CHAPTER 1

Web Essentials
Clients, Servers, and Communication

The essential elements of the World Wide Web are the web browsers used to surf the Web, the server systems used to supply information to these browsers, and the computer networks supporting browser-server communication. This chapter will provide an overview of all of these elements. We'll begin by considering communication, with a focus on the Internet and some of its key communication protocols, especially the Hypertext Transport Protocol used for the bulk of web communication. The chapter also reviews features common to modern web browsers and introduces web servers, the software applications that provide web pages to browsers.

1.1 The Internet

"So, you're into computers. Maybe you can answer a question I've had for a while: I hear people talk about the Internet, and I'm not sure exactly what it is, or where it came from. Can you tell me?"

You may have already been asked a question like this. If not, if you work with computers long enough, you'll probably hear it at least once in your career, and more likely several times. At this point in your career, you may even be curious about the Internet yourself: you use it a lot, but what exactly is it?

The Internet traces its roots to a project of the U.S. Department of Defense's then-named Advanced Research Projects Agency, or ARPA. The ARPANET project was intended to support DoD research on computer networking. As this project began in the late 1960s, there had been only a few small experimental networks providing communication between geographically dispersed computers from different manufacturers running different operating systems. The purpose of ARPANET was to create a larger such network, both in order to electronically connect DoD-sponsored researchers and in order to experiment with and develop tools for heterogeneous computer networking.

The ARPANET computer network was launched in 1969 and by year's end consisted of four computers at four sites running four different operating systems. ARPANET grew steadily, but because it was restricted to DoD-funded organizations and was a research project, it was never large. By 1983, when many ARPANET nodes were split off to form a separate network called MILNET, there were only 113 nodes in the entire network, and these were primarily at universities and other organizations involved in DoD-sponsored research.

Despite the relatively small number of machines actually on the ARPANET, the benefits of networking were becoming known to a wide audience. For example, e-mail was available on ARPANET beginning in 1972, and it soon became an extremely popular

application for those who had ARPANET access. It wasn't long before other networks were being built, both internationally and regionally within the United States. The regional U.S. networks were often cooperative efforts between universities. As one example, SURAnet (Southeastern University Research Association Network) was organized by the University of Maryland beginning in 1982 and eventually included essentially all of the major universities and research institutions in the southeastern United States. Another of these networks, CSNET (Computer Science Network), was partially funded by the U.S. National Science Foundation (NSF) to aid scientists at universities without ARPANET access, laying the groundwork for future network developments that we'll say more about in a moment.

While these other networks were springing up, the ARPANET project continued to fund research on networking. Several of the most widely used Internet protocols—including the File Transfer Protocol (FTP) and Simple Mail Transfer Protocol (SMTP), which underlie many of the Internet's file transfer and e-mail operations, respectively—were initially developed under ARPANET. But perhaps most crucial to the emergence of the Internet as we know it was the development of the TCP/IP (Transmission Control Protocol/Internet Protocol) communication protocol. TCP/IP was designed to be used for host-to-host communication both within *local area networks* (that is, networks of computers that are typically in close proximity to one another, such as within a building) and between networks. ARPANET switched from using an earlier protocol to TCP/IP during 1982. At around the same time, an ARPA Internet was created, allowing computers on some outside networks such as CSNET to communicate via TCP/IP with computers on the ARPANET.

A "connection" from CSNET to the ARPA Internet often meant that a modem connection was made from one computer to another for the purpose of sending along an e-mail message. This form of communication was asynchronous. That is, the e-mail might be delayed some time before it was actually delivered, which precluded interactive communication of any type. Furthermore, each institution connecting to CSNET was largely on its own in determining how it was going to connect to the network. At first, many institutions connected through the so-called PhoneNet (modem) approach for passing e-mail messages. This generally involved long distance calls, and the expense of these calls could be a problem. Other options, such as leasing telephone lines for dedicated use, could be even more expensive. It was obvious to everyone that the CSNET institutions were still not enjoying all the potential benefits of the ARPA Internet.

Beginning in 1985, the NSF began work on a new network based on TCP/IP, called NSFNET. One of the primary goals of this network was to connect the NSF's new regional supercomputing centers. But it was also decided that regional networks should be able to connect to NSFNET, so that the NSFNET would provide a *backbone* through which other networks could interconnect synchronously. Figure 1.1 shows the geographic distribution of the six supercomputer centers connected by the early NSFNET backbone. Regional networks connecting to the backbone included SURAnet as well as NYSERNet (with primary connections through the Ithaca center), JvNCnet (with primary connection through the Princeton center) and SDSCnet (with primary connection through the San Diego center). In addition, many universities and other organizations connected to the NSFNET backbone either directly or through agreements with other institutions that had NSFNET access, either directly or indirectly.

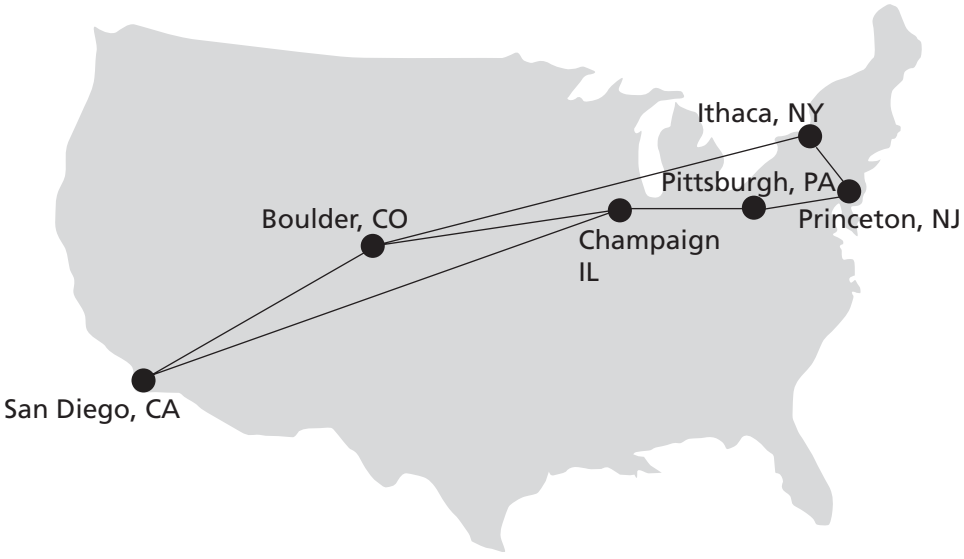

FIGURE 1.1 Geographic distribution of and connections between nodes on the early NSFNET backbone.

The original backbone operated at only 56 kbit/s, the maximum speed of a home dial-up line today. But at the time the primary network traffic was still textual, so this was a reasonable starting point. Once operational, the number of machines connected to NSFNET grew quickly, in part because the NSF directly or indirectly provided significant support—both technically and with monetary grants—to educational and research organizations that wished to connect. The backbone rate was upgraded to 1.5 Mbit/s (T1) in 1988 and then to 45 Mbit/s (T3) in 1991. Furthermore, the backbone was expanded to directly include several research networks in addition to the supercomputer centers, making it that much easier for sites near these research networks to connect to the NSFNET. In 1988, networks in Canada and France were connected to NSFNET; in each succeeding year for the remaining seven years of NSFNET's existence, networks from 10 or so new countries were added per year.

NSFNET quickly supplanted ARPANET, which was officially decommissioned in 1990. At this point, NSFNET was at the center of the *Internet*, that is, the collection of computer networks connected via the public backbone and communicating across networks using TCP/IP. This same year, commercial Internet dial-up access was first offered. But the NSFNET terms of usage stipulated that purely commercial traffic was not to be carried over the backbone: the purpose of the Internet was still, in the eyes of the NSF, research and education.

Increasingly, though, it became clear that there could be significant benefits to allowing commercial traffic on the Internet as well. One of the arguments for allowing commercial traffic was economic: commercial traffic would increase network usage, leading to reduced unit costs through economies of scale. This in turn would provide a less expensive network for research and educational purposes. Whatever the motivation, the restriction on commercial traffic was rescinded in 1991, spurring further growth of the Internet and laying the

groundwork for the metamorphosis of the Internet from a tool used primarily by scientists at research institutions to the conduit for information, entertainment, and commerce that we know today. This also led fairly quickly to the NSF being able to leave its role as the operator of the Internet backbone in the United States. Those responsibilities were assumed by private telecommunication firms in 1995. These firms are paid by other firms, such as some of the larger Internet service providers (ISPs), who connect directly with the Internet backbone. These ISPs, in turn, are paid by their users, which may include smaller ISPs as well as end users.

In summary, the Internet is the collection of computers that can communicate with one another using TCP/IP over an open, global communications network. Before describing how the World Wide Web is related to the Internet, we'll take a closer look at several of the key Internet protocols. This will be helpful in understanding the place of the Web within the wider Internet.

1.2 Basic Internet Protocols

Before covering specific protocols, it may be helpful to explain exactly what the term "protocol" means in the context of networked communication. A computer *communication protocol* is a detailed specification of how communication between two computers will be carried out in order to serve some purpose. For example, as we will learn, the Internet Protocol specifies both the high-level behavior of software implementing the protocol and the low-level details such as the specific fields of information that will be contained in a communication message, the order in which these fields will appear, the number of bits in each field, and how these bits should be interpreted. We are primarily interested in a high-level view of general-purpose Internet protocols in this section; we'll look at a key Web protocol, HTTP, in more detail in the next section.

1.2.1 TCP/IP

Since TCP/IP is fundamental to the definition of the Internet, it's natural to begin our study of Internet protocols with these protocols. Yes, I said protocols (plural), because although so far I have treated TCP/IP as if it were a single protocol, TCP and IP are actually two different protocols. The reason that they are often treated as one is that the bulk of the services we associate with the Internet—e-mail, Web browsing, file downloads, accessing remote databases—are built on top of both the TCP and IP protocols. But in reality, only one of these protocols—IP, the Internet Protocol—is fundamental to the definition of the Internet. So we'll begin our study of Internet protocols with IP.

A key element of IP is the *IP address*, which is simply a 32-bit number. At any given moment, each device on the Internet has one or more IP addresses associated with it (although the device associated with a given address may change over time). IP addresses are normally written as a sequence of four decimal numbers separated by periods (called "dots"), as in 192.0.34.166. Each decimal number represents one byte of the IP address.

The function of IP software is to transfer data from one computer (the *source*) to another computer (the *destination*). When an application on the source computer wants to send information to a destination, the application calls IP software on the source machine

and provides it with data to be transferred along with an IP address for each of the source and destination computers. The IP software running on the source creates a *packet*, which is a sequence of bits representing the data to be transferred along with the source and destination IP addresses and some other header information, such as the length of the data. If the destination computer is on the same local network as the source, then the IP software will send the packet to the destination directly via this network. If the destination is on another network, the IP software will send the packet to a *gateway*, which is a device that is connected to the source computer's network as well as to at least one other network. The gateway will select a computer on one of the other networks to which it is attached and send the packet on to that computer. This process will continue, with the packet going through perhaps a dozen or more *hops*, until the packet reaches the destination computer. IP software on that computer will receive the packet and pass its data up to an application that is waiting for the data.

For example, returning to the Internet as it existed in the mid-1980s, suppose that a computer in the SURAnet network (say, at the University of Delaware) was a packet source and that a computer in a network directly connected to the NSFNET backbone at San Diego (say, at the San Diego Supercomputer Center) was the destination. The IP packet would first go through the Delaware local computer network to a gateway device connecting the Delaware network to SURAnet. The gateway device would then send the packet on to another SURAnet gateway device (how this gateway is chosen is discussed later in this subsection) until it reached a gateway on the NSFNET backbone at Ithaca (the primary SURAnet connection to the NSFNET backbone). As there was no direct connection from Ithaca to San Diego in the NSFNET at the time (Figure 1.1), the packet would need to go through at least one other gateway on the NSFNET backbone before reaching the San Diego node. From there, it would be passed to the San Diego Supercomputer Center local network, and from there on to the destination machine.

The sequence of computers that a packet travels through from source to destination is known as its *route*. How does each computer choose the next computer in the route for a packet? A separate protocol (the current standard is BGP-4, the Border Gateway Protocol) is used to pass network connectivity information between gateways so that each can choose a good next hop for each packet it receives.

IP software also adds some error detection information (a *checksum*) to each packet it creates, so that if a packet is corrupted during transmission, this can usually be detected by the recipient. The IP standard calls for IP software to simply discard any corrupted packets. Thus, IP-based communication is unreliable: packets can be lost. Obviously, IP alone is not a particularly good form of communication for many Internet applications.

TCP, the Transmission Control Protocol, is a higher-level protocol that extends IP to provide additional functionality, including reliable communication based on the concept of a *connection*. A connection is established between TCP software running on two machines by one of the machines (let's call it A) sending a connection-request message via IP to the other (B). That is, the IP message contains a message conforming to the TCP protocol and representing a TCP connection request. If the connection is accepted by B, then B returns a message to A requesting a connection in the other direction. If A responds affirmatively, then the connection is established. Notice that this means that A and B can both send messages to one another at the same time; this is known as *full duplex* communication. When A and

B are both done sending messages to one another (or at least done for the time being), a similar set of three messages is used to close the connection.

Once a connection has been established, TCP provides reliable data transmission by demanding an *acknowledgment* for each packet it sends via IP. Essentially, the software sets a timer after sending each packet. The TCP software on the receiving side sends a packet containing an acknowledgment for every TCP-based packet it receives that passes the checksum test. If the TCP software sending a packet does not receive an acknowledgment packet before its timer expires, then it resends the packet and restarts the timer.

Another important feature that TCP adds to IP is the concept of a *port*. The port concept allows TCP to be used to communicate with many different applications on a machine. For example, a machine connected to the Internet may run a mail server for users on its local network, a file download server, and also a server that allows users to log in to the machine and execute commands from remote locations. As illustrated in Figure 1.2 (which ignores connections and acknowledgments for simplicity), such a server application will make a call to the TCP software on its system to request that any incoming TCP connection requests that specify a certain port number as part of the TCP/IP message be sent to the application. For example, a mail server conforming with SMTP will typically ask TCP to listen for requests to port 25. If at a later time an IP message is received by the machine running the mail server application and that IP message contains a TCP message with port

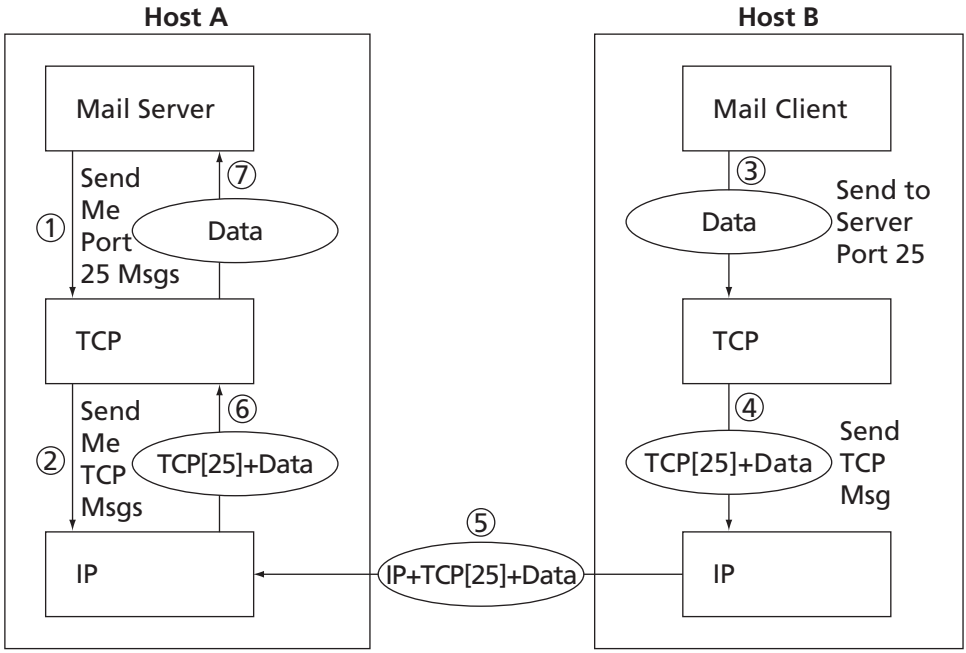

FIGURE 1.2 Simplified view of communication using TCP/IP. Boxes represent software applications on the respective host machines, ovals represent data transmitted between applications, and circled numbers denote the time order of operations. "TCP[25]" represents a TCP header containing 25 as the port number.

25 indicated in its header, then the data contained within the TCP message will be returned to the mail server application. Such an IP message could be generated by a mail client calling on TCP software on another system, as illustrated on the right side of the figure.

Though the connection between port numbers and applications is managed individually by every machine on the Internet, certain broadly useful applications (such as e-mail over SMTP) have had port numbers assigned to them by the Internet Assigned Numbers Authority (IANA) [IANA-PORTS]. These port numbers, in the range 0–1023, can usually be requested only by applications that are run by the system at boot-up or that are run by a user with administrative permissions on the system. Other possible port numbers, from 1024 to 65535, can generally be used by the first application on a system that requests the port.

TCP and IP provide many other functions, such as splitting long messages into shorter ones for transport over the Internet and transparently reassembling them on the receiving side. But this brief overview of TCP/IP covers the essential concepts for our purposes.

1.2.2 UDP, DNS, and Domain Names

UDP (User Datagram Protocol) is an alternative protocol to TCP that also builds on IP. The main feature that UDP adds to IP is the port concept that we have just seen in TCP. However, it does not provide the two-way connection or guaranteed delivery of TCP. Its advantage over TCP is speed for simple tasks. For example, if all you want to do is send a short message to another computer, you're expecting a single short response message, and you can handle resending if you don't receive the response within a reasonable amount of time, then UDP is probably a good alternative to TCP.

One Internet application that is often run using UDP rather than TCP is the Domain Name Service (DNS). While every device on the Internet has an IP address such as 192.0.34.166, humans generally find it easier to refer to machines by names, such as `www.example.org`. DNS provides a mechanism for mapping back and forth between IP addresses and host names. Basically, there are a number of DNS servers on the Internet, each listening through UDP software to a port (port 53 if the server is following the current IANA assignment). When a computer on the Internet needs DNS services—for example, to convert a host name such as `www.example.org` to a corresponding IP address—it uses the UDP software running on its system to send a UDP message to one of these DNS servers, requesting the IP address. If all goes well, this server will then send back a UDP message containing the IP address. Recall that it took three messages just to get a TCP connection set up, so the UDP approach is much more efficient for sporadic DNS queries. (UDP is sometimes referred to as a lightweight communication protocol and TCP as a heavyweight protocol, at least in comparison with UDP. In general, the terms *lightweight* and *heavyweight* in computer science are used to describe alternative software solutions to some problem, with the lightweight solution having less functionality but also less overhead.)

Internet host names consist of a sequence of *labels* separated by dots. The final label in a host name is a *top-level domain*. There are two standard types of top-level domain: generic (such as `.com`, `.edu`, `.org`, and `.biz`) and country-code (such as `.de`, `.il`, and `.mx`). The top-level domain names are assigned by the Internet Corporation for Assigned Names and

Numbers (ICANN), a private nonprofit organization formed to take over technical Internet functions that were originally funded by the U.S. government.

Each top-level domain is divided into subdomains (second-level domains), which may in turn be further divided, and so on. The assignment of second-level domains within each top-level domain is performed (for a fee) by a *registry operator* selected by ICANN. The owner of a second-level domain can then further divide that domain into subdomains, and so on. Ultimately, the subdomains of a domain are individual computers. Such a subdomain, consisting of a local host name followed by a domain name (typically consisting of at least two labels) is sometimes called a *fully qualified domain name* for the computer. For example, `www.example.org` is a fully qualified domain name for a host with local name `www` that belongs to the `example` second-level domain of the `org` top-level domain.

Some user-level tools are available that allow you to query the Internet DNS. For example, on most systems the `nslookup` command can be typed at a command prompt (see *Appendix A* for instructions on obtaining a command prompt on some systems) in order to find the IP address given a fully qualified domain name or vice versa. Typical usage of `nslookup` is illustrated by the following (user input is italicized):

```
C:\>nslookup www.example.org
Server: slave9.dns.stargate.net
Address: 209.166.161.121

Name: www.example.org
Address: 192.0.34.166

C:\>nslookup 192.0.34.166
Server: slave9.dns.stargate.net
Address: 209.166.161.121

Name: www.example.com
Address: 192.0.34.166
```

The first two lines following the command line identify the qualified name and IP address of the DNS server that is providing the domain name information that follows. Also notice that a single IP address can be associated with multiple domain names. In this example, both `www.example.org` and `www.example.com` are associated with the IP address 192.0.34.166. A lookup that specifies an IP address, such as the second lookup in the example, is sometimes referred to as a *reverse lookup*. As shown, even if multiple qualified names are associated with an IP address, only one of the names will be returned by a reverse lookup. This is known as the *canonical name* for the host; all other names are considered *aliases*. The reverse lookup in the example indicates that `www.example.com` is the canonical name for the host with IP address 192.0.34.166.

1.2.3 Higher-Level Protocols

The following analogy may help to relate the computer networking concepts described in Sections 1.2.1 and 1.2.2 with something more familiar: the telephone network. The Internet

is like the physical telephone network: it provides the basic communications infrastructure. UDP is like calling a number and leaving a message rather than actually speaking with the intended recipient. DNS is the Internet version of directory assistance, associating names with numbers. TCP is roughly equivalent to placing a phone call and having the other party answer: you now have a connection and are able to communicate back and forth.

However, in the cases of both TCP and a phone call, different protocols can be used to communicate once a connection has been established. For example, when making a telephone call, the parties must agree on the language(s) that will be used to communicate. Beyond that, there are also conventions that are followed to decide which party will speak first, how the parties will take turns speaking, and so on. Furthermore, different conventions may be used in different contexts: I answer the phone differently at home ("Hello") than I do at work ("Mathematics and Computer Science Department, this is Jeff Jackson"), for example.

Similarly, a variety of *higher-level protocols* are used to communicate once a TCP connection has been established. SMTP and FTP, mentioned earlier, are two examples of widely used higher-level protocols that are used to communicate over TCP connections. SMTP supports transfer of e-mail between different e-mail servers, while FTP is used for transferring files between machines. Another higher-level TCP protocol, Telnet, is used to execute commands typed into one computer on a remote computer. As we will see, Telnet can also be used to communicate directly (via keyboard entries) with some TCP-based applications. As described earlier, which protocol will be used to communicate over a TCP connection is normally determined by the port number used to establish the connection.

The primary TCP-based protocol used for communication between web servers and browsers is called the Hypertext Transport Protocol (HTTP). In some sense, just as IP is a key component in the definition of the Internet, HTTP is a key component in the definition of the World Wide Web. So, before getting into details of HTTP, let's briefly consider what the Web is, and in particular how HTTP figures in its definition.

1.3 The World Wide Web

Public sharing of information has been a part of the Internet since its early days. For example, the Usenet newsgroup service began in 1979 and provided a means of "posting" information that could be read by users on other systems with the appropriate software (the Google Groups™ Usenet discussion forum at `http://www.google.com` provides one of several modern interfaces to Usenet). Large files were (and still are) often shared by running an FTP server application that allowed any user to transfer the files from their origin machine to the user's machine. The first Internet chat software in widespread use, Internet Relay Chat (IRC), provided both public and private chat facilities.

However, as the amount of information publicly available on the Internet grew, the need to locate information also grew. Various technologies for supporting information management and search on the Internet were developed. Some of the more popular information management technologies in the early 1990s were Gopher information servers, which provided a simple hierarchical view of documents; the Wide Area Information System

(WAIS) system for indexing and retrieving information; and the ARCHIE tool for searching online information archives accessible via FTP.

The World Wide Web also was developed in the early 1990s (we'll learn more about its development in the next chapter), and for a while was just one among several Internet information management technologies. To understand why the Web supplanted the other technologies, it will be helpful to know a bit about the mechanics of the Web and other Internet information management technologies. All of these technologies consist of (at least) two types of software: server and client. An Internet-connected computer that wishes to provide information to other Internet systems must run *server* software, and a system that wishes to access the information provided by servers must run *client* software (for the Web, the client software is normally a web browser). The server and client applications communicate over the Internet by following a communication protocol built on top of TCP/IP.

The protocol used by the Web, as just noted, is the Hypertext Transport Protocol, HTTP. As we will learn in the next section, this is a rather generic protocol that for the most part supports a client requesting a document from a server and the server returning the requested document. This generic nature of HTTP gives it the advantage of somewhat more flexibility than is present in the protocols used by WAIS and Gopher.

Perhaps a bigger advantage for the Web is the type of information communicated. Most web pages are written using the Hypertext Markup Language, HTML, which along with HTTP is a fundamental web technology. HTML pages can contain the familiar web links (technically called *hyperlinks*) to other documents on the Web. While certain Gopher pages could also contain links, normal Gopher documents were plain text. WAIS and ARCHIE provided no direct support for links. In addition to hyperlinks, modern versions of HTML also provide extensive page layout facilities, including support for inline graphics, which (as you might guess) has added significantly to the commercial appeal of the Web.

The World Wide Web, then, can be defined in much the same way as the Internet. While the Internet can be thought of as the collection of machines that are globally connected via IP, the World Wide Web can be informally defined as the collection of machines (web servers) on the Internet that provide information via HTTP, and particularly those that provide HTML documents.

Given this overview, we'll now spend some time looking closely at HTTP.

1.3.1 Hypertext Transport Protocol

HTTP is a form of communication protocol, in particular a detailed specification of how web clients and servers should communicate. The basic structure of HTTP communication follows what is known as a *request–response* model. Specifically, the protocol dictates that an HTTP interaction is initiated by a client sending a request message to the server; the server is then expected to generate a response message. The format of the request and response messages is dictated by HTTP. HTTP does not dictate the network protocol to be used to send these messages, but does expect that the request and response are both sent within a TCP-style connection between the client and the server. So most HTTP implementations send these messages using TCP.

Let's relate this to what happens when you browse the Web. Figure 1.3 shows a browser window in which I typed `http://www.example.org` in the Location bar (note that this is technically not a web site address and therefore might not be operational by the time you read this). When I pressed the Enter key after typing this address, the browser created a message conforming to the HTTP protocol, used DNS to obtain an IP address for `www.example.org`, created a TCP connection with the machine at the IP address obtained, sent the HTTP message over this TCP connection, and received back a message containing the information that is shown displayed in the *client area* of the browser (the portion of the browser containing the information received from the web server).

A nice feature of HTTP is that these request and response messages often consist entirely of plain text in a fairly readable form. An HTTP request message consists of a start line followed by a message header and optionally a message body. The start line always consists of printable ASCII characters, and the header normally does as well. What's more, the HTTP response (or at least most of it) is often also a stream of printable characters. So, to see an example of HTTP in action, let's connect to the same web server shown in Figure 1.3 using Telnet. This can be done on most modern systems by entering `telnet` at a command prompt. Specifically, we will Telnet to port 80, the IANA standard port for HTTP web servers, type in an HTTP request message corresponding to the Internet address entered into the browser before, and view the response (the request consists of the three lines beginning with the GET and ending with a blank line, and user input is again italicized):

```
$ telnet www.example.org 80
Trying 192.0.34.166...
Connected to www.example.com (192.0.34.166).
Escape character is '^]'.
GET / HTTP/1.1
Host: www.example.org

HTTP/1.1 200 OK
Date: Thu, 09 Oct 2003 20:30:49 GMT
```

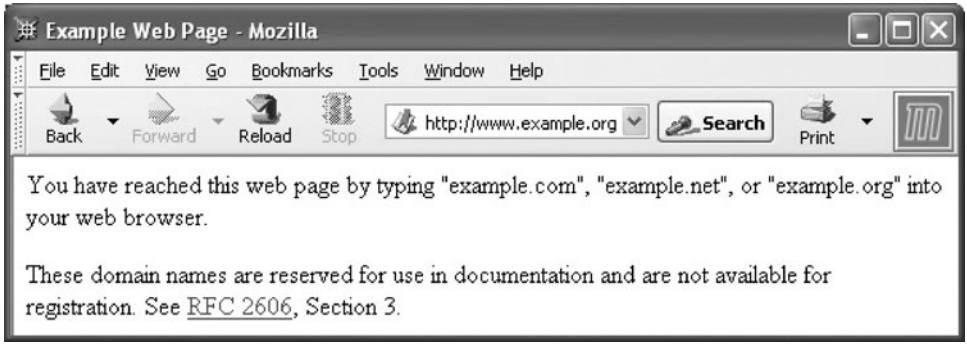

FIGURE 1.3 Web browser displaying information received in an HTTP response message received after the browser sent an HTTP request message to a web server. The content shown is subject to copyright and used by permission of the Internet Assigned Numbers Authority (IANA).

```
Server: Apache/1.3.27 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Example Web Page</TITLE>
</HEAD>
<body>
<p>You have reached this web page by typing &quot;example.com&quot;,
  &quot;example.net&quot;,
  or &quot;example.org&quot; into your web browser.</p>
<p>These domain names are reserved for use in documentation and are
  not available for registration. See
  <a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC 2606</a>,
  Section 3.</p>
</BODY>
</HTML>
```

The response message in this case begins with the line

```
HTTP/1.1 200 OK
```

which is known as the *status line* of the response, and continues to the end of the example. The portion of the response between the status line and the first blank line following it is the header of the response. The part following this blank line—from `<HTML>` down—is the body of the response and is written using the HTML language, which will be discussed in the next chapter. For now, just notice that this body contains the information displayed by the browser.

Now that we have an idea of HTTP's basic structure, we'll look at some details of request and response messages.

1.4 HTTP Request Message

1.4.1 Overall Structure

Every HTTP request message has the same basic structure:

Start line
Header field(s) (one or more)
Blank line
Message body (optional)

The start line in the example request in Section 1.3.1 was

```
GET / HTTP/1.1
```

Every start line consists of three parts, with a single space used to separate adjacent parts:

1. Request method
2. Request-URI portion of web address
3. HTTP version

We'll cover each of these parts of the start line—in reverse order—in the next several subsections, then move on to the header fields and body.

1.4.2 HTTP Version

The initial version of HTTP was referred to as HTTP/0.9, and the first Internet RFC (Request for Comments; see the References section (Section 1.9) for more on RFCs) regarding HTTP described HTTP/1.0. In 1997, HTTP/1.1 was formally defined, and is currently an Internet Draft Standard [RFC-2616]. Essentially all operational browsers and servers support HTTP/1.1, including the server that generated the example in Section 1.3.1 (as indicated by the HTTP version portion of the status line). We will therefore focus on HTTP/1.1 in this chapter. If a new version of HTTP is developed in the future, the new standard defining this version will specify a new value for the version portion of the start line (assuming that the new standard has the same start line). The version string for HTTP/1.1 must appear in the start line exactly as shown, with all capital letters and no embedded white space.

1.4.3 Request-URI

The second part of the start line is known as the *Request-URI*. The concatenation of the string `http://`, the value of the Host header field (`www.example.org`, in this example), and the Request-URI (`/` in this example) forms a string known as a *Uniform Resource Identifier* (URI). A URI is an identifier that is intended to be associated with a particular resource (such as a web page or graphics image) on the World Wide Web. Every URI consists of two parts: the *scheme*, which appears before the colon (:), and another part that depends on the scheme. Web addresses, for the most part, use the `http` scheme (the scheme name in URIs is case insensitive, but is generally written in lowercase letters). In this scheme, the URI represents the location of a resource on the Web. A URI of this type is said to be a *Uniform Resource Locator* (URL). Therefore, URIs using the `http` scheme are both URIs and URLs. Some other URI schemes that mark the URI as a URL are shown in Table 1.1. A complete list of the currently registered URI schemes along with references to details on each scheme can be found at [IANA-SCHEMES].

In addition to the URL type of URI, there is one other type, called a *Uniform Resource Name* (URN). While not as common as URLs, URNs are sometimes used in web development (see Section 8.6 for an example). A URN is designed to be a unique name for a resource rather than specifying a location at which to find the resource. For example, an edition of *War and Peace* has an ISBN (International Standard Book Number) of 0-1404-4417-3 associated with it, and this is the only book worldwide with this number.

TABLE 1.1 Some Non-`http` URL Schemes

Scheme Name	Example URL	Type of Resource
`ftp`	`ftp://ftp.example.org/pub/afile.txt`	File located on FTP server
`telnet`	`telnet://host.example.org/`	Telnet server
`mailto`	`mailto:someone@example.org`	Mailbox
`https`	`https://secure.example.org/sec.txt`	Resource on web server supporting encrypted communication
`file`	`file:///C:/temp/localFile.txt`	File accessible from machine processing this URL

So it makes sense to associate information regarding this book, such as bibliographic data, with its ISBN. In fact, this book has an associated URN, which can be written as follows:

```
urn:ISBN:0-1404-4417-3
```

The URI for a URN always consists of three colon-separated parts, as illustrated here. The first part is the scheme name, which is always `urn` for a URN-type URI. The second part is the *namespace identifier*, which in this example is `ISBN`. Other currently registered URN namespace identifiers along with pointers to documentation for each are listed at [IANA-URNS]. The third part is the *namespace-specific string*. The exact format and meaning of this string varies with the namespace. In this example it represents the ISBN of a book and has a format defined by the documentation linked to at [IANA-URNS].

We will have more to say about URLs, particularly those with an `http` scheme, in Section 1.6. For now, we will complete our coverage of the HTTP request start line by examining the first part, the request method.

1.4.4 Request Method

The standard HTTP methods and a brief description of each are shown in Table 1.2. The method part of the start line of an HTTP request must be written entirely in uppercase letters, as shown in the table. In addition to the methods shown, the HTTP/1.1 standard defines a CONNECT method, which can be used to create certain types of secure connections. However, its use is beyond our scope and therefore will not be discussed further here.

The primary HTTP method is GET. This is the method used when you type a URL into the Location bar of your browser. It is also the method that is used by default when you click on a link in a document displayed in your browser and when the browser downloads images for display within an HTML document. The POST method is typically used to send information collected from a form displayed within a browser, such as an order-entry form, back to the web server. The other methods are not frequently used by web developers, and we will therefore not discuss them further here.

TABLE 1.2 Standard HTTP/1.1 Methods

Method	Requests server to . . .
GET	return the resource specified by the Request-URI as the body of a response message.
POST	pass the body of this request message on as data to be processed by the resource specified by the Request-URI.
HEAD	return the same HTTP header fields that would be returned if a GET method were used, but not return the message body that would be returned to a GET (this provides information about a resource without the communication overhead of transmitting the body of the response, which may be quite large).
OPTIONS	return (in Allow header field) a list of HTTP methods that may be used to access the resource specified by the Request-URI.
PUT	store the body of this message on the server and assign the specified Request-URI to the data stored so that future GET request messages containing this Request-URI will receive this data in their response messages.
DELETE	respond to future HTTP request messages that contain the specified Request-URI with a response indicating that there is no resource associated with this Request-URI.
TRACE	return a copy of the complete HTTP request message, including start line, header fields, and body, received by the server. Used primarily for test purposes.

1.4.5 Header Fields and MIME Types

We have already learned that the Host header field is used when forming the URI associated with an HTTP request. The Host header field is required in every HTTP/1.1 request message. HTTP/1.1 also defines a number of other header fields, several of which are commonly used by modern browsers. Each header field begins with a *field name*, such as Host, followed by a colon and then a *field value*. White space is allowed to precede or follow the field value, but such white space is not considered part of the value itself. The following slightly modified example of an actual HTTP request sent by a browser consists of a start line, 10 header fields, and a short message body:

```
POST /servlet/EchoHttpRequest HTTP/1.1
host: www.example.org:56789
user-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4)
  Gecko/20030624
accept: text/xml,application/xml,application/xhtml+xml,
  text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,
  image/gif;q=0.2,*/*;q=0.1
accept-language: en-us,en;q=0.5
accept-encoding: gzip,deflate
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
connection: keep-alive
keep-alive: 300
content-type: application/x-www-form-urlencoded
content-length: 13

doit=Click+me
```

Before describing each of the header fields, it will be helpful to understand some common header field features. First, header names are not case sensitive, although I will throughout this text refer to header field names following the capitalization used by the HTTP/1.1 reference [RFC-2616]. So, while the browser used "host" to name the first header field, I will refer to this as the "Host" header field. Second, a header field value may wrap onto several lines by preceding each continuation line with one or more spaces or tabs, as shown for the User-Agent and Accept fields of the preceding example. This also means that a header field name must begin at the first character of a line, with no preceding white space.

A third common feature is the use of so-called MIME types in several header field values. *MIME* is an acronym standing for Multipurpose Internet Mail Extensions, and refers to a standard that can be used to pass a variety of types of information, including graphics and applications, through e-mail as well as through other Internet message protocols. In particular, as defined in the MIME Internet Draft Standard [RFC-2045], the content of a MIME message is specified using a two-part, case-insensitive string which, in web applications, is known as the *content type* of the message. Two examples of standard MIME content-type strings are text/html and image/jpeg. The substring preceding the slash in these strings is the *top-level type*, and is normally one of a small number of standard types shown in Table 1.3. The substring following the slash, called the *subtype*, specifies the particular type of content relative to the top-level type. A complete list of current registered top-level types and subtypes can be found at [IANA-MIME]. In addition, *private* (unregistered) MIME top-level types and subtypes may be used. A private type or subtype is indicated by an "x-" (or "X-") prefix. Table 1.4 lists some common MIME types.

Yet another common feature of header fields is that many header field values use so-called *quality values* to indicate preferences. A quality value is specified by a string of

TABLE 1.3 Standard Top-level MIME Content Types

Top-level Content Type	Document Content
application	Data that does not fit within another content type and that is intended to be processed by application software, or that is itself an executable binary.
audio	Audio data. Subtype defines audio format.
image	Image data, typically static. Subtype defines image format. Requires appropriate software and hardware in order to be displayed.
message	Another document that represents a MIME-style message. For example, following an HTTP TRACE request message to a server, the server sends a response with a body that is a copy of the HTTP request. The value of the Content-Type header field in the response is message/http.
model	Structured data, generally numeric, representing physical or behavioral models.
multipart	Multiple entities, each with its own header and body.
text	Displayable as text. That is, a human can read this document without the need for special software, although it may be easier to read with the assistance of other software.
video	Animated images, possibly with synchronized sound.

TABLE 1.4 Some Common MIME Content Types

MIME Type	Description
text/html	HTML document
image/gif	Image represented using Graphics Interchange Format (GIF)
image/jpeg	Image represented using Joint Picture Expert Group (JPEG) format
text/plain	Human-readable text with no embedded formatting information
application/octet-stream	Arbitrary binary data (may be executable)
application/x-www-form-urlencoded	Data sent from a web form to a web server for processing

the form `;q=`*num*, where *num* is a decimal number between 0 and 1, with a higher number representing greater preference. Each quality value applies to all of the comma-separated field values preceding it back to the next earlier quality value. So, for example, according to the Accept header field (explained in Section 1.5) the browser in this example prefers text/xml (quality value 0.9) over image/jpeg (quality value 0.2). A final common header field feature is the use of the `*` character in a header field value as a wildcard character. For instance, the string `*/*` in the Accept header field value represents all possible MIME types.

Each of the header fields shown in the example, along with the Referer field (yes, this misspelling of "referrer" is the name of the field in the HTTP/1.1 standard), are briefly described in Table 1.5. The field values for Accept-Charset are discussed in detail in Section 1.5.4. Full details on all of these header fields, along with descriptions of the many other header fields defined in HTTP/1.1 plus an explanation of how you can define your own header fields, are contained in [RFC-2616].

1.5 HTTP Response Message

As we have seen earlier, an HTTP response message consists of a status line, header fields, and the body of the response, in the following format:

Status line
Header field(s) (one or more)
Blank line
Message body (optional)

In this section, we'll begin by describing the status line and then move on to an overview of some of the response header fields and related topics. The message body, if present, is often an HTML document; HTML is covered in the next chapter.

1.5.1 Response Status Line

The example status line shown earlier was

```
HTTP/1.1 200 OK
```

TABLE 1.5 Some Common HTTP/1.1 Request Header Fields

Field Name	Use
Host	Specify *authority* portion of URL (host plus port number; see Section 1.6.2). Used to support *virtual hosting* (running separate web servers for multiple fully qualified domain names sharing a single IP address).
User-Agent	A string identifying the browser or other software that is sending the request.
Accept	MIME types of documents that are acceptable as the body of the response, possibly with indication of preference ranking. If the server can return a document according to one of several formats, it should use a format that has the highest possible preference rating in this header.
Accept-Language	Specifies preferred language(s) for the response body. A server may have several translations of a document, and among these should return the one that has the highest preference rating in this header field. For complete information on registered language tags, see [RFC-3066] and [ISO-639-2].
Accept-Encoding	Specifies preferred encoding(s) for the response body. For example, if a server wishes to send a compressed document (to reduce transmission time), it may only use one of the types of compression specified in this header field.
Accept-Charset	Allows the client to express preferences to a server that can return a document using various character sets (see Section 1.5.4).
Connection	Indicates whether or not the client would like the TCP connection kept open after the response is sent. Typical values are `keep-alive` if connection should be kept open (the default behavior for servers/clients compatible with HTTP/1.1), and `close` if not.
Keep-Alive	Number of seconds TCP connection should be kept open.
Content-Type	The MIME type of the document contained in the message body, if one is present. If this field is present in a request message, it normally has the value shown in the example, application/x-www-form-urlencoded.
Content-Length	Number of bytes of data in the message body, if one is present.
Referer	The URI of the resource from which the browser obtained the Request-URI value for this HTTP request. For example, if the user clicks on a hyperlink in a web page, causing an HTTP request to be sent to a server, the URI of the web page containing the hyperlink will be sent in the Referer field of the request. This field is not present if the HTTP request was generated by the user entering a URI in the browser's Location bar.

Like the start line of a request message, the status line consists of three fields: the HTTP version used by the server software when formatting the response; a numeric *status code* indicating the type of response; and a text string (the *reason phrase*) that presents the information represented by the numeric status code in human-readable form. In this example, the status code is 200 and the reason phrase is `OK`. This particular status code indicates that no errors were detected by the server. The body of a response having this status code should contain the resource requested by the client.

All status codes are three-digit decimal numbers. The first digit represents the general class of status code. The five classes of HTTP/1.1 status codes are given in Table 1.6. The last two digits of a status code define the specific status within the specified class. A few of the more common status codes are shown in Table 1.7. The HTTP standard recommends

TABLE 1.6 HTTP/1.1 Status Code Classes (First Digit of Status Code)

Digit	Class	Standard Use
1	Informational	Provides information to client before request processing has been completed.
2	Success	Request has been successfully processed.
3	Redirection	Client needs to use a different resource to fulfill request.
4	Client Error	Client's request is not valid.
5	Server Error	An error occurred during server processing of a valid client request.

reason phrases for all status codes, but a server may use alternative but equivalent phrases. All status codes and recommended reason phrases are contained in [RFC-2616].

1.5.2 Response Header Fields

Some of the header fields used in HTTP request messages, including Connection, Content-Type, and Content-Length, are also valid in response messages. The Content-Type of a response can be any one of the MIME type values specified by the Accept header field of the corresponding request. Some other common response header fields are shown in Table 1.8.

TABLE 1.7 Some Common HTTP/1.1 Status Codes

Status Code	Recommended Reason Phrase	Usual Meaning
200	OK	Request processed normally.
301	Moved Permanently	URI for the requested resource has changed. All future requests should be made to URI contained in the Location header field of the response. Most browsers will automatically send a second request to the new URI and display the second response.
307	Temporary Redirect	URI for the requested resource has changed at least temporarily. This request should be fulfilled by making a second request to URI contained in the Location header field of the response. Most browsers will automatically send a second request to the new URI and display the second response.
401	Unauthorized	The resource is password protected, and the user has not yet supplied a valid password.
403	Forbidden	The resource is present on the server but is read protected (often an error on the part of the server administrator, but may be intentional).
404	Not Found	No resource corresponding to the given Request-URI was found at this server.
500	Internal Server Error	Server software detected an internal failure.

TABLE 1.8 Some Common HTTP/1.1 Response Header Fields

Field Name	Use
Date	Time at which response was generated. Used for cache control (see Section 1.5.3). This field must be supplied by the server.
Server	Information identifying the server software generating this response.
Last-Modified	Time at which the resource returned by this request was last modified. Can be used to determine whether cached copy of a resource is valid or not (see Section 1.5.3).
Expires	Time after which the client should check with the server before retrieving the returned resource from the client's cache (see Section 1.5.3).
ETag	A hash code of the resource returned. If the resource remains unchanged on subsequent requests, then the ETag value will also remain unchanged; otherwise, the ETag value will change. Used for cache control (see Section 1.5.3).
Accept-Ranges	Clients can request that only a portion (*range*) of a resource be returned by using the Range header field. This might be used if the resource is, say, a large PDF file and only a single page is currently needed. Accept-Ranges specifies the units that may be used by the client in a range request, or `none` if range requests are not accepted by this server for this resource.
Location	Used in responses with redirect status code to specify new URI for the requested resource.

1.5.3 Cache Control

Several of the response header fields described in Table 1.8 are used in conjunction with cache control. In computer systems, a *cache* is a repository for copies of information that originates elsewhere. A copy of information is placed in a cache in order to improve system performance. For example, most personal computer systems use a small, high-speed memory cache to hold copies of some of the data contained in RAM memory, which is slower than cache memory.

Most web browsers automatically cache on the client machine many of the resources that they request from servers via HTTP. For example, if an image such as a button icon is included in a web page, a copy of the image obtained from the server will typically be cached in the client's file system. Then if another page at the same site uses the same image, the image can be retrieved from the client file system rather than sending another HTTP request to the server and waiting for the server's response containing the image. HTTP caching, when successful, generally leads to quicker display by the browser, reduced network communication, and reduced load on the web server.

However, there is a key drawback to using a cache: information in a cache can become *invalid*. For example, if the button image in the preceding example is modified on the server, but a client accesses its cached copy of the older version of the image, then the client will display an invalid version of the image. This problem can be avoided in several ways.

One approach to guaranteeing that a cached copy of a resource is valid is for the client to ask the server whether or not the client's copy is valid. This can be done with relatively little communication by sending an HTTP request for the resource using the HEAD method, which returns only the status line and header portion of the response. If

the response message contains a Last-Modified time, and this time precedes the value of the Date header field returned with the cached resource, then the cached copy is still valid and can be used. Otherwise, the cached copy is invalid and the browser should send a normal GET request for the resource.

A somewhat simpler approach can be used if the server returns an ETag with the resource. The client can then simply compare the ETag returned by a HEAD request with the ETag stored with the cached resource. If the ETag values match, then the cached copy is valid; otherwise, it is not. This approach avoids the complexity of comparing two dates to determine which is larger.

Finally, if the server can determine in advance the earliest time at which a resource will change, the server can return that time in an Expires header. In this case, as long as the Expires time has not been reached, the client may use the cached copy of the resource without the need to validate with the server. If an Expires time is not included in a response, a browser may use a heuristic algorithm to choose an expiration time and then behave as if this time had been passed to it in an Expires header. This behavior can be prevented by sending an Expires time that precedes the Date value (a value of 0 is commonly used for this purpose). If this is done, then an HTTP/1.1-compliant browser will validate before each access to the resource.

The HTTP/1.1 specification provides a variety of other header fields related to caching; see [RFC-2616] for full details.

1.5.4 Character Sets

Finally, a word about how characters are represented in web documents. As you know, characters are represented by integer values within a computer. A *character set* defines the mapping between these integers, or *code points*, and characters. For example, US-ASCII [RFC-1345] is the character set used to represent the characters used in HTTP header field names, and is also used in key portions of many other Internet protocols. Each US-ASCII character can be represented by a 7-bit integer, which is convenient in part because the messages transmitted by the Internet Protocol are viewed as streams of 8-bit bytes, and therefore each character can be represented by a single byte.

However, many characters in common use in modern languages are not contained in the US-ASCII character set. Over the years, a wide variety of other character sets have been defined for use with languages other than U.S. English and also for representing characters that are not associated with human language representation, such as mathematical and graphical symbols.

For web pages, which are meant to be viewed throughout the world, it is vital that a single worldwide character set be used. So, as in the Java™ programming language, the underlying character set used internally by web browsers is defined by the Unicode™ Standard [UNICODE]. The Unicode Standard is an attempt to provide a single character set that encompasses every human language representation as well as all other commonly used symbols. The Unicode Standard's Basic Multilingual Plane (BMP), which covers most of the commonly used characters in every modern language, uses 16-bit character codes, and the full character code space of the Unicode Standard extends to 21-bit integers.

Of course, if the resource requested by a client is written using the US-ASCII character set, then sending 21 (or more) bits per character from the server to the client would take roughly three times as long as sending the ASCII characters. Therefore, most browsers, for purposes of efficiency and compatibility, accept a variety of character sets in addition to those in Unicode. See [IANA-CHARSETS] for a complete (long) list of character sets currently registered for use over the Internet.

More generally, in addition to a variety of character sets, most browsers also accept certain character encodings. A *character encoding* is a bit string that must be decoded into a code-point integer that is then mapped to a character according to the definition provided by some character set. A character encoding often represents characters using variable-length bit strings, with common characters represented using shorter strings and less-common characters using longer strings. For example, UTF-8 and UTF-16 are encodings of the character set in Unicode that use variable numbers of 8- and 16-bit values to encode all possible Unicode Standard characters. (Don't confuse character encoding with the message encoding concept mentioned earlier. Message encoding typically involves applying a general-purpose compression algorithm to the body of a message, regardless of the character encoding used.)

The Accept-Charset header field is used by a client to tell a server the character sets and character encodings that it will accept as well as its preferred character sets or encodings, if more than one is available for the requested document. In our earlier example, the header field

```
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

said that the client would prefer to receive documents using the ISO-8859-1 character set or the UTF-8 encoding of the characters in Unicode, but that it would also accept any other valid Internet character set/encoding. (ISO-8859-1 is an 8-bit superset of US-ASCII that contains many characters found in Latin-based languages but not in English. ISO-8859-1 and UTF-8 are preferred even though they have the same quality value as `*` because specific field values are given preference over the `*` wildcard.)

A web server can inform a client about the character set/encoding used in a returned document by adding a `charset` parameter to the value of the Content-Type header field. For example, the following Content-Type header field in an HTTP response would indicate that the body of the message is an HTML document written using the UTF-8 character encoding:

```
Content-Type: text/html; charset=UTF-8
```

The US-ASCII character set is a subset of both the ISO-8859-1 character set and the UTF-8 character encodings, so the `charset` parameter is set to one of these two values for many US-ASCII documents in order to ensure international compatibility. We will learn other ways to indicate the character set/encoding for a document in later chapters.

Now that we have covered HTTP in some detail, we're ready to look at the primary software applications that communicate using HTTP: web clients and servers. We'll begin with the more familiar client software before moving on to web servers.

1.6 Web Clients

A *web client* is software that accesses a web server by sending an HTTP request message and processing the resulting HTTP response. Web browsers running on desktop or laptop computers are the most common form of web client software, but there are many other forms of client software, including text-only browsers, browsers running on cell phones, and browsers that speak a page (over the phone, for example) rather than displaying the page. In general, any web client that is designed to directly support user access to web servers is known as a *user agent*. Furthermore, some web clients are not designed to be used directly by humans at all. For example, software *robots* are often used to automatically crawl the Web and download information for use by search engines (and, unfortunately, e-mail spammers).

We will focus here on traditional browsers, since they are the most widely used web client software and have features that are generally a superset of those found in other clients. A brief history of these browsers will provide some useful background.

Early web browsers generally either were text-based or ran on specialized platforms, such as computers from Sun Microsystems or the now-defunct NeXT Systems. The Mosaic™ browser, developed at the National Center for Supercomputer Applications (NCSA) in 1993, was the starting point for bringing graphical web browsing to the general public. The developers of Mosaic founded Netscape Communications Corporation, which dedicated a large team to developing and marketing a series of Netscape Navigator® browsers based on Mosaic. Microsoft soon followed with the Microsoft® Internet Explorer (IE) browser, which was originally based on Mosaic.

For a time, a "browser war" was waged between Netscape and Microsoft, with each company trying to add features and performance to its browser in order to increase its market share. Netscape soon found itself at a disadvantage, however, as Microsoft began bundling IE with its popular Windows® operating system. The war soon ended, and Microsoft was victorious. Netscape, acquired by America Online (at the time primarily an Internet service provider), chose to make its source code public and launched the Mozilla project as an open-source approach to developing new core functionality for the Netscape® browser. In particular, Netscape browser releases starting with version 6.0 have been based on software developed as part of the Mozilla project.

At the time of this writing, IE is by far the most widely used browser in the world. However, the Mozilla™ and Firefox™ browsers from the Mozilla Foundation are increasingly popular, and other browsers, including the Opera™ and Safari™ browsers, also have significant user communities.

Despite this diversity, all of the major modern browsers support a common set of basic user features and provide similar support for HTTP communication. A number of common browser features are discussed in the remainder of this section. For concreteness, I will also explain how to access the features described using one particular browser, Mozilla 1.4, and will also use the Mozilla browser for most examples in later chapters. A primary reason for choosing to use Mozilla as a concrete browser example is that it runs on Linux®, Windows, and Macintosh® systems. Also, the fact that it is open source means that if you're curious about details of how a feature operates, you have access to the source code itself. In addition to having essentially all of the features found in IE, Mozilla has some nice

tools for software developers that are not found in basic IE distributions. Finally, as we will learn in later chapters, Mozilla browsers are designed to comply with HTML and other Internet standards, while IE is (at this time, at least) less standards compliant. Instructions for downloading and installing Mozilla 1.4 are found in Appendix A.

1.6.1 Basic Browser Functions

The window of a typical modern browser is split into several rectangular regions, most of which are known as *bars*. Figure 1.4 shows five standard regions in a Mozilla 1.4 window. The primary region is the *client area*, which displays a document. For many documents, the *title bar* displays a title assigned by the document author to the document currently displayed within the client area. The title bar also displays the browser name as well as standard window-management controls. The *menu bar* contains a set of dropdown menus, much like most other applications that incorporate a graphical user interface (GUI). We'll take a closer look at the Mozilla menus in Section 1.6.3. The browser's *Navigation toolbar* contains standard push-button controls that allow the user to return to a previously viewed web page (Back), reverse the effect of pressing Back (Forward), ask the server for an updated version of the page currently viewed (Reload), halt page downloading currently in progress (Stop), and print the client area of the window (Print). Clicking the small down-arrow to the right of some buttons produces a menu allowing users to override the default behavior of the associated button. For example, clicking the arrow to the right of Back produces a menu of titles of a number of documents that have been recently viewed, any of which can be loaded into the client area by selecting its title from the menu. The Navigation toolbar also contains a text box, known as the *Location bar*, where a user can enter a URL and press the Enter key in order to request the browser to display the document located at the specified URL. Clicking the Search button instead of pressing Enter causes the information entered in the text box to be sent to a search engine. Clicking the down-arrow at the right side of the Location bar produces a dropdown menu of recently visited URLs that can be visited again with a single click. Finally, the *status bar* displays messages and icons related to the

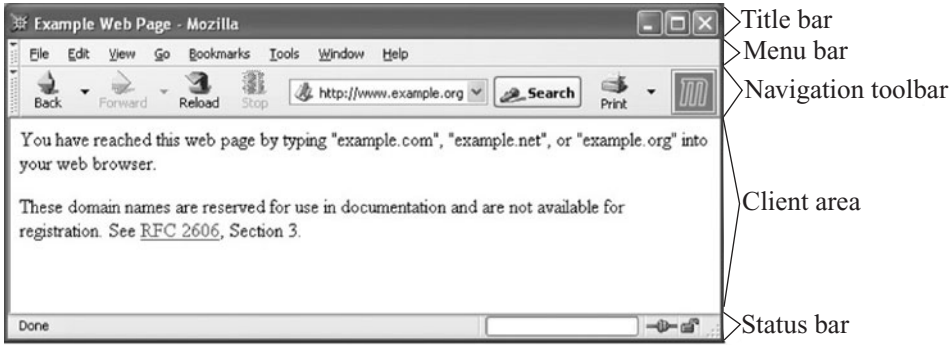

FIGURE 1.4 Some of the standard Mozilla bars. The content shown is subject to copyright and used by permission of IANA.

status of the browser. For example, the two icons in the right portion of the status bar in Figure 1.4 show that the browser is online (left icon) and that the browser is communicating with the server over an insecure communication channel. The messages displayed in the left portion of the status bar are normally information about the communication between client and server (Table 1.9).

A primary task of any browser is to make HTTP requests on behalf of the browser user. If a user types an `http`-scheme URL in Mozilla's Location bar, for example, the browser must perform a number of tasks:

1. Reformat the URL entered as a valid HTTP request message.
2. If the server is specified using a host name (rather than an IP address), use DNS to convert this name to the appropriate IP address.
3. Establish a TCP connection using the IP address of the specified web server.
4. Send the HTTP request over the TCP connection and wait for the server's response.
5. Display the document contained in the response. If the document is not a plain-text document but instead is written in a language such as HTML, this involves *rendering* the document: positioning text and graphics appropriately within the browser window, creating table borders, using appropriate fonts and colors, etc.

Before discussing various features of browsers that can be controlled by users, it will be helpful to have a more complete understanding of URLs.

1.6.2 URLs

An `http`-scheme URL consists of a number of pieces. In order to show the main possibilities, let's consider the following example URL:

```
http://www.example.org:56789/a/b/c.txt?t=win&s=chess#para5
```

The portion of an `http` URL following the `://` string and before the next slash (`/`) (or through the completion of the URL, if there is no trailing slash) is known as the *authority* of the URL. It consists of either a fully qualified domain name (or other name that can be resolved to an IP address, such as an unqualified name of a machine on the local network) or an IP

TABLE 1.9 Some Mozilla Status Messages

Status Message	Meaning
Resolving host www.example.org . . .	Requested IP address from DNS; waiting for response.
Connecting to www.example.org . . .	Creating TCP connection to server.
Waiting for www.example.org . . .	Sent HTTP request to server; waiting for HTTP response.
Transferring data from www.example.org . . .	HTTP response has begun, but has not completed.
Done	HTTP response has been received, although further processing may be needed before the document will be displayed.

address of an Internet web server, optionally followed by a colon (:) and a port number. As indicated earlier, if the port number is omitted, then a TCP connection to port 80 is implied. In this example, the authority is `www.example.org:56789` and consists of the fully qualified domain name `www.example.org` followed by the port number 56789.

The portion from the slash following the authority through the question mark (?) (or through the end of the URL, if there is no question mark) is called the *path* of the URL. The leading slash is part of the path, but the question mark is not. So the path in the example URL just given is `/a/b/c.txt`. The fact that this looks a great deal like a Linux file reference to a file named `c.txt` located within the `b` subdirectory of the `a` directory of the root (`/`) of the file system is not entirely a coincidence. In many cases, the path portion of a URL is in fact concatenated by the server with a base file path in order to form an actual file path on the server's system. We'll learn more about how servers use URL paths later in this chapter as well as in later chapters.

Following the path there may be a question mark followed by information up to a number sign (#). The information between but not including the question mark and number sign is the *query* portion of the URL, and in general a string of the form shown is known as a *query string*. The query portion of the example URL is `t=win&s=chess`. Originally, the query portion of a URL was intended to pass search terms to a web server. So in this example, it might be that the user is seeking a resource with a title containing the string "win" that is related to the subject "chess." As we will learn in later chapters, while query strings are still sometimes used to represent search terms, they are also used for a variety of other purposes in modern web systems. We will also learn that query strings may appear in the body of POST requests, as well as how to encode special characters in the query strings sent to web servers.

A browser forms the Request-URI portion of an HTTP request from a URL by concatenating the path and query portions of the URL with an intervening question mark. Thus, the Request-URI for the example URL would be

```
/a/b/c.txt?t=win&s=chess
```

Syntactically, the query portion of a URL can only be present if the path portion is present. If both the path and query are missing from a URL, then the Request-URI must be set to `/`, which is known as the *root* path. This is why we used a `/` as the Request-URI in the example of Section 1.3.1.

The final optional part of an `http`-scheme URL—the portion following but not including the number sign—is known as the *fragment* of the URL, and the string contained in the fragment is known as a *fragment identifier*. Fragment identifiers are used by browsers to scroll HTML documents; details are given in the next chapter, which covers HTML.

Summarizing, if a user types a URL such as the one considered into a browser's Location bar and presses Enter, the browser will generate an HTTP request message as follows. The request start line will begin with GET. The path and query portions of the URL will be used as the second, Request-URI portion of the start line. Assuming the browser is HTTP/1.1 compliant, the final portion of the start line will be the string `HTTP/1.1` (this string must be uppercase). The request will also contain a Host header field having as its value the authority portion of the URL. The fragment portion of the URL is not sent to

the web server, but is instead used by the browser to modify the way in which it displays any HTML document sent to the browser in the HTTP response returned as a result of this request. Other header fields will also generally be included, as described earlier.

So, given the example URL, the browser would send a request containing the lines (spacing and some capitalization might vary from that shown):

```
GET /a/b/c.txt?t=win&s=chess HTTP/1.1
...
Host: www.example.org:56789
...
```

1.6.3 User-Controllable Features

Graphical browsers also provide many user-controllable features, including:

- *Save:* Most documents can be saved by the user to the client machine's file system. If the document is an HTML page that contains other documents, such as images, then the browser will attempt to save all of these documents locally so that the entire page can be displayed from the local file system. A user saves a document in Mozilla under the **File**|**Save Page As** menu.
- *Find in page:* Standard documents (text and HTML) can be searched with a function that is similar to that provided by most word processors. In Mozilla, the find function is accessed under the **Edit**|**Find in This Page** menu. (Mozilla also provides a "find as you type" feature under Edit that is similar to the incremental search in Emacs, for users familiar with that paradigm.)
- *Automatic form filling:* The browser can "remember" information entered on certain forms, such as billing address, phone numbers, etc. When another form is visited at a later date, the browser can automatically fill in previously saved data. The **Edit**|**Save Form Info** and **Edit**|**Fill in Form** menu options can be used to save and retrieve form information in Mozilla. The **Tools**|**Form Manager** menu can be used to manage saved form information.
- *Preferences:* Users can customize browser functionality in a wide variety of ways. In Mozilla, a window presenting preference options is obtained by selecting **Edit**|**Preferences** (Figure 1.5). The Appearance, Navigator, and Advanced categories (left subwindow) and their subcategories are used to customize Mozilla. Some preference settings directly related to the HTTP topics covered earlier are:
  - *Accept-Language:* The non-* values sent by the browser for this HTTP request header field can be set under the **Navigator**|**Languages** category, **Languages for Web Pages** box.
  - *Default character set/encoding:* The character set/encoding to be assumed for documents that do not specify one is also set under **Navigator**|**Languages** in the **Character Coding** box.
  - *Cache properties:* The amount of local storage allocated to the cache and the conditions controlling when a cached file will be validated are set under **Advanced**|**Cache** in the **Set Cache Options** box.

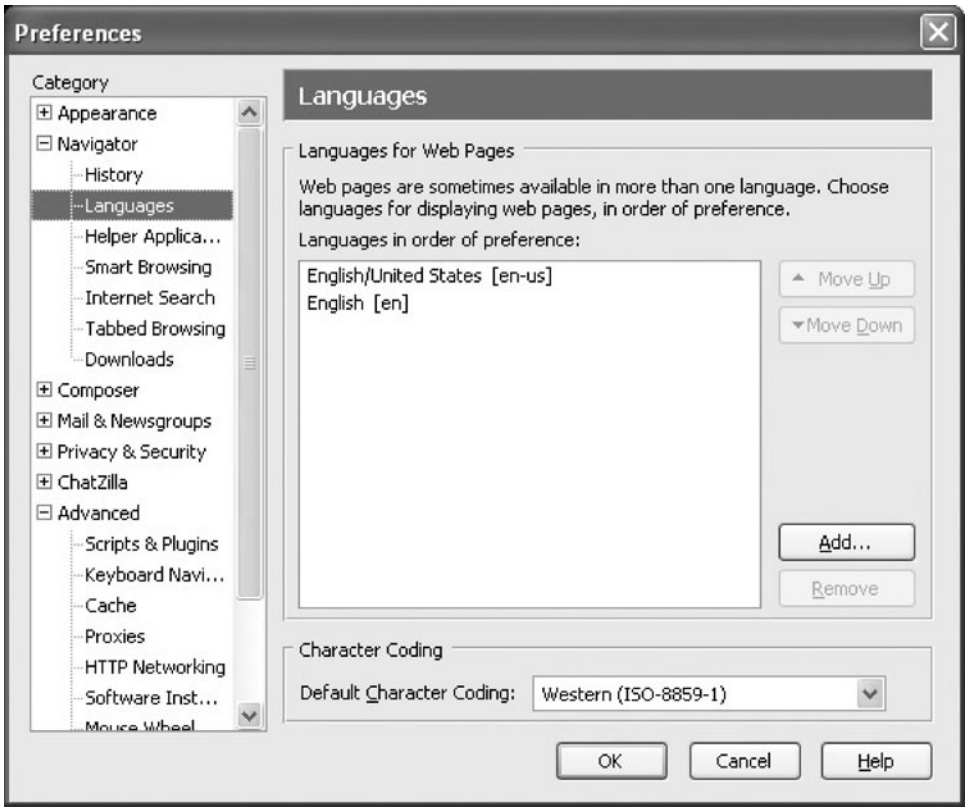

FIGURE 1.5 Preferences window with Languages category selected.

  - *HTTP settings:* The version of HTTP used and whether or not the client will keep connections alive is set under **Advanced**|**HTTP Networking** in the **Direct Connection Options** box.
- *Style definition:* The user can define certain aspects affecting how the browser renders HTML pages, such as font sizes, background and foreground colors, etc. In Mozilla, the font size can be modified using **View**|**Text Zoom**. If a page offers alternative styles, they can be selected using the **View**|**Use Style** menu as discussed in Chapter 3, where methods for changing default browser style settings are also described.
- *Document meta-information:* Interested users can view information about the displayed document, such as the document's MIME type, character encoding, size, and, if the document was written using HTML, the raw HTML source from which the rendering in the client area was produced. In Mozilla, **View**|**Page Source** is used to view raw HTML, and **View**|**Page Info** to view other so-called *meta-information*, that is, information about the document rather than information contained in the document itself.
- *Themes:* The look of one or more of the browser bars, particularly the navigation bar, can be modified by applying a certain theme (sometimes called a "skin"). In Mozilla, the

browser scheme can be modified using **View|Apply Theme**. Additional themes can be obtained from **View|Apply Theme|Get New Themes**.

- *History:* The browser will automatically maintain a list of all pages visited within the last several days. Users can use the history list to easily return to any recently visited page. In Mozilla, the history list can be reached by selecting **Go|History**.
- *Bookmarks ("favorites" in Internet Explorer):* Users can explicitly *bookmark* a web page, that is, save the URL for that page for an indefinite length of time. At any later time, the browser's bookmark facility can be used to easily return to any bookmarked page. Options under the **Bookmarks** menu in Mozilla allow users to bookmark a page, return to a bookmarked page, and edit the bookmark list.

1.6.4 Additional Functionality

In addition to the facilities for end users described in the preceding subsection, browsers perform a number of other functions, including:

- *Automatic URL completion:* If the user has entered a URL in the Location bar and begins to type it again (within the next several days), the URL will be completed automatically by the browser.
- *Script execution:* In addition to displaying documents, browsers can run programs (scripts). These programs can perform a variety of tasks, from validating data entered on a form before sending it to a web server to creating various dynamic effects on web pages, such as drop-down menus.
- *Event handling:* When the user performs an action, such as clicking on a link or a button in a web page, the browser treats this as the occurrence of an *event*. Browsers recognize a number of different types of events, including mouse button clicks, mouse movement, and even events not directly under user control such as the completion of the browser's rendering of a document. A browser can perform a variety of actions in response to an event—loading a document from a URL, clearing a form, or calling a script function defined by the document author, for example.
- *Management of form GUI:* If a web page contains a form with fill-in fields, the browser must allow the user to perform standard text-editing functions within these fields. It also needs to automatically provide certain graphical feedback, such as changing a button image when it is pressed or providing a text cursor in a text field that will receive keyboard input.
- *Secure communication:* When the user sends sensitive information, such as a credit card number, to a web server, the browser can encode this information in a way the prevents any machines along the IP route from the client to the server from obtaining the information.
- *Plug-in execution:* While the browser itself normally understands only a limited number of MIME types, most browsers support some form of *plug-in* protocol that allows the browser's capabilities to be supplemented by other software. If a browser has a plug-in for displaying, say, a document conforming to the application/pdf MIME type, then when the browser receives such a document it will pass it—via the plug-in protocol—to the appropriate plug-in for display. Some plug-ins may display the document within the

browser's client area, while others may display in a separate window that is controlled by the plug-in itself. Plug-ins are often installed automatically, after user permission is obtained, when an unsupported MIME type is encountered. To see a list of plug-ins installed in your copy of Mozilla, select **Help|About Plug-ins**.

Some other standard browser features, such as a facility for managing so-called *cookies*, are described in later chapters. In addition to standard browser features, Mozilla also provides a number of tools specifically designed for use by software developers, such as a script console and debugging tools. Some of these tools will also be described in later chapters.

This completes our coverage of web browsers. It's now time to move to the software running on the other end of the HTTP communications pipeline: web servers.

1.7 Web Servers

In this section, we'll cover basic functionality found in most web servers as well as some specific instructions for accessing and modifying the parameters for one particular web server, Tomcat 5.0. We'll also briefly look at how web servers support secure communication with browsers.

1.7.1 Server Features

The primary feature of every web server is to accept HTTP requests from web clients and return an appropriate resource (if available) in the HTTP response. Even this basic functionality involves a number of steps (the quoted terms used in this list are defined in subsequent paragraphs):

1. The server calls on TCP software and waits for connection requests to one or more ports.
2. When a connection request is received, the server dedicates a "subtask" to handling this connection.
3. The subtask establishes the TCP connection and receives an HTTP request.
4. The subtask examines the Host header field of the request to determine which "virtual host" should receive this request and invokes software for this host.
5. The virtual host software maps the Request-URI field of the HTTP request start line to a resource on the server.
6. If the resource is a file, the host software determines the MIME type of the file (usually by a mapping from the file-name extension portion of the Request-URI), and creates an HTTP response that contains the file in the body of the response message.
7. If the resource is a program, the host software runs the program, providing it with information from the request and returning the output from the program as the body of an HTTP response message.
8. The server normally logs information about the request and response—such as the IP address of the requester and the status code of the response—in a plain-text file.

9. If the TCP connection is kept alive, the server subtask continues to monitor the connection until a certain length of time has elapsed, the client sends another request, or the client initiates a connection close.

A few definitions will be helpful before proceeding to more detailed coverage of web server features. First, all modern servers can concurrently process multiple requests. It is as if multiple copies of the server were running simultaneously, each devoted to handling the requests received over a single TCP connection. The specifics of how this concurrency is actually implemented on a system may depend on many factors, including the number of processors available in the system, the programming language used, and programmer choices. We will learn more about concurrent server processing in Chapter. 6. For now, I will simply use the term *subtask* to refer to the concept of a single "copy" of the server software handling a single client connection.

Another term that may need some explanation is *virtual host*. As noted earlier, every HTTP request must include a Host header field. The reason for this requirement is that multiple host names may all be mapped by the Internet DNS system to a single IP address. For example, a single server machine within a college may host web sites for multiple departments. Each web site would be assigned its own fully qualified domain name, such as `www.cs.example.edu`, `www.physics.example.edu`, and so on. But DNS would be configured to map all of these domain names to a single IP address. When an HTTP request is received by the web server at this address, it can determine which *virtual* host is being requested by examining the Host header. Separately configured software can then be used to handle the requests for each virtual host.

Finally, as noted in point 7, the documents returned by web servers are often produced by executing software at the time of the HTTP request rather than being generated beforehand and stored in the server's file system for later retrieval. One significant difference between web servers concerns the support that each has for executing software written in various traditional programming languages as well as in scripting languages. We'll touch on some of these differences in the next subsection, which briefly surveys the history of web server development.

1.7.2 Server History

Just as the NCSA Mosaic™ browser was the starting point for subsequent browser development efforts by Netscape and Microsoft, NCSA's *httpd* web server was also a starting point for server development. httpd was used on a large fraction of the early web servers, but the NCSA discontinued development of the server in the mid-1990s. When this happened, several individuals who were running httpd at their sites joined forces and began developing their own updates to the open-source httpd software. Their updates were called "patches," and this led to calling their work "a patchy server," which soon became known as "the Apache server." They made the first public release of their free, open-source server in April 1995, and within a year Apache was the most widely used server on the Web. It has held that distinction to this day, although many large corporate and government sites tend to use commercial server software instead.

As with web browsers, Microsoft began development of web servers well after others had begun, but quickly caught up. Microsoft's Internet Information Server (IIS) provides

essentially all of the features found in Apache, although IIS does have the drawback of running only on Windows systems, while Apache runs on Windows, Linux, and Macintosh systems. IIS and Apache are, at the time of this writing, by far the most widely used servers on the market.

Both servers can be configured to run a variety of types of programs, although certain programming languages tend to be used more frequently on one system than the other. For example, many IIS servers run programs written in VBScript (a derivative of Visual Basic), while a typical Apache server might run programs written in either Perl or the PHP scripting language (PHP stands for "PHP Hypertext Processor"; yes, the definition is infinitely recursive). A number of IIS and Apache servers also run Java programs. When running a Java program, both Apache and IIS servers are usually configured to run the program by using separate software called a *servlet container*. The servlet container provides the Java Virtual Machine that runs the Java program (known as a *servlet*), and also provides communication between the servlet and the Apache or IIS web server.

Tomcat is a popular, free, and open-source servlet container developed and maintained by the Apache Software Foundation, the same organization that is continuing development of the Apache web server. In addition to running as a servlet container called on by web servers, Tomcat can also be run as a standalone web server that communicates directly with web clients. Furthermore, the standalone Tomcat server can serve documents stored in the server machine's file system and run programs written in non-Java languages.

To provide a concrete illustration of server configuration, we will next cover configuration of a Tomcat 5.0 server in some detail (this is the server you will have if you follow the instructions for installing JWSDP in Appendix A). The Tomcat material presented here is not meant to be a comprehensive reference, but is primarily intended to introduce you to some key terms and concepts that are encountered when setting up any web server, not just Tomcat. Since we will be using Java servlets and related technologies to illustrate server-side programming in later chapters, it is natural for us to focus on Tomcat rather than non-Java servers in this chapter. If you understand Tomcat configuration well, configuring a basic IIS or Apache server should not be particularly difficult.

1.7.3 Server Configuration and Tuning

Modern servers have a large number of configuration parameters. In this section, we will cover many of the key configuration items found in Tomcat. Similar features, along with some not found in Tomcat, are included in the Apache and IIS servers.

Broadly speaking, server configuration can be broken into two areas: external communication and internal processing. In Tomcat, this corresponds to two separate Java packages: Coyote, which provides the HTTP/1.1 communication, and Catalina, which is the actual servlet container. Some of the Coyote parameters, affecting external communication, include the following:

- IP addresses and TCP ports that may be used to connect to this server.
- Number of subtasks (called *threads* in Java) that will be created when the server is initialized. This many TCP connections can be established simultaneously with minimal overhead.

- Maximum number of threads that will be allowed to exist simultaneously. If this is larger than the previous value, then the number of threads maintained by the server may change, either up or down, over time.
- Maximum number of TCP connection requests that will be queued if the server is already running its maximum number of threads. Connection requests received if the queue is full will be refused.
- Length of time the server will wait after serving an HTTP request over a TCP connection before closing the connection if another request is not received.

The settings of these parameters can have a significant influence on the performance of a server; changing the values of these and similar parameters in order to optimize performance is often referred to as *tuning* the server. As with all optimization problems, there are various trade-offs involved in attempting to tune a server. For example, increasing the maximum number of simultaneous threads that may execute increases memory requirements and thread-management overhead, and may lead to slower responses to individual requests, due to sharing CPU cycles among the large number of threads. On the other hand, lower values for this parameter may lead to some clients having their connection requests refused, which may lead some users to believe that the site is down. Tuning is therefore often performed by trial and error: if a server seems to be running poorly by some measure, the server administrator may try to vary one or more of these parameters and observe the impact, retaining parameter values that seem to help. *Load generation* or *stress test* tools can be used to simulate requests to a web server, and can therefore be helpful for experimenting with tuning parameters based on anticipated traffic patterns even before a web site "goes live." A fuller discussion of server tuning is beyond the scope of this book.

The internal Catalina portion of Tomcat also has a number of parameter settings that affect functionality. These settings can determine:

- Which client machines may send HTTP requests to the server.
- Which virtual hosts are listening for TCP connections on a given port.
- What logging will be performed.
- How the path portion of Request-URIs will be mapped to the server's file system or other resources.
- Whether or not the server's resources will be password protected.
- Whether or not resources will be cached in the server's memory.

The Tomcat 5.0 server you have installed if you followed the instructions in Appendix A has a web interface for setting most of these parameters. If your server is installed at the default port 8080 and you open a browser on the machine running the server, then browsing to the URL

```
http://localhost:8080
```

(more on `localhost` in Section 1.7.4) and clicking the Server Administration link (you may need to scroll down to find this link) should cause a log-in page to be displayed. Otherwise, if the server is not on the machine you are browsing from, or if your browser is not at port

8080, modify the URL to contain the correct host and/or port number. You were asked for a user name and password when you installed Tomcat; enter them on this log-in page. You should then see a page such as the one in Figure 1.6.

Because your copy of Tomcat was included in the Java Web Services Developer Pack (JWSDP), there is already a JWSDP Service entry in the list on the left side of the browser window. Each Service in Tomcat is almost its own web server, except that a Service cannot be individually stopped and started (only the underlying server can be stopped and started, as described in Appendix A). We will only cover here how to change parameters of the JWSDP Service; the procedures for creating a new Service are similar.

First, click on the handle icon next to the JWSDP Service entry in order to reveal its associated server components (Figure 1.7). This Service has five components: one each of Connector, Host, Logger, Realm, and Valve. A Connector is a Coyote component that handles HTTP communications directed to a particular port. Clicking on the Connector item in the JWSDP Service list will produce a window such as the one shown in Figure 1.8. The panel on the right in this figure is typical of the panels displayed for creating and editing Tomcat components. At the top of the panel is a dropdown menu of possible actions that can be performed for this component, such as creating subcomponents or deleting a component (there are no actions for this particular component). Below this menu is a Save button that must be clicked after entering data in the fields further below in order to save this data. This temporarily saves the data from these fields in memory, but any changes made are not saved permanently to disk until the Commit Changes button at the top of the window is clicked. Furthermore, the server will, in general, ignore the committed changes until it is restarted.

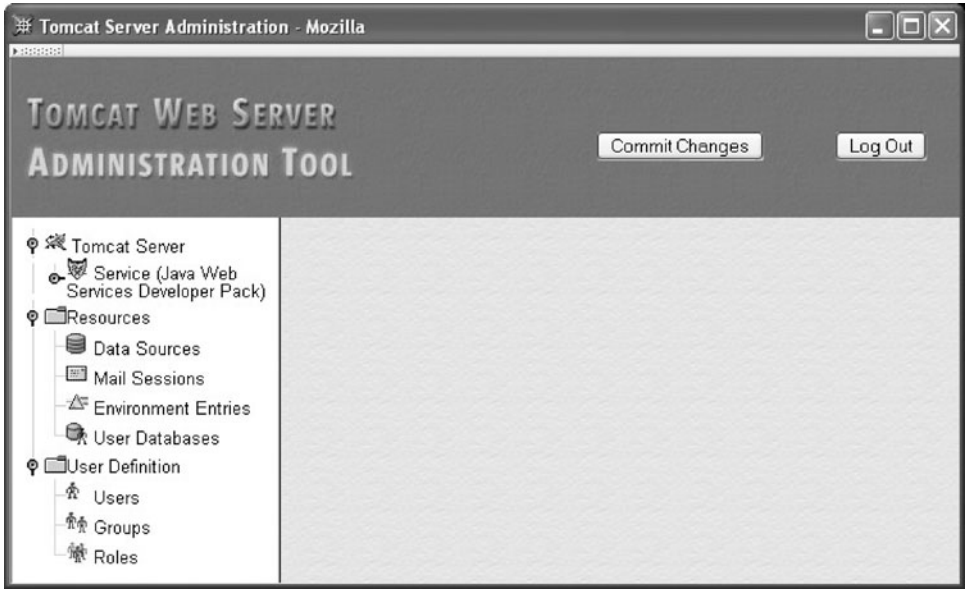

FIGURE 1.6 Tomcat administration tool entry page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

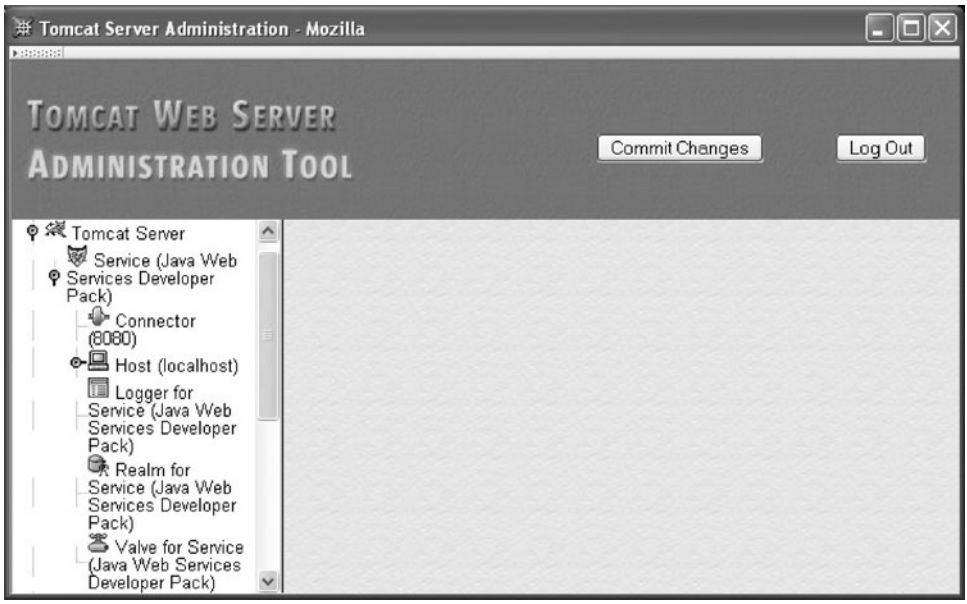

FIGURE 1.7 List of Service components produced by clicking on Service "handle" icon. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

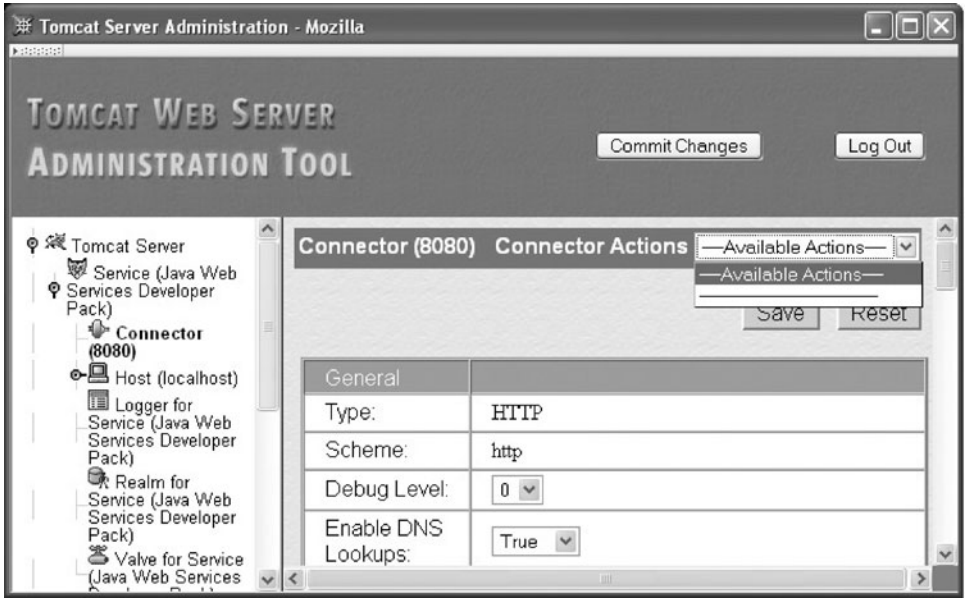

FIGURE 1.8 Connector edit page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

Some of the data fields in a panel, such as Edit Connector, have fixed values, while others can be edited (if we were creating a Connector, all fields would be editable). Some of the key fields for the Connector component type are listed in Table 1.10. Notice that the Port Number field value (8080 in this example) is used as the name of the Connector in the Service list. This is because the Port Number value for this Connector will be unique to this Connector, since each IP port can "belong" to, at most, one application on a system. On the other hand, multiple Connectors can be associated with a single Service, so a Service can potentially be accessed through multiple ports.

1.7.4 Defining Virtual Hosts

The Host component (Figure 1.9) is used to define a virtual host. Some of the key fields are described in Table 1.11. The virtual host name should normally be a fully qualified domain name that would be used by visitors to your web site, although the Host supplied as part of the JWSDP Service is given the unqualified name `localhost`. This is a special name that the DNS system treats as a reference to a special IP address, 127.0.0.1. If an IP message is sent to this address, the IP software causes the message to loop back to itself for receipt. In short, browsing to a URL with domain name `localhost` causes the browser to send the HTTP request to a web server on the machine running the browser. So it would seem that this virtual host should only be accessible if the browser runs on the server machine. However, clicking on the JWSDP Service link in the left panel reveals (in the right panel) that the value of the Default Hostname field for this Service is `localhost`. This means that if a user browses to this Service using a URL with a host name other than `localhost`, the request will be passed to the `localhost` virtual host. In essence, this Host component will respond to any HTTP request sent to the Service's Connector (at port 8080), regardless of the value of the request's Host header field.

TABLE 1.10 Some of the Fields for the Connector Component

Field Name	Description
Accept Count	Length of the TCP connection wait queue.
Connection Timeout	Server will close connection if it is idle for this many milliseconds.
IP Address	Blank indicates that this Connector will accept TCP connections directed to any IP address associated with this machine. Specifying an address restricts connections to requests for that address.
Port Number	Port number on which this Connection will listen for TCP connection requests.
Min Spare Threads	Initial number of threads that will be allocated to process TCP connections associated with this Connector. Once connections are established with the Connector, the server will maintain at least this many *idle* processing threads, that is, threads waiting for new connections but otherwise unused.
Max Threads	Maximum number of threads that will be allocated to process TCP connections associated with this Connector.
Max Spare Threads	Maximum number of idle threads allowed to exist at any one time. The server will begin stopping threads if the number of idle threads exceeds this value.

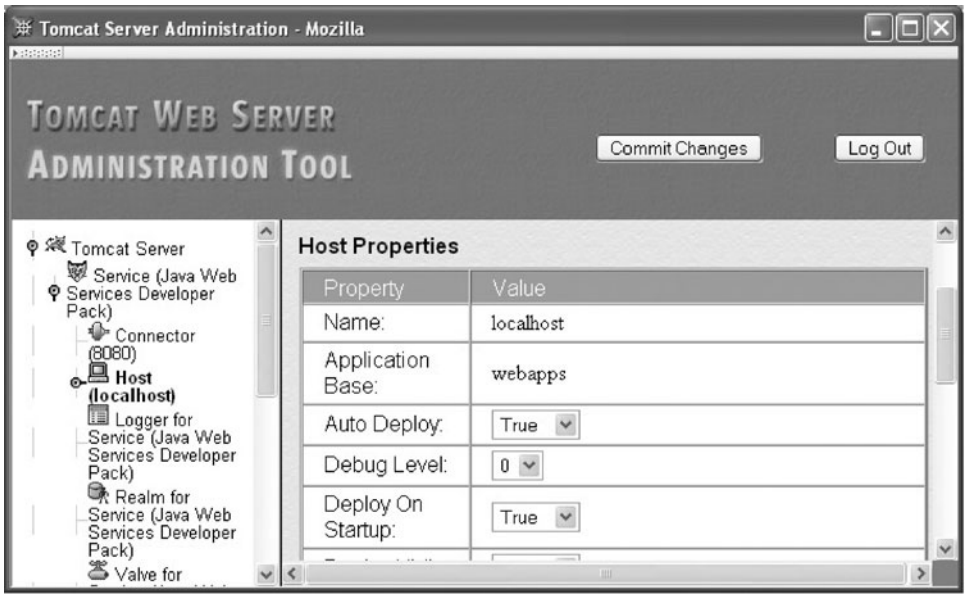

FIGURE 1.9 Host component panel for the JWSDP Service. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

Now let's assume an additional Host component with name, say, `www.example.org` was added to this Service (through the Tomcat Administration Tool by clicking on the Service in the left panel of the web page and then selecting the Create New Host item from the Service Actions menu in the right panel). Then this new virtual host would handle requests containing a Host header field with value `www.example.org`, while all requests with any other Host value would continue to be handled by the default `localhost` virtual host.

Several of the fields listed in Table 1.11 are associated with web applications. A *web application* is a collection of files and programs that work together to provide a particular function to web users. For example, a Web site might run two web applications: one for

TABLE 1.11 Key Fields for Host Component

Field Name	Description
Name	Usually the fully qualified domain name (or `localhost`) that clients will use to access this virtual host.
Application Base	Directory containing *web applications* for this virtual host (see text).
Deploy on Startup	Boolean value indicating whether or not web applications should be automatically initialized when the server starts.
Auto Deploy	Boolean value indicating whether or not web applications added to the Application Base while the server is running should be automatically initialized.

use by administrators of the site that provides maintenance functionality, and another for use by external clients that provides customer functionality. In Tomcat, a web application is represented by a Context component. Clicking on a Host handle icon will reveal the list of Contexts provided with that virtual host. If you open the `localhost` Host, you will find that it has several contexts predefined (Figure 1.10).

Each Host and Context is associated with a directory in the server's file system. The directory associated with a Host is specified by the value of the Application Base field. If this value is a *relative pathname*—a pathname that does not begin with a `/` in Linux or with a drive specification such as `C:\` in Windows—then it is taken as relative to the directory in which JWSDP 1.3 (and therefore Tomcat 5.0) was installed. For example, on my Linux machine I installed JWSDP 1.3 at `/usr/java/jwsdp-1.3`, so the relative pathname `webapps` given in Figure 1.9 corresponds on my machine to the directory `/usr/java/jwsdp-1.3/webapps`. [I will normally use forward slash (`/`) as the separator in file paths; change this to backslash (`\`) if you are using Windows.] This is known as the *absolute pathname* for the directory, and could have been specified instead of the relative pathname. Using a relative pathname for the Application Base value is generally recommended, since this allows your JWSDP 1.3 installation to be moved to another location within the server file system without the need to change the Application Base value.

The directory associated with a Context is specified by the value of the Document Base field (Figure 1.10). The figure shows an absolute pathname value (on a Windows system), but again the pathname can be relative instead. However, if a relative pathname is specified, it will be relative to the Application Base, not relative to the JWSDP 1.3 installation directory. So, assuming that the Application Base is at `C:\jwsdp-1.3\webapps`, the Document Base in Figure 1.10 could have been specified as simply `ROOT`. If you create a Context (by selecting Create New Context from the Host Action menu for a Host), be sure to create the directory that will be specified in the Document Base field before clicking the Save button for the Context.

As we will discuss in some detail in Chapter 8, a Context associates certain URLs with the specified Document Base. Figure 1.10, for example, shows that the root URL path (`/`) is associated with a directory named ROOT. And, in fact, if you examine the `webapps/ROOT` directory of your JWSDP 1.3 installation, you will find a file `THIRDPARTYLICENSEREADME.html` that contains the text (and some other information, discussed in the next chapter) that is displayed when you navigate to `http://localhost:8080/THIRDPARTYLICENSEREADME.html` (or the equivalent URL for your server). Similarly, if you navigate to `http://localhost:8080/`, you will see the contents of the `webapps/ROOT/index.html` file. This is because the server by default displays certain "welcome" files (such as `index.html`) if you do not explicitly specify a file name at the end of the path portion of the URL used to visit the server. What's more, navigating to `http://localhost:8080/servlets-examples` will display the contents of the `webapps/servlets-examples/index.html` file, because (as you can verify by clicking on the `/servlets-examples` Context object) the Document Base for URLs with paths beginning with `/servlets-examples` is the `webapps/servlets-examples` subdirectory of your JWSDP 1.3 installation. Note that the URL path for the Context object is specified using the Path field of the edit page.

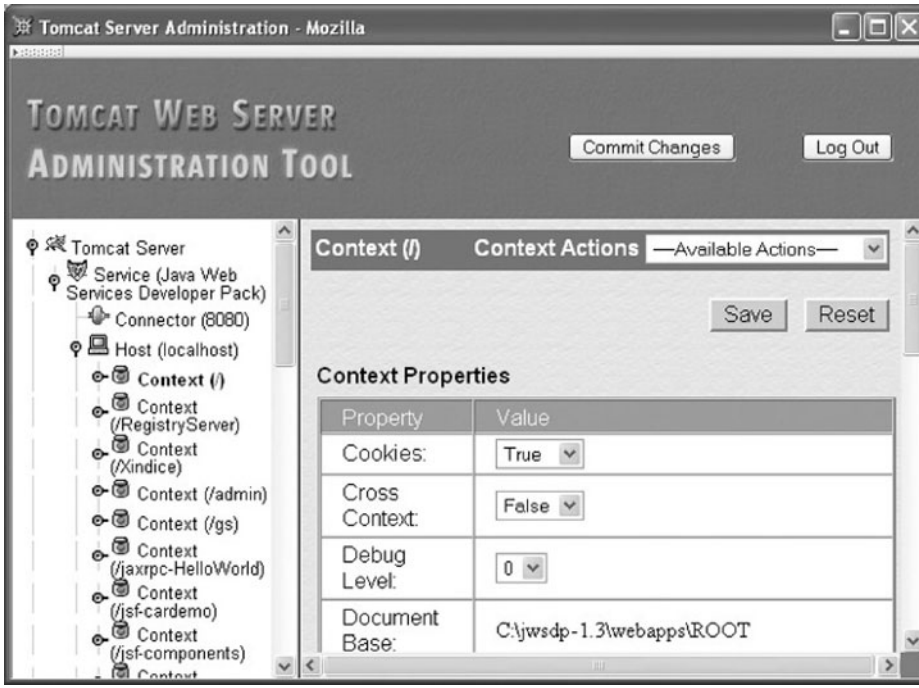

FIGURE 1.10 Context edit page. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

This brief introduction to virtual host concepts is intended to provide you with enough information to be able to set up your own virtual host that will serve simple text files. Again, we will have much more to say about associating URLs with server resources in later chapters. For now, we will move on to some other server capabilities.

1.7.5 Logging

Web server *logs* record information about server activity. The primary web server log recording normal activity is an *access log*, a file that records information about every HTTP request processed by the server. A web server may also produce one or more *message logs* containing a variety of debugging and other information generated by web applications as well as possibly by the web server itself. Finally, information written to the standard output and error streams by the web server or applications may also be logged. We will cover Tomcat's handling of these types of logs as well as some general logging concepts in this subsection.

Access logging in Tomcat is performed by adding a Valve component to a Service. For example, Figure 1.7 shows that the JWSDP Service includes a Valve, and if you click on it, you will find that it is of type AccessLogValve (some other types of Valves are discussed in the next subsection). The primary fields for an AccessLogValve are shown in Table 1.12.

TABLE 1.12 Key Fields for Valve Component of Type AccessLogValve

Field Name	Description
Directory	Directory (relative to Tomcat installation directory or absolute) where log file will be written
Pattern	Information to be written to the log (see text)
Prefix	String that will be used to begin log file name
Resolve Hosts	Whether IP addresses (`False` value) or host names (`True` value) should be written to the log file
Rotatable	Whether or not date should be added to file name and file should be automatically rotated each day
Suffix	String that will be used to end log file name

The combination of values for the Directory, Prefix, Rotatable, and Suffix fields determine the file system path to the access log. The JWSDP Service settings for the values of these Valve fields cause the access log for this Service to be written to the `logs` directory under the JWSDP 1.3 installation directory in a file that starts with the string `access_log.` and ends with the string `.txt`. In between these strings, because Rotatable is given the value `True`, Tomcat inserts the current date, in YYYY-MM-DD (year-month-day) format. So an example JWSDP access log name might be `access_log.2005-07-20.txt`. If you have started and browser to your Tomcat server, you should see one or more files of this form in the `logs` directory under your JWSDP 1.3 installation directory.

The Tomcat server writes one line of information per HTTP request processed to the access log, with the information to be output and its format specified by the Pattern field. The Pattern for the JWSDP Service access log Valve is

```
%h %l %u %t "%r" %s %b
```

This corresponds to what is often called the *common* access log format (in fact, the word `common` can be specified as the value of the Pattern field to specify this log format). The following is an example access log line in common format (this example is split into two lines for readability):

```
www.example.org - admin [20/Jul/2005:08:03:22 -0500]
"GET /admin/frameset.jsp HTTP/1.1" 200 920
```

The following information is contained in this log entry:

- Host name (or IP address; see Table 1.12) of client machine making the request
- User name used to log in, if server password protection is enabled (user "admin" logged in here)
- Date and time of response, plus the time zone (offset from GMT) of the time
- Start line of HTTP request (quoted)
- HTTP status code of response (200 in this example)
- Number of bytes sent in body of response

The Tomcat 5.0 server always returns the hyphen character (-) as the value of the %l pattern.

An advantage of using this log format is that a variety of *log analyzers* have been developed that can read logs in this (and some other) formats and produce reports on various aspects of a site's usage. For example, a log analyzer might report on the number of accesses per day, the percentage of requests that received error status codes, or a breakdown of accesses by domain. Such information can be useful for server tuning, locating software problems, or modifying site content to better target a desired audience. Analog (http://www.analog.cx) is one popular free log analyzer available at the time of this writing.

Another standard log format can be obtained by specifying the value combined for the Pattern. The combined format is the same as the common format but also has the Referer and User-Agent HTTP header field values appended. Custom log formats can also be created; see the section on the Valve component in the Tomcat 5 Server Configuration Reference [APACHE-TOMCAT-5-CONFIG] for details.

The Tomcat Logger component can be used to create a message log for a Service such as the JWSDP service (see Figure 1.7). A message log records informational, debugging, and error messages passed to logging methods by either servlets or Tomcat itself. Some of the key fields for File Loggers (the standard type of message log) are described in Table 1.13.

The JWSDP service sets the values of these fields so that the message log produced is written to the logs directory under the JWSDP 1.3 installation directory in a file that starts with the string jwsdp_log. followed by the date (this is not an option for message logs in Tomcat) and ends with the string .txt. If you look at the contents of one of these files, you will see lines such as

```
2005-08-02 07:38:54 createObjectName with StandardEngine[Catalina]
```

Because the JWSDP service has its Timestamp property set to true, the beginning of each message log entry begins with a *timestamp*, that is, with the date and time at which the entry was written to the log. Timestamps can be useful, particularly when trying to debug an application. One thing to be aware of when using timestamps is that some applications may write timestamps in universal (GMT) time, whereas others, including Tomcat, use local time.

Loggers can be associated with different levels of the Tomcat object hierarchy: with a Service (such as JWSDP, the example just given); with a Host within a Service (such as localhost); and even with a Context, or web application, within a Host (such as admin,

TABLE 1.13 Key Fields for Logger Component of Type File Logger

Field Name	Description
Directory	Directory (relative to Tomcat installation directory or absolute) where log file will be written
Prefix	String that will be used to begin log file name
Suffix	String that will be used to end log file name
Timestamp	Whether or not date and time should be added to beginning of each message written to the log file

the web application that implements the Tomcat administration tool). For example, if you examine the `admin` Context under the `localhost` Host using the Tomcat administration tool, you will see that this Context has its own Logger that produces files beginning with `localhost_admin_log.`, also within the `logs` directory. Messages sent to logging methods by servlets within the `admin` web application will go to this message log file rather than to the JWSDP Service's logger. In general, logging methods will search for a Logger beginning in the Context, then the Host, and finally the Service, sending the log message to the first Logger found. Access logs in Tomcat can also be associated with different levels of the server object hierarchy, although typically there is only one access log per service.

Finally, Tomcat itself or servlets it runs may write directly to the Java standard output and error streams `System.out` and `System.err`. The JWSDP 1.3 installation of Tomcat redirects both of these streams to a file named `launcher.server.log` in the `logs` subdirectory of the JWSDP 1.3 installation directory. Thus, if you write an application that prints an exception stack trace, this is likely where you will find it.

1.7.6 Access Control

Tomcat can provide automatic password protection for resources that it serves. At its heart, this is a two-stage process. First, a database of user names is created. Each user name is assigned a password and a list of *roles*. Think of a role as a user's functional relationship to a web application: administrator, developer, end user, etc. Some users may be assigned to multiple roles. The second stage is to tell Tomcat that certain resources can only be accessed by users who belong to certain roles and who have authenticated themselves as belonging to one of these roles by logging in with an appropriate user name and password. For example, the Tomcat administration tool application (`admin` Context) can only be accessed by users who have logged in and who belong to the admin role.

The second stage of this process—associating resources with required roles—is normally performed by web application developers, as described in Section 8.3.3. The first stage—defining one or more user databases—can be performed by web system administrators, application developers, or both. The JWSDP Service contains an example of a database defined at the Service level through the use of a Realm component, which associates a user database with a Service (Figure 1.11). This particular type of Realm indicates that a Tomcat Resource—an object representing a file or other static resource on the server—will be used to store the user database. The Realm's Resource Name field contains the name of the Resource, which in this case is UserDatabase. If you click on the User Databases link in the Resources list in the left panel of your Tomcat administration tool window and then click on the UserDatabase link in the User Databases panel, you will see that this Resource is associated with a file located at `conf/tomcat-users.xml` (this is relative to the Tomcat installation directory). The administration tool also automatically loads the contents of this file under the User Definition folder in the left panel. Clicking on the Users link under this folder shows that there is one user name in this user database: the user name that you chose for the Tomcat administrator when you installed Tomcat. Finally, clicking on this user name in the right panel shows the roles to which a user logged in with this user name belongs: admin and manager.

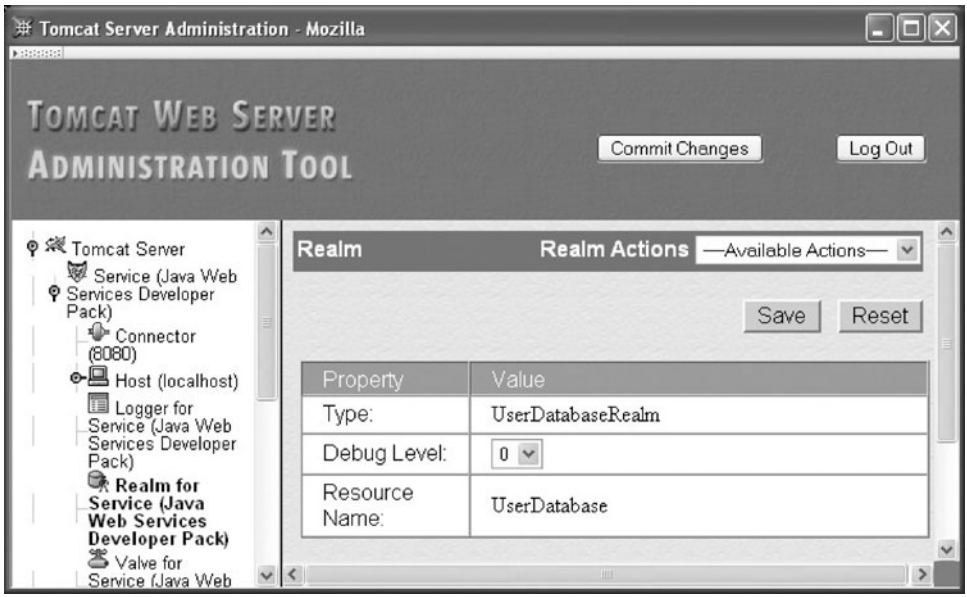

FIGURE 1.11 Realm component panel for the JWSDP Service. The content of this screen shot is reproduced by permission of the Apache Software Foundation.

As mentioned, the admin role is the role required to run the Tomcat administration tool. So, if you wanted to allow another user to run this tool (and other web applications accessible in the admin role), you would simply create that user by selecting Create New User from the Actions dropdown menu of the Users panel and be sure to check the admin role for that user.

A coarser-grained access control can be provided by using Valve objects of type RemoteHostValve and RemoteAddressValve. Both are used to specify client machines that should be rejected if they request a connection to the server. They differ only in whether client machine host names or IP addresses are specified. Each type of Valve has two possible lists of clients: an Allow list and a Deny list. If one or more host names (comma-separated) is entered in the Allow list, then only these hosts can access the server. You can use the `*` wildcard in place of any label within a host name. So, for example, to allow access only from machines in the `example.org` and `example.net` domains, you would enter in the Allow list

```
*.example.org,*.example.net
```

In addition, whether or not any names are entered in the Allow list, any hosts (possibly wildcarded) entered in the Deny list will be prevented from accessing the server. So, to exclude a single machine from the `example.org` domain while allowing all of the others, we might enter in the Deny list something like

```
baduser.example.org
```

1.7.7 Secure Servers

Normally, the HTTP request and response messages are sent as simple text files. Because these messages are carried by TCP/IP, each message may travel through a number of machines before reaching its destination. It is possible that some machine along the route will extract information from the IP messages it forwards for nefarious purposes. Furthermore, it is often possible for other machines sharing a local network with the sending or receiving machine to snoop the network and view messages associated with other machines as if they were sent to the snooper. In general, any machine other than the sender or receiver that extracts information from network messages is known as an *eavesdropper*.

To prevent eavesdroppers from obtaining sensitive information, such as credit card numbers, all such sensitive information should be *encrypted* before being transmitted over any public communication network. The standard means of indicating to a browser that it should encrypt an HTTP request is to use the `https` scheme on the URL for the request. For example, entering the URL

```
https://www.example.org
```

in Mozilla's Location bar will cause the browser to attempt to send an encrypted HTTP GET request to `www.example.org`.

Various protocols have been used to support encryption of HTTP messages. Many browsers and servers support one or more versions of the Secure Socket Layer (SSL) protocol as well as the newer Transport Layer Security (TLS) protocol, which is based on SSL 3.0. The following description of HTTP encryption is derived from the TLS 1.0 specification [RFC-2246], but the same general ideas apply to the earlier SSL protocols as well.

A client browser that wishes to communicate securely with a server begins by initiating (over TCP/IP) a *TLS Handshake* with the server. During the Handshake process, the server and client agree on various parameters that will be used to encrypt messages sent between them. The server also sends a *certificate* to the client. The certificate enables the client to be sure that the machine it is communicating with is the one the client intends (as specified by the host name in the URL the browser is requesting). Certificates are necessary to avoid so-called *man-in-the-middle attacks*, in which some machine intercepts a message intended for another machine (the target), prevents the message from further forwarding, and returns an HTTP reply to the sender pretending to be from the target. Such an interception could occur at a rogue Internet bridge device on the route between client and server, or through unauthorized alteration of the DNS system, for example.

At the conclusion of the TLS Handshake, the client uses the cryptographic parameter information obtained to encrypt its HTTP request message before sending it to the server over TCP/IP. The server's TLS software decrypts this request before any other server processing is performed. The server similarly encrypts its response before sending it to the client, and the client immediately decrypts the received message. Therefore, other HTTP-processing software running on the client and server are, for the most part, unaffected by the encryption process.

One small point involves the port used for the TCP/IP communication of TLS data. Since the TLS protocol begins with a TLS Handshake, and not with an HTTP request start line, different communication ports are used for the two types of communication. Whereas the default port for HTTP communication is 80, the default for TLS/SSL is 443. This port can be overridden just as the HTTP port can be overridden, by explicitly adding a port number after the host name in an `https`-scheme URL. So, for example, to access the root of a secure server on localhost at port 8443, you would use the URL

```
https://localhost:8443/
```

Tomcat supports the TLS 1.0 and earlier protocols. To enable the secure server Tomcat features, you must do two things:

1. Obtain and install a certificate.
2. Configure the server to listen for TLS connections on some port.

For test purposes, you can generate your own "self-signed" certificates using the `keytool` program distributed with Sun Java™ JDK™ development software. This program is located in the same directory as the `javac` and `java` programs. Assuming that this directory is included in your PATH environment variable, you can begin to create a self-signed certificate suitable for use with Tomcat 5.0 by entering the following at a command prompt:

```
keytool -genkey -alias tomcat -keyalg RSA
```

This says that you want to generate a self-signed certificate that can be referenced by the name `tomcat` and that the encryption/decryption keys generated for use with this certificate should be compatible with the RSA encryption/decryption algorithm (which is the algorithm Tomcat uses). You will be prompted to enter several pieces of information. Since this certificate is self-signed and will be used for test purposes only, for the most part, it does not matter what you enter. However, I suggest entering the fully qualified domain name of your machine when asked to enter your first and last name, as this will prevent a warning later when you try to use the certificate. Also, I suggest using the password `changeit` when asked, which will allow you to use defaults when you configure the server to use this certificate (but use this password for testing purposes only).

Configuring the server to listen for TLS connections simply involves adding a second Connector to a Service (by selecting Create New Connector from the Service's Action drop-down menu). The Type field of the new Connector must be set to HTTPS. On the resulting Connector panel, make sure that the Secure field is set to `True` (since this is a secure connection), and fill in the port number (say 8443) to be used for this connection. Other fields can retain their default values if you run `keytool` with its defaults. After Saving and Committing the changes made in order to create your new Connector, stop your server. If you have not already performed the JWSDP 1.3 postinstallation tasks described in the appendix (Section A.4.2), do so at this time. Now restart your Tomcat server, close and reopen your browser, and then browse to `https://localhost:8443` (modify this as appropriate for the host name and port number for your secure server). If you created a self-signed certificate,

you should see a message asking you whether or not you wish to accept the certificate. After accepting it, you should see the default JWSDP web page produced by your server. Note that a small padlock icon at the bottom of your browser window is shown locked, indicating that the page is being viewed securely.

Since there is no independent validation of self-signed certificates, anyone can generate a self-signed certificate for your machine. This is why browsers will typically display a warning message if a self-signed certificate is presented by a server: while it is syntactically a certificate, it does not prevent a man-in-the-middle attack, because an attacker could easily have generated the certificate. In order for your server to provide transparent secure communication using certificates that browsers will trust automatically, you must have your certificate verified and then digitally "signed" (for a fee) by a certificate authority, such as VeriSign. Details are provided in the SSL section of the Tomcat User Guide [APACHE-TOMCAT-5-UG].

1.8 Case Study

To provide some context for the various technologies we will be covering, an ongoing case study will be part of most chapters. Specifically, we'll create a simple tool for writing and reading a web log (*blog*). One user, the *blogger*, will be able to add text entries to the blog. The most recent entry will appear at the beginning of a web page, followed by the next most recent, and so on for all entries made during the current month. Links elsewhere on the page will provide access to entries made in earlier months (Figure 1.12). Other capabilities will be described in later chapters.

Although we're not ready to start developing any software for this application, we can make some decisions related to the material covered in this chapter:

- Which browsers will we support?
- Which web server(s) will we use?
- How extensive will our security measures be?

At the time of this writing, if our application runs well with IE6 and Mozilla-like browsers, then we will have covered a large percentage of browsers in use, so we will test our application against Mozilla 1.4 and IE6. We will use the Tomcat server distributed with JWSDP 1.3 because it is freely available, runs on multiple platforms, is simple to configure (compared with running, say, both Apache and Tomcat), is sufficiently fast for our needs, and supports the technologies that we will be covering in later chapters.

The question of security is somewhat more difficult. The key security task for the case study application is to prevent everyone but the blogger from adding entries to the blog. A preceding task—one that is beyond the scope of this textbook—is to make sure that the machine running the web server is itself secure from unauthorized access. Obviously, if someone can gain administrative privileges on the server machine, then no amount of work we put into securing the application itself will make it truly secure.

Assuming a secure server machine, the weakest level of security would be to have a "secret" URL that the blogger visits in order to add an entry to the blog. This approach is open to several attacks, one of which is to simply try a variety of reasonable URLs. For

CHAPTER 3

Style Sheets
CSS

As we have learned, HTML markup can be used to indicate both the *semantics* of a document (e.g., which parts are elements of lists) and its *presentation* (e.g., which words should be italicized). However, as noted in the previous chapter, it is advisable to use markup predominantly for indicating the semantics of a document and to use a separate mechanism to determine exactly how information contained in the document should be presented. *Style sheets* provide such a mechanism. This chapter presents basic information about *Cascading Style Sheets* (CSS), a style sheet technology designed to work with HTML and XML documents.

CSS provides a great deal of control over the presentation of a document, but to exercise this control intelligently requires an understanding of a number of features. And, while you as a software developer may not be particularly interested in getting your web page to look "just so," many web software developers are members of teams that include professional web page designers, some of whom may have precise presentation requirements. Thus, while I have tried to focus on what I consider key features of CSS, I've also included a number of finer points that I believe may be more useful to you in the future than you might expect on first reading.

While CSS is used extensively to style HTML documents, it is not the only style-related web technology. In particular, we will study the Extensible Stylesheet Language (XSL)—which is used for transforming and possibly styling general XML documents—in Chapter 7.

3.1 Introduction to Cascading Style Sheets

Before getting into details, let's take a quick look at an XHTML document that uses simple style sheets to define its presentation. Specifically, let's consider once again the "Hello World!" document of Figure 2.1, but with the addition of two `link` elements in the `head` of the document (`CSSHelloWorld.html`, shown in Fig. 3.1). Notice that the body of this document is identical to that of Figure 2.1. However, viewing this document in Mozilla 1.4 produces the result shown in Figure 3.2, which is quite different from the way Mozilla displayed the original "Hello World!" document (Fig. 2.2).

The difference between the two browser renderings, of course, has to do with the `link` element, which imports a *style sheet* located at the URL specified as the value of its `href` attribute. In this example, the style sheet is written in the CSS language, as indicated by the MIME type value of the `type` attribute. The `style1.css` file contains the lines

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      CSSHelloWorld.html
    </title>
    <link rel="stylesheet" type="text/css" href="style1.css"
          title="Style 1" />
    <link rel="alternate stylesheet" type="text/css" href="style2.css"
          title="Style 2" />
  </head>
  <body>
    <p>
      Hello World!
    </p>
  </body>
</html>
```

FIGURE 3.1 HTML source for "Hello World!" using style sheets.

```
body  { background-color:lime }
p     { font-size:x-large; background-color:yellow }
```

The first line simply says that, for rendering purposes, the `body` element of the document should be treated as if it contained the attribute `style="background-color:lime"`. The second line is similar, except that it specifies a style that should be applied to every `p` element of the document. The second line also specifies values for two different style *properties*, `font-size` and `background-color`. We'll learn details about these and many other style properties later in this chapter, but for now their meaning should be clear from their names and the effects shown in Figure 3.2.

The file `style2.css` contains the single line

```
p     { font-size:smaller; letter-spacing:1em }
```

This says that `p` elements should be set in a smaller than normal font size and that there should be space between adjacent letters. However, this style is not applied to the document

FIGURE 3.2 Browser rendering of `CSSHelloWorld.html`.

FIGURE 3.3 Selecting the style sheet to be used by Mozilla.

rendered in Figure 3.2, because this style sheet is specified as an *alternate style sheet* by the `rel` (relationship) attribute of the `link` element that imports this sheet. A style sheet such as the one in `style1.css`, which is referenced by a `link` element having a `rel` with value `stylesheet` as well as a `title` specification, is known as a *preferred style sheet*. An alternate sheet can be selected by the user, as illustrated in Figure 3.3. Notice that the values of the `title` attributes of the `link` tags are displayed in the Use Style menu along with the default Basic Page Style; preferred and alternate style sheet `link` elements must always contain `title` attribute specifications. After the alternate style sheet is selected, the page renders in the second style, as shown in Figure 3.4. (Alternate style sheets are not used often at the time of this writing, because the user interface for IE6 does not support user selection of alternate style sheets.)

Now that we have some understanding of what a style sheet is, we will discuss some of the major features of CSS.

3.2 Cascading Style Sheet Features

The key property of style sheet technology is that it can be used to separate the presentation of information from the information content and semantic tagging. The content and

FIGURE 3.4 Browser rendering of `CSSHelloWorld.html` using style sheet from `style2.css`.

TABLE 3.1 Possible Values for `media` Attribute Defined by HTML 4.01 Standard

Value	Media Type
`all`	All types (default)
`aural`	Speech synthesizer
`braille`	Tactile device generating braille characters
`handheld`	Handheld device, such as a cell phone or PDA
`print`	Printer
`projection`	Projector, such as one used to display a large monitor image on a screen
`screen`	Computer monitor
`tty`	Fixed-width character output device
`tv`	Television (monitor with low resolution and little or no scrolling)

semantics of the "Hello World!" page did not change in the previous example: it consisted of a single paragraph containing some text. Put another way, the `body` elements of the `HelloWorld.html` and `CSSHelloWorld.html` files will have exactly the same abstract syntax tree. But by changing the style sheet used by the browser to display this tree, we can achieve different presentations of the same information.

There are significant advantages to having such a separation between the information contained in a document and its presentation. First, it allows the information in the document to be presented without change in a variety of ways. We have already seen an example of this feature with user-selectable alternative style sheets. But CSS can do even more than this. For example, the `link` element defines a `media` attribute that can be used to define the types of media for which a style sheet is designed, such as for display on a monitor or output to a printer (see Table 3.1 for a complete list of media types defined by the HTML 4.01 standard). So, for example, if we had used the `link` elements

```
<link rel="stylesheet" type="text/css" href="style1.css"
      media="screen, tv, projection" />
<link rel="stylesheet" type="text/css" href="style2.css"
      media="handheld, print" />
```

then the style sheet of `style1.css` would be used for display on monitors, televisions, and projectors, the style sheet of `style2.css` for output to handheld devices and printers, and the browser's default style sheet for all other forms of output. (The example file `CSSHelloWorldPrint.html` demonstrates this feature: try loading it into your browser and then printing it.) You'll notice that the `title` attribute does not appear in the `link` elements in this example. This is because these style sheets cannot be selected by the user, but instead will apply regardless of user actions. Such style sheets are called *persistent* and can be recognized by their lack of a `title` attribute specification in the `link` element referencing the style sheet.

From a developer's perspective, another useful feature of using style sheets is that it is relatively easy to give all of the elements on a page a consistent appearance. That is, if we want all of the `h1` headers on a page to have a certain size, we can accomplish this easily

using a style sheet. Furthermore, if at a later time we wish to change the size of the headers, we need only make the change in that one style sheet. More generally, if we use a single style sheet for all of the pages at a site, then all of the site pages will have a consistent style, and one that can be changed with little work.

In addition to these properties, which apply to any style sheet language—including older print-oriented style sheet languages—the cascading quality of CSS makes it particularly appealing for use with web documents. As we will learn, both the document author and the person viewing the document can specify aspects of the document style as it is displayed by the browser (or other user agent displaying the document). For example, a user may instruct their browser to display all HTML documents using only a white background, regardless of the setting of the `background-color` property in style rules supplied by the page author. This can be an important feature to, for example, a user who because of an eyesight limitation needs high contrast between text and its background.

It should also be noted that, though I am going to cover CSS in the context of providing style for HTML documents, it can also be used with non-HTML XML documents (Section 7.10 contains an example).

So, there are many reasons to learn about style sheet technology in general, and CSS in particular. We'll start by covering some of the core CSS syntactic elements. After that, we'll study the cascading aspects of CSS in more detail. Finally, we'll consider details of a number of specific style properties and apply CSS to the blogging case study.

3.3 CSS Core Syntax

As with HTML, there are several W3C-recommended versions of CSS. At the time of this writing, there are technically two W3C recommendations for CSS: CSS level 1 [W3C-CSS-1.0] and CSS level 2 [W3C-CSS-2.0] (often referred to as CSS1 and CSS2). Work is also underway on CSS level 3, and several specialized versions of CSS for limited devices, such as cell phones, low-cost printers, and televisions, are in various stages of standardization.

Although CSS2 has been a W3C recommendation since 1998, at this time no widely used browser implements the entire recommendation. Recognizing this fact, the W3C has been developing CSS 2.1, which is largely a scaled-back version of CSS2 that attempts to capture those features of CSS2 that are—as of the time of the recommendation's official publication—implemented by multiple browsers. Using the February 2004 candidate version of CSS 2.1 [W3C-CSS-2.1] as a guide, in this chapter I will specifically focus on key aspects of CSS2 that are implemented in both IE6—the latest generally-available version of Internet Explorer at the time of the writing—and Mozilla 1.4. For the most part, the basic CSS syntax is the same for both levels 1 and 2, so much of what is presented should also be compatible with older browsers. Furthermore, just as browsers generally ignore HTML elements that they do not recognize, they also generally ignore CSS style properties that they do not recognize. So, if you use CSS as described in this chapter, almost all browsers should be able to display your document (although some older ones may not style it properly). It will of course be advisable for you to monitor the progress of the CSS 2.1 and CSS 3 recommendations so that you can use newer style sheet features as they become widely available; see the References section (Section 3.12) for more on this.

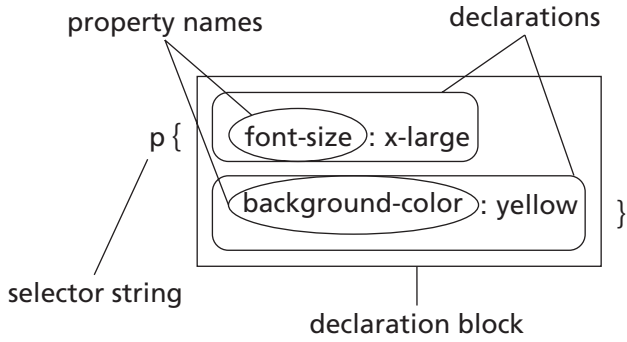

FIGURE 3.5 Parts of a single ruleset-type style rule.

One other word of warning is that versions of the Internet Explorer browser before IE6 supported style sheets but deviated from the CSS recommendation in several ways. Even in IE6, these deviations will be present unless you use a document type declaration such as the one for XHTML 1.0 Strict used in our examples. At the time of this writing, IE5 is still used on a substantial number of machines, although its usage is dwindling rapidly. So, if you develop real-world CSS style sheets in the near term, you may need to deviate somewhat from the material presented in this chapter. However, the concepts taught here are similar to those found in IE5, and as time goes on the details presented here should apply to the bulk of browsers in use. Again, see Section 3.12 for more information.

A CSS style sheet consists of one or more style rules (sometimes called *statements*). Each line in the `style1.css` file in Section 3.1 is an example of a rule. This form of rule is called a *ruleset* and consists of two parts: a *selector string* followed by a *declaration block*, which is enclosed in curly braces (`{` and `}`) (see Fig. 3.5). The declaration block contains a list (possibly empty) of *declarations* separated by semicolons (;) (the final declaration can also be followed by a semicolon, and many style sheet authors follow this convention). The selector string indicates the elements to which the rule should apply, and each declaration within the declaration block specifies a value for one style property of those elements. While the example shows one rule per line, it is syntactically legal to split a rule over several lines or (though not recommended) write multiple rules on a single line. No special character is needed to mark the end of a rule (no semicolon as in Java), due to the use of the braces to distinguish the parts of the rule.

We'll have much more to say about the properties that may be set within declarations in a later section. For the moment, the properties that we use, such as `color` (text color) and `font-style`, should be fairly self-explanatory. Before considering other properties, we will focus on selector strings.

3.3.1 Selector Strings

In the following paragraphs, we will be referring to an example style sheet and HTML document shown in Figure 3.6 and Figure 3.7, respectively. Notice that comments are written using the Java-style multiline syntax; HTML-style comments are not recognized in

```
/* Headers have dark background */

h1,h2,h3,h4,h5,h6 { background-color:purple }

/* All elements bold */

* { font-weight:bold }

/* Elements with certain id's have light background */

#p1, #p3 { background-color:aqua }

/* Elements in certain classes are italic, large font,
   or both */

#p4, .takeNote { font-style:italic }
span.special { font-size:x-large }

/* Hyperlink ('a' element) styles */

a:link { color:black }
a:visited { color:yellow }
a:hover { color:green }
a:active { color:red }

/* Descendant selectors */

ul span { font-variant:small-caps }
ul ol li { letter-spacing:1em }
```

FIGURE 3.6 Style sheet file `sel-demo.css` used to demonstrate various types of CSS selectors.

CSS, nor are Java end-of-line (`//`) comments. A browser rendering of this HTML document using the given style sheet is shown in Figure 3.8.

Probably the simplest form of selector string, which we have already seen, consists of the name of a single element type, such as `body` or `p`. A rule can also apply to multiple element types by using a selector string consisting of the comma-separated names of the element types. For example, the rule

```
h1,h2,h3,h4,h5,h6 { background-color:purple }
```

says that any of the six heading element types should be rendered with a purple background. Therefore, in our example document, the markup

```
<h1>Selector Tests</h1>
```

has a purple background when displayed in the browser.

In the preceding style rule, each of the *selectors* (comma-separated components of the selector string) was simply the name of an element type. This form of selector is called a *type selector*. Several other forms of selector are also defined in CSS. One is the *universal*

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Selectors.html
    </title>
    <link rel="stylesheet" type="text/css" href="sel-demo.css" />
  </head>
  <body>
    <h1>Selector Tests</h1>
    <p id="P1" class="takeNote">
      Paragraph with id="P1" and class="takeNote".
    </p>
    <p id="p2" class="special">
      Second paragraph.  <span class="takeNote special cool">This span
      belongs to classes takeNote, special, and cool.</span>

      <ul>
        <li>Span's within this list are in <span>small-cap</span>
            style.</li>
        <ol>
          <li>This item spaces letters.</li>
        </ol>
      </ul>
    </p>
    <p id="p3">
      Third paragraph (id="p3") contains a
      <a href="http://www.example.net">hyperlink</a>.
      <ol>
        <li>This item contains a span but does not display it in
          <span>small caps</span>, nor does it space letters.</li>
      </ol>
    </p>
  </body>
</html>
```

FIGURE 3.7 HTML document used to demonstrate various types of CSS selectors.

selector, which is denoted by an asterisk (`*`). The universal selector represents every possible element type. So, for example, the rule

```
* { font-weight:bold }
```

specifies a value of `bold` for the `font-weight` property of every element in the document.

Another form of selector is the *ID selector*. Recall that every element in an XHTML document has an associated `id` attribute, and that if a value is assigned to the `id` attribute for an element then no other element's `id` can be assigned the same value. If a selector is preceded by a number sign (`#`), then it represents an `id` value rather than an element type name. So, for example, if a document contains the markup

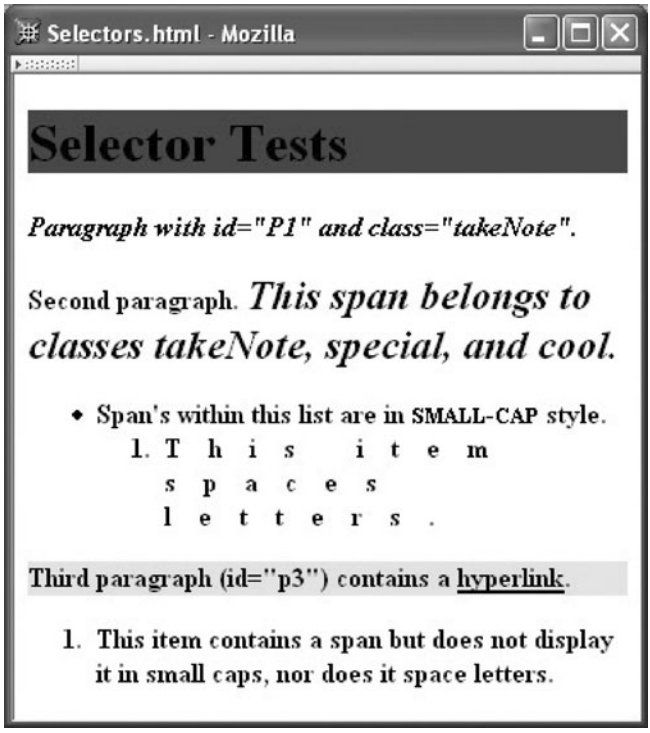

FIGURE 3.8 Browser rendering of `Selectors.html` after applying style sheet `sel-demo.css`.

```
<p id="p3">
  ...
</p>
```

then the following rule will cause this paragraph (and another element with `id` value `p1`, if such an element exists) to be displayed with an aqua background:

```
#p1, #p3 { background-color:aqua }
```

Note that `id` values are case-sensitive, so this rule will not apply to an element that has an `id` value of `P1`. This is why the first paragraph in Figure 3.8 does not have a background color.

Another HTML attribute that is frequently used with style sheets is `class`. This attribute is used to associate style properties with an element as follows. First, the style sheet must contain one or more rulesets having *class selectors*, which are selectors that are preceded by a period (`.`), such as `.takeNote` in the rule

```
#p4, .takeNote { font-style:italic }
```

Then any element that specifies `takeNote` (without the leading period) as the value of its `class` attribute will be given the properties specified in the declaration block of the corresponding style rule. Thus, the first paragraph of the example is displayed in an italic font. An element can be assigned to multiple style classes by using a space-separated list of class names as the value of the `class` attribute. For example, a `span` element with start tag

```
<span class="takeNote special cool">
```

will be affected by any rules for for the `takeNote`, `special`, and `cool` classes. Thus, the second sentence of the second paragraph of the example is italicized, since it belongs to the `takeNote` class, among others. If a class name does not correspond to a class selector in any of the style rules for a document (for example, `.cool` is not used as a class selector in `sel-demo.css`), then that class value is ignored.

Note that, like `id` values, `class` values are case sensitive and cannot begin with a decimal digit. However, unlike `id`, multiple elements can have the same value for their `class` attributes. All but a few elements, such as `html`, `head`, and elements that appear as content of `head`, have the `class` attribute.

ID and class selectors can also be prefixed by an element type name, which restricts the selector to elements of the specified type. For example, the style rule

```
span.special { font-size:x-large }
```

applies only to `span` elements that have a `class` value of `special`. So, in our example, the second paragraph itself is not set in the extra large (`x-large`) font size, but the second sentence of that paragraph is displayed using the extra large font, because the sentence is contained in a span with `class` value `special`. Also, an asterisk can be used in place of an element name in such a prefix, and (as with the universal selector) represents the set of all element names. In other words, the selectors `*.takeNote` and `.takeNote` are equivalent.

In addition to ID and class selectors, several predefined *pseudo-classes* are associated with `a` (anchor) elements that have an `href` attribute (source anchors). Table 3.2 lists these pseudo-class selectors. Figure 3.8 shows a link that has not been visited recently, and is therefore displayed in black. Positioning the cursor over that link without clicking the mouse button will cause the link to change to green, and clicking and holding the mouse button will change the color to red. If the link is visited, then the next time `Selectors.html` is

TABLE 3.2 Pseudo-Classes Associated with a Element Type

Selector	Associated a Elements
`a:visited`	Any element with `href` corresponding to a URL that has been visited recently by the user
`a:link`	Any element that does not belong to the `a:visited` pseudo-class
`a:active`	An element that is in the process of being selected; for example, the mouse has been clicked on the element but not released
`a:hover`	An element over which the mouse cursor is located but that does not belong to the `a:active` pseudo-class

loaded into the browser the link will be yellow. A fine point is that the current CSS 2.1 draft recommendation [W3C-CSS-2.1] allows a browser to ignore a pseudo-class style rule that would change the positioning of any elements within the browser. Color changes are therefore good choices as declarations for a rule that uses a pseudo-class selector, while even a seemingly innocuous declaration involving boldfacing should be used with caution (since boldfacing can increase the width of text and therefore move other elements).

Finally, a selector may be specialized so that it holds only within the content of certain element types. For example, the rule

```
ul span { font-variant:small-caps }
```

says that the text within a `span` element that is in turn part of the content of an unordered, or bulleted, list (`ul` element) should be displayed using a small-cap font form. Such a selector is known as a *descendant selector*. Notice that only the span within the bulleted list item in Figure 3.8 is displayed in the small-cap format.

Class selectors can also be included in the ancestor list; for example, the selector

```
.special span
```

would apply to any `span` element within the content of any element belonging to the class `special`. More generally, a white-space-separated list of element and/or class names may be used as a selector, representing a chain of elements each of which must be a descendant of the element to its left in order for the selector to apply. For example, the rule

```
ul ol li { letter-spacing:1em }
```

applies only to an `li` element within an `ol` (ordered, or numbered, list) element that is within a `ul` element. Thus, the numbered item in the second paragraph displays in the letterspaced format, because this paragraph's numbered list is contained within a bulleted list; but the numbered list in the third paragraph does not use this format, because it is not contained within a bulleted list.

3.3.2 At-Rules

So far, we have covered the ruleset form of style rules. The other form of rule is called an *at-rule*. The only at-rule that is widely supported and used at the time of this writing is the rule beginning with `@import`. This rule is used to input one style sheet file into another one. For example, a style sheet such as

```
@import url("general-rules.css");
h1, h2 { background-color: aqua }
```

will first read in rules from the file `general-rules.css` before continuing with the other rule in this style sheet. The `url()` function is used to mark its string argument as a URL. Single quotes can be used for this argument rather than double quotes; in fact, the quotes are not required at all. The URL can be absolute or relative. If it is a relative URL, like the one

shown in this example, then it will be taken as relative to the URL of the file containing the import at-rule, rather than relative to the HTML document. The `@import` rule must end with a semicolon, as shown. Also, all `@import` rules must appear at the beginning of a style sheet, before any ruleset statements.

3.4 Style Sheets and HTML

So far, the style sheets we have used have been stored in files and included in an HTML document through the use of a `link` element. Such style sheets are known as *external* style sheets. Another option is to embed a style sheet directly in an HTML document as the content of the HTML `style` element, which can appear any number of times in the `head` content of a document. For example, an XHTML document might contain the following markup:

```
<head>
  <title>InternalStyleSheet.html</title>
  <style type="text/css">
    h1, h2 { background-color:aqua }
  </style>
</head>
```

As you would expect, this will have the same effect as if the given style rule had been contained in an external style sheet and included in the HTML document via a `link` element. A style sheet that is included in the content of a `style` element is known as an *embedded* style sheet.

I have two notes of caution about using embedded style sheets. First, if any XML special character, such as less-than (<) or ampersand (&), appears in the style rules, then the character must be replaced by the appropriate entity or character reference. On the other hand, such references should *not* be used in an external style sheet, because an external style sheet is not an XML document and therefore is not processed like one. Second, the HTML 4.01 specification suggests enclosing the content of a `style` element within an SGML comment, for example,

```
<style type="text/css">
  <!--
  h1, h2 { background-color:aqua }
  -->
</style>
```

This was suggested because some older browsers did not recognize the `style` element. Such a browser would ignore the `style` start and end tags but would still attempt to process the content of the element, as discussed in Chapter 2. Therefore, a `style` element could produce strange behavior in such browsers. To circumvent this problem, CSS was defined so that the SGML comment start and end delimiters `<!--` and `-->` are ignored by style sheet processors (the delimiters themselves are ignored, but the content within the delimiters is not ignored). So an older browser would ignore both the `style` tags and the content in a `style` element

written as shown, while a style-cognizant HTML 4.01 browser would process the `style` element as if the comment delimiters were not present.

However, using SGML comment delimiters in embedded style sheets is not recommended in XHTML, as XHTML parsers are allowed to strip out comments and their content regardless of what elements may contain the comments. So, in an XHTML-compliant browser an embedded style sheet enclosed within comment delimiters may be ignored. Given that almost all browsers in use today recognize the `style` element, and given this potential difficulty in XHTML browsers, I suggest that you not use SGML comment delimiters within `style` elements.

The `media` attribute described earlier can be used with the `style` element as well as with `link` elements, and therefore applies to both external and embedded style sheets. However, the `rel` attribute applies only to the `link` element, not to `style`. So an embedded style sheet is treated much the same as a persistent external style sheet: it cannot be selected or deselected by the browser user, but instead always applies to the document.

As we learned in the previous chapter, most HTML elements have a `style` attribute that can be used to define style properties for the element. Technically speaking, the value of a `style` attribute is not a style sheet, since it is not a set of style rules but is instead essentially a single list of style declarations that applies to a single document element. In fact, the use of style sheets is recommended over the use of `style` attributes, for a number of reasons. One reason is ease of coding: if you want all of the paragraphs in your document to have the same style applied, it is much easier to accomplish this by writing a single style rule than by adding a `style` attribute specification to every `p` element. Similarly, it is generally much easier to modify the style of a document that uses style sheets to define style than it is to modify one that uses `style` attributes. A `style` attribute value also cannot vary automatically with media type. This last observation is a special case of the more general recommendation that since markup is designed to carry structural and semantic information, it is generally best to keep all style information out of the body of an HTML document. All that said, there are times when the `style` attribute is convenient (e.g., to make an image cover an entire table cell, as in Section 2.7). So, while you shouldn't necessarily avoid its use altogether, try to use the `style` attribute wisely.

3.5 Style Rule Cascading and Inheritance

Before describing in detail many of the key CSS style properties, it will be helpful to understand two concepts: cascading of style sheet rules and element inheritance of style properties.

3.5.1 Rule Cascading

The style sheet of Figure 3.6 contains the rule

```
* { font-weight:bold }
```

which applies to every element of the HTML document. It also contains the rule

```
#p1, #p3 { background-color:aqua }
```

As we have seen, both of these rules applied to an element with `id` attribute value `p3`. That is, if multiple rules apply to an element, and those rules provide declarations for different properties, then all of the declarations are applied to the element. But what would happen if the rule

```
#p3 { font-weight:normal }
```

also appeared in a style sheet for the document? Which rule would apply to the `font-weight` property of the `p3` element?

This is one example of a more general question: For every property of every element on a page, the browser must decide on a value to use for that property. How does it determine this value if multiple style declarations apply to that property of that element? Furthermore, what should the browser do if no declaration at all directly applies to that element property? We'll deal with the first question in this subsection, and the second question in the next.

In order to choose between multiple declarations that all apply to a single property of a single element, the browser (or other user agent) applies *rule cascading*, a multistage sorting process that selects a single declaration that will supply the property value. The very first step of this process involves deciding which external and embedded style sheets apply to the document. For example, if alternate external style sheets are available, only one will apply, and rules in the other alternate style sheets will be ignored. Similarly, if a media type is specified for an embedded or external style sheet and that type is not supported by the user agent rendering the page, then that style sheet's rules will be ignored.

Once the appropriate external and embedded style sheets have been identified, the next stage of the sorting process involves associating an origin and weight with every declaration that applies to a given property of a given element. The *origin* of a style sheet declaration has to do with who wrote the declaration: the person who wrote the HTML document, the person who is viewing the document, or the person who wrote the browser software that is displaying the document. Specifically, the origin of a declaration is one of the following:

- *Author:* If the declaration is part of an external or embedded style sheet or is part of the value specified for the `style` attribute of the given element, then it originated with the author of the document that is being styled.
- *User agent:* A browser or other user agent may define default style property values for HTML elements. In the Mozilla 1.4 **View**|**Use Style** menu, this is the style sheet represented by the "Basic Page Style" option. Appendix A of the CSS 2.0 recommendation [W3C-CSS-2.0] contains an example user agent style sheet.
- *User:* Most modern browsers allow users to provide a style sheet or to otherwise indicate style preferences that are treated as style rules.

In Mozilla 1.4, the user style rules can be defined in two ways. First, under the **Edit**|**Preferences**|**Appearance** category, the Fonts and Colors panels allow a user to select various style options, which will be treated as user style rules. Second, the user can explicitly create a style sheet file that the browser will input when it is started. However, this is not an easy-to-use feature in Mozilla 1.4: you must create a file with a certain filename (`userContent.css`) and place it in a certain directory (the `chrome` subdirectory of the

directory specified by the Cache Folder field of **Edit| Preferences | Advanced | Cache**). Similar features are provided in IE6 under the **General** tab of the **Tools | Internet Options** window. The Colors and Fonts buttons allow the user to set style options, and a style sheet file can be read into IE by clicking the Accessibility button, checking the checkbox in the User Style Sheet panel, and selecting the file.

In addition to an origin, every author and user style declaration has one of two *weight* values: normal and important. A declaration has important weight if it ends with an exclamation mark (`!`) followed by the string `important` (or similar strings: case is not important, and there may be white space before or after the exclamation mark). So the rule

```
p { text-indent:3em; font-size:larger !important }
```

gives important weight to the declaration of the `font-size` property. A declaration without the `!important` string—such as the declaration of the `text-indent` property in that example—would have normal weight. All user-agent declarations can also be considered to have normal weight.

Once the origin and weight of all declarations applying to an element property have been established, they are prioritized (from high to low) as follows:

1. Important declaration with user origin
2. Important declaration with author origin
3. Normal declaration with author origin
4. Normal declaration with user origin
5. Any declaration with user agent origin

That is, we can think of each declaration as falling into one of five priority bins. We then look through the bins, starting with the first, until we find a nonempty bin. If that bin has a single declaration, the declaration is applied to the element property and we are done. Otherwise, there are multiple declarations in the first nonempty bin, and we continue to the next sorting stage in order to select a single declaration from among the candidates within this bin.

Before getting to this next stage, you may be wondering why important user declarations have higher priority than author declarations while normal-weight user declarations have lower priority. The reason is accessibility. If a visually impaired web user must have high contrast between text and background along with large bold fonts in order to read text on a monitor, that user can be accommodated by writing declarations with important weight, regardless of the page author's design decisions. On the other hand, a user who is merely stating style preferences will generally not want their default preferences to override those of a web site author who made specific style choices for his or her web site. One significant change between CSS1 and CSS2 was the adoption of the sort order just listed, which is also supported by the major modern browsers.

Now we return to the case in which the top nonempty bin of the weight-origin sort contains multiple style declarations for a single element property. The next step is to sort these declarations according to their *specificity*. First, if a declaration is part of the value of a `style` attribute of the element, then it is given the highest possible specificity value

(technically, in CSS2 this specificity value can be overridden, but that feature does not seem to be widely implemented by current browsers). If a declaration is part of a ruleset, then its specificity is determined by the selector(s) for the ruleset. We begin by treating a ruleset with a comma-separated selector string as if it were multiple rulesets each with a single selector; that is, a ruleset such as

```
h1, #head5, .big { font-size:x-large }
```

is treated as the equivalent three rulesets

```
h1 { font-size:x-large }
#head5 { font-size:x-large }
.big { font-size:x-large }
```

Next, we conceptually place each ruleset in one or more bins, each bin labeled with a class of selectors. The bins we use for this purpose, from highest to lowest specificity, are:

1. ID selectors
2. Class and pseudo-class selectors
3. Descendant and type selectors (the more element type names, the more specific)
4. Universal selectors

A ruleset with a selector such as `li.special` would go in two bins, since this is both a class and a type selector. Now we select a ruleset from the first nonempty bin. If, say, two rulesets appears in this bin, we search lower bins for the first recurrence of either ruleset. If one of the rulesets recurs before the other, then it is chosen. So, for example, `li.special` would be chosen over `*.special`.

Even after this sorting process, two or more declarations may still have equally high weight-origin ranking and specificity. The final step in the style cascade is then applied, and is guaranteed to produce a single declaration for a given property of a given element. First, if there is a declaration in the `style` attribute for the element, then it is used. Otherwise, conceptually, all of the style sheet rules are listed in the order in which they would be processed in a top-to-bottom reading of the document, with external and imported style sheets inserted at the point of the `link` element or `@import` rule that causes the style sheet to be inserted. The declaration corresponding to the rule that appears farthest down in this list is chosen. As an example, if the file `imp1.css` contains the statements

```
@import url("imp2.css");
p { color:green }
```

and the file `imp2.css` contains the statement

```
p { color:blue }
```

and a document `head` contains the markup

```
<title>StyleRuleOrder.html</title>
<style type="text/css">
  p { color:red }
</style>
<link rel="stylesheet" type="text/css" href="imp1.css" />
<style type="text/css">
  p { color:yellow }
</style>
```

then the style rulesets are effectively in the order

```
p { color:red }
p { color:blue }
p { color:green }
p { color:yellow }
```

and `p` elements will be displayed with yellow text. Notice that since the import at-rules must always come at the beginning of a style sheet, any imported rules can always be overridden by rules in the body of the style sheet causing the import. This is sensible, since rules imported from a file are presumably meant to be of a reusable, general-purpose nature and therefore should be subject to revision for a specific task.

Finally, certain (often deprecated) HTML attributes other than `style` can be used to affect the presentation of an HTML document. For example, the `height` attribute of the `img` element type can affect presentation. But `img` also has a `height` style property that can be set to achieve the same effect. If both are defined for an `img` element, which should take precedence: the attribute or the style property? The general answer is that any CSS style declaration takes precedence over style-type declarations made via HTML attribute specifications. More specifically, the browser or user agent treats non-CSS attribute styling as if an equivalent CSS style rule had been inserted at the very beginning of the author (normal weight) style sheet with a specificity lower than that for the universal selector. So any important-weight user style rule as well as any style rule written by the document author will take precedence over style rules derived from attributes such as `height`, which in turn will take precedence over normal-weight user and user-agent style rules.

The style cascade is summarized in Figure 3.9. We're now ready to tackle the other question posed earlier: if a property of an element has no associated style declarations, how is the value of the property determined? The answer is that the value is inherited from ancestors of the element, as discussed next.

3.5.2 Style Inheritance

While cascading is based on the structure of style sheets, inheritance is based on the tree structure of the document itself. That is, conceptually an element *inherits* a value for one of its properties by checking to see if its parent element in the document has a value for that property, and if so, inheriting the parent's value. The parent may in turn inherit its property value from its parent, and so on. Put another way, when attempting to inherit a property value, an element (say with `id` value `needValue`) will search upward through its tree of ancestor elements, beginning with its parent and ending either at the root `html` element or

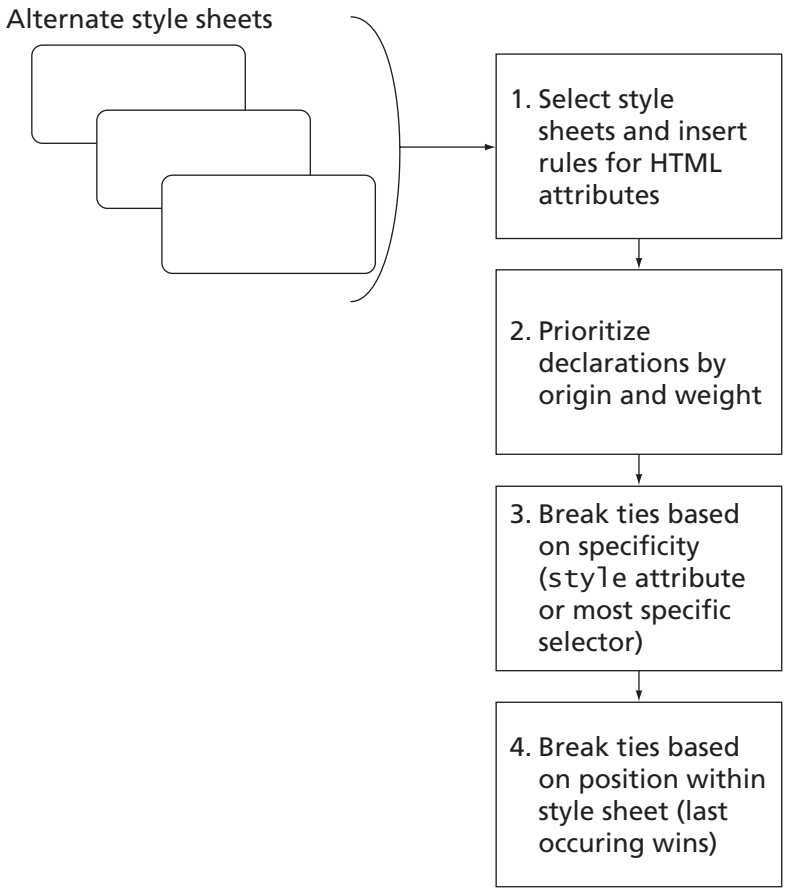

FIGURE 3.9 Steps in the CSS cascade.

at the first element that has a value for the property. If the search ends at an element with a value for the property, that value will be used by `needValue` as its property value. If no ancestor element has a value for the property, then as a last resort the property will be given a value specified for each property by the CSS specification [W3C-CSS-2.0] and known as the property's *initial value*. This terminology makes sense if you think of each element property as having its initial value assigned when the document is first read and then having this value changed if either the cascade or the inheritance mechanism supplies a value.

Figure 3.10 shows the source of an HTML document that illustrates inheritance. Notice that the style sheet for this document contains `font-weight` declarations for both the `body` and `span` element types. So for `span` elements, the `font-weight` is specified by an author rule, and no value will be inherited for this property. For other elements within the `body`, though, there is no author rule, and assuming that there is also no user or user-agent rule, the `font-weight` property value will be inherited from the `body` element. Therefore,

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      Inherit.html
    </title>
    <style type="text/css">
      body { font-weight:bold }
      li { font-style:italic }
      p { font-size:larger }
      span { font-weight:normal }
    </style>
  </head>
  <body>
    <ul>
      <li>
        List item outside and <span>inside</span> a span.
        <p>
          Embedded paragraph outside and <span>inside</span> a span.
        </p>
      </li>
    </ul>
  </body>
</html>
```

FIGURE 3.10 HTML document demonstrating inheritance.

as shown in Figure 3.11, the word "inside" (which is the content of two `span` elements) appears with a normal font weight, while all other text is boldfaced. However, since there are no other property declarations for the two `span` elements, these elements do inherit other property values from their ancestors. The first `span` inherits italicization from its parent `li` element, while the second inherits a larger font size from its `p` element parent and italicization from its `li` element grandparent. The `p` element similarly inherits italicization from its `li` parent.

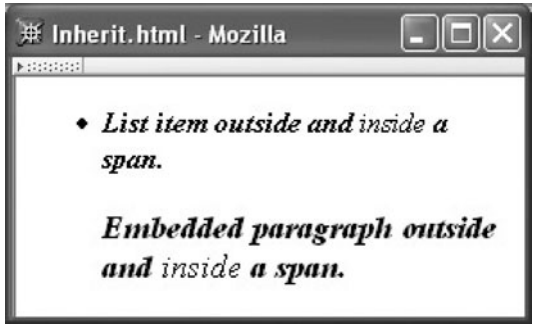

FIGURE 3.11 Rendering of document demonstrating inheritance.

A few final points should be mentioned about inheritance. First, while many CSS properties are inheritable, a number of other properties are not. In general, your intuition about whether or not a property will be inherited should be correct. For example, the `height` property of an element is not inherited from its parent, which is sensible, since often the parent has many children on many lines and therefore has a greater height than any one child. As I cover specific CSS properties in later sections of this chapter, you should assume that each property is inherited unless I explicitly say otherwise. Of course, you can consult the CSS specifications [W3C-CSS-2.0] if in doubt or for information about inheritance of properties not covered in this chapter.

A second point about inheritance has to do with exactly which of several possible property values is inherited. The value contained in a style declaration for a property is known as the *specified value* for the property. This value can be either relative or absolute. An *absolute value* is a value that can be understood without the need for any context, such as the value `2cm` (two centimeters). A *relative value*, on the other hand, cannot be understood without knowing some context. For example, the property declaration `font-size:larger` uses the relative value `larger` to set the size of the font of an element. Exactly what this value is relative to is discussed in Section 3.6.3. For now, it's sufficient to know that the browser must perform a calculation—which depends on the particular relative value—to obtain a *computed value* for the property. In the case of the font-size value `larger`, this calculation might involve multiplying the base font size by a factor such as 1.2 to obtain the computed font size. If the specified value is absolute, then the computed value is identical to the specified value. Finally, the computed value may not be suitable for immediate use by the browser. For example, a specified font size—relative or absolute—may not be available for the font currently in use, so the browser may need to substitute the closest available font size. The value actually used by the browser for a property is known, appropriately enough, as the *actual value*.

In terms of inheritance, the computed value is normally inherited for a property, not the specified or actual value. The one exception to this among the properties discussed in this chapter is `line-height`; its inheritance properties will be described in detail in Section 3.6.4.

A final note about inheritance is that the CSS2 recommendation allows every style property to be given the value `inherit`, whether or not the property is inherited normally. When this value is specified for a property, the computed value of the property is supposed to be obtained from its parent. However, you should be aware that this inheritance feature is not supported by IE6, and therefore should be used with care if at all. I am mentioning it mainly because it appears often in the CSS2 recommendation. Since this value can be used for every CSS2 property, I will not mention it explicitly when listing possible values for properties in the following sections.

We are now ready to begin learning about many of the available CSS2 properties. We'll begin with a number of text properties.

3.6 Text Properties

In this section, we will cover many of the CSS properties related to the display of text. Specifically, we will learn about how to select a font and how to modify text properties such as color. We'll also cover in some detail how browsers determine the spacing between

lines of text and how document authors can influence this spacing. Later sections will cover some other aspects of text, such as alignment, once we have covered necessary background material.

One note before beginning: CSS defines a `direction` property that can be thought of as defining the default direction in which text is written. It takes two possible keyword values: `ltr` indicates a left-to-right language, and `rtl` indicates right-to-left. This property affects the default behavior of many other CSS properties as well as some of their initial values. For example, the initial value for the `text-align` property, used to specify how a paragraph of text should be aligned, is `left` if `direction`'s value is `ltr` and is `right` otherwise. For simplicity, I will assume left-to-right languages throughout this chapter; if there is an asymmetry between left and right for a property (such as the initial value of `text-align`, which gives preference to `left`), simply switch the roles of left and right if you use a right-to-left language.

3.6.1 Font Families

Figure 3.12 is a browser rendering of an HTML document that displays characters using four different font families (we'll learn later how to write a document such as the one that generated this figure). A *font family* is a collection of related fonts, and a *font* is a mapping from a character (Unicode Standard code point) to a visual representation of the character (a *glyph*). Each glyph is drawn relative to a rectangular *character cell* (also known as the character's *content area*), which is shown shaded for each character in the figure. The fonts within a font family differ from one another in attributes such as boldness or degree of slantedness, but they all share a common design. The font families used in this example are, in order of use, Jenkins v2.0, Times New Roman®, Jokewood, and Helvetica™; they illustrate well how different font family designs can be from one another. (The Jenkins and Jokewood fonts may not be available on your machine, so this example may not appear the same in your browser as it does in Fig. 3.12.)

The font family to be used for displaying text within an HTML element is specified using the `font-family` style property. For example, the start tag

```
<p style="font-family:'Jenkins v2.0'">
```

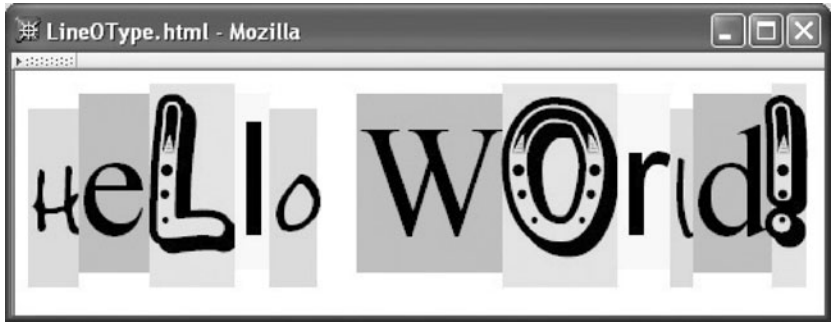

FIGURE 3.12 Rendering of document illustrating four different font families.

indicates that the text within the paragraph started by this tag should use the Jenkins v2.0 font (unless a child element specifies a different font). Some font family names must be quoted and/or special characters contained in the names must be escaped; for simplicity, I recommend that you always quote font family names. Either single or double quotes can be used, which is especially convenient when the declaration appears within a `style` attribute as shown.

Most end-user computers contain files describing a variety of font families. However, there is no guarantee that a font family that you would like to display in an HTML document you are authoring will be available on all of the client machines viewing your document. Although IE6 has a mechanism for downloading fonts from the Web for use within an HTML document, this facility is not included in the current version of CSS 2.1 [W3C-CSS-2.1]. Instead, a recommended mechanism for specifying a font family in CSS is to use a comma-separated list of font families as the value of the `font-family` property, such as

```
font-family:"Edwardian Script ITC","French Script MT",cursive
```

The browser will attempt to use the first family specified (`Edwardian Script ITC` in this example), but if that family is not available on the browser's system, then the browser will proceed through the list until it finds a family that is available. The last element in the list (`cursive` in this example) should be the name of a *generic* font family. The generic font families defined by CSS are listed in Table 3.3. Unlike normal font family names, the names of generic families are CSS keywords and therefore must not be quoted within a `font-family` declaration.

The browser will attempt to associate a reasonable font family available on the user's system with each generic name. In Mozilla 1.4, the user can specify the actual font family associated with each generic family through a preference setting as illustrated in Figure 3.13.

TABLE 3.3 CSS Generic Font Families

Font Family	Description
`serif`	A *serif* is a short, decorative line at the end of a stroke of a letter. There are three serifs at the top of the W in Figure 3.12, for example. Most glyphs in a serif font family will have serifs, and such a family is typically *proportionately* spaced (different glyphs occupy different widths).
`sans-serif`	Usually proportionately spaced, but glyphs lack serifs, so they don't look as fancy as serif fonts.
`cursive`	Looks more like cursive handwriting than like printing.
`fantasy`	Glyphs are still recognizable as characters, but are nontraditional.
`monospace`	All glyphs have the same width. Since monospace fonts are often used in editors when programming, these font families are frequently used to display program code or other computer data.

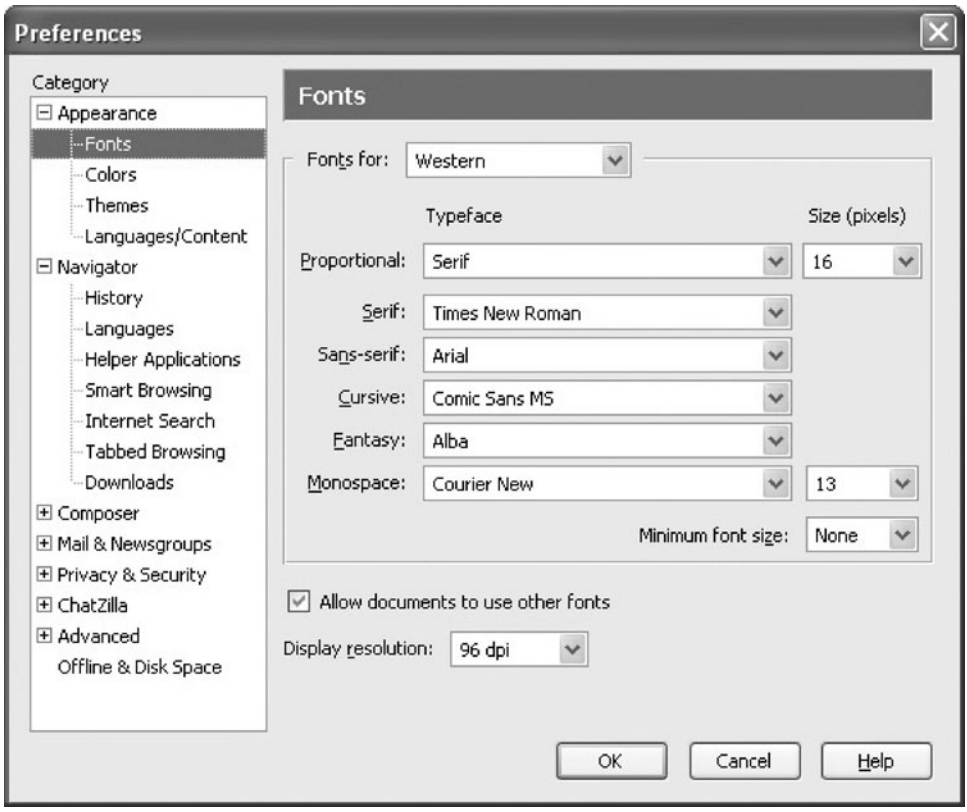

FIGURE 3.13 Example of associations of actual with generic font families in Mozilla.

3.6.2 Length Specifications in CSS

Font size is one of the key features used to distinguish between individual fonts within a font family. In CSS, the size of a font is specified using the `font-size` property. One type of value that can be assigned to `font-size` is a *CSS length*. In fact, CSS lengths can be assigned to many CSS properties, not just `font-size`. Therefore, we will cover length specification separately in this section before moving on to how to specify font properties such as size in CSS.

In CSS, a length value is represented either by the number `0` or by a number followed by one of the unit identifiers given in Table 3.4. Some example declarations involving length values are:

```
font-size:0.25in
font-size:12pt
font-size:15px
```

TABLE 3.4 CSS Length Unit Identifiers

Identifier	Meaning
`in`	Inch
`cm`	Centimeter
`mm`	Millimeter
`pt`	Point: 1/72 inch
`pc`	Pica: 12 points
`px`	Pixel: typically 1/96 inch (see text)
`em`	Em: reference font size (see text)
`ex`	Ex: roughly the height of the lowercase "x" character in the reference font (see text)

The first six units in Table 3.4 are, in practice, all related to one another by multiplicative scale factors. In particular, both Mozilla 1.4 and IE6 appear to maintain the relationships 1 in. = 2.54 cm = 25.4 mm = 72 pt = 6 pc = 96 px both on screen and when printing a document. Both also appear to use pixels to define all of the other units when displaying a document on a monitor. For example, if your monitor resolution is set to 1024 by 768 pixels and you specify a horizontal length as `1024px`, then this length will roughly correspond to the width of the monitor's display area. The display area will also be treated as if it were $1024/96 \approx 10.7$ in. across, regardless of its true physical width. Thus, the units `in`, `cm`, `mm`, `pt`, and `pc` are all only approximations on screen, and depending on the resolution may be off by 50% or more (all by the same factor). When printing, however, it appears that both browsers define `px` as 1/96 in. (or close to it) and define the other units accordingly.

Note that, despite the imprecisions, lengths defined using the first five units in Table 3.4 are absolute lengths in the sense defined earlier: they do not depend on other style property values. Such units are referred to as *absolute units*. The other three units are *relative units*. Technically, CSS defines `px` as a relative unit, since its physical value should depend on the medium displaying the document. However, as indicated, in practice it seems to be treated by typical browsers as an absolute unit. We'll see why `em` and `ex` are relative units in a moment.

Before defining the `em` and `ex` units formally, it will be helpful to understand several details about fonts. First, all character cells within a given font have the same height. However, generally speaking, this height is not exactly the same as the computed or even the actual value of the CSS `font-size` property. For example, in Figure 3.12, a single `font-size` value—72 pt—applies to all of the characters, yet obviously the character cells vary somewhat in height. Thus a combination of the font family and the `font-size` property determines the actual height of character cells. The `font-size` computed value is known as the *em height;* for most font families, the cell height is 10–20% greater than the em height.

Another feature that the font defines is the baseline height. The *baseline height* is the distance from the bottom of a character cell to an imaginary line known as the *baseline*, which is the line that characters appear to rest on. As shown in Figure 3.12, when a single line of text contains characters from different fonts, the characters are by default aligned

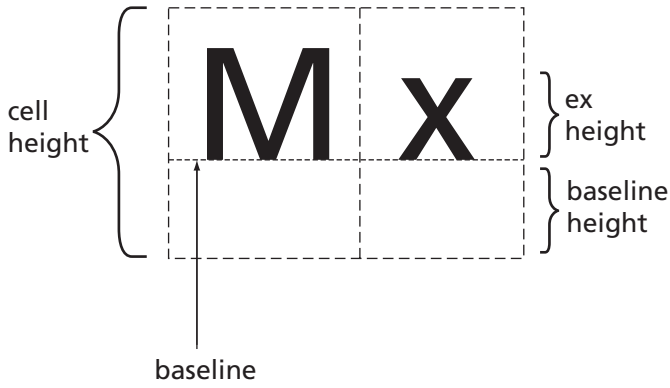

FIGURE 3.14 Some features and quantities defined by a font.

vertically by aligning their baselines. Thus, although we can see from the figure that the characters cells do not align vertically, the character glyphs themselves appear to all be written on a single horizontal line.

Yet another quantity defined by each font is the *ex height*. This quantity should be thought of as the font designer's definition of the height (above the baseline) of lowercase letters such as "x." Figure 3.14 illustrates this quantity and several other font features.

Now we can define the `em` and `ex` units. First, as noted in Table 3.4, these units are defined relative to a reference font (and are therefore relative units). With one exception explained in the paragraph after next, the reference font is just the font of the element to which the relative unit applies. So, for example, in the markup

```
<p style="width:20em">
```

the reference font is the font that applies (via a style rule or inheritance) to the `p` element.

Once the reference font is known, 1 em is simply the em height of the font, that is, the computed value of the `font-size` property of the reference font. So, continuing our example, if the computed value of the `p` element's `font-size` property is 0.25 in., then the computed value for its `width` property will be 5 in. Similarly, 1 ex is the ex height of the reference font.

The one exception when determining the reference font for these reference units is when one of them is used in a `font-size` declaration. In this case, the reference font is the font of the parent of the element to which the declaration applies. So in the markup

```
<div id="d1" style="font-size:12pt">
  <div id="d2" style="font-size:2em">
```

the reference font for the `div` with `id` attribute value `d2` will be `d1`'s font. Since the computed `font-size` for `d1` will be 12 pt (because absolute units are used), the computed `font-size` for `d2` will be 24 pt.

Now we are ready to more fully describe `font-size` and several other font properties.

3.6.3 Font Properties

The CSS font-size property, we now know, is used to specify the approximate height of character cells in the desired font within a font family. This property has an initial (default) value of medium, which is associated with a physical font size by the browser (these may vary with font family; Mozilla 1.4 defaults to 14 pt for proportional font families and 12 pt for monospace). A variety of other values can be specified for this property.

First, of course, a length value can be specified for font-size, using any of the length units described in the previous section. A second way that a font-size property may be specified is as a percentage of the computed font-size of the parent element. Since 1em represents the computed value of the parent's font-size, the following specifications are essentially equivalent:

```
font-size:0.85em
font-size:85%
```

Third, the font-size specification may be given using what is termed an *absolute size* keyword. One of these keywords is medium, the initial value for font-size. The remaining keywords are xx-small, x-small, small, large, x-large, and xx-large. The browser or other user agent creates a table of actual lengths corresponding to each of these size keywords. The CSS2 recommendation [W3C-CSS-2.0] is that each of these be approximately 20% larger than its next-smaller size.

Finally, the *relative size* keywords smaller and larger may be specified. Again, like the relative units em and ex, each of these keywords specifies the font size for the current element relative to the font size of its parent. These relative size keywords conceptually say "move one position in the font-size table." So, if the parent element has a font size of large, then a relative size specification of larger for its child is equivalent to an absolute size specification of x-large. If the parent font size is outside the range of the browser's font-size table, then an appropriate numerical font change (for example, 20%) is applied instead.

CSS also provides several other font style properties; three of the most commonly used are shown in Table 3.5. Several other font-related properties, including color, are covered later in this section.

TABLE 3.5 Additional Font Style Properties

Property	Possible Values
font-style	normal (initial value), italic (more cursive than normal), or oblique (more slanted than normal)
font-weight	bold or normal (initial value) are standard values, although other values can be used with font families having multiple gradations of boldness (see CSS2 [W3C-CSS-2.0] for details)
font-variant	small-caps, which displays lowercase characters using uppercase glyphs (small uppercase glyphs if possible), or normal (initial value)

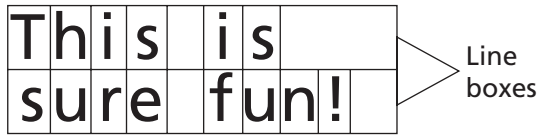

FIGURE 3.15 A box representing a p element that consists of two line boxes, each partially filled with character cells.

3.6.4 Line Boxes

We now want to consider how a browser determines where text should be rendered within an HTML element. We will assume in this section that the text is the content of an HTML p element, but the details are essentially the same for most other HTML block elements, such as div.

The simplest case is the one in which the content of the p element consists solely of text. In this case, we can think of the p element as being rendered as a rectangular box composed entirely of a stack of imaginary boxes called line boxes. Each *line box* contains one line of text. The height of a line box is precisely the height of a character cell in the p element's font. Character cells representing the text are placed side by side in the topmost line box until it contains as many words (strings of contiguous non-white-space characters) as possible; a single space (with width defined by the font) separates each word. When the first line box can hold no more words, this process continues with the second line box, and so on until all of text of the p element has been added to line boxes. There will be just enough line boxes to contain the text, so the height of the box representing the p element will be the number of line boxes multiplied by the height of a character cell. Figure 3.15 illustrates the rendering of the text "This is sure fun!" using a monospace font within a p-element box that is only wide enough to hold 10 characters. Notice that the box consists entirely of two line boxes, and that neither of the line boxes in this case is completely filled by character cells.

The browser's default setting of the height of line boxes can be overridden by specifying a value for the p element's line-height property. The initial value of this property is normal, which as we have seen sets the height of line boxes equal to the height of a character cell (a typical value might be 1.15 em, or 15% greater than the computed value of font-size). Other legal values for this property are a CSS length (using any of the units defined earlier), a percentage (treated as a percentage of the computed value of the p element's font-size), or a number without units. In the final case, the number is treated as if its units are em, except in terms of inheritance (we deal with this in a moment). Thus, the following declarations are all equivalent in terms of their effect on the p element itself:

```
line-height:1.5em
line-height:150%
line-height:1.5
```

If the height of a line box is greater than the character cell height, then the character cells are vertically centered within the line box. The distance between the top of a character

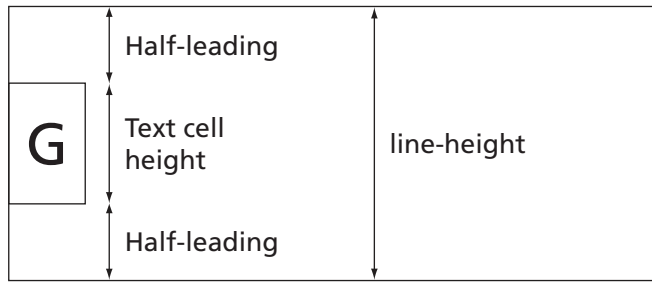

FIGURE 3.16 Default placement of text cell within a line box when the value of `line-height` exceeds the height of a text cell. An equal amount of space (half-leading) is inserted above and below the text cell.

cell and the top of a line box (which is the same as the distance between the bottom of a cell and the bottom of the line box) is sometimes called the *half-leading* (pieces of lead were often used to separate lines of type in early manual typesetting systems, hence the term). Thus, increasing the `line-height` value above its `normal` value not only increases the distance between lines, but actually moves the text of the first line down by the half-leading distance as well as increasing the distance between the last line of text and whatever follows the `p` element's box by the same distance (Fig. 3.16). We will learn how to override this default centering of text within tall line boxes later in this chapter.

A fine point about inheritance of this property: If `normal` or a number without units is specified as the value of `line-height`, then this specified value is inherited rather than the computed value. For any other specified value, such as `1.5em`, the computed value is inherited. An exercise explores this further.

Now that we have learned about the `line-height` property, I can describe a convenient property called `font`. This property is an example of a CSS *shortcut property*, which is a property that allows values to be specified for several nonshorthand properties with a single declaration. As an example of the use of the `font` shortcut, the declaration block

```
{ font: italic bold 12pt "Helvetica",sans-serif }
```

is equivalent to the the declaration block

```
{ font-style: italic;
  font-variant: normal;
  font-weight: bold;
  font-size: 12pt;
  line-height: normal;
  font-family: "Helvetica",sans-serif }
```

Notice that the `font` shortcut always affects all six of the properties shown, resetting those for which a value is not specified explicitly in the `font` declaration to their initial (default) values. The font size and font family (in this order) must be included in the specified value for `font`. If values for any of style, variant, and weight appear, they must appear

TABLE 3.6 Primary CSS Text Properties

Property	Values
`text-decoration`	`none` (initial value), `underline`, `overline`, `line-through`, or space-separated list of values other than `none`.
`letter-spacing`	`normal` (initial value) or a length representing additional space to be included between adjacent letters in words. Negative value indicates space to be removed.
`word-spacing`	`normal` (initial value) or a length representing additional space to be included between adjacent words. Negative value indicates space to be removed.
`text-transform`	`none` (initial value), `capitalize` (capitalizes first letter of each word), `uppercase` (converts all text to uppercase), `lowercase` (converts all text to lowercase).
`text-indent`	Length (initial value 0) or percentage of box width, possibly negative. Specify for block elements and table cells to indent text within first line box.
`text-align`	`left` (initial value for left-to-right contexts), `right`, `center`, or `justified`. Specify for block elements and table cells.
`white-space`	`normal` (initial value), `pre`. Use to indicate whether or not white space should be retained.

before the font size and may appear in any order among themselves. To specify a value for `line-height`, immediately follow the font size value by a slash (/) and the `line-height` value. For example,

```
{ font: bold oblique small-caps 12pt/2 "Times New Roman",serif }
```

is a valid `font` declaration that explicitly sets all six font properties.

3.6.5 Text Formatting and Color

Beyond font selection, several other CSS properties can affect the appearance of text. These are listed in Table 3.6. All of these properties except `text-decoration` are inherited. And, while not inherited, `text-decoration` automatically applies to all text within the element, while skipping nontext, such as images. The decoration uses the element's color value. Some of these properties may interfere with one another. For example, since text justification (lining up text with a straight edge on both left and right sides) generally involves inserting space between letters and/or words, specifying `justify` for `text-align` and also specifying values for `letter-spacing` and `word-spacing` may not produce the results you expect. As usual, see the CSS2 recommendation [W3C-CSS-2.0] for details on such special cases.

Finally, as we learned in early examples in this chapter, the `color` property is used to specify the color for text within an element. There are many possible values for the `color` property, which we now cover. It should be noted that these values can also be specified for several other CSS properties, as discussed later.

CSS2 color properties can be assigned several types of values. The most flexible type is a numerical representation of the color. In particular, three numerical values are used

TABLE 3.7 Alternative Formats for Specifying Numeric Color Values

Format	Example	Meaning
Functional, integer arguments	`rgb(255,170,0)`	Use arguments as RGB values.
Functional, percentage arguments	`rgb(100%,66.7%,0%)`	Multiply arguments by 255 and round to obtain RGB values (at most one decimal place allowed in arguments).
Hexadecimal	`#ffaa00`	The first pair of hexadecimal digits represents the red intensity; the second and third represent green and blue, respectively.
Abbreviated hexadecimal	`#fa0`	Duplicate the first hexadecimal digit to obtain red intensity; duplicate the second and third to obtain green and blue, respectively.

to specify a color, representing intensities of red, green, and blue to be mixed together in order to simulate the desired color (the typical human eye can be "tricked" into perceiving light from multiple sources at various intensities and wavelengths as if it were from a single source with a single intensity and wavelength). The specific color model used involves specifying an integer between 0 and 255, inclusive, for each of the intensities of red, green, and blue, in that order (early Web pages used a limited range of intensities due to hardware limitations of many computers in use at the time, but most machines today can reliably display any of these intensities). Such an integer is known as an *RGB value*. Many readily available software tools, including Microsoft Paint, provide visual maps from colors to RGB values. Four different formats can be used to specify these three values, as shown in Table 3.7. All of the examples in this table specify the same color value. (A word of caution: it's easy to forget the leading `#` for the third and fourth formats.)

Many of our earlier style sheet examples used a second, more convenient way to specify common colors: many color values have a standard name associated with them. A list of the 16 colors named in CSS2 and their associated RGB values is given in Table 3.8. The current CSS 2.1 specification also adds `orange` (`#ffa500`) to the list. Furthermore, Mozilla 1.4 supports all and IE6 supports almost all (there are some exceptions containing `gray` or `grey`) of the 147 color names recognized as part of the W3C's Scalable Vector Graphics recommendation [W3C-SVG-1.1]. This provides 130 color names in addition to those of CSS 2.1, from `aliceblue` through `yellowgreen` (see `http://www.w3.org/TR/SVG11/types.html#ColorKeywords` for a complete list).

Finally, color values can be specified by referencing colors set for other purposes on the user's system. For example, the keyword `Menu` represents the color used for menu backgrounds, and `MenuText` the color used for text within menus. This can be useful, for example, if you plan to provide menus within your page and want them to use colors familiar to your users, regardless of user selected menu color preferences. A full list of these so-called system color names is provided in Section 18.2 of the CSS2 specification [W3C-CSS-2.0]. However, be advised that the draft for CSS3 current at the time of this writing deprecates such system color names.

TABLE 3.8 CSS2 Color Names and RGB Values

Color Name	RGB Value
black	#000000
gray	#808080
silver	#c0c0c0
white	#ffffff
red	#ff0000
lime	#00ff00
blue	#0000ff
yellow	#ffff00
aqua	#00ffff
fuchsia	#ff00ff
maroon	#800000
green	#008000
navy	#000080
olive	#808000
teal	#008080
purple	#800080

We have referred throughout this section to the notion of a box corresponding to a p element. In the next section, we begin to make this concept more precise.

3.7 CSS Box Model

In this section we define a number of CSS properties that relate to the boxes that a browser renders corresponding to the elements in an HTML document. In subsequent sections we will learn how browsers position these boxes relative to the browser client area and relative to one another.

3.7.1 Basic Concepts and Properties

In CSS, each element of an HTML or XML document, if it is rendered visually, occupies a rectangular area—a *box*—on the screen or other visual output medium. What's more, every box consists conceptually of a nested collection of rectangular subareas, as shown in Figure 3.17. Specifically, there is an innermost rectangle known as the *content area* that encloses the actual content of the element (line boxes or boxes for other elements, or both). *Padding* separates the content area from the box's *border*. There is then a *margin* surrounding the border. The content and margin edges are not displayed in a browser, but are drawn in Figure 3.17 for definitional purposes. Note the similarity between the CSS box model and the concept of a cell in HTML tables. However, as we will see, style properties in CSS provide finer-grained control over boxes than HTML provides for table cells.

Some other terminology related to the box model will also be helpful. The content and margin edges of an element's box are sometimes referred to as the *inner* and *outer* edges of the box, respectively. Also, as shown in Figure 3.18, the outer (margin) edges of a

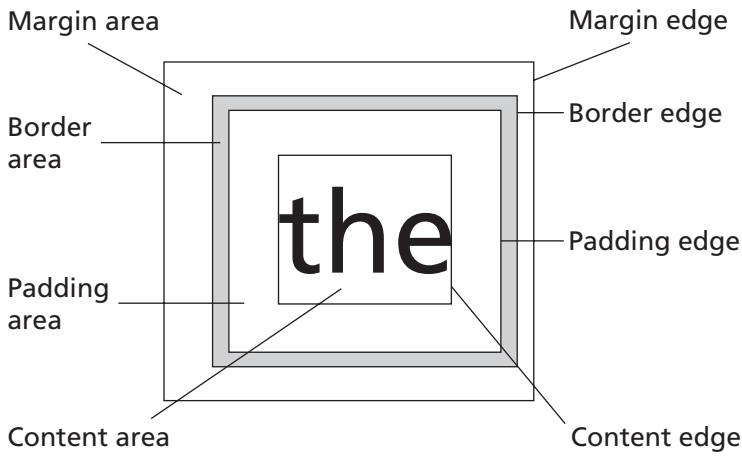

FIGURE 3.17 Definition of areas and edges in the CSS box model.

box define the *box width* and *box height*, while the inner (content) edges define the *content width* and *content height* of the box.

Figure 3.18 also gives the CSS property names corresponding to the 12 distances between adjacent edges in the box model. Notice that the border properties have the suffix `-width`. This suffix is used to distinguish border properties related to distances from other border properties that affect the color and style of borders (and have the suffixes `-color` and `-style`, respectively). Note that the same suffix is used for both horizontal and vertical distances, which can be confusing, since in the rest of the box model "width" normally refers to a horizontal distance.

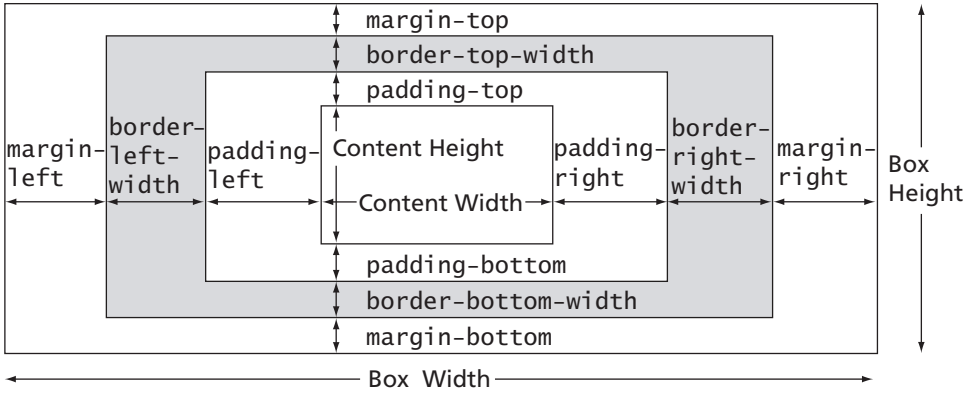

FIGURE 3.18 Definition of various lengths in the CSS box model.

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      SpanBoxStyle.html
    </title>
    <link rel="stylesheet" type="text/css" href="span-box-style.css" />
  </head>
  <body>
    <p>
      The <span>first span</span> and <span>second span</span>.
    </p>
  </body>
</html>
```

FIGURE 3.19 HTML document demonstrating basic box model style properties.

As a simple example of what can be done with what we have already learned (and a few other things that we will learn shortly about border-style property values), consider the following style sheet:

```
/* span-box-style.css */
/* solid is a border style (as opposed to dashed, say). */
span { margin-left: 1cm;
       border-left-width: 10px;
       border-left-color: silver;
       border-left-style: solid;
       padding-left: 0.5cm;
       border-right-width: 5px;
       border-right-color: silver;
       border-right-style: solid }
```

and assume that this style sheet is contained in a file named `span-box-style.css`, as indicated by the comment. Then the HTML document shown in Figure 3.19 will be rendered by a CSS2-compliant browser as illustrated in Figure 3.20. Note that for `span` elements, any margin, border, or padding distance that is not specified by an author or user style sheet is given the value 0.

FIGURE 3.20 Rendering of document demonstrating basic style properties.

3.7.2 Box Model Shorthand Properties

CSS2 defines a number of shorthand properties related to the box model. For example, the declaration

```
padding: 30px;
```

is shorthand for four declarations:

```
padding-top: 30px;
padding-right: 30px;
padding-bottom: 30px;
padding-left: 30px;
```

Table 3.9 lists a number of such shorthand properties as well as the properties already covered and gives for each property its allowable values. None of the properties in this table is inherited.

The `auto` value that can be used when setting margin widths has a meaning that depends on its context, so we will defer discussing it to the appropriate later sections. I'll try to answer other questions you may have about Table 3.9 here.

First, notice that five of the properties in Table 3.9 (`padding`, `border-width`, `border-color`, `border-style`, and `margin`) take from one to four space-separated values. Each of these properties is a shorthand for specifying values for the four

TABLE 3.9 Basic CSS Style Properties Associated with the Box Model

Property	Values
`padding-{top,right,bottom,left}`	CSS length (Section 3.6.2).
`padding`	One to four length values (see text).
`border-{top,right,bottom,left}-width`	`thin`, `medium` (initial value), `thick`, or a length.
`border-width`	One to four `border-*-width` values.
`border-{top,right,bottom,left}-color`	Color value. Initial value is value of element's `color` property.
`border-color`	`transparent` or one to four `border-*-color` values.
`border-{top,right,bottom,left}-style`	`none` (initial value), `hidden`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`.
`border-style`	One to four `border-*-style` values.
`border-{top,right,bottom,left}`	One to three values (in any order) for `border-*-width`, `border-*-color`, and `border-*-style`. Initial values are used for any unspecified values.
`border`	One to three values; equivalent to specifying given values for each of `border-top`, `border-right`, `border-bottom`, and `border-left`.
`margin-{top,right,bottom,left}`	`auto` (see text) or length.
`margin`	One to four `margin-*` values.

TABLE 3.10 Meaning of Values for Certain Shorthand Properties that Take One to Four Values

Number of Values	Meaning
One	Assign this value to all four associated properties (`top`, `right`, `bottom`, and `left`).
Two	Assign first value to associated `top` and `bottom` properties, second value to associated `right` and `left` properties.
Three	Assign first value to associated `top` property, second value to `right` and `left`, and third value to `bottom`.
Four	Assign first value to associated `top` property, second to `right`, third to `bottom`, and fourth to `left`.

associated properties that include `top`, `right`, `bottom`, or `left` in their names. For example, `border-style` is a shorthand for specifying values for `border-top-style`, `border-right-style`, `border-bottom-style`, and `border-left-style`. Table 3.10 shows the meaning of the values for these properties. We have just seen an example of such a shorthand declaration, when a single padding declaration was equivalent to four declarations. As another example, the style declaration

```
margin: 15px 45px 30px
```

is equivalent to

```
margin-top: 15px
margin-right: 45px
margin-left: 45px
margin-bottom: 30px
```

You may also have a question about the border styles listed in Table 3.9. There is no precise definition for most of these border styles, so their visual appearance may vary somewhat when displayed in different browsers. For example, Figure 3.21 shows two paragraphs `p1` and `p2` displayed in two different browsers using the following style sheet:

```
/* border-styles.css */
#p1 {
        background-color: yellow;
        border: 6px maroon;
        border-style: solid dashed dotted double
}

#p2 {
        border: 16px teal;
        border-style: groove ridge inset outset
}
```

Obviously, you may want to experiment with these border-style values in browsers you are targeting before using these values. Also note that both of the border style values

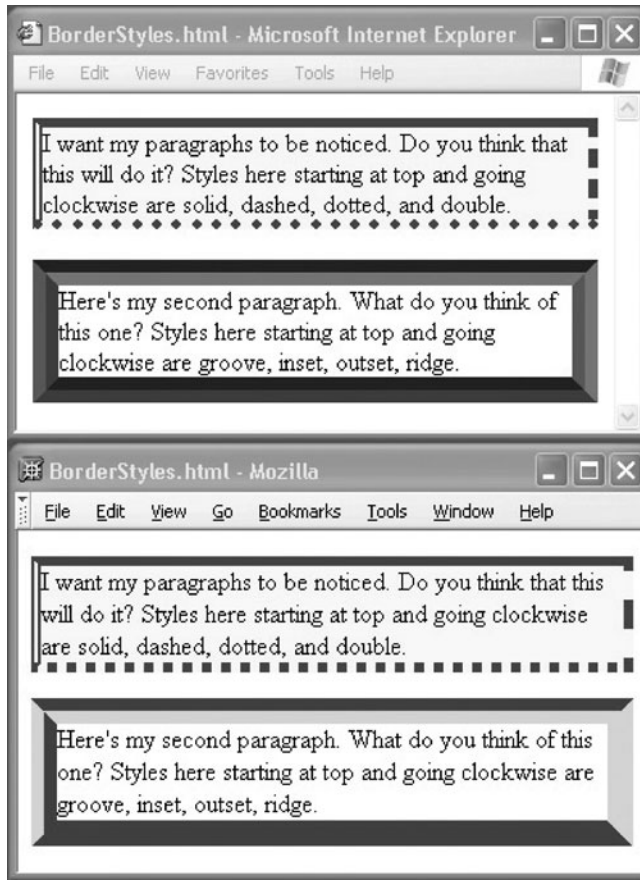

FIGURE 3.21 Illustration of some border styles in different browsers.

`hidden` and `none` effectively eliminate a border from a box element, but `hidden` behaves slightly different within HTML table elements in certain circumstances (refer to the CSS2 specification [W3C-CSS-2.0] for details).

Finally, you probably have noticed that shorthand properties make it possible to declare multiple values for a single property within a single declaration block. For example, the value of `border-top-style` can be specified by a direct declaration of this property as well as by declarations for the `border-top` and `border` shorthand properties. If multiple declarations within a single declaration block apply to a property, the last declaration takes precedence over any earlier declarations. So, for example, in the declaration block

```
{ border: 15px solid;
  border-left: 30px inset red;
  color: blue }
```

the border on the top, right, and bottom will be 15-px-wide solid blue, while the left border will be a 30-px-wide red inset style. This is because the first declaration sets all four borders to 15 px wide and solid, with the border color set to its initial value, which for border colors is the value specified for the element's color property (blue in this case). The second declaration effectively overrides these values for the `border-left` property.

3.7.3 Background Colors and Images

The `background-color` property specifies the color underlying the content, padding, and border areas of an element's box. The background color in the border area will normally be covered by the border itself, but will be visible if the border color is specified as `transparent` or partly visible if the border style is `dashed`, `dotted`, or `double` (see Fig. 3.21). Notice that the margin area is not affected by the background color. The margin area is always transparent, which allows the background color of the parent element to be seen in the margin area. Strictly speaking, the `background-color` property's value is not inherited; however, the initial value of `background-color` is `transparent`, and the background color of an element will be visible through transparent areas of child elements. In other words, for CSS box model purposes, we should think of the browser as rendering parent elements first and then rendering the nontransparent portions of the child elements over top of the parents.

A related property that is used in many Web pages is `background-image`. The acceptable values for this property are `none`, the initial value, or a URL specified using the same `url()` functional notation used with the `@import` style rule. By default, the image found at the specified URL will be *tiled* over the padding and content areas of the element to which this property is applied (such as the `body` element of an HTML document). *Tiling* simply means that if an image is too small to cover the element, either from left to right or from top to bottom or both, then the image is repeated as needed.

Like `background-color`, `background-image` is not inherited. Conceptually, the element to which the background image will be applied is first drawn, including its background color if any. Then the background image is drawn over top of the element, with the element showing through any transparent areas of the image. Finally, any child elements are drawn over top of the background image. The positioning of a background image and whether it is tiled or not can be specified using various CSS properties; see the CSS2 specification [W3C-CSS-2.0] for complete details.

This concludes our discussion of the basic CSS box model. We next turn to considering how this model relates to some specific HTML elements.

3.8 Normal Flow Box Layout

In a browser's standard rendering model (which is called *normal flow processing*), every HTML element rendered by the browser has a corresponding rectangular box that contains the rendered content of the element. The primary question we address in this section is where these boxes will be placed within the client area of the browser.

3.8.1 Basic Box Layout

First, recall that the `html` element is the root element of an HTML document. The browser generates a box corresponding to this element, which is called the *initial containing block*. The CSS2 recommendation does not specify the lengths (margin, padding, etc.) or dimensions (width and height) of this box, but instead leaves it to each browser to choose values for these parameters. In both IE6 and Mozilla 1.4, by default the border, margin, and padding lengths are all zero, so the inner (content) and outer edges of the box coincide. As for dimensions of the initial containing block box, if the browser's horizontal and vertical scrollbars are not active, then the box coincides with the browser's client area, and therefore has the same dimensions. On the other hand, if either scrollbar is active, then the the outer edges of the initial containing block are located at the edges of the underlying area over which the browser can be scrolled. Conceptually, it is as if the document is drawn on an imaginary *canvas*. The browser client area acts as a *viewport* through which all or part of the canvas is viewed. The initial containing block's height is the total height of this canvas, or the height of the browser's client area if that is greater than the canvas height. The width of the initial containing block is defined similarly. Figure 3.22 illustrates the relationship

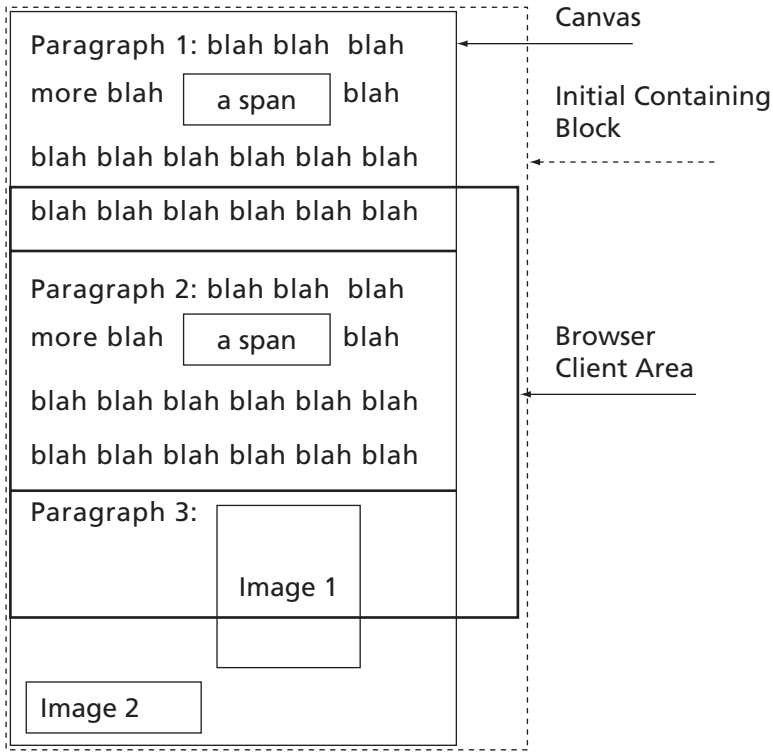

FIGURE 3.22 Initial containing block box when canvas is taller than client area but client area is wider than canvas.

between the canvas, client area, and initial containing block box when the canvas is taller than the client area but not as wide. Note that if the browser window is resized, the initial containing block's box will be resized automatically as needed.

All other CSS boxes within the client area are laid out (either directly or indirectly) relative to the initial containing block box. For an HTML document, the first such box to be added to the client area is the one corresponding to the `body` element. Because the `body` element is contained within the `html` element, the box corresponding to the `body` element is placed within the initial containing block box (which corresponds to the `html` element). This is the default behavior for all boxes: if one HTML element is part of the content of a second HTML element, then the box corresponding to the first element will be contained within the content area of the box for the second element. This default behavior is known as *normal flow* processing of boxes. Thus, if normal flow processing is used for an entire HTML document, all of the boxes corresponding to elements within the `body` element of the document will be contained within the box generated for the `body`, which in turn will be contained within the initial containing block box. In essence, in normal flow processing, the block corresponding to the `body` element is the canvas on which boxes for all other elements will be drawn.

By default, the `body` box will be placed so that its left, top, and right outer edges coincide with the left, top, and right inner (content) edges of the initial containing block. If the width of the browser window is changed, then the width of the `body` box may change as well, since the width of the initial containing block can change automatically when the browser width changes. The height of the `body` box, on the other hand, is determined by its content. You might think of the box as starting with the height of its content area set to 0. Then, as the browser generates boxes corresponding to elements contained within the `body`, it increases this height so that it is just sufficient to contain all the generated boxes. The height when this process is done determines the final height of the content area of the `body` element's box (the overall height of the box also depends on the values of style properties such as `margin-top`).

Similar rules apply to the default placement of boxes within the `body` box. That is, the first child element's box will be placed so that its left, top, and right outer edges coincide with the corresponding content edges of the `body` box. The height of this box will then be determined by generating boxes for all of the elements contained within the first element and laying these boxes out within this first child box (by recursively applying the layout rules being described). The second child element's box will be placed so that its top outer edge coincides with the bottom outer edge of the first child box (this isn't quite correct; see Section 3.8.3 for more details). The left and right edges of this second child box will also coincide with the left and right content edges of the `body`. The second child is then filled with all of its descendants' boxes. This process continues with the remaining children of the `body`.

Figure 3.23 is an HTML document that illustrates the layout concepts discussed thus far (I have used an embedded style sheet in this document and several others in this chapter for ease of reading, but in practice I would probably have used an external style sheet). Figure 3.24 shows how Mozilla 1.4 renders this document (the IE6 rendering is similar, although my copy of IE6 incorrectly draws the initial containing block's border so that it always coincides with the client area, regardless of how the browser window is sized).

```
<!DOCTYPE html
       PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      BlockBoxes.html
    </title>
    <style type="text/css">
      html, body { border:solid red thin }
      html { border-width:thick }
      body { padding:15px }
      div { margin:0px; padding:15px; border:solid black 2px }
      .shade { background-color:aqua }
      .topMargin { margin-top:10px }
    </style>
  </head>
  <body>
    <div id="d1">
      <div id="d2">
        <div id="d3" class="shade"></div>
      </div>
      <div id="d4" class="shade topMargin"></div>
    </div>
  </body>
</html>
```

FIGURE 3.23 HTML document containing nested `div` elements.

Figure 3.24 shows the borders of a number of boxes. The outermost border (thick red border at the edges of the browser's client area) is for the initial containing block box generated by the `html` element. The thin red border immediately inside the `html` element's box belongs to the `body` element's box. You'll notice that the `body` border does not touch the `html` border. This is because the Mozilla 1.4 user agent style sheet specifies a nonzero margin value (apparently about 8 px in Mozilla 1.4 and 10 px in IE6) for the `body` box, and the embedded author style sheet does not override this value. Inside the `body` block box there is a box with a medium-width black border generated by the `div` with ID `d1`. Inside this box are two child boxes, one for each of the `div` children of `d1` (`d2` and `d4`). Finally, the first of these child elements (`d2`) itself has a child `div` with id `d3`, which generates its own box. The boxes for the `div` elements `d3` and `d4`, which have no content, are given a background color in Figure 3.24.

3.8.2 The `display` Property

The layout rules described so far only apply to HTML elements that CSS recognizes as *block elements*. These are elements for which the CSS `display` property has the value `block`. Of the elements covered in Chapter 2, standard user agent style sheets will define the following HTML elements as block elements: `body`, `dd`, `div`, `dl`, `dt`, `fieldset`, `form`, `frame`, `frameset`, `hr`, `html`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `ol`, `p`, `pre`, and `ul`. You may recall from the last chapter that

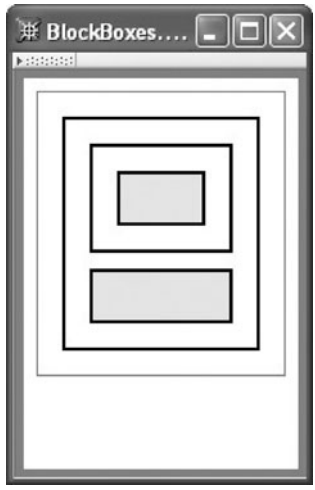

FIGURE 3.24 Nested boxes.

we informally introduced the concept of block elements as those elements for which the browser essentially generates a new line before and after the element. Now we see that what actually happens is that the browser stacks the boxes for these elements one on top of the next.

CSS defines a number of other possible values for the `display` property. Many of these values are associated with specific HTML elements. For example, there is a `list-item` value that is intended to be used as the `display` value for `li` elements, a `table` value for HTML `table` elements, and a `table-row` value for `tr` elements. In fact, nearly every element associated with the HTML table model has its own value for the `display` property (`td` and `th` share the `table-cell` value). We will not discuss these values further; see [W3C-CSS-2.0] for details.

In addition to these and other somewhat specialized values for `display`, there is another value that is shared by a number of HTML elements: `inline`. Again, recall from the previous chapter that inline HTML elements are those that do not interrupt the flow of a document by starting a new line as block elements do. Examples of inline elements were `span` and `strong`. In a typical browser, all of the HTML elements discussed in the last chapter except the block elements listed at the begining of this subsection, the `li` element, and table-related elements will be treated as having the value `inline`, which is the initial value for the `display` property.

As you might expect, the rules for laying out the boxes for elements with a `display` value of `inline` (which I'll refer to as *inline boxes*) are different from those for laying out boxes for elements with a `display` value of `block` (*block boxes*). In fact, how content is laid out within inline and block boxes also differs. We'll cover some more details concerning block boxes in the next few sections and then look more closely at inline boxes.

Before leaving this section, let me mention that an author style sheet can override the default value of an element's `display` property just as any other default property value

can be overridden. For example, suppose that an HTML document has a large number of consecutive p elements but that for some reason we would like—with a minimal amount of change to the document—to have all of these separate paragraphs in the document displayed as one (long) paragraph. We can accomplish this by adding to the document the style rule

```
p { display:inline }
```

Obviously, this style rule significantly changes the expected semantics of the p element, so a rule such as this should be used with some caution.

3.8.3 Margin Collapse

Earlier I said that, roughly speaking, consecutive block boxes are positioned one on top of the next. I'll now explain why this isn't exactly the case.

When two consecutive block boxes are rendered (the first on top of the second), a special rule called *margin collapse* is used to determine the vertical separation between the boxes. As the name implies, two margins—the bottom margin of the first (upper) box and the top margin of the second (lower) box—are collapsed into a single margin.

Specifically, let m_1 represent the value of margin-bottom for the top box, and let m_2 represent the value of margin-top for the lower box. Without margin collapse, the distance between the borders of these boxes would be $m_1 + m_2$. With border collapse, the distance will instead be $\max(m_1, m_2)$ (see Fig. 3.25).

3.8.4 Block Box Width and Height

Each block element has a width property (not inherited) that defines the width of the content area of the element's block box. The initial value of this property is auto, which produces the width-defining behavior described earlier: the box will automatically be stretched horizontally so that its left and right outer edges align with the left and right content edges of its parent box. As an example, if the browser window shown in Figure 3.24 is widened, the block boxes displayed in the content area will also become wider (Fig. 3.26).

More precisely, if the value of width is auto, and if a value other than auto is specified for both margin-left and margin-right (the initial value for these properties is 0), then for display purposes width will be given the value

```
width = parent's displayed content width -
       (margin-left + border-left-width + padding-left +
        padding-right + border-right-width + margin-right)
```

This value is not in any way associated with the width property itself, as a specified or computed value is. Instead, it is used by the browser strictly for display purposes. Such a value is sometimes referred to as a property's *used value*.

In addition to auto, a length value can be specified as the value of the width property of a block element. The length value can use any of the units described in Section 3.6.2. Furthermore, the specified length value can be a *percentage*, which is a number (integer or decimal) immediately followed by a percent sign (%). In the case of the width property,

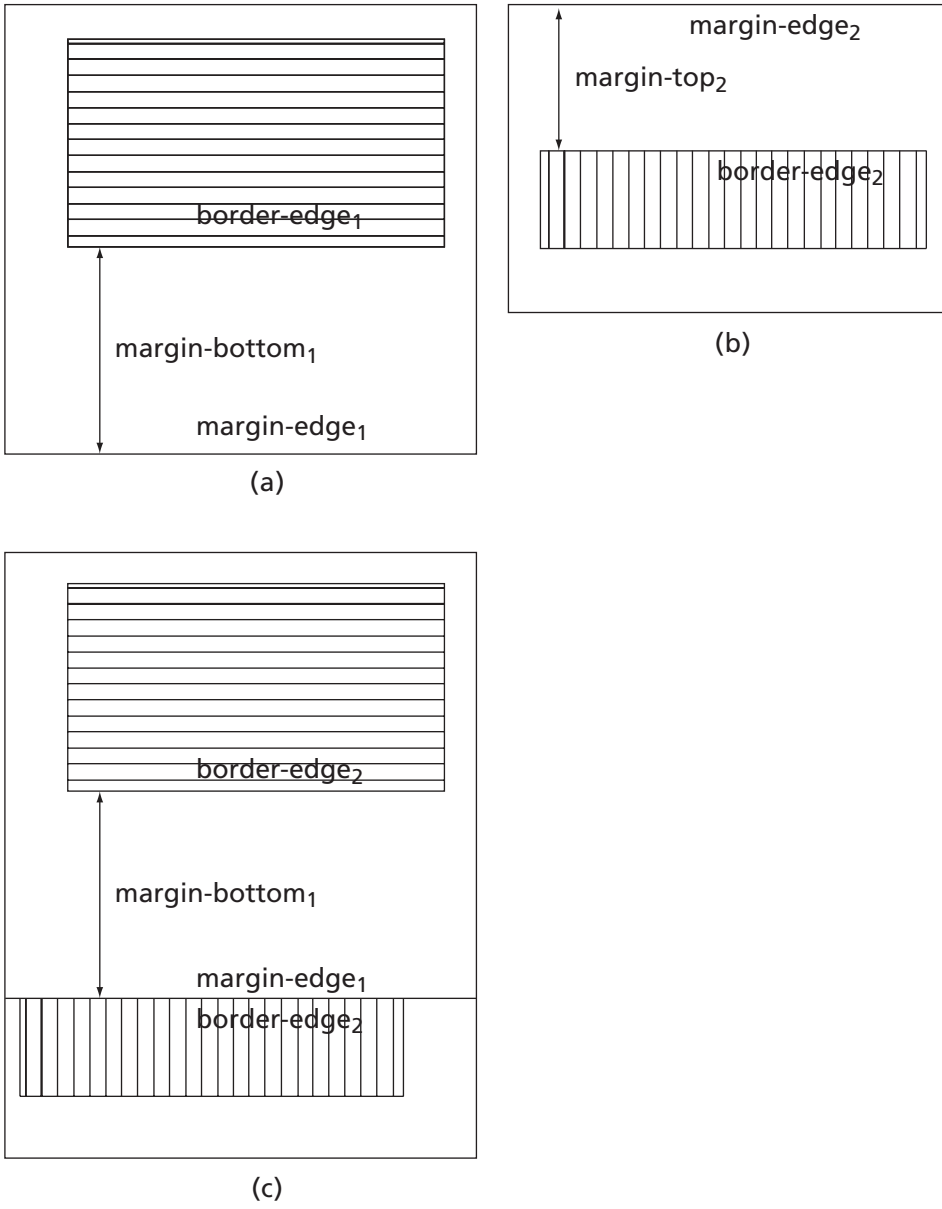

FIGURE 3.25 (a) A block box (only margin and border edges are shown). (b) A second block box with `margin-top` smaller than `margin-bottom` of first box. (c) First and then second boxes rendered, illustrating margin collapse.

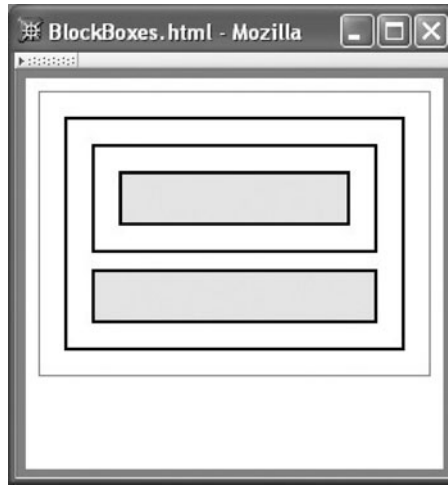

FIGURE 3.26 `BlockBoxes.html` displayed in a wider window.

this represents a percentage of the width of the parent element's content area, or more precisely, a percentage of the used value associated with the content width. For example, the declaration

```
width:50%
```

says that the width of the content area of a box should be half the (used) width of the content area of its parent box. Percentages can also be used with many other CSS properties that take a length value, although the length to which the percentage is applied varies from property to property. See the CSS2 recommendation [W3C-CSS-2.0] for details regarding properties not explicitly mentioned in this chapter.

You might expect (I did initially) that the percentage `width` declaration would cause an associated element to be centered within the parent box. However, this is not the case by default. Instead, the element will appear left-justified within its parent box. In essence, when only the `width` is specified for an element, the browser computes a used value for the `margin-right` property of the element's box so that the overall width of the box (sum of the element width plus left and right margins, borders, and paddings) is equal to the width of its parent's content area. The `margin-left`, however, is unchanged. To center an element, in addition to specifying a value for the element's `width` property, the value `auto` should be specified for both the `margin-left` and `margin-right` properties of the element. The browser will then use a single value for both margins, with the used value being computed so that the borders (but not necessarily the content) of the box will be centered within the content area of the parent box.

So, for example, assume that we create an HTML document `BlockBoxesWidth.html` from the earlier `BlockBoxes.html` example (Fig. 3.23) by adding the following two rules to the embedded style sheet:

```
#d3 { width:50% }
#d4 { width:50%; margin-left:auto; margin-right:auto }
```

Note that these rules will override the declarations having element selector `div`, due to the higher specificity of ID selectors. The new `BlockBoxesWidth.html` document will be rendered as shown in Figure 3.27. Notice that although both of the shaded boxes have specified widths of `50%`, their actual widths are different because the percentage is applied to parent boxes that have different content widths. Also, keep in mind that the value of the `width` property defines the width of the content area of a box. Thus, the shaded boxes are both wider than half the width of their parents' content areas, because each box includes a total of 30 px of horizontal padding (15 px for each side) in addition to the content area.

In general, the value `auto` can be specified for any combination of `width`, `margin-left`, and `margin-right`. For example, if for a given box `margin-left` is `auto`, `width` is a specified length, and `margin-right` is 0, then the box will be right-justified within its containing block. See the CSS2 recommendation [W3C-CSS-2.0] for details on how CSS interprets other possible combinations for values of these three properties.

Block boxes also have a `height` property (not inherited) with an initial value of `auto`. As with the `width` property, the default block box height calculation described earlier can be overridden by specifying a value (length with units or percentage) for the block element's `height` property. If a percentage is specified, it is interpreted as a percentage of the value (if any) specified for the parent block's `height` property. If no value was specified for the parent's `height`, then the percentage specification is essentially ignored and treated as a specification of `auto`.

We're now ready to consider how inline boxes are rendered within a block box.

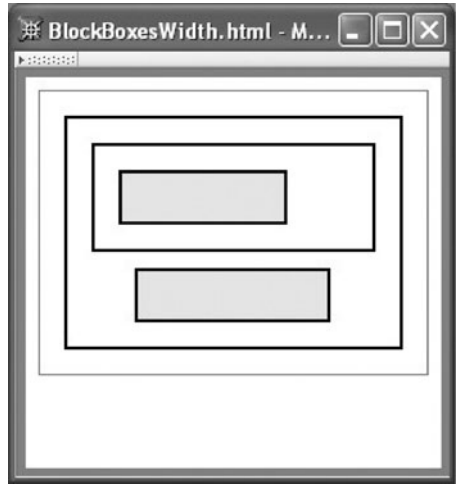

FIGURE 3.27 Rendering when widths of shaded boxes are specified as percentages. Lower box is centered because left and right margins are `auto`.

3.8.5 Simple Inline Boxes

Until now, we have thought of block boxes as either containing text (or more precisely, a stack of line boxes containing character cells) or containing other block boxes. But a block element can also contain inline elements, such as `span` and `strong`, and the browser will generate inline boxes corresponding to these elements. These inline boxes will be added to line boxes within the containing block box, much like text characters. In fact, we have already seen an example of this in Figure 3.20. In this section, we will look more closely at how browsers lay out *simple inline boxes*, that is, boxes for inline elements that contain only text or that are of type `img`. We'll briefly consider more complex inline elements in the next section.

We will first consider simple inline boxes consisting of text, generated by elements such as

```
<span style="font-size:36pt">BIG</span>
```

The height of the content area of such a box will be determined exactly as the height of a line box is determined: the height of a character cell (in the inline element's font) will be used unless the inline element's `line-height` property has a value other than `normal`, in which case this value will determine the content area height. Character glyphs are then added to the content area of the inline box as if they were being added to a line box, with half-leading added if needed to center the character cells vertically. Note that this process defines a baseline for the inline box: it is at the baseline height (as defined by the inline element's font) above the bottom edge of the content area of the inline box. We therefore now have a box that has a well-defined height (the height of the content area), width (the overall width of the box, including left and right padding, border, and margin lengths), and baseline height. These are essentially the same characteristics that a character cell has, so the browser can add this inline box to a line box as if it were a character cell, vertically aligning the baseline of the inline box with the baseline of the line box. If the inline box is too long to fit within the current line box, it may be broken on word boundaries into a sequence of shorter inline boxes that will each be added to a separate line box. If the top or bottom of an inline box extends beyond the corresponding edge of the line box, the line box height will automatically be expanded as needed to contain the inline box. If the line box height is extended upward, then the line box will be moved down within the containing block box by the same amount so that the line boxes within the block box will still effectively be stacked one on top of the other (Fig. 3.28).

You probably noticed that there is an asymmetry in how the height and width of the "character" representation of an inline box are determined. Specifically, the height of this "character" is determined by the content height of the inline box, but the width is determined by the overall box width. To illustrate, suppose we change the `d3` element of the document of Figure 3.23 as follows:

```
<div id="d3" class="shade">
  Here are
  <span style="border:dotted silver 10px">some</span>
  lines of text.
</div>
```

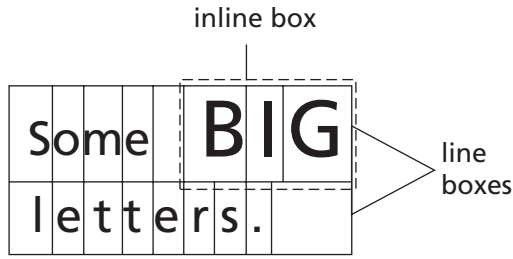

FIGURE 3.28 Two line boxes, the top box containing an inline box with a larger font size than the text elsewhere in the line boxes. Notice that the baselines are aligned in the top line box and that the line boxes stack despite having different heights.

Then the document will be rendered as in Figure 3.29. Notice that the word "some" is moved to the right to make room for the border, but the line height is unchanged, and in fact the border overlaps somewhat the text in the second line box.

The other type of simple inline element is an `img`. An `img` element is similarly treated, for rendering purposes, as a character to be added to a line box. However, the height and width of the "character" are the values specified for the `height` and `width` properties of the element (or, if these properties are not specified, the values of the `height` and `width` attributes, or, if these values are also not provided, then values contained within the image file itself). The baseline height of an image is always considered to be 0. Therefore, the bottom of the image will coincide with the baseline of the line box. As with inline boxes, if the top of an `img` box extends past the top of the line box, then the height of the line box

FIGURE 3.29 A span element with a border is added to the text.

will be increased to fit. Unlike other inline elements, the border, margin, and padding of an `img` element are considered part of the height of the image for purposes of determining the height of a line box containing the image.

The default vertical placement of an inline box within a line box can be overridden by specifying a value for the `vertical-align` property (not inherited) of the element generating the inline box. The initial value of `vertical-align` is `baseline`, which produces the default behavior described. Some other possible values are `text-bottom`, which aligns the bottom of the inline box with where the bottom of any character cell written into the line box would be located; `text-top`, which is similar except it aligns the top of the inline box with the top of the location for character cells; and `middle`, which aligns the vertical midpoint of the inline box with the *character* middle of the line box, a location that is one-half the ex height above the baseline of the line box. The CSS2 recommendation specifies several additional keyword values for `vertical-align`, but my copy of IE6 does not seem to handle them properly, so I will not cover them here.

In addition to these keywords, the value specified for `vertical-align` can be a length or a percentage (of the value of the `height` of an `img` element or the `line-height` of any other inline element). For both percentage and length specifications, a positive value indicates that the inline box should be raised by the specified distance relative to the default baseline position, and a negative value indicates that it should be lowered.

3.8.6 Nested Inline Boxes

How are text and boxes laid out within an inline element? We will consider the standard case for HTML, in which an inline element contains only text and other inline elements (see the CSS specification [W3C-CSS-2.0] for details on how a block box is handled within an inline box).

Actually, the layout of inline boxes and text within an inline box is essentially identical to the layout of inline boxes and text within a line box. In particular, the content area of the containing inline box is treated as a line box that initially has a height and baseline location defined by the font and `line-height` properties of the corresponding inline element. Characters and boxes are then added to this content area just as they would be to a line box, including having the vertical alignment of boxes determined by the `vertical-align` property. One difference is that the height of the content area is not adjusted to contain inline boxes whose top or bottom edges extend beyond the respective edges of the content area. However, when these boxes are eventually transferred from the content area to a line box, that line box's height will be adjusted. For example, changing the `d3` element of the document of Figure 3.23 to

```
<div id="d3" class="shade">
  Here are some lines of text.
  This
  <strong style="line-height:3">bold</strong>
  word has a line-height of 3.
</div>
```

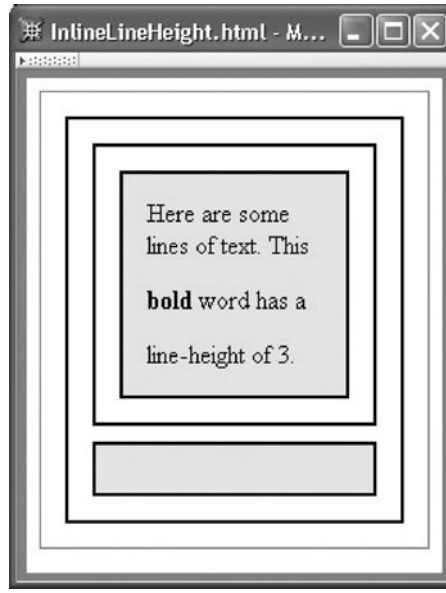

FIGURE 3.30 Effect of setting `line-height` on an inline element (the word "bold").

causes the browser to increase the height of the line box containing the word "bold," as shown in Figure 3.30. However, the heights of other line boxes are not affected.

3.9 Beyond the Normal Flow

What we have described so far is the default way in which a browser will format an HTML document. In this section, we'll learn that there are several CSS alternatives to the normal flow processing we have seen so far that can be used to control the position of boxes within a browser window. Three alternatives to normal flow positioning supported by both Mozilla 1.4 and IE6 are: relative positioning, in which the position is altered relative to its normal flow position; float positioning, in which an inline element moves to one side or the other of its line box; and absolute positioning, in which the element is removed entirely from the normal flow and placed elsewhere in the browser window.

We'll look at each of these three positioning schemes in detail in Sections 3.9.2–3.9.4. First, though, we'll briefly learn about the CSS properties used to indicate whether or not a box should use an alternative positioning scheme.

3.9.1 Properties for Positioning

The type of positioning for an element is defined by specifying two style properties. The `position` property takes the value `static` (the initial value) to indicate normal flow, `relative` and `absolute` to represent the respective flow positionings, or `fixed`, which is a special type of absolute positioning discussed in the exercises. The `float` property can be set for elements with either `static` or `relative` specified for `position`. Possible values for

FIGURE 3.31 HTML document using relative positioning to nudge text to the left.

`float` are `none` (the initial value), `left`, or `right`. The latter two values indicate that the element's box should move to the far left or far right side of the current line box, respectively. Neither `position` nor `float` is an inherited property.

Any element with a `position` value other than `static` is said to be *positioned*. If the `position` value of a positioned element is `absolute` or `fixed`, then it is said to be *absolutely positioned*; otherwise it is *relatively positioned*. Four (noninherited) properties apply specifically to positioned elements: `left`, `right`, `top`, and `bottom`. Each of these properties takes a value that is either a length, a percentage, or the keyword `auto` (the initial value). The meaning of these properties is explained for each positioning scheme in the following Sections 3.9.2–3.9.4.

3.9.2 Relative Positioning

Relative positioning is useful when you want to nudge a box a bit from the position where the browser would normally render it, and you want to do so without disturbing the placement of any other boxes. That is, all other boxes are positioned as if the nudged box had never moved.

For example, suppose that you were asked to produce the rendered HTML document shown in Figure 3.31. Notice that the first letter of each of the words "Red," "Yellow," and "Green" has a background that is partly shaded and partly not. This is not an effect that we would expect to produce in the normal flow processing model. But with relative positioning, it's easy: we use a style rule

```
.right { position:relative; right:0.25em }
```

and wrap each word to be moved in a `span` that specifies `right` for the value of its `class` attribute. As a side benefit, we get a little more separation between each word and the shaded box to its right than we would have had in normal flow processing, since the locations of these boxes is not affected by the relative shifting of the words.

Notice that for relatively positioned boxes, a positive value for the `right` property moves the box to the left by the specified amount. You can think of this as adding space to the right margin of the box. Recall that the initial value of `left` is `auto`; in this example, the corresponding computed value for the `left` property will be −0.25 em. Alternatively, if the style rule had been

```
.right { position:relative; left:-0.25em }
```

then the browser would have displayed the same rendering and `left` and `right` would have had the same computed values as they did with the original style rule. If, for some reason, both `left` and `right` have specified values other than `auto`, then the value of `left` will be used for the positioning, and the computed value of `right` will be set to the negative of the value of `left` (assuming `direction` is `ltr`). Similar rules apply to `top` and `bottom`, with `top` "winning" if both properties have non-`auto` values.

3.9.3 Float Positioning

Float positioning is often used when embedding images within a paragraph. For example, recall that the HTML markup

```
<p>
  <img
      src="http://www.w3.org/Icons/valid-xhtml10"
      alt="Valid XHTML 1.0!" height="31" width="88"
      style="float:right" />
  See
  <a href="http://www.w3.org/TR/html4/index/elements.html">the
    W3C HTML 4.01 Element Index</a>
  for a complete list of elements.
</p>
```

is part of the document displayed in Figure 2.13. The `float:right` declaration causes the image to be treated specially in several ways. First, the image is not added to a line box. Instead, the widths of one or more line boxes are shortened in order to make room for the image along the right content edge of the box containing the line boxes and image (the block box generated by the `p` element, in this case). The first shortened line box is the one that would have held the image if it had not been floated. Subsequent line boxes may also be shortened if necessary to make room for the image. Line boxes below the floated box extend to the full width of the containing block, producing a visual effect of text wrapping around the floated block (Fig. 3.32).

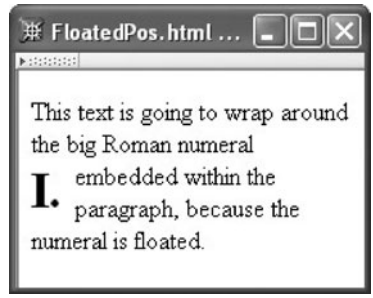

FIGURE 3.32 Wrapping of text around a floated box.

The markup used to generate this figure includes the following:

```
<style type="text/css">
  .bigNum { float:left; font-size:xx-large; font-weight:bold }
</style>
...
<p>
  This text is going to wrap
  around the
  <span class="bigNum">I. </span>
  big Roman numeral
  embedded within the paragraph, because the numeral is floated.
</p>
```

Notice that, unlike a relatively positioned box, the words "the" and "big" are not separated by the width of the floated `span` that separates these words in the source document. In other words, portions of this document that are part of the normal flow are formatted as if the floated element were not present at all (except for its effect on the width of line boxes). We say that float boxes are *removed from the normal flow* to indicate that making them float has an impact on how normal flow elements are rendered.

One small detail about floated boxes is that a floated inline box becomes a block box for display purposes; that is, an inline box's `display` property will have a computed value of `block` if the box is floated. This means, for example, that values can be specified for the `height` and `width` of a floated inline element.

For more details on float positioning, such as what happens when multiple floated boxes touch one another or when floated inline boxes extend below their containing block, see the CSS2 specification [W3C-CSS-2.0].

3.9.4 Absolute Positioning

Absolute positioning offers total control over the placement of boxes on the canvas. This power should be used with care: while you can create interesting visual effects this way, any information conveyed by these effects will generally not be available to users who are accessing the document in other ways (text-based browsing, speech synthesis, etc.). That said, there are certain times when it is useful to be able to place a box exactly where you want it.

For example, suppose that you would like to be able to easily add marginal notes to the left of paragraphs in an HTML document, as shown in Figure 3.33. Specifically, you'd like to be able to embed each note within the paragraph to which it applies, as in

```
<p>
  This second paragraph has a note.
  <span class="marginNote">This note is pretty long, so
  it could cause trouble...</span>
</p>
```

Then you would like the browser to automatically place the note next to the paragraph, starting vertically at the top of the paragraph.

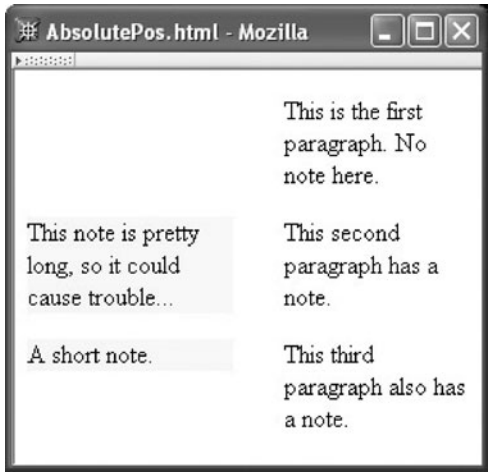

FIGURE 3.33 Absolute positioning used to create marginal notes.

This can be done easily using absolute positioning. When a box is absolutely positioned, as indicated by specifying `absolute` for the `position` property, the `left`, `top`, `right`, and `bottom` properties can be used to place the box relative to a containing block. The *containing block* for purposes of absolute positioning is defined as follows. First, we locate the nearest positioned ancestor of the absolutely positioned element (recall that a positioned element has `position` value other than `static`). If this ancestor is a block element (which we will assume; see the CSS2 recommendation for other possibilities), then the containing block is formed by the padding edge of the element's box, *not* the content edge as you might expect (the next example will show why this is a good choice of edge). If there is no positioned ancestor, then the initial containing block is used as the containing block.

Similar to relative positioning, specifying a value such as `10em` for the `left` property of an absolutely positioned box tells the browser to position the left outer edge of that box 10 ems to the right of the left (padding) edge of the containing block. Positive values for the other three positioning properties have similar effects, while negative values for these properties have the opposite effects (e.g., a negative value of `left` moves the box to the left rather than to the right). Like floats, if the box of an inline element is positioned absolutely, the box becomes a block box, and therefore can have its width set explicitly.

In our marginal note application, we would like each note to be positioned starting vertically at the top of the paragraph containing the note and horizontally to the left of the paragraph. This means that we want the paragraphs containing notes to be positioned, so that they can act as containing blocks for absolutely positioned elements. Also, we want to leave room next to paragraphs for the notes. So we will use the style rule

```
p { position:relative; margin-left:10em }
```

In relative positioning, if no value is specified for `left` or the other positioning properties, then the element is not moved. So the `position:relative` declaration has no visible effect. Instead, it marks `p` elements as positioned, making them eligible to act as containing blocks for absolutely positioned elements.

The rule for the `marginNote` class is longer, but not particularly difficult to understand. The rule is

```
.marginNote { position:absolute;
              top:0; left:-10em; width:8em;
              background-color:yellow }
```

This says to put any box belonging to the `marginNote` class and that is contained within a `p` element where we have said we would like it placed: beginning to the left of the top line of the paragraph. Notice that, for the given style rules, the outer left edge of a `marginNote` box will coincide with the outer left edge of the `p` element containing the `marginNote` element, regardless of the padding value of the `p` element (and assuming that the `p` element has no border). You can now see an advantage to positioning the box relative to the padding edge of the containing element rather than the content edge: we did not have to add in the padding distance in order to place our note in the margin.

As with float positioning, elements that are absolutely positioned are removed from the normal flow. This can be seen from the fact that there is no additional space between the second and third paragraphs in the figure. In contrast with float boxes, however, the normal flow will not flow around absolute boxes. In fact, absolute boxes will not flow around one another, either. For example, if we widen the browser window so that the second paragraph fits on a single line, the two marginal note boxes collide and the second obscures some of the text of the first (Fig. 3.34). This is another reason to use absolute positioning with care.

3.9.5 Positioning-Related Properties

A few additional CSS properties deserve mention in relation to positioning. The first of these is related to the overlay phenomenon that we have just discussed with absolute positioning.

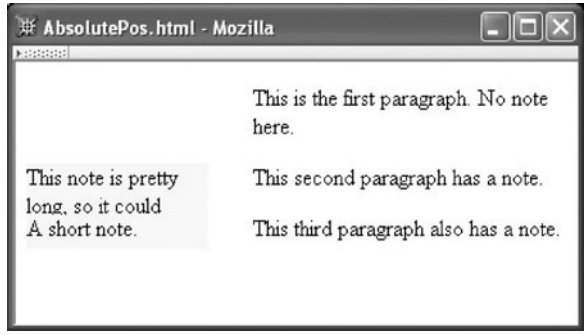

FIGURE 3.34 Absolutely positioned boxes can obscure one another.

FIGURE 3.35 An overlay of one box on top of another.

There may be times when you want several boxes to overlie one another, at least partially, but you want to be certain that they overlie in a certain order. For example, you may want one box to be drawn first, then a second box to be drawn as an overlay over top of the first box, possibly obscuring some or all of the first box. Figure 3.35 illustrates this: a box containing text is drawn first, and then an empty box (basically just a border) is drawn over top of the text box.

The `z-index` property can be used to define a drawing order for positioned boxes. In its simplest form, the root element of each positioned box is assigned an integer `z-index` value (this assumes that normal flow is followed for all of the descendants of a positioned box). If this is done, then drawing will proceed as follows. First, the box with the most negative `z-index` value (if any) will be drawn. Other boxes with negative `z-index` values will then be drawn on top of this box, proceeding in ascending order, until the box with the negative value closest to 0 has been drawn. At this point, all of the elements that are not positioned are drawn. Finally, all elements with positive `z-index` values are drawn, again in ascending order. The full definition of `z-index`, including how ties are broken between elements with the same `z-index` value (or no value at all) is contained in Section 9.9.1 of the CSS2 specification. But for most purposes, the simple use of `z-index` described should be sufficient to guarantee the drawing order you want.

To produce the effect shown in Figure 3.35, I used the following style rules:

```
#text { position:absolute; top:10px; left:10px;
        font-family:"Courier",monospace; letter-spacing:0.1ex;
        background-color:yellow;
        z-index:1 }
#overlay { position:absolute; top:10px; left:10px;
           width:1.1ex; height:4.5em;
           border:solid red 1px;
           z-index:2 }
```

The first rule is applied to a `div` containing the text, and the second to an empty `div`. The key item to notice is that the `z-index` value of the second `div` is greater than that of the first, so the second `div` is drawn on top of the first.

We discussed the `display` property earlier, but it has a keyword value that we did not cover. Specifying `none` for the value of `display` tells the browser to, for display purposes, treat the element and all of its descendants as if they did not exist. In other words, the

element is effectively removed from the normal flow and is also not displayed elsewhere. This is sometimes used with scripting to allow portions of a document to be easily added to or removed from the browser window.

The final property, `visibility`, is related. If the value of this property is `hidden`, then the element and its children—except those that specify `visible` for this property—will not be rendered. However, much as with relative positioning, the space occupied by the element will remain rendered. In other words, whether an element is visible or not does not affect the rendering of other nondescendant elements. Like `display`, this property is generally used in scripting contexts.

3.10 Some Other Useful Style Properties

While we have covered a significant portion of the CSS2 specification, we have also omitted a number of details and quite a few properties. A few of the remaining items are covered in this section.

3.10.1 Lists

The `list-style-type` property can be used to vary the default styles used for bulleted and numbered list items. In HTML, this property normally only applies to the `li` element type. However, it is inherited, so can be set on a parent `ol` or `ul` element in order to affect the style of all of that element's children. For bulleted lists, the values `disc`, `circle`, and `square` may be specified. For numbered lists, some of the normal values are `decimal` (1, 2, ...), `lower-roman` (i, ii, ...), `upper-roman` (I, II, ...), `lower-alpha` (a, b, ...), and `upper-alpha` (A, B, ...). A value of `none` can also be specified to indicate that no *marker* (leading bullet or number) should be generated for an `li` element.

A related `li` element type property is `list-style-image`, which has an initial value of `none`. If a URI is specified for this property (using the `url("...")` syntax described in Section 3.3), and if an image can be loaded from this URI, then this image will be used in place of the normal marker as specified by `list-style-type`. Once again, this property is inherited and is often set on parent elements rather than directly on `li` elements.

The `list-style-position` property can be used to change the location of the marker relative to the content of an `li`. A browser normally generates two boxes for an `li`: a special box to hold the marker, and a block box to hold the element content. If `list-style-position` has its initial value of `outside`, the marker box is outside the content block box. However, if the value is specified as `inside`, then the box generated for the marker will be the first inline box placed within the content block box. The visual effect in this case will be that the first line of the list item content is indented to make room for the marker.

Finally, the shortcut property `list-style` can be used to specify values for any or all of the mentioned properties, in any order.

3.10.2 Tables

For the most part, the box model properties discussed in this chapter, such as `border-style` and `padding`, can be used with elements related to tables (`table`, `td`, etc.), although their effect on table elements may vary slightly from their effect with other boxes. Furthermore,

the values top, bottom, middle, and baseline may be specified for the vertical-align property of td and th elements. A top value causes the top of the content of the cell to coincide with the top of the row containing the cell, bottom makes the bottom of the content coincide with the row bottom, and middle centers the content within the row. If multiple cells specify baseline, then the baselines of their first lines of text will be aligned with one another. If baseline is specified for a cell containing a single img element, then the bottom of the image is treated as the baseline of the cell for alignment purposes. The baseline cells are displayed as high as possible within the row while keeping the content of all cells within the row.

CSS2 also specifies two different models for how borders should be handled: a *separate* model in which each cell has its own border, and a *collapse* model in which adjacent cell borders may be collapsed into a single border. The user agent style sheets for both Mozilla 1.4 and IE6 apparently specify the separate model as the default. You can override the default by assigning a value of collapse or separate to the border-collapse style property of a table element. Full details of CSS support for tables are well covered in Chapter 17 of the CSS2 specification, which should not be hard to understand if you have mastered the material in this chapter. So I will not cover tables further here.

3.10.3 Cursor Styles

CSS specifies a number of different cursor styles that can be used. The initial value for the cursor property is auto, which allows the browser to choose a cursor style as it deems appropriate. Mozilla 1.4, for example, will display a text cursor when the mouse is over text, a pointing finger when over a hyperlink, an arrow and hourglass when a link is clicked and a new document is loading, and an arrow in most other contexts. Other keywords that can be used to specify a value for the cursor property include default (often an arrow), text (used over text that may be selected), pointer (often used over hyperlinks), and progress (often used when a link is clicked). Some other keywords produce cursors that would normally be seen outside the browser client area, such as move (used to indicate window movement), various resizing arrows (e-resize, ne-resize, sw-resize, and other compass points), wait (program-busy, often an hourglass), and help (often an arrow with a question mark).

Like some other properties, cursor is normally used by scripts running within the browser, a topic covered in Chapter 5.

3.11 Case Study

We'll now create a style sheet suitable for our blogging application and also modify our previous view-blog document, formatting it using style properties rather than tables. For colors, fonts, and to a lesser extent spacing, our style sheet will be similar to the Oldstyle style sheet available as part of W3C's Core Styles project (http://www.w3.org/StyleSheets/Core/), giving us an opportunity to see some real-world styling (and something that looks much better than anything I would have produced). Ultimately, the style sheet and markup changes presented in this section will transform the view-blog page from that shown in Figure 2.31 to that shown in Figure 1.12.

```
/* The W3C Core Styles, Copyright © 1998 W3C (mit, inria, Keio). All
Rights Reserved. W3C liability, trademark, document use and software
licensing rules apply. See
http://www.w3.org/Consortium/Legal/ipr-notice.html

This is a modified version of the Oldstyle style sheet available
at http://www.w3.org/StyleSheets/Core/
A list of modifications made to the original Oldstyle style sheet is
at http://www.mathcs.duq.edu/~jackson/webtech/OldstyleMods.txt */
        /* Elements */
body    {
        background-color: #fffff5;
        font-family:    "Verdana", "Myriad Web", "Syntax", sans-serif;
        font-size-adjust: .58;
        margin:         1.5em 5%;
        }
p       {
        margin-top:     .75em;
        margin-bottom:  .75em;
        }
h1      {
        font-family:    "Georgia", "Minion Web", "Palatino",
                        "Book Antiqua", "Utopia", "Times New Roman",
                        serif;
        font-size-adjust: .4;
        font-size:      2.0em;
        font-weight:    600;
        color:          #C00;
        }
hr      {
        height:         1px;
        background-color: #999;
        border-style:   none;
        }
```

FIGURE 3.36 Style rules for nonanchor elements.

The style sheet rules we create will all be stored in a file named `style.css`. Each of the HTML documents for the application will be modified to include this style sheet file using markup such as

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

in the `head` element of the document. Note that the relative URL used in the `href` attribute assumes that the HTML files and `style.css` file all reside within the same directory.

Figure 3.36 shows the first portion of the `style.css` file. The four rules shown each have a selector string that is a type selector. The first rule states that the background color of the body will be slightly off-white (recall that white is `#ffffff`). The default font family (unless overridden by another element) will be Verdana or, if Verdana is not available to the browser, one of three other font families listed (the final option is the generic sans-serif

```
        /* Hyperlinks */
a       {
        font-weight:    bold;
        text-decoration: none;
        }
a:link  {
        color:          #33F;
        background:     #fffff5;
        }
a:visited {
        color:          #93C;
        background:     #fffff5;
        }
a:active {
        color:          black;
        background:     #ccf;
        }
a:hover {
        color:          #fffff5;
        background:     #33F;
        }
```

FIGURE 3.37 Style rules for anchor elements.

family). The `font-size-adjust` property, which is not supported by IE6, has an effect if the first font family is not available. Given an appropriate specified value, the size of the selected font family is scaled so that its ex height is roughly the same as that of the first font family. This should make the letters appear to be about the same size regardless of the font actually used. Finally, notice that the left and right margins of the body are set at 5% of the width of the browser window, providing side margins that change as the window width changes.

The `p` and `h1` rules are reasonably straightforward, although the `h1` rule does use a numeric value for its `font-weight` property. This value corresponds to two steps bolder than `normal` and one step lighter than `bold`. The `hr` rule turns off the border, which the user agent style sheets for both IE6 and Mozilla 1.4 apparently turn on, and instead displays only a 1-pixel-high gray line. Note the use of both three-digit and six-digit color values.

Figure 3.37 shows the style rules related to hyperlinks (anchor elements). The first rule makes links bold and removes the underlining that would normally be associated with links. The remaining pseudo-class rules change the text and background colors of a hyperlink depending on its status, as described earlier.

So far, we have been slightly adapting the W3C Oldstyle Core style for our purposes. We next want to create a number of style rules specifically for the view-blog document. Recall that this document has three overall segments: an image above two segments, the blog entries on the left, and some navigation hyperlinks on the right. It is natural to lay out these segments by creating `div` elements and positioning them using CSS. Figure 3.38 shows the structure of the body of the new document (still called `index.html`).

The corresponding style rules are given in Figure 3.39. The first two rules center the top image and the body (main portion) of the document, which contains the blog entries and navigation links. It also fixes a width for the body portion of the document. This value

```
<div class="imgcentered">
  <!-- Banner image -->
  <img src="banner.gif" width="438" height="120"
       alt='"My Own Blog!" Banner' />
</div>
<div class="bodycentered">
  <div class="leftbody">
    <!-- Blog entries -->
    <div class="entry">
      ...
    </div>
    <hr />
    <div class="entry">
      ...
    </div>
  </div>
  <div class="rightbody">
    <!-- Side information -->
    ...
  </div>
</div>
```

FIGURE 3.38 Structure of the HTML document for the view-blog page using CSS rather than a table for layout.

```
        /* Classes for view-blog page */
.imgcentered {
        width:          438px;
        margin-left:    auto;
        margin-right:   auto;
        }
.bodycentered {
        width:          660px;
        margin-left:    auto;
        margin-right:   auto;
        }
.leftbody {
        width:          410px;
        float:          left;
        }
.rightbody {
        width:          230px;
        float:          right;
        }
.entry  {
        margin-top:     .75em;
        margin-bottom:  .75em;
        }
```

FIGURE 3.39 Style rules for `div` elements used for positioning.

```
<div class="datetime">AUGUST 9, 2005, 5:04 PM EDT</div>
<div class="entrytitle">I'm hungry</div>
<div class="entrybody">
  <p>
    Strange.  I seem to get hungry at about the same time
    every day.  Maybe it's something in the water.
  </p>
</div>
<hr />
```

FIGURE 3.40 Markup for a blog entry.

is narrow enough to be viewed without horizontal scrolling on almost any modern monitor, yet wide enough to display a reasonable number of words per line in the blog entries. The `div`'s for the entries and navigation links are then floated to the left and right, respectively, within this body `div`. Notice that the sum of the widths of these `div`'s is 20 px less than the width of the containing `div`, providing some visual separation between the blog entries and the navigation links. The final rule defines vertical spacing between blog entries, or more specifically, between blog entries and the horizontal rule separating them.

We can also use CSS to style the components of a blog entry. For example, the markup for the first entry of our example is shown in Figure 3.40, and Figure 3.41 gives style rules corresponding to this markup. Given the earlier discussion, these rules should not need any explanation.

Finally, let's use CSS to add a "displayed quote" feature, as illustrated in Figure 3.42. The basic idea is that if markup such as

```
.datetime {
        color:          #999;
        font-size:      x-small;
        }
.entrytitle {
        /* based on h2 of Oldstyle */
        font-family:    "Georgia", "Minion Web", "Palatino",
                        "Book Antiqua", "Utopia", "Times New Roman",
                        serif;
        font-size-adjust: .4;
        font-size:      1.75em;
        font-weight:    500;
        color:          #C00;
        margin-top:     .25em;
        }
.entrybody {
        font-size:      small;
        }
```

FIGURE 3.41 Style rules used for formatting components of a blog entry.

FIGURE 3.42 Example of a displayed quote (in a preview window, which suppresses the navigation links).

```
<span class='dquote'>It's more important than that.</span>
```

is included within text, then the content of the `span` will be displayed within the entry and also floated to the left of the enclosing text, enlarged, and enclosed within a three-sided, dotted border. This displayed-quote feature is not foolproof: if the `span` is included near the bottom of the text, then it might overlap with the next entry, since a floated element is taken out of the normal flow. But, if used carefully, it provides an interesting effect.

Figure 3.43 gives a suitable rule for producing the displayed-quote effect. One thing to notice is that the three-sided border was created using two declarations, and that the order of these declarations is important (the second rule overrides a portion of the first due to the cascade rules).

```
        /* For displaying a quote */
.dquote {
        float:          left;
        font-size:      larger;
        padding:        1px;
        margin-right:   5px;
        width:          10em;
        border-style:   dotted;
        border-left-style: none;
        border-color:   #999;
        }
```

FIGURE 3.43 Style rules for `span` element used to display a quote.

CHAPTER 4

Client-Side Programming
The JavaScript™ Language

This chapter will introduce you to the JavaScript™ programming language, which is supported by nearly all traditional web browsers in use today. Our focus in this chapter will be on the study of JavaScript as a programming language, largely independently of how it might relate to a web browser. In the next chapter we will cover in more detail how JavaScript programs can interact with documents contained in a web browser by, for example, changing the style or even the content of a document. Therefore, this chapter will not include a section on the case study blogging application, which we defer to the next chapter.

While you may know that JavaScript is often used for relatively simple tasks within browsers, you should not be misled into thinking that it is therefore a language of limited power. To the contrary, as we will learn, it is in several respects a particularly powerful language, and in fact incorporates some programming language concepts that may be new to you. So, in addition to being useful for web programming, studying JavaScript may broaden your understanding of programming languages in general.

4.1 History and Versions of JavaScript

JavaScript was initially developed by Brendan Eich as part of the Netscape 2.0 release. The language was called LiveScript for a while in its early stages. But on December 4, 1995, before the final release of Netscape 2.0, the language was publicly announced as JavaScript. This name change was apparently intended to link the scripting language with the rising popularity of Sun's Java programming language, but has caused a fair amount of confusion over the years. Although there are strong similarities between the core syntaxes of JavaScript and Java (and C++, for that matter), and although JavaScript was designed to interact with Java applets, there are also tremendous differences between JavaScript and Java.

JavaScript became especially popular after the 1.1 release as part of Netscape's 3.0 browser. This version of JavaScript allowed web page developers to produce the familiar rollover effect: as the mouse moves over images, the images change (we'll learn how to produce this effect in Chapter 5). Other JavaScript versions followed as part of later Netscape browser releases. Meanwhile, Microsoft introduced the JScript® programming language (the JavaScript name was owned by Sun) in July of 1996, and this language similarly went through a series of revisions as new versions of Internet Explorer were released.

To confuse matters somewhat more, there is yet another flavor of JavaScript, known as ECMAScript. Soon after announcing JavaScript, Sun and Netscape approached an organization then known as the European Computer Manufacturers Association (ECMA)

to produce a standard for JavaScript. The third edition of the ECMA-262 standard [ECMA-262] was released in 1999 and has been a significant help in bringing JavaScript and JScript closer together. JavaScript 1.5, the version implemented in Mozilla 1.4, conforms with ECMAScript Edition 3 (barring bugs), and JScript versions 5.5 (the version implemented in IE5.5) and beyond also appear to be compliant. Other browsers, such as Opera and Safari, also support the ECMAScript standard.

I will therefore focus on ECMAScript Edition 3 in this chapter, and refer to this simply as "JavaScript." All of the examples presented run in both Mozilla 1.4 and IE6, except as noted.

4.2 Introduction to JavaScript

I'd like to begin with a "Hello World!" JavaScript program. There's just one problem: JavaScript itself has no statement for performing output. Instead, the JavaScript language specification leaves it up to browsers to supply output (and input) methods. Happily, certain methods are supported by almost all modern browsers, even though technically they are not part of any standard. So we'll use one of these methods, `alert()`, to write a JavaScript "Hello World!" program (the `window` object is discussed in Section 4.3.1):

```
window.alert("Hello World!");
```

We can execute this program by referencing a file containing it from a `script` element within an HTML document. For example, if the JavaScript code given is placed within a file named `JSHelloWorld.js` in the same directory as the HTML document of Figure 4.1, then loading this document into a browser will cause the code to be executed. If this document is loaded into Mozilla 1.4, then the pop-up window shown in Figure 4.2 will appear. The browser window will be unresponsive until this *alert box* is dismissed, either by clicking the OK button or by closing the window (in Windows, as shown here, by clicking the × icon in the upper right corner of the alert box window). The client area of the browser window itself will be completely empty after the alert box is dismissed.

```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      JSHelloWorld.html
    </title>
    <script type="text/javascript" src="JSHelloWorld.js">
    </script>
  </head>
  <body>
  </body>
</html>
```

FIGURE 4.1 HTML document that loads and executes the JavaScript program in file `JSHelloWorld.js`.

FIGURE 4.2 Alert box generated by a JavaScript statement.

As with output, there is no input statement in JavaScript itself. Again, though, most browsers implement a `prompt()` method that can be called as illustrated by the following code:

```
var inString = window.prompt("Enter JavaScript code to be tested:",
                             "");
```

This pops up a window that displays the value of its first string argument and also provides a text box in which a user can enter information and that initially contains the string given by the second argument (Figure 4.3). The value returned by the `prompt()` method is the string entered by the user, assuming that the user clicks OK after entering the string (fuller details of `prompt()` are provided in Chapter 5).

Now that you've been introduced to JavaScript, let's briefly consider some of the ways in which developing JavaScript programs can differ from more traditional (Java/C++) program development before delving into details of the language.

4.3 JavaScript in Perspective

This section provides an overview of how JavaScript relates to various categories of programming languages and how this impacts development of JavaScript programs. We'll also look briefly at a few JavaScript software development tools so that you can experiment with your own code as you read the rest of the chapter.

FIGURE 4.3 Prompt pop-up window with user input entered.

4.3.1 Scripting Languages

As you probably noticed, we did not need to compile the "Hello World!" JavaScript program before executing it. Programming languages that do not need to be compiled before execution are known as *interpreted* languages. Software that reads and executes a program written in an interpreted language is known as an *interpreter*. Most modern browsers contain a JavaScript interpreter.

Programs written in interpreted languages are generally easier to maintain than programs written in compiled languages such as Java. For example, there's no need to remember to recompile interpreted programs after modifying them, and there are fewer files to manage, since there are no compiled versions of source files. Programs written in an interpreted language are also often simpler than equivalent programs written in a traditional compiled language. The one-line "Hello World!" program is one example of this. As another example, consider that while Java has a wide variety of numeric data types (`int`, `float`, `double`, etc.), JavaScript has a single Number data type. So some of the Java complexities of choosing an appropriate numeric data type and casting between types are not present in JavaScript.

There are, of course, some advantages to compiled languages. First, interpreted programs generally take longer to execute than similar programs written in compiled languages. One reason for this is that something like compilation has to occur during execution of an interpreted language, whereas it occurs before execution in a compiled language. Similarly, the simplicity of an interpreted language such as JavaScript comes at the cost of less efficient code. For example, the reason that Java provides many different numeric data types is so that the compiler can choose an implementation for an arithmetic operation that is optimized for the problem at hand. JavaScript, on the other hand, uses a generic—and therefore generally less than optimal—implementation for every arithmetic operation.

Many interpreted languages are, like JavaScript, also scripting languages. A *scripting language* is a specialized programming language designed to be used to automate tasks within a particular software environment. For example, if you have used Linux, you may have heard of *shell scripts*. These are generally small programs that issue Linux shell commands (the sorts of commands you type in within a shell prompt window) in order to cause the operating system to perform certain functions. For instance, shell scripting is a convenient way to change the file extension on all of the files within a directory. This could be done by hand, but scripting can be used to automate what might otherwise be a tedious process.

JavaScript is also intended to run within a certain software environment, or more precisely, within many software environments. The original LiveScript language was intended to be used as part of Netscape's LiveWire™ web server software as well as within Netscape browsers. JScript can similarly be run within Microsoft's IIS web server as well as within Internet Explorer browsers. In addition, many Microsoft operating systems also provide a Script Host application that essentially allows the Windows equivalent of Linux shell scripting to be performed using JScript. Certain other software applications also contain JavaScript interpreters, such as the Dreamweaver web-development tool.

To accommodate this diversity of script environments, each JavaScript implementation effectively consists of two software components. The primary component is a *scripting engine*, which includes a JavaScript interpreter as well as core ECMAScript functionality that must be present regardless of the script environment. The JavaScript scripting engine is

the focus of this chapter. The second component of a complete JavaScript implementation is a *hosting environment*, which provides environment-specific capabilities to JavaScript programs running within the environment. The `alert()` and `prompt()` methods are part of the hosting environment of Mozilla and IE6. These and many other hosting environment features found in most modern browsers will be covered in Chapter 5.

All of the hosting environment capabilities as well as many scripting engine capabilities are provided through objects. Objects that are required by the ECMAScript definition and therefore provided by the scripting engine are known as *native* objects, and those provided by the hosting environment are known as *host* objects. In the JavaScript "Hello World!" example, `window` is the name of a native object (even though some of its methods are provided by the hosting environment). A *built-in* object is a native object that is automatically constructed during scripting engine initialization rather than being constructed as a result of a call to a constructor during program execution. The `window` object is an example of a built-in object, since it can be used immediately without constructing it explicitly.

4.3.2 Writing and Testing JavaScript Programs

JavaScript code can be written using various tools. Many JavaScript programs are small; for these, a simple text editor such as Windows Notepad is probably sufficient. If you use the GNU® Emacs editor (which runs on multiple platforms), it will colorize your JavaScript code if you include the following statements in the `.emacs` initialization file:

```
;; Turn on C++/other highlighting
(global-font-lock-mode)

;; Turn on a javascript mode for .js files
(require 'generic-x)
(add-to-list 'generic-extras-enable-list 'javascript-generic-mode)
```

For larger projects, commercial tools tailored to JavaScript development are available. For example, Microsoft's Visual InterDev® Web development system includes various JavaScript productivity features such as automatically providing lists of object properties and methods.

Once you've entered your JavaScript code, the next step is to execute your program, since JavaScript is not compiled and therefore does not require a compilation step before execution. While the lack of compilation streamlines the development process, it also means that there are no compiler error messages to alert you to potential problems before you execute a JavaScript program. Instead, any error messages are produced when you execute the program. What's more, to keep the language simple and flexible, JavaScript forgoes many safety features—commonly found in compiled languages—that help programmers to avoid hard-to-debug errors. For instance, it is not necessary to declare variables in JavaScript, which might seem like a programmer-friendly feature at first. But as an illustration of the problems this can cause, note that if you misspell the variable name on the left-hand side of a JavaScript assignment statement, then the scripting engine will silently create a new variable with the misspelled name. Obviously, you'll want to keep these sorts of problems in mind when you are debugging your JavaScript code.

As illustrated in Figure 4.1, executing a browser-based JavaScript program requires opening an HTML document in your browser. You can write your own documents to run each of your programs, or you can edit an existing document such as the one in Figure 4.1 so that it loads the file containing your program rather than `JSHelloWorld.js`. A simpler alternative using software available from the textbook Web site is described in the next section.

If you run your program and it does not behave as expected, the first step is to check for any browser-generated error messages. The details of this step of course depend on the browser you're using for your tests. If you have installed a full version of Mozilla as described in Appendix A, then error messages appear in its JavaScript console. You can open this console by selecting **Tools|WebDevelopment|JavaScript Console** from the browser menu. This console will display any syntax errors or run-time exceptions encountered (the code implementing Mozilla's features sometimes throws exceptions to this window, so I suggest opening the console and clicking its Clear button before running your JavaScript code). A nice feature of the Mozilla JavaScript console is that its error messages normally contain a hyperlink that, if clicked, will open a window displaying your JavaScript code and highlighting the offending line. This window is not an editor, though, so you'll still need to go back to your development software in order to make changes to your code.

IE6's response to JavaScript errors depends on the browser options you have selected as well as on whether or not certain other software is available on your system. A JavaScript problem is often indicated by the presence of a small yellow exclamation-mark icon in the lower left corner of the browser window (on the status bar). Clicking this icon will pop up a dialog box describing the error. However, instead of relying on this icon, I suggest selecting **Tools|Internet Options . . .** from the IE6 menu and click on the Advanced tab in the pop-up window that appears. Then check the "Display a notification about every script error" checkbox. The browser should then automatically pop up the error dialog box whenever a JavaScript syntax or run-time problem is encountered. Note that if you have debugging software installed and debugging enabled (also controlled via **Internet Options . . .**), instead of seeing this error window you will be asked whether or not you want to begin a debugging session (see Section 4.13 about JavaScript debuggers).

The Details section of the IE6 error window tells the type of error and its location (file name, line within the file, and character within the line). If multiple errors are encountered, the pop-up window will contain information about one of the errors and a Next button that will allow you to step through other error messages.

If your browser does not produce any error messages but your program does not behave as expected, you'll need to locate the source of the problem. The same sorts of bug location techniques that apply to other languages apply to JavaScript. Some common techniques are to desk check (simulate by hand) your program, add output statements so that you can see the results of intermediate computations, or run an interactive debugger. You'll need to understand certain JavaScript concepts in order to use an interactive debugger effectively, so this approach is covered in Section 4.13.

After locating the source of errors and changing your code, you can retest your program by reloading the HTML document containing the code into your browser. If your browser is currently accessing this document, this is as simple as clicking the Reload (Mozilla) or Refresh (IE6) button in the browser toolbar.

4.4 Basic Syntax

Much of JavaScript's basic syntax is identical to that of Java or C++. To illustrate these similarities, consider the JavaScript program of Figure 4.4, which plays the high-low guessing game with the user. That is, the program randomly picks a whole number between 1 and 1000, and the user attempts to guess the number. After each guess, the user is either told that the guess was correct or is told whether the guessed number is too high or too low. This continues until the user guesses the computer's number.

Let's begin by considering some ways in which JavaScript syntax is similar to Java and C++. First, comments in JavaScript are the same as those in Java: `//` begins a comment that ends at the end of the line containing the comment, and `/*` and `*/` may be used to contain multiline comments (not shown in this example). Also, JavaScript's use of braces to enclose statement blocks and its syntax for assignment statements and control constructs such as `if` and `while` should be familiar. Another similarity is that semicolons (`;`) can be used at the end of JavaScript statements as they are in Java and C++. Technically, most JavaScript statements do not require this, but for compatibility with compiled languages and to avoid possible hard-to-locate errors, I suggest always using semicolons just as you would in Java.

JavaScript uses the familiar dot notation to represent properties and methods of objects. So we can infer, for instance, that `random()` is a method of the `Math` object (which like `window` is a built-in object). This method returns a randomly chosen random

```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
while (guess != thinkingOf)
{

  // Evaluate the user's guess
  if (guess < thinkingOf) {
     guess = window.prompt("Your guess of " + guess +
                           " was too low.  Guess again.", "");
  }
  else {
     guess = window.prompt("Your guess of " + guess +
                           " was too high.  Guess again.", "");
  }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

FIGURE 4.4 JavaScript high-low program.

decimal number from 0 up to but not including 1, and `Math.ceil()` returns the smallest integer greater than or equal to its argument. The net effect of applying these methods in the program is that a random integer between 1 and 1000 is assigned to `thinkingOf`.

Some other similarities between Java and JavaScript that will be covered in more detail later in this chapter include the following. First, variable names in JavaScript are case sensitive, and the rules for choosing variable names are similar to those used in C++ and Java (although JavaScript of course has a somewhat different collection of reserved words that cannot be used as variable names). The syntax for argument passing in method calls and for array access (not shown in the example program in Figure 4.4) is also similar to Java's.

In addition, most of the strings used to represent various logical, string, and numeric operators, such as `!=`, `+`, and `*`, are identical to those used in Java and C++. (Recall that *operators* in a programming language appear within expressions, where an *expression* is a portion of a program that can be *evaluated* in order to obtain a value. For example, the expression `1+2` contains the operator `+` and, when evaluated, returns the value `3`. The `+` operator has two *operands*, `1` and `2`.)

While JavaScript thus has many similarities with Java and C++, there are also some significant differences. First, there is no `main()` function or method. Instead, the JavaScript scripting engine simply begins evaluating the first line of code it reads. Second, the variable declarations (statements beginning with the keyword `var`) do not specify a data type for the variables declared. Typed variables are another safety feature that is missing in JavaScript: if you assign a string value to a variable that you were intending to treat as a numeric variable, JavaScript will silently assign the string value to the variable. On the other hand, this can be viewed as adding to JavaScript's flexibility: if you want to (I don't recommend this), you can deliberately store a Number value in a variable at one time in a program's execution and at a later time store a String value in the same variable. In computer science terminology, JavaScript is a *dynamically typed* language whereas Java and C++ are *statically typed*.

This brings us to a third significant difference: JavaScript performs many conversions between data types automatically. In this program, the `prompt()` method returns a string value, so the values assigned to `guess` are all strings. But the value of `thinkingOf` is a number. In Java, an attempt to apply a relational operator such as `<` or `!=` to compare String and int operands would result in a compile error. But JavaScript automatically converts the string argument to a number so that the comparison can be performed. We'll learn more about JavaScript's type conversion rules later. For now, note that this automatic conversion feature, while convenient, comes at the expense of run-time computation required to detect the need for conversion and to choose an appropriate conversion to apply.

The JavaScript program of Figure 4.4—as well as every other example in this chapter that begins with a comment consisting of a file name—is part of the `chapter 4` directory of the download package available from the textbook supplements Web site (the URL is given in the Preface). The download package also includes an HTML document named `TestJs.html` that you can use to load and execute this chapter's examples. Simply open the `TestJs.html` file in your browser; in Mozilla, you can do this by selecting **File|Open File . . .** from the menu and navigating to the `TestJs.html` file in the file selector window that appears. When the file loads, the browser's client area will be blank, but a pop-up prompt box (as in Figure 4.3) will appear. Enter the name of the example file (such as `HighLow.js`) in this prompt box, and click OK. Your browser will then load the example file and execute the JavaScript code that it contains.

Now that you have some familiarity with the basic syntax of JavaScript, the next several sections will explore how JavaScript handles specific programming issues, including variables and data types, statements and operators, objects, arrays, and functions (methods).

4.5 Variables and Data Types

As previously mentioned, variables do not have data types in JavaScript. However, every variable has a value, and every value belongs to one of six JavaScript data types. Every numeric value is of type Number, string values are of type String, the literals `true` and `false` represent the two Boolean values, the literal `null` represents the one value of type Null, and every object is of type Object. The remaining type, Undefined, is the type of the value represented by any variable that has been declared but has not yet been assigned a value. Attempts to use such variables in Java will be flagged as errors by the Java compiler, but it is up to the programmer to avoid this situation in JavaScript. The five JavaScript data types other than Object are sometimes referred to as *primitive data types*.

In JavaScript, `typeof` is an operator that provides information about the data type of a value stored in a variable, as shown in Table 4.1. For example, the JavaScript code

```
// TypeOf.js
var i;
var j;
j = "Not a number";
alert("i is " + (typeof i) + "\n" +
      "j is " + (typeof j));
```

produces the output of Figure 4.5. A common use of the `typeof` operator is to test that a variable has been defined before attempting to use it. Notice, though, that `typeof` returns the string `undefined` for several different reasons, not only when a variable contains the value of type Undefined.

As mentioned previously, JavaScript can perform automatic conversions between data types. Tables 4.2, 4.3, and 4.4 show rules used to convert to Boolean, String, and Number types, respectively. NaN ("not a number") and Infinity, mentioned in these tables,

TABLE 4.1 Values Returned by `typeof` for Various Operands

Operand Value	String `typeof` Returns
`null`	`object`
Boolean	`boolean`
Number	`number`
String	`string`
Native Object representing function	`function`
Native Object not representing function	`object`
Declared variable with no value	`undefined`
Undeclared variable	`undefined`
Nonexistent property of an Object	`undefined`

FIGURE 4.5 Result of executing `TypeOf.js`.

TABLE 4.2 Data Type Conversions to Boolean

Original Value	Value as Boolean
Undefined	`false`
`null`	`false`
`0`	`false`
NaN	`false`
"" (empty string)	`false`
Any other value	`true`

TABLE 4.3 Data Type Conversions to String

Original Value	Value as String
Undefined	`undefined`
`null`	`null`
`true`, `false`	`true`, `false`
NaN	`NaN`
Infinity, −Infinity	`Infinity`, `-Infinity`
Other Number up to ≈20 digits	Integer or decimal representation
Number over ≈20 digits	Scientific notation
Object	Call to `toString()` method on the object

TABLE 4.4 Data Type Conversions to Number

Original Value	Value as Number
Undefined	`NaN`
`null`, `false`, "" (empty string)	0
`true`	1
String representing number	Represented number
Other String	`NaN`
Object	Call to `valueOf()` method on the object

are special Number values that can occur if, for example, you divide by 0 (0/0 produces NaN while any positive number divided by 0 produces Infinity). `toString()` (Table 4.3) and and `valueOf()` (Table 4.4) are methods that are defined (either implicitly or explicitly) for every JavaScript object. We'll have more to say about these methods for some of JavaScript's built-in objects later in this chapter.

We learned earlier that, when comparing a String and a Number value with a comparison operator such as <, JavaScript will automatically convert the String value to a Number before performing the comparison. Some other common situations in which the scripting engine performs automatic type conversion are:

- The condition portion of statements such as `if` and `while` is automatically converted to Boolean.
- The value of the accessor portion of an array access (for example, the `3` in an expression such as `records[3]`) is automatically converted to String.

Additional type-conversion scenarios are covered in Section 4.7.

The strings used to name variables are called *identifiers*. Any string that begins with a letter or underscore (_), consists only of these characters and digits, and is not one of the reserved words listed in Figure 4.6 is a valid identifier in JavaScript. (The JavaScript specification also allows certain other characters in identifiers; see Section 7.6 of the ECMAScript specification [ECMA-262] for full details.) As mentioned previously, identifiers are case sensitive.

Finally, if a JavaScript program assigns a value to a variable without first declaring the variable (with a `var` statement), then the scripting engine will automatically create the variable. So, for example, the following is a legal JavaScript program that will create a variable, assign it a value, and then display the value in an alert box:

```
testing = "Does this work?";
window.alert(testing);
```

However, I strongly recommend using `var` to declare your variables. First, using `var` consistently to create variables makes it easier to determine all of the variables used in your program, which should facilitate future program maintenance. Moreover, you can avoid

```
abstract     boolean   break      byte        case          catch
char         class     const      continue    debugger      default
delete       do        double     else        enum          export
extends      false     final      finally     float         for
function     goto      if         implements  import        in
instanceof   int       interface  long        native        new
null         package   private    protected   public        return
short        static    super      switch      synchronized
this         throw     throws     transient   true          try
typeof       var       void       volatile    while         with
```

FIGURE 4.6 JavaScript reserved words.

certain subtle errors if you use `var` rather than assignments to create all variables (but I'll leave the details to the exercises).

4.6 Statements

JavaScript has three types of statements. First, there is the *expression statement*, that is, a statement that consists entirely of an expression. Typical expression statements include assignment and increment statements, such as

```
i = 5;
j++;
```

The second type is the *block statement*, which is simply a set of statements enclosed in braces { and }. Finally, there are statements that begin with keywords such as `var` and `if`, which are (of course) called *keyword statements*.

We have already met the `var` statement, which is JavaScript's one statement for declaring a variable. Syntactically, the `var` keyword is followed by a variable declaration list, which is a comma-separated list of identifiers. Each identifier in a variable declaration list may optionally be followed by an equals sign (=) and then an arbitrary expression that acts as an initializer. So the JavaScript code

```
var i, msg="hi", o=null;
```

declares three variables named `i`, `msg`, and `o`, assigning them values of types Undefined, String, and Null, respectively.

Table 4.5 lists the JavaScript keyword statements that have direct analogs in Java and C++. I assume that you're already familiar with these statements as they're used in Java, so I won't cover them in detail here. Instead, Figure 4.7 illustrates the use of several of

TABLE 4.5 Some JavaScript Keyword Statements

Statement Name	Syntax
If-then	`if (`*expr*`)` *stmt*
If-then-else	`if (`*expr*`)` *stmt* `else` *stmt*
Do	`do` *stmt* `while (`*expr*`)`
While	`while (`*expr*`)` *stmt*
For	`for (`*part1* `;` *part2* `;` *part3*`)` *stmt*
Continue	`continue`
Break	`break`
Return-void	`return`
Return-value	`return` *expr*
Switch	`switch (`*expr*`) {` *cases* `}`
Try	`try` *try-block catch-part*
Throw	`throw` *expr*

these statements. This program will successively display three alert boxes. The first two boxes will contain the message `A JavaScript exception can be anything`, and the third will contain `i = 3`. Comments in the code point out small differences between some of the JavaScript keyword statements and their Java analogs.

4.7 Operators

In this section, we'll cover many of JavaScript's operators. Since most of these operators are similar to operators, in Java and C++, our focus will not be on the basic functionality of the operators, but instead on some specific ways in which JavaScript operators differ from those in Java.

Before covering the operators themselves, let's review some terminology related to operators. A *binary operator* is one such as `*` that has two operands. Other operators, such as `!` (the logical NOT operator) and `typeof`, have a single operand and are called *unary operators*. A unary operator may be either *prefix*, meaning that it precedes its operand, or *postfix*, meaning that it follows.

JavaScript also has one *ternary operator* that takes three operands: the *conditional operator*, which separates its first and second operands with a question mark (`?`) and its second and third operands with a colon (`:`). Although this operator also appears in Java and C++, you may not have used it before, so I'll describe it briefly here. The conditional operator provides a convenient means to code if-then-else logic within a single statement: the first operand (preceding the question mark) is treated as a Boolean value, and the overall

```
// KeywordStmts.js

// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

  switch (i) {

    // 'case' value can be any expression and data type,
    // not just constant int as in Java.  Automatic
    // type conversion is performed if needed.
    case 1.0 + 2:
      window.alert("i = " + i);
      break;
    default:
      try {
        throw("A JavaScript exception can be anything");
        window.alert("This is not executed.");
      }
      // Do not supply exception data type in 'catch'
      catch (e) {
        window.alert("Caught: " + e);
      }
      break;
  }
}
```

FIGURE 4.7 JavaScript program illustrating several keyword statements.

expression evaluates to either the value of the second operand, if the Boolean value is `true`, or the value of the third operand otherwise. For example,

```
window.alert( "Error " + (debugLevel>2 ? details : "") );
```

adds a string representing the value of the `details` variable to an output message if the relational expression `debugLevel>2` evaluates to `true`. Otherwise, the empty string is appended to the message.

4.7.1 Precedence

Table 4.6 lists the JavaScript operators covered in this chapter, in order of their operator precedence. Recall that, in an unparenthesized expression, an operator of higher *precedence* is performed before one of lower precedence. Table 4.6 lists operators from highest to lowest precedence. As in most languages, parentheses can be used to override these default precedences, with innermost parenthesized subexpressions evaluated before those farther out.

The assignment, conditional, and prefix unary operators are *right associative*, which means that if there is a tie among multiple operators at the same precedence level, the rightmost operator is applied first. For example, an expression such as

```
a *= b += c
```

is equivalent to

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

TABLE 4.6 Precedence (High to Low) for Selected JavaScript Operators

Operator Category	Operators
Object creation	`new`
Postfix unary	`++`, `--`
Prefix unary	`delete`, `typeof`, `++`, `--` , `+`, `-`, `~`, `!`
Multiplicative	`*`, `/`, `%`
Additive	`+`, `-`
Shift	`<<`, `>>`, `>>>`
Relational	`<`, `>`, `<=`, `>=`
(In)equality	`==`, `!=`, `===`, `!==`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional and assignment	`?:`, `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`

The postfix unary operators cannot be applied repeatedly. For example,

```
i++ ++
```

is not a syntactically correct expression. Therefore, associativity does not apply to these operators. The remaining operators—those that are not in the conditional, assignment, or unary categories—are *left associative*, so ties among these operators are broken in favor of the leftmost operator.

Almost all of the operator symbols in Table 4.6 should already be familiar to you, either from your knowledge of Java or C++ or from material earlier in this chapter (two operators that are not found in Java or C++, `===` and `!==`, are covered in some detail later). However, many of these symbols have a somewhat different meaning in JavaScript than they do in Java or C++. We'll give an overview of the primary differences in the remainder of this section.

4.7.2 Type Conversion

One of the key differences between JavaScript and Java/C++ operators has to do with automatic type conversion. Let's begin with the additive and multiplicative operators `+`, `-`, `*`, `/`, and `%`. Each of these operators will convert its operands to Number (according to the rules of Table 4.4) if they are not already of this type. Thus, for example, the following is legal JavaScript code and outputs 3:

```
window.alert("5" - 2);
```

One exception to this conversion rule involves the `+` operator: much as in Java, if either or both of the operands of the `+` operator are of type String, then the non-String operand (if any) will automatically be converted to String and the two String operand values will be concatenated. (Use of the `+` operator in this way was illustrated in the calls to the `alert()` method within the high-low game program of Figure 4.4.)

Similarly, we learned earlier that if a String and a Number value are compared with a relational operator such as `<`, then the String will be converted to a Number before the comparison is performed. More generally, for each of the four relational operators listed in Table 4.6, unless both of its operands are of type String, the operands are converted to Number as needed before the comparison is performed. (Although a bit off the topic of conversions, note that relational expressions involving the special Infinity Number value generally behave as you would expect. For example, everything except Infinity is less than Infinity. The special NaN value, on the other hand, is treated as if it is not less than, equal to, or greater than any other value, including itself.)

If both operands to one of the relational operators or one of the operators `==` and `!=` are of type String, then a lexicographic string comparison is performed. Recall that a *lexicographic comparison* of two strings `s1` and `s2` is performed as follows. String `s1` is lexicographically less than string `s2` if, at the first character position where the strings differ, the character value for `s1` is numerically less than the corresponding character value for `s2`. If the strings do not differ in any character position, then `s1` is lexicographically less than `s2` if the length of `s1` is less than the length of `s2`. If neither of these conditions holds, then the strings are lexicographically equal.

As you're probably aware, lexicographic comparisons of `String` objects in Java cannot be performed with relational or equality operators, but instead are normally performed with calls to methods such as `equals()` and `compareTo()`. For example, if `str1` and `str2` are two Java `String` objects, then to compare these strings lexicographically we would normally use code such as

```
if (str1.equals(str2)) { ... }
```

On the other hand, if `str3` and `str4` are JavaScript variables containing String values, JavaScript's automatic type conversion feature allows us to instead write

```
if (str3 == str4) { ... }
```

The type conversion rules for the `==` and `!=` operators generally follow the pattern we have seen earlier: if the operands are not both of type String, then both are typically converted to Number for comparison purposes. However, there are several exceptions. First, the value of the Undefined type (the type of a declared variable that has not been assigned a value) is considered equal to `null`, the one value of the Null type. Also, if one of the operands is an Object value and this object is an instance of the `Date` built-in object (described in Section 4.12.3), then the Object is converted to String instead of Number. Finally, if one of the operands is an Object value representing a host object rather than a native object, then the type conversion of the object is implementation dependent.

To allow for (in)equality comparisons without type conversion, JavaScript provides the two *strict* operators. The strict equality (`===`) operator evaluates to `true` only if both operands are of the same type (without conversion) and have exactly the same value. The strict inequality (`!==`) operator is of course just the logical NOT of strict equality. For values of type Object, the values are "the same" only if both operands are references to exactly the same object. If you're not quite sure what an object reference is, this concept is covered in detail in Section 4.10.4. For now, it's enough to know that this is how Java defines its `==` operator when applied to two Java objects (and in fact the JavaScript `==` operator also behaves this way when applied to two Object values).

Certain other operators perform natural automatic conversions. For example, each of the unary `+` and `-` operators converts its operand to Number (this is in fact all that the unary `+` operation does). Likewise, the logical AND, logical OR, and logical negation (`!`) operators will all convert their operands to Boolean as needed. (Technically, the logical AND and OR operators behave somewhat differently than this. But if you only use them in the normal way—within the condition expression of statements such as an `if` or `while` statement—then they will effectively behave as just described.)

4.7.3 Bit Operators

A JavaScript programmer cannot access the internal bit representation of a Number value. However, JavaScript does supply the same bit-level operators found in Java: bitwise (ones complement) NOT(`~`); the bitwise AND, XOR, and OR operators; and the shift operators (Table 4.6). The JavaScript operators first convert their operands to 32-bit twos-complement integers by converting them to Number (Table 4.4) if needed, then truncating decimal values

to integers, and finally retaining only the low-order 32 bits for integers that exceed 32 bits. The operators are then applied as if they were operating within a machine with 32-bit registers, which implies, for example, that the result of a left shift operator `<<` will be truncated to 32 bits if necessary. After the operator processing is completed, the 32-bit result is converted back to a Number value. For purposes of this final conversion, the 32-bit result is treated as an unsigned 32-bit integer by the unsigned right shift (`>>>`) operator and as a 32-bit twos-complement integer by the other operators.

As an example of using bit operators in JavaScript, consider the program

```
// ComplementOne.js
window.alert(~1);
```

This program will output -2, because `~1` evaluates to a string of 31 1 bits followed by a 0 bit, which represents -2 when treated as a signed twos-complement 32-bit integer for purposes of conversion to Number. On the other hand,

```
// ComplementOne.js (part 2)
window.alert((~1) >>> 0);
```

outputs 4294967294 ($2^{32} - 2$), which is the value of the same 32-bit string treated as an unsigned integer for purposes of conversion to Number.

4.8 Literals

The most basic elements of any programming language are its *literals*, that is, the strings of characters that directly represent values in the language. We've already noted that `null` is a JavaScript literal representing the (one) value of the Null data type, and that `true` and `false` are the literals representing the two Boolean values. This section provides some syntactic details about JavaScript number and string literals.

The JavaScript number literals are much like those in Java. For example, numbers can be written as integers (whole numbers) or decimals, and in either of these forms scientific notation can be used to indicate a power of 10. For example, `-12.4e12` represents the value -12.4×10^{12}. Java hexadecimal notation, e.g., `0xfa0`, is also a valid form of number literal. Every JavaScript number literal represents a value belonging to the JavaScript Number data type. Since JavaScript has only one type of number, a Java literal that represents a number of a specified type (such as `27.3f`, which represents a floating point value) is not a valid number literal in JavaScript. All Number values are represented using a 64-bit floating point format that provides approximately 16 decimal digits of precision for both integer and decimal values and can represent numbers with magnitudes as large as approximately 10^{308} and as small as approximately 10^{-323}. Literals beyond these limits will be represented as accurately as possible. For example, if a literal represents a value that is too large for the Number data type to represent accurately, then the literal will evaluate to the special Number value Infinity.

A string literal is a quoted string of characters all on a single line, and, as in HTML, the quoting characters may be either a pair of single quotes or a pair of double quotes. Note that this differs from Java, where a character contained in single quotes belongs to

the `char` data type. It is much more efficient to store single characters using `char` than it is to use the Java `String` class. JavaScript, on the other hand, has only one data type for strings—named String—just as it has only one data type for numbers. This is another example of how JavaScript's design favors simplicity over efficiency.

Like Java, escape codes can be embedded in string literals to represent various characters. For example, `\n` represents the line feed (newline) character, and `\"` (`\'`) can be used to represent a double (single) quote within a string literal that is enclosed in double (single) quotes. Of course, because the backslash character is special, it must be escaped within string literals; the escape code `\\` can be used for this purpose. Finally, any 16-bit Unicode character value may be represented by immediately following the `\u` escape code with four hexadecimal digits. For instance, `\u005c` represents the backslash character.

4.9 Functions

The structure of the high-low program of Figure 4.4 is not typical of many JavaScript programs designed to be run within browsers. Instead, such programs often are structured as a set of functions. A *function* in JavaScript is much like a method in Java: it consists of a function name, an argument list, and code that is executed when the function is called. However, unlike Java methods, a function does not need to be associated with an object.

To illustrate, consider Figure 4.8, which is the initial portion of a revision of the high-low program from Figure 4.4 (the remainder of the revised `HighLowWithFunction.js` program is identical to the corresponding portion of the original `HighLow.js`). The code beginning with the keyword `function` and ending with the next closing brace is a *function declaration*. Every JavaScript function declaration follows the same syntax: the keyword `function` followed by an identifier followed by a parenthesized list of zero of more comma-separated identifiers followed by a curly-brace bracketed sequence of statements. The first identifier is the *function name*, the list of identifiers is the function's *formal parameter list*,

```
// HighLowWithFunction.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// This function returns a random number between 1 and
// the argument value 'high'.
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}

// Initialize the computer's number
thinkingOf = oneTo(1000);

...
```

FIGURE 4.8 Initial portion of JavaScript high-low game program that defines and uses a function named `oneTo()` for computing a random number between 1 and a specified value.

and the statements represent the *function body*. In this example, the function name is `oneTo`, there is a single formal parameter named `high`, and the function body consists of a single return-value statement.

Unlike Java and C++, JavaScript does not require (or even allow) the data types of formal parameters or the data type of the return value of a function to be declared. (This shouldn't surprise us, since the data types of variables are also not declared in JavaScript.)

The code

```
oneTo(1000)
```

is a type of JavaScript expression known as a *function call*. Syntactically, the typical function call consists of an identifier followed by a parenthesized list of zero or more comma-separated *arguments*. In the example, the argument is `1000`.

When a scripting engine evaluates a function call, the effect is similar to that of a method call in Java. First, the engine evaluates each of the arguments, producing a value for each. In our example, `1000` is a Number literal and therefore evaluates to a Number value. If the function call had instead been written as `oneTo(999+1)`, the result of the function call would be the same, since the argument value would still be 1000. Next, after evaluating the arguments, the resulting values are associated with the appropriate formal parameters. In our example, the value 1000 will be associated with the formal parameter `high`. Next, the body of the function is executed. Finally, the value of the function call expression is determined as follows. If a return-value statement is executed, then the value of the expression in the return-value statement becomes the value of the function call expression. On the other hand, if a return-void statement is executed or if the final statement in the body has been executed and is not a return statement, then the Undefined value becomes the value of the function call expression. In our example, a return-value statement will be executed, and the value of the expression in this statement is of type Number. So a Number value will be assigned to `thinkingOf`.

As in Java, code within a JavaScript function body can assign values to the function's formal parameters, and such assignments will not change the values of any variables in the function call's argument list, even if the variable and the parameter use the same identifier. For example, the following JavaScript program will produce two alert boxes, the first with the message `hi` and the second with the message `bye`:

```
// ArgChange.js

var message = "bye";

function change(message) {
  message = "hi";
  window.alert(message);
  return;
}

change(message);
window.alert(message);
```

As we'll learn later, JavaScript's handling of object and array arguments is also similar to Java's.

In Java, a compile error is generated if a method call's arguments do not match the number and data types of the formal parameters of the method. In JavaScript, the arguments and parameters do not have data types, so there is no need to match their types. What's more, a function call does not need to have the same number of arguments as there are formal parameters in the function declaration. If too few arguments are supplied, the formal parameters without arguments will be given the Undefined value. If too many arguments are supplied, the excess arguments are ignored.

If a `var` statement appears within a function body, then the variable declared is called a *local variable*, while variables that are declared outside of any function are called *global variables*. The relationship between global and local variables is very similar to the relationship between the instance variables of a Java object—those variables declared outside of any method of the class—and the variables declared within its methods. Global variables can be accessed from any part of a program, while local variables can only be accessed from the function that declares them. Also, global variables exist from the beginning of execution of a program until the program terminates, while a local variable exists only from the time the function declaring the variable is called until the function returns (see Section 4.10.5 for an exception to this rule). If a function is called multiple times, new copies of its local variables are created every time the function is called.

The following program demonstrates JavaScript's behavior if a local and a global variable share the same identifier:

```
// LocalScope.js
var j=6;   // global variable declaration and initialization
function test()
{
  var j;   // local variable declaration
  j=7;     // Which variable(s) does this change?
  return;
}
test();
window.alert(j);
```

When the program is run, it displays an alert box containing 6, even though the `test()` function is called and assigns the value 7 to a variable named `j`. This assignment does not affect the value of the global variable by the same name because of JavaScript's *scope rules*, which dictate that if a variable reference is ambiguous (could refer to either a local or a global variable) then the scripting engine should associate the reference with the local variable. In such a situation, the global variable is said to be *shadowed* by the local variable with the same name.

To override JavaScript's default scope rules and access a shadowed global variable, you can prefix the name of the variable with `window`. Thus, if we changed the assignment statement within `test()` to

```
window.j = 7;
```

then the global rather than the local variable would be affected, and the output of the program would be 7 rather than 6. This works because global variables (as well as declared functions) are stored as properties of the `window` object. You'll want to keep this in mind if you run a debugger on a JavaScript program.

JavaScript supports recursive function calls. If a function calls itself recursively, each recursive execution will have its own local variables and formal parameter values. Thus, a recursive execution of a function does not have direct access to the parameters or local variables of other in-progress executions of the function. So, if you're familiar with the concept of a static local variable in C++ (a variable whose value is shared between calls to a recursive function or between successive nonrecursive calls to the same function), you should be aware that JavaScript has no direct support for this concept.

What if one function calls another function: does it matter which function is declared first in the program? No, it does not matter. This is because the scripting engine operates on a program in two *passes*, or phases. During the first pass, the scripting engine processes all `var` statements and function declarations. This creates global variables and function objects but doesn't actually execute any statements. In the second pass, the engine executes statements. So you can declare functions and variables in any order, because they will all be known to the scripting engine before it begins executing any statements that refer to the functions or variables.

Finally, JavaScript supplies some built-in functions that can be called just as you would call functions you have declared. In particular, the built-in functions `Boolean()`, `String()`, and `Number()` can be called to convert a value from any data type to a Boolean, String, or Number, respectively. Each function takes a single argument of any data type and returns a value of the specified type by applying the appropriate rules from Table 4.2, 4.3, or 4.4.

4.10 Objects

At this point, we've covered JavaScript's basic syntax, and we know a good deal about working with values belonging to the primitive (non-Object) data types. We'll now turn to JavaScript objects. The differences between JavaScript and Java are especially pronounced when it comes to objects, so if you've been skimming this chapter you may want to read this section more carefully.

4.10.1 Object Properties

An *object* in JavaScript is a set of *properties*, each of which consists of a unique *name* along with a *value* belonging to one of JavaScript's six data types. Properties in JavaScript are comparable to Java instance variables (i.e., nonstatic variables declared outside any method).

Like JavaScript variables, object properties themselves do not have data types; only the values assigned to properties have data types. For example, the following sequence of statements that successively assign Boolean, String, and Number values to a single property `prop` of an object `o` (assumed to be declared earlier) is syntactically valid in JavaScript, but would cause a compile error in Java or C++ regardless of how `o` was declared:

```
o.prop = true;
o.prop = "true";
o.prop = 1;
```

An even bigger difference with Java and C++ is that JavaScript programs do not define classes. There are some classlike features. For example, object constructors can be defined (as described later in this section) to create objects and automatically define properties for the objects created. Also, JavaScript uses a so-called *prototype* mechanism to provide a form of inheritance (this is an advanced topic that will not be covered in this chapter). However, unlike Java and C++, in which the class of an object defines its variables and methods, properties and methods can be added to or removed from a JavaScript object after it has been created.

For example, the following is valid in JavaScript:

```
var o1 = new Object();
o1.testing = "This is a test";
delete o1.testing;
```

We'll go into detail on these statements later in this subsection; for now, here's a brief description. The first line creates a variable named `o1` and initializes it with a value of type Object by calling on the built-in constructor `Object()` using a JavaScript `new` expression. The second line then adds a property named `testing` to the `o1` object and assigns a String value to this property. And the third line then deletes this property from `o1`.

Dynamic property creation is yet another example of JavaScript's flexibility, but also of its lack of certain safety features: if you misspell a property name in an assignment statement, you won't get an error message, but instead you'll create a new property. This is something to watch for when you're debugging JavaScript code.

Now we'll cover object creation and dynamic creation and removal of properties in somewhat more detail. First, as shown in the example, expressions beginning with the keyword `new` are used to create objects. Syntactically, a `new` expression begins with the `new` keyword followed by an identifier corresponding to an object constructor followed by a parenthesized list of zero of more arguments. We'll learn how to create user-defined constructors later in this chapter.

A `new` expression causes a new empty object to be created and then calls the specified constructor, supplying it with this new object plus the specified argument values. The constructor can then perform initialization on the object, which might involve creating and initializing properties, adding methods to the object, and adding the object to an inheritance hierarchy from which it can inherit additional properties and methods. In the case of the `Object()` constructor, no properties or methods are added directly to the new object by the constructor, but the object is modified so that it inherits several generic methods including default `toString()` and `valueOf()` methods used when converting the object to String and Number values, respectively. The values produced by these default methods are not particularly useful, but they at least prevent a run-time error in case an attempt is made to apply data type conversion to the object.

We next turn to property creation. As we saw, when a JavaScript statement attempts to assign a value to an object property, and the property does not exist in the object, then a property with the given name is created in the object and assigned the specified value. This happens even if the object has inherited a property with the same name, since an inherited property or method actually resides in a different object.

Finally, as shown, the keyword `delete` can be used to remove a property from an object. Syntactically, a `delete` expression begins with the `delete` unary operator followed by a reference to an object property, such as `o.testing`. You should not attempt to delete a nonexistent property (technically, this should be legal, but some browsers will throw an exception).

JavaScript provides a handy shortcut called an *object initializer* for creating an empty object (as if by a call to `new Object()`), creating properties on this object, and assigning values to these properties. For example, the statement

```
var o2 = { p1:5+9, p2:null, testing:"This is a test" };
```

creates an object with three properties `p1`, `p2`, and `testing` and assigns these properties the values `14`, `null`, and `This is a test`, respectively. A reference to this object is then assigned to the variable `o2`.

4.10.2 Enumerating Properties

If properties can be dynamically created and destroyed, how can your program know which properties an object has at any given time? JavaScript provides a special for-in statement that can be used to iterate through all of the property names of an object. The following JavaScript code illustrates the use of this statement:

```
var hash = new Object();
hash.kim = "85";
hash.sam = "92";
hash.lynn = "78";
for (var aName in hash) {
  window.alert(aName + " is a property of hash.");
}
```

Executing this program will produce three alert boxes, each with one of the names `kim`, `sam`, or `lynn`. However, the order in which they appear is implementation dependent.

The syntax of a for-in statement is as shown: It begins with the reserved word `for` followed by a parenthesized portion followed by a statement (often a brace-enclosed block statement). The parenthesized portion begins with a variable identifier, optionally preceded by `var` if the variable was not previously declared. Then comes the reserved word `in` followed by an expression that evaluates to an object.

4.10.3 Array Notation

The code in the preceding example prints the property names. What if we want to print the values of these properties? The notation `hash.aName` represents the property named `aName`, which is not what we want. What we want is a notation that means "Evaluate the expression

aName and convert it to a String if necessary. Then retrieve the property that has this string value as its name from the hash object."

As you might guess, JavaScript provides just such a notation: hash[aName] has exactly this meaning. So, for example, if we change the window.alert() call in the code to

```
window.alert(aName + " scored " + hash[aName]);
```

then the alert boxes output by the program will contain both the name and the value for each of the three properties in hash.

In short, JavaScript provides two different notations for accessing object properties. The first notation is the familiar dot notation, for example hash.kim. The second notation uses an array reference syntax in which the index is viewed as a String value, for example hash["kim"]. As we have seen, the array notation has the advantage that any expression, not just a string literal, can be used to represent the property name. So the array notation is more general, and in fact scripting engines view dot notation such as hash.kim as shorthand for the equivalent array notation hash["kim"].

Therefore, an object in JavaScript can also be viewed as a sort of array in which the elements are indexed by strings. Such an array is sometimes called an *associative array*. This is a powerful form of data structure, similar to Java's java.util.Hashtable class but built directly into the JavaScript language rather than provided by an API. In addition to the associative array feature of all JavaScript objects, JavaScript objects constructed using the Array() constructor (covered in Section 4.11) also have additional array-specific properties and methods.

4.10.4 Object References

Recall that if you execute the Java code

```
StringBuffer s1 = new StringBuffer("Hello");
StringBuffer s2 = s1;
```

then a single StringBuffer is created and both s1 and s2 will be references to it. This is because what is stored in a Java variable that represents an object is a reference (pointer) to the object and not the object itself. Thus, the second statement in the code copies the reference from s1 to s2 rather than making a copy of the entire object. Therefore, if the code is followed by

```
s2.append(" World!");
System.out.println(s1);
```

then even though it might appear that s2 is modified and s1 is not, the output will be Hello World!.

In the same way, JavaScript values of type Object are also references to objects, not actual objects themselves. So the following JavaScript code will also output `Hello World!`:

```
// ObjRef.js
var o1 = new Object();
o1.data = "Hello";
var o2 = o1;
o2.data += " World!";
window.alert(o1.data);
```

Similarly, when an Object value is used as an argument to a function or method, the object reference is passed and not a copy of the object itself. To illustrate this, consider the JavaScript program of Figure 4.9. The program creates two objects `o1` and `o2` and passes them as arguments to the function `objArgs()`. Thus, immediately after the function is called, `param1` is a copy of the object reference contained in `o1`, and `param2` a copy of `o2` (part (a) of Figure 4.10). The first statement of the function changes the `data` property of the object referenced by `param1` and `o1` (Figure 4.10(b)). The function then changes `param2` to be a copy of the object reference contained in `param1`. Note that this has no impact on the variable `o2` or on the object referenced by this variable (Figure 4.10(c)). The output of the program will therefore be the two alert boxes shown in Figure 4.11.

```
// ObjArg.js

function objArgs(param1, param2) {
  // Change the data in param1 and its argument
  param1.data = "changed";
  // Change the object referenced by param2, but not its argument
  param2 = param1;

  window.alert("param1 is " + param1.data + "\n" +
               "param2 is " + param2.data);
  return;
}

// Create two different objects with identical data
var o1 = new Object();
o1.data = "original";
var o2 = new Object();
o2.data = "original";

// Call the function on these objects and display the results
objArgs(o1, o2);
window.alert("o1 is " + o1.data + "\n" +
             "o2 is " + o2.data);
```

FIGURE 4.9 JavaScript program illustrating the use of objects as arguments.

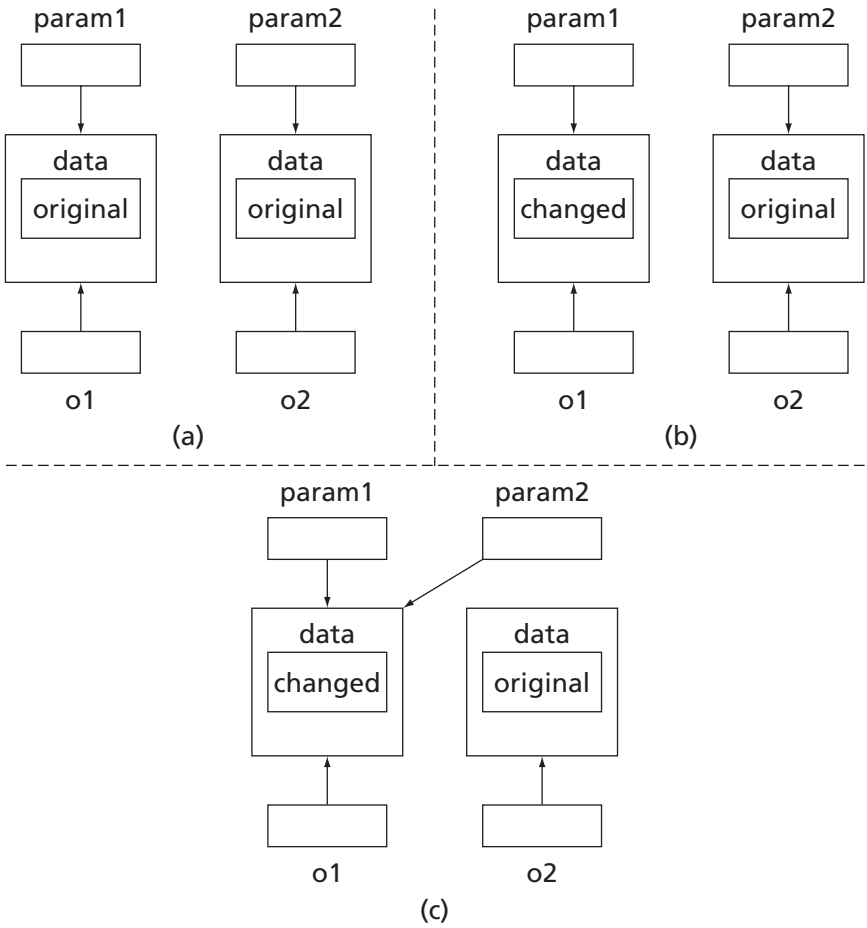

FIGURE 4.10 State of variables and parameters of the `ObjArg.js` program (a) immediately before the first statement of the function `objArgs()` is executed; (b) immediately after this statement is executed; and (c) immediately after the second statement of the function is executed.

4.10.5 Methods

If you read the description of the `typeof` operator (Table 4.1) carefully, you noticed that a value of type Object can represent a function. In fact, internally, JavaScript represents every function as an Object value. That is, when JavaScript processes a function declaration such as the one for `objArgs()`, it creates a specialized object that represents the function. A variable that has the same name as the function is also created, and a reference to the function object is assigned to this variable. Thus, when the program of Figure 4.9 is executed, it automatically creates a variable named `objArgs` that is a reference to an Object value representing the function `objArgs()`.

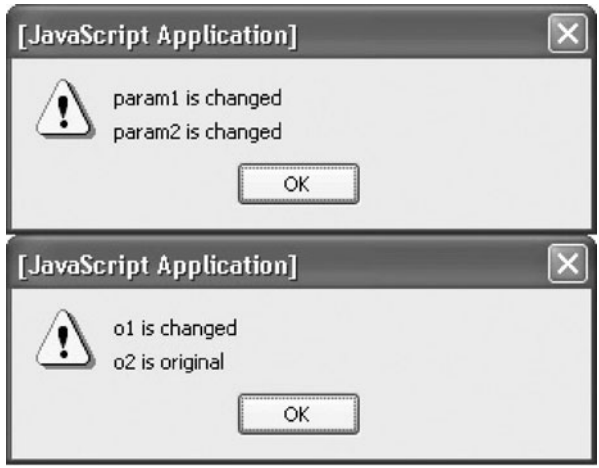

FIGURE 4.11 The two output alert windows produced by running `ObjArg.js`.

A *method* in JavaScript is simply a function that has been assigned as the value of a property of an object. For example, consider the JavaScript code of Figure 4.12, which creates a `node` object that could be suitable for use in a binary search tree data structure (see Section 4.10.7 for more on binary search trees). The code begins by declaring a function named `leaf()` that returns `true` exactly when a node represents a leaf (i.e., when both its `left` and `right` properties have the value `null`). As just noted, this function declaration also creates a variable named `leaf` that contains a reference to the object representing the function. The program subsequently contains the statement

```
node.isLeaf = leaf;
```

This creates a property named `isLeaf` and assigns to it a reference to the `leaf` function object. In other words, this creates a method on the `node` object. Although the method name can be the same as the function name, this is not necessary. The `isLeaf()` method is called later in the program with expressions such as

```
node1.isLeaf()
```

If you run the program, you'll see that the first node is not a leaf and the second one is a leaf.

As illustrated in the body of `leaf()`, the JavaScript keyword `this` is used to allow a method to access other properties of the object to which the method is assigned. That is, if `this` is used in an expression in the body of a function that is a method on some object, then when the method is called, `this` will evaluate to a reference to the object. So, when the `isLeaf()` method is called on `node1` causing `leaf()` to be called, the expression `this` within `leaf()` evaluates to a reference to `node1`.

```
// NodeWithMethod.js

// leaf() returns true iff this node is a leaf node.
// It is designed to be called as a method, not directly
// as a function.
function leaf() {
  return this.left == null && this.right == null;
}

// makeBTNode(value) creates a binary tree node and
// initializes its value to the given argument.
// It also adds an isLeaf() method to the node.
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}

// Create and initialize two node objects, making the second
// a child of the first.
var node1 = makeBTNode(3);
var node2 = makeBTNode(7);
node1.right = node2;

// Output the value of isLeaf() on each node
window.alert("node1 is a leaf: " + node1.isLeaf());
window.alert("node2 is a leaf: " + node2.isLeaf());
```

FIGURE 4.12 Program that creates a node object containing a `clear()` method.

Notice that the function `leaf()` is declared as a function but is intended to be used only as a method, not a function. To make this intention clear (which could be helpful to someone else maintaining this program, for example), it would be better if we could avoid declaring the function at all and instead simply create a method directly. In fact, we can do this by removing the declaration of `leaf()` and replacing the assignment to `node.isLeaf` as follows:

```
// NodeWithMethod2.js
// Remove leaf() declaration found in NodeWithMethod.js
...
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
```

When a `function` keyword appears where an expression is expected, such as on the right-hand side of an assignment statement, it marks the beginning of a *function expression*. Such an expression evaluates to an Object value representing a function, just as a function declaration does. However, a key difference is that the scripting engine does not create

a variable for the object created by a function expression. Therefore, if we create the `isLeaf()` method using a function expression, then it is no longer possible to to call `leaf()` directly.

I should explain one syntactic aspect of this code: Notice that there is a semicolon following the final closing brace of the `leaf()` function expression. This semicolon marks the end of the statement that assigns a value to the `isLeaf` property. While JavaScript does not always require this (since the syntax rules often allow semicolons to be omitted), it's good practice to include it.

One final note: If a function declares a local variable and also evaluates a function expression, and if the function expression contains a reference to the local variable, then the local variable will continue to exist after the function declaring the variable returns.

4.10.6 Constructors

The function `makeBTNode()` in the previous example acts as a kind of constructor for objects representing nodes in a binary search tree. While this code is fairly simple, it can be made even simpler by using JavaScript's constructor mechanism.

Figure 4.13 illustrates the use of a JavaScript constructor. It will help you to read this code if you understand that every function can be called not only as a function and as

```
// BTNode.js

// BTNode(value) is a constructor for a binary tree node.
// It initializes its value to the given argument.
// It also adds an isLeaf() method to the node.
function BTNode(value) {
  // Notice that we no longer need to create an Object
  // and that we use "this" to reference the object
  // initialized.
  this.left = this.right = null;
  this.value = value;
  this.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  // Notice that we no longer return a value.
}
// Create and initialize two node objects, making the second
// a child of the first.
// Notice the use of "new" to call a function as a constructor.
var node1 = new BTNode(3);
var node2 = new BTNode(7);
node1.right = node2;

// Output the value of isLeaf() on each node
window.alert("node1 is a leaf: " + node1.isLeaf());
window.alert("node2 is a leaf: " + node2.isLeaf());
```

FIGURE 4.13 Program that defines and uses an object constructor.

a method, but also as a constructor. That is, every JavaScript function you declare is also automatically a constructor. Therefore, the `BTNode()` function declared at the beginning of Figure 4.13 can be used as a constructor, even though its declaration is no different than any other function declaration.

You tell the scripting engine that you want to call a function as a constructor by prefixing the call with the `new` keyword. Thus, in the statements initializing `node1` and `node2`, we are calling `BTNode()` as a constructor rather than as a standard function. When a function is called as a constructor, the scripting engine automatically creates an empty object and associates the `this` keyword with this object within the body of the function. This is why we do not need to explicitly create an object in the `BTNode()` function as we did in the earlier `makeBTNode()` function. This is also why we use the keyword `this` in the assignment statements in the body of `BTNode()`.

Finally, although the `BTNode()` function does not return a value, each `new` expression used to call this function as a constructor will return as its value the object that was automatically created by the scripting engine and initialized by the code in the body of `BTNode()`. So, as before, the output of this program will show that the first node is not a leaf and the second is.

Thus, although JavaScript does not have a class concept exactly like that found in Java and C++, we see that the constructor mechanism can be used to provide somewhat similar functionality. In fact, when we construct an object in JavaScript using `new`, the constructed object is known as an *instance* of the function (object) used as the constructor. In the preceding example, we could call `node1` and `node2` instances of `BTNode`. The JavaScript `instanceof` operator can be used to test whether one object is an instance of another object. For example, adding the code

```
// Test that node1 is an instance of BTNode
window.alert("node1 is instance of BTNode: " +
             (node1 instanceof BTNode));
```

to the end of the `BTNode.js` file produces the output `node1 is instance of BTNode: true`.

4.10.7 Example: Binary Tree

We'll end this section on JavaScript objects by writing code for creating and using a simple binary search tree. Recall that in a binary tree, every node has a left and a right child, although either or both children may be empty (null). In a binary search tree, each node N has the property that all values (if any) stored in nodes in the left subtree of N are smaller than N's value, and all values stored in its right subtree are larger than N's value. Therefore, we can search to see whether or not a value is stored in such a tree by first comparing the search value with the root node's value. If the values match, we're done. Otherwise, if the search value is smaller than the root value, we continue our search in the left subtree, because we know that the search value must be there if it is in the tree at all. Similarly, we need only search the right subtree if the search value is larger than the root value.

Figures 4.14 and 4.15 show JavaScript code for a constructor function `SearchTree()` that creates a binary search tree object having methods `insert()` and `search()`. The

```
// SearchTree.js (part 1)

// Constructor for BTNode objects
function BTNode(value) {
...
}

// Constructor for SearchTree objects
function SearchTree() {

  /**** Variables ****/
  this.root = null;

  /**** Methods ****/

  // locate(value) returns a reference to the node
  // containing the value if the given value is in the search tree,
  // returns null if the search tree is empty,
  // and otherwise returns a reference to the node that
  // should be this value's parent.
  // This method is only intended to be used by other
  // methods, not called directly.
  this.locate =
    function locate(value) {
      var curr = this.root;  // Node currently being visited
      var parent = null;     // Parent of current node
      // Search for value, remembering parent nodes
      while (curr != null && curr.value != value) {
        parent = curr;
        if (value < parent.value) {
          curr = parent.left;
        }
        else {
          curr = parent.right;
        }
      }

      // If curr is null, we did not locate the value.
      // Return parent (which is null if tree is empty) instead.
      if (curr == null) {
        curr = parent;
      }
      return curr;
    };
```

FIGURE 4.14 Initial portion of program for creating a binary search tree.

`insert()` method first searches for the value to be inserted and, if the value is not found, creates a `BTNode` (using the constructor of Figure 4.13). It then inserts this node into an appropriate location so that the tree retains the binary search property described in the preceding paragraph. Both methods rely on another method named `locate()` that performs the actual search for a value in the tree and returns a value that either indicates that the search

```
// SearchTree.js (part 2)

  // insert(value) adds the given value to the search
  // tree if it is not already present.
  this.insert =
    function insert(value) {

      // Search for location of value or its parent
      var location = this.locate(value);

      // If value is not present (location is non-null
      // and location.value is not same as given value),
      // then create a node containing the value
      // and insert it at the appropriate location.
      var node = new BTNode(value);
      if (location == null) { // Empty tree
        this.root = node;
      }
      else if (location.value != value) { // location is parent
        if (value < location.value) {
          location.left = node;
        }
        else {
          location.right = node;
        }
      }
    };

  // search(value) returns true iff the given value is
  // present in the tree
  this.search =
    function search(value) {

      // Search for location of value or its parent
      var location = this.locate(value);

      // Found value iff location is not null and value
      // of location is the given value
      return location != null && location.value == value;
    };
}
```

FIGURE 4.15 Remainder of constructor for binary search tree program.

succeeded or provides information that can be used to insert the value into the appropriate location within the tree. Figure 4.16 then shows code that constructs a tree, inserts values into the tree, and searches the tree. When run, this code will indicate that the value 12 is contained in the tree but that 6 is not.

This program illustrates several points. First, notice that a method can declare local variables (such as `curr` in `locate()`) just like any other function, and that these variables are referenced within a method just as they would be within a normal function.

```
// SearchTree.js (part 3)

// Create a search tree and insert values
var tree = new SearchTree();
tree.insert(7);
tree.insert(3);
tree.insert(12);
tree.insert(12); // Try inserting a second time

// Search the tree
window.alert("Search for 12 produces " + tree.search(12));
window.alert("Search for 6 produces " + tree.search(6));
```

FIGURE 4.16 Code for using binary search tree constructor and calling methods on constructed tree object.

Second, note that if one method (such as `insert()`) wants to call another method (such as `locate()`) of the same object, the call must be prefixed with `this`. Otherwise, the scripting engine would interpret a call without the leading `this`—such as `locate(value)`—as a call to a regular function and not to a method. Third, notice that although `locate()` is intended to be called only by other methods within a `SearchTree` object, there is no `private` declaration in JavaScript to enforce such an intention. Since most JavaScript programs are small, this simplification compared with Java and C++ is a reasonable trade-off.

Although the test program in Figure 4.16 uses integer values for nodes, notice that `SearchTree` and `BTNode` objects can accommodate values of any type. In fact, values of different types can be stored in a single `SearchTree`, because JavaScript's comparison operators (`!=`, `<`, etc.) can be used to compare arbitrary data types, as explained in Section 4.7. However, note that because the code uses the nonstrict equality operators `!=` and `==`, values belonging to two different data types might map to a single node. For example, if the statement

```
window.alert('Search for "3" produces ' + tree.search("3"));
```

is added to the end of the `SearchTree.js` program, this search will evaluate to `true`, even though no node in the tree contains the String value `3`. We could avoid this effect by using the strict equality operators `!==` and `===` in place of the nonstrict operators in the comparisons involving the `value` property of `BTNode`'s.

4.11 Arrays

We learned earlier that all object properties can be accessed using an arraylike notation, and that objects can therefore be viewed as associative arrays. In addition, JavaScript supplies a native function object named `Array` that can be used to construct objects that have special array characteristics and that inherit a number of array-oriented methods. In this section, we'll learn more about creating and using `Array` instances, which we'll refer to simply as *arrays*.

4.11.1 Creating an Array

One way to create an array is to use the `Array` constructor directly in a call with no arguments:

```
var ary1 = new Array();
```

Alternatively, an array can be constructed and initialized with values by supplying two or more arguments to the `Array` constructor:

```
var ary2 = new Array(4, true, "OK");
```

After this statement is executed, `ary2[0]` will have the Number value `4`, `ary2[1]` the Boolean value `true`, and `ary2[2]` the String value `OK`. Keep in mind that array notation is just a way of specifying the name of a property in JavaScript. So an expression such as `ary2[0]` evaluates to the value of the property with the name `0` (a String consisting of a single numeric character) belonging to the object referenced by the variable `ary2`. On the other hand, attempting to execute the expression `ary2.0` would generate a syntax error in JavaScript. This is because an identifier must be used for the property name in dot notation, and the syntax rules for identifiers require that they must not begin with a numeric character. There is, however, no such restriction on the actual names that can be used for properties: any string can be used.

I will refer to properties of an array object that have names that are string representations of natural numbers as the *elements* of the array. So `ary2` in the second example has three elements immediately after the `Array` constructor is executed. An array object can also have other nonelement properties. In fact, every array object is automatically given a special property named `length`. When an array is constructed with a no-argument constructor (as in the case of `ary1` in the first example), the value of `length` will be 0. When constructed with two or more arguments, `length` is set to the number of arguments. Thus, after the statement creating the variable `ary2`, `ary2.length` will have the value 3. We'll have more to say about this property in the next section.

Another way to create and initialize an array that produces identical results to those of the previous example is

```
var ary3 = [4, true, "OK"];
```

An expression such as the one on the right side of the assignment in this example is called an *array initializer*. This is essentially a shorthand that implicitly calls the `Array` constructor for us.

The elements of a JavaScript array can be any JavaScript values, including other instances of `Array`. This feature can be used to create multidimensional arrays. For example, one way to create a two-dimensional array is

```
// array2d.js
var ttt = [ [ "X", "O", "O" ],
            [ "O", "X", "O" ],
            [ "O", "X", "X" ] ];
```

In this example, the outer one-dimensional array (named `ttt`) contains three elements, each itself a one-dimensional array. The notation `ttt[1]` will evaluate to a reference to the second of these internal arrays (the `O X O` array). Since this is an array object itself, the notation `ttt[1][2]` will evaluate to the value of the third element of this array, `O`. Higher-dimensional arrays can be created and accessed similarly.

4.11.2 Dynamically Changing Array Length

Just as we can dynamically add properties to any JavaScript object, we can add properties to an array. In particular, in Section 4.11.1, we could follow the code creating `ary2` (which had three elements) with the statement

```
ary2[3] = -12.6;
```

This will create a new property named (as a String) `3` with the value `-12.6`. In addition, the `length` property of `ary2` will be changed to 4. Thus, unlike arrays in Java, the effective size of JavaScript arrays can change dynamically. In this way, JavaScript arrays are more like instances of the `java.util.Vector` class than they are like Java arrays.

Elements can also be removed from a JavaScript array by decreasing the value of the `length` property. For example, if we next executed the statement

```
ary2.length = 2;
```

then the properties `2` and `3` would automatically be removed from `ary2`.

While we're on the topic of the `length` property, let me mention a special case that might cause you trouble if you're not aware of it. If the `Array` constructor is called with a single Number argument, then this argument value is assigned to the `length` property but no array elements are actually created. As an example, the expression

```
new Array(200)
```

creates an array having `length` 200 but does not actually create or initialize the properties with names `0` through `199`. Similarly, if you increase the `length` value of an array object, this does not automatically add any elements (properties with numeric names) to the array. Thus, your code should not assume that `length` always represents the actual number of elements in an array.

4.11.3 Array Methods

Every array object automatically inherits a number of useful methods. Many of these methods are described briefly in Table 4.7, and some additional details are given in this subsection. Full algorithmic details of every method are provided in Section 15.4.4 of the ECMAScript specification [ECMA-262].

The argument to the `sort()` method should be an object representing a function. This function in turn should take two arguments representing array elements and return a Number value. A negative return value indicates that the element corresponding to the first argument should come before the element corresponding to the second in the sorted array,

TABLE 4.7 Methods Inherited by Array Objects. Unless Otherwise Specified, Methods Return a Reference to the Array on Which They are Called.

Method	Description
`toString()`	Return a String value representing this array as a comma-separated list.
`sort(Object)`	Modify this array by sorting it, treating the Object argument as a function that specifies sort order (see text).
`splice(Number, 0, any type)`	Modify this array by adding the third argument as an element at the index given by the first argument, shifting elements up one index to make room for the new element.
`splice(Number, Number)`	Modify this array by removing a number of elements specified by the second argument (a positive integer), starting with the index specified by the first element, and shifting elements down to take the place of those elements removed. Returns an array of the elements removed.
`push(any type)`	Modify this array by appending an element having the given argument value. Returns `length` value for modified array.
`pop()`	Modify this array by removing its last element (the element at index `length`−1). Returns the value of the element removed.
`shift()`	Modify this array by removing its first element (the element at index 0) and shifting all remaining elements down one index. Returns the value of the element removed.

a positive value indicates that the second should come before the first, and 0 indicates that the elements can be in either order (for sorting purposes, they are equivalent). For example, in the program of Figure 4.17, the return value will be positive if the first argument is greater than the second and negative if the first is less than the second. This will result in an array of Number values being sorted into ascending order, as shown. Also notice that the argument to `sort()` can be a function expression rather than the name of a function declared elsewhere.

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

FIGURE 4.17 Program illustrating use of several of the methods inherited by array objects.

Figure 4.17 also illustrates two versions of the `splice()` method, which is used both for inserting an element into an array (and automatically making room for the new element by shifting other elements) and for removing one or more elements from an array (and automatically shifting other elements to take the place of those removed). Notice that the version of `splice()` that removes elements also returns an array containing the elements removed.

The `push()`, `pop()`, and `shift()` methods make it easy to implement basic stack and queue data structures using arrays (if you're not familiar with these data structures, you may want to either read about them in any textbook on data structures or skip the remainder of this section). In the case of a stack, the `length` property acts as the stack pointer, with an empty stack indicated by a 0 value. The following code illustrates usage of these methods to implement a stack:

```
// stack.js
var stack = new Array();

stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3);  // displays "Hi!"
```

You can similarly implement a basic queue structure using `push()` to add elements to the end of the queue and `shift()` to remove elements from the front of the queue. In this case, `length` will point to the back of the queue, and the front of the queue will be at index 0. Other array methods can be used for additional functionality. For example, `splice()` could be used to insert an element at a location other than the end of the queue, which might be useful as part of an implementation of priority queues.

4.12 Built-in Objects

In this section, we will briefly overview several of JavaScript's built-in objects. Chapter 15 of the ECMAScript reference [ECMA-262] describes all built-in objects and should be consulted for details.

4.12.1 The Global Object (`window`)

In every JavaScript host environment, some object is designated as the *global object*. In common browsers, this object is named `window`. It is called the global object because, as mentioned earlier, all global variables declared by your program are actually stored as properties of this object. What's more, other built-in objects, such as `Object`, `Array`, and other objects described later, are also stored as properties of the global object. For example, the `Object` object is stored in `window.Object`.

Furthermore, all host objects are stored (directly or indirectly) as properties of the global object. I've emphasized this fact in the JavaScript code in this chapter by prefixing `window.` before the calls to the `alert()` and `prompt()` host methods. Usually, though, this isn't necessary, because if your code calls a function or refers to a variable and JavaScript can't locate the function or variable among your function declarations and local variables (and formal parameters, if the code is in a function body), then it will look for a property by the given name in the `window` object.

In addition to properties representing built-in and host objects, the global object contains several other useful properties. For instance, the global object property `Infinity` contains the Number value representing Infinity, which is sometimes useful (as an initial value in code for finding the minimum value in an array, for example). See the ECMAScript specification [ECMA-262] for additional global object properties.

4.12.2 `String`, `Number`, and `Boolean`

We're previously learned that JavaScript has built-in functions `String()`, `Number()`, and `Boolean()` that can be used to convert values to each of these data types. If any of these functions is called as a constructor, it will not only convert its argument value to the appropriate type if necessary, but will also wrap the converted value in an object that is an instance of the constructor object. For example,

```
// WrappedNumber.js
var wrappedNumber = new Number(5.625);
```

creates an object that is an instance of `Number`. This object will inherit certain methods associated with the `Number` object. One of these methods will be a version of `valueOf()` that returns the wrapped value (a value of type Number in the case of an instance of `Number`). So, for example, executing

```
// WrappedNumber.js (part 2)
window.alert(typeof wrappedNumber.valueOf());
```

will indicate that `wrappedNumber.valueOf()` is of type Number.

In addition to `valueOf()`, `Number` instances inherit several methods that can be used to format a numeric value for output purposes. For example,

```
// WrappedNumber.js (part 3)
window.alert(wrappedNumber.toFixed(2));
```

will output `5.63`, which is the value of the original number rounded to two decimal places. The `toExponential(Number)` method is similar:

```
// WrappedNumber.js (part 4)
window.alert(wrappedNumber.toExponential(2));
```

TABLE 4.8 Some of the Methods Inherited by `String` Instances

Method	Description
`charAt(Number)`	Return string consisting of single character at position (0-based) Number within this string.
`concat(String)`	Return concatenation of this string to String argument.
`indexOf(String, Number)`	Return location of leftmost occurrence of String within this string at or after character Number, or −1 if no occurrence exists.
`replace(String, String)`	Return string obtained by replacing first occurrence of first String in this string with second String.
`slice(Number, Number)`	Return substring of this string starting at location given by first Number and ending one character before location given by second Number.
`toLowerCase()`	Return this string with each character having a Unicode Standard lowercase equivalent replaced by that character.
`toUpperCase()`	Return this string with each character having a Unicode Standard uppercase equivalent replaced by that character.

will output `5.63e+0`. Finally,

```
// WrappedNumber.js (part 5)
window.alert(wrappedNumber.toString(2));
```

outputs `101.101`, which is 5.625 represented in binary (base 2). This method can be used to output a number in any standard number base, including base 8 (octal) and base 16 (hexadecimal).

The `Number` object itself also provides properties `Number.MIN_VALUE` and `Number.MAX_VALUE`, which contain values that are the smallest and largest (in absolute value) numbers that can be represented in JavaScript.

Every `String` instance has a property named `length`, which is a Number value representing the number of characters in the internal string value. In addition, instances of `String` inherit a number of methods, including those shown in Table 4.8. Note that none of these methods change the value of the `String` instance on which they are called. Instead, they return a value that uses the instance's underlying String value as an input.

Finally, JavaScript's automatic type conversion allows you to apply `String` and `Number` inherited methods directly to primitive values of type String and Number, respectively. For example, the following is syntactically legal JavaScript that will output `Str`:

```
// slicestring.js
var primitiveString = "a String value";
window.alert(primitiveString.slice(2,5));
```

When the scripting engine encounters this code, it automatically wraps the String value contained in `primitiveString` within a `String` object instance, since the String value is being used where an Object value is expected (preceding a dot). Thus, the code given is a shorthand for

```
// slicestring.js (part 2)
window.alert((new String(primitiveString)).slice(2,5));
```

In fact, you can even use a string literal as the "object" on which a `String` method is called:

```
// slicestring.js (part 3)
window.alert("a String value".slice(2,5));
```

4.12.3 `Date`

The expression

```
new Date()
```

returns an instance of the built-in `Date` object representing the current date and time on the host machine (which, when JavaScript is run within a browser, is the machine running the browser). `Date` instances can be used for a variety of purposes. One common use is to generate a string representing the current date and time in the user's time zone; this string can then be displayed in the browser (using techniques discussed in Chapter 5) to make a web page appear fresh. The inherited `toLocaleString()` method, when called on a `Date` instance, returns a date/time string formatted according to the conventions used in the user's location (as determined by browser settings). Alternatively, `toLocaleDateString()` and `toLocaleTimeString()` can be used to obtain only the date or time, respectively. Other methods allow you to extract the year, month, day within the month or week, hour, and so on. See Section 15.9.5 of the ECMAScript reference [ECMA-262] for a complete list of inherited methods.

Another use of `Date` instances is to determine how much time has elapsed between two events. For example, consider the following JavaScript code:

```
// ElapsedTime.js

var startTime = new Date();

// Perform some processing
...

var endTime = new Date();
window.alert("Processing required " +
             (endTime - startTime)/1000 +
             " seconds.");
```

Recall that when JavaScript attempts to perform arithmetic involving Object values (such as `Date` instances), it first casts the objects to Number values by calling `valueOf()` on each object. The `valueOf()` method inherited by a `Date` instance returns an integer representing the number of milliseconds that have elapsed between the date and time represented by the instance and a certain fixed date and time (midnight universal (UTC) time on January 1, 1970). So taking the difference of two `Date` instances gives the number of milliseconds

that have elapsed from the date and time represented by the first instance to the date and time of the second. Dividing this difference by 1000 gives the elapsed time in seconds.

Similarly, if `date1` and `date2` are `Date` instances, then the following code will execute the block statement controlled by the if statement only if the date and time represented by `date1` is prior to that represented by `date2`:

```
if (date1 < date2) { ... }
```

You should understand why this code works in view of the preceding discussion. It's worth noting that to compare dates in Java requires a method call. Thus, this code once again illustrates how intuitive JavaScript code can be due to its use of automatic type conversion.

4.12.4 `Math`

The `Math` object is like the global object in that you do not construct instances from it, but instead you call methods directly on it. For instance, to perform a square root operation, you can use an expression such as

```
Math.sqrt(15.3)
```

All of the methods of this object are listed in Table 4.9. In addition to these methods, `Math` has several properties with values of type Number, including `E` and `PI`, which represent

TABLE 4.9 Methods of the `Math` Built-in Object

Method	Return Value
`abs(Number)`	Absolute value of Number
`acos(Number)`	Arc cosine of Number (treated as radians)
`asin(Number)`	Arcsine of Number
`atan(Number)`	Arctangent of Number (range `-Math.PI/2` to `Math.PI/2`)
`atan2(Number, Number)`	Arctangent of first Number divided by second (range `-Math.PI` to `Math.PI`)
`ceil(Number)`	Smallest integer no greater than Number
`cos(Number)`	Cosine of Number (in radians)
`exp(Number)`	`Math.E` raised to power Number
`floor(Number)`	Largest integer no less than Number
`log(Number)`	Natural logarithm of Number
`max(Number, Number, ...)`	Maximum of given values
`min(Number, Number, ...)`	Minimum of given values
`pow(Number, Number)`	First Number raised to power of second Number
`random()`	Pseudo-random floating point number in range 0 to 1
`round(Number)`	Nearest integer value to Number
`sin(Number)`	Sine of Number
`sqrt(Number)`	Square root of Number
`tan(Number)`	Tangent of Number

the base of the natural logarithm and the ratio of the circumference to the diameter of a circle, respectively.

4.12.5 RegExp

JavaScript provides regular expression capabilities through instances of the built-in `RegExp` object. A *regular expression* is a certain way of representing a set of strings. Regular expressions are frequently used to test that a string entered in an HTML form has a certain format, or, in the terminology of regular expressions, belongs to the set of strings that have the correct format.

As a simple example, the set of strings that consist of exactly three digits—which might represent, for example, the set of valid area codes in a phone number—can be represented by the JavaScript regular expression

```
\d\d\d
```

In a JavaScript regular expression, the string `\d` stands for the set of digit characters `0` through `9`. Concatenating this string with itself three times as shown represents the set of all possible strings of three consecutive digits.

We could use this regular expression as follows to test whether or not a String value contained in a variable named `areaCode` consists of exactly three digits:

```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

(We'll soon see why this regular expression doesn't look exactly like the preceding one.) The `test()` method of a `RegExp` instance returns `true` if its String argument is a member of the set of strings represented by the instance's regular expression, and returns `false` otherwise. In this example, we want to output a message if `areaCode` is not in the set of valid area codes, so we complement the value returned by `test()` in the condition of the `if` statement.

There are several differences between the string argument supplied to the `RegExp()` constructor and the regular expression given earlier. First, there are more backslash characters (\) in the string. This is because, within a String literal, the backslash character is normally used to begin an escape code used to represent some Unicode character (Section 4.8). But within a regular expression the backslash is used to represent, not a Unicode character, but a special subexpression within the larger regular expression. Therefore, when a backslash appears in a String literal as part of a regular expression, the backslash must be escaped by preceding it with another backslash.

The other difference is that the string argument passed to `RegExp()` begins with a caret (`^`) and ends with a dollar sign (`$`). These are necessary because of the way JavaScript interprets the argument to `RegExp()`. Specifically, although the regular expression `\d\d\d` represents strings *consisting of* three consecutive digits, when it is passed to `RegExp()` it is treated as though it represents all strings *containing* three consecutive digits. The caret and dollar sign characters can be used to override this `RegExp()` interpretation.

In particular, in a JavaScript regular expression, the caret and dollar sign characters represent the beginning and end of a string, respectively. Thus, the argument represents any string in which the beginning of the string is immediately followed by three consecutive digits, the last of which is immediately followed by the end of the string. These characters can be used for other purposes as well. For example, if we removed the dollar sign from the argument to `RegExp` we obtain

```
var acTest = new RegExp("^\\d\\d\\d");
```

which represents the set of all strings that begin with three digits.

JavaScript provides an alternate syntax for creating a `RegExp` instance that avoids the need to duplicate backslashes. For example, the preceding declaration of `acTest` can be rewritten as

```
var acTest = /^\d\d\d/;
```

The expression on the right-hand side of the assignment is known as a *regular expression literal*. The scripting engine conceptually converts such an expression to a call to the `RegExp()` constructor, automatically escaping any backslash characters contained in the regular expression literal. Notice that it is still necessary to use the caret and/or dollar sign characters if you desire to override the `RegExp()` treatment of regular expressions. JavaScript regular expressions are normally written using this literal syntax.

Now that you know something about what regular expressions are and how they can be used in a JavaScript program, the remainder of this section will discuss several regular expression features that are frequently used in browser-based JavaScript programs. Refer to the ECMAScript reference [ECMA-262], particularly Section 15.10, for complete details.

The simplest form of regular expression is a character that is not one of the regular expression special characters, which are

```
^ $ \ . * + ? ( ) [ ] { } |
```

For example, underscore (_) is not one of the special characters, so `_` is the regular expression that represents the set of all strings that contain the underscore character. A special character is escaped by preceding it with a backslash. As an example, `\$$` represents the set of strings that end with a dollar sign.

Another simple regular expression is the period (dot) `.`, which represents any character except for a line terminator. In other words, dot is the wildcard character in regular expressions. *Escape code* regular expressions, such as `\d`, similarly represent multiple characters. Table 4.10 lists all of the JavaScript escape codes representing multiple characters.

Simple regular expressions can be composed into more complex regular expressions using one of three types of operators. We've already seen an example of the first of these operators, *concatenation*. In general, the concatenation of two regular expressions represents the set of strings that consist of any string represented by the first regular expression concatenated with any string represented by the second expression. Thus,

```
^\d\. \w$
```

TABLE 4.10 JavaScript Multicharacter Escape Codes

Escape Code	Characters Represented
`\d`	Digit: 0 through 9
`\D`	Any character except those matched by `\d`
`\s`	Space: any JavaScript white space or line terminator (space, tab, line feed, etc.)
`\S`	Any character except those matched by `\s`
`\w`	"Word" character: any letter (`a` through `z` and `A` through `Z`), digit (`0` through `9`), or underscore (`_`)
`\W`	Any character except those matched by `\w`

represents the set of strings beginning with a single digit followed immediately by a period, a space, and terminating a "word" character (letter, digit, or underscore). Notice that white space is significant within a regular expression. For instance, the set of strings represented by the last regular expression does not include either of the strings

```
3.A   or   7.  J
```

—the first because it has no space, and the second because it has two spaces.

When we want to concatenate a regular expression with itself multiple times, we can use the *quantifier* shorthand notation. For example, the set of all strings of exactly three digits is represented by the regular expression

```
\d{3}
```

This notation, viewed as a postfix unary operator, has higher precedence than both concatenation and union (discussed next). Therefore,

```
-\d{3}
```

represents strings that begin with a `-` followed by three digits.

The second regular expression operator is *union*, which is represented by the pipe symbol `|`. For example,

```
\d|\s
```

represents the set consisting of all digit and white space characters. The concatenation operator takes precedence over union, so the set of strings represented by the regular expression

```
\+|-\d|\s
```

consists of `+` (escaped because it is a special character), the two-character strings beginning with `-` followed by a digit, and the white space characters. The default precedence can be overridden using parentheses. Thus,

```
(\+|-)\d|\s
```

represents the set of two-character strings beginning with either a + or – followed by a digit plus the white space characters.

It would be tedious (and error prone) to use the union notation directly to represent a set such as the set of all lowercase letters. JavaScript provides a *character class* notation that can be used for such purposes. For instance, the set of lowercase letters can be represented by

```
[a-z]
```

Similarly, the escape code `\w` is equivalent to the regular expression

```
[a-zA-Z0-9]|_
```

Notice that, while - is not normally a special character in regular expressions, it is special within a character class and must be escaped by a preceding backslash in order to represent the - character itself.

A variation of the quantifier notation can be used to represent unions of concatenations. For example,

```
\d{3,6}
```

is shorthand for any string of from three through six digits, that is,

```
\d\d|\d\d\d|\d\d\d\d|\d\d\d\d\d
```

By definition, zero concatenations of any string with itself results in the empty string. Thus, the regular expression

```
(\+|-){0,1}\d
```

represents the set of strings that begin with either the empty string or a plus or minus sign, followed by a digit. In other words, the leading sign character is optional. JavaScript uses the symbol `?` for this purpose. That is to say, the quantifier `{0,1}` occurs so often that it has its own shorthand symbol, `?`. Therefore, the expression displayed would normally be written as

```
(\+|-)?\d
```

So far, the regular expressions we've seen have all represented finite sets of strings. The final operator, the *Kleene star*, allows us to represent infinitely large sets. For example, the regular expression

```
\d*
```

represents the set of strings of any number of digits, including the string of no digits at all (the empty string). In general, the Kleene star following a regular expression represents the set created by concatenating strings from the original set with one another an arbitrary number of times. This postfix unary operator has the highest precedence of all regular expression operators.

For example, assume that a certain application requires the use of passwords and that each password must contain at least one digit and at least one letter and may only contain digits, letters, and underscores. The following is a JavaScript regular expression that represents the set of valid passwords for this application:

```
\w*(\d\w*[a-zA-Z]|[a-zA-Z]\w*\d)\w*
```

4.13 JavaScript Debuggers

At this point, you should know quite a bit about writing JavaScript programs. We'll now briefly learn about debugging JavaScript programs using interactive debugging software. I'll also mention some of the ways in which JavaScript debugging differs from, say, Java debugging.

The first question is whether or not you already have a debugger and, if not, where to obtain one. If you are debugging your JavaScript code using IE6, your system may or may not already have a debugger present. To find out, select **Tools|Internet Options . . .** from the IE6 menu and click on the Advanced tab in the popup window that appears. Make sure that the checkbox for "Disable Script Debugging (Internet Explorer)" is unchecked, and click the OK button in the pop-up. Now load a JavaScript program into IE6 that has an error in it (the file `error.js`, available at the textbook Web site, can be used). If a pop-up window appears asking if you would like to debug, then you have a debugger installed. Otherwise, Microsoft provides a free Script Debugger that is currently available at `http://msdn.microsoft.com/downloads/list/webdev.asp`.

On the other hand, if you are debugging using Mozilla and you downloaded Mozilla according to the instructions in Appendix A, then you automatically have an interactive JavaScript debugger named Venkman. Since Venkman can be run on all major operating systems, I'll use it as a specific example in this section. Other interactive debuggers, such as Microsoft's Script Debugger, generally have similar features, although of course details will differ. You should see the manufacturer's documentation for your debugger for complete details. For Venkman, the documentation can be accessed through the Venkman **Help** menu.

To debug a JavaScript program with the Venkman debugger, begin by starting Mozilla and then opening the debugger window (Figure 4.18) by selecting **Tools|Web Development|JavaScript Debugger** from the Mozilla menu. Next, in the browser window, open the HTML document that loads the JavaScript program to be debugged. The Loaded Scripts panel in the upper left portion of the debugger window will display the name of your JavaScript file. Right-clicking the file name and selecting Find File from the pop-up context menu then loads the file into the source code panel in the upper right portion of the debugger window.

Figure 4.18 shows the Venkman debugger window after I loaded the `TestJs.html` document into Mozilla and then entered `LocalScope.js` when prompted. `TestJs.html`

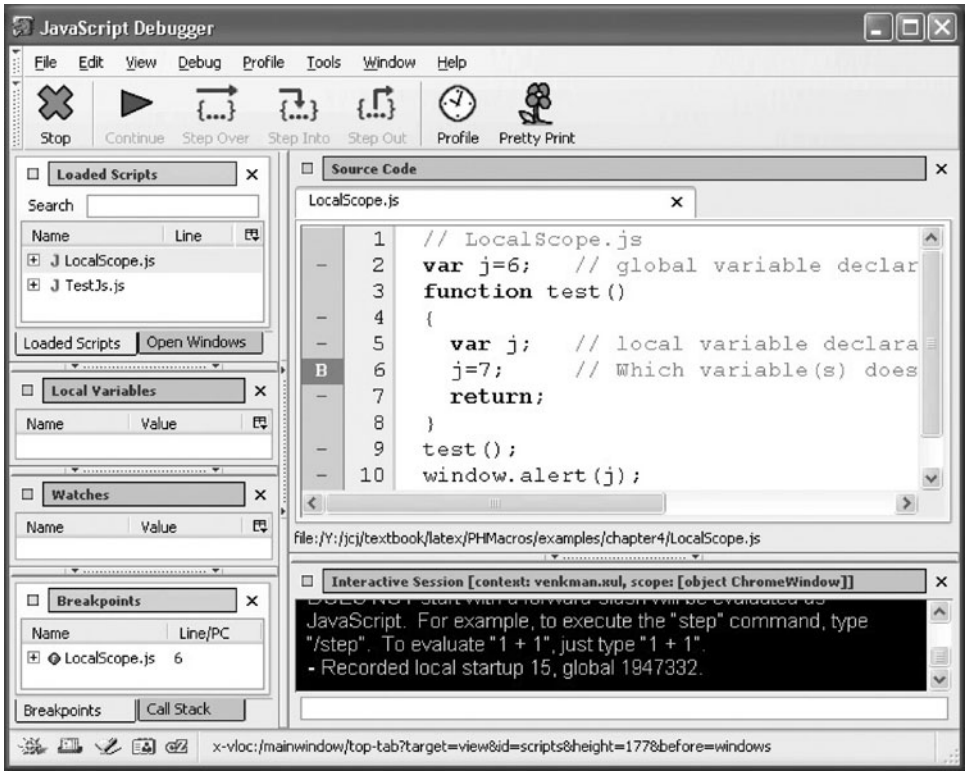

FIGURE 4.18 The Venkman JavaScript debugger with a breakpoint set in a loaded JavaScript file.

loaded `TestJs.js`, which in turn loaded `LocalScope.js`. Thus, the Loaded Scripts panel shows two JavaScript files. I then selected `LocalScope.js` for debugging as just described. Next, I set a *breakpoint* on line 6 of this file by clicking to the left of the line number, which caused a `B` to appear where I clicked. The breakpoint is also listed in the Breakpoints panel at the bottom left of the debugger window.

Now if I return to my browser window, reload it, and again enter `LocalScope.js` when prompted, the scripting engine will *break* (pause) just before executing line 6 of `LocalScope.js`. I can then use the Step buttons to step through the code one line at a time. The Step Over button causes the scripting engine to execute until reaching the next line of code within the current file, so this would take me to line 7 of `LocalScope.js`. The Step Into button behaves similarly unless the current line is a call to a user-defined function or method, in which case the scripting engine enters that function or method and then breaks. Step Out, on the other hand, executes a function or method until it has returned to the calling statement, at which point a break occurs. Clicking Step Out after the break at line 6 of `LocalScope.js` will cause the scripting engine to execute until reaching the end of line 9, the statement that called the `test()` function containing line 6. These Step buttons enable you to carefully examine the flow of control through the program. Alternatively, clicking

on Continue will cause the scripting engine to execute until either another breakpoint or the end of the program is reached.

In addition to control flow, we are often interested in examining the values of variables as a program executes. The Local Variables panel in Venkman provides this information, but in a way that is somewhat confusing due to JavaScript's internal representation of variables. To begin with, I suggest clicking on the small box in the upper left corner of the Local Variables panel; this will float the panel out into a separate window that can be independently resized (Figure 4.19).

As shown in Figure 4.19, the Local Variables panel has just two top-level variables: `scope` and `this`. This particular figure shows the local variables at the time of the break at line 6 of `LocalScope.js`. When local variables are examined in the midst of executing a function or method, `scope` represents an object that contains the local variables and formal parameters of the function as properties. So the `j` under `scope` represents the local variable declared at line 5 of `LocalScope.js`. The variable's value is shown as `void` (Venkman's representation of the Undefined value) because line 6 has not yet been executed, so the local `j` has not yet been assigned a value. If we view Local Variables at a point in the execution that is outside all functions and methods, then `scope` will represent the `window` (global) object. Similarly, except when executing a method, `this` represents the `window` object. In a method, as mentioned earlier, `this` represents the object on which the method is called.

What about global variables? As indicated earlier, global variables are stored as properties of the `window` object. Therefore, if either `scope` or `this` represents the `window` object, then all global variables will appear as properties and can be examined by clicking the `+` to the left of the object name. So, in this example, we would find the global `j` (with value 6) as a property under the `this` object. However, be warned that `window` is given a great many other properties automatically, so your program's global variables will not be immediately apparent. If neither `scope` nor `this` represents the `window` object (which is the case if a method is being executed), then the `window` property under the `this` object will represent the global object and can be examined for the values of global variables.

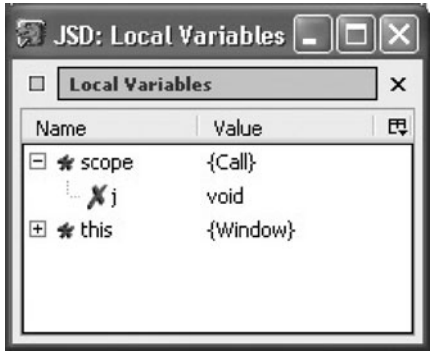

FIGURE 4.19 The Venkman Local Variables panel floated into a separate window.