is usually used to specify italic, as in

```
font-style: italic
```

An alternative to `italic` is `oblique`, but when displayed, the two are nearly identical,[13] so `oblique` is not a terribly useful font style.

There is one other possible value for font style, `normal`, which specifies that the font be the normal style. This tells the browser to stop using whatever non-normal style it had been using until instructed otherwise.

3.6.5 Font Weights

The `font-weight` property is used to specify the degree of boldness, as in

```
font-weight: bold
```

Besides `bold`, the possible values `normal` (the default), `bolder`, and `lighter` can be specified. The `bolder` and `lighter` values are taken as relative to the level of boldness of the parent element. Specific numbers also can be given in multiples of `100` from `100` to `900`, where `400` is the same as `normal` and `700` is the same as `bold`. Because many fonts are available only in normal and bold, the use of these numbers often just causes the browser to choose either `normal` or `bold`.

13. Actually, italic fonts have slightly extended serifs, whereas oblique fonts have normal serifs. Both are slanted to the right.

3.6.6 Font Shorthands

If more than one font property must be specified, the values can be stated in a list as the value of the `font` property. The browser then determines which properties to assign from the forms of the values. For example, the property

```
font: bold 1.1em 'Times New Roman' Palatino
```

specifies that the font weight should be `bold`, the font size should be 1.1 times that of its parent element, and either Times New Roman or Palatino font should be used, with precedence given to Times New Roman.

The order in which the property values are given in a `font` value list is important. The order must be as follows: The font names must be last, the font size must be second to last, and the font style, font variant, and font weight, when they are included, can be in any order but must precede the font size. Only the font size and the font family are required in the `font` value list.

The document `fonts.html` illustrates some aspects of style-sheet specifications of the font properties in headings and paragraphs:

```
<!DOCTYPE html>
<!-- fonts.html
     An example to illustrate font properties
     -->
<html lang = "en">
  <head>
    <title> Font properties </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      p.major {font-size: 1.1em;
               font-style: italic;
               font-family: 'Times New Roman';
              }
      p.minor {font: bold 0.9em 'Courier New';}
      h2 {font-family: 'Times New Roman';
          font-size: 2em; font-weight: bold;}
      h3 {font-family: 'Courier New'; font-size: 1.5em;}
    </style>
  </head>
  <body>
    <p class = "major">
      If a job is worth doing, it's worth doing right.
    </p>
    <p class = "minor">
      Two wrongs don't make a right, but they certainly
      can get you in a lot of trouble.
    </p>
```

```
    <h2> Chapter 1 Introduction </h2>
    <h3> 1.1 The Basics of Computer Networks </h3>
  </body>
</html>
```

Figure 3.1 shows a browser display of `fonts.html`.

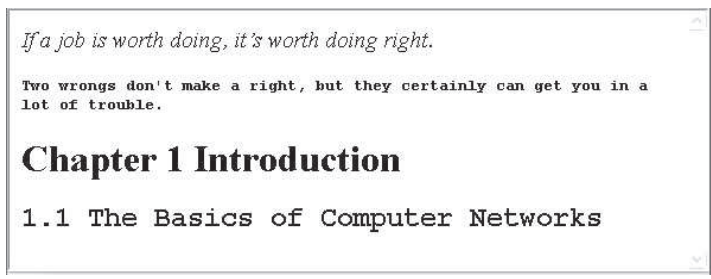


Figure 3.1 Display of `fonts.html`

The following document, called `fonts2.html`, is a revision of `fonts .html` that uses an external style sheet in place of the document style sheet used in `fonts .html` (the external style sheet, `styles.css`, follows the revised document):

```
<!DOCTYPE html>
<!-- fonts2.html
     An example to test external style sheets
     -->
<html lang = "en">
  <head>
    <title> External style sheets </title>
    <meta charset = "utf-8" />
    <link rel = "stylesheet"  type = "text/css"
          href = "styles.css" />
  </head>
  <body>
    <p class = "major">
      If a job is worth doing, it's worth doing right.
    </p>
    <p class = "minor">
      Two wrongs don't make a right, but they certainly
      can get you in a lot of trouble.
    </p>
    <h2> Chapter 1 Introduction </h2>
    <h3> 1.1 The Basics of Computer Networks </h3>
  </body>
</html>
```

```
/* styles.css - an external style sheet
     for use with fonts2.html
   */
  p.major {font-size: 1.1em;
           font-style: italic;
           font-family: 'Times New Roman';
         }
  p.minor {font: bold 0.9em 'Courier New';}
  h2 {font-family: 'Times New Roman';
      font-size: 2em; font-weight: bold;}
  h3 {font-family: 'Courier New';
      font-size: 1.5em;}
```

3.6.7 Text Decoration

The `text-decoration` property is used to specify some special features of text. The available values are `line-through`, `overline`, `underline`, and `none`, which is the default. Many browsers implicitly underline links. The `none` value can be used to avoid this underlining.[14] Note that `text-decoration` is not inherited. The following document, `decoration.html`, illustrates the `line-through`, `overline`, and `underline` values:

```
<!DOCTYPE html>
<!-- decoration.html
     An example that illustrates several of the
     possible text decoration values
     -->
<html lang = "en">
  <head>
    <title> Text decoration </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      p.delete {text-decoration: line-through;}
      p.cap {text-decoration: overline;}
      p.attention {text-decoration: underline;}
    </style>
  </head>
```

14. Setting `text-decoration` to `none` for a link is a bad idea, because it makes it less likely the link will be noticed.

```
    <body>
      <p class = "delete">
        This illustrates line-through
      </p>
      <p class= "cap">
        This illustrates overline
      </p>
      <p class = "attention">
        This illustrates underline
      </p>
    </body>
  </html>
```

Figure 3.2 shows a browser display of `decoration.html`.

Figure 3.2 Display of `decoration.html`

3.6.8 Text Spacing

The `letter-spacing` property controls the amount of space between the letters in words. This spacing is called *tracking*. The possible values of `letter-spacing` are `normal` or any length property value. Positive values increase the letter spacing; negative values decrease it. For example, `letter-spacing: 1px` spreads the letters of words. Likewise, `letter-spacing: -1px` squeezes the letters of words together. The value `normal` resets `letter-spacing` back to that of the parent element.

The space between words in text can be controlled with the `word-spacing` property, whose values are `normal` or any length value. Once again, a positive value increases the space between words and negative values decrease that space, and `normal` resets word spacing back to that of the parent element.

The space between lines of text can be controlled with the `line-height` property. This spacing is called *leading*. The value of `line-height` can be a number, in which case a positive number sets the line spacing to that number times the font size (`2.0` means double spacing). The value could be a length, such as `24px`. If the font size is 12 pixels, this would specify double spacing. The value could be a percentage, which is relative to the spacing of the parent element.

Finally, `normal`, which overrides the current value, is used to set line spacing back to that of the parent element.

The following document, `text_space.html`, illustrates the text spacing properties:

```
<!DOCTYPE html>
<!-- text_space.html
     An example to illustrate text spacing properties
     -->
<html lang = "en">
  <head>
    <title> Text spacing properties </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      p.big_tracking {letter-spacing: 0.4em;}
      p.small_tracking {letter-spacing: -0.08em;}
      p.big_between_words {word-spacing: 0.4em;}
      p.small_between_words {word-spacing: -0.1em;}
      p.big_leading {line-height: 2.5;}
      p.small_leading {line-height: 1.0;}
    </style>
  </head>
  <body>
    <p class = "big_tracking">
      On the plains of hesitation [letter-spacing: 0.4em]
    </p> <p />
    <p class = "small_tracking">
       Bleach the bones of countless millions [letter-
       spacing: -0.08em]
    </p> <br />
    <p class = "big_between_words">
      Who at the dawn of victory [word-spacing: 0.4em]
    </p> <p />
    <p class = "small_between_words">
       Sat down to wait and waiting died [word-spacing: -0.1em]
    </p> <br />
    <p class = "big_leading">
      If you think CSS is simple, [line-height: 2.5] <br />
      You are quite mistaken
    </p> <br />
    <p class = "small_leading">
       If you think HTML5 is all old stuff, [line-height: 1.0]
       <br />
      You are quite mistaken
    </p>
  </body>
</html>
```

Figure 3.3 shows a display of `text_space.html`.

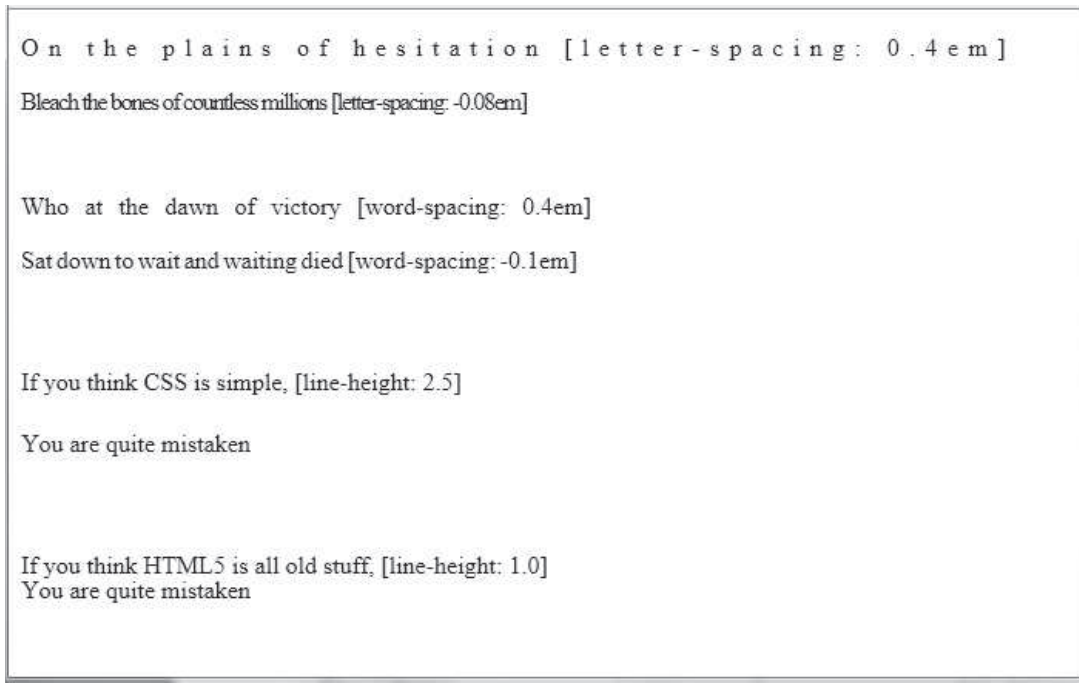


Figure 3.3 A display of `text_space.html`

3.7 List Properties

Two presentation details of lists can be specified in HTML documents: the shape of the bullets that precede the items in an unordered list and the sequencing values that precede the items in an ordered list. The `list-style-type` property is used to specify both of these. If `list-style-type` is set for a `ul` or an `ol` tag, it applies to all the list items in the list. If a `list-style-type` is set for an `li` tag, it only applies to that list item.

The `list-style-type` property of an unordered list can be set to `disc` (the default), `circle`, `square`, or `none`. A `disc` is a small filled circle, a `circle` is an unfilled circle, and a `square` is a filled square. For example, the following markup illustrates a document style sheet that sets the bullet type in all items in unordered lists to `square`:

```
<!-- bullets1 -->
<style type = "text/css">
  ul {list-style-type: square;}
</style>
...
<h3> Some Common Single-Engine Aircraft </h3>
  <ul>
```

```
    <li> Cessna Skyhawk </li>
    <li> Beechcraft Bonanza </li>
    <li> Piper Cherokee </li>
  </ul>
```

The following illustrates setting the style for individual list items:

```
<!-- bullets2 -->
<style type = "text/css">
  li.disc {list-style-type: disc;}
  li.square {list-style-type: square;}
  li.circle {list-style-type: circle;}
</style>
...
<h3> Some Common Single-Engine Aircraft </h3>
  <ul>
    <li class = "disc"> Cessna Skyhawk </li>
    <li class = "square"> Beechcraft Bonanza </li>
    <li class = "circle"> Piper Cherokee </li>
  </ul>
```

Figure 3.4 shows a browser display of these two lists.

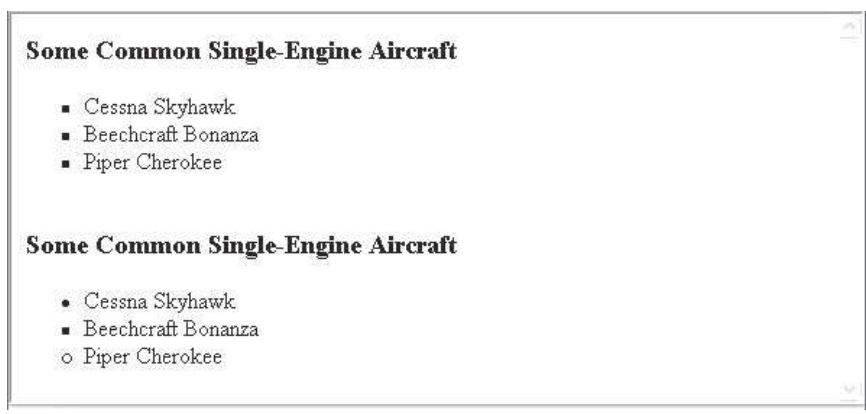


Figure 3.4 Examples of unordered lists

Bullets in unordered lists are not limited to discs, squares, and circles. An image can be used in a list item bullet. Such a bullet is specified with the `list-style-image` property, whose value is specified with the `url` form. For example, if `small_plane.gif` is a small image of an airplane that is stored in the same directory as the HTML document, it could be used as follows:

```
<style type = "text/css">
  li.image {list-style-image: url(small_airplane.gif);}
</style>
  ...
  <li class = "image"> Beechcraft Bonanza </li>
```

When ordered lists are nested, it is best to use different kinds of sequence values for the different levels of nesting. The `list-style-type` property can be used to specify the types of sequence values. Table 3.2 lists the different possibilities defined by CSS2.1.

Table 3.2 Possible sequencing value types for ordered lists in CSS2.1

Property Value	Sequence Type
`decimal`	Arabic numerals starting with `1`
`decimal-leading-zero`	Arabic numerals starting with `0`
`lower-alpha`	Lowercase letters
`upper-alpha`	Uppercase letters
`lower-roman`	Lowercase Roman numerals
`upper-roman`	Uppercase Roman numerals
`lower-greek`	Lowercase Greek letters
`lower-latin`	Same as `lower-alpha`
`upper-latin`	Same as `upper-alpha`
`armenian`	Traditional Armenian numbering
`georgian`	Traditional Georgian numbering
`None`	No bullet

The following example illustrates the use of different sequence value types in nested lists:

```
<!DOCTYPE html>
<!-- sequence_types.html
     An example to illustrate sequence type styles
     -->
<html lang = "en">
  <head>
    <title> Sequence types </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      ol {list-style-type: upper-roman;}
      ol ol {list-style-type: upper-alpha;}
      ol ol ol {list-style-type: decimal;}
    </style>
  </head>
```

```
  <body>
    <h3> Aircraft Types </h3>
    <ol>
      <li> General Aviation (piston-driven engines)
        <ol>
          <li> Single-Engine Aircraft
            <ol>
              <li> Tail wheel </li>
              <li> Tricycle </li>
            </ol>
          </li>
          <li> Dual-Engine Aircraft
            <ol>
              <li> Wing-mounted engines </li>
              <li> Push-pull fuselage-mounted engines </li>
            </ol>
          </li>
        </ol>
      </li>
      <li> Commercial Aviation (jet engines)
        <ol>
          <li> Dual-Engine
            <ol>
              <li> Wing-mounted engines </li>
              <li> Fuselage-mounted engines </li>
            </ol>
          </li>
          <li> Tri-Engine
            <ol>
              <li> Third engine in vertical stabilizer </li>
              <li> Third engine in fuselage </li>
            </ol>
          </li>
        </ol>
      </li>
    </ol>
  </body>
</html>
```

The document style sheet in this example appears to be ambiguous. The first selector, `ol`, would seem to apply to all ordered list elements. However, that is not the case, because longer contextual selectors have precedence over shorter selectors. So, the selector `ol ol ol` has precedence over `ol ol`, and `ol ol` has precedence over `ol`.

Figure 3.5 shows a browser display of `sequence_types.html`.

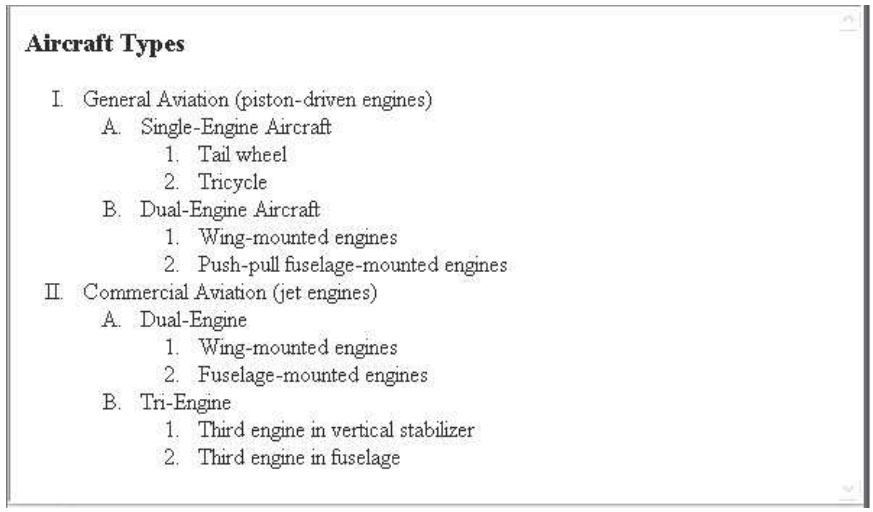


Figure 3.5 Display of `sequence_types.html`

3.8 Alignment of Text

The `text-indent` property can be used to indent the first line of a paragraph. This property takes either a length or a percentage value, as in the following markup:

```
<style type = "text/css">
  p.indent {text-indent: 2em}
</style>
...
<p class = "indent">
  Now is the time for all good Web developers to begin
  using cascading style sheets for all presentation
  details in their documents. No more deprecated tags
  and attributes, just nice, precise style sheets.
</p>
```

This paragraph would be displayed as shown in Figure 3.6.

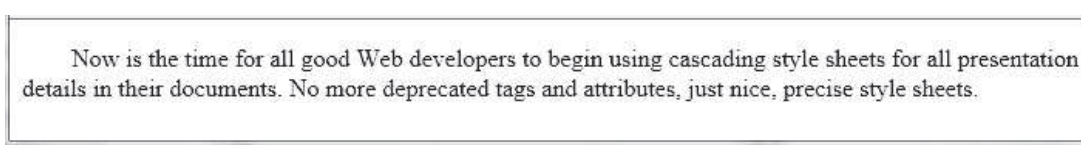


Figure 3.6 Indenting text

The `text-align` property, for which the most commonly used keyword values are `left`, `center`, `right`, and `justify`, is used to arrange text horizontally. For example, the following document-level style sheet entry causes the content of paragraphs to be aligned on the right margin:

```
p {text-align: right}
```

The default value for `text-align` is `left`.

The `float` property is used to specify that text should flow around some element, often an image or a table. The possible values for `float` are `left`, `right`, and `none`, which is the default. For example, suppose we want an image to be on the right side of the display and have text flow around the left side of the image. To specify this condition, the `float` property of the image is set to `right`. Because the default value for `text-align` is `left`, `text-align` need not be set for the text. In the following example, the text of a paragraph is specified to flow to the left of an image until the bottom of the image is reached, at which point the paragraph text flows across the whole window:

```
<!DOCTYPE html>
<!-- float.html
     An example to illustrate the float property
     -->
<html lang = "en">
  <head>
    <title> The float property </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      img {float: right;}
    </style>
  </head>
  <body>
    <p>
      <img src =  "c210new.jpg"  alt = "Picture of a Cessna 210" />
    </p>
    <p>
      This is a picture of a Cessna 210. The 210 is the flagship
      single-engine Cessna aircraft. Although the 210 began as a
      four-place aircraft, it soon acquired a third row of seats,
      stretching it to a six-place plane. The 210 is classified
      as a high-performance airplane, which means its landing
      gear is retractable and its engine has more than 200
      horsepower. In its first model year, which was 1960,
      the 210 was powered by a 260-horsepower fuel-injected
      six-cylinder engine that displaced 471 cubic inches.
```

```
        The 210 is the fastest single-engine airplane ever
        built by Cessna.
      </p>
    </body>
  </html>
```

When rendered by a browser, `float.html` might appear as shown in Figure 3.7, depending on the width of the browser display window.

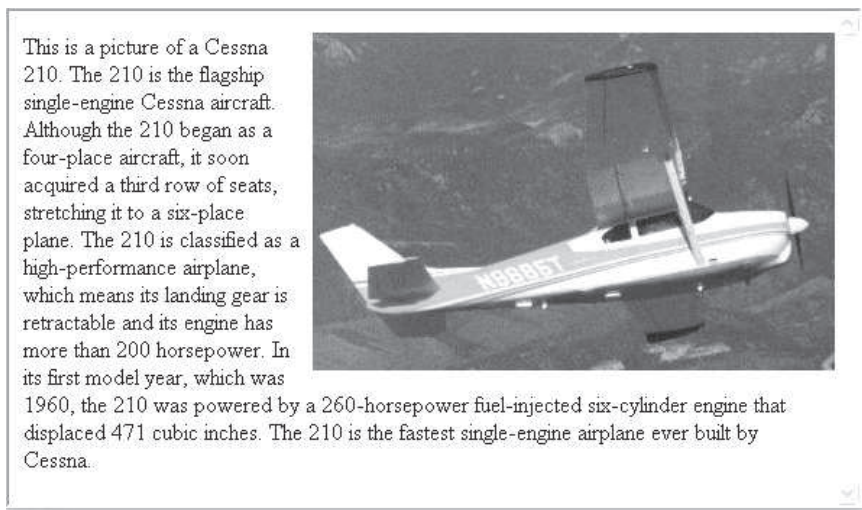


Figure 3.7 Display of `float.html`

3.9 Color

Over the last decade the issue of color in Web documents has become much more settled. In the past, one had to worry about the range of colors client machine monitors could display, as well as the range of colors browsers could handle. Now, however, there are few color limitations with the great majority of client machine monitors and browsers.

3.9.1 Color Groups

There are three groups of predefined colors that were designed for Web documents, the original group of seventeen named colors, which included far too few colors to be useful, a group of 147 named colors that are widely supported by browsers (see Appendix B), and the so-called Web palette (see http://www.web-source.net/216_color_chart.htm), which includes 216 named colors, which at one time were the only predefined colors that were widely supported

by browsers. Rather than being restricted to the use of only predefined named colors, contemporary professional Web designers are more likely to define their own colors.

3.9.2 Color Properties

The `color` property is used to specify the foreground color[15] of HTML elements. For example, consider the following description of a small table:

```
<style type = "text/css">
  th.red {color: red;}
  th.orange {color: orange;}
</style>
  ...
<table>
  <tr>
    <th class = "red"> Apple </th>
    <th class = "orange"> Orange </th>
    <th class = "orange"> Screwdriver </th>
  </tr>
</table>
```

The `background-color` property is used to set the background color of an element, where the element could be the whole body of the document. For example, consider the following paragraph element:

```
<style type = "text/css">
  p.standout {font-size: 2em; color: blue;
              background-color: magenta";}
</style>
...
  <p class = "standout">
  To really make it stand out, use a magenta background!
</p>
```

When displayed by a browser, this might appear as shown in Figure 3.8.

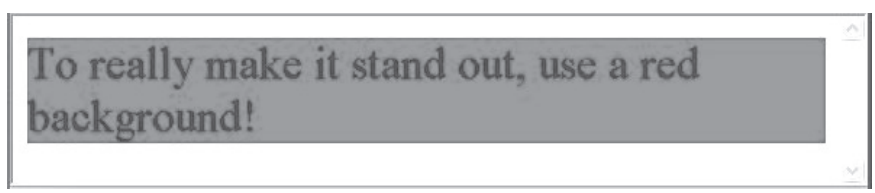


Figure 3.8 The `background-color` property

15. The foreground color of an element is the color in which it is displayed.

3.10 The Box Model

Virtually all document elements can have borders with various styles, such as color and width. Furthermore, the amount of space between the content of an element and its border, known as *padding*, can be specified, as well as the space between the border and an adjacent element, known as the *margin*. This model, called the *box model*, is shown in Figure 3.9.

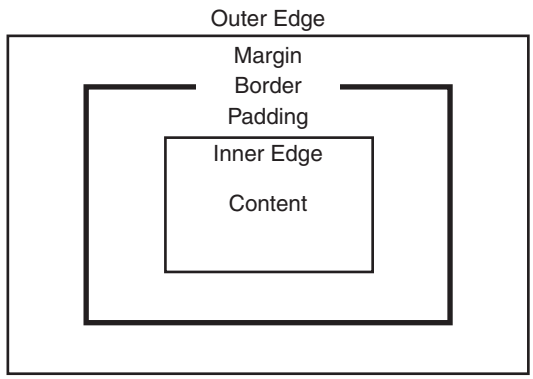


Figure 3.9 The box model

3.10.1 Borders

Every element has the `border-style` property, which controls whether the element's content has a border and also specifies the style of the border. CSS provides several different border styles, among them `dotted`, `dashed`, `solid`, and `double`. The default value for `border-style` is `none`, which is why the contents of elements normally do not have borders. The styles of one particular side of an element can be set with `border-top-style`, `border-bottom-style`, `border-left-style`, or `border-right-style`.

The `border-width` property is used to specify the thickness of a border. Its possible values are `thin`, `medium` (the default), `thick`, or a length value, which is in pixels. Setting `border-width` sets the thickness of all four sides of an element. The width of one particular border of an element can be specified with `border-top-width`, `border-bottom-width`, `border-left-width`, or `border-right-width`.

The color of a border is controlled by the `border-color` property. Once again, the individual borders of an element can be colored differently through the properties `border-top-color`, `border-bottom-color`, `border-left-color`, or `border-right-color`.

There is shorthand for setting the style properties of all four borders of an element, `border`. For example, we could have the following:

```
border: 5px solid blue;
```

This is equivalent to the following:

```
border_width: 5px; border-style: solid; border-color: blue;
```

The cells of a table can have borders, like any other element. We could specify borders on the cells with the following:

```
td, th {border: thin solid black;}
```

In many cases, we want just one border between table cells, rather than the default double borders that result from this specification (e.g., the right border of a cell and the left border of its neighbor cell to the right together form a double border). To get just one border between table cells, we set the table property `border-collapse` to `collapse` (its default value is `separate`).

The following document, `borders.html`, illustrates borders, using a table and a short paragraph as examples:

```
<!-- borders.html
     Examples of various borders
     -->
<html lang = "en">
  <head>
    <title> Borders </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      td, th {border: thin solid black;}
      table {border: thin solid black;
             border-collapse: collapse;
             border-top-width: medium;
             border-bottom-width: thick;
             border-top-color: red;
             border-bottom-color: blue;
             border-top-style: dotted;
             border-bottom-style: dashed;
            }
      p {border: thin dashed green;}
    </style>
  </head>
  <body>
    <table>
      <caption> Fruit Juice Drinks </caption>
      <tr>
        <th> </th>
        <th> Apple </th>
        <th> Orange </th>
        <th> Screwdriver </th>
      </tr>
```

```
      <tr>
        <th> Breakfast </th>
        <td> 0 </td>
        <td> 1 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Lunch </th>
        <td> 1 </td>
        <td> 0 </td>
        <td> 0 </td>
      </tr>
      <tr>
        <th> Dinner </th>
        <td> 0 </td>
        <td> 0 </td>
        <td> 1 </td>
      </tr>
    </table>
    <p>
      Now is the time for all good Web developers to
      learn to use style sheets.
    </p>
  </body>
</html>
```

Notice that if a table has borders that were specified with its `border` or `border-style` attribute, as well as border properties that are specified for one particular border, as in `borders.html`, those for the particular border override those of the original border. In `borders.html`, the table element uses its `border` attribute to set the border to `thin`, but the top and bottom borders are replaced by those specified with the `border-top` and `border-bottom` properties.

The display of `borders.html` is shown in Figure 3.10.

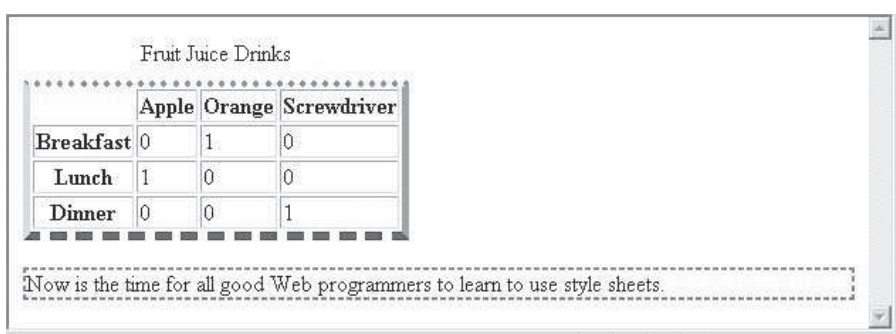


Figure 3.10 Borders

3.10.2 Margins and Padding

Recall from the box model that padding is the space between the content of an element and its border. The margin is the space between the border of an element and its neighbors. When there is no border, the margin plus the padding is the space between the content of an element and its neighbors. In this scenario, it may appear that there is no difference between padding and margins. However, there is a difference when the element has a background: The background extends into the padding, but not into the margin.

The margin properties are named `margin`, which applies to all four sides of an element, `margin-left`, `margin-right`, `margin-top`, and `margin-bottom`. The padding properties are named `padding`, which applies to all four sides, `padding-left`, `padding-right`, `padding-top`, and `padding-bottom`.

The following example, `marpads.html`, illustrates several combinations of margins and padding, both with and without borders:

```
<!DOCTYPE html>
<!-- marpads.html
     An example to illustrate margins and padding
     -->
<html lang = "en">
  <head>
    <title> Margins and Padding </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      p.one   {margin: 15px;
               padding: 15px;
               background-color: #C0C0C0;
               border-style: solid;
              }
      p.two   {margin: 5px;
               padding: 25px;
               background-color: #C0C0C0;
               border-style: solid;
              }
      p.three {margin: 25px;
               padding: 5px;
               background-color: #C0C0C0;
               border-style: solid;
              }
      p.four  {margin: 25px;
               background-color: #C0C0C0;}
```

```
        p.five  {padding: 25px;
                 background-color: #C0C0C0;
                }
      </style>
    </head>
    <body>
      <p>
        Here is the first line.
      </p>
      <p class = "one">
        Now is the time for all good Web programmers to
        learn to use style sheets. <br /> [margin = 15px,
        padding = 15px]
      </p>
      <p class = "two">
        Now is the time for all good Web programmers to
        learn to use style sheets. <br /> [margin = 5px,
        padding = 25px]
      </p>
      <p class = "three">
        Now is the time for all good Web programmers to
        learn to use style sheets. <br /> [margin = 25px,
        padding = 5px]
      </p>
      <p class = "four">
        Now is the time for all good Web programmers to
        learn to use style sheets. <br /> [margin = 25px,
        no padding, no border]
      </p>
      <p class = "five">
        Now is the time for all good Web programmers to
        learn to use style sheets. <br /> [padding = 25px,
        no margin, no border]
      </p>
      <p>
        Here is the last line.
      </p>
    </body>
  </html>
```

Figure 3.11 shows a browser display of `marpads.html`.

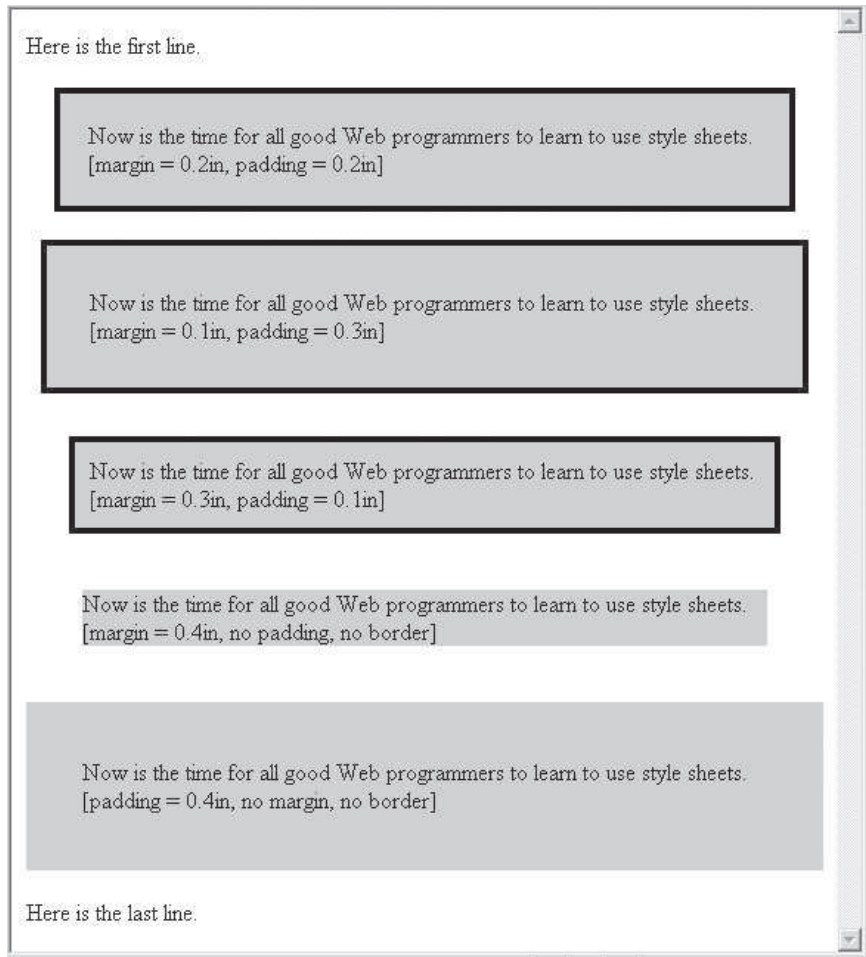


Figure 3.11 Display of `marpads.html`

3.11 Background Images

The `background-image` property is used to place an image in the background of an element. For example, an image of an airplane might be an effective background for text about the airplane. The following example, `back_image.html`, illustrates background images:

```
<!DOCTYPE html>
<!-- back_image.html
     An example to illustrate background images
     -->
<html lang = "en">
```

```
  <head>
    <title> Background images </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      body {background-image: url(../images/plane1.jpg);
            background-size: 375px 300px;}
      p {margin-left: 30px; margin-right: 30px;
         margin-top: 50px; font-size: 1.1em;}
    </style>
  </head>
  <body>
    <p>
      The Cessna 172 is the most common general aviation airplane
      in the world. It is an all-metal, single-engine piston,
      high-wing, four-place monoplane. It has fixed gear and is
      categorized as a non-high-performance aircraft. The current
      model is the 172R.
      The wingspan of the 172R is 36'1". Its fuel capacity is 56
      gallons in two tanks, one in each wing. The takeoff weight
      is 2,450 pounds. Its maximum useful load is 837 pounds.
      The maximum speed of the 172R at sea level is 142 mph.
      The plane is powered by a 360-cubic-inch gasoline engine
      that develops 160 horsepower. The climb rate of the 172R
      at sea level is 720 feet per minute.
    </p>
  </body>
</html>
```

Figure 3.12 shows a browser display of `back_image.html`.

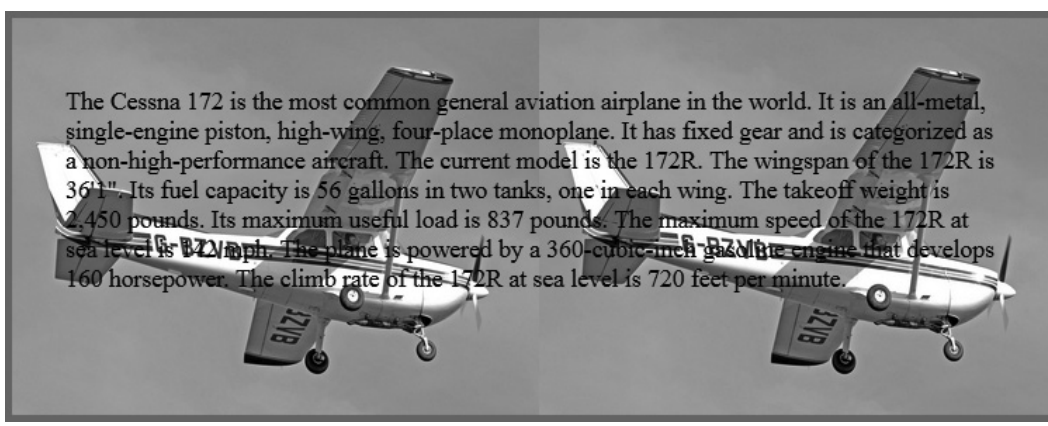


Figure 3.12 Display of `back_image.html`

Text over a background image can be difficult or even impossible to read if the image has areas that are nearly the same color as the text. Therefore, care must be taken in selecting the color of background images. In many cases, images with various textures in light-gray colors are best, assuming the text is in black.

In the example, notice that the background image is replicated as necessary to fill the area of the element. This replication is called *tiling*. Tiling can be controlled with the `background-repeat` property, which can take the value `repeat` (the default), `no-repeat`, `repeat-x`, or `repeat-y`. The `no-repeat` value specifies that just one copy of the image is to be displayed. The `repeat-x` value means that the image is to be repeated horizontally; `repeat-y` means that the image is to be repeated vertically. In addition, the position of a nonrepeated background image can be specified with the `background-position` property, which can take a large number of different values. The keyword values are `top`, `center`, `bottom`, `left`, and `right`, all of which can be used in combinations. It is easiest to use one keyword to specify the horizontal placement and one to specify the vertical placement, such as `top left`, `bottom right`, and `top center`. If only one keyword is given, the other is assumed to be `center`. So, `top` is equivalent to `top center` (or `center top`), and `left` is the same as `center left` (or `left center`).

3.12 The `<span>` and `<div>` Tags

In many situations, we want to apply special font properties to less than a whole paragraph of text. For example, it is often useful to have a word or phrase in a line appear in a different font size or color. The `<span>` tag is designed for just this purpose. Unlike most other tags, there is no default layout for the content of `<span>`. So, in the following example, the word `total` is not displayed differently from the rest of the paragraph:

```
<p>
  It sure is fun to be in <span> total </span>
  control of text
</p>
```

The purpose of `<span>` is to change property values of part of a line of content, as in the following example:

```
<style type = "text/css" >
  .spanred {font-size: 2em;
            font-family: Ariel; color: red;}
</style>
...
<p>
  It sure is fun to be in
  <span class = "spanred"> total </span>
  control of text
</p>
```

The display of this paragraph is shown in Figure 3.13.

Figure 3.13 The `<span>` tag

It is common for documents to have sections, each consisting of some number of paragraphs that have their own presentation styles. Using style classes on paragraphs, you can do this with what has already been discussed. It is more convenient, however, to be able to apply a style to a section of a document rather than to each paragraph. This can be done with the `<div>` tag. As with `<span>`, there is no implied layout for the content of the `<div>` tag, so its primary use is to specify presentation details for a section or division of a document.

Consider the following example, in which a division of a document is to use a specific paragraph style:

```
<div class = "primary">
  <p>
  ...
  </p>
  <p>
  ...
  </p>
  <p>
  ...
  </p>
</div>
```

The `span` and `div` elements are used in examples in Chapter 6.

Recall that HTML5 has several new elements that provide more detailed sectioning of a document than is possible with `div`.

3.13 Conflict Resolution

When there are two different values for the same property on the same element in a document, there is an obvious conflict that the browser (or other HTML processor) must resolve. So far, we have considered only one way this conflict can occur: when style sheets at two or more levels specify different values for the same property on the same element. This particular kind of conflict is resolved by the precedence of the three different levels of style sheets. Inline style sheets have precedence over document and external style sheets, and document style sheets have precedence over external style sheets. The following shows an external style sheet and an HTML document that has three paragraph elements. The first paragraph uses the

external style sheet to determine the font size. In the second, the document-level style sheet specifies the font size. In the third, although the document-level style sheet applies, the inline style sheet overrides it to specify the font size.

```
/* cstyle.css - an external style sheet
     for use with cascade.html
   */
  p {font-size: 0.8em;}

<!DOCTYPE html>
<!-- cascade.html
     An example to illustrate the three levels
     of style sheets
     -->
<html lang = "en">
  <head>
    <title> Style sheet levels </title>
    <meta charset = "utf-8" />
    <link rel = "stylesheet"  type = "text/css"
          href = "cstyle.css" />
    <style type = "text/css">
      p.docstyle {font-size: 1.2em;}
    </style>
  </head>
  <body>
    <p>
      Now is the time
    </p>
    <p class = "docstyle">
      for all good men
    </p>
    <p class = "docstyle"  style = "font-size: 1.6em">
      to come to the aid
    </p>
  </body>
</html>
```

Property-value conflicts can occur in several other ways. For example, a conflict may occur within a single style sheet. Consider the following style specifications, which are next to each other in the same document-level style sheet:

```
h3 {color: blue;}
body h3 {color: red;}
```

Both these specifications apply to all `h3` elements in the body of the document.

Another source of conflict can arise from the fact that there can be several different origins of the specification of property values. For example, they may

come from a style sheet written by the author of the document itself, but they may also come from the browser user and from the browser. For example, an FX user can set a minimum font size in the *Tools-Options-Advanced* window. Furthermore, browsers allow their users to write and use their own style sheets. Property values with different origins can have different levels of precedence.

Inheritance is another source of property-value conflicts; although as we already know, the inherited property value is always overridden by the property value given to the descendant element.

In addition, every property-value specification has a particular *specificity*, depending on the particular kind of selector that is used to set it, and those specificities have different levels of precedence. These different levels are used to resolve conflicts among different specifications.

Finally, property-value specifications can be marked as being important by including `!important` in the specification. For example, in the specification

```
p.special {font-style: italic !important; font-size: 1.2em}
```

`font-style: italic` is important, but `font-size: 1.2em,` is normal. Whether a specification has been marked as being important is called the *weight* of the specification. The weight can be either normal or important. Obviously, this is another way to specify the relative precedence that a specification should have in resolving conflicts.

The details of property-value conflict resolution, which are complex, will not be discussed here. Rather, what follows is a relatively brief overview of the process of property-value conflict resolution.

Conflict resolution is a multistage sorting process. The first step in the process is to gather the style specifications from the three possible levels of style sheets. These specifications are sorted into order by the relative precedence of the style sheet levels. Next, all the available specifications (those from style sheets, those from the user, and those from the browser) are sorted by origin and weight in accordance with the following list of rules, in which the first has the highest precedence:

1. Important declarations with user origin
2. Important declarations with author origin
3. Normal declarations with author origin
4. Normal declarations with user origin
5. Any declarations with browser (or other user agent) origin

Note that user-origin specifications are considered to have the highest precedence. The rationale for this approach is that such specifications often are declared because of some diminished capability of the user, most often a visual impairment.

If there are conflicts after this first sorting takes place, the next step in their resolution is a sort by specificity. This sort is based on the following rules, in which the first has the highest precedence:

1. id selectors
2. Class and pseudo class selectors

3. Contextual selectors (more element type names means that they are more specific)
4. Universal selectors

If there are still conflicts, they are resolved by giving precedence to the most recently seen specification. For this process, the specifications in an external style sheet are considered to occur at the point in the document where the link element or `@import` rule that references the external style sheet appears. For example, if a style sheet specifies the following, and there are no further conflicting specifications before the element is displayed, the value used will be the last (in this case, `0.9em`):

```
p {font-size: 1em}
p {font-size: 0.9em}
```

The whole sorting process that is used to resolve style specification conflicts is called *the cascade*. The name is apropos because the rules apply the lowest priority styles first and then cascade progressively to those with higher priorities.

Summary

Cascading style sheets were introduced to provide a consistent way to specify presentation details in HTML documents. Many of the style elements and attributes designed for specifying styles that had crept into HTML were deprecated in HTML 4.0 in favor of style sheets, which can appear at three levels: inline, which apply only to the content of one specific tag; document, which can apply to all appearances of specific tags in the body of a document; and external, which are stored in files by themselves and can apply to any number of documents. The property values in inline style sheets are specified in the string value of the `style` attribute. Document style sheets are specified in the content of a `style` element in the head of the document. External style sheets appear in separate files. Both document-level and external style specifications have the form of a list of style rules. Each style rule has a selector and a list of property-value pairs. The latter applies to all occurrences of the selected elements. There are a variety of selectors, such as simple, child, descendant, and id.

A style class, which is defined in the content of a `style` element, allows different occurrences of the same element to have different property values. A generic style-class specification allows elements with different names to use the same presentation style. A pseudo class takes effect when a particular event occurs. There are many different property-value forms, including lengths, percentage values, URLs, and colors. Several different properties are related to fonts. The `font-family` property specifies one or more font names. Because different browsers support different sets of fonts, there are five generic font names. Each browser supports at least one font in each generic category. The `font-size` property can specify a length value or one of a number of different named size categories. The `font-style` property can be set to `italic` or `normal`. The `font-weight` property is used to specify

the degree of boldness of text. The `font` property provides an abbreviated form for font-related properties. The `text-decoration` property is used to specify underlining, overlining, and line-through text.

The `letter-spacing` property is used to set the space between letters in words. The `word-spacing` property is used to set the space between words in text. The `line-height` property is used to set the amount of vertical space between lines of text.

The `list-style-type` property is used to specify the bullet form for items in unordered lists. It is also used to specify the sequence type for the items in ordered lists.

The foreground and background colors of the content of a document are specified by the `color` and `background-color` properties, respectively. Colors can be specified by name, by hex number, or by `rgb`.

The first line of a paragraph can be indented with `text-indent`. Text can be aligned with the `text-align` property, whose values are `left`, `right`, and `justify`, which means both left and right alignment. When the `float` property is set to `left` or `right` on an element, text can be made to flow around that element on the right or left, respectively, in the display window.

Borders can be specified to appear around any element, in any color and any of the forms—dotted, solid, dashed, or double. The margin, which is the space between the border (or the content of the element if it has no border) and the element's neighbors, can be set with the margin properties. The padding, which is the space between the content of an element and its border (or neighbors if it has no border), can be set with the padding properties.

When the cells of a table have borders, the double borders between cells can be eliminated with the `border-collapse` property.

The `background-image` property is used to place an image in the background of an element.

The `span` element provides a way to include an inline style sheet that applies to a range of text that is smaller than a line or a paragraph. The `div` element provides a way to define a section of a document that has its own style properties.

Conflict resolution for property values is a complicated process, using the origin of specifications, their specificity, inheritance, and the relative position of specifications.

Review Questions

3.1 What is the advantage of document-level style sheets over inline style sheets?

3.2 What is the purpose of external style sheets?

3.3 What attributes are required in a link to an external style sheet?

3.4 What is the format of an inline style sheet?

3.5 What is the format of a document-level style sheet, and where does the sheet appear?

3.6 What is the format of an external style sheet?

3.7 What is the form of comments within the rule list of a document-level style sheet?

3.8 What is the purpose of a style class selector?

3.9 What is the purpose of a generic class?

3.10 What is the difference between the two selectors `ol ul` and `ol > ul`?

3.11 Describe the two pseudo classes that are used exclusively for links.

3.12 Are keyword property values case sensitive or case insensitive?

3.13 Why is a list of font names given as the value of a `font-family` property?

3.14 What are the five generic fonts?

3.15 Why is it better to use `em` than `pt` for font sizes?

3.16 In what order must property values appear in the list of a `font` property?

3.17 In what ways can text be modified with `text-decoration`?

3.18 What are tracking and leading?

3.19 How is the `list-style-type` property used with unordered lists?

3.20 What are the possible values of the `list-style-type` property when it is used with ordered lists?

3.21 If you want text to flow around the right side of an image, which value, `right` or `left`, must be assigned to the `float` property of the image?

3.22 Why must background images be chosen with care?

3.23 What are the possible values for the `text-align` property?

3.24 What purpose does the `text-indent` property serve?

3.25 What properties are used to set margins around elements?

3.26 What are the three ways color property values can be specified?

3.27 If you want a background image to be repeated vertically but not horizontally, what value must be set to what property?

3.28 What properties and values must be used to put a dotted border around a text box if the border is red and thin on the left and blue and thick on the right?

3.29 What is the shorthand property for border styles?

3.30 What is the purpose of the `border-collapse` property?

3.31 What layout information does a `<span>` tag by itself indicate to the browser?

3.32 What is the purpose of the `div` element?

3.33 Which has higher precedence, an `id` selector or a universal selector (*)?

3.34 Which has higher precedence, a user-origin specification or a browser specification?

3.35 If there are two conflicting specifications in a document-level style sheet, which of the two has precedence?

Exercises

3.1 Create an external style sheet for the chapters of this book.

3.2 Create and test an HTML document that displays a table of football scores from a collegiate football conference in which the team names have one of the primary colors of their respective schools. The winning scores must appear larger and in a different font than the losing scores. The team names must be in a script font.

3.3 Create and test an HTML document that includes at least two images and enough text to precede the images, flow around them (one on the left and one on the right), and continue after the last image.

3.4 Create and test an HTML document that has at least a half page of text and a small box of text embedded on the left margin, with the main text flowing around the small box. The embedded text must appear in a smaller font and also must be set in italic.

3.5 Create and test an HTML document that has six short paragraphs of text that describe various aspects of the state in which you live. You must define three different paragraph styles, `p1`, `p2`, and `p3`. The `p1` style must use left and right margins of 20 pixels, a background color of pink, and a foreground color of blue. The `p2` style must use left and right margins of 30 pixels, a background color of black, and a foreground color of yellow. The `p3` style must use a text indent of 1 centimeter, a background color of green, and a foreground color of white. The first and fourth paragraphs must use `p1`, the second and fifth must use `p2`, and the third and sixth must use `p3`.

3.6 Create and test an HTML document that describes nested ordered lists of cars. The outer list must have three entries: compact, midsize, and sports. Inside each of these three lists there must be two sublists of body styles. The compact- and midsize-car sublists are two door and four door; the sports-car sublists are coupe and convertible. Each body-style sublist must have at least three entries, each of which is the make and model of a particular car that fits the category. The outer list must use uppercase Roman numerals, the middle lists must use uppercase letters, and the inner lists must use Arabic numerals. The background color for the compact-car list must be pink; for the midsize-car list, it must be blue; for the sports-car list, it must be red. All the styles must be in a document style sheet.

3.7 Rewrite the document of Exercise 3.6 to put all style-sheet information in an external style sheet. Validate your external style sheet with the W3C CSS validation service.

3.8 Rewrite the document of Exercise 3.6 to use inline style sheets only.

3.9 Create and test an HTML document that contains at least five lines of text from a newspaper story. Every verb in the text must be green, every noun must be blue, and every preposition must be yellow.

3.10 Create and test an HTML document that describes an unordered list of at least five popular books. The bullet for each book must be a small image of the book's cover. Find the images on the Web.

3.11 Use a document style sheet to modify the HTML document, `nested_lists.html` in Section 2.7.2 to make the different levels of lists different colors.

3.12 Using a document style sheet, modify the HTML document, `definition.html` in Section 2.7.3 to set the font in the `dt` elements to Courier `1em` font and the `dd` elements to Times Roman `1.1em` italic font.

CHAPTER

4

The Basics of JavaScript

This chapter takes the reader on a quick tour of the basics of JavaScript, introducing its most important concepts and constructs, but, for the sake of brevity, leaving out many of the details of the language. Topics discussed include the following: primitive data types and their operators and expressions, screen output and keyboard input, control statements, objects

and constructors, arrays, functions, and pattern matching. An experienced programmer should be able to become an effective JavaScript programmer by studying this brief chapter, along with Chapters 5 and 6. More comprehensive descriptions of JavaScript can be found in the numerous books devoted solely to the language.

4.1 Overview of JavaScript

This section discusses the origins of JavaScript, a few of its characteristics, and some of its uses. Included are a comparison of JavaScript and Java and a brief introduction to event-driven programming.

4.1.1 Origins

JavaScript, which was originally developed at Netscape by Brendan Eich, was initially named Mocha but soon after was renamed LiveScript. In late 1995 LiveScript became a joint venture of Netscape and Sun Microsystems, and its name again was changed, this time to JavaScript. A language standard for JavaScript was developed in the late 1990s by the European Computer Manufacturers Association (ECMA) as ECMA-262. This standard has also been approved by the International Standards Organization (ISO) as ISO-16262. The ECMA-262 standard is now in version 5. Most contemporary browsers implement languages that conform to ECMA-262 version 3 (at least). The current standard specification can be found at

`http://www.ecma-international.org/publications/standards/Ecma-262.htm`

The official name of the standard language is ECMAScript. Because it is nearly always called JavaScript elsewhere, we will use that term exclusively in this book. Microsoft's version of JavaScript is named JScript.

JavaScript can be divided into three parts: the core, client side, and server side. The *core* is the heart of the language, including its operators, expressions, statements, and subprograms. *Client-side* JavaScript is a collection of objects that support the control of a browser and interactions with users. For example, with JavaScript, a hypertext markup language (HTML) document can be made to respond to user inputs such as mouse clicks and keyboard use. *Server-side* JavaScript is a collection of objects that make the language useful on a Web server—for example, to support communication with a database management system. Server-side JavaScript is used far less frequently than client-side JavaScript. Therefore, this book does not cover server-side JavaScript.

This chapter introduces core JavaScript from the client-side perspective. Client-side JavaScript programming is covered in Chapters 5 and 6.

Client-side JavaScript is an HTML-embedded scripting language. We refer to every collection of JavaScript code as a *script*. An HTML document can include any number of embedded scripts.

4.1.2 JavaScript and Java

Although the name JavaScript appears to connote a close relationship with Java, JavaScript and Java are actually very different. One important difference is support for object-oriented programming. Although JavaScript is sometimes said to be an object-oriented language, its object model is quite different from that of Java and C++, as you will see in Section 4.2. In fact, JavaScript does not support the object-oriented software development paradigm.[1]

Java is a strongly typed language. Types are all known at compile time and the operand types are checked for compatibility. Variables in JavaScript need not be declared and are dynamically typed,[2] making compile-time type checking impossible. Another important difference between Java and JavaScript is that objects in Java are static in the sense that their collection of data members and methods is fixed at compile time. JavaScript objects are dynamic: The number of data members and methods of an object can change during execution.

The main similarity between Java and JavaScript is the syntax of their expressions, assignment statements, and control statements.

4.1.3 Uses of JavaScript

The original goal of JavaScript was to provide programming capability at both the server and the client ends of a Web connection. Since then, JavaScript has grown into a full-fledged programming language that can be used in a variety of application areas. As stated, this book focuses on client-side JavaScript.

Client-side JavaScript can serve as an alternative for some of what is done with server-side programming, in which computational capability resides on the server and is requested by the client. Because client-side JavaScript is embedded in HTML documents (either physically or logically) and is interpreted by the browser, this transfer of load from the often-overloaded server to the often-underloaded client can obviously benefit other clients. Client-side JavaScript cannot replace all server-side computing, however. In particular, although server-side software supports file operations, database access, and networking, client-side JavaScript supports none of these.

Interactions with users through form elements, such as buttons and menus, can be conveniently described in JavaScript. Because button clicks and mouse movements are easily detected with JavaScript, they can be used to trigger computations and provide feedback to the user. For example, when a user moves the mouse cursor from a text box, JavaScript can detect that movement and check the appropriateness of the text box's value (which presumably was just filled by the user). Even without forms, user interactions are both possible and simple to program in JavaScript. These interactions, which take place in dialog windows, include getting input from the user and allowing the user to make choices

1. Microsoft's JScript .NET is an extended dialect of JavaScript that does support object-oriented programming.

2. The type of dynamically typed variables cannot be determined before the script is executed.

through buttons. It is also easy to generate new content in the browser display dynamically with JavaScript.

Another interesting capability of JavaScript was made possible by the development of the Document Object Model (DOM), which allows JavaScript scripts to access and modify the style properties and content of the elements of a displayed HTML document, making formally static documents highly dynamic. Various techniques for designing dynamic HTML documents with JavaScript are discussed in Chapter 6.

Much of what JavaScript scripts typically do is event driven, meaning that the actions often are executed in response to the browser user's actions, among them mouse clicks and form submissions. This sort of computation supports user interactions through the HTML form elements on the client display. The mechanics of event-driven computation in JavaScript are discussed in detail in Chapter 5.

4.1.4 Browsers and HTML-JavaScript Documents

If an HTML document does not include embedded scripts, the browser reads the lines of the document and renders its window according to the tags, attributes, and content it finds. When a JavaScript script is encountered in the document, the browser uses its JavaScript interpreter to *execute* the script. Output from the script becomes the next markup to be rendered. When the end of the script is reached, the browser goes back to reading the HTML document and displaying its content.

There are two different ways to embed JavaScript in an HTML document: implicitly and explicitly. In *explicit embedding*, the JavaScript code physically resides in the HTML document. This approach has several disadvantages. First, mixing two completely different kinds of notation in the same document makes the document difficult to read. Second, in some cases, the person who creates and maintains the HTML is distinct from the person who creates and maintains the JavaScript. Having two different people doing two different jobs working on the same document can lead to many problems. To avoid these problems, the JavaScript can be placed in its own file, separate from the HTML document. This approach, called *implicit embedding*, has the advantage of hiding the script from the browser user. It also avoids the problem of hiding scripts from older browsers, a problem that is discussed later in this section. Except for the chapter's first simple example, which illustrates explicit embedding of JavaScript in an HTML document, all the JavaScript example scripts in this chapter are implicitly embedded.

When JavaScript scripts are explicitly embedded, they can appear in either part of an HTML document—the head or the body—depending on the purpose of the script. On the one hand, scripts that produce content only when requested or that react to user interactions are placed in the head of the document. Generally, these scripts contain function definitions and code associated with form elements such as buttons. On the other hand, scripts that are to be interpreted just once, when the interpreter finds them, are placed in the document body. Accordingly, the interpreter notes the existence of scripts that appear in the head of a document, but it does not interpret them while processing the head. Scripts

that are found in the body of a document are interpreted as they are found. When implicit embedding is used, these same guidelines apply to the markup code that references the external JavaScript files.

4.2 Object Orientation and JavaScript

As stated previously, JavaScript is not an object-oriented programming language. Rather, it is an object-based language. JavaScript does not have classes. Its objects serve both as objects and as models of objects. Without classes, JavaScript cannot have class-based inheritance, which is supported in object-oriented languages such as C++ and Java. It does, however, support a technique that can be used to simulate some of the aspects of inheritance. This is done with the prototype object; thus, this form of inheritance is called *prototype-based inheritance* (not discussed in this book).

Without class-based inheritance, JavaScript cannot support polymorphism. A polymorphic variable can reference related methods of objects of different classes within the same class hierarchy. A method call through such a polymorphic variable can be dynamically bound to the method in the object's class.[3]

Despite the fact that JavaScript is not an object-oriented language, much of its design is rooted in the concepts and approaches used in object-oriented programming. Specifically, client-side JavaScript deals in large part with documents and document elements, which are modeled with objects.

4.2.1 JavaScript Objects

In JavaScript, objects are collections of properties, which correspond to the members of classes in Java and C++. Each property is either a data property or a function or method property. Data properties appear in two categories: primitive values and references to other objects. (In JavaScript, variables that refer to objects are often called *objects* rather than *references*.) Sometimes we will refer to the data properties simply as *properties*; we often refer to the method properties simply as *methods* or *functions*. We prefer to call subprograms that are called through objects methods and subprograms that are not called through objects functions.

JavaScript uses nonobject types for some of its simplest types; these nonobject types are called *primitives*. Primitives are used because they often can be implemented directly in hardware, resulting in faster operations on their values (faster than if they were treated as objects). Primitives are like the simple scalar variables of non-object-oriented languages such as C, C++, Java, and JavaScript. All these languages have both primitives and objects; Primitives of JavaScript are described in Section 4.4.

All objects in a JavaScript program are indirectly accessed through variables. Such variables are like references in Java. All primitive values in JavaScript are

3. This technique is often called *dynamic binding*. It is an essential part of full support for object-oriented programming in a language.

accessed directly—these are like the scalar types in Java and C++. These are often called *value types*. The properties of an object are referenced by attaching the name of the property to the variable that references the object. For example, if `myCar` is a variable referencing an object that has the property `engine`, the `engine` property can be referenced with `myCar.engine`.

The root object in JavaScript is `Object`. It is the ancestor, through prototype inheritance, of all objects. `Object` is the most generic of all objects, having some methods but no data properties. All other objects are specializations of `Object`, and all inherit its methods (although they are often overridden).[4]

A JavaScript object appears, both internally and externally, as a list of property–value pairs. The properties are names; the values are data values or functions. All functions are objects and are referenced through variables. The collection of properties of a JavaScript object is dynamic: Properties can be added or deleted at any time during execution.

Every object is characterized by its collection of properties, although objects do not have types in any formal sense. Recall that `Object` is characterized by having no properties. Further discussions of objects appear in Sections 4.7 and 4.11.

4.3 General Syntactic Characteristics

In this book all JavaScript scripts are embedded, either directly or indirectly, in HTML documents. Scripts can appear directly as the content of a `<script>` tag. The `type` attribute of `<script>` must be set to `"text/javascript"`.[5] The JavaScript script can be indirectly embedded in an HTML document with the `src` attribute of a `<script>` tag, whose value is the name of a file that contains the script—for example,

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

Notice that the `script` element requires the closing tag, even though it has no content when the `src` attribute is included.

There are some situations when a small amount of JavaScript code is embedded in an HTML document. Furthermore, some documents have more than a few places where JavaScript code is embedded. Therefore, it is sometimes inconvenient and cumbersome to place all JavaScript codes in a separate file.

In JavaScript, identifiers, or names, are similar to those of other common programming languages. They must begin with a letter, an underscore (_), or a dollar sign ($).[6] Subsequent characters may be letters, underscores, dollar signs, or digits.

---

4. It sounds like a contradiction when we say that all objects inherit methods from `Object`, because we said earlier that `Object` has no properties. The resolution of this paradox lies in the design of prototype inheritance in JavaScript. Every object has a prototype object associated with it. It is `Object`'s prototype object that defines the methods that are inherited by all other objects.

5. With HTML5, the default value of the `type` attribute is `"text/javascript"`, so it need not be included in the script tag. However, we keep it in our documents in case some older browser might require it.

6. Dollar signs are not intended to be used by user-written scripts, although using them is valid.

There is no length limitation for identifiers. As with most C-based languages, the letters in a variable name in JavaScript are case sensitive, meaning that `FRIZZY`, `Frizzy`, `FrIzZy`, `frizzy`, and `friZZy` are all distinct names. However, by convention, programmer-defined variable names do not include uppercase letters.

JavaScript has 25 reserved words, which are listed in Table 4.1.

Table 4.1 JavaScript reserved words

`break`	`delete`	`function`	`return`	`typeof`
`case`	`do`	`if`	`switch`	`var`
`catch`	`else`	`in`	`this`	`void`
`continue`	`finally`	`instanceof`	`throw`	`while`
`default`	`for`	`new`	`try`	`with`

Besides its reserved words, another collection of words is reserved for future use in JavaScript—these can be found at the ECMA Web site. In addition, JavaScript has a large collection of predefined words, including `alert`, `open`, `java`, and `self`.

JavaScript has two forms of comments, both of which are used in other languages. First, whenever two adjacent slashes (`//`) appear on a line, the rest of the line is considered a comment. Second, `/*` may be used to introduce a comment, and `*/` to terminate it, in both single- and multiple-line comments.

Two issues arise regarding embedding JavaScript in HTML documents. First, some browsers that are still in use recognize the `<script>` tag but do not have JavaScript interpreters. Fortunately, these browsers simply ignore the contents of the script element and cause no problems. Second, a few browsers that are still in use are so old that they do not recognize the `<script>` tag. Such browsers would display the contents of the script element as if it were just text. It has been customary to enclose the contents of all script elements in HTML comments to avoid this problem. Because there are so few browsers that do not recognize the `<script>` tag, we believe that the issue no longer exists. However, the HTML validator can have problems with embedded JavaScript. When the embedded JavaScript happens to include recognizable tags—for example `<br />` tags in the output of the JavaScript—these tags can cause validation errors. Therefore, we still enclose embedded JavaScript in HTML comments when we explicitly embed JavaScript.

The HTML comment used to hide JavaScript uses the normal beginning syntax, `<!--`. However, the syntax for closing such a comment is special. It is the usual HTML comment closer, but it must be on its own line and must be preceded by two slashes (which makes it a JavaScript comment). The following HTML comment form hides the enclosed script from browsers that do not have JavaScript interpreters, as well as the validator, but makes it visible to browsers that do support JavaScript:

```
<!--
-- JavaScript script --
// -->
```

There are other problems with putting embedded JavaScript in comments in HTML documents. These problems are discussed in Chapter 6. The best solution to all these problems is to put all JavaScript scripts that are of significant size in separate files and embed them implicitly.

The use of semicolons in JavaScript is unusual. The JavaScript interpreter tries to make semicolons unnecessary, but it does not always work. When the end of a line coincides with what could be the end of a statement, the interpreter effectively inserts a semicolon there. But this implicit insertion can lead to problems. For example, consider the following lines of code:

```
return
x;
```

The interpreter will insert a semicolon after `return`, because `return` need not be followed by an expression, making `x` an invalid orphan. The safest way to organize JavaScript statements is to put each on its own line whenever possible and terminate each statement with a semicolon. If a statement does not fit on a line, be careful to break the statement at a place that will ensure that the first line does not have the form of a complete statement.

In the following complete, but trivial, HTML document that simply greets the client who requests it, there is but one line of JavaScript—the call to `write` through the `document` object to display the message:[7]

```
<!DOCTYPE.html>
<!-- hello.html
     A trivial hello world example of HTML/JavaScript
     -->
<html lang = "en">
  <head>
    <title> Hello world </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <script type = "text/javascript">
      <!--
      document.write("Hello, fellow Web programmers!");
      // -->
    </script>
  </body>
</html>
```

7. The `document` object and its `write` method are described in Section 4.5.

4.4 Primitives, Operations, and Expressions

The primitive data types, operations, and expressions of JavaScript are similar to those of other common programming languages. Therefore, our discussion of them is brief.

4.4.1 Primitive Types

JavaScript has five primitive types: Number, String, Boolean, Undefined, and Null.[8] Each primitive value has one of these types. JavaScript includes predefined objects that are closely related to the Number, String, and Boolean types, named `Number`, `String`, and `Boolean`, respectively. (Is this confusing yet?) These objects are called *wrapper objects*. Each contains a property that stores a value of the corresponding primitive type. The purpose of the wrapper objects is to provide properties and methods that are convenient for use with values of the primitive types. In the case of `Number`, the properties are more useful; in the case of `String`, the methods are more useful. Because JavaScript coerces values between the Number type primitive values and `Number` objects and between the String type primitive values and `String` objects, the methods of `Number` and `String` can be used on variables of the corresponding primitive types. In fact, in most cases you can simply treat Number and String type values as if they were objects.

The difference between primitives and objects is shown in the following example. Suppose that `prim` is a primitive variable with the value `17` and `obj` is a `Number` object whose property value is `17`. Figure 4.1 shows how `prim` and `obj` are stored.

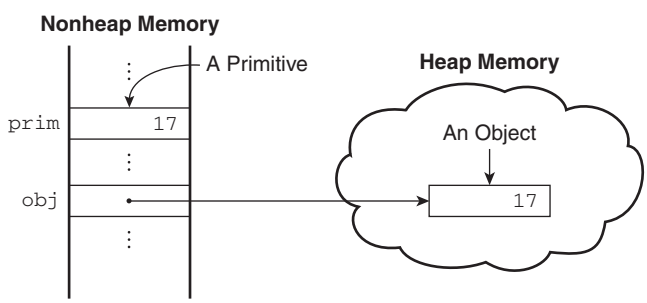


Figure 4.1 Primitives and objects

4.4.2 Numeric and String Literals

All numeric literals are primitive values of type Number. The Number type values are represented internally in double-precision floating-point form. Because there is only a single numeric data type, numeric values in JavaScript are often called *numbers*. Literal numbers in a script can have the forms of either integer

8. Undefined and Null are often called *trivial* types, for reasons that will be obvious when these types are discussed in Section 4.4.3.

or floating-point values. Integer literals are strings of digits. Floating-point literals can have decimal points, exponents, or both. Exponents are specified with an uppercase or a lowercase `e` and a possibly signed integer literal. The following are valid numeric literals:

```
72   7.2   .72   72.   7E2   7e2   .7e2   7.e2   7.2E-2
```

A string literal is a sequence of zero or more characters delimited by either single quotes (`'`) or double quotes (`"`). String literals can include characters specified with escape sequences, such as `\n` and `\t`. If you want an actual single-quote character in a string literal that is delimited by single quotes, the embedded single quote must be preceded by a backslash:

```
'You\'re the most freckly person I\'ve ever seen'
```

A double quote can be embedded in a double-quoted string literal by preceding it with a backslash. An actual backslash character in any string literal must be itself backslashed, as in the following example:

```
"D:\\bookfiles"
```

There is no difference between single-quoted and double-quoted literal strings. The null string (a string with no characters) can be denoted with either `''` or `""`. All string literals are primitive values.

4.4.3 Other Primitive Types

The only value of type Null is the reserved word `null`, which indicates no value. A variable is `null` if it has not been explicitly declared or assigned a value. If an attempt is made to use the value of a variable whose value is null, it will cause a runtime error.

The only value of type Undefined is `undefined`. Unlike `null`, there is no reserved word `undefined`. If a variable has been explicitly declared, but not assigned a value, it has the value `undefined`. If the value of an undefined variable is displayed, the word `undefined` is displayed.

The only values of type Boolean are `true` and `false`. These values are usually computed as the result of evaluating a relational or Boolean expression (see Section 4.6.1). The existence of both the Boolean primitive type and the `Boolean` object can lead to some confusion (also discussed in Section 4.6.1).

4.4.4 Declaring Variables

One of the characteristics of JavaScript that sets it apart from most common nonscripting programming languages is that it is dynamically typed. This means that a variable can be used for anything. Variables are not typed; values are. A variable can have the value of any primitive type, or it can be a reference to any object. The type of the value of a particular appearance of a variable in a program can be determined by the interpreter. In many cases, the interpreter converts the type of a value to whatever is needed for the context in which it appears.

A variable can be declared either by assigning it a value, in which case the interpreter implicitly declares it to be a variable, or by listing it in a declaration statement that begins with the reserved word `var`. Initial values can be included in a `var` declaration, as with some of the variables in the following declaration:

```
var counter,
    index,
    pi = 3.14159265,
    quarterback = "Elway",
    stop_flag = true;
```

We recommend that all variables be explicitly declared.

As stated previously, a variable that has been declared but not assigned a value has the value `undefined`.

4.4.5 Numeric Operators

JavaScript has the typical collection of numeric operators: the binary operators `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for modulus. The unary operators are plus (`+`), negate (`-`), decrement (`--`), and increment (`++`). The increment and decrement operators can be either prefix or postfix.[9] As with other languages that have the increment and decrement unary operators, the prefix and postfix uses are not always equivalent. Consider an expression consisting of a single variable and one of these operators. If the operator precedes the variable, the value of the variable is changed and the expression evaluates to the new value. If the operator follows the variable, the expression evaluates to the current value of the variable and then the value of the variable is changed. For example, if the variable `a` has the value `7`, the value of the following expression is `24`:

```
(++a) * 3
```

But the value of the following expression is `21`:

```
(a++) * 3
```

In both cases, `a` is set to `8`.

All numeric operations are done in double-precision floating point.

The *precedence rules* of a language specify which operator is evaluated first when two operators with different precedences are adjacent in an expression. Adjacent operators are separated by a single operand. For example, in the following code, `*` and `+` are adjacent:

```
a * b + 1
```

9. *Prefix* means that the operator precedes its operand; *postfix* means that the operator follows its operand.

The *associativity rules* of a language specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression. The precedence and associativity of the numeric operators of JavaScript are given in Table 4.2.

Table 4.2 Precedence and associativity of the numeric operators

Operator*	Associativity
++, --, unary -, unary +	Right (though it is irrelevant)
*, /, %	Left
Binary +, binary -	Left

*The first operators listed have the highest precedence.

As examples of operator precedence and associativity, consider the following code:

```
var a = 2,
    b = 4,
    c,
    d;
c = 3 + a * b;
// * is first, so c is now 11 (not 24)
d = b / a / 2;
// / associates left, so d is now 1 (not 4)
```

Parentheses can be used to force any desired precedence. For example, the addition will be done before the multiplication in the following expression:

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

4.4.6 The `Math` Object

The `Math` object provides a collection of properties of `Number` objects and methods that operate on `Number` objects. The `Math` object has methods for the trigonometric functions, such as `sin` (for sine) and `cos` (for cosine), as well as for other commonly used mathematical operations. Among these are `floor`, to truncate a number; `round`, to round a number; and `max`, to return the largest of two given numbers. The `floor` and `round` methods are used in the example script in Section 4.10. All the `Math` methods are referenced through the `Math` object, as in `Math.sin(x)`.

Table 4.3 Properties of `Number`

Property	Meaning
`MAX_VALUE`	Largest representable number on the computer being used
`MIN_VALUE`	Smallest representable number on the computer being used
`NaN`	Not a number
`POSITIVE_INFINITY`	Special value to represent infinity
`NEGATIVE_INFINITY`	Special value to represent negative infinity
`PI`	The value of π

4.4.7 The `Number` Object

The `Number` object includes a collection of useful properties that have constant values. Table 4.3 lists the properties of `Number`. These properties are referenced through `Number`. For example,

```
Number.MIN_VALUE
```

references the smallest representable number on the computer being used.

Any arithmetic operation that results in an error (e.g., division by zero) or that produces a value that cannot be represented as a double-precision floating-point number, such as a number that is too large (an overflow), returns the value "not a number," which is displayed as `NaN`. If `NaN` is compared for equality against any number, the comparison fails. Surprisingly, in a comparison, `NaN` is not equal to itself. To determine whether a variable has the `NaN` value, the predefined predicate function `isNaN()` must be used. For example, if the variable `a` has the `NaN` value, `isNaN(a)` returns `true`.

The `Number` object has a method, `toString`, which it inherits from `Object` but overrides. The `toString` method converts the number through which it is called to a string. Because numeric primitives and `Number` objects are always coerced to the other when necessary, `toString` can be called through a numeric primitive, as in the following code:

```
var price = 427,
    str_price;
...
str_price = price.toString();
```

4.4.8 The String Catenation Operator

JavaScript strings are not stored or treated as arrays of characters; rather, they are unit scalar values. String catenation is specified with the operator denoted by a plus sign (+). For example, if the value of `first` is `"Freddie"`, the value of the following expression is `"Freddie Freeloader"`:

```
first + " Freeloader"
```

4.4.9 Implicit Type Conversions

The JavaScript interpreter performs several different implicit type conversions. Such conversions are called *coercions*. In general, when a value of one type is used in a position that requires a value of a different type, JavaScript attempts to convert the value to the type that is required. The most common examples of these conversions involve primitive string and number values.

If either operand of a + operator is a string, the operator is interpreted as a string catenation operator. If the other operand is not a string, it is coerced to a string. For example, consider the following expression:

```
"August " + 1977
```

In this expression, because the left operand is a string, the operator is considered to be a catenation operator. This forces string context on the right operand, so the right operand is implicitly converted to a string. Therefore, the expression evaluates to

```
"August 1997"
```

The number `1977` in the following expression is also coerced to a string:

```
1977 + "August"
```

Now consider the following expression:

```
7 * "3"
```

In this expression, the operator is one that is used only with numbers. This forces numeric context on the right operand. Therefore, JavaScript attempts to convert it to a number. In this example, the conversion succeeds, and the value of the expression is `21`. If the second operand were a string that could not be converted to a number, such as `"August"`, the conversion would produce `NaN`, which would then be the value of the expression.

When used as a number, `null` is `0`. Unlike its usage in C and C++, however, `null` is not the same as `0`. When used as a number, `undefined` is interpreted as `NaN`. (See Section 4.4.7.)

4.4.10 Explicit Type Conversions

There are several different ways to force type conversions, primarily between strings and numbers. Strings that contain numbers can be converted to numbers with the `String` constructor, as in the following statement:

```
var str_value = String(value);
```

This conversion can also be done with the `toString` method, which has the advantage that it can be given a parameter to specify the base of the resulting

number (although the base of the number to be converted is taken to be decimal). An example of such a conversion is

```
var num = 6;
var str_value = num.toString();
var str_value_binary = num.toString(2);
```

In the first conversion, the result is `"6"`; in the second, it is `"110"`.

A number also can be converted to a string by catenating it with the empty string.

Strings can be explicitly converted to numbers in several different ways. One way is with the `Number` constructor, as in the following statement:

```
var number = Number(aString);
```

The same conversion could be specified by subtracting zero from the string, as in the following statement:

```
var number = aString - 0;
```

Both these conversions have the following restriction: The number in the string cannot be followed by any character except a space. For example, if the number happens to be followed by a comma, the conversion will not work. JavaScript has two predefined string functions that do not have this problem. The two, `parseInt` and `parseFloat`, are not `String` methods, so they are not called through `String` objects. They operate on the strings given as parameters. The `parseInt` function searches its string parameter for an integer literal. If one is found at the beginning of the string, it is converted to a number and returned. If the string does not begin with a valid integer literal, `NaN` is returned. The `parseFloat` function is similar to `parseInt`, but it searches for a floating-point literal, which could have a decimal point, an exponent, or both. In both `parseInt` and `parseFloat`, the numeric literal could be followed by any nondigit character.

Because of the coercions JavaScript normally does, as discussed in Section 4.4.9, `parseInt` and `parseFloat` often are not needed.

4.4.11 `String` Properties and Methods

Because JavaScript coerces primitive string values to and from `String` objects when necessary, the differences between the `String` object and the String type have little effect on scripts. `String` methods can always be used through String primitive values, as if the values were objects. The `String` object includes one property, `length`, and a large collection of methods.

The number of characters in a string is stored in the `length` property as follows:

```
var str = "George";
var len = str.length;
```

In this code, `len` is set to the number of characters in `str`, namely, `6`. In the expression `str.length`, `str` is a primitive variable, but we treated it as if it were

an object (referencing one of its properties). In fact, when `str` is used with the `length` property, JavaScript implicitly builds a temporary `String` object with a property whose value is that of the primitive variable. After the second statement is executed, the temporary `String` object is discarded.

A few of the most commonly used `String` methods are shown in Table 4.4.

Table 4.4 `String` methods

Method	Parameter	Result
`charAt`	A number	Returns the character in the `String` object that is at the specified position
`indexOf`	One-character string	Returns the position in the `String` object of the parameter
`substring`	Two numbers	Returns the substring of the `String` object from the first parameter position to the second
`toLowerCase`	None	Converts any uppercase letters in the string to lowercase
`toUpperCase`	None	Converts any lowercase letters in the string to uppercase

Note that, for the `String` methods, character positions start at zero. For example, suppose `str` has been defined as follows:

```
var str = "George";
```

Then the following expressions have the values shown:

```
str.charAt(2)  is 'o'
str.indexOf('r')  is 3
str.substring(2, 4)  is 'org'
str.toLowerCase()  is 'george'
```

Several `String` methods associated with pattern matching are described in Section 4.12.

4.4.12 The `typeof` Operator

The `typeof` operator returns the type of its single operand. This operation is quite useful in some circumstances in a script. `typeof` produces `"number"`, `"string"`, or `"boolean"` if the operand is of primitive type Number, String, or Boolean, respectively. If the operand is an object or `null`, `typeof` produces `"object"`. This illustrates a fundamental characteristic of JavaScript: Objects do not have types. If the operand is a variable that has not been assigned a value, `typeof` produces `"undefined"`, reflecting the fact that

variables themselves are not typed. Notice that the `typeof` operator always returns a string. The operand for `typeof` can be placed in parentheses, making it appear to be a function. Therefore, `typeof x` and `typeof(x)` are equivalent.

4.4.13 Assignment Statements

The assignment statement in JavaScript is exactly like the assignment statement in other common C-based programming languages. There is a simple assignment operator, denoted by =, and a host of compound assignment operators, such as += and /=. For example, the statement

```
a += 7;
```

means the same as

```
a = a + 7;
```

In considering assignment statements, it is important to remember that JavaScript has two kinds of values: primitives and objects. A variable can refer to a primitive value, such as the number `17`, or an object, as shown in Figure 4.1. Objects are allocated on the heap, and variables that refer to them are *reference* variables. When used to refer to an object, a variable stores an address only. Therefore, assigning the address of an object to a variable is fundamentally different from assigning a primitive value to a variable.

4.4.14 The `Date` Object

There are occasions when information about the current date and time is useful in a program. Likewise, sometimes it is convenient to be able to create objects that represent a specific date and time and then manipulate them. These capabilities are available in JavaScript through the `Date` object and its rich collection of methods. In what follows, we describe this object and some of its methods.

A `Date` object is created with the `new` operator and the `Date` constructor, which has several forms. Because we focus on uses of the current date and time, we use only the simplest `Date` constructor, which takes no parameters and builds an object with the current date and time for its properties. For example, we might have

```
var today = new Date();
```

The date and time properties of a `Date` object are in two forms: local and Coordinated Universal Time (UTC), which was formerly named Greenwich Mean Time. We deal only with local time in this section.

Table 4.5 shows the methods, along with the descriptions, that retrieve information from a `Date` object.

Table 4.5 Methods for the `Date` object

Method	Returns
`toLocaleString`	A string of the `Date` information
`getDate`	The day of the month
`getMonth`	The month of the year, as a number in the range from 0 to 11
`getDay`	The day of the week, as a number in the range from 0 to 6
`getFullYear`	The year
`getTime`	The number of milliseconds since January 1, 1970
`getHours`	The hour, as a number in the range from 0 to 23
`getMinutes`	The minute, as a number in the range from 0 to 59
`getSeconds`	The second, as a number in the range from 0 to 59
`getMilliseconds`	The millisecond, as a number in the range from 0 to 999

The use of the `Date` object is illustrated in Section 4.6.

4.5 Screen Output and Keyboard Input

A JavaScript script is interpreted when the browser finds the script or a reference to a separate script file in the body of an HTML document. Thus, the normal output screen for JavaScript is the same as the screen in which the content of the host HTML document is displayed. JavaScript models the HTML document with the `Document` object. The window in which the browser displays an HTML document is modeled with the `Window` object. The `Window` object includes two properties, `document` and `window`. The `document` property refers to the `Document` object. The `window` property is self-referential; it refers to the `Window` object.

The `Document` object has several properties and methods. The most interesting and useful of its methods, at least for now, is `write`, which is used to create output, which is dynamically created HTML document content. This content is specified in the parameter of `write`. For example, if the value of the variable `result` is `42`, the following statement produces the screen shown in Figure 4.2:

```
document.write("The result is: ", result, "<br />");
```

Figure 4.2 An example of the output of `document.write`

Because `write` is used to create markup, the only useful punctuation in its parameter is in the form of HTML tags. Therefore, the parameter of `write` often includes `<br />`. The `writeln` method implicitly adds `"\n"` to its parameter, but since browsers ignore line breaks when displaying HTML, it has no effect on the output.[10]

The parameter of `write` can include any HTML tags and content. The parameter is simply given to the browser, which treats it exactly like any other part of the HTML document. The `write` method actually can take any number of parameters. Multiple parameters are catenated and placed in the output.

As stated previously, the `Window` object is the JavaScript model for the browser window. `Window` includes three methods that create dialog boxes for three specific kinds of user interactions. The default object for JavaScript is the `Window` object currently being displayed, so calls to these methods need not include an object reference. The three methods—`alert`, `confirm`, and `prompt`—which are described in the following paragraphs, often are used for debugging rather than as part of a servable document.

The `alert` method opens a dialog window and displays its parameter in that window. It also displays an *OK* button. The string parameter of `alert` is not HTML code; it is plain text. Therefore, the string parameter of `alert` may include `\n` but never should include `<br />`. As an example of an `alert`, consider the following code, in which we assume that the value of `sum` is `42`.

```
alert("The sum is:" + sum + "\n");
```

This call to `alert` produces the dialog window shown in Figure 4.3.

Figure 4.3 An example of the output of `alert`

The `confirm` method opens a dialog window in which the method displays its string parameter, along with two buttons: *OK* and *Cancel*. `confirm` returns a Boolean value that indicates the user's button input: `true` for *OK* and `false` for *Cancel*. This method is often used to offer the user the choice of continuing

10. The `writeln` method is useful only if the browser is used to view a non-HTML document, which is rarely done.

some process. For example, the following statement produces the screen shown in Figure 4.4:

```
var question =
    confirm("Do you want to continue this download?");
```

After the user presses one of the buttons in the `confirm` dialog window, the script can test the variable, `question`, and react accordingly.

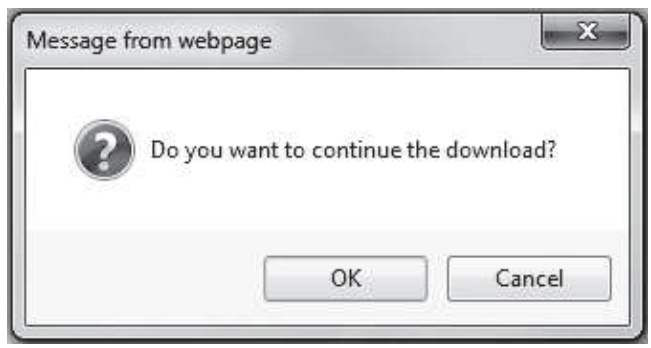


Figure 4.4 An example of the output of `confirm`

The `prompt` method creates a dialog window that contains a text box used to collect a string of input from the user, which `prompt` returns as its value. As with `confirm`, this window also includes two buttons: *OK* and *Cancel*. `prompt` takes two parameters: the string that prompts the user for input and a default string in case the user does not type a string before pressing one of the two buttons. In many cases, an empty string is used for the default input. Consider the following example:

```
name = prompt("What is your name?", "");
```

Figure 4.5 shows the screen created by this call to `prompt`.

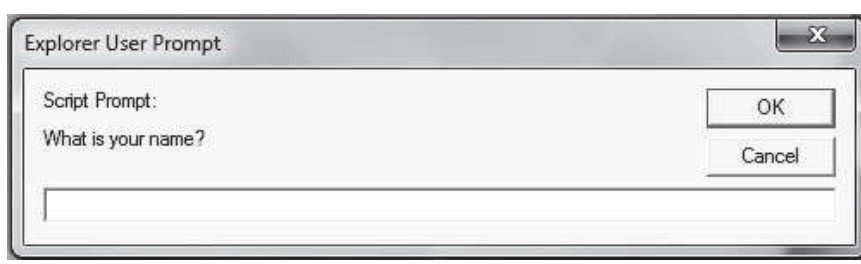


Figure 4.5 An example of the output of `prompt`

`alert`, `prompt`, and `confirm` cause the browser to wait for a user response. In the case of `alert`, the *OK* button must be clicked for the JavaScript interpreter to continue. The `prompt` and `confirm` methods wait for either *OK* or *Cancel* to be clicked.

The following example of HTML and JavaScript files—`roots.html` and `roots.js`—illustrates some of the JavaScript features described so far. The JavaScript script gets the coefficients of a quadratic equation from the user with `prompt` and computes and displays the real roots of the given equation. If the roots of the equation are not real, the value `NaN` is displayed. This value comes from the `sqrt` function, which returns `NaN` when the function is given a negative parameter. This result corresponds mathematically to the equation not having real roots.

```
<!DOCTYPE html>
<!-- roots.html
     A document for roots.js
     -->
<html lang = "en">
  <head>
    <title> roots.html </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <script type = "text/javascript"  src = "roots.js" >
    </script>
  </body>
</html>
// roots.js
//   Compute the real roots of a given quadratic
//   equation. If the roots are imaginary, this script
//   displays NaN, because that is what results from
//   taking the square root of a negative number

// Get the coefficients of the equation from the user
var a = prompt("What is the value of 'a'? \n", "");
var b = prompt("What is the value of 'b'? \n", "");
var c = prompt("What is the value of 'c'? \n", "");

// Compute the square root and denominator of the result
var root_part = Math.sqrt(b * b - 4.0 * a * c);
var denom = 2.0 * a;

// Compute and display the two roots
var root1 = (-b + root_part) / denom;
var root2 = (-b - root_part) / denom;
document.write("The first root is: ", root1, "<br />");
document.write("The second root is: ", root2, "<br />");
```

In the examples in the remainder of this chapter, the HTML documents that use the associated JavaScript files are not shown.

4.6 Control Statements

This section introduces the flow-control statements of JavaScript. Before discussing the control statements, we must describe control expressions, which provide the basis for controlling the order of execution of statements. Once again, the similarity of these JavaScript constructs to their counterparts in Java and C++ makes them easy to learn for those who are familiar with one of those languages.

Control statements often require some syntactic container for sequences of statements whose execution they are meant to control. In JavaScript, that container is the compound statement. A *compound statement* in JavaScript is a sequence of statements delimited by braces. A *control construct* is a control statement together with the statement or compound statement whose execution it controls.

Unlike several related languages, JavaScript does not allow compound statements to create local variables. If a variable is declared in a compound statement, access to it is not confined to that compound statement. Such a variable is visible in the whole HTML document.[11] Local variables are discussed in Section 4.9.2.

4.6.1 Control Expressions

The expressions upon which statement flow control can be based include primitive values, relational expressions, and compound expressions. The result of evaluating a control expression is one of the Boolean values `true` or `false`. If the value of a control expression is a string, it is interpreted as `true` unless it is either the empty string (`""`) or a zero string (`"0"`). If the value is a number, it is `true` unless it is zero (`0`). If the special value, `NaN`, is interpreted as a Boolean, it is false. If `undefined` is used as a Boolean, it is false. When interpreted as a Boolean, `null` is `false`. When interpreted as a number, `true` has the value `1` and `false` has the value `0`.

A relational expression has two operands and one relational operator. Table 4.6 lists the relational operators.

Table 4.6 Relational operators

Operation	Operator
Is equal to	`==`
Is not equal to	`!=`
Is less than	`<`
Is greater than	`>`
Is less than or equal to	`<=`
Is greater than or equal to	`>=`
Is strictly equal to	`===`
Is strictly not equal to	`!==`

11. The only exception to this rule is if the variable is declared in a function.

If the two operands in a relational expression are not of the same type and the operator is neither `===` nor `!==`, JavaScript will attempt to convert the operands to a single type. In the case in which one operand is a string and the other is a number, JavaScript attempts to convert the string to a number. If one operand is Boolean and the other is not, the Boolean value is converted to a number (`1` for `true`, `0` for `false`).

The last two operators in Table 4.6 disallow type conversion of either operand. Thus, the expression `"3" === 3` evaluates to `false`, while `"3" == 3` evaluates to `true`.

Comparisons of variables that reference objects are rarely useful. If `a` and `b` reference different objects, `a == b` is never true, even if the objects have identical properties. `a == b` is true only if `a` and `b` reference the same object.

JavaScript has operators for the AND, OR, and NOT Boolean operations. These are `&&` (AND), `||` (OR), and `!` (NOT). Both `&&` and `||` are short-circuit operators, as they are in Java and C++. This means that if the value of the first operand of either `||` or `&&` determines the value of the expression, the second operand is not evaluated and the Boolean operator does nothing. JavaScript also has bitwise operators, but they are not discussed in this book.

The properties of the object `Boolean` must not be confused with the primitive values `true` and `false`. If a `Boolean` object is used as a conditional expression, it evaluates to `true` if it has any value other than `null` or `undefined`. The `Boolean` object has a method, `toString`, which it inherits from `Object`, that converts the value of the object through which it is called to one of the strings `"true"` or `"false"`.

The precedence and associativity of all operators discussed so far in this chapter are shown in Table 4.7.

Table 4.7 Operator precedence and associativity

Operators*	Associativity
`++`, `--`, unary `-`	Right
`*`, `/`, `%`	Left
`+`, `-`	Left
`>`, `<`, `>=` , `<=`	Left
`==`, `!=`	Left
`===`, `!==`	Left
`&&`	Left
`||`	Left
`=`, `+=`, `-=`, `*=`, `/=`, `&&=`, `||=`, `%=`	Right

*Highest-precedence operators are listed first.

4.6.2 Selection Statements

The selection statements (`if-then` and `if-then-else`) of JavaScript are similar to those of the common programming languages. Either single statements or compound statements can be selected—for example,

```
if (a > b)
    document.write("a is greater than b <br />");
else {
    a = b;
    document.write("a was not greater than b <br />",
                   "Now they are equal <br />");
}
```

4.6.3 The `switch` Statement

JavaScript has a `switch` statement that is similar to that of Java. The form of this construct is as follows:

```
switch (expression) {
    case value_1:
        // statement(s)
    case value_2:
        // statement(s)
    ...
    [default:
        // statement(s) ]
}
```

In any `case` segment, the *statement(s)* can be either a sequence of statements or a compound statement.

The semantics of a `switch` construct is as follows: The expression is evaluated when the `switch` statement is reached in execution. The value is compared with the values in the cases in the construct (those values that immediately follow the `case` reserved words). If one matches, control is transferred to the statement immediately following that case value. Execution then continues through the remainder of the construct. In the great majority of situations, it is intended that only the statements in one case be executed in each execution of the construct. To implement this option, a `break` statement appears as the last statement in each sequence of statements following a case. The `break` statement is exactly like the `break` statement in Java and C++: It transfers control out of the compound statement in which it appears.

The control expression of a `switch` statement could evaluate to a number, a string, or a Boolean value. Case labels also can be numbers, strings, or Booleans, and different case values can be of different types. Consider the following script, which includes a `switch` construct:

```
// borders2.js
//   An example of a switch statement for table border
//   size selection
var bordersize;
var err = 0;
bordersize = prompt("Select a table border size: " +
                    "0 (no border), " +
                    "1 (1 pixel border), " +
                    "4 (4 pixel border), " +
                    "8 (8 pixel border), ");

switch (bordersize) {
  case "0": document.write("<table>");
            break;
  case "1": document.write("<table border = '1'>");
            break;
  case "4": document.write("<table border = '4'>");
            break;
  case "8": document.write("<table border = '8'>");
            break;
  default: {
            document.write("Error - invalid choice: ",
                           bordersize, "<br />");
            err = 1;
           }
}

If (err == 0) {
  document.write("<caption> 2012 NFL Divisional",
                 " Winners </caption>");
  document.write("<tr>",
                 "<th />",
                 "<th> American Conference </th>",
                 "<th> National Conference </th>",
                 "</tr>",
                 "<tr>",
                 "<th> East </th>",
                 "<td> New England Patriots </td>",
                 "<td> Washington Redskins </td>",
                 "</tr>",
                 "<tr>",
                 "<th> North </th>",
                 "<td> Baltimore Ravens </td>",
                 "<td> Green Bay Packers </td>",
```

```
                    "</tr>",
                    "<tr>",
                    "<th> West </th>",
                    "<td> Denver Broncos </td>",
                    "<td> San Francisco 49ers </td>",
                    "</tr>",
                    "<tr>",
                    "<th> South </th>",
                    "<td> Houston Texans </td>",
                    "<td> Atlanta Falcons </td>",
                    "</tr>",
                    "</table>");
}
```

The entire table element is produced with calls to `write`. Alternatively, we could have given all the elements of the table, except the `<table>` and `</table>` tags, directly as HTML in the HTML document. Because `<table>` is in the content of the script element, the validator would not see it. Therefore, the `</table>` tag would also need to be hidden.

Browser displays of the prompt dialog box and the output of `borders2.js` are shown in Figures 4.6 and 4.7, respectively.

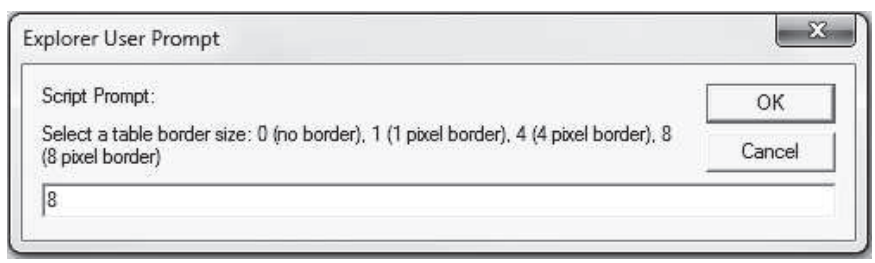


Figure 4.6 Dialog box from `borders2.js`

2010 NFL Divisional Winners

	American Conference	National Conference
East	New England Patriots	Washington Redskins
North	Baltimore Ravens	Green Bay Packers
West	Denver Broncos	San Francisco 49ers
South	Houston Texans	Atlanta Falcons

Figure 4.7 Display produced by `borders2.js`

4.6.4 Loop Statements

The JavaScript `while` and `for` statements are similar to those of Java and C++. The general form of the `while` statement is as follows:

`while` (*control expression*)
  statement or compound statement

The general form of the `for` statement is as follows:

`for` (*initial expression*; *control expression*; *increment expression*)
  statement or compound statement

Both the initial expression and the increment expression of the `for` statement can be multiple expressions separated by commas. The initial expression of a `for` statement can include variable declarations. Such variables are visible in the entire script unless the `for` statement is in a function definition, in which case the variable is visible in the whole function. The following code illustrates a simple `for` construct:

```
var sum = 0,
    count;
for (count = 0; count <= 10; count++)
    sum += count;
```

The following example illustrates the `Date` object and a simple `for` loop:

```
// date.js
//   Illustrates the use of the Date object by
//   displaying the parts of a current date and
//   using two Date objects to time a calculation

// Get the current date
var today = new Date();

// Fetch the various parts of the date
var dateString = today.toLocaleString();
var day = today.getDay();
var month = today.getMonth();
var year = today.getFullYear();
var timeMilliseconds = today.getTime();
var hour = today.getHours();
var minute = today.getMinutes();
var second = today.getSeconds();
var millisecond = today.getMilliseconds();

// Display the parts
document.write(
  "Date: " + dateString + "<br />",
  "Day: " + day + "<br />",
```

```
    "Month: " + month + "<br />",
    "Year: " + year + "<br />",
    "Time in milliseconds: " + timeMilliseconds + "<br />",
    "Hour: " + hour + "<br />",
    "Minute: " + minute + "<br />",
    "Second: " + second + "<br />",
    "Millisecond: " + millisecond + "<br />");

// Time a loop
var dum1 = 1.00149265, product = 1;
var start = new Date();

for (var count = 0; count < 10000; count++)
  product = product + 1.000002 * dum1 / 1.00001;

var end = new Date();
var diff = end.getTime() - start.getTime();
document.write("<br />The loop took " + diff +
               " milliseconds <br />");
```

A display of `date.js` is shown in Figure 4.8.

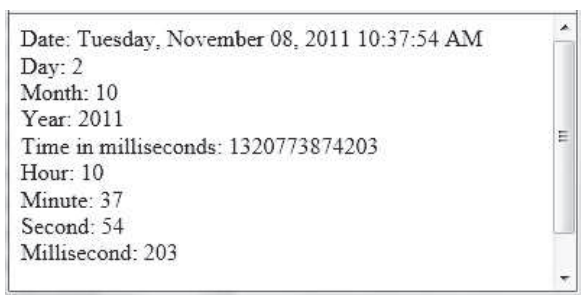


Figure 4.8 Display produced by `date.js`

In addition to the `while` and `for` loop statements, JavaScript has a `do-while` statement, whose form is as follows:

`do` *statement or compound statement*
`while (`*control expression*`)`

The `do-while` statement is related to the `while` statement, but the test for completion is logically (and physically) at the end, rather than at the beginning, of the loop construct. The body of a `do-while` construct is always executed at least once. The following is an example of a `do-while` construct:

```
do {
    count++;
    sum = sum + (sum * count);
} while (count <= 50);
```

JavaScript includes one more loop statement, the `for-in` statement, which is most often used with objects. The `for-in` statement is discussed in Section 4.7.

4.7 Object Creation and Modification

Objects are often created with a `new` expression, which must include a call to a constructor method. The constructor that is called in the `new` expression creates the properties that characterize the new object. In an object-oriented language such as Java, the `new` operator creates a particular object, meaning an object with a type and a specific collection of members. Thus, in Java, the constructor initializes members but does not create them. In JavaScript, however, the `new` operator creates a blank object—that is, one with no properties. Furthermore, JavaScript objects do not have types. The constructor both creates and initializes the properties.

The following statement creates an object that has no properties:

```
var my_object = new Object();
```

In this case, the constructor called is that of `Object`, which endows the new object with no properties, although it does have access to some inherited methods. The variable `my_object` references the new object. Calls to constructors must include parentheses, even if there are no parameters. Constructors are discussed in detail in Section 4.11.

The properties of an object can be accessed with dot notation, in which the first word is the object name and the second is the property name. Properties are not actually variables—they are just the names of values. They are used with object variables to access property values. Because properties are not variables, they are never declared.

The number of members of a class in Java or C++ is fixed at compile time. However, the number of properties in a JavaScript object is dynamic. At any time during interpretation, properties can be added to or deleted from an object. A property for an object is created by assigning a value to its name. Consider the following example:

```
// Create an Object object
var my_car = new Object();
// Create and initialize the make property
my_car.make = "Ford";
// Create and initialize model
my_car.model = "Fusion";
```

This code creates a new object, `my_car`, with two properties: `make` and `model`.

There is an abbreviated way to create an object and its properties. For example, the object referenced with `my_car` in the previous example could be created with the following statement:

```
var my_car = {make: "Ford", model: "Fusion"};
```

Notice that this statement includes neither the `new` operator nor the call to the `Object` constructor.

Because objects can be nested, you can create a new object that is a property of my_car with properties of its own, as in the following statements:

```
my_car.engine = new Object();
my_car.engine.config = "V6";
my_car.engine.hp = 263;
```

Properties can be accessed in two ways. First, any property can be accessed in the same way it is assigned a value, namely, with the object-dot-property notation. Second, the property names of an object can be accessed as if they were elements of an array. To do so, the property name (as a string literal) is used as a subscript. For example, after execution of the statements

```
var prop1 = my_car.make;
var prop2 = my_car["make"];
```

the variables prop1 and prop2 both have the value "Ford".

If an attempt is made to access a property of an object that does not exist, the value undefined is used. A property can be deleted with delete, as in the following example:

```
delete my_car.model;
```

JavaScript has a loop statement, for-in, that is perfect for listing the properties of an object. The form of for-in is

```
for (identifier in object)
  statement or compound statement
```

Consider the following example:

```
for (var prop in my_car)
  document.write("Name: ", prop, "; Value: ",
                 my_car[prop], "<br />");
```

In this example, the variable, prop, takes on the values of the properties of the my_car object, one for each iteration. So, this code lists all the values of the properties of my_car.

4.8 Arrays

In JavaScript, arrays are objects that have some special functionality. Array elements can be primitive values or references to other objects, including other arrays.

4.8.1 Array Object Creation

Array objects, unlike most other JavaScript objects, can be created in two distinct ways. The usual way to create any object is to apply the new operator to a call to a constructor. In the case of arrays, the constructor is named Array:

```
var my_list = new Array(1, 2, "three", "four");
var your_list = new Array(100);
```

In the first declaration, an `Array` object of length 4 is created and initialized. Notice that the elements of an array need not have the same type. In the second declaration, a new `Array` object of length 100 is created, without actually creating any elements. Whenever a call to the `Array` constructor has a single parameter, that parameter is taken to be the number of elements, not the initial value of a one-element array.

The second way to create an `Array` object is with a literal array value, which is a list of values enclosed in brackets:

```
var my_list_2 = [1, 2, "three", "four"];
```

The array `my_list_2` has the same values as the `Array` object `my_list` previously created with `new`.

4.8.2 Characteristics of Array Objects

The lowest index of every JavaScript array is zero. Access to the elements of an array is specified with numeric subscript expressions placed in brackets. The length of an array is the highest subscript to which a value has been assigned, plus 1. For example, if `my_list` is an array with four elements and the following statement is executed, the new length of `my_list` will be 48.

```
my_list[47] = 2222;
```

The length of an array is both read and write accessible through the `length` property, which is created for every array object by the `Array` constructor. Consequently, the length of an array can be set to whatever you like by assigning the `length` property, as in the following example:

```
my_list.length = 1002;
```

Now, the length of `my_list` is 1002, regardless of what it was previously. Assigning a value to the `length` property can lengthen, shorten, or not affect the array's length (if the value assigned happens to be the same as the previous length of the array).

Only the assigned elements of an array actually occupy space. For example, if it is convenient to use the subscript range of 100 to 150 (rather than 0 to 50), an array of length 151 can be created. But if only the elements indexed 100 to 150 are assigned values, the array will require the space of 51 elements, not 151. The `length` property of an array is not necessarily the number of elements allocated. For example, the following statement sets the `length` property of `new_list` to `1002`, but `new_list` may have no elements that have values or occupy space:

```
new_list.length = 1002;
```

To support dynamic arrays of JavaScript, all array elements are allocated dynamically from the heap. Assigning a value to an array element that did not previously exist creates that element.

The next example, `insert_names.js`, illustrates JavaScript arrays. This script has an array of names, which are in alphabetical order. It uses `prompt` to get new names, one at a time, and inserts them into the existing array while maintaining its alphabetical order. Our approach for the insertion is to move elements down one at a time, starting at the end of the array, until the correct position for the new name is found. Then the new name is inserted, and the new array is displayed. Each new name causes the array to grow by one element. This is achieved by assigning a value to the element following what was the last allocated element. Here is the code:

```
// insert_names.js
//   This script has an array of names, name_list,
//   whose values are in alphabetical order. New
//   names are input through a prompt. Each new
//   name is inserted into the name_list array,
//   after which the new list is displayed.

// The original list of names
var name_list = new Array("Al", "Betty", "Kasper",
                          "Michael", "Roberto", "Zimbo");
var new_name, index, last;

// Loop to get a new name and insert it
while (new_name =
       prompt("Please type a new name", "")) {
  last = name_list.length - 1;

// Loop to find the place for the new name
  while (last >= 0 && name_list[last] > new_name) {
    name_list[last + 1] = name_list[last];
    last--;
  }

// Insert the new name into its spot in the array
  name_list[last + 1] = new_name;

// Display the new array
  document.write("<p><strong>The new name list is:</strong> ",
                 "<br />");
  for (index = 0; index < name_list.length; index++)
    document.write(name_list[index], "<br />");
  document.write("</p>");
} //** end of the outer while loop
```

4.8.3 `Array` Methods

`Array` objects have a collection of useful methods, most of which are described in this section. The `join` method converts all the elements of an array to strings and catenates them into a single string. If no parameter is provided to `join`, the values in the new string are separated by commas. If a string parameter is provided, it is used as the element separator. Consider the following example:

```
var names = new Array["Mary", "Murray", "Murphy", "Max"];
...
var name_string = names.join(" : ");
```

The value of `name_string` is now `"Mary : Murray : Murphy : Max"`.

The `reverse` method does what you would expect: It reverses the order of the elements of the `Array` object through which it is called.

The `sort` method coerces the elements of the array to become strings if they are not already strings and sorts them alphabetically. For example, consider the following statement:

```
names.sort();
```

The value of `names` is now `["Mary", "Max", "Murphy", "Murray"]`. Section 4.9.4 discusses the use of `sort` for different orders and nonstring elements.

The `concat` method catenates its actual parameters to the end of the `Array` object on which it is called. Consider the following statements:

```
var names = new Array["Mary", "Murray", "Murphy", "Max"];
...
var new_names = names.concat("Moo", "Meow");
```

The `new_names` array now has length 6, with the elements of `names`, along with `"Moo"` and `"Meow"` as its fifth and sixth elements.

The `slice` method does for arrays what the `substring` method does for strings, returning the part of the `Array` object specified by its parameters, which are used as subscripts. The array returned has the elements of the `Array` object through which it is called, from the first parameter up to, but not including, the second parameter. For example, consider the following statements:

```
var list = [2, 4, 6, 8, 10];
...
var list2 = list.slice(1, 3);
```

The value of `list2` is now `[4, 6]`. If `slice` is given just one parameter, the array that is returned has all the elements of the object, starting with the specified index. In the following statements

```
var list = ["Bill", "Will", "Jill", "dill"];
...
var listette = list.slice(2);
```

the value of `listette` is set to `["Jill", "dill"]`.

When the `toString` method is called through an `Array` object, each of the elements of the object is converted (if necessary) to a string. These strings are catenated, separated by commas. So, for `Array` objects, the `toString` method behaves much like `join`.

The `push`, `pop`, `unshift`, and `shift` methods of `Array` allow the easy implementation of stacks and queues in arrays. The `pop` and `push` methods respectively remove and add an element to the high end of an array, as in the following statements:

```
var list = ["Dasher", "Dancer", "Donner", "Blitzen"];
var deer = list.pop();    // deer is now "Blitzen"
list.push("Blitzen");
  // This puts "Blitzen" back on list
```

The `shift` and `unshift` methods respectively remove and add an element to the beginning of an array. For example, assume that `list` is created as before, and consider the following statements:

```
var deer = list.shift(); // deer is now "Dasher"
list.unshift("Dasher");  // This puts "Dasher" back on list
```

A two-dimensional array is implemented in JavaScript as an array of arrays. This can be done with the `new` operator or with nested array literals, as shown in the script `nested_arrays.js`:

```
// nested_arrays.js
//   An example illustrating an array of arrays

// Create an array object with three arrays as its elements
var nested_array = [[2, 4, 6], [1, 3, 5], [10, 20, 30]];

// Display the elements of nested_list
for (var row = 0; row <= 2; row++) {
  document.write("Row ", row, ":  ");

  for (var col = 0; col <=2; col++)
    document.write(nested_array[row][col], " ");

  document.write("<br />");
}
```

Figure 4.9 shows a browser display of `nested_arrays.js`.

Figure 4.9 Display of `nested_arrays.js`

4.9 Functions

JavaScript functions are similar to those of other C-based languages, such as C and C++.

4.9.1 Fundamentals

A *function definition* consists of the function's header and a compound statement that describes the actions of the function. This compound statement is called the *body* of the function. A function *header* consists of the reserved word `function`, the function's name, and a parenthesized list of parameters if there are any. The parentheses are required even if there are no parameters.

A `return` statement returns control from the function in which it appears to the function's caller. Optionally, it includes an expression, whose value is returned to the caller. A function body may include one or more `return` statements. If there are no `return` statements in a function or if the specific `return` that is executed does not include an expression, the value returned is `undefined`. This is also the case if execution reaches the end of the function body without executing a `return` statement (an action that is valid).

Syntactically, a call to a function with no parameters states the function's name followed by an empty pair of parentheses. A call to a function that returns `undefined` is a standalone statement. A call to a function that returns a useful value appears as an operand in an expression (often, the whole right side of an assignment statement). For example, if `fun1` is a parameterless function that returns `undefined`, and if `fun2`, which also has no parameters, returns a useful value, they can be called with the following code:

```
fun1();
result = fun2();
```

JavaScript functions are objects, so variables that reference them can be treated as are other object references—they can be passed as parameters, be assigned to other variables, and be the elements of an array. The following example is illustrative:

```
function fun() { document.write(
                   "This surely is fun! <br/>");}
ref_fun = fun;   // Now, ref_fun refers to the fun object
fun();           // A call to fun
ref_fun();       // Also a call to fun
```

Because JavaScript functions are objects, their references can be properties in other objects, in which case they act as methods.

To ensure that the interpreter sees the definition of a function before it sees a call to the function—a requirement in JavaScript—function definitions are placed in the head of an HTML document (either explicitly or implicitly). Normally, but not always, calls to functions appear in the document body.

4.9.2 Local Variables

The *scope* of a variable is the range of statements over which it is visible. When JavaScript is embedded in an HTML document, the scope of a variable is the range of lines of the document over which the variable is visible.

A variable that is not declared with a `var` statement is implicitly declared by the JavaScript interpreter at the time it is first encountered in the script. Variables that are implicitly declared have *global scope*—that is, they are visible in the entire HTML document (or entire file if the script is in its own file)—even if the implicit declaration occurs within a function definition. Variables that are explicitly declared outside function definitions also have global scope. As stated earlier, we recommend that all variables be explicitly declared.

It is usually best for variables that are used only within a function to have *local scope*, meaning that they are visible and can be used only within the body of the function. Any variable explicitly declared with `var` in the body of a function has local scope.

If a name that is defined both as a local variable and as a global variable appears in a function, the local variable has precedence, effectively hiding the global variable with the same name. This is the advantage of local variables: When you make up their names, you need not be concerned that a global variable with the same name may exist somewhere in the collection of scripts in the HTML document.

Although JavaScript function definitions can be nested, the need for nested functions in client-side JavaScript is minimal. Furthermore, they can greatly complicate scripts. Therefore, we do not recommend the use of nested functions and do not discuss them.

4.9.3 Parameters

The parameter values that appear in a call to a function are called *actual parameters*. The parameter names that appear in the header of a function definition, which correspond to the actual parameters in calls to the function, are called *formal parameters*. Like C, C++, and Java, JavaScript uses the pass-by-value parameter-passing method. When a function is called, the values of the actual parameters specified in the call are, in effect, copied into their corresponding formal parameters, which behave exactly like local variables. Because references are passed as the actual parameters of objects, the called function has access to the objects and can change them, thereby providing the semantics of pass-by-reference parameters. However, if a reference to an object is passed to a function and the function changes its corresponding formal parameter (rather than the object to which it points), then the change has no effect on the actual parameter. For example, suppose an array is passed as a parameter to a function, as in the following code:

```
function fun1(my_list) {
  var list2 = new Array(1, 3, 5);
  my_list[3] = 14;
  ...
  my_list = list2;
  ...
```

```
}
...
var list = new Array(2, 4, 6, 8)
fun1(list);
```

The first assignment to `my_list` in `fun1` changes the object to which `my_list` refers, which was created in the calling code. However, the second assignment to `my_list` changes it to refer to a different array object. This does not change the actual parameter in the caller.

Because of dynamic typing of JavaScript, there is no type checking of parameters. The called function itself can check the types of parameters with the `typeof` operator. However, recall that `typeof` cannot distinguish between different objects. The number of parameters in a function call is not checked against the number of formal parameters in the called function. In the function, excess actual parameters that are passed are ignored; excess formal parameters are set to `undefined`.

All parameters are communicated through a property array, `arguments`, that, like other array objects, has a property named `length`. By accessing `arguments.length`, a function can determine the number of actual parameters that were passed. Because the `arguments` array is directly accessible, all actual parameters specified in the call are available, including actual parameters that do not correspond to any formal parameters (because there were more actual parameters than formal parameters). The following example illustrates a variable number of function parameters:

```
// params.js
//   The params function and a test driver for it.
//   This example illustrates a variable number of
//   function parameters

// Function params
// Parameters: A variable number of parameters
// Returns: nothing
// Displays its parameters
function params(a, b) {
  document.write("Function params was passed ",
      arguments.length, " parameter(s) <br />");
  document.write("Parameter values are: <br />");

  for (var arg = 0; arg < arguments.length; arg++)
    document.write(arguments[arg], "<br />");

  document.write("<br />");
}

// A test driver for function params
params("Mozart");
params("Mozart", "Beethoven");
params("Mozart", "Beethoven", "Tchaikowsky");
```

Figure 4.10 shows a browser display of `params.js`.

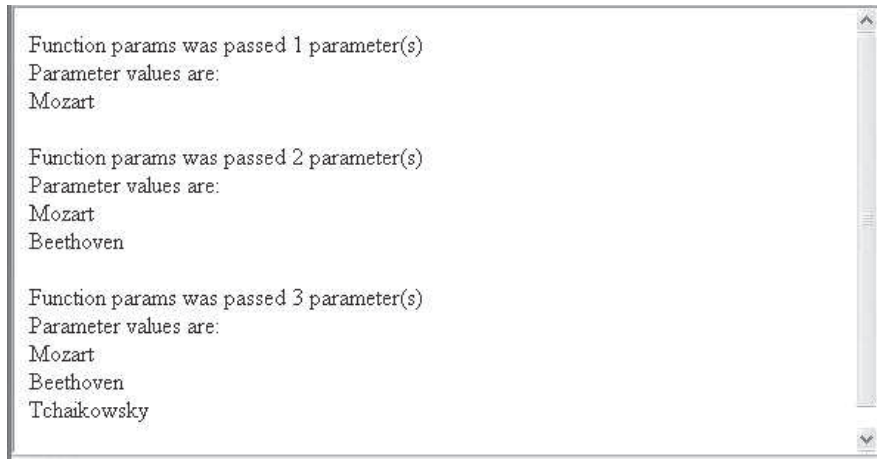


Figure 4.10 Display of `params.js`

There is no elegant way in JavaScript to pass a primitive value by reference. One inelegant way is to put the value in an array and pass the array, as in the following script:

```
// Function by10
//    Parameter: a number, passed as the first element
//               of an array
// Returns: nothing
// Effect: multiplies the parameter by 10
function by10(a) {
    a[0] *= 10;
}
...
var x;
var listx = new Array(1);
...
listx[0] = x;
by10(listx);
x = listx[0];
```

This approach works because arrays are objects.

Another way to have a function change the value of a primitive-type actual parameter is to have the function return the new value as follows:

```
function by10_2(a) {
    return 10 * a;
}
```

```
...
var x;
...
x = by10_2(x);
```

4.9.4 The `sort` Method, Revisited

Recall that the `sort` method for array objects converts the array's elements to strings, if necessary, and then sorts them alphabetically. If you need to sort something other than strings, or if you want an array to be sorted in some order other than alphabetically as strings, the comparison operation must be supplied to the `sort` method by the caller. Such a comparison operation is passed as a parameter to `sort`. The comparison function must return a negative number if the two elements being compared are in the desired order, zero if they are equal, and a number greater than zero if they must be interchanged. For numbers, simply subtracting the second from the first produces the required result. For example, if you want to use the `sort` method to sort the array of numbers `num_list` into descending order, you could do so with the following code:

```
// Function num_order
// Parameter: Two numbers
// Returns: If the first parameter belongs before the
//            second in descending order, a negative number
//          If the two parameters are equal, 0
//          If the two parameters must be
//            interchanged, a positive number
function num_order(a, b) {return b - a;}
// Sort the array of numbers, list, into
// ascending order
  num_list.sort(num_order);
```

Rather than defining a comparison function elsewhere and passing its name, the function definition can appear as the actual parameter in the call to `sort`. Such a function is nameless and can be used only where its definition appears. A nameless function is illustrated in the sample script in Section 4.10.

4.10 An Example

The following example of an HTML document contains a JavaScript function to compute the median of an array of numbers. The function first uses the `sort` method to sort the array. If the length of the given array is odd, the median is the middle element and is determined by dividing the length by `2` and truncating the result with the use of `floor`. If the length is even, the median is the average of

the two middle elements. Note that `round` is used to compute the result of the average computation. Here is the code:

```
// medians.js
//   A function and a function tester
//   Illustrates array operations

// Function median
//   Parameter: An array of numbers
//   Result: The median of the array
//   Return value: none
function median(list) {
  list.sort(function (a, b) {return a - b;});
  var list_len = list.length;

// Use the modulus operator to determine whether
//   the array's length is odd or even
// Use Math.floor to truncate numbers
// Use Math.round to round numbers
  if ((list_len % 2) == 1)
    return list[Math.floor(list_len / 2)];
  else
    return Math.round((list[list_len / 2 - 1] +
                       list[list_len / 2]) / 2);
}  // end of function median

// Test driver
var my_list_1 = [8, 3, 9, 1, 4, 7];
var my_list_2 = [10, -2, 0, 5, 3, 1, 7];
var med = median(my_list_1);
document.write("Median of [", my_list_1, "] is: ",
               med, "<br />");
med = median(my_list_2);
document.write("Median of [", my_list_2, "] is: ",
               med, "<br />");
```

Figure 4.11 shows a browser display of `medians.js`.

Figure 4.11 Display of `medians.js`

One significant side effect of the median function is that it leaves the given array in ascending order, which may not always be acceptable. If it is not, the array could be moved to a local array in median before the sorting operation.

Notice that this script depends on the fact that the array subscripts begin with 0.

4.11 Constructors

JavaScript constructors are special functions that create and initialize the properties of newly created objects. Every new expression must include a call to a constructor whose name is the same as that of the object being created. As you saw in Section 4.8, for example, the constructor for arrays is named Array.

Obviously, a constructor must be able to reference the object on which it is to operate. JavaScript has a predefined reference variable for this purpose, named this. When the constructor is called, this is a reference to the newly created object. The this variable is used to construct and initialize the properties of the object. For example, the following constructor:

```
function car(new_make, new_model, new_year) {
    this.make = new_make;
    this.model = new_model;
    this.year = new_year;
}
```

could be used as in the following statement:

```
my_car = new car("Ford", "Fusion", "2012");
```

So far, we have considered only data properties. If a method is to be included in the object, it is initialized the same way as if it were a data property. For example, suppose you wanted a method for car objects that listed the property values. A function that could serve as such a method could be written as follows:

```
function display_car() {
    document.write("Car make: ", this.make, "<br/>");
    document.write("Car model: ", this.model, "<br/>");
    document.write("Car year: ", this.year, "<br/>");
}
```

The following line must then be added to the car constructor:

```
this.display = display_car;
```

Now the call my_car.display() will produce the following output:

```
Car make: Ford
Car model: Fusion
Car year: 2012
```

The collection of objects created by using the same constructor is related to the concept of class in an object-oriented programming language. All such objects

have the same set of properties and methods, at least initially. These objects can diverge from each other through user code changes. Furthermore, there is no convenient way to determine in the script whether two objects have the same set of properties and methods.

4.12 Pattern Matching by Using Regular Expressions

JavaScript has powerful pattern-matching capabilities based on regular expressions. There are two approaches to pattern matching in JavaScript: one that is based on the methods of the `RegExp` object and one that is based on methods of the `String` object. The regular expressions used by these two approaches are the same and based on the regular expressions of the Perl programming language. This book covers only the `String` methods for pattern matching.

As stated previously, patterns are specified in a form that is based on regular expressions, which originally were developed to define members of a simple class of formal languages. Elaborate and complex patterns can be used to describe specific strings or categories of strings. Patterns, which are sent as parameters to the pattern-matching methods, are delimited with slashes.

The simplest pattern-matching method is `search`, which takes a pattern as a parameter. The `search` method returns the position in the `String` object (through which it is called) at which the pattern matched. If there is no match, `search` returns `–1`. Most characters are normal, which means that, in a pattern, they match themselves. The position of the first character in the string is 0. As an example, consider the following statements:

```
var str = "Rabbits are furry";
var position = str.search(/bits/);
if (position >= 0)
    document.write("'bits' appears in position", position,
                   "<br />");
else
    document.write("'bits' does not appear in str <br />");
```

These statements produce the following output:

```
'bits' appears in position 3
```

4.12.1 Character and Character-Class Patterns

The *normal* characters are those that are not metacharacters. Metacharacters are characters that have special meanings in some contexts in patterns. The following are the pattern metacharacters:

```
\ | ( ) [ ] { } ^ $ * + ? .
```

Metacharacters can themselves be placed in a pattern by being immediately preceded by a backslash.

A period matches any character except newline. So, the following pattern matches `"snowy"`, `"snowe"`, and `"snowd"`, among others:

```
/snow./
```

To match a period in a string, the period must be preceded by a backslash in the pattern. For example, the pattern `/3\.4/` matches `3.4`. The pattern `/3.4/` matches `3.4` and `374`, among others.

It is often convenient to be able to specify classes of characters rather than individual characters. Such classes are defined by placing the desired characters in brackets. Dashes can appear in character class definitions, making it easy to specify sequences of characters. For example, the following character class matches `'a'`, `'b'`, or `'c'`:

```
[abc]
```

The following character class matches any lowercase letter from `'a'` to `'h'`:

```
[a-h]
```

If a circumflex character (^) is the first character in a class, it inverts the specified set. For example, the following character class matches any character except the letters `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`:

```
[^aeiou]
```

Because they are frequently used, some character classes are predefined and named and can be specified by their names. These are shown in Table 4.8, which gives the names of the classes, their literal definitions as character classes, and descriptions of what they match.

Table 4.8 Predefined character classes

Name	Equivalent Pattern	Matches
`\d`	`[0-9]`	A digit
`\D`	`[^0-9]`	Not a digit
`\w`	`[A-Za-z_0-9]`	A word character (alphanumeric)
`\W`	`[^A-Za-z_0-9]`	Not a word character
`\s`	`[ \r\t\n\f]`	A white-space character
`\S`	`[^ \r\t\n\f]`	Not a white-space character

The following examples show patterns that use predefined character classes:

```
/\d\.\d\d/    // Matches a digit, followed by a period,
              // followed by two digits
/\D\d\D/      // Matches a single digit
/\w\w\w/      // Matches three adjacent word characters
```

In many cases, it is convenient to be able to repeat a part of a pattern, often a character or character class. To repeat a pattern, a numeric quantifier, delimited by braces, is attached. For example, the following pattern matches `xyyyyz`:

```
/xy{4}z/
```

There are also three symbolic quantifiers: asterisk (`*`), plus (`+`), and question mark (`?`). An asterisk means zero or more repetitions, a plus sign means one or more repetitions, and a question mark means one or none. For example, the following pattern matches strings that begin with any number of `x`'s (including zero), followed by one or more `y`'s, possibly followed by `z`:

```
/x*y+z?/
```

The quantifiers are often used with the predefined character-class names, as in the following pattern, which matches a string of one or more digits followed by a decimal point and possibly more digits:

```
/\d+\.\d*/
```

As another example, the pattern

```
/[A-Za-z]\w*/
```

matches the identifiers (a letter, followed by zero or more letters, digits, or underscores) in some programming languages.

There is one additional named pattern that is often useful: `\b` (boundary), which matches the boundary position between a word character (`\w`) and a nonword character (`\W`), in either order. For example, the following pattern matches `"A tulip is a flower"` but not `"A frog isn't"`:

```
/\bis\b/
```

The pattern does not match the second string because the "`is`" is followed by another word character (`n`).

The boundary pattern is different from the named character classes in that it does not match a character; instead, it matches a position between two characters.

4.12.2 Anchors

Frequently, it is useful to be able to specify that a pattern must match at a particular position in a string. The most common example of this type of specification is requiring a pattern to match at one specific end of the string. A pattern is tied to a position at one of the ends of a string with an anchor. It can be specified to match only at the beginning of the string by preceding it with a circumflex (`^`) anchor. For example, the following pattern matches `"pearls are pretty"` but does not match `"My pearls are pretty"`:

```
/^pearl/
```

A pattern can be specified to match at the end of a string only by following the pattern with a dollar sign anchor. For example, the following pattern matches `"I like gold"` but does not match `"golden"`:

```
/gold$/
```

Anchor characters are like boundary-named patterns: They do not match specific characters in the string; rather, they match positions before, between, or after characters. When a circumflex appears in a pattern at a position other than the beginning of the pattern or at the beginning of a character class, it has no special meaning. (It matches itself.) Likewise, if a dollar sign appears in a pattern at a position other than the end of the pattern, it has no special meaning.

4.12.3 Pattern Modifiers

Modifiers can be attached to patterns to change how they are used, thereby increasing their flexibility. The modifiers are specified as letters just after the right delimiter of the pattern. The `i` modifier makes the letters in the pattern match either uppercase or lowercase letters in the string. For example, the pattern `/Apple/i` matches '`APPLE`', '`apple`', '`APPle`', and any other combination of uppercase and lowercase spellings of the word "apple."

The `x` modifier allows white space to appear in the pattern. Because comments are considered white space, this provides a way to include explanatory comments in the pattern. For example, the pattern

```
/\d+           # The street number
\s             # The space before the street name
[A-Z][a-z]+    # The street name
/x
```

is equivalent to

```
/\d+\s[A-Z][a-z]+/
```

4.12.4 Other Pattern-Matching Methods of `String`

The `replace` method is used to replace substrings of the `String` object that match the given pattern. The `replace` method takes two parameters: the pattern and the replacement string. The `g` modifier can be attached to the pattern if the replacement is to be global in the string, in which case the replacement is done for every match in the string. The matched substrings of the string are made available through the predefined variables `$1`, `$2`, and so on. For example, consider the following statements:

```
var str = "Fred, Freddie, and Frederica were siblings";
str.replace(/Fre/g, "Boy");
```

In this example, `str` is set to `"Boyd, Boyddie, and Boyderica were siblings"`, and `$1`, `$2`, and `$3` are all set to `"Fre"`.

The `match` method is the most general of the `String` pattern-matching methods. The `match` method takes a single parameter: a pattern. It returns an array of the results of the pattern-matching operation. If the pattern has the `g` modifier, the returned array has all the substrings of the string that matched. If the pattern does not include the `g` modifier, the returned array has the match as its first element, and the remainder of the array has the matches of parenthesized parts of the pattern if there are any:

```
var str =
  "Having 4 apples is better than having 3 oranges";
var matches = str.match(/\d/g);
```

In this example, `matches` is set to `[4, 3]`.

Now consider a pattern that has parenthesized subexpressions:

```
var str = "I have 428 dollars, but I need 500";
var matches = str.match(/(\d+)([^\d]+)(\d+)/);
document.write(matches, "<br />");
```

The following is the value of the `matches` array after this code is interpreted:

```
["428 dollars, but I need 500", "428",
" dollars, but I need ", "500"]
```

In this result array, the first element, `"428 dollars, but I need 500"`, is the match; the second, third, and fourth elements are the parts of the string that matched the parenthesized parts of the pattern, `(\d+)`, `([^\d]+)`, and `(\d+)`.

The `split` method of `String` splits its object string into substrings on the basis of a given string or pattern. The substrings are returned in an array. For example, consider the following code:

```
var str = "grapes:apples:oranges";
var fruit = str.split(":");
```

In this example, `fruit` is set to `[grapes, apples, oranges]`.

4.13 Another Example

One of the common uses for JavaScript is to check the format of input from HTML forms, which is discussed in detail in Chapter 5. The following example presents a simple function that uses pattern matching to check a given string that is supposed to contain a phone number, in order to determine whether the format of the phone number is correct:

```
// forms_check.js
//   A function tst_phone_num is defined and tested.
//   This function checks the validity of phone
//   number input from a form
```

```
// Function tst_phone_num
//   Parameter: A string
//   Result: Returns true if the parameter has the form of a valid
//           seven-digit phone number (3 digits, a dash, 4 digits)

function tst_phone_num(num) {

// Use a simple pattern to check the number of digits and the dash
  var ok = num.search(/^\d{3}-\d{4}$/);

  if (ok == 0)
    return true;
  else
    return false;

}  // end of function tst_phone_num

// A script to test tst_phone_num
var tst = tst_phone_num("444-5432");
if (tst)
  document.write("444-5432 is a valid phone number <br />");
else
  document.write("Error in tst_phone_num <br />");

tst = tst_phone_num("444-r432");
if (tst)
  document.write("Program error <br />");
else
  document.write(
             "444-r432 is not a valid phone number <br />");

tst = tst_phone_num("44-1234");
if (tst)
  document.write("Program error <br />");
else
  document.write("44-1234 is not a valid phone number <br /");
```

Figure 4.12 shows a browser display of `forms_check.js`.

Figure 4.12 Display of `forms_check.js`

4.14 Errors in Scripts

The JavaScript interpreter is capable of detecting various errors in scripts. These are primarily syntax errors, although uses of undefined variables are also detected. Debugging a script is a bit different from debugging a program in a more typical programming language, mostly because errors that are detected by the JavaScript interpreter are found while the browser is attempting to display a document. In some cases, a script error causes the browser not to display the document and does not produce an error message. Without a diagnostic message, you must simply examine the code to find the problem. This is, of course, unacceptable for all but the smallest and simplest scripts. Fortunately, there are ways to get some debugging assistance.

Although the default settings for Internet Explorer 8 (IE8) provide JavaScript syntax error detection and debugging, IE9 and its successors have these features turned off by default. To turn them on, select *Tools/Internet Options* and the *Advanced* tab. Under *Browsing* remove the check on *Disable script debugging (Internet Explorer)* and set the check on *Display a notification about every script error*. Then you will get syntax error detection and the display of error messages, along with the offending line and character position in the line with the error. These messages are shown in a small window. For example, consider the following sample script:

```
// debugdemo.js
//   An example to illustrate debugging help

var row;
row = 0;

while(row != 4 {
  document.write("row is ", row, "<br />");
  row++;
}
```

Notice the syntax error in the `while` statement (a missing right parenthesis). Figure 4.13 shows the browser display of what happens when an attempt is made to run `debugdemo.js`.

The FX3+ browsers have a special console window that displays script errors. Select *Tools/Web Developer/Error Console* to open the window.[12] When you use this browser to display documents that include JavaScript, the window should be opened. After an error message has appeared and has been used to fix a script, press the *Clear* button on the console. Otherwise, the old error message will remain there and possibly cause confusion about subsequent problems. An example of the FX3 JavaScript Console window is shown in Figure 4.14.

12. If the menu bar is not displayed, in which case *Tools* is not visible, click *f10*.

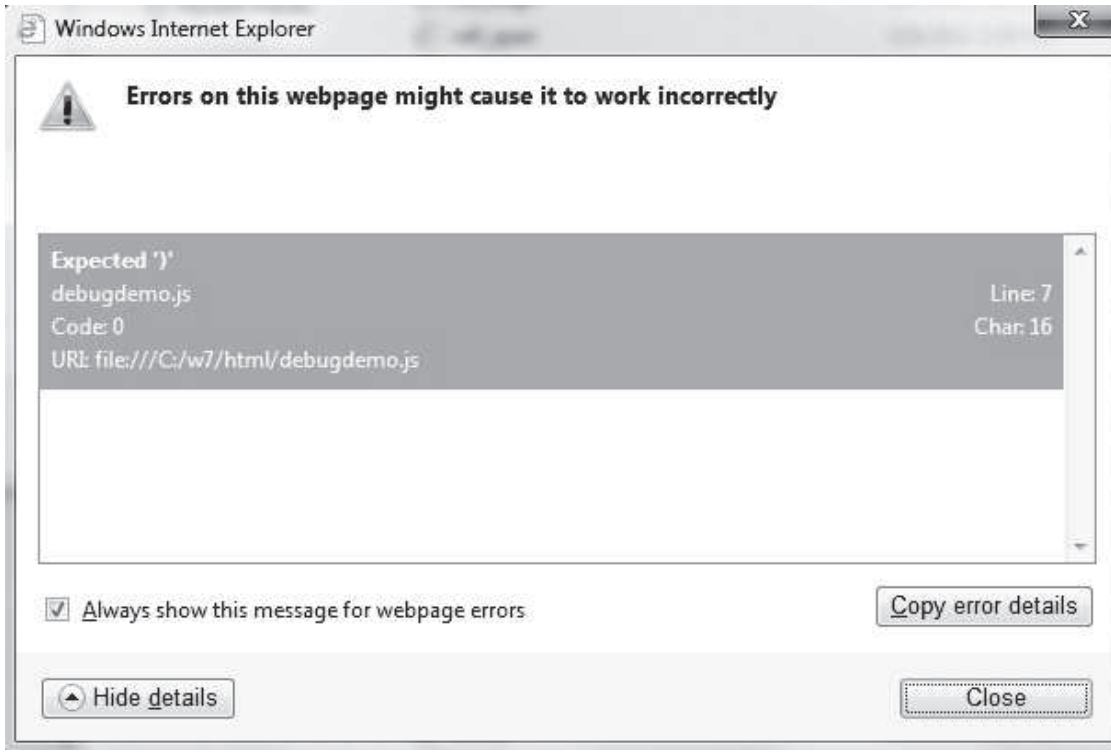


Figure 4.13 Result of running `debugdemo.js` with IE10

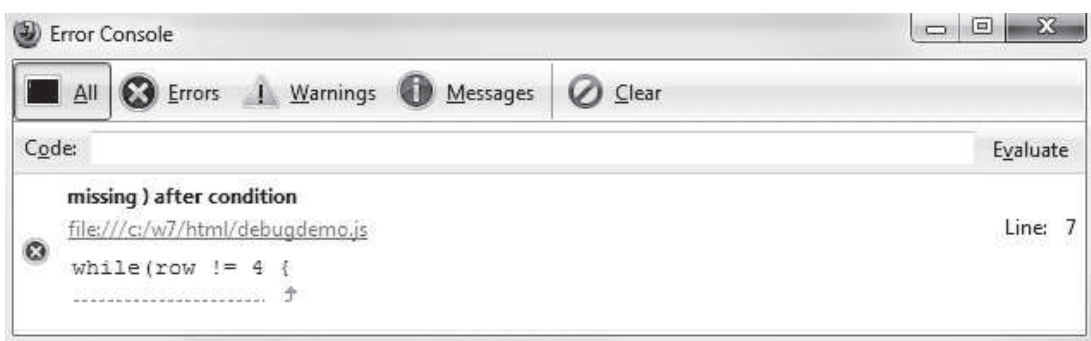


Figure 4.14 Display of the FX3 error console after attempting to run `debugdemo.js`

In the Chrome browsers, the JavaScript error console is accessed by selecting the upper-right icon (three horizontal bars), *Tools*, *JavaScript console*. An example of this console is shown in Figure 4.15.

Figure 4.15 The Chrome error console

The more interesting and challenging programming problems are detectable only during execution or interpretation. For these problems, a debugger is used. Both IE and FX browsers have debuggers for JavaScript.

In IE10, click on *Tools/Developer Tools* to get the built-in JavaScript debugger. The JavaScript debugger for the FX browsers, which was produced by Mozilla and is named Venkman, is available at `http://www.mozilla.org/projects/venkman/`. Another JavaScript debugger, named Firebug, is available for the FX browsers at `https://addons.mozilla.org/en-US/firefox/addon/1843`.

Summary

Client-side JavaScript scripts are embedded in HTML files as the content of `<script>` tags. A file containing a script can be included by specifying its name as the value of the `<script>` attribute `src`. The script itself must appear in a special HTML comment.

Values in JavaScript are either primitives or objects. The primitive types are Number, String, Boolean, Undefined, and Null. Numbers are represented in double-precision floating-point format. The Number, String, and Boolean types have corresponding objects named `Number`, `String`, and `Boolean`, which act as wrapper objects. String literals can use either single or double quotes as delimiters.

JavaScript is dynamically typed, which is not the same as being a typeless language. Variables are typeless, but the values they reference are typed. The type of the value referenced by a variable can change every time a new value is assigned to the variable. It is best to declare all variables explicitly.

The `Number` object includes a collection of useful properties such as `MIN_VALUE` and `PI`. The `Math` object has many methods for commonly used operations on numbers, such as `round` and `cos`. The catenation operator, `+`, creates a new string by putting two operand strings together. The `String` property `length` stores the number of characters in a string. There are `String` methods to return the character at a specified position in the string, the position of a specified character in the string, and a specified substring of the string. There are a large number of other `String` methods as well.

The `typeof` operator returns the type name of its operand if the operand is a primitive type; otherwise, it returns `"object"`.

The Date object provides the current time and date. It includes a large number of methods to produce various parts of the time and date, such as the day of the week and the hour of the day.

The alert method of Window produces output in a dialog box. The confirm method of Window asks the user to select either an *OK* button or a *Cancel* button. The prompt method of Window asks the user for textual input. The document.write method dynamically produces HTML content. The control statements of JavaScript are closely related to those of other common programming languages. Included is a switch statement.

Arrays in JavaScript are objects, as they are in Java. They have dynamic length. An Array object can be created in a new expression, which includes a call to the Array constructor, or simply by assigning an Array literal to a variable. Array literals are lists of values enclosed in brackets. Every Array object has a length property, which is both readable and writable. The length property stores the number of elements in the array. Array objects have a collection of useful methods, among which are join, which joins the elements of an array in a string; reverse, which reverses the order of elements in an array; sort, which converts the elements of the array to strings and sorts them alphabetically; and slice, which returns a specified part of the array. The array methods pop, push, shift, and unshift were designed to implement stacks and queues in arrays.

Function definitions name their formal parameters, but do not include type names. All functions return values, but the type of the value is not specified in the function's definition. Variables declared in a function with var are local to that function. Parameters are passed by value, resulting in pass-by-value semantics for primitives and pass-by-reference semantics for objects. The arguments property stores the values of the parameters that are passed. Neither the types of the parameters nor the number of parameters is checked by the JavaScript interpreter.

The regular expressions used in the pattern-matching facilities of JavaScript are like the regular expressions of Perl. Pattern matches are specified by one of the three methods—search, replace, or match—of the String object. The regular expressions, or patterns, comprise special characters, normal characters, character classes, and operators. Patterns are delimited with slashes. Character classes are delimited with brackets. If a circumflex appears at the left end of a character class, it inverts the meaning of the characters in the class. Several of the most common character classes are predefined. Subpatterns can be followed by numeric or symbolic quantifiers. Patterns can be anchored at the left or right end of the string against which the pattern is being matched. The search method searches its object string for the pattern given as its parameter. The replace method replaces matches in its object string with its second parameter. The match method searches its object string for the given pattern and returns an array of all matches.

JavaScript syntax error messages are produced by both IE9 and FX3. IE9 and FX3 have JavaScript debuggers available, although with FX3 it must be downloaded and installed.

Review Questions

4.1 Describe briefly three major differences between Java and JavaScript.

4.2 Describe briefly three major uses of JavaScript on the client side.

4.3 Describe briefly the basic process of event-driven computation.

4.4 What are the two categories of properties in JavaScript?

4.5 Why does JavaScript have two categories of data variables, namely, primitives and objects?

4.6 Describe the two ways to embed a JavaScript script in an HTML document.

4.7 What are the two forms of JavaScript comments?

4.8 Why are JavaScript scripts sometimes hidden in HTML documents by putting them into HTML comments?

4.9 What are the five primitive data types in JavaScript?

4.10 Do single-quoted string literals have any characteristics different from those of double-quoted string literals?

4.11 In what circumstances would a variable have the value `undefined`?

4.12 If the value `undefined` is used as a Boolean expression, is it interpreted as `true` or `false`?

4.13 What purpose do rules of operator precedence serve in a programming language?

4.14 What purpose do rules of operator associativity serve in a programming language?

4.15 Describe the purpose and characteristics of `NaN`.

4.16 Why is `parseInt` not used more often?

4.17 What value does `typeof` return for an object operand?

4.18 What is the usual end-of-line punctuation for the string operand passed to `document.write`?

4.19 What is the usual end-of-line punctuation for the string operand passed to `alert`?

4.20 Describe the operation of the `prompt` method.

4.21 What is a control construct?

4.22 What are the three possible forms of control expressions in JavaScript?

4.23 What is the difference between `==` and `===`?

4.24 What does short-circuit evaluation of an expression mean?

4.25 What is the semantics of a `break` statement?

4.26 What is the difference between a `while` statement and a `do-while` statement?

4.27 When is a JavaScript constructor called?

4.28 What is the difference between a constructor in Java and one in JavaScript?

4.29 What are the properties of an object created with a `new` operator and the `Object` constructor?

4.30 Describe the two ways the properties of an object can be referenced.

4.31 How is a new property of an object created?

4.32 Describe the semantics of the `for-in` statement.

4.33 Describe the two ways an `Array` object can be created.

4.34 What is the relationship between the value of the `length` property of an `Array` object and the actual number of existing elements in the object?

4.35 Describe the semantics of the `join` method of `Array`.

4.36 Describe the semantics of the `slice` method when it is given just one parameter.

4.37 What is the form of a nested array literal?

4.38 What value is returned by a function that contains no `return` statement?

4.39 Define the scope of a variable in a JavaScript script embedded in an HTML document when the variable is not declared in a function.

4.40 Is it possible to reference global variables in a JavaScript function?

4.41 What is the advantage of using local variables in functions?

4.42 What parameter-passing method does JavaScript use?

4.43 Does JavaScript check the types of actual parameters against the types of their corresponding formal parameters?

4.44 How can a function access actual parameter values for those actual parameters that do not correspond to any formal parameter?

4.45 What is one way in which primitive variables can be passed by reference to a function?

4.46 In JavaScript, what exactly does a constructor do?

4.47 What is a character class in a pattern?

4.48 What are the predefined character classes, and what do they mean?

4.49 What are the symbolic quantifiers, and what do they mean?

4.50 Describe the two end-of-line anchors.

4.51 What does the `i` pattern modifier do?

4.52 What exactly does the `String` method `replace` do?

4.53 What exactly does the `String` method `match` do?

Exercises

Write, test, and debug (if necessary) JavaScript scripts for the problems that follow. When required to write a function, you must include a script to test the function with at least two different data sets. In all cases, for testing, you must write an HTML file that references the JavaScript file.

4.1 *Output*: A table of the numbers from 5 to 15 and their squares and cubes, using `alert`.

4.2 *Output*: The first 20 Fibonacci numbers, which are defined as in the sequence

$$1, 1, 2, 3, \ldots$$

where each number in the sequence after the second is the sum of the two previous numbers. You must use `document.write` to produce the output.

4.3 *Input*: Three numbers, using `prompt` to get each.

Output: The largest of the three input numbers.

Hint: Use the predefined function `Math.max`.

4.4 Modify the script of Exercise 4.2 to use `prompt` to input a number `n` that is the number of the Fibonacci number required as output.

4.5 *Input*: A text string, using `prompt`.

Output: Either `"Valid name"` or `"Invalid name"`, depending on whether the input names fit the required format, which is

```
Last name, first name, middle initial
```

where neither of the names can have more than 15 characters.

4.6 *Input*: A line of text, using `prompt`.

Output: The words of the input text, in alphabetical order.

4.7 Modify the script of Exercise 4.6 to get a second input from the user, which is either `"ascending"` or `"descending"`. Use this input to determine how to sort the input words.

4.8 *Function*: `no_zeros`

Parameter: An array of numbers.

Result: The given array must be modified to remove all zero values.

Returns: `true` if the given array included zero values; `false` otherwise.

4.9 *Function*: `e_names`

Parameter: An array of names, represented as strings.

Returns: The number of names in the given array that end in either `"ie"` or `"y"`.

4.10 *Function*: `first_vowel`

Parameter: A string.

Returns: The position in the string of the leftmost vowel.

4.11 *Function*: `counter`

Parameter: An array of numbers.

Returns: The numbers of negative elements, zeros, and values greater than zero in the given array.

Note: You must use a `switch` statement in the function.

4.12 *Function*: `tst_name`

Parameter: A string.

Returns: `true` if the given string has the form

```
string1, string2 letter
```

where both strings must be all lowercase letters except for the first letter and `letter` must be uppercase; `false` otherwise.

4.13 *Function*: `row_averages`

Parameter: An array of arrays of numbers.

Returns: An array of the averages of each of the rows of the given matrix.

4.14 *Function*: `reverser`

Parameter: A number.

Returns: The number with its digits in reverse order.

This page intentionally left blank

CHAPTER

5

JavaScript and HTML Documents

Client-side JavaScript does not include language constructs that are not in core JavaScript. Rather, it defines the collection of objects, methods, and properties that allow scripts to interact with HTML documents on the browser. This chapter describes some of these features and illustrates their use with examples.

The chapter begins with a description of the execution environment of client-side JavaScript. Then it gives a brief overview of the Document Object Model (DOM), noting that you need not know all the details of this model to be able to use client-side JavaScript. Next, the techniques for accessing HTML document elements in JavaScript are discussed. The fundamental concepts of events and

event handling are then introduced, using the DOM 0 event model. Although the event-driven model of computation is not a new idea in programming, it has become more important with the advent of Web programming. Next, the chapter describes the relationships among event objects, HTML tag attributes, and tags, primarily by means of two tables.

Applications of basic event handling are introduced through a sequence of complete HTML-JavaScript examples. The first of these illustrates handling the `load` event from a body element. The next two examples demonstrate the use of the `click` event created when radio buttons are pressed. This is followed by an example that uses the `blur` event to compare passwords that are input twice. The next example demonstrates the use of the `change` event to validate the format of input to a text box. The last example shows the use of the `blur` event to prevent user changes to the values of text box elements.

Next, the event model of DOM 2 is discussed, using a revision of an earlier example to illustrate the new features of this model. The following section introduces the `canvas` element. The chapter then introduces the use of the `navigator` object to determine which browser is being used. Finally, a few of the methods and properties used to traverse and modify DOM structures are briefly introduced.

Nearly all the JavaScript in the examples in this chapter is in separate files. Therefore, each of the examples consists of one or two JavaScript files and an HTML document.

5.1 The JavaScript Execution Environment

A browser displays an HTML document in a window on the screen of the client. The JavaScript `Window` object represents the window that displays the document.

All JavaScript variables are properties of some object. The properties of the `Window` object are visible to all JavaScript scripts that appear either implicitly or explicitly in the window's HTML document, so they include all the global variables. When a global variable is created in a client-side script, it is created as a new property of the `Window` object, which provides the largest enclosing referencing environment for JavaScript scripts.

There can be more than one `Window` object. In this book, however, we deal only with scripts with a single `Window` object.

The JavaScript `Document` object represents the displayed HTML document. Every `Window` object has a property named `document`, which is a reference to the `Document` object that the window displays. The `Document` object is used more often than any other object in client-side JavaScript. Its `write` method was used extensively in Chapter 4.

Every `Document` object has a `forms` array, each element of which represents a form in the document. Each `forms` array element has an `elements` array as a property, which contains the objects that represent the HTML form elements, such as buttons and menus. The JavaScript objects associated with the elements in a document can be addressed in a script in several ways, discussed in Section 5.3.

`Document` objects also have property arrays for anchors, links, images, and applets. There are many other objects in the object hierarchy below a `Window` object, but in this chapter we are interested primarily in documents, forms, and form elements.

5.2 The Document Object Model

The Document Object Model (DOM) has been under development by the W3C since the mid-1990s. DOM Level 3 (usually referred to as DOM 3) is the current approved version. The original motivation for the standard DOM was to provide a specification that would allow Java programs and JavaScript scripts that deal with HTML documents to be portable among various browsers.

Although the W3C never produced such a specification, DOM 0 is the name often applied to describe the document model used by the early browsers that supported JavaScript. Specifically, DOM 0 is the version of the document model implemented in the Netscape 3.0 and Internet Explorer (IE) 3.0 browsers. The DOM 0 model was partially documented in the HTML 4 specification.

DOM 1, the first W3C DOM specification, issued in October 1998, focused on the HTML and XML (see Chapter 7) document model. DOM 2, issued in November 2000, specified a style-sheet object model and defined how style information attached to a document can be manipulated. It also included document traversals and provided a complete and comprehensive event model. DOM 3, issued in 2004, dealt with content models for XML (DTDs and schemas), document validation, and document views and formatting, as well as key events and event groups. As stated previously, DOM 0 is supported by all JavaScript-enabled browsers. DOM 2 is nearly completely supported by Firefox 3 (FX3+), IE10, and Chrome, but IE9 leaves some parts unimplemented. No part of DOM 3 is covered in this book.

The DOM is an Application Programming Interface (API) that defines an interface between HTML documents and application programs. It is an abstract model because it must apply to a variety of application programming languages. Each language that interfaces with the DOM must define a binding to that interface. The actual DOM specification consists of a collection of interfaces, including one for each document tree node type. These interfaces are similar to Java interfaces and C++ abstract classes. They define the objects, methods, and properties that are associated with their respective node types. With the DOM, users can write code in programming languages to create documents, move around in their structures, and change, add, or delete elements and their content.

Documents in the DOM have a treelike structure, but there can be more than one tree in a document (although that is unusual). Because the DOM is an abstract interface, it does not dictate that documents be implemented as trees or collections of trees. Therefore, in an implementation, the relationships among the elements of a document could be represented in several different ways.

As previously stated, a language that is designed to support the DOM must have a binding to the DOM constructs. This binding amounts to a correspondence between constructs in the language and elements in the DOM. In the

JavaScript binding to the DOM, the elements of a document are objects, with both data and operations. The data are called *properties*, and the operations are, naturally, called *methods*. For example, the following HTML element would be represented as an object with two properties, `type` and `name`, with the values `"text"` and `"address"`, respectively:

```
<input type = "text"  name = "address">
```

In most cases, the property names in JavaScript are the same as their corresponding attribute names in HTML.

IE8+, FX3+, and C10+ provide a way of viewing the DOM structure of a displayed document. After displaying a document with IE10, select *Tools/Developer Tools*. The lower-left area of the resulting display will show an elided version of the DOM structure.[1] By clicking all the eliding icons (square boxes that have plus signs in them), the whole structure will be displayed. Consider the following simple document:

```
<!DOCTYPE html>
<!-- table2.html
     A simple table to demonstrate DOM trees
     -->
<html lang = "en">
  <head>
    <title> A simple table </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <table>
      <tr>
        <th> </th>
        <th> Apple </th>
        <th> Orange </th>
      </tr>
      <tr>
        <th> Breakfast </th>
        <td> 0 </td>
        <td> 1 </td>
      </tr>
    </table>
  </body>
</html>
```

The display of this document and its complete DOM structure are shown in Figure 5.1.

1. Eliding abstracts away parts of the structure. An elided part can be restored to the display.

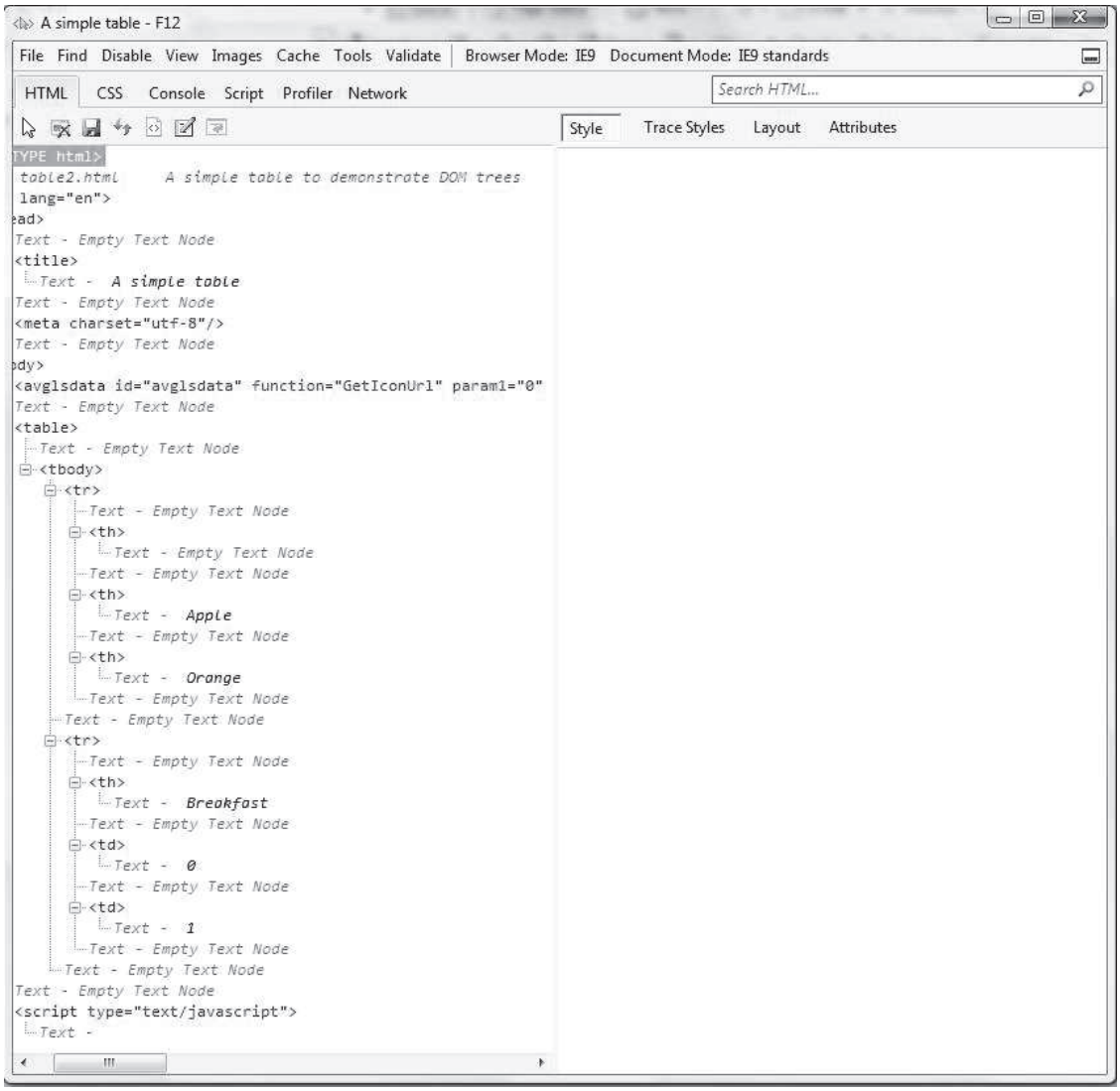


Figure 5.1 The DOM structure of `table2.html` with IE10

The IE10 Developer Tools are helpful for developing and analyzing HTML documents. However, our interest here is in the DOM structure it produces.

To be able to display the DOM structure of a document with FX3+, an add-on must be downloaded. The source of the download is `https://addons.mozilla.org/en-US/firefox/addon/6622`. After the DOM Inspector add-on has been downloaded and installed, the DOM of a document can be displayed by selecting *Tools/DOM Inspector*. If this selection is made while a document is being displayed, FX3+ opens a new window that is similar to the IE10 window for DOM viewing. As with IE10, the elements are initially elided. The upper-right

area is for displaying information about the DOM structure. As was the case with the IE9 Developer Tools, the DOM Inspector of FX3+ has far more uses than simply viewing the DOM structure of a document but they are not discussed here. The FX3+ DOM Inspector display of `table2.html` is shown in Figure 5.2.

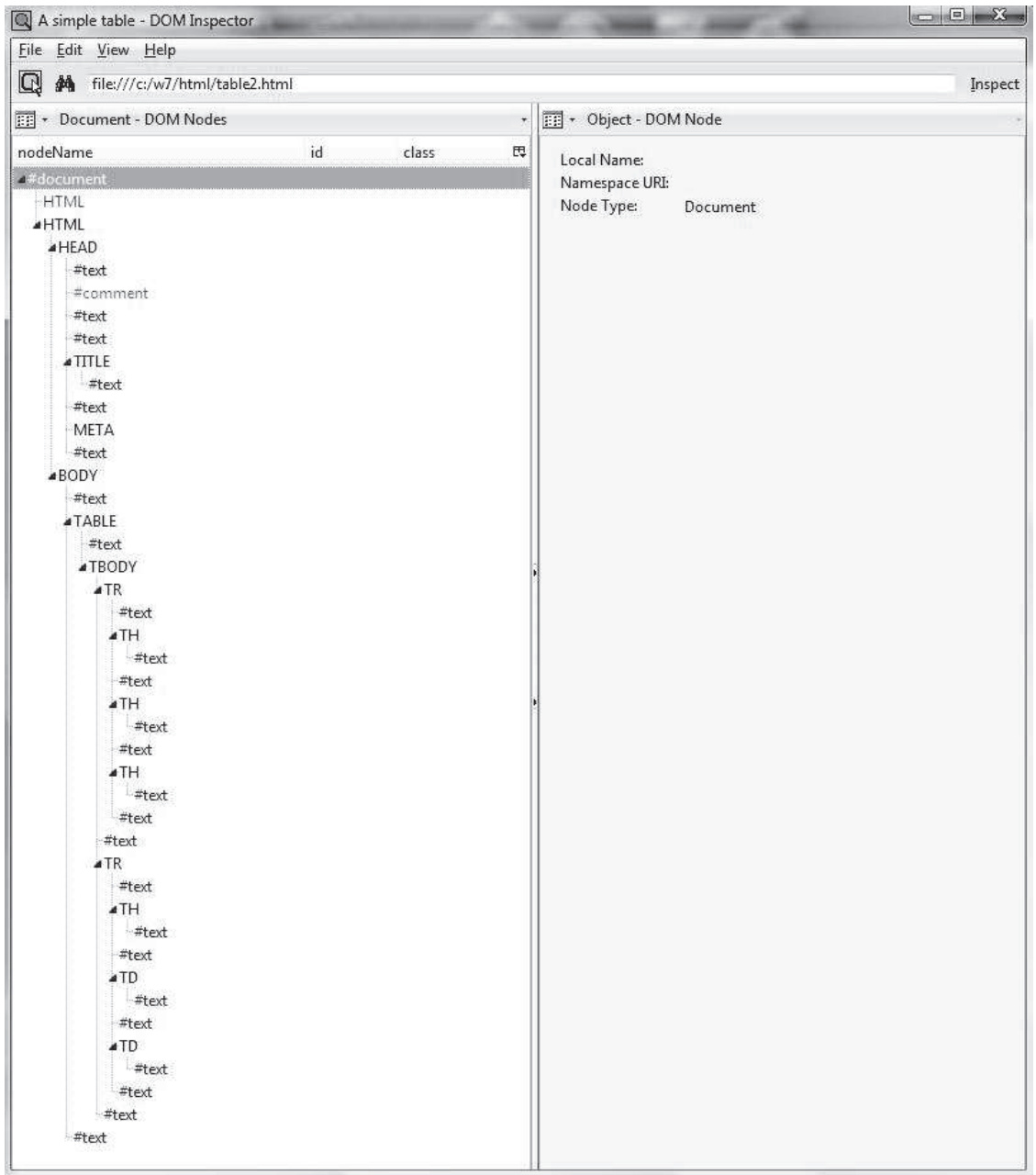


Figure 5.2 The DOM Inspector display of `table2.html` with FX3

With Chrome, to get the DOM one selects the wrench icon, *Tools*, and *Developer Tools*.

Anything resembling a complete explanation of the DOM is far beyond the scope of this book. Our introduction to the DOM here is intended only to provide the basis for our discussion of how JavaScript can be used to respond to document-related events and to modify element attributes, styles, and content dynamically.[2] A detailed description of the DOM can be found at the W3C Web site.

5.3 Element Access in JavaScript

The elements of an HTML document have corresponding objects that are visible to an embedded JavaScript script. The addresses of these objects are required, both by the event handling discussed in this chapter and by the code for making dynamic changes to documents, which is discussed in Chapter 6.

There are several ways the object associated with a form element can be addressed in JavaScript. The original (DOM 0) way is to use the `forms` and `elements` arrays of the `Document` object, which is referenced through the `document` property of the `Window` object. As an example, consider the following document:

```
<html lang = "en">
  <head>
    <title> Access to form elements </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <form action = "">
      <input type = "button"  name = "turnItOn" />
    </form>
  </body>
</html>
```

We refer to the address of the JavaScript object that is associated with an HTML element as the *DOM address* of the element. The DOM address of the button in this example, using the `forms` and `elements` arrays, is as follows:

```
var dom = document.forms[0].elements[0];
```

The problem with this approach to element addressing is that the DOM address is defined by the position of elements in the document, which could change. For example, if a new button were added before the `turnItOn` button in the form, the DOM address shown would be wrong.

Another approach to DOM addressing is to use element names. For this, the element and its enclosing elements, up to but not including the body

2. We will discuss modifications of style properties in Chapter 6.

element, must include name attributes. For example, consider the following document:

```
<html lang = "en">
  <head>
    <title> Access to form elements </title>
    <meta charset = "utf-8" />
  </head>
  <body>
    <form name = "myForm"  action = "">
      <input type = "button"  name = "turnItOn" />
    </form>
  </body>
</html>
```

Using the name attributes, the button's DOM address is as follows:

```
var dom = document.myForm.turnItOn;
```

One minor drawback of this approach is that the XHTML 1.1 standard does not allow the name attribute in the form element, even though the attribute is now valid for form elements. This is a validation problem, but it is not a problem for browsers. Furthermore, the name attribute is valid in form tags in HTML5.

Yet another approach to element addressing is to use the JavaScript method getElementById, which is defined in DOM 1. Because an element's identifier (id) is unique in the document, this approach works, regardless of how deeply the element is nested in other elements in the document. For example, if the id attribute of our button is set to "turnItOn", the following could be used to get the DOM address of that button element:

```
var dom = document.getElementById("turnItOn");
```

The parameter of getElementById can be any expression that evaluates to a string. In many cases, it is a variable.

Because ids are useful for DOM addressing and names are required for form-processing code, form elements often have both ids and names, set to the same value.

Buttons in a group of checkboxes often share the same name. The buttons in a radio button group *always* have the same name. In these cases, the names of the individual buttons obviously cannot be used in their DOM addresses. Of course, each radio button and checkbox can have an id, which would make them easy to address by using getElementById. However, this approach does not provide a convenient way to search a group of radio buttons or checkboxes to determine which is checked.

An alternative to both names and ids is provided by the implicit arrays associated with each checkbox and radio button group. Every such group has an array, which has the same name as the group name, that stores the DOM addresses of

the individual buttons in the group. These arrays are properties of the form in which the buttons appear. To access the arrays, the DOM address of the form object must first be obtained, as in the following example:

```
<form id = "vehicleGroup">
  <input type = "checkbox"  name = "vehicles"
         value = "car" />  Car
  <input type = "checkbox"  name = "vehicles"
         value = "truck" />  Truck
  <input type = "checkbox"  name = "vehicles"
         value = "bike" />  Bike
</form>
```

The implicit array, `vehicles`, has three elements, which reference the three objects associated with the three checkbox elements in the group. This array provides a convenient way to search the list of checkboxes in a group. The `checked` property of a checkbox object is set to `true` if the button is checked. For the preceding sample checkbox group, the following code counts the number of checkboxes that were checked:

```
var numChecked = 0;
var dom = document.getElementById("vehicleGroup");
for (index = 0; index < dom.vehicles.length; index++)
  if (dom.vehicles[index].checked)
    numChecked++;
```

Radio buttons can be addressed and handled exactly as are the checkboxes in the foregoing code.

5.4 Events and Event Handling

The HTML 4.0 standard provided the first specification of an event model for markup documents. This model is sometimes referred to as the DOM 0 event model. Although the DOM 0 event model is limited in scope, it is the only event model supported by all browsers that support JavaScript. A complete and comprehensive event model was specified by DOM 2. The DOM 2 model is supported by the FX3+, IE9+, and C6+ browsers. Our discussion of events and event handling is divided into two parts, one for the DOM 0 model and one for the DOM 2 model.

5.4.1 Basic Concepts of Event Handling

One important use of JavaScript for Web programming is to detect certain activities of the browser and the browser user and provide computation when those activities occur. These computations are specified with a special form of programming called *event-driven programming*. In conventional (nonevent-driven)

programming, the code itself specifies the order in which it is executed, although the order is usually affected by the program's input data. In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the program that is executing.

An *event* is a notification that something specific has occurred, either in the browser, such as the completion of the loading of a document, or a browser user action, such as a mouse click on a form button. Strictly speaking, an event is an object that is implicitly created by the browser and the JavaScript system in response to something having happened.

An *event handler* is a script that is implicitly executed in response to the appearance of an event. Event handlers enable a Web document to be responsive to browser and user activities. One of the most common uses of event handlers is to check for simple errors and omissions in user input to the elements of a form, either when they are changed or when the form is submitted. This kind of checking saves the time of sending incorrect form data to the server.

If you are familiar with the exceptions and exception-handling capabilities of a programming language such as C++ or Java, you should see the close relationship between events and exceptions. Events and exceptions occur at unpredictable times, and both often require some special program actions.

Because events are JavaScript objects, their names are case sensitive. The names of all event objects have only lowercase letters. For example, `click` is an event, but `Click` is not.

Events are created by activities associated with specific HTML elements. For example, the `click` event can be caused by the browser user clicking a radio button or the link of an anchor tag, among other things. Thus, an event's name is only part of the information pertinent to handling the event. In most cases, the specific HTML element that caused the event is also needed.

The process of connecting an event handler to an event is called *registration*. There are two distinct approaches to event handler registration, one that assigns tag attributes and one that assigns handler addresses to object properties. These are further discussed in Sections 5.5 and 5.6.

The `write` method of `document` should never be used in an event handler. Remember that a document is displayed as its markup is parsed by the browser. Events usually occur after the whole document is displayed. If `write` appears in an event handler, the content produced by it might be placed over the top of the currently displayed document.

The remainder of this section and Sections 5.5 through 5.7 describe the DOM 0 event model and some of its uses.

5.4.2 Events, Attributes, and Tags

HTML4 defined a collection of events that browsers implement and with which JavaScript can deal. These events are associated with HTML tag attributes, which can be used to connect the events to handlers. The attributes have names that are closely related to their associated events. Table 5.1 lists the most commonly used events and their associated tag attributes.

Table 5.1 Events and their tag attributes

Events	Tag Attribute
blur	onblur
change	onchange
click	onclick
dblclick	ondblclick
focus	onfocus
keydown	onkeydown
keypress	onkeypress
keyup	onkeyup
load	onload
mousedown	onmousedown
mousemove	onmousemove
mouseout	onmouseout
mouseover	onmouseover
mouseup	onmouseup
reset	onreset
select	onselect
submit	onsubmit
unload	onunload

In many cases, the same attribute can appear in several different tags. The circumstances under which an event is created are related to a tag and an attribute, and they can be different for the same attribute when it appears in different tags.

An HTML element is said to *get focus* when the user puts the mouse cursor over it and clicks the left mouse button. An element can also get focus when the user tabs to the element. When a text element has focus, any keyboard input goes into that element. Obviously, only one text element can have focus at one time. An element becomes blurred when the user moves the cursor away from the element and clicks the left mouse button or when the user tabs away from the element. An element obviously becomes blurred when another element gets focus. Several nontext elements can also have focus, but the condition is less useful in those cases.

Table 5.2 shows (1) the most commonly used attributes related to events, (2) tags that can include the attributes, and (3) the circumstances under which the associated events are created. Only a few of the situations shown in the table are discussed in this chapter.

Table 5.2 Event attributes and their tags

Attributes	Tag	Description
onblur	<a>	The link loses focus.
	<button>	The button loses focus.
	<input>	The input element loses focus.
	<textarea>	The text area loses focus.
	<select>	The selection element loses focus.
onchange	<input>	The input element is changed and loses focus.
	<textarea>	The text area is changed and loses focus.
	<select>	The selection element is changed and loses focus.
onclick	<a>	The user clicks the link.
	<input>	The input element is clicked.
ondblclick	Most elements	The user double-clicks the left mouse button.
onfocus	<a>	The link acquires focus.
	<input>	The input element acquires focus.
	<textarea>	A text area acquires focus.
	<select>	A selection element acquires focus.
onkeydown	<body>, form elements	A key is pressed.
onkeypress	<body>, form elements	A key is pressed and released.
onkeyup	<body>, form elements	A key is released.
onload	<body>	The document is finished loading.
onmousedown	Most elements	The user clicks the left mouse button.
onmousemove	Most elements	The user moves the mouse cursor within the element.
onmouseout	Most elements	The mouse cursor is moved away from being over the element.

Table 5.2 Event attributes and their tags (*continued*)

Attributes	Tag	Description
onmouseover	Most elements	The mouse cursor is moved over the element.
onmouseup	Most elements	The left mouse button is unclicked.
onreset	<form>	The reset button is clicked.
onselect	<input>	Any text in the content of the element is selected.
	<textarea>	Any text in the content of the element is selected.
onsubmit	<form>	The *Submit* button is pressed.
onunload	<body>	The user exits the document.

As mentioned previously, there are two ways to register an event handler in the DOM 0 event model. One of these is by assigning the event handler script to an event tag attribute, as in the following example:

```
<input type = "button" id = "myButton"
       onclick = "alert('You clicked my button!');" />
```

In many cases, the handler consists of more than a single statement. In these cases, often a function is used and the literal string value of the attribute is the call to the function. Consider the following example of a button element:

```
<input type = "button" id = "myButton"
       onclick = "myButtonHandler();" />
```

An event handler function could also be registered by assigning its name to the associated event property on the button object, as in the following example:

```
document.getElementById("myButton").onclick =
                                    myButtonHandler;
```

This statement must follow both the handler function and the form element so that JavaScript has seen both before assigning the property. Notice that only the name of the handler function is assigned to the property—it is neither a string nor a call to the function.

5.5 Handling Events from Body Elements

The events most often created by body elements are `load` and `unload`. As our first example of event handling, we consider the simple case of producing an alert

message when the body of the document has been loaded. In this case, we use the `onload` attribute of `<body>` to specify the event handler:

```
<!DOCTYPE html>
<!-- load.html
     A document for load.js
     -->
<html lang = "en">
  <head>
    <title> load.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "load.js" >
    </script>
  </head>
  <body onload="load_greeting();">
    <p />
  </body>
</html>
```

```
// load.js
//   An example to illustrate the load event
// The onload event handler
function load_greeting () {
  alert("You are visiting the home page of \n" +
        "Pete's Pickled Peppers \n" + "WELCOME!!!");
}
```

Figure 5.3 shows a browser display of `load.html`.

Figure 5.3 Display of `load.html`

The `unload` event is probably more useful than the `load` event. It is used to do some cleanup before a document is unloaded, as when the browser user goes on to some new document. For example, if the document opened a second browser window, that window could be closed by an `unload` event handler.

5.6 Handling Events from Button Elements

Buttons in a Web document provide an effective way to collect simple input from the browser user. The most commonly used event created by button actions is `click`. Section 5.4.2 includes an example of a plain button.

The next example presents a set of radio buttons that enables the user to select information about a specific airplane. The `click` event is used in this example to trigger a call to `alert`, which presents a brief description of the selected airplane. The calls to the event handlers send the value of the pressed radio button to the handler. This is another way the handler can determine which of a group of radio buttons is pressed. Here is the document and the JavaScript file:

```
<!DOCTYPE html>
<!-- radio_click.hmtl
     A document for radio_click.js
     Creates four radio buttons that call the planeChoice
     event handler to display descriptions
     -->
<html lang = "en">
  <head>
    <title> radio_click.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "radio_click.js" >
    </script>
  </head>
  <body>
    <h4> Cessna single-engine airplane descriptions </h4>
    <form id = "myForm"  action = "">
        <p>
        <label> <input type = "radio"  name = "planeButton"
                       value = "152"
                       onclick = "planeChoice(152)" />
        Model 152 </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "172"
                       onclick = "planeChoice(172)" />
        Model 172 (Skyhawk) </label>
```

```
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "182"
                       onclick = "planeChoice(182)" />
        Model 182 (Skylane) </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "210"
                       onclick = "planeChoice(210)" />
        Model 210 (Centurian) </label>
      </p>
    </form>
  </body>
</html>
```

```
// radio_click.js
//   An example of the use of the click event with radio buttons,
//   registering the event handler by assignment to the button
//   attributes


// The event handler for a radio button collection
function planeChoice (plane) {


// Produce an alert message about the chosen airplane
  switch (plane) {
    case 152:
      alert("A small two-place airplane for flight training");
      break;
    case 172:
      alert("The smaller of two four-place airplanes");
      break;
    case 182:
      alert("The larger of two four-place airplanes");
      break;
    case 210:
      alert("A six-place high-performance airplane");
      break;
    default:
      alert("Error in JavaScript function planeChoice");
      break;
  }
}
```

Figure 5.4 shows a browser display of `radio_click.html`. Figure 5.5 shows the `alert` window that results from choosing the *Model 182* radio button in `radio_click.html`.

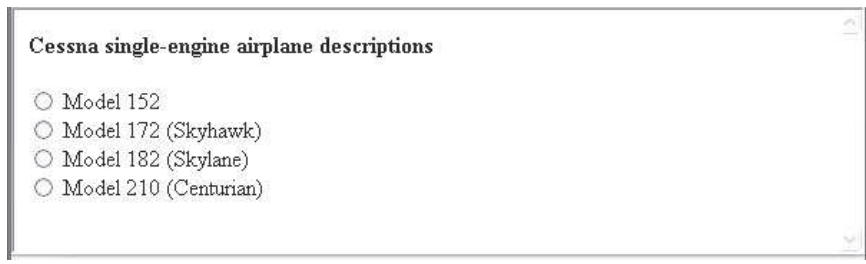


Figure 5.4 Display of `radio_click.html`

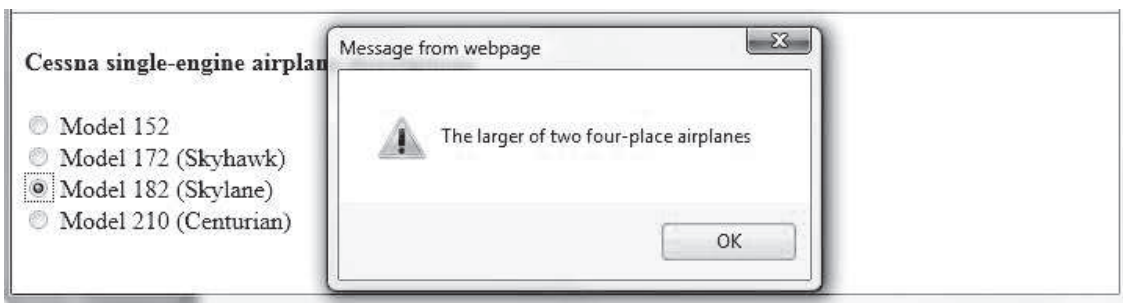


Figure 5.5 The result of pressing the *Model 182* button in `radio_click.html`

In `radio_click.html`, the event handler is registered by assigning its call to the `onclick` attribute of the radio buttons. The specific button that was clicked is identified by the parameter sent in the handler call in the button element. An alternative to using the parameter would be to include code in the handler to determine which radio button was pressed.

The next example, `radio_click2.html`, whose purpose is the same as that of `radio_click.html`, registers the event handler by assigning the name of the handler to the event properties of the radio button objects. For example, the following line of code registers the handler on the first radio button:

```
document.getElementById("myForm").elements[0].onclick =
                planeChoice;
```

Recall that this statement must follow both the handler function and the HTML form specification so that JavaScript sees both before assigning the property. The following example uses three files—one for the HTML, one for the script for the event handlers, and one for the script to register the handlers:

```
<!DOCTYPE html>
<!-- radio_click2.hmtl
     A document for radio_click2.js
     -->
<html lang = "en">
  <head>
    <title> radio_click2.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "radio_click2.js" >
    </script>


  </head>
  <body>
    <h4> Cessna single-engine airplane descriptions </h4>
    <form id = "myForm"  action = "">
      <p>
        <label> <input type = "radio"  name = "planeButton"
                       value = "152" />
        Model 152 </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "172" />
        Model 172 (Skyhawk) </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "182" />
        Model 182 (Skylane) </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "210" />
        Model 210 (Centurian) </label>
      </p>
    </form>


<!-- Script for registering the event handlers -->
    <script type = "text/javascript"  src = "radio_click2r.js" >
    </script>
  </body>
</html>
```

```
// radio_click2.js
//   An example of the use of the click event with radio buttons,
//   registering the event handler by assigning an event property

// The event handler for a radio button collection
function planeChoice (plane) {

// Put the DOM address of the elements array in a local variable
  var dom = document.getElementById("myForm");

// Determine which button was pressed
  for (var index = 0; index < dom.planeButton.length;
       index++) {
    if (dom.planeButton[index].checked) {
      plane = dom.planeButton[index].value;
      break;
  }
}

// Produce an alert message about the chosen airplane
  switch (plane) {
    case "152":
      alert("A small two-place airplane for flight training");
      break;
    case "172":
      alert("The smaller of two four-place airplanes");
      break;
    case "182":
      alert("The larger of two four-place airplanes");
      break;
    case "210":
      alert("A six-place high-performance airplane");
      break;
    default:
      alert("Error in JavaScript function planeChoice");
      break;
  }
}
```

```
// radio_click2r.js
//   The event registering code for radio_click2
var dom = document.getElementById("myForm");
dom.elements[0].onclick = planeChoice;
dom.elements[1].onclick = planeChoice;
dom.elements[2].onclick = planeChoice;
dom.elements[3].onclick = planeChoice;
```

In `radio_click2r.js` (the JavaScript file that registers the event handlers), the form elements (radio buttons in this case) are addressed as elements of the `elements` array. An alternative would be to give each radio button an `id` attribute and use the id to register the handler. For example, the first radio button could be defined as follows:

```
<input type = "radio"  name = "planeButton" value = "152"
   id = "152" />
```

Then the event handler registration would be as follows:

```
document.getElementById("152").onclick = planeChoice;
document.getElementById("172").onclick = planeChoice;
document.getElementById("182").onclick = planeChoice;
document.getElementById("210").onclick = planeChoice;
```

There is no way to specify parameters on the handler function when it is registered by assigning its name to the event property. Therefore, event handlers that are registered this way cannot use parameters—clearly a disadvantage of this approach. In `radio_click2.js`, the handler includes a loop to determine which radio button created the `click` event.

There are two advantages to registering handlers as properties over registering them in HTML attributes. First, it is good to keep HTML and JavaScript separated in the document. This allows a kind of modularization of HTML documents, resulting in a cleaner design that will be easier to maintain. Second, having the handler function registered as the value of a property allows for the possibility of changing the function during use. This could be done by registering a different handler for the event when some other event occurred—an approach that would be impossible if the handler were registered with HTML.

5.7 Handling Events from Text Box and Password Elements

Text boxes and passwords can create four different events: `blur`, `focus`, `change`, and `select`.

5.7.1 The Focus Event

Suppose JavaScript is used to compute the total cost of an order and display it to the customer before the order is submitted to the server for processing. An unscrupulous user may be tempted to change the total cost before submission, thinking that somehow an altered (and lower) price would not be noticed at the server end. Such a change to a text box can be prevented by an event handler that blurs the text box every time the user attempts to put it in focus. Blur can be forced on an element with the `blur` method. The following example illustrates this method:

```
<!DOCTYPE html>
<!-- nochange.html
     A document for nochange.js
     -->
<html lang = "en">
  <head>
    <title> nochange.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "nochange.js" >
    </script>
    <style type = "text/css">
      td, th, table {border: thin solid black}
    </style>

  </head>
  <body>
    <form action = "">
      <h3> Coffee Order Form </h3>

<!-- A bordered table for item orders -->
      <table>

<!-- First, the column headings -->
        <tr>
          <th> Product Name </th>
          <th> Price </th>
          <th> Quantity </th>
        </tr>

<!-- Now, the table data entries -->
        <tr>
          <th> French Vanilla (1 lb.) </th>
          <td> $3.49 </td>
```

```
          <td> <input type = "text"  id = "french"
                      size ="2" /> </td>
        </tr>
        <tr>
          <th> Hazlenut Cream (1 lb.) </th>
          <td> $3.95 </td>
          <td> <input type = "text"  id = "hazlenut"
                size = "2" /> </td>
        </tr>
        <tr>
          <th> Colombian (1 lb.) </th>
          <td> $4.59 </td>
          <td> <input type = "text"  id = "colombian"
                size = "2" /></td>
        </tr>
      </table>

<!-- Button for precomputation of the total cost -->
      <p>
        <input type = "button"  value = "Total Cost"
               onclick = "computeCost();" />
        <input type = "text"  size = "5"  id = "cost"
               onfocus = "this.blur();" />
      </p>

<!-- The submit and reset buttons -->
      <p>
        <input type = "submit"  value = "Submit Order" />
        <input type = "reset"  value = "Clear Order Form" />
      </p>
    </form>
  </body>
</html>
```

```
// nochange.js
//   This script illustrates using the focus event
//   to prevent the user from changing a text field

// The event handler function to compute the cost
function computeCost() {
  var french = document.getElementById("french").value;
```

```
  var hazlenut = document.getElementById("hazlenut").value;
  var colombian = document.getElementById("colombian").value;


// Compute the cost
  document.getElementById("cost").value =
  totalCost = french * 3.49 + hazlenut * 3.95 +
              colombian * 4.59;
}  //* end of computeCost
```

In this example, the button labeled `Total Cost` allows the user to compute the total cost of the order before submitting the form. The event handler for this button gets the values (input quantities) of the three kinds of coffee and computes the total cost. The cost value is placed in the text box's value property, and it is then displayed for the user. Whenever this text box acquires focus, it is forced to blur with the `blur` method, which prevents the user from changing the value.

5.7.2 Validating Form Input

One of the common uses of JavaScript is to check the values provided in forms by users to determine whether the values are sensible. Without client-side checks of such values, form values must be transmitted to the server for processing in the absence of any prior reality checks. The program or script on the server that processes the form data checks for invalid input data. When invalid data is found, the server must transmit that information back to the browser, which then must ask the user to resubmit corrected input. It is obviously more efficient to perform input data checks and carry on the user dialog concerning invalid input data entirely on the client. This approach shifts the task from the usually busy server to the client, which in most cases is only lightly used. It also results in less network traffic, because it avoids sending bad data to the server, only to have it returned without being processed. Furthermore, detecting incorrect form data on the client produces quicker responses to users. Validity checking of form data is often performed on the server as well, in part because client-side validity checking can be subverted by an unscrupulous user. Also, for some data, validity is crucial. For example, if the data is to be put in a database, invalid data could corrupt the database. Even though form data is checked on the server, any errors that can be detected and corrected on the client save server and network time.

When a user fills in a form input element incorrectly and a JavaScript event handler function detects the error, the function should produce an `alert` message indicating the error to the user and inform the user of the correct format for the input. Next, it would be good to put the element in focus, which would position the cursor in the element. This could be done with the `focus` method, but unfortunately, many recent versions of browsers

do not implement that method in a way that it consistently operates correctly. Therefore, we will not use it.

If an event handler returns `false`, that tells the browser not to perform any default actions of the event. For example, if the event is a click on the *Submit* button, the default action is to submit the form data to the server for processing. If user input is being validated in an event handler that is called when the `submit` event occurs and some of the input is incorrect, the handler should return `false` to avoid sending the bad data to the server. We use the convention that event handlers that check form data always return `false` if they detect an error and `true` otherwise.

When a form requests a password from the user and that password will be used in future sessions, the user is often asked to enter the password a second time for verification. A JavaScript function can be used to determine whether the entered passwords are the same.

The form in the next example includes the two password input elements, along with *Reset* and *Submit* buttons. The JavaScript function that checks the passwords is called either when the *Submit* button is clicked, using the `onsubmit` event to trigger the call, or when the second text box loses focus, using the `blur` event. The function performs two different tests. First, it determines whether the user typed the initial password (in the first input box) by testing the value of the element against the empty string. If no password has been typed into the first field, the function calls `alert` to produce an error message and returns `false`. The second test determines whether the two typed passwords are the same. If they are different, once again the function calls `alert` to generate an error message and returns `false`. If they are the same, it returns `true`. Following is the HTML document that creates the text boxes for the passwords, as well as the *Reset* and *Submit* buttons, and the two scripts for the event handlers and the event handler registrations for the example:

```
<!DOCTYPE html>
<!-- pswd_chk.html
     A document for pswd_chk.ps
     Creates two text boxes for passwords
     -->
<html lang = "en">
  <head>
    <title> Illustrate password checking> </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "pswd_chk.js" >
    </script>
  </head>
  <body>
    <h3> Password Input </h3>
    <form id = "myForm"  action = "" >
```

```
      <p>

      <label> Your password
        <input type = "password" id = "initial"
               size = "10" />
      </label>
      <br /><br />

      <label> Verify password
        <input type = "password"  id = "second"
               size = "10" />
      </label>
      <br /><br />

      <input type = "reset"  name = "reset" />
      <input type = "submit"  name = "submit" />
      </p>
    </form>

<!-- Script for registering the event handlers  -->
    <script type = "text/javascript"  src = "pswd_chkr.js">
    </script>

  </body>
</html>
```

```
// pswd_chk.js
//   An example of input password checking using the submit
//   event
// The event handler function for password checking
function chkPasswords() {
  var init = document.getElementById("initial");
  var sec = document.getElementById("second");
  if (init.value == "") {
    alert("You did not enter a password \n" +
          "Please enter one now");
    return false;
  }
```

```
  if (init.value != sec.value) {
    alert("The two passwords you entered are not the same \n" +
          "Please re-enter both now");
    return false;
  } else
    return true;
}
```

```
// pswd_chkr.js
//   Register the event handlers for pswd_chk.html

document.getElementById("second").onblur = chkPasswords;
document.getElementById("myForm").onsubmit = chkPasswords;
```

Figure 5.6 shows a browser display of `pswd_chk.html` after the two password elements have been input but before *Submit Query* has been clicked.

Figure 5.6 Display of `pswd_chk.html` after it has been filled out

Figure 5.7 shows a browser display that results from pressing the *Submit Query* button on `pswd_chk.html` after different passwords have been entered.

Figure 5.7 Display of `pswd_chk.html` after *Submit Query* has been clicked

We now consider an example that checks the validity of the form values for a name and phone number obtained from text boxes. Functions are used to check the form of each input when the values of the text boxes are changed—an event that is detected by the appearance of a `change` event.

In both cases, if an error is detected, an `alert` message is generated to prompt the user to fix the input. The `alert` message includes the correct format, which, for the name, is last-name, first-name, middle-initial, where the first and last names must begin with uppercase letters and have at least one lowercase letter. Both must be followed immediately by a comma and, possibly, one space. The middle initial must be uppercase and may or may not be followed by a period. There can be no characters before or after the whole name. The pattern for matching such names is as follows:

```
/^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/
```

Note the use of the anchors `^` and `$` on the ends of the pattern. This precludes any leading or trailing characters. Note also the question marks after the spaces (following the first and last names) and after the period. Recall that the question mark qualifier means zero or one of the qualified subpatterns. The period is backslashed, so it matches only a period.

The correct format of the phone number is three digits and a dash, followed by three digits and a dash, followed by four digits. As with names, no characters can precede or follow the phone number. The pattern for phone numbers is as follows:

```
/^d{3}-\d{3}-\d{4}$/
```

The following is the HTML document, `validator.html`, that displays the text boxes for a customer's name and phone number:

```
<!DOCTYPE html>
<!-- validator.html
     A document for validator.js
     Creates text boxes for a name and a phone number
     -->
<html lang = "en">
  <head>
    <title> Illustrate form input validation> </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "validator.js" >
    </script>
  </head>
  <body>
    <h3> Customer Information </h3>
    <form action = "">
      <p>
        <label>
```

```
            <input type = "text"  id = "custName" />
            Name (last name, first name, middle initial)
          </label>
          <br /><br />

          <label>
            <input type = "text"  id = "phone" />
            Phone number (ddd-ddd-dddd)
          </label>
          <br /><br />

          <input type = "reset"  id = "reset" />
          <input type = "submit"  id = "submit" />
        </p>
      </form>
      <script type = "text/javascript"  src = "validatorr.js">
      </script>
    </body>
  </html>
```

The following are the scripts for the event handlers and event registration for `validator.html`:

```
// validator.js
//   An example of input validation using the change and submit
//   events
// The event handler function for the name text box
function chkName() {
  var myName = document.getElementById("custName");
// Test the format of the input name
//   Allow the spaces after the commas to be optional
//   Allow the period after the initial to be optional
  var pos = myName.value.search(
            /^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/);
  if (pos != 0) {
    alert("The name you entered (" + myName.value +
          ") is not in the correct form. \n" +
          "The correct form is:" +
          "last-name, first-name, middle-initial \n" +
          "Please go back and fix your name");
    return false;
  } else
```

```
    return true;
}

// The event handler function for the phone number text box
function chkPhone() {
  var myPhone = document.getElementById("phone");

// Test the format of the input phone number
  var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);
  if (pos != 0) {
    alert("The phone number you entered (" + myPhone.value +
          ") is not in the correct form. \n" +
          "The correct form is: ddd-ddd-dddd \n" +
          "Please go back and fix your phone number");
    return false;
  } else
    return true;
}
```

```
// validatorr.js
//   Register the event handlers for validator.html

document.getElementById("custName").onchange = chkName;
document.getElementById("phone").onchange = chkPhone;
```

Figure 5.8 shows the browser screen of `validator.html` after entering a name in the correct format, followed by an invalid telephone number. The screen is shown before the user causes the phone text field to lose focus, either by pressing *Enter* or by clicking the left mouse button outside the phone text field.

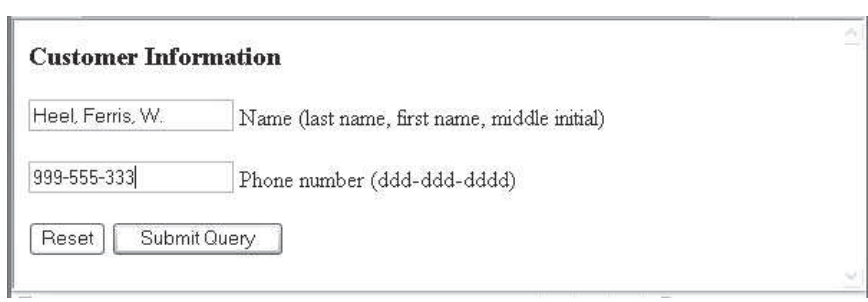


Figure 5.8 Display of `validator.html`, with an invalid phone number, while the `phone` text field has focus

Figure 5.9 shows the `alert` dialog box generated by pressing the *Enter* button in the `phone` text field of the screen of Figure 5.8.

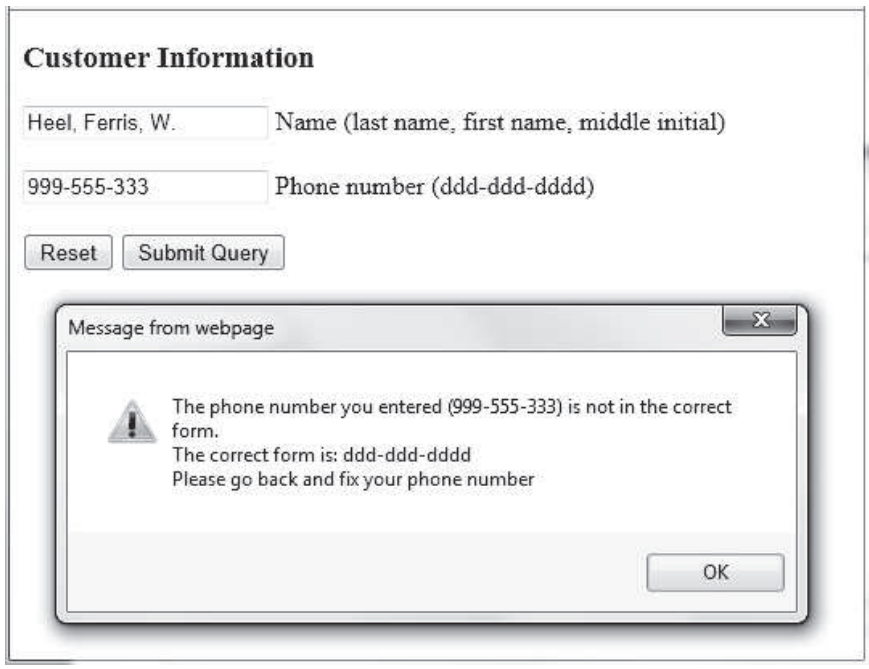


Figure 5.9 The message created by entering an invalid telephone number in `validator.html`

5.8 The DOM 2 Event Model

On the one hand, the DOM 2 event model does not include the features of the DOM 0 event model. However, there is no chance that support for those features will be dropped from browsers anytime soon. Therefore, Web authors should not hesitate to continue to use them. On the other hand, the DOM 2 event model is more sophisticated and powerful than that of DOM 0. One drawback of using the DOM 2 model is that versions of IE before IE9 do not support it.

The DOM 2 model is a modularized interface. One of the DOM 2 modules is Events, which includes several submodules. Those most commonly used are `HTMLEvents` and `MouseEvents`. Table 5.3 shows the interfaces and events defined by these modules.

Table 5.3

Module	Event Interface	Event Types
`HTMLEvents`	`Event`	`abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload`
`MouseEvents`	`MouseEvent`	`click, mousedown, mousemove, mouseout, mouseover, mouseup`

When an event occurs and an event handler is called, an object that implements the event interface associated with the event type is implicitly passed to the handler. (Section 5.8.1 explains how a handler is chosen to be called.) The properties of this object have information associated with the event.

The DOM 2 event model is relatively complex. This section covers only the basics of the model. A description of the rest of the model can be found at the W3C Web site.

5.8.1 Event Propagation

The connection between an event and the handler that deals with it is very simple in the DOM 0 event model. When the browser senses that an event has occurred, the object associated with the element that caused the event is checked for event handlers. If that object has a registered handler for the particular event that occurred, that handler is executed. The event handler connection for the DOM 2 event model is much more complicated.

Briefly, what happens is as follows: An event object is created at some node in the document tree. For that event, that node is called the *target node*. Event creation causes a three-phase process to begin.

The first of these phases is called the *capturing phase*. The event created at the target node starts at the document root node and propagates down the tree to the target node. If there are any handlers for the event that are registered on any node encountered in this propagation, including the document node but not the target node, these handlers are checked to determine whether they are enabled. (Section 5.8.2 explains how a handler can be defined to be enabled.) Any enabled handler for the event that is found during capturing is executed. When the event reaches the target node, the second phase, called the *target node phase*, takes place. In this phase, the handlers registered for the event at the target node are executed, regardless of whether they are enabled or not. The process is similar to what happens with the DOM 0 event model. After execution of any appropriate handlers at the target node, the third phase begins. This is the *bubbling phase*, in which the event bubbles back up the document tree to the document node. On this trip back up the tree, any handler registered for the event at any node on the way is executed (whether it is enabled or not).

Not all events bubble. For example, the `load` and `unload` events do not bubble. All of the mouse events, however, do. In general, if it makes sense to handle an event farther up the document tree than the target node, the event bubbles; otherwise, it does not.

Any handler can stop the event from further propagation by using the `stopPropagation` method of the event object.

The bubbling idea was borrowed from exception handling. In a large and complicated document, having event handlers for every element would require a great deal of code. Much of this code would be redundant, both in the handlers and in the registering of handlers for events. Therefore, it was better to define a way for a single handler to deal with events created from a number of similar or related elements. The approach is simple: Events can be propagated to some central place for handling, rather than always being handled locally. In the DOM,

the natural central place for event handling is at the document or window level, so that is the direction of bubbling.

Many events cause the browser to perform some action; for example, a mouse click on a link causes the document referenced in the link to replace the current document. In some cases, we want to prevent this action from taking place. For example, if a value in a form is found to be invalid by a *Submit* button event handler, we do not want the form to be submitted to the server. In the DOM 0 event model, the action is prevented by having the handler return false. The DOM 2 event interface provides a method, `preventDefault`, that accomplishes the same thing. Every event object implements `preventDefault`.

5.8.2 Event Handler Registration

The DOM 0 event model uses two different ways of registering event handlers. First, the handler code can be assigned as a string literal to the event's associated attribute in the element. Second, the name of the handler function can be assigned to the property associated with the event. Handler registration in the DOM 2 event model is performed by the method `addEventListener`, which is defined in the `EventTarget` interface, which is implemented by all objects that descend from Document.[3]

The `addEventListener` method takes three parameters, the first of which is the name of the event as a string literal. For example, `"mouseup"` and `"submit"` would be legitimate first parameters. The second parameter is the handler function, which could be specified as the function code itself in the form of an anonymous function definition or as the name of a function that is defined elsewhere. Note that this parameter is not a string type, so it is not quoted. The third parameter is a Boolean value that specifies whether the handler is enabled for calling during the capturing phase. If the value `true` is specified, the handler is enabled for the capturing phase. In fact, an enabled handler can be called *only* during capturing. If the value is `false`, the handler is not enabled and can be called either at the target node or on any node reached during bubbling.

When a handler is called, it is passed a single parameter, the `event` object. For example, suppose we want to register the event handler `chkName` on the text input element whose `id` is `custName` for the `change` event. The following call accomplishes the task:

```
document.custName.addEventListener(
                    "change", chkName, false);
```

In this case, we want the handler to be called at the target node, which is `custName` in this example, so we passed `false` as the third parameter.

Sometimes it is convenient to have a temporary event handler. This can be done by registering the handler for the time when it is to be used and then deleting that registration. The `removeEventListener` method deletes the

3. The name of this method includes "listener" rather than "handler" because handlers are called listeners in the DOM 2 specification. This is also the term used in Java for widget event handlers.

registration of an event handler. This method takes the same parameters as `addEventListener`.

With the DOM 0 event model, when an event handler is registered to a node, the handler becomes a method of the object that represents that node. This approach makes every use of `this` in the handler a reference to the target node. FX3+ browsers implement event handlers for the DOM 2 model in the same way. However, this is not required by the DOM 2 model, so some other browsers may not use such an approach, making the use of `this` in a handler potentially nonportable. The safe alternative is to use the `currentTarget` property of `Event`, which will always reference the object on which the handler is being executed. If the handler is called through the object of the target node, `currentTarget` is the target node. However, if the handler is called during capturing or bubbling, `currentTarget` is the object through which the handler is called, which is not the target node object. Another property of `Event`, `target`, is a reference to the target node.

The `MouseEvent` interface inherits from the Event interface. It adds a collection of properties related to mouse events. The most useful of these are `clientX` and `clientY`, which have the x- and y-coordinates of the mouse cursor, relative to the upper-left corner of the client area of the browser window. The whole browser window is taken into account, so if the user has scrolled down the document, the `clientY` value is measured from the top of the document, not the top of the current display.

5.8.3 An Example of the DOM 2 Event Model

The next example is a revision of the `validator.html` document and `validator.js` script from Section 5.7, which used the DOM 0 event model. Because this version uses the DOM 2 event model, it does not work with IE8. Notice that no call to `preventDefault` appears in the document. The only event handled here is `change`, which has no default actions, so there is nothing to prevent. Here is the document and the JavaScript file:

```
<!DOCTYPE html>
<!-- validator2.html
     A document for validator2.js
     Creates text boxes for a name and a phone number
     Note: This document does not work with IE browsers before IE9
     -->
<html lang = "en">
  <head>
    <title> Illustrate form input validation with DOM 2> </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "validator2.js" >
    </script>
  </head>
```

```
  <body>
    <h3> Customer Information </h3>
    <form action = "">
      <p>
        <label>
          <input type = "text"  id = "custName" />
           Name (last name, first name, middle initial)
        </label>
        <br /><br />


        <label>
          <input type = "text"  id = "phone" />
          Phone number (ddd-ddd-dddd)
        </label>
        <br /><br />


        <input type = "reset" />
        <input type = "submit"  id = "submitButton" />
      </p>
    </form>
<!-- Script for registering event handlers -->
    <script type = "text/javascript"  src = "validator2r.js" />
  </body>
</html>
```

```
// validator2.js
//   An example of input validation using the change and submit
//   events using the DOM 2 event model
//   Note: This document does not work with IE8

// ******************************************************** //
// The event handler function for the name text box
function chkName(event) {

// Get the target node of the event
  var myName = event.currentTarget;


// Test the format of the input name
//  Allow the spaces after the commas to be optional
//  Allow the period after the initial to be optional
  var pos = myName.value.search(
                    /^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/);
```

```
  if (pos != 0) {
    alert("The name you entered (" + myName.value +
          ") is not in the correct form. \n" +
          "The correct form is: " +
          "last-name, first-name, middle-initial \n" +
          "Please go back and fix your name");
  }
}
// ******************************************************* //
// The event handler function for the phone number text box
function chkPhone(event) {

// Get the target node of the event
  var myPhone = event.currentTarget;

// Test the format of the input phone number
  var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);

  if (pos != 0) {
    alert("The phone number you entered (" + myPhone.value +
          ") is not in the correct form. \n" +
          "The correct form is: ddd-ddd-dddd \n" +
          "Please go back and fix your phone number");
  }
}
```

```
// validator2r.js
//   The last part of validator2. Registers the
//   event handlers
//   Note: This script does not work with IE8

// Get the DOM addresses of the elements and register
//  the event handlers
      var customerNode = document.getElementById("custName");
      var phoneNode = document.getElementById("phone");
      customerNode.addEventListener("change", chkName, false);
      phoneNode.addEventListener("change", chkPhone, false);
```

Note that the two event models can be mixed in a document. If a DOM 0 feature happens to be more convenient than the corresponding DOM 2 feature, there is no reason it cannot be used. Chapter 6 includes an example of the use of the DOM 2 event model for something that is more difficult to do with the DOM 0 event model.

5.9 The `canvas` Element

A canvas element creates a rectangle into which bit-mapped graphics can be drawn using JavaScript. The canvas element usually includes three attributes, `height`, `width`, and `id`, although all three are optional. The attributes for height and width are given as nonnegative integers, which specify the dimensions of the canvas rectangle in pixels.[4] The default values for `height` and `width` are `150` and `300`, respectively. The `id` attribute is required to allow anything to be drawn on the canvas rectangle. The content of a canvas element is displayed when the browser does not support canvas. Following is an example of a canvas element:

```
<canvas  id = "myCanvas"  height = "200"  width = "400">
Your browser does not support the canvas element
</canvas>
```

The content of the `canvas` element is displayed if the browser does not support the element.

As with the image tag, canvas requires the explicit closing tag.

To allow any kind of drawing on a canvas element, you must first get a drawing context for it. This is done by getting its DOM address and then calling the `getContext` method on it, including the parameter `'2d'`, which specifies two-dimensional drawing. For example, if we have created a canvas element with the following:

```
<canvas id = "myCanvas"  width = "400"  height = "150" >
Your browser cannot display canvas drawings
</canvas>
```

We can get a drawing context for this canvas with the following:

```
var context = myCanvas.getContext('2d');
```

To avoid error messages, the call to `getContext` and the actual drawing code is usually placed in a selector construct that chooses it only if the `getContext` method exists. The following code is often used for this:

```
if (myCanvas.getContext) {
  var context = myCanvas.getContext('2d');
   // Drawing code
}
```

In many cases, whatever is drawn with canvas is displayed when the document in which it is embedded is loaded. For this, we put the drawing code in a function and call the function with the `onload` attribute of the body element.

The canvas element creates a rectangular drawing surface that is initially blank. The origin of this surface is the upper-left corner.

There is only one primitive shape, rectangle, that can be drawn on a canvas surface. Rectangles can be drawn as just a stroke with `strokeRect` and as filled

4. The dimensions are not always interpreted as pixels. On a high-resolution display, two actual pixels may represent each specified pixel.

shapes with fillRect. These two methods take the same four parameters, the first two of which are for the horizontal and vertical pixel position of the rectangle, measured from the upper-left corner of the canvas. The last two parameters specify the width and height in pixels of the rectangle to be drawn. There is also a third rectangle method which takes the same parameters as the other two, clearRect, which erases the fill in the rectangle it defines. The following example, consisting of an HTML document and a JavaScript file, illustrates the use of these three methods:

```
// rects.js
// This script illustrates the use of the rectangle methods of
//  the canvas element to draw two rectangles.

function draw() {
  var dom = document.getElementById("myCanvas");
  if (dom.getContext) {
    var context = dom.getContext('2d');

// Draw the outer filled rectangle
    context.fillRect(100, 100, 200, 200);

// Clear a rectangle inside the first rectangle
    context.clearRect(150, 150, 100, 100);

// Draw a stroke rectangle inside the others
    context.strokeRect(180, 180, 40, 40);

// Draw a small filled rectangle in the center of the others
    context.fillRect(195, 195, 10, 10);
  }
}
```

Figure 5.10 shows the display that results from calling the draw method of rects.js.

Straight lines and curves are drawn by creating a path and then directing the path wherever it takes to draw the figure you want. The path is created with a call to beginPath, which takes no parameters. The initial point on the path is specified with a call to moveTo, which takes the horizontal and vertical pixel positions within the canvas as its two parameters. This method in effect moves the drawing pen after raising from the canvas. The lineTo method draws a straight line from the current position to the position specified by its two parameters. The stroke method actually draws the lines. Alternatively, the fill method can be called to fill the drawn figure. If neither of these two is called, nothing is drawn. These two

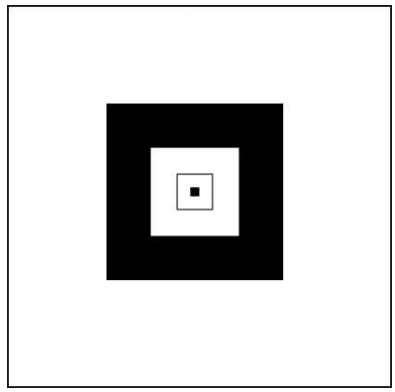


Figure 5.10 Display of the `draw` method of `rects.js`

methods do not take parameters. The following `draw` method uses `beginPath`, `moveTo`, and `lineTo`, and `stroke` to draw a parallelogram:

```
// parallel.js
// This script illustrates the use of the methods for
//  drawing lines by drawing a parallelogram.

function draw() {
  var dom = document.getElementById("myCanvas");
  if (dom.getContext) {
    var context = dom.getContext('2d');
    context.beginPath();
    context.moveTo(50, 150);
    context.lineTo(100, 100);
    context.lineTo(200, 100);
    context.lineTo(150, 150);
    context.lineTo(50, 150);
    context.stroke();
  }
}
```

Figure 5.11 shows the display that results from calling the `draw` method of `parallel.js`.

The arc method is used to draw arcs and circles. It takes six parameters, the first two of which provide the position of the center of the arc or circle. The third parameter is the radius of the arc or circle. The fourth parameter is the angle at which the drawing should begin. The fifth parameter is the angle at which the drawing should end. These two parameters must be given in radians. Degrees can be converted to radians with the following expression:

```
radians = (Math.PI / 180) * degrees
```

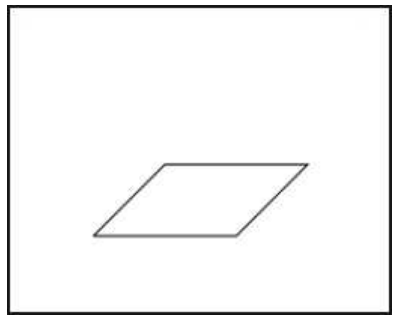


Figure 5.11 Display of the `draw` method of `parallel.js`

The sixth parameter, which is a Boolean value, specifies whether the drawing is to be clockwise (`false`) or counterclockwise (`true`).

The following `draw` method draws two figures, one that shows two concentric circles, the outer just a stroke and the inner filled; the second shows a filled circle with a notch cut out of the right side:

```
// circles.js
// This script illustrates the use of the methods for
//  drawing circles.

function draw() {
  var dom = document.getElementById("myCanvas");
  if (dom.getContext) {
    var context = dom.getContext('2d');

// Draw the outer stroke circle
    context.beginPath();
    context.arc(200, 200, 100, 0, 2 * Math.PI, false);
    context.stroke();

// Draw the inner filled circle
    context.beginPath();
    context.arc(200, 200, 50, 0, 2 * Math.PI, false);
    context.fill();

// Draw Pac-Man
    context.beginPath();
    context.arc(500, 200, 50, Math.PI/7, -Math.PI/7, false);
    context.lineTo(500, 200);
    context.fill();
  }
}
```

Figure 5.12 shows the display that results from calling the `draw` method of `circles.js`.

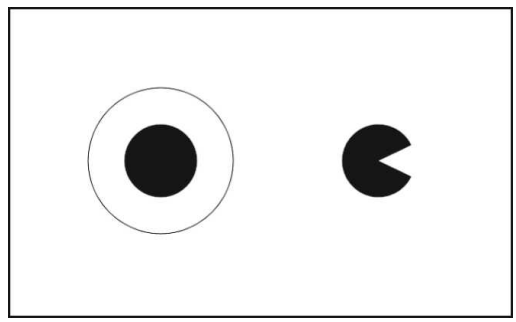


Figure 5.12 Display of the `draw` method of `circles.js`

Notice that the second figure only required one line. This is because a filled figure will complete its own boundary.

There are many more capabilities with the canvas elements than are shown here. For example, in addition to arcs and rectangles, Bezier and quadratic curves can be drawn. Images can be integrated into the content of a canvas element. Furthermore, the content of a canvas element can be animated.

5.10 The `navigator` Object

The navigator object indicates which browser is being used to view the HTML document. The browser's name is stored in the `appName` property of the object. The version of the browser is stored in the `appVersion` property of the object. These properties allow the script to determine which browser is being used and to use processes appropriate to that browser. The following example illustrates the use of `navigator`, in this case just to display the browser name and version number:

```
<!DOCTYPE html>
<!-- navigate.html
     A document for navigate.js
     Calls the event handler on load
     -->
<html lang = "en">
```

```
  <head>
    <title> navigate.html </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "navigate.js" >
    </script>
  </head>
  <body onload = "navProperties()">
  </body>
</html>
```

```
// navigate.js
//  An example of using the navigator object


// The event handler function to display the browser name
//  and its version number
function navProperties() {
  alert("The browser is: " + navigator.appName + "\n" +
    "The version number is: " + navigator.appVersion + "\n");
}
```

Figure 5.13 shows the result of displaying `navigate.html` with FX3. Figure 5.14 shows the result of displaying `navigate.html` with IE10. Notice that the version number of IE10 is 5. Microsoft intentionally set the version number to 5 because of some compatibility issues with earlier browsers. Firefox is not any better in this regard: Using FX3, it displays `version 5.0`. The C12 browser says it is Netscape version 5.0, as with FX3.

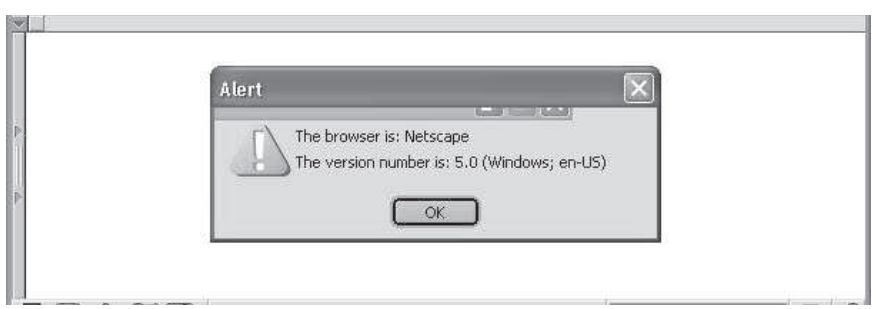


Figure 5.13 The `navigator` properties `appName` and `appVersion` for FX3

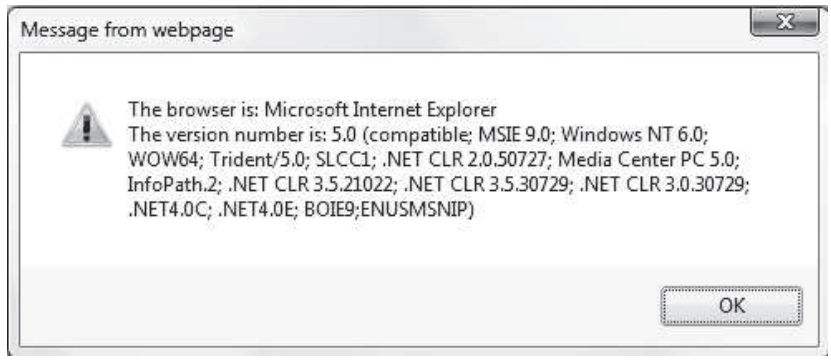


Figure 5.14 The `navigator` properties `appName` and `appVersion` for IE10

5.11 DOM Tree Traversal and Modification

There are many objects, properties, and methods associated with DOM 2 document representations that we have not discussed. One collection of these is defined in the `Node` interface, which is implemented by all node objects in the DOM structure. Some can be used to traverse and modify the DOM tree structure of the document being displayed. In this section, a few of the most useful ones are briefly described. All the properties and methods mentioned here are supported by IE9+, C10+, and FX3+.

5.11.1 DOM Tree Traversal

The `parentNode` property has the DOM address of the parent node of the node through which it is referenced. The `childNodes` property is an array of the child nodes of the node through which it is referenced. For example, if the document has an unordered list with the id `mylist`, the number of items in the list can be displayed with the following code:

```
var nod = document.getElementById("mylist");
var listitems = nod.childNodes.length;
document.write("Number of list items is: " +
                listitems + "<br />");
```

The `previousSibling` property has the DOM address of the previous sibling node of the node through which it is referenced. The `nextSibling` property has the DOM address of the next sibling node of the node through which it is referenced. The `firstChild` and `lastChild` properties have the DOM addresses of the first and last child nodes, respectively, of the node through which they are referenced. The `nodeType` property has the type of the node through which it is referenced.

5.11.2 DOM Tree Modification

A number of methods allow JavaScript code to modify an existing DOM tree structure. The `insertBefore(newChild, refChild)` method places the `newChild` node before the `refChild` node. The `replaceChild(newChild, oldChild)`

method replaces the `oldChild` node with the `newChild` node. The `removeChild(oldChild)` method removes the specified node from the DOM structure. The `appendChild(newChild)` method adds the given node to the end of the list of siblings of the node through which it is called.

Summary

The highest levels of the execution environment of client-side JavaScript are represented with the `Window` and `Document` objects. The `Document` object includes a `forms` array property, which contains references to all forms in the document. Each element of the `forms` array has an `elements` array, which contains references to all elements in the form.

The DOM is an abstract interface whose purpose is to provide a language-independent way to access the elements of an HTML document. Also included are the means to navigate around the structure in which the HTML elements appear. HTML tags are represented in JavaScript as objects; tag attributes are represented as properties.

There are three different ways to access HTML elements in JavaScript: through the `forms` and `elements` arrays, through the names of the element and its enclosing elements, and through the `getElementById` method.

Events are simply notifications that something specific has happened that may require some special processing. Event-handling code provides that special processing. There are two distinct event models currently in use. The first is the model implemented by all browsers that support JavaScript: the DOM 0 model. The second is the more elaborate and powerful model defined in DOM 2.

With the DOM 0 model, there are two ways to register an event handler. First, an attribute of the tag that defines the HTML element can be assigned the handler code. Second, the property associated with the event of the object that represents the HTML element can be assigned the name of a function that implements the handler. The `write` method of `document` should not be used in event handlers.

With the DOM 0 model, each event has an associated tag attribute. A particular attribute may appear in several different tags. Each of these appearances is identified as a different occurrence of the same event. The `load` and `unload` events are often used with the `<body>` tag to perform some operation when a document has been loaded and unloaded, respectively. The `click` event is used for all the different HTML buttons, as well as for the link of an anchor tag. Form input can be conveniently checked using the `change` event. The `submit` event can also be used to check form data just before the form is submitted.

The DOM 2 event model defines three phases of event processing: capturing, target node, and bubbling. During the capturing phase, the event object travels from the document root to the target node, where the event was created. During the bubbling phase, the event travels back up the document tree to the root, triggering any handlers registered on nodes that are encountered. Event handlers can be set to allow them to be triggered during the capturing phase. Event handler registration is done with the `addEventListener` method, which sets whether capturing-phase triggering will take place. Events can be unregistered with the

removeEventListener method. The currentTarget property of Event has the object through which the handler was called. The target property has the target node object. The mouseEvent object has two properties—clientX and clientY—which have the coordinates of the position of the mouse cursor in the browser display window when a mouse event occurs.

The canvas element creates a rectangular area in the display of a document in which lines, rectangles, and arcs can be drawn with JavaScript methods.

The navigator object has information about which browser is being used, as well as its version number and other related information.

There are many objects, methods, and properties defined in DOM 2 that are used to traverse and modify the DOM tree structure of a document.

Review Questions

5.1 Global variables in JavaScript are properties of what object?

5.2 How are HTML elements and attributes represented in the JavaScript binding to DOM?

5.3 What is an event?

5.4 What is an event handler?

5.5 What is the origin of the DOM 0 event model?

5.6 What are the two ways in which an event handler can be associated with an event generated by a specific HTML element in the DOM 0 event model?

5.7 Why should document.write not be used in an event handler?

5.8 In what ways can an HTML form element acquire focus?

5.9 Describe the approach to addressing HTML elements using forms and elements.

5.10 Describe the approach to addressing HTML elements using name attributes.

5.11 Describe the approach to addressing HTML elements using getElementById.

5.12 What is the disadvantage of assigning event handlers to event properties?

5.13 What are the advantages of assigning event handlers to event properties?

5.14 Why is it good to use JavaScript to check the validity of form inputs before the form data is sent to the server?

5.15 What three things should be done when a form input element is found to have incorrectly formatted data?

5.16 What happens when an event handler for the onsubmit event returns false?

5.17 What event is used to trigger an event handler that checks the validity of input for a text button in a form?

5.18 What event propagation takes place in the DOM 0 event model?

5.19 Explain the three phases of event processing in the DOM 2 event model.

5.20 Give two examples of default actions of events.

5.21 Explain the first two parameters of the `addEventListener` method.

5.22 How is an event handler registered so that it will be called during the capturing phase?

5.23 How can an event handler be unregistered?

5.24 What exactly do the `clientX` and `clientY` properties store?

5.25 What is the purpose of the `canvas` element?

5.26 What exactly does the `moveTo` method do?

5.27 Explain the parameters to the `arc` method.

5.28 What purpose does the `navigator` object have?

Exercises

5.1 Modify the `radio_click.html` example to have five buttons, labeled *red*, *blue*, *green*, *yellow*, and *orange*. The event handlers for these buttons must produce messages stating the chosen favorite color. The event handler must be implemented as a function whose name must be assigned to the `onclick` attribute of the radio button elements. The chosen color must be sent to the event handler as a parameter.

5.2 Rewrite the document for Exercise 5.1 to assign the event handler to the event property of the button element. This requires the chosen color to be obtained from the `value` property of the button element rather than through the parameter.

5.3 Develop and test an HTML document that has checkboxes for apple (59 cents each), orange (49 cents each), and banana (39 cents each), along with a *Submit* button. Each of the checkboxes should have its own `onclick` event handler. These handlers must add the cost of their fruit to a total cost. An event handler for the *Submit* button must produce an `alert` window with the message *Your total cost is $xxx*, where *xxx* is the total cost of the chosen fruit, including 5 percent sales tax. This handler must return `false` (to avoid actual submission of the form data).

5.4 Develop and test an HTML document that is similar to that of Exercise 5.3. In this case, use text boxes rather than checkboxes. These text boxes take

a number, which is the purchased number of the particular fruit. The rest of the document should behave exactly like that of Exercise 5.3.

5.5 Add reality checks to the text boxes of the document in Exercise 5.4. The checks on the text box inputs should ensure that the input values are numbers in the range from 0 to 99.

5.6 Range checks for element inputs can be represented as new properties of the object that represents the element. Modify the document in Exercise 5.5 to add a `max` property value of 99 and a `min` property value of 0. Your event handler must use the properties for the range checks on values input through the text boxes.

5.7 Develop and test an HTML document that collects the following information from the user: last name, first name, middle initial, age (restricted to be greater than 17), and weight (restricted to the range from 80 to 300). You must have event handlers for the form elements that collect this information. These handlers must check the input data for correctness. Messages in `alert` windows must be produced when errors are detected.

5.8 Revise the document of Exercise 5.1 to use the DOM 2 event model.

5.9 Revise the document of Exercise 5.3 to use the DOM 2 event model.

5.10 Develop and test an HTML document and a JavaScript script to draw a filled square with an empty circle inside it.

5.11 Develop and test an HTML document and a JavaScript script to draw the Olympics logo.

CHAPTER

6

Dynamic Documents with JavaScript

Informally, a dynamic Hypertext Markup Language (HTML) document is one that, in some way, can be changed while it is being displayed by a browser. The most common client-side approach to providing dynamic documents is to use JavaScript to manipulate the objects of the Document Object Model (DOM) of the displayed document. Changes to documents can occur when they are explicitly requested by user interactions, at regular timed intervals, or when browser events occur.

HTML elements can be initially positioned at any given location on the browser display. If they're positioned in a specific way, elements can be dynamically moved to new positions on the display. Elements can be made to disappear

and reappear. The colors of the background and the foreground (the elements) of a document can be changed. The font, font size, and font style of displayed text can be changed. The content of an element also can be changed. Overlapping elements in a document can be positioned in a specific top-to-bottom stacking order, and their stacking arrangement can be dynamically changed. The position of the mouse cursor on the browser display can be determined when a mouse button is clicked. Elements can be made to move around the display screen. Finally, elements can be defined to allow the user to drag and drop them anywhere in the display window. This chapter discusses the JavaScript code that can create all these effects.

6.1 Introduction

Dynamic HTML is not a new markup language. It is a collection of technologies that allows dynamic changes to documents defined with HTML. Specifically, a *dynamic HTML document* is an HTML document whose tag attributes, tag contents, or element style properties can be changed by user interaction or the occurrence of a browser event after the document has been, and is still being, displayed. Such changes can be made with an embedded script that accesses the elements of the document as objects in the associated DOM structure.

Support for dynamic HTML is not uniform across the various browsers. As in Chapter 5, the discussion here is restricted to W3C-standard approaches rather than including features defined by a particular browser vendor. All the examples in this chapter, except the document in Section 6.11, use the DOM 0 event model and work on both Internet Explorer 8 (IE8) and Firefox 3 (FX3) browsers. The example in Section 6.11 uses the DOM 2 event model because it cannot be designed in a standard way with the DOM 0 event model. Because IE8 (and earlier versions of IE) does not support the DOM 2 event model, that example does not work with IE8. However, the IE9+ browsers support the DOM 2 event model, so that example works with them.

This chapter discusses user interactions through HTML documents using client-side JavaScript. Chapters 8 through 10 discuss user interactions through HTML documents using server-side technologies.

6.2 Positioning Elements

Before the browsers that implemented HTML 4.0 appeared, Web site authors had little control over how HTML elements were arranged on a display. In many cases, the elements found in the HTML file were simply placed on the display the way text is placed in a document with a word processor: Fill a row, start a new row, fill it, and so forth. HTML tables provide a framework of columns for arranging elements, but they lack flexibility and also take a considerable time to display.[1] This lack of powerful and efficient element placement control ended

1. Frames provide another way to arrange elements, but they were deprecated in XHTML 1.0 and eliminated in XHTML 1.1 and HTML5.

when Cascading Style Sheets–Positioning (CSS-P) was released by the World Wide Web Consortium (W3C) in 1997.

CSS-P is completely supported by IE8+, FX3+, and C12+. It provides the means not only to position any element anywhere in the display of a document, but also to move an element to a new position in the display dynamically, using JavaScript to change the positioning style properties of the element. These style properties, which are appropriately named `left` and `top`, dictate the distance from the left and top of some reference point to where the element is to appear. Another style property, `position`, interacts with `left` and `top` to provide a higher level of control of placement and movement of elements. The `position` property has three possible values: `absolute`, `relative`, and `static`.

6.2.1 Absolute Positioning

The `absolute` value for `position` is specified when the element is to be placed at a specific location in the document display without regard to the positions of other elements. For example, if a paragraph of text is to appear 100 pixels from the left edge and 200 pixels from the top of the display window, the following element could be used:

```
<p style = "position: absolute; left: 100px; top: 200px";>
    -- text --
</p>
```

One use of absolute positioning is to superimpose special text over a paragraph of ordinary text to create an effect similar to a watermark on paper. A larger italicized font, in a light-gray color and with space between the letters, could be used for the special text, allowing both the ordinary text and the special text to be legible. Remember that `em` is a relative size, so the text size of embedded elements is relative to their parents. The HTML document that follows provides an example that implements this effect. In this example, a paragraph of normal text that describes apples is displayed. Superimposed on this paragraph is the somewhat subliminal message APPLES ARE GOOD FOR YOU. Here is the document:

```
<!DOCTYPE html>
<!-- absPos.html
     Illustrates absolute positioning of elements
     -->
<html lang = "en">
  <head>
    <title> Absolute positioning </title>
    <meta charset = "utf-8" />
    <style type = "text/css">

/* A style for a paragraph of text */
     .regtext {font-family: Times; font-size: 1.2em; width: 500px}
```

```
/* A style for the text to be absolutely positioned */
     .abstext {position: absolute; top: 25px; left: 25px;
               font-family: Times; font-size: 1.9em;
               font-style: italic; letter-spacing: 1em;
               color: rgb(160,160,160); width: 450px}
    </style>
  </head>
  <body>
    <p class = "regtext">
      Apple is the common name for any tree of the genus Malus,
      of the family Rosaceae. Apple trees grow in any of the
      temperate areas of the world. Some apple blossoms are white,
      but most have stripes or tints of rose. Some apple blossoms
      are bright red. Apples have a firm and fleshy structure that
      grows from the blossom. The colors of apples range from
      green to very dark red. The wood of apple trees is fine
      grained and hard. It is, therefore, good for furniture
      construction. Apple trees have been grown for many
      centuries. They are propagated by grafting because they
      do not reproduce themselves.
      <span class = "abstext">
        APPLES ARE GOOD FOR YOU
      </span>
    </p>
  </body>
</html>
```

Figure 6.1 shows a display of `absPos.html`.

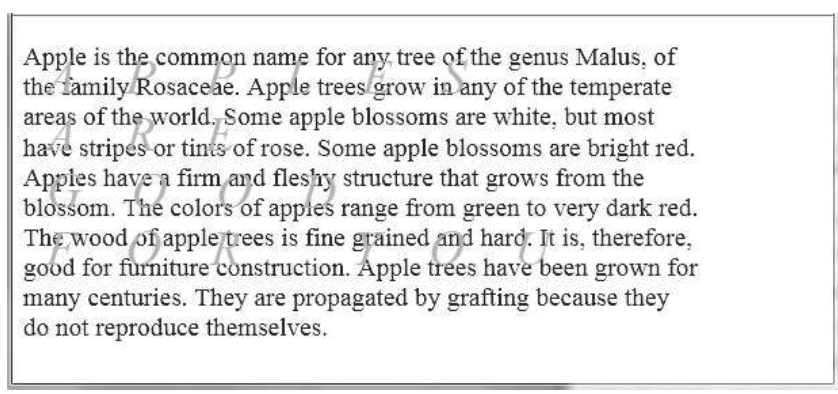


Figure 6.1 Display of `absPos.html`

Notice that a `width` property value is included in the style for both the regular and the special text. This property is used here to ensure that the special text is uniformly embedded in the regular text. Without it, the text would extend to the right end of the browser display window—and, of course, the width of the window could vary widely from client to client and even from minute to minute on the same client (because the user can resize the browser window at any time).

When an element is absolutely positioned inside another positioned element (one that has the `position` property specified), the `top` and `left` property values are measured from the upper-left corner of the enclosing element (rather than the upper-left corner of the browser window).

To illustrate the placement of nested elements, the document `absPos.html` is modified to place the regular text 100 pixels from the top and 100 pixels from the left. The special text is nested inside the regular text by using `<div>` and `<span>` tags. The modified document, which is named `absPos2.html`, is as follows:

```
<!DOCTYPE html>
<!-- absPos2.html
     Illustrates nested absolute positioning of elements
     -->
<html lang = "en">
  <head>
    <title> Nested absolute positioning </title>
    <meta charset = "utf-8" />
    <style type = "text/css">

/* A style for a paragraph of text */
     .regtext {font-family: Times; font-size: 1.2em; width: 500px;
               position: absolute; top: 100px; left: 100px;}

/* A style for the text to be absolutely positioned */
     .abstext {position: absolute; top: 25px; left: 25px;
               font-family: Times; font-size: 1.9em;
               font-style: italic; letter-spacing: 1em;
               color: rgb(160,160,160); width: 450px;}
    </style>
  </head>
  <body>
    <p class = "regtext">
      Apple is the common name for any tree of the genus Malus,
      of the family Rosaceae. Apple trees grow in any of the
      temperate areas of the world. Some apple blossoms are white,
```

```
        but most have stripes or tints of rose. Some apple blossoms
        are bright red. Apples have a firm and fleshy structure that
        grows from the blossom. The colors of apples range from
        green to very dark red. The wood of apple trees is fine
        grained and hard. It is, therefore, good for furniture
        construction. Apple trees have been grown for many
        centuries. They are propagated by grafting because they
        do not reproduce themselves.
        <span class = "abstext">
          APPLES ARE GOOD FOR YOU
        </span>
      </p>
    </body>
  </html>
```

Figure 6.2 shows a display of `absPos2.html`.

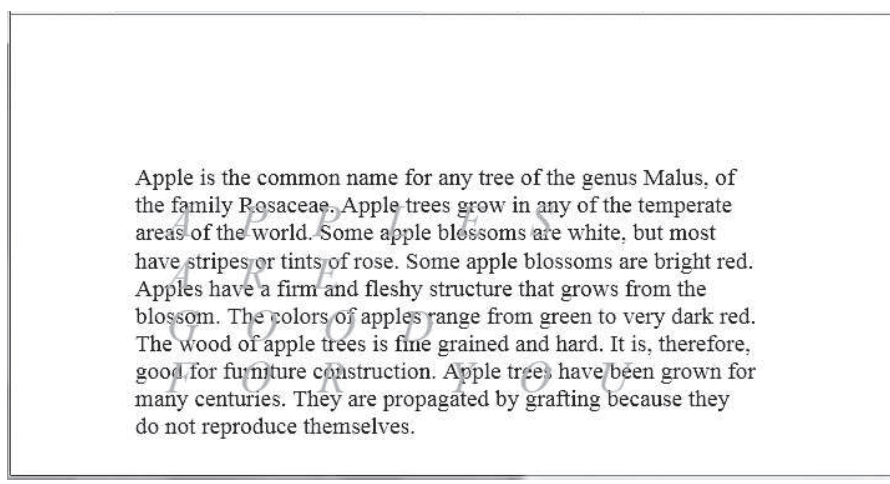


Figure 6.2 Display of `absPos2.html`

6.2.2 Relative Positioning

An element that has the `position` property set to `relative`, but does not specify `top` and `left` property values, is placed in the document as if the `position` attribute were not set at all. However, such an element can be moved later. If the `top` and `left` properties are given values, they displace the element by the specified amount from the position where it would have been placed (if `top` and `left` had not been set). For example, suppose that two buttons are placed in a document and the `position` attribute has its default value, which is `static`.

Then the buttons would appear next to each other in a row, assuming that the current row has sufficient horizontal space for them. If `position` has been set to `relative` and the second button has its `left` property set to `50px`, the effect would be to move the second button 50 pixels farther to the right than it otherwise would have appeared.

In both, the case of an absolutely positioned element inside another element and the case of a relatively positioned element, negative values of `top` and `left` displace the element upward and to the left, respectively.[2]

Relative positioning can be used for a variety of special effects in placing elements. For example, it can be used to create superscripts and subscripts by placing the values to be raised or lowered in `<span>` tags and displacing them from their regular positions. In the next example, a line of text is set in a normal font style in `2em` size. Embedded in the line is one word that is set in italic, `2em`, red font. Its size is also set to `2em`, but because its parent is the paragraph in which it is embedded, the `2em` means it will be twice as large as the paragraph text. Normally, the bottom of the special word would align with the bottom of the rest of the line. In this case, the special word is to be vertically centered in the line, so its `position` property is set to `relative` and its `top` property is set to 15 pixels, which lowers it by that amount relative to the surrounding text. The HTML document to specify this, which is named `relPos.html`, is as follows:

```
<!DOCTYPE html>
<!-- relPos.html
     Illustrates relative positioning of elements
     -->
<html lang = "en">
  <head>
    <title> Relative positioning </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      .regtext {font: 2em Times}
      .spectext {font: 2em Times; color: red; position: relative;
                 top: 15px;}
    </style>
  </head>
  <body>
    <p class = "regtext">
      Apples are
      <span class = "spectext"> GOOD </span> for you.
    </p>
  </body>
</html>
```

2. Of course, if the `left` or `top` property is set to a negative value for an absolutely positioned element, only part of the element (or possibly none of the element) will be visibly displayed.

Figure 6.3 shows a display of `relPos.html`.

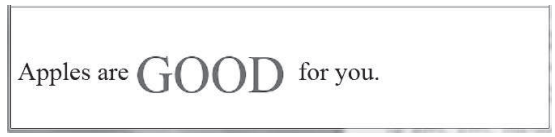


Figure 6.3 Display of `relPos.html`

6.2.3 Static Positioning

The default value for the `position` property is `static`. A statically positioned element is placed in the document as if it had the `position` value of `relative` but no values for `top` or `left` were given. The difference is that a statically positioned element cannot have its `top` or `left` properties initially set or changed later. Therefore, a statically placed element initially cannot be displaced from its normal position and cannot be moved from that position later.

6.3 Moving Elements

As stated previously, an HTML element whose `position` property is set to either `absolute` or `relative` can be moved. Moving an element is simple: Changing the `top` or `left` property values causes the element to move on the display. If its `position` is set to `absolute`, the element moves to the new values of `top` and `left`; if its `position` is set to `relative`, it moves from its original position by distances given by the new values of `top` and `left`.

In the next example, an image is absolutely positioned in the display. The document includes two text boxes, labeled `x-coordinate` and `y-coordinate`, respectively. The user can enter new values for the `left` and `top` properties of the image in these boxes. When the *Move It* button is pressed, the values of the `left` and `top` properties of the image are changed to the given values, and the element is moved to its new position.

A JavaScript function, stored in a separate file, is used to change the values of `left` and `top` in this example. Although it is not necessary here, the id of the element to be moved is sent to the function that does the moving, just to illustrate that the function could be used on any number of different elements. The values of the two text boxes are also sent to the function as parameters. The actual parameter values are the DOM addresses of the text boxes, with the `value` attribute attached, which provides the complete DOM addresses of the text box values. Notice that `style` is attached to the DOM address of the image to be moved because `top` and `left` are style properties. Because the input `top` and `left` values from the text boxes are just string representations of numbers, but

the `top` and `left` properties must end with some unit abbreviation, the event handler catenates `"px"` to each value before assigning it to the `top` and `left` properties. This document, called `mover.html`, and the associated JavaScript file, `mover.js`, are as follows:

```
<!DOCTYPE html>
<!-- mover.html
     Uses mover.js to move an image within a document
     -->
<html lang = "en">
  <head>
    <title> Moving elements </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "mover.js" >
    </script>
  </head>
  <body>
    <form action = "">
      <p>
        <label>
          x-coordinate:
          <input type = "text"  id = "leftCoord" size = "3" />
        </label>
        <br />
        <label>
          y-coordinate:
          <input type = "text"  id = "topCoord" size = "3" />
        </label>
        <br />
        <input type = "button"  value = "Move it"
               onclick =
                 "moveIt('saturn',
                 document.getElementById('topCoord').value,
                 document.getElementById('leftCoord').value)" />
      </p>
    </form>
    <div id = "saturn"  style = "position: absolute;
         top: 115px; left: 0;">
      <img src = "../images/saturn.png"
           alt = "(Picture of Saturn)" />
    </div>
  </body>
</html>
```

```
// mover.js
//   Illustrates moving an element within a document

// The event handler function to move an element
function moveIt(movee, newTop, newLeft) {
  dom = document.getElementById(movee).style;

// Change the top and left properties to perform the move
//  Note the addition of units to the input values
  dom.top = newTop + "px";
  dom.left = newLeft + "px";
}
```

Figures 6.4 and 6.5 respectively show the initial and new positions of an image in `mover.html`.

Figure 6.4 Display of `mover.html` (before pressing the *Move It* button)

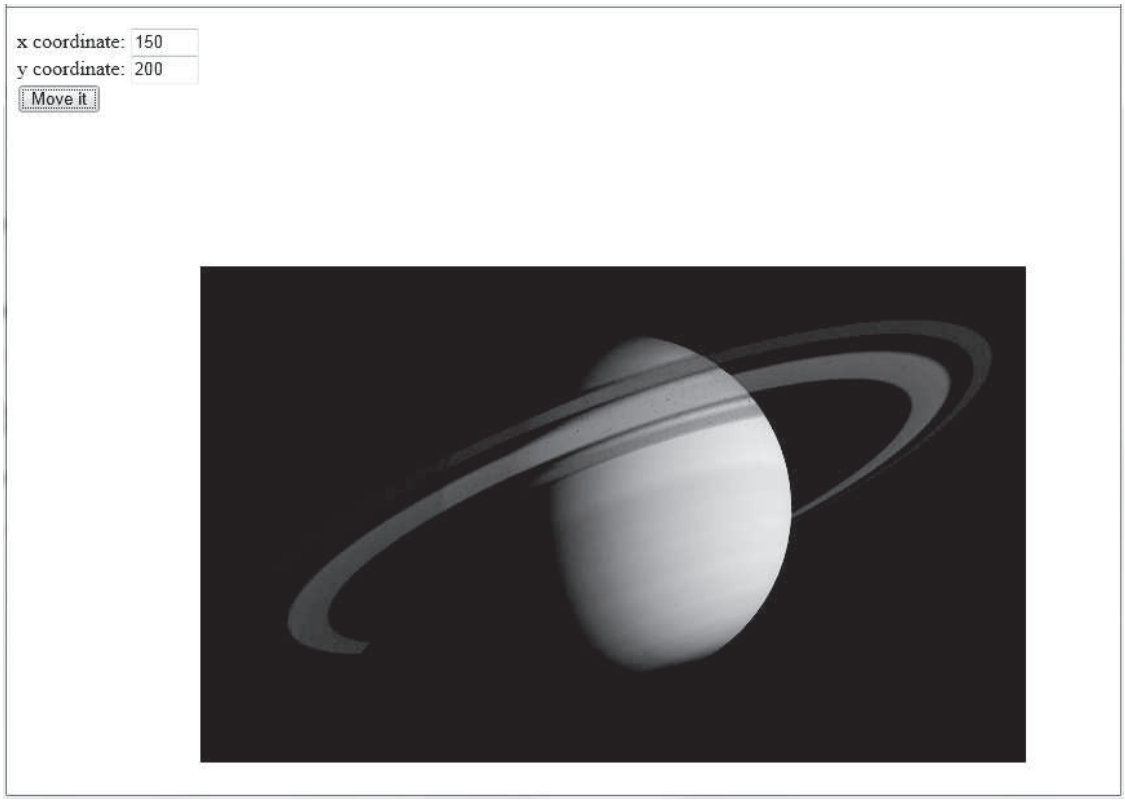


Figure 6.5 Display of `mover.html` (after pressing the *Move It* button)

6.4 Element Visibility

Document elements can be specified to be visible or hidden with the value of their `visibility` property. The two possible values for `visibility` are, quite naturally, `visible` and `hidden`. The appearance or disappearance of an element can be controlled by the user through a widget.

The following example displays an image and allows the user to toggle a button, causing the image to appear and not appear in the document display (once again, the event handler is in a separate file):

```
<!DOCTYPE html>
<!-- showHide.html
      Uses showHide.js
      Illustrates visibility control of elements
     -->
```

```
<html lang = "en">
  <head>
    <title> Visibility control </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "showHide.js" >
    </script>
  </head>
  <body>
    <form action = "">
      <div id = "saturn"  style = "position: relative;
           visibility: visible;">
        <img src = "../images/saturn.png"
             alt = "(Picture of Saturn)" />
      </div>
      <p>
        <br />
        <input type = "button"  value = "Toggle Saturn"
               onclick = "flipImag()" />
      </p>
    </form>
  </body>
</html>
```

```
// showHide.js
//   Illustrates visibility control of elements

// The event handler function to toggle the visibility
//   of the images of Saturn
function flipImag() {
  dom = document.getElementById("saturn").style;
// Flip the visibility adjective to whatever it is not now
 if (dom.visibility == "visible")
   dom.visibility = "hidden";
 else
   dom.visibility = "visible";
}
```

6.5 Changing Colors and Fonts

The background and foreground colors of the document display can be dynamically changed, as can the font properties of the text.

6.5.1 Changing Colors

Dynamic changes to colors are relatively simple. In the next example, the user is presented with two text boxes into which color specifications can be typed—one for the document background color and one for the foreground color. The colors can be specified in any of the three ways that color properties can be given in CSS. A JavaScript function that is called whenever one of the text boxes is changed makes the change in the document's appropriate color property: `backgroundColor` or `color`. The first of the two parameters to the function specifies whether the new color is for the background or foreground; the second specifies the new color. The new color is the `value` property of the text box that was changed by the user.

In this example, the calls to the handler functions are in the HTML text box elements. This approach allows a simple way to reference the element's DOM address. The JavaScript `this` variable is a reference to the object that represents the element in which it is referenced. A reference to such an object is its DOM address. Therefore, in a text element, the value of `this` is the DOM address of the text element. So, in the example, `this.value` is used as an actual parameter to the handler function. Because the call is in an input element, `this.value` is the DOM address of the value of the input element. This document, called `dynColors.html`, and the associated JavaScript file are as follows:

```
<!DOCTYPE html>
<!-- dynColors.html
     Uses dynColors.js
     Illustrates dynamic foreground and background colors
     -->
<html lang = "en">
  <head>
    <title> Dynamic colors </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "dynColors.js" >
    </script>
  </head>
  <body>
    <p style = "font-family: Times; font-style: italic;
                font-size: 2em;" >
      This small page illustrates dynamic setting of the
      foreground and background colors for a document
    </p>
    <form action = "">
      <p>
```

```
        <label>
          Background color:
          <input type = "text"  name = "background" size = "10"
                 onchange = "setColor('background', this.value)" />
        </label>
        <br />
        <label>
          Foreground color:
          <input type = "text"  name = "foreground" size = "10"
                 onchange = "setColor('foreground', this.value)" />
        </label>
        <br />
      </p>
    </form>
  </body>
</html>
```

```
// dynColors.js
//   Illustrates dynamic foreground and background colors
// The event handler function to dynamically set the
// color of background or foreground
function setColor(where, newColor) {
  if (where == "background")
    document.body.style.backgroundColor = newColor;
  else
    document.body.style.color = newColor;
}
```

6.5.2 Changing Fonts

Web users are accustomed to having links in documents change color when the cursor is placed over them. Use of the `mouseover` event to trigger a JavaScript event handler allows us to change any property of any element in a document, including text, when the mouse cursor is placed over it. Thus, the font style and font size, as well as the color and background color of text, can be changed when the cursor is placed over the text. The text can be changed back to its original form when an event handler is triggered with the `mouseout` event.

For CSS attribute names that are single words without hyphens, the associated JavaScript property names are the same as the attribute names. But

when an attribute name includes a hyphen, as in `font-size`, the associated property name must be different (because a property name cannot include a hyphen). The convention is that when an attribute name has a hyphen, the hyphen is deleted and the letter that follows is capitalized in its associated property name. So, the property name associated with the attribute `font-size` is `fontSize`.

In the next example, the only element is a one-line paragraph with an embedded special word. The special word is the content of a span element, so its attributes can be changed. The foreground color for the document is the default black. The word is presented in blue. When the mouse cursor is placed over the word, its color changes to red, its font style changes to italic, and its size changes from `1.1em` to `2em`. When the cursor is moved off the word, it reverts to its original style. Here is this document, called `dynFont.html`:

```
<!DOCTYPE html>
<!-- dynFont.html
     Illustrates dynamic font styles and colors
     -->
<html lang = "en">
  <head>
    <title> Dynamic fonts </title>
    <meta charset = "utf-8" />
    <style type = "text/css">
      .regText {font: 1.1em 'Times New Roman';}
      .wordText {color: blue;}
    </style>
  </head>
  <body>
    <p class = "regText">
      The state of
      <span class = "wordText";
         onmouseover = "this.style.color = 'red';
                        this.style.fontStyle = 'italic';
                        this.style.fontSize = '2em';";
         onmouseout = "this.style.color = 'blue';
                       this.style.fontStyle = 'normal';
                       this.style.fontSize = '1.1em';";>
         Washington
      </span>
      produces many of our nation's apples.
    </p>
  </body>
</html>
```

Notice that the event handlers in this example are embedded in the markup. This is one of those cases where the small amount of JavaScript needed does not justify putting it in a separate file.

Figures 6.6 and 6.7 show browser displays of the `dynFont.html` document with the mouse cursor not over, and then over, the link.

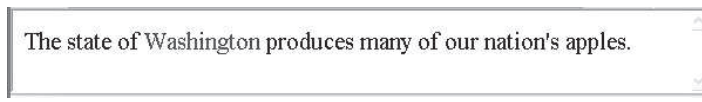


Figure 6.6 Display of `dynFont.html` with the mouse cursor not over the word

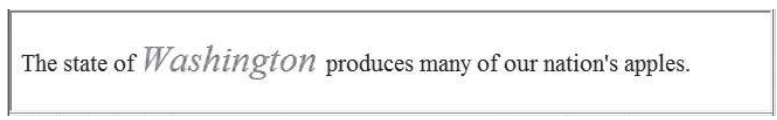


Figure 6.7 Display of `dynFont.html` with the mouse cursor over the word

6.6 Dynamic Content

We have explored the options of dynamically changing the positions of elements, their visibility, colors, background colors, and the styles of text fonts. This section investigates changing the content of HTML elements. The content of an element is accessed through the `value` property of its associated JavaScript object. So, changing the content of an element is not essentially different from changing the style properties of the element. We now develop an example that illustrates one use of dynamic content.

Assistance to a browser user filling out a form can be provided with an associated text area, often called a *help box*. The content of the help box can change, depending on the element over which the mouse cursor is placed. When the cursor is placed over a particular input field, the help box displays advice on how the field is to be filled in. When the cursor is moved away from all input fields, the help box content is changed to indicate that assistance is available.

In the next example, an array of messages to be displayed in the help box is defined in JavaScript. When the mouse cursor is placed over an input field, the `mouseover` event is used to call a handler function that changes the help box content to the appropriate value (the one associated with the input field). The appropriate value is specified with a parameter sent to the handler function. The `mouseout` event is used to trigger the change of the content of the help box back to the *standard* value. Following is the markup document and associated JavaScript file:

```
<!DOCTYPE html>
<!-- dynValue.html
     Illustrates dynamic values
     -->
<html lang = "en">
  <head>
    <title> Dynamic values </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "dynValue.js" >
    </script>
    <style type = "text/css">
      textarea {position: absolute; left: 250px; top: 0px;}
      span {font-style: italic;}
      p {font-weight: bold;}
    </style>
  </head>
  <body>
    <form action = "">
      <p>
        <span>
          Customer information
        </span>
        <br /><br />
        <label>
          Name:
          <input type = "text"  onmouseover = "messages(0)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
        <label>
          Email:
          <input type = "text"  onmouseover = "messages(1)"
                 onmouseout = "messages(4)" />
        </label>
        <br /> <br />
        <span>
          To create an account, provide the following:
        </span>
        <br /> <br />
        <label>
          User ID:
          <input type = "text"  onmouseover = "messages(2)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
```

```
        <label>
          Password:
          <input type = "password"
                 onmouseover = "messages(3)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
      </p>
      <textarea id = "adviceBox"  rows = "3"  cols = "50">
        This box provides advice on filling out the form
        on this page. Put the mouse cursor over any input
        field to get advice.
      </textarea>
      <br /><br />
      <input type = "submit"  value = "Submit" />
      <input type = "reset"  value = "Reset" />
    </form>
  </body>
</html>
```

```
// dynValue.js
//   Illustrates dynamic values

var helpers = ["Your name must be in the form: \n \
 first name, middle initial., last name",
  "Your email address must have the form: \
 user@domain",
  "Your user ID must have at least six characters",
  "Your password must have at least six \
 characters and it must include one digit",
  "This box provides advice on filling out\
 the form on this page. Put the mouse cursor over any \
 input field to get advice"]

// ********************************************************
// The event handler function to change the value of the
// textarea

function messages(adviceNumber) {
  document.getElementById("adviceBox").value =
                                    helpers[adviceNumber];
}
```

Note that the backslash characters that terminate some of the lines of the literal array of messages specify that the string literal is continued on the next line.

Figure 6.8 shows a browser display of the document defined in `dynValue.html`.

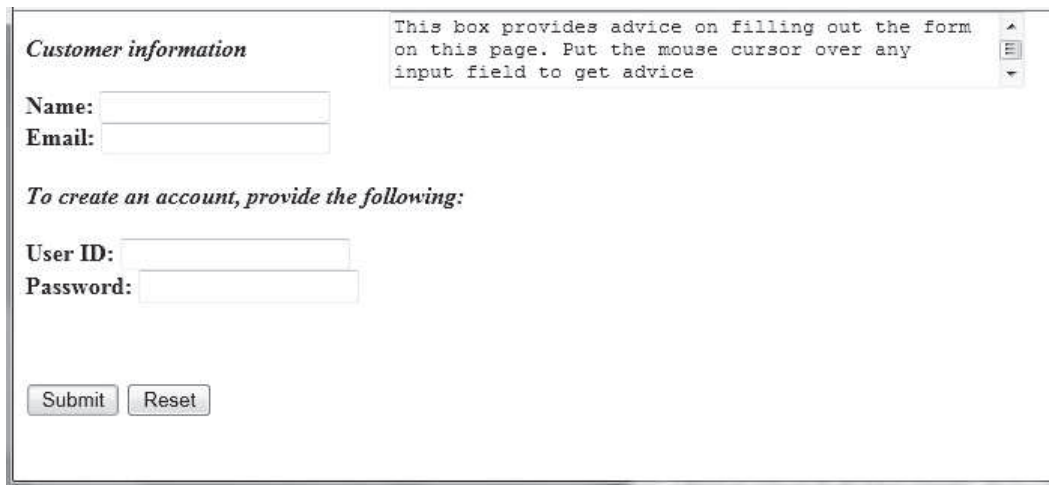


Figure 6.8 Display of `dynValue.html`

6.7 Stacking Elements

The `top` and `left` properties allow the placement of an element anywhere in the two dimensions of the display of a document. Although the display is restricted to two physical dimensions, the effect of the third dimension is possible through the simple concept of stacked elements, such as that used to stack windows in graphical user interfaces. Although multiple elements can occupy the same space in the document, one is considered to be on top and is displayed. The top element hides the parts of the lower elements on which it is superimposed. The placement of elements in this third dimension is controlled by the `z-index` attribute of the element. An element whose `z-index` is greater than that of an element in the same space will be displayed over the other element, effectively hiding the element with the smaller `z-index` value. The JavaScript style property associated with the `z-index` attribute is `zIndex`.

In the next example, three images are placed on the display so that they overlap. In the HTML description of this situation, each image tag includes an `onclick` attribute, which is used to trigger the execution of a JavaScript handler function. First, the function defines DOM addresses for the last top element and the new top element. Then, the function sets the `zIndex` value of the two elements so that the old top element has a value of `0` and the new top element has the value `10`, effectively putting it at the top. The script keeps track of which image is currently on top with the global variable

topp,[3] which is changed every time a new element is moved to the top with the toTop function. Note that the zIndex value, as is the case with other properties, is a string. This document, called stacking.html, and the associated JavaScript file are as follows:

```
<!DOCTYPE html>
<!-- stacking.html
     Uses stacking.js
     Illustrates dynamic stacking of images
     -->
<html lang = "en">
  <head>
    <title> Dynamic stacking of images </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "stacking.js" >
    </script>
    <style type = "text/css">
      .plane1 {position: absolute;
               top: 0; left: 0; z-index: 0;}
      .plane2 {position: absolute;
               top: 50px; left: 50px; z-index: 0;}
      .plane3 {position: absolute;
               top: 100px; left: 100px; z-index: 0;}
    </style>
  </head>
  <body>
    <p>
      <img class = "plane1"  id = "plane1"  height = "300"
           width = "450"  src = "../images/plane1.jpg"
           alt = "(Picture of an airplane)"
           onclick = "toTop('plane1')" />
      <img class = "plane2"  id = "plane2"  height = "300"
           width = "450"  src = "../images/plane2.jpg"
           alt = "(Picture of an airplane)"
           onclick = "toTop('plane2')" />
      <img class = "plane3"  id = "plane3"  height = "300"
           width = "450"  src = "../images/plane3.jpg"
           alt = "(Picture of an airplane)"
           onclick = "toTop('plane3')" />
    </p>
  </body>
</html>
```

3. We use topp, rather than top, because there is a JavaScript keyword top, which is a property of window. Using top confuses Chrome browsers, although it does not affect IE or FX browsers.

```
// stacking.js
//   Illustrates dynamic stacking of images
var topp = "plane3";
// The event handler function to move the given element
// to the top of the display stack
function toTop(newTop) {

// Set the two dom addresses, one for the old top
// element and one for the new top element
  domTop = document.getElementById(topp).style;
  domNew = document.getElementById(newTop).style;

// Set the zIndex properties of the two elements, and
// reset topp to the new top
  domTop.zIndex = "0";
  domNew.zIndex = "10";
  topp = newTop;
}
```

Figures 6.9 through 6.11 show the document described by `stacking.html` in three of its possible configurations.

Figure 6.9 The initial display of `stacking.html`

Figure 6.10 The display `of stacking.html` after clicking the second image

Figure 6.11 The display `of stacking.html` after clicking the bottom image

6.8 Locating the Mouse Cursor

Recall from Chapter 5 that every event that occurs while an HTML document is being displayed creates an event object. This object includes some information about the event. A mouse-click event is an implementation of the `Mouse-Event` interface, which defines two pairs of properties that provide geometric coordinates of the position of the element in the display that created the event. One of these pairs, `clientX` and `clientY`, gives the coordinates of the element relative to the upper-left corner of the browser display window, in pixels. The other pair, `screenX` and `screenY`, also gives coordinates of the element, but relative to the client's computer screen. Obviously, the former pair is usually more useful than the latter.

In the next example, `where.html`, two pairs of text boxes are used to display these four properties every time the mouse button is clicked. The handler is triggered by the `onclick` attribute of the body element. An image is displayed just below the display of the coordinates, but only to make the screen more interesting.

The call to the handler in this example sends `event`, which is a reference to the event object just created in the element, as a parameter. This is a bit of magic, because the event object is implicitly created. In the handler, the formal parameter is used to access the properties of the coordinates. Note that the handling of the event object is not implemented the same way in the popular browsers. The FX browsers send it as a parameter to event handlers, whereas IE and Chrome browsers make it available as a global property. The code in `where.html` works for both these approaches by sending the event object in the call to the handler. It is available in the call with IE and Chrome browsers because it is visible there as a global variable. Of course, for these browsers, it need not be sent at all. The `where.html` document and its associated JavaScript file are as follows:

```
<!DOCTYPE html>
<!-- where.html
     Uses where.js
     Illustrates x- and y-coordinates of the mouse cursor
     -->
<html lang = "en">
  <head>
    <title> Where is the cursor? </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "where.js" >
    </script>
  </head>
```

```
  <body onclick = "findIt(event)">
    <form action = "">
      <p>
        Within the client area: <br />
        x:
        <input type = "text"  id = "xcoor1"  size = "4" />
        y:
        <input type = "text"  id = "ycoor1"  size = "4" />
        <br /><br />
        Relative to the origin of the screen coordinate system:
        <br />
        x:
        <input type = "text"  id = "xcoor2"  size = "4" />
        y:
        <input type = "text"  id = "ycoor2"  size = "4" />
      </p>
    </form>
    <p>
      <img src = "..//images/plane1.jpg"  height = "300"  alt =
                 "(Picture of an airplane)"  width = "450" />
    </p>
  </body>
</html>
```

```
// where.js
//   Show the coordinates of the mouse cursor position
//   in an image and anywhere on the screen when the mouse
//   is clicked


// The event handler function to get and display the
//  coordinates of the cursor, both in an element and
//  on the screen
function findIt(evt) {
  document.getElementById("xcoor1").value = evt.clientX;
  document.getElementById("ycoor1").value = evt.clientY;
  document.getElementById("xcoor2").value = evt.screenX;
  document.getElementById("ycoor2").value = evt.screenY;
}
```

Figure 6.12 shows a browser display of `where.html`.

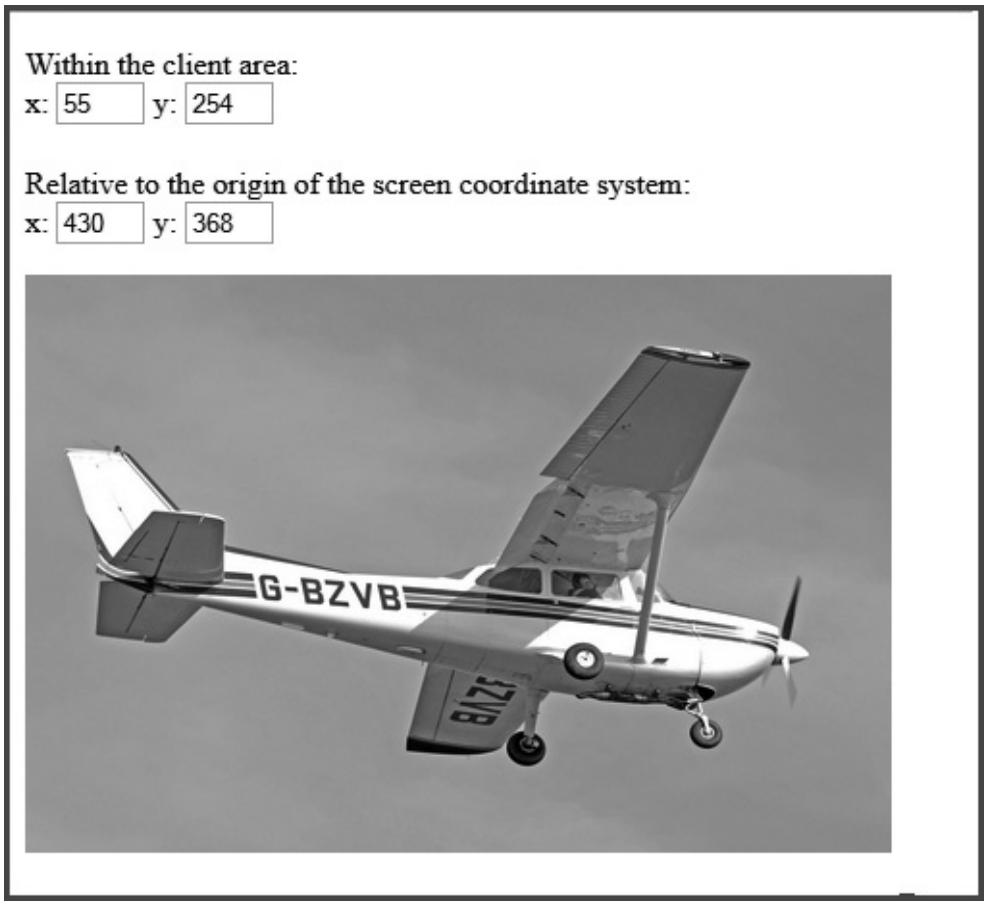


Figure 6.12 Display of `where.html` (the cursor was in the tail section of the plane)

One interesting note about the preceding cursor-finding example is that, with IE and Chrome browsers, the mouse clicks are ignored if the mouse cursor is below the last element on the display. The FX browsers always respond the same way, regardless of where the cursor is on the display.

6.9 Reacting to a Mouse Click

The next example is another one related to reacting to mouse clicks. In this case, the `mousedown` and `mouseup` events are used, respectively, to show and hide the message `"Please don't click here!"` on the display under the mouse cursor whenever the mouse button is clicked, regardless of where the cursor is at the time. The offsets (`-130` for `left` and `-25` for `top`) modify

the actual cursor position so that the message is approximately centered over it. Here is the document and its associated JavaScript file:

```
<!DOCTYPE html>
<!-- anywhere.html
     Uses anywhere.js
     Display a message when the mouse button is pressed,
     no matter where it is on the screen
     -->
<html lang = "en">
  <head>
    <title> Sense events anywhere </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "anywhere.js" >
    </script>
  </head>
  <body onmousedown = "displayIt(event);"
        onmouseup = "hideIt();">
    <p>
      <span id= "message"
            style = "color: red; visibility: hidden;
                     position: relative;
                     font-size: 1.7em; font-style: italic;
                     font-weight: bold;">
         Please don't click here!
      </span>
      <br /><br /><br /><br /><br /><br /><br /><br />
      <br /><br /><br /><br /><br /><br /><br /><br />
    </p>
  </body>
    </html>
```

```
// anywhere.js
//   Display a message when the mouse button is pressed,
//   no matter where it is on the screen

// The event handler function to display the message
function displayIt(evt) {
  var dom = document.getElementById("message");
  dom.style.left = (evt.clientX - 130) + "px";
  dom.style.top = (evt.clientY - 25) + "px";
  dom.style.visibility = "visible";
}
```

```
// ***************************************************
// The event handler function to hide the message
function hideIt() {
  document.getElementById("message").style.visibility =
      "hidden";
}
```

As was the case with `where.html`, with IE and Chrome browsers, the only clicks that cause the text to be displayed are those that occur in the area of the display defined by the `br` elements. With FX browsers, a click anywhere on the screen works.

6.10 Slow Movement of Elements

So far, only element movements that happen instantly have been considered. These movements are controlled by changing the `top` and `left` properties of the element to be moved. The only way to move an element slowly is to move it by small amounts many times, with the moves separated by small amounts of time. JavaScript has two `Window` methods that are capable of this task: `setTimeout` and `setInterval`.

The `setTimeout` method takes two parameters: a string of JavaScript code to be executed and a number of milliseconds of delay before executing the given code. For example, the call

```
setTimeout("mover()", 20);
```

causes a 20-millisecond delay, after which the function `mover` is called.

The `setInterval` method has two forms. One form takes two parameters, exactly as does `setTimeout`. It executes the given code repeatedly, using the second parameter as the interval, in milliseconds, between executions. The second form of `setInterval` takes a variable number of parameters. The first parameter is the name of a function to be called, the second is the interval in milliseconds between the calls to the function, and the remaining parameters are used as actual parameters to the function being called.

The example presented here, `moveText.html`, moves a string of text from one position (100, 100) to a new position (300, 300). The move is accomplished by using `setTimeout` to call a mover function every millisecond until the final position (300, 300) is reached. The initial position of the text is set in the span element that specifies the text. The `onload` attribute of the body element is used to call a function, `initText`, to initialize the x- and y-coordinates of the initial position with the `left` and `top` properties of the element and call the mover function.

The mover function, named `moveText`, takes the current coordinates of the text as parameters, moves them one pixel toward the final position, and then, using `setTimeout`, calls itself with the new coordinates. The recomputation of

the coordinates is complicated by the fact that we want the code to work regardless of the direction of the move (even though in our example the move is always down and to the right).

One consideration with this script is that the properties of the coordinates are stored as strings with units attached. For example, if the initial position of an element is (100, 100), its `left` and `top` property values both have the string value `"100px"`. To change the properties arithmetically, they must be numbers. Therefore, the property values are converted to strings with just numeric digit characters in the `initText` function by stripping the nondigit unit parts. This conversion allows them to be coerced to numbers when they are used as operands in arithmetic expressions. Before the `left` and `top` properties are set to the new coordinates, the units abbreviation (in this case, `"px"`) is catenated back onto the coordinates.

It is interesting that, in this example, placing the event handler in a separate file avoids a problem that would occur if the JavaScript were embedded in the markup. The problem is the use of HTML comments to hide JavaScript and having possible parts of HTML comments embedded in the JavaScript. For example, if the JavaScript statement `x--;` is embedded in an HTML comment, the validator complains that the `--` in the statement is an invalid comment declaration.[4]

In the code file, `moveText.js`, note the complexity of the call to the `moveText` function in the call to `setTimeout`. This level of complexity is required because the call to `moveText` must be built from static strings with the values of the variables `x` and `y` catenated in.

The `moveText.html` document and the associated JavaScript file, `moveText.js`, are as follows:

```
<!DOCTYPE html>
<!-- moveText.html
     Uses moveText.js
     Illustrates a moving text element
     -->
<html lang = "en">
  <head>
    <title> Moving text </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"
            src = "moveText.js">
    </script>
  </head>
<!-- Call the initializing function on load, giving the
      destination coordinates for the text to be moved
      -->
  <body onload = "initText()">
```

4. In the JavaScript code of our example, the statement `x--` is used to move the x-coordinate of the text being moved.

```
<!-- The text to be moved, including its initial position -->
    <p>
      <span id = 'theText' style =
                "position: absolute; left: 100px; top: 100px;
                 font: bold 1.7em 'Times Roman';
                 color: blue;"> Jump in the lake!
      </span>
    </p>
  </body>
</html>
```

```
//*************************************************************
// This is moveText.js - used with moveText.html
   var dom, x, y, finalx = 300, finaly = 300;

// ********************************************** //
// A function to initialize the x- and y-coordinates
//  of the current position of the text to be moved
//  and then call the mover function
   function initText() {
      dom = document.getElementById('theText').style;

   /* Get the current position of the text */
      var x = dom.left;
      var y = dom.top;

   /* Convert the string values of left and top to
      numbers by stripping off the units */
      x = x.match(/\d+/);
      y = y.match(/\d+/);
   /* Call the function that moves it */
      moveText(x, y);
   } /*** end of function initText */

// ********************************************** //
// A function to move the text from its original
//  position to (finalx, finaly)
   function moveText(x, y) {

   /* If the x-coordinates are not equal, move
       x toward finalx */
      if (x != finalx)
         if (x > finalx) x--;
         else if (x < finalx) x++;
```

```
/* If the y-coordinates are not equal, move
    y toward finaly */
   if (y != finaly)
      if (y > finaly) y--;
      else if (y < finaly) y++;

/* As long as the text is not at the destination,
    call the mover with the current position */
   if ((x != finalx) || (y != finaly)) {

/* Put the units back on the coordinates before
   assigning them to the properties to cause the
   move */
      dom.left = x + "px";
      dom.top = y + "px";

/* Recursive call, after a 1-millisecond delay */
      setTimeout("moveText(" + x + "," + y + ")", 1);
   }

 } /*** end of function moveText */
```

The speed of the animation in `moveText.html` varies considerably among browsers. On our system, it took about two seconds with C12, about three seconds with FX3, and more than five seconds with IE9.

6.11 Dragging and Dropping Elements

One of the more powerful effects of event handling is allowing the user to drag and drop elements around the display screen. The `mouseup`, `mousedown`, and `mousemove` events can be used to implement this capability. Changing the `top` and `left` properties of an element, as seen earlier in the chapter, causes the element to move. To illustrate drag and drop, we develop an HTML document and a JavaScript file that creates a magnetic poetry system, which shows two static lines of a poem and allows the user to create the last two lines from a collection of movable words.

This example uses both the DOM 0 and the DOM 2 event models. The DOM 0 model is used for the call to the handler for the `mousedown` event. The rest of the process is designed with the DOM 2 model. The `mousedown` event handler, `grabber`, takes the `Event` object as its parameter. It gets the element to be moved from the `currentTarget` property of the `Event` object and puts it in a global variable so that it is available to the other handlers. Then it determines

the coordinates of the current position of the element to be moved and computes the difference between them and the corresponding coordinates of the position of the mouse cursor. These two differences, which are used by the handler for `mousemove` to actually move the element, are also placed in global variables. The `grabber` handler also registers the event handlers for `mousemove` and `mouseup`. These two handlers are named `mover` and `dropper`, respectively. The `dropper` handler disconnects mouse movements from the element-moving process by unregistering the handlers `mover` and `dropper`. The following is the document we have just described, called `dragNDrop.html`. Following it is the associated JavaScript file.

```
<!DOCTYPE html>
<!-- dragNDrop.html
     An example to illustrate the DOM 2 Event model
     Allows the user to drag and drop words to complete
     a short poem.
     Does not work with IE browsers before IE9
     -->
<html lang = "en">
  <head>
    <title> Drag and drop </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "dragNdrop.js" >
    </script>
  </head>
  <body style = "font-size: 20;">
    <p>
      Roses are red <br />
      Violets are blue <br />

      <span style = "position: absolute; top: 200px; left: 0px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> candy </span>
      <span style = "position: absolute; top: 200px; left: 75px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> cats </span>
      <span style = "position: absolute; top: 200px; left: 150px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> cows </span>
      <span style = "position: absolute; top: 200px; left: 225px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> glue </span>
      <span style = "position: absolute; top: 200px; left: 300px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> is </span>
```

```
        <span style = "position: absolute; top: 200px; left: 375px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> is </span>
        <span style = "position: absolute; top: 200px; left: 450px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> meow </span>
        <span style = "position: absolute; top: 250px; left: 0px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> mine </span>
        <span style = "position: absolute; top: 250px; left: 75px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> moo </span>
        <span style = "position: absolute; top: 250px; left: 150px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> new </span>
        <span style = "position: absolute; top: 250px; left: 225px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> old </span>
        <span style = "position: absolute; top: 250px; left: 300px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> say </span>
        <span style = "position: absolute; top: 250px; left: 375px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> say </span>
        <span style = "position: absolute; top: 250px; left: 450px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> so </span>
        <span style = "position: absolute; top: 300px; left: 0px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> sticky </span>
        <span style = "position: absolute; top: 300px; left: 75px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> sweet </span>
        <span style = "position: absolute; top: 300px; left: 150px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> syrup </span>
        <span style = "position: absolute; top: 300px; left: 225px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> too </span>
        <span style = "position: absolute; top: 300px; left: 300px;
                       background-color: lightgrey;"
              onmousedown = "grabber(event);"> yours </span>
    </p>
  </body>
</html>
```

```
// dragNDrop.js
//   An example to illustrate the DOM 2 Event model
//   Allows the user to drag and drop words to complete
//   a short poem.
//   Does not work with IE browsers before IE9

// Define variables for the values computed by
// the grabber event handler but needed by mover
// event handler
      var diffX, diffY, theElement;

// *******************************************************
// The event handler function for grabbing the word
function grabber(event) {

// Set the global variable for the element to be moved
  theElement = event.currentTarget;

// Determine the position of the word to be grabbed,
// first removing the units from left and top
  var posX = parseInt(theElement.style.left);
  var posY = parseInt(theElement.style.top);

// Compute the difference between where it is and
// where the mouse click occurred
  diffX = event.clientX - posX;
  diffY = event.clientY - posY;

// Now register the event handlers for moving and
// dropping the word
  document.addEventListener("mousemove", mover, true);
  document.addEventListener("mouseup", dropper, true);

// Stop propagation of the event and stop any default
// browser action
  event.stopPropagation();
  event.preventDefault();

}  //** end of grabber

// *******************************************************
// The event handler function for moving the word
function mover(event) {
```

```
// Compute the new position, add the units, and move the word
  theElement.style.left = (event.clientX - diffX) + "px";
  theElement.style.top = (event.clientY - diffY) + "px";

// Prevent propagation of the event
  event.stopPropagation();
}  //** end of mover

// *******************************************************
// The event handler function for dropping the word
function dropper(event) {

// Unregister the event handlers for mouseup and mousemove
  document.removeEventListener("mouseup", dropper, true);
  document.removeEventListener("mousemove", mover, true);

// Prevent propagation of the event
  event.stopPropagation();
}  //** end of dropper
```

Figure 6.13 shows a browser display of `dragNDrop.html`, after some interaction.

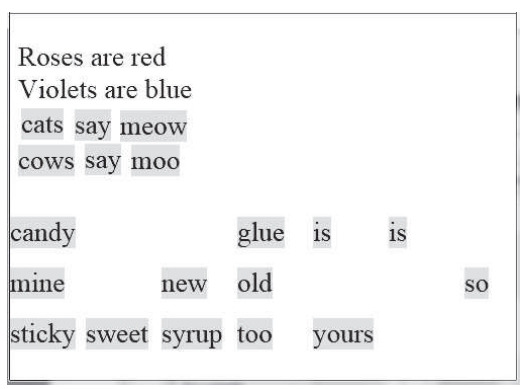


Figure 6.13 Display of `dragNDrop.html`

Note that the drag-and-drop process can be written with the DOM 0 event model. However, it can be made portable only by having the script detect which browser is being used and using different codes for the different browsers. We have chosen to write it with the DOM 2 event model rather than deal with that untidy situation.

Summary

The CSS-P standard enables us initially to place HTML elements wherever we want in a document and move them later. Elements can be positioned at any given location in the display of the document if their `position` property is set to `absolute` or `relative`. Absolute positioning uses the `left` and `top` properties of an element to place the element at a position relative to the upper-left corner of the display of the document. Relative positioning is used to place an element at a specified offset from the `top` and `left` coordinates of where it would have gone with the default static positioning. Relative positioning also allows an element to be moved later. Static positioning, which is the default, disallows both specific initial placement and dynamic moving of the element.

An HTML element can be made to disappear and reappear by changing its `visibility` property.

The color of the background of a document is stored in its `backgroundColor` property; the color of an element is stored in its `color` property. Both of these can be dynamically changed. The font, font size, and font style of text also can be changed.

The content of an element can be changed by changing its `value` property. An element in a document can be set to appear to be in front of other elements, and this top-to-bottom stacking order can be dynamically changed. The coordinates of the mouse cursor can be found by means of properties of the event object every time a mouse button is pressed. An element can be animated, at least in a crude way, by changing its `top` and `left` properties repeatedly by small amounts. Such an operation can be controlled by the `Window` method `setTimeout`. Event handlers for the mouse events can be written to allow the user to drag and drop elements anywhere on the display screen.

Review Questions

6.1 Define a dynamic HTML document.

6.2 If you know the id of an HTML element, how can you get the DOM address of that element in JavaScript?

6.3 If you have a variable that has the id of an HTML element, how can you get the DOM address of that element in JavaScript?

6.4 In what additional way can you obtain the DOM addresses of individual radio buttons and checkboxes?

6.5 What is CSS-P?

6.6 Describe all the differences between the three possible values of the `position` property.

6.7 What are the standard values for the `visibility` property?

6.8 What properties control the foreground and background colors of a document?

6.9 What events can be used to change a font when the mouse cursor is moved over and away from an element?

6.10 What property has the content of an element?

6.11 What JavaScript variable is associated with the `z-index` property?

6.12 To move an element to the top of the display, do you set its `z-index` property to a large number or a small number?

6.13 What exactly is stored in the `clientX` and `clientY` properties after a mouse click?

6.14 What exactly is stored in the `screenX` and `screenY` properties after a mouse click?

6.15 Describe the parameters and actions of the `setTimeout` function.

Exercises

Write, test, validate, and debug (if necessary) markup documents and JavaScript files for the following:

6.1 The document must have a paragraph of at least 10 lines of text that describe you. This paragraph must be centered on the page and have space for 20 characters per line only. A light-gray image of yourself must be superimposed over the center of the text as a nested element.

6.2 Modify the document described in Exercise 6.1 to add four buttons labeled, respectively, *Northwest*, *Northeast*, *Southwest*, and *Southeast*. When they're pressed, the buttons must move your image to the specified corner of the text. Initially, your image must appear in the northwest (upper-left) corner of the text.

6.3 Modify the document described in Exercise 6.2 to make the buttons toggle their respective copies of your image on and off so that, at any time, the document may include none, one, two, three, or four copies of your image. The initial document should have no images shown.

6.4 The document must have a paragraph of text that describes your home. Choose at least three different phrases (three to six words each) of this paragraph, and make them change font, font style, color, and font size when the mouse cursor is placed over them. Each of the different phrases must change to a different font, font style, color, and font size.

6.5 The document must display an image and three buttons. The buttons should be labeled simply *1*, *2*, and *3*. When pressed, each button should change the content of the image to that of a different image.

6.6 The document must contain four short paragraphs of text, stacked on top of each other, with only enough of each showing so that the mouse

cursor can always be placed over some part of them. When the cursor is placed over the exposed part of any paragraph, it should rise to the top to become completely visible.

6.7 Modify the document of Exercise 6.6 so that when a paragraph is moved from the top stacking position, it returns to its original position rather than to the bottom.

6.8 The document must have a small image of yourself, which must appear at the position of the mouse cursor when the mouse button is clicked, regardless of the position of the cursor at the time.

6.9 The document must contain the statement "Save time with TIME-SAVER 2.2," which continuously moves back and forth across the top of the display.

6.10 Modify the document of Exercise 6.9 to make the statement change color between red and blue every fifth step of its movement (assuming that each move is one pixel long).

6.11 Modify the `mover` example in Section 6.10 to input the starting and ending positions of the element to be moved.