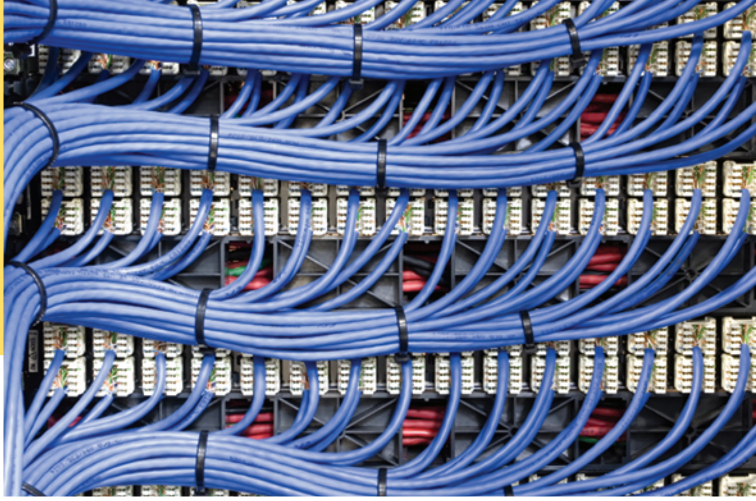
CHAPTER 23

# INTERNET NETWORKING


© Felix Alim/iStockphoto.

## CHAPTER GOALS

To understand the concept of sockets

To send and receive data through sockets

To implement network clients and servers

To communicate with web servers and server-side applications through the Hypertext Transfer Protocol (HTTP)

## CHAPTER CONTENTS

**23.1 THE INTERNET PROTOCOL** W988

**23.2 APPLICATION LEVEL PROTOCOLS** W990

**23.3 A CLIENT PROGRAM** W993

**23.4 A SERVER PROGRAM** W996

HT 1 Designing Client/Server Programs W1003

**23.5 URL CONNECTIONS** W1004

PT 1 Use High-Level Libraries W1007

You probably have quite a bit of experience with the Internet, the global network that links together millions of computers. In particular, you use the Internet whenever you browse the World Wide Web. Note that the Internet is not the same as the "Web". The World Wide Web is only one of many services offered over the Internet. E-mail, another popular service, also uses the Internet, but its implementation differs from that of the Web. In this chapter, you will see what goes on "under the hood" when you send an e-mail message or when you retrieve a web page from a remote server. You will also learn how to write programs that fetch data from sites across the Internet and how to write server programs that can serve information to other programs.

## 23.1 The Internet Protocol

The *Internet* is a worldwide collection of networks, routing equipment, and computers using a common set of protocols to define how each party will interact with each other.

Computers can be connected with each other through a variety of physical media. In a computer lab, for example, computers are connected by network cabling. Electrical impulses representing information flow across the cables. If you use a DSL modem to connect your computer to the Internet, the signals travel across a regular telephone wire, encoded as tones. On a wireless network, signals are sent by transmitting a modulated radio frequency. The physical characteristics of these transmissions differ widely, but they ultimately consist of sending and receiving streams of zeroes and ones along the network connection.

These zeroes and ones represent two kinds of information: *application data,* the data that one computer actually wants to send to another, and *network protocol data,* the data that describe how to reach the intended recipient and how to check for errors and data loss in the transmission. The protocol data follow certain rules set forth by the Internet Protocol Suite, also called TCP/IP, after the two most important protocols in the suite. These protocols have become the basis for connecting computers around the world over the Internet. We will discuss TCP and IP in this chapter.

Suppose that a computer A wants to send data to a computer B, both on the Internet. The computers aren't connected directly with a cable, as they could be if both were on the same local area network. Instead, A may be someone's home computer and connected to an *Internet service provider (ISP),* which is in turn connected to an *Internet access point;* B might be a computer on a local area network belonging to a large firm that has an Internet access point of its own, which may be half a world away from A. The **Internet** itself, finally, is a complex collection of pathways on which a message can travel from one Internet access point to, eventually, any other Internet access point (see Figure 1). Those connections carry millions of messages, not just the data that A is sending to B.

For the data to arrive at its destination, it must be marked with a *destination address.* In IP, addresses are denoted by sequences of four numbers, each one byte (that is, between 0 and 255); for example, 130.65.86.66. (Because there aren't enough four-byte addresses for all devices that would like to connect to the Internet, these addresses have been extended to sixteen bytes. For simplicity, we use the classic four-byte addresses in this chapter.) In order to send data, A needs to know the Internet

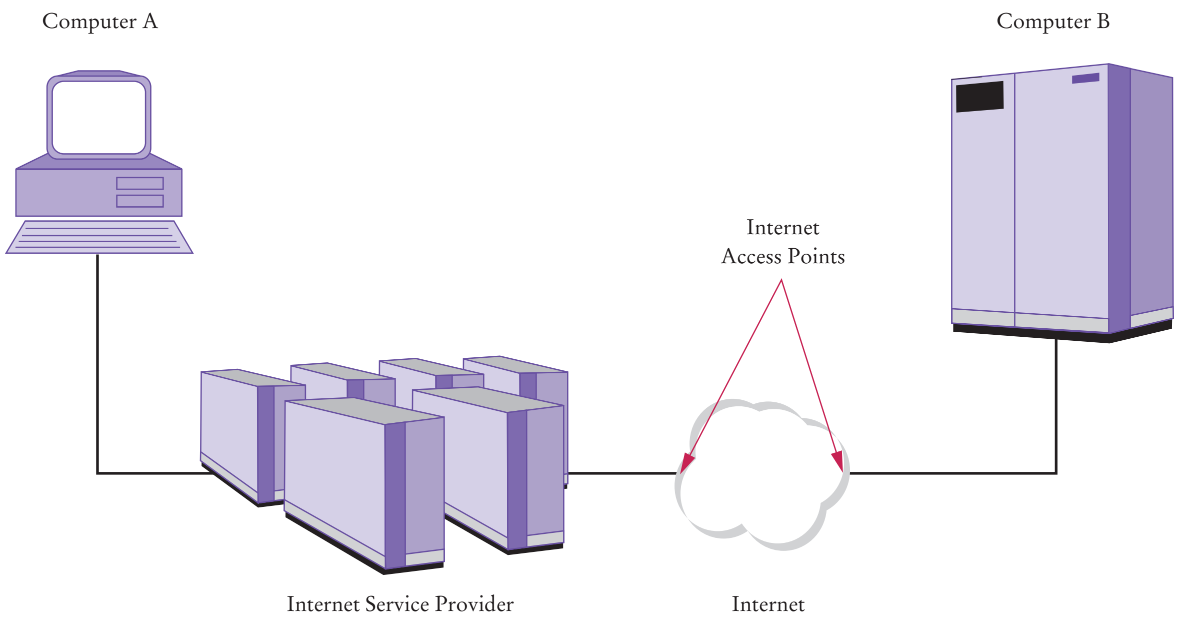


**Figure 1** Two Computers Communicating Across the Internet

address of B and include it in the protocol portion when sending the data across the Internet. The routing software that is distributed across the Internet can then deliver the data to B.

Of course, addresses such as 130.65.86.66 are not easy to remember. You would not be happy if you had to use number sequences every time you sent e-mail or requested information from a web server. On the Internet, computers can have so-called *domain names* that are easier to remember, such as `cs.sjsu.edu` or `horstmann.com`. A special service called the *Domain Name System (DNS)* translates between domain names and Internet addresses. Thus, if computer A wants to have information from `horstmann.com`, it first asks the DNS to translate this domain name into a numeric Internet address; then it includes the numeric address with the request.

One interesting aspect of IP is that it breaks large chunks of data up into more manageable *packets.* Each packet is delivered separately, and different packets that are part of the same transmission can take different routes through the Internet. Packets are numbered, and the recipient reassembles them in the correct order.

TCP/IP is the abbreviation for *Transmission Control Protocol and Internet Protocol*, the pair of communication protocols designed to establish reliable transmission of data between two computers on the Internet.

The Internet Protocol is used when attempting to deliver data from one computer to another across the Internet. If some data get lost or garbled in the process, IP has safeguards built in to make sure that the recipient is aware of that unfortunate fact and doesn't rely on incomplete data. However, IP has no provision for retrying an incomplete transmission. That is the job of a higher-level protocol, the *Transmission Control Protocol (TCP).* This protocol attempts reliable delivery of data, with retries if there are failures, and it notifies the sender whether or not the attempt succeeded. Most, but not all, Internet programs use TCP for reliable delivery. (Exceptions are "streaming media" services, which bypass the slower TCP for the highest possible throughput and tolerate occasional information loss. However, the most popular Internet services—the World Wide Web and e-mail—use TCP.) TCP is independent of the Internet Protocol; it could in principle be used with another lower-level network protocol. However, in practice, TCP over IP (often called TCP/IP) is the most commonly used combination. We will focus on TCP/IP networking in this chapter.

A TCP connection requires the Internet addresses and port numbers of both end points.

A computer that is connected to the Internet may have programs for many different purposes. For example, a computer may run both a web server program and a mail server program. When data are sent to that computer, they need to be marked so that they can be forwarded to the appropriate program. TCP uses *port numbers* for this purpose. A port number is an integer between `0` and 65,535. The sending computer must know the port number of the receiving program and include it with the transmitted data. Some applications use "well-known" port numbers. For example, by convention, web servers use port `80`, whereas mail servers running the Post Office Protocol (POP) use port `110`. A TCP connection, therefore, requires

- The Internet address of the recipient.
- The port number of the recipient.
- The Internet address of the sender.
- The port number of the sender.

You can think of a TCP connection as a "pipe" between two computers that links the two ports together. Data flow in either direction through the pipe. In practical programming situations, you simply establish a connection and send data across it without worrying about the details of the TCP/IP mechanism. You will see how to establish such a connection in Section 23.3.

**SELF CHECK**

1. What is the difference between an IP address and a domain name?
2. Why do some streaming media services not use TCP?

**Practice It** Now you can try these exercises at the end of the chapter: R23.1, R23.2, R23.3.

## 23.2 Application Level Protocols

HTTP, or *Hypertext Transfer Protocol*, is the protocol that defines communication between web browsers and web servers.

A URL, or *Uniform Resource Locator*, is a pointer to an information resource (such as a web page or an image) on the World Wide Web.

In the preceding section you saw how the TCP/IP mechanism can establish an Internet connection between two ports on two computers so that the two computers can exchange data. Each Internet application has a different *application protocol*, which describes how the data for that particular application are transmitted.

Consider, for example, HTTP: the **Hypertext Transfer Protocol**, which is used for the World Wide Web. Suppose you type a web address, called a **Uniform Resource Locator** (URL), such as `http://horstmann.com/index.html`, into the address window of your browser and ask the browser to load the page.

The browser now takes the following steps:

1. It examines the part of the URL between the double slash and the first single slash ("`horstmann.com`"), which identifies the computer to which you want to connect. Because this part of the URL contains letters, it must be a domain name rather than an Internet address, so the browser sends a request to a DNS

server to obtain the Internet address of the computer with domain name `horstmann.com`.

2. From the `http:` prefix of the URL, the browser deduces that the protocol you want to use is HTTP, which by default uses port 80.
3. It establishes a TCP/IP connection to port 80 at the Internet address it obtained in Step 1.
4. It deduces from the `/index.html` suffix that you want to see the file `/index.html`, so it sends a request, formatted as an HTTP command, through the connection that was established in Step 3. The request looks like this:

   ```
   GET /index.html HTTP/1.1
   Host: horstmann.com
   blank line
   ```

   (The host is needed because a web server can host multiple domains with the same Internet address.)
5. The web server running on the computer whose Internet address is the one the browser obtained in Step 1 receives the request and decodes it. It then fetches the file `/index.html` and sends it back to the browser on your computer.
6. The browser displays the contents of the file. Because it happens to be an HTML file, the browser translates the HTML tags into fonts, bullets, separator lines, and so on. If the HTML file contains images, then the browser makes more `GET` requests, one for each image, through the same connection, to fetch the image data. (Appendix J contains a summary of the most frequently used HTML tags.)

The Telnet program is a useful tool for establishing test connections with servers.

You can try the following experiment to see this process in action. The "Telnet" program enables a user to type characters for sending to a remote computer and view characters that the remote computer sends back. On Windows, you need to enable the Telnet program in the control panel. UNIX, Linux, and Mac OS X systems normally have Telnet preinstalled.

For this experiment, you want to start Telnet with a host of `horstmann.com` and port 80. To start the program from the command line, simply type

```
telnet horstmann.com 80
```

Table 1 HTTP Commands

| Command | Meaning |
|---|---|
| GET | Return the requested item |
| HEAD | Request only the header information of an item |
| OPTIONS | Request communications options of an item |
| POST | Supply input to a server-side command and return the result |
| PUT | Store an item on the server |
| DELETE | Delete an item on the server |
| TRACE | Trace server communication |

Once the program starts, type very carefully, without making any typing errors and without pressing the backspace key,

```
GET / HTTP/1.1
Host: horstmann.com
```

Then press the Enter key twice.

The first / denotes the root page of the web server. Note that there are spaces before and after the first /, but there are no spaces in HTTP/1.1.

On Windows, you will not see what you type, so you should be extra careful when typing in the commands.

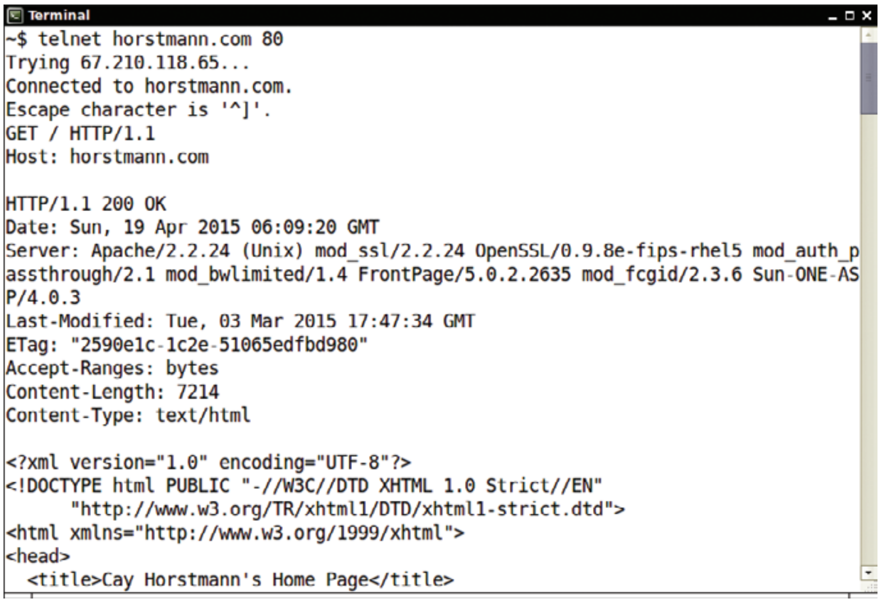
The server now sends a response to the request—see Figure 2. The response, of course, consists of the root web page that you requested. The Telnet program is not a browser and does not understand HTML tags, so it simply displays the HTML file—text, tags, and all.

The HTTP GET command requests information from a web server. The web server returns the requested item, which may be a web page, an image, or other data.

The GET command is one of the commands of HTTP. Table 1 shows the other commands of the protocol. As you can see, the protocol is pretty simple.

By the way, be sure not to confuse HTML with HTTP. **HTML** is a *document format* (with commands such as <h1> or <ul>) that describes the structure of a document, including headings, bulleted lists, images, hyperlinks, and so on. **HTTP** is a *protocol* (with commands such as GET and POST) that describes the command set for web server requests. Web *browsers* know how to display HTML documents and how to issue HTTP commands. Web *servers* know nothing about HTML. They merely understand HTTP and know how to fetch the requested items. Those items may be HTML documents, GIF or JPEG images, or any other data that a web browser can display.

HTTP is just one of many application protocols in use on the Internet. Another commonly used protocol is the Post Office Protocol (POP), which is used to download received messages from e-mail servers. To *send* messages, you use yet another protocol called the Simple Mail Transfer Protocol (SMTP). We don't want to go into



```
~$ telnet horstmann.com 80
Trying 67.210.118.65...
Connected to horstmann.com.
Escape character is '^]'.
GET / HTTP/1.1
Host: horstmann.com

HTTP/1.1 200 OK
Date: Sun, 19 Apr 2015 06:09:20 GMT
Server: Apache/2.2.24 (Unix) mod_ssl/2.2.24 OpenSSL/0.9.8e-fips-rhel5 mod_auth_p
assthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635 mod_fcgid/2.3.6 Sun-ONE-AS
P/4.0.3
Last-Modified: Tue, 03 Mar 2015 17:47:34 GMT
ETag: "2590e1c-1c2e-51065edfbd980"
Accept-Ranges: bytes
Content-Length: 7214
Content-Type: text/html

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Cay Horstmann's Home Page</title>
```

**Figure 2** Using Telnet to Connect to a Web Server

```
+OK San Quentin State POP server
USER harryh
+OK Password required for harryh
PASS secret
+OK harryh has 2 messages (320 octets)
STAT
+OK 2 320
RETR 1
+OK 120 octets
the message is included here
DELE 1
+OK message 1 deleted
QUIT
+OK POP server signing off
```

Black = mail client requests
Color = mail server responses

**Figure 3** A Sample POP Session

the details of these protocols, but Figure 3 gives you a flavor of the commands used by the Post Office Protocol.

Both HTTP and POP use plain text, which makes it particularly easy to test and debug client and server programs (see How To 23.1).

3. Why don't you need to know about HTTP when you use a web browser?
4. Why is it important that you don't make typing errors when you type HTTP commands in Telnet?

**Practice It** Now you can try these exercises at the end of the chapter: R23.13, R23.14, R23.15.

## 23.3 A Client Program

In this section you will see how to write a Java program that establishes a TCP connection to a server, sends a request to the server, and prints the response.

A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.

In the terminology of TCP/IP, there is a **socket** on each side of the connection (see Figure 4). In Java, a client establishes a socket with a call

```
Socket s = new Socket(hostname, portnumber);
```

For example, to connect to the HTTP port of the server `horstmann.com`, you use

```
final int HTTP_PORT = 80;
Socket s = new Socket("horstmann.com", HTTP_PORT);
```

The socket constructor throws an `UnknownHostException` if it can't find the host.

Once you have a socket, you obtain its input and output streams:

```
InputStream instream = s.getInputStream();
OutputStream outstream = s.getOutputStream();
```

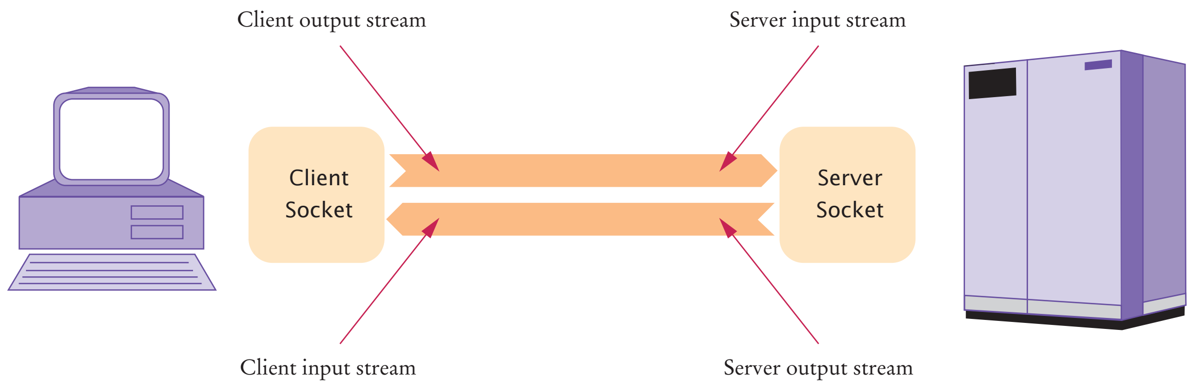


**Figure 4** Client and Server Sockets

When you send data to `outstream`, the socket automatically forwards it to the server. The socket catches the server's response, and you can read the response through `instream` (see Figure 4).

When transmission over a socket is complete, remember to close the socket.

When you are done communicating with the server, you should close the socket. This is best done with a `try`-with-resources statement:

```
try (Socket s = . . . )
{
   . . .
} // s.close() called here
```

For text protocols, turn the socket streams into scanners and writers.

In Chapter 21, you saw that the `InputStream` and `OutputStream` classes are used for reading and writing bytes. If you want to communicate with the server by sending and receiving text, you should turn the streams into scanners and writers, as follows:

```
Scanner in = new Scanner(instream);
PrintWriter out = new PrintWriter(outstream);
```

A print writer *buffers* the characters that you send to it. That is, characters are not immediately sent to their destination. Instead, they are placed into an array. When the array is full, then the print writer sends all characters in the array to its destination. The advantage of buffering is increased performance—it takes some amount of time to contact the destination and send it data, and it is expensive to pay for that contact time for every character. However, when communicating with a server that responds to requests, you want to make sure that the server gets a complete request. Therefore, you need to *flush* the buffer manually whenever you send a command:

```
out.print(command);
out.flush();
```

Flush the writer attached to a socket at the end of every command. Then the command is sent to the server, even if the writer's buffer is not completely filled.

The `flush` method empties the buffer and forwards all waiting characters to the destination.

The `WebGet` program at the end of this section lets you retrieve any item from a web server. You need to specify the host and the item from the command line. For example,

```
java WebGet horstmann.com /
```

The `/` item denotes the root page of the web server that listens to port 80 of the host `horstmann.com`. Note that there is a space before the `/`.

The `WebGet` program establishes a connection to the host, sends a `GET` command to the host, and then receives input from the server until the server closes its connection.

**section_3/WebGet.java**

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
   This program demonstrates how to use a socket to communicate
   with a web server. Supply the name of the host and the
   resource on the command line, for example,
   java WebGet horstmann.com index.html.
*/
public class WebGet
{
   public static void main(String[] args) throws IOException
   {
      // Get command-line arguments

      String host;
      String resource;

      if (args.length == 2)
      {
         host = args[0];
         resource = args[1];
      }
      else
      {
         System.out.println("Getting / from horstmann.com");
         host = "horstmann.com";
         resource = "/";
      }

      // Open socket

      final int HTTP_PORT = 80;
      try (Socket s = new Socket(host, HTTP_PORT))
      {
         // Get streams

         InputStream instream = s.getInputStream();
         OutputStream outstream = s.getOutputStream();

         // Turn streams into scanners and writers

         Scanner in = new Scanner(instream);
         PrintWriter out = new PrintWriter(outstream);

         // Send command

         String command = "GET " + resource + " HTTP/1.1\n"
            + "Host: " + host + "\n\n";
         out.print(command);
         out.flush();

         // Read server response

         while (in.hasNextLine())
         {
            String input = in.nextLine();
            System.out.println(input);
         }
      }

      // The try-with-resources statement closes the socket
   }
}
```

**Program Run**

```
Getting / from horstmann.com
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 14:15:04 GMT
Server: Apache/1.3.41 (Unix) Sun-ONE-ASP/4.0.2
. . .
Content-Length: 6654
Content-Type: text/html

<html>
<head><title>Cay Horstmann's Home Page</title></head>
<body>
<h1>Welcome to Cay Horstmann's Home Page</h1>
. . .
</body>
</html>
```

SELF CHECK

5. What happens if you call `WebGet` with a nonexistent resource, such as `wombat.html` at `horstmann.com`?
6. How do you open a socket to read e-mail from the POP server at `e-mail.sjsu.edu`?

**Practice It** Now you can try these exercises at the end of the chapter: R23.7, R23.8, E23.1, E23.2.

## 23.4 A Server Program

Now that you have seen how to write a network client, we will turn to the server side. In this section we will develop a server program that enables clients to manage a set of bank accounts in a bank.

Whenever you develop a server application, you need to specify some application-level protocol that clients can use to interact with the server. For the purpose of this example, we will create a "Simple Bank Access Protocol". Table 2 shows the protocol format. Of course, this is just a toy protocol to show you how to implement a server.

The server program waits for clients to connect to a particular port. We choose port 8888 for this service. This number has not been preassigned to another service, so it is unlikely to be used by another server program. To listen to incoming connections, you use a *server socket.* To construct a server socket, you need to supply the port number:

```
ServerSocket server = new ServerSocket(8888);
```

The ServerSocket class is used by server applications to listen for client connections.

The accept method of the ServerSocket class waits for a client connection. When a client connects, then the server program obtains a socket through which it communicates with the client:

```
Socket s = server.accept();
BankService service = new BankService(s, bank);
```

The BankService class carries out the service. This class implements the Runnable interface, and its run method will be executed in each thread that serves a client connection. The run method gets a scanner and writer from the socket in the same way as we discussed in the preceding section. Then it executes the following method:

```
public void doService() throws IOException
{
   while (true)
   {
      if (!in.hasNext()) { return; }
      String command = in.next();
      if (command.equals("QUIT")) { return; }
      executeCommand(command);
   }
}
```

The executeCommand method processes a single command. If the command is DEPOSIT, then it carries out the deposit:

```
int account = in.nextInt();
double amount = in.nextDouble();
bank.deposit(account, amount);
```

The WITHDRAW command is handled in the same way. After each command, the account number and new balance are sent to the client:

```
out.println(account + " " + bank.getBalance(account));
```

The doService method returns to the run method if the client closed the connection or the command equals "QUIT". Then the run method closes the socket and exits.

Let us go back to the point where the server socket accepts a connection and constructs the BankService object. At this point, we could simply call the run method. But then our server program would have a serious limitation: only one client could connect to it at any point in time. To overcome that limitation, server programs spawn a new thread whenever a client connects. Each thread is responsible for serving one client.

Table 2 A Simple Bank Access Protocol

| Client Request | Server Response | Description |
|---|---|---|
| BALANCE *n* | *n* and the balance | Get the balance of account *n* |
| DEPOSIT *n a* | *n* and the new balance | Deposit amount *a* into account *n* |
| WITHDRAW *n a* | *n* and the new balance | Withdraw amount *a* from account *n* |
| QUIT | None | Quit the connection |

Our `BankService` class implements the `Runnable` interface. Therefore, the server program `BankServer` simply starts a thread with the following instructions:

```
Thread t = new Thread(service);
t.start();
```

The thread dies when the client quits or disconnects and the `run` method exits. In the meantime, the `BankServer` loops back to accept the next connection.

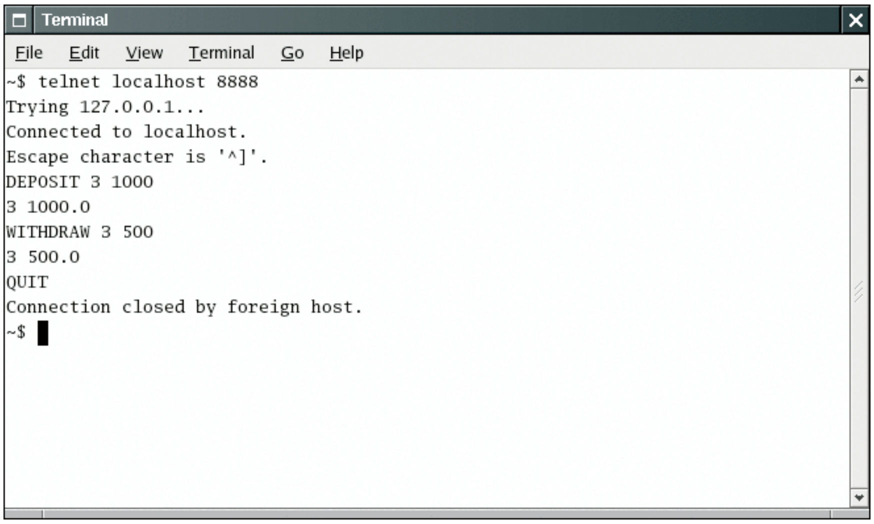
```
while (true)
{
   try (Socket s = server.accept())
   {
      BankService service = new BankService(s, bank);
      Thread t = new Thread(service);
      t.start();
   }
}
```

The server program never stops. When you are done running the server, you need to kill it. For example, if you started the server in a shell window, press Ctrl+C.

To try out the program, run the server. Then use Telnet to connect to `localhost`, port number 8888. Start typing commands. Here is a typical dialog (see Figure 5):

```
DEPOSIT 3 1000
3 1000.0
WITHDRAW 3 500
3 500.0
QUIT
```

Alternatively, you can use a client program that connects to the server. You will find a sample client program at the end of this section.



```
~$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
DEPOSIT 3 1000
3 1000.0
WITHDRAW 3 500
3 500.0
QUIT
Connection closed by foreign host.
~$
```

**Figure 5** Using the Telnet Program to Connect to the Bank Server

### section_4/BankServer.java

```java
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
   A server that executes the Simple Bank Access Protocol.
*/
public class BankServer
{
   public static void main(String[] args) throws IOException
   {
      final int ACCOUNTS_LENGTH = 10;
      Bank bank = new Bank(ACCOUNTS_LENGTH);
      final int SBAP_PORT = 8888;
      ServerSocket server = new ServerSocket(SBAP_PORT);
      System.out.println("Waiting for clients to connect...");

      while (true)
      {
         try (Socket s = server.accept())
         {
            System.out.println("Client connected.");
            BankService service = new BankService(s, bank);
            Thread t = new Thread(service);
            t.start();
         }
      }
   }
}
```

### section_4/BankService.java

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
   Executes Simple Bank Access Protocol commands
   from a socket.
*/
public class BankService implements Runnable
{
   private Socket s;
   private Scanner in;
   private PrintWriter out;
   private Bank bank;

   /**
      Constructs a service object that processes commands
      from a socket for a bank.
      @param aSocket the socket
      @param aBank the bank
   */
   public BankService(Socket aSocket, Bank aBank)
   {
      s = aSocket;
      bank = aBank;
   }

   public void run()
   {
      try
      {
         in = new Scanner(s.getInputStream());
         out = new PrintWriter(s.getOutputStream());
         doService();
      }
      catch (IOException exception)
      {
         exception.printStackTrace();
      }
   }

   /**
      Executes all commands until the QUIT command or the
      end of input.
   */
   public void doService() throws IOException
   {
      while (true)
      {
         if (!in.hasNext()) { return; }
         String command = in.next();
         if (command.equals("QUIT")) { return; }
         else { executeCommand(command); }
      }
   }

   /**
      Executes a single command.
      @param command the command to execute
   */
   public void executeCommand(String command)
   {
      int account = in.nextInt();
      if (command.equals("DEPOSIT"))
      {
         double amount = in.nextDouble();
         bank.deposit(account, amount);
      }
      else if (command.equals("WITHDRAW"))
      {
         double amount = in.nextDouble();
         bank.withdraw(account, amount);
      }
      else if (!command.equals("BALANCE"))
      {
         out.println("Invalid command");
         out.flush();
         return;
      }
      out.println(account + " " + bank.getBalance(account));
      out.flush();
   }
}
```

**section_4/Bank.java**

```java
/**
   A bank consisting of multiple bank accounts.
*/
public class Bank
{
   private BankAccount[] accounts;

   /**
      Constructs a bank account with a given number of accounts.
      @param size the number of accounts
   */
   public Bank(int size)
   {
      accounts = new BankAccount[size];
      for (int i = 0; i < accounts.length; i++)
      {
         accounts[i] = new BankAccount();
      }
   }

   /**
      Deposits money into a bank account.
      @param accountNumber the account number
      @param amount the amount to deposit
   */
   public void deposit(int accountNumber, double amount)
   {
      BankAccount account = accounts[accountNumber];
      account.deposit(amount);
   }

   /**
      Withdraws money from a bank account.
      @param accountNumber the account number
      @param amount the amount to withdraw
   */
   public void withdraw(int accountNumber, double amount)
   {
      BankAccount account = accounts[accountNumber];
      account.withdraw(amount);
   }

   /**
      Gets the balance of a bank account.
      @param accountNumber the account number
      @return the account balance
   */
   public double getBalance(int accountNumber)
   {
      BankAccount account = accounts[accountNumber];
      return account.getBalance();
   }
}
```

**section_4/BankClient.java**

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
   This program tests the bank server.
*/
public class BankClient
{
   public static void main(String[] args) throws IOException
   {
      final int SBAP_PORT = 8888;
      try (Socket s = new Socket("localhost", SBAP_PORT))
      {
         InputStream instream = s.getInputStream();
         OutputStream outstream = s.getOutputStream();
         Scanner in = new Scanner(instream);
         PrintWriter out = new PrintWriter(outstream);

         String command = "DEPOSIT 3 1000\n";
         System.out.print("Sending: " + command);
         out.print(command);
         out.flush();
         String response = in.nextLine();
         System.out.println("Receiving: " + response);

         command = "WITHDRAW 3 500\n";
         System.out.print("Sending: " + command);
         out.print(command);
         out.flush();
         response = in.nextLine();
         System.out.println("Receiving: " + response);

         command = "QUIT\n";
         System.out.print("Sending: " + command);
         out.print(command);
         out.flush();
      }
   }
}
```

### Program Run

```
Sending: DEPOSIT 3 1000
Receiving: 3 1000.0
Sending: WITHDRAW 3 500
Receiving: 3 500.0
Sending: QUIT
```

**SELF CHECK**

7. Why didn't we choose port 80 for the bank server?
8. Can you read data from a server socket?

**Practice It** Now you can try these exercises at the end of the chapter: E23.3, E23.4, P23.2.

## HOW TO 23.1 Designing Client/Server Programs

The bank server of this section is a typical example of a client/server program. A web browser/web server is another example. This How To outlines the steps to follow when designing a client/server application.

**Step 1** Determine whether it really makes sense to implement a stand-alone server and a matching client.

Many times it makes more sense to build a web application instead. Chapter 26 discusses the construction of web applications in detail. For example, the bank application of this section could easily be turned into a web application, using an HTML form with Withdraw and Deposit buttons. However, programs for chat or peer-to-peer file sharing cannot easily be implemented as web applications.

**Step 2** Design a communication protocol.

Figure out exactly what messages the client and server send to each other and what the success and error responses are.

With each request and response, ask yourself how the *end of data* is indicated.

- Do the data fit on a single line? Then the end of the line serves as the data terminator.
- Can the data be terminated by a special line (such as a blank line after the HTTP header or a line containing a period in SMTP)?
- Does the sender of the data close the socket? That's what a web server does at the end of a `GET` request.
- Can the sender indicate how many bytes are contained in the request? Web browsers do that in `POST` requests.

Use text, not binary data, for the communication between client and server. A text-based protocol is easier to debug.

**Step 3** Implement the server program.

The server listens for socket connections and accepts them. It starts a new thread for each connection. Supply a class that implements the `Runnable` interface. The `run` method receives commands, interprets them, and sends responses back to the client.

**Step 4** Test the server with the Telnet program.

Try out all commands in the communication protocol.

**Step 5** Once the server works, write a client program.

The client program interacts with the program user, turns user requests into protocol commands, sends the commands to the server, receives the response, and displays the response for the program user.

## 23.5 URL Connections

The URLConnection class makes it easy to communicate with a web server without having to issue HTTP commands.

In Section 23.3, you saw how to use sockets to connect to a web server and how to retrieve information from the server by sending HTTP commands. However, because HTTP is such an important protocol, the Java library contains a `URLConnection` class, which provides convenient support for the HTTP. The `URLConnection` class takes care of the socket connection, so you don't have to fuss with sockets when you want to retrieve from a web server. As an additional benefit, the `URLConnection` class can also handle FTP, the *file transfer protocol*.

The `URLConnection` class makes it very easy to fetch a file from a web server given the file's URL as a string. First, you construct a `URL` object from the URL in the familiar format, starting with the `http` or `ftp` prefix. Then you use the `URL` object's `openConnection` method to get the `URLConnection` object itself:

```
URL u = new URL("http://horstmann.com/index.html");
URLConnection connection = u.openConnection();
```

Then you call the `getInputStream` method to obtain an input stream:

```
InputStream instream = connection.getInputStream();
```

You can turn the stream into a scanner in the usual way, and read input from the scanner.

The URLConnection and HttpURLConnection classes can give you additional information about HTTP requests and responses.

The `URLConnection` class can give you additional useful information. To understand those capabilities, we need to have a closer look at HTTP requests and responses. You saw in Section 23.2 that the command for getting an item from the server is

```
GET item HTTP/1.1
Host: hostname
blank line
```

You may have wondered why you need to provide a blank line. This blank line is a part of the general request format. The first line of the request is a command, such as `GET` or `POST`. The command is followed by *request properties* (such as `Host:`). Some commands—in particular, the `POST` command—send input data to the server. The reason for the blank line is to denote the boundary between the request property section and the input data section.

A typical request property is `If-Modified-Since`. If you request an item with

```
GET item HTTP/1.1
Host: hostname
If-Modified-Since: date
blank line
```

the server sends the item only if it is newer than the date. Browsers use this feature to speed up redisplay of previously loaded web pages. When a web page is loaded, the browser stores it in a *cache* directory. When the user wants to see the same web page again, the browser asks the server to get a new page only if it has been modified since the date of the cached copy. If it hasn't been, the browser simply redisplays the cached copy and doesn't spend time downloading another identical copy.

The `URLConnection` class has methods to set request properties. For example, you can set the `If-Modified-Since` property with the `setIfModifiedSince` method:

```
connection.setIfModifiedSince(date);
```

You need to set request properties before calling the `getInputStream` method. The `URLConnection` class then sends to the web server all the request properties that you set.

Similarly, the response from the server starts with a status line followed by a set of response parameters. The response parameters are terminated by a blank line and followed by the requested data (for example, an HTML page). Here is a typical response:

```
HTTP/1.1 200 OK
Date: Thu, 09 Apr 2015 00:15:48 GMT
Server: Apache/1.3.3 (Unix)
Last-Modified: Tue, 03 Mar 2015 20:53:38 GMT
Content-Length: 4813
Content-Type: text/html
blank line
requested data
```

Normally, you don't see the response code. However, you may have run across bad links and seen a page that contained a response code `404 Not Found`. (A successful response has status `200 OK`.)

To retrieve the response code, you need to cast the `URLConnection` object to the `HttpURLConnection` subclass. You can retrieve the response code (such as the number 200 in this example, or the code 404 if a page was not found) and response message with the `getResponseCode` and `getResponseMessage` methods:

```
HttpURLConnection httpConnection = (HttpURLConnection) connection;
int code = httpConnection.getResponseCode(); // e.g., 404
String message = httpConnection.getResponseMessage(); // e.g., "Not found"
```

As you can see from the response example, the server sends some information about the requested data, such as the content length and the content type. You can request this information with methods from the `URLConnection` class:

```
int length = connection.getContentLength();
String type = connection.getContentType();
```

You need to call these methods after calling the `getInputStream` method.

To summarize: You don't need to use sockets to communicate with a web server, and you need not master the details of the HTTP protocol. Simply use the `URLConnection` and `HttpURLConnection` classes to obtain data from a web server, to set request properties, or to obtain response information.

The program at the end of this section puts the `URLConnection` class to work. The program fulfills the same purpose as that of Section 23.3—to retrieve a web page from a server—but it works at a higher level of abstraction. There is no longer a need to issue an explicit `GET` command. The `URLConnection` class takes care of that. Similarly, the parsing of the HTTP request and response headers is handled transparently to the programmer. Our sample program takes advantage of that fact. It checks whether the server response code is 200. If not, it exits. You can try that out by testing the program with a bad URL, like `http://horstmann.com/wombat.html`. Then the program prints a server response, such as `404 Not Found`.

This program completes our introduction to Internet programming with Java. You have seen how to use sockets to connect client and server programs. You also saw how to use the higher-level `URLConnection` class to obtain information from web servers.

### section_5/URLGet.java

```java
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLConnection;
import java.util.Scanner;

/**
   This program demonstrates how to use a URL connection
   to communicate with a web server. Supply the URL on
   the command line, for example
   java URLGet http://horstmann.com/index.html
*/
public class URLGet
{
   public static void main(String[] args) throws IOException
   {
      // Get command-line arguments

      String urlString;
      if (args.length == 1)
      {
         urlString = args[0];
      }
      else
      {
         urlString = "http://horstmann.com/";
         System.out.println("Using " + urlString);
      }

      // Open connection

      URL u = new URL(urlString);
      URLConnection connection = u.openConnection();

      // Check if response code is HTTP_OK (200)

      HttpURLConnection httpConnection
            = (HttpURLConnection) connection;
      int code = httpConnection.getResponseCode();
      String message = httpConnection.getResponseMessage();
      System.out.println(code + " " + message);
      if (code != HttpURLConnection.HTTP_OK)
      {
         return;
      }

      // Read server response

      InputStream instream = connection.getInputStream();
      Scanner in = new Scanner(instream);

      while (in.hasNextLine())
      {
         String input = in.nextLine();
         System.out.println(input);
      }
   }
}
```

**Program Run**

```
Using http://horstmann.com/
200 OK
<html>
<head><title>Cay Horstmann's Home Page</title></head>
<body>
<h1>Welcome to Cay Horstmann's Home Page</h1>
. . .
</body>
</html>
```

SELF CHECK

9. Why is it better to use a `URLConnection` instead of a socket when reading data from a web server?
10. What happens if you use the `URLGet` program to request an image (such as `http://horstmann.com/cay-tiny.gif`)?

**Practice It** Now you can try these exercises at the end of the chapter: P23.5, P23.6, P23.7.

Programming Tip 23.1

### Use High-Level Libraries

When you communicate with a web server to obtain data, you have two choices. You can make a socket connection and send `GET` and `POST` commands to the server over the socket. Or you can use the `URLConnection` class and have it issue the commands on your behalf.

Similarly, to communicate with a mail server, you can write programs that send SMTP and POP commands, or you can learn how to use the Java mail extensions. (See `http://oracle.com/technetwork/java/javamail/index.html` for more information on the Java Mail API.)

In such a situation, you may be tempted to use the low-level approach and send commands over a socket connection. It seems simpler than learning a complex set of classes. However, that simplicity is often deceptive. Once you go beyond the simplest cases, the low-level approach usually requires hard work. For example, to send binary e-mail attachments, you may need to master complex data encodings. The high-level libraries have all that knowledge built in, so you don't have to reinvent the wheel.

For that reason, you should not actually use sockets to connect to web servers. Always use the `URLConnection` class instead. Why did this book teach you about sockets if you aren't expected to use them? There are two reasons. Some client programs don't communicate with web or mail servers, and you may need to use sockets when a high-level library is not available. And, just as importantly, knowing what the high-level library does under the hood helps you understand it better. For the same reason, you saw in Chapter 16 how to implement linked lists, even though you probably will never program your own lists and will just use the standard `LinkedList` class.

## CHAPTER SUMMARY

**Describe the IP and TCP protocols.**

- The Internet is a worldwide collection of networks, routing equipment, and computers using a common set of protocols to define how each party will interact with each other.
- TCP/IP is the abbreviation for *Transmission Control Protocol and Internet Protocol*, the pair of communication protocols designed to establish reliable transmission of data between two computers on the Internet.
- A TCP connection requires the Internet addresses and port numbers of both end points.

**Describe the HTTP protocol.**

- HTTP, or *Hypertext Transfer Protocol*, is the protocol that defines communication between web browsers and web servers.
- A URL, or *Uniform Resource Locator,* is a pointer to an information resource (such as a web page or an image) on the World Wide Web.
- The Telnet program is a useful tool for establishing test connections with servers.
- The HTTP `GET` command requests information from a web server. The web server returns the requested item, which may be a web page, an image, or other data.

**Implement programs that use network sockets for reading data.**

- A socket is an object that encapsulates a TCP connection. To communicate with the other end point of the connection, use the input and output streams attached to the socket.
- When transmission over a socket is complete, remember to close the socket.
- For text protocols, turn the socket streams into scanners and writers.
- Flush the writer attached to a socket at the end of every command. Then the command is sent to the server, even if the writer's buffer is not completely filled.

**Implement programs that serve data over a network.**

- The `ServerSocket` class is used by server applications to listen for client connections.

**Use the `URLConnection` class to read data from a web server.**

- The `URLConnection` class makes it easy to communicate with a web server without having to issue HTTP commands.
- The `URLConnection` and `HttpURLConnection` classes can give you additional information about HTTP requests and responses.

## STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.net.HttpURLConnection
   getResponseCode
   getResponseMessage
java.net.ServerSocket
   accept
   close
java.net.Socket
   close
   getInputStream
   getOutputStream
java.net.URL
   openConnection
java.net.URLConnection
   getContentLength
   getContentType
   getInputStream
   setIfModifiedSince
```

## REVIEW EXERCISES

■ **R23.1** What is the IP address of the computer that you are using at home? Does it have a domain name?

■ **R23.2** Can a computer somewhere on the Internet establish a network connection with the computer at your home? If so, what information does the other computer need to establish the connection?

■ **R23.3** What is a port number? Can the same computer receive data on two different ports?

■ **R23.4** What is a server? What is a client? How many clients can connect to a server at one time?

■ **R23.5** What is a socket? What is the difference between a `Socket` object and a `ServerSocket` object?

■ **R23.6** Under what circumstances would an `UnknownHostException` be thrown?

■■ **R23.7** What happens if the `Socket` constructor's second argument is not the same as the port number at which the server waits for connections?

■ **R23.8** When a socket is created, which of the following Internet addresses is used?

**a.** The address of the computer to which you want to connect

**b.** The address of your computer

**c.** The address of your ISP

■ **R23.9** What is the purpose of the `accept` method of the `ServerSocket` class?

■ **R23.10** After a socket establishes a connection, which of the following mechanisms will your client program use to read data from the server computer?

**a.** The `Socket` will fill a buffer with bytes.

**b.** You will use a `Reader` obtained from the `Socket`.

**c.** You will use an `InputStream` obtained from the `Socket`.

■ **R23.11** Why is it not common to work directly with the `InputStream` and `OutputStream` objects obtained from a `Socket` object?

■ **R23.12** When a client program communicates with a server, it sometimes needs to flush the output stream. Explain why.

■ **R23.13** What is the difference between HTTP and HTML?

■ **R23.14** Try out the `HEAD` command of the HTTP protocol. What command did you use? What response did you get?

■■ **R23.15** Connect to a POP server that hosts your e-mail and retrieve a message. Provide a record of your session (but remove your password). If your mail server doesn't allow access on port 110, access it through SSL encryption (usually on port 995). Get a copy of the `openssl` utility and use the command

```
openssl s_client -connect servername:995
```

■ **R23.16** How can you communicate with a web server without using sockets?

■ **R23.17** What is the difference between a `URL` instance and a `URLConnection` instance?

■ **R23.18** What is a URL? How do you create an object of class `URL`? How do you connect to a URL?

## PRACTICE EXERCISES

■ **E23.1** Modify the `WebGet` program to print only the HTTP header of the returned HTML page. The HTTP header is the beginning of the response data. It consists of several lines, such as

```
HTTP/1.1 200 OK
Date: Tue, 14 Apr 2015 16:10:34 GMT
Server: Apache/1.3.19 (Unix)
Cache-Control: max-age=86400
Expires: Wed, 15 Apr 2015 16:10:34 GMT
Connection: close
Content-Type: text/html
```

followed by a blank line.

■ **E23.2** Modify the `WebGet` program to print only the *title* of the returned HTML page. An HTML page has the structure

```
<html><head><title> . . . </title></head><body> . . . </body></html>
```

For example, if you run the program by typing at the command line

```
java WebGet horstmann.com /
```

the output should be the title of the root web page at `horstmann.com`, such as `Cay Horstmann's Home Page`.

■■ **E23.3** Modify the `BankServer` program so that it can be terminated more elegantly. Provide another socket on port 8889 through which an administrator can log in. Support the commands `LOGIN` *password*, `STATUS`, `PASSWORD` *newPassword*, `LOGOUT`, and `SHUTDOWN`. The `STATUS` command should display the total number of clients that have logged in since the server started.

■■ **E23.4** Modify the `BankServer` program to provide complete error checking. For example, the program should check to make sure that there is enough money in the account when withdrawing. Send appropriate error reports back to the client. Enhance the protocol to be similar to HTTP, in which each server response starts with a number indicating the success or failure condition, followed by a string with response data or an error description.

■■ **E23.5** Write a program to display the protocol, host, port, and file components of a URL. *Hint:* Look at the API documentation of the `URL` class.

## PROGRAMMING PROJECTS

■■ **P23.1** Write a client application that executes an infinite loop that

**a.** Prompts the user for a number.

**b.** Sends that value to the server.

**c.** Receives a number from the server.

**d.** Displays the new number.

Also write a server that executes an infinite loop whose body accepts a client connection, reads a number from the client, computes its square root, and writes the result to the client.

■■ **P23.2** Implement a client-server program in which the client will print the date and time given by the server. Two classes should be implemented: `DateClient` and `DateServer`. The `DateServer` simply prints `new Date().toString()` whenever it accepts a connection and then closes the socket.

■■■ **P23.3** Write a simple web server that recognizes only the `GET` request (without the `Host:` request parameter and blank line). When a client connects to your server and sends a command, such as `GET` *filename* `HTTP/1.1`, then return a header

```
HTTP/1.1 200 OK
```

followed by a blank line and all lines in the file. If the file doesn't exist, return `404 Not Found` instead.

Your server should listen to port 8080. Test your web server by starting up your web browser and loading a page, such as `localhost:8080/c:\cs1\myfile.html`.

■■■ **P23.4** Write a chat server and client program. The chat server accepts connections from clients. Whenever one of the clients sends a chat message, it is displayed for all other clients to see. Use a protocol with three commands: `LOGIN` *name,* `CHAT` *message,* and `LOGOUT`.

■■ **P23.5** A query such as

```
http://aa.usno.navy.mil/cgi-bin/aa_moonphases.pl?year=2011
```

returns a page containing the moon phases in a given year. Write a program that asks the user for a year, month, and day and then prints the phase of the moon on that day.

■■■ **P23.6** A page such as

```
http://www.nws.noaa.gov/view/states.php
```

contains links to pages showing the weather reports for many cities in the fifty states. Write a program that asks the user for a state and city and then prints the weather report.

■■■ **P23.7** A page such as

```
https://www.cia.gov/library/publications/the-world-factbook/geos/
   countrytemplate_ca.html
```

contains information about a country (here Canada, with the symbol ca—see `https://www.cia.gov/library/publications/the-world-factbook/print/textversion.html` for the country symbols). Write a program that asks the user for a country name and then prints the area and population.

## ANSWERS TO SELF-CHECK QUESTIONS

1. An IP address is a numerical address, consisting of four or sixteen bytes. A domain name is an alphanumeric string that is associated with an IP address.
2. TCP is reliable but somewhat slow. When sending sounds or images in real time, it is acceptable if a small amount of the data is lost. But there is no point in transmitting data that is late.
3. The browser software translates your requests (typed URLs and mouse clicks on links) into HTTP commands that it sends to the appropriate web servers.
4. Some Telnet implementations send all keystrokes that you type to the server, including the backspace key. The server does not recognize a character sequence such as G W Backspace E T as a valid command.
5. The program makes a connection to the server, sends the `GET` request, and prints the error message that the server returns.
6. `Socket s = new Socket("e-mail.sjsu.edu", 110);`
7. Port 80 is the standard port for HTTP. If a web server is running on the same computer, then one can't open a server socket on an open port.
8. No, a server socket just waits for a connection and yields a regular `Socket` object when a client has connected. You use that socket object to read the data that the client sends.
9. The `URLConnection` class understands the HTTP protocol, freeing you from assembling requests and analyzing response headers.
10. The bytes that encode the images are displayed on the console, but they will appear to be random gibberish.