Section 11.4

Networking

AS FAR AS A PROGRAM IS CONCERNED, a network is just another possible source of input data, and another place where data can be output. That does oversimplify things, because networks are not as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called `java.net`. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World-Wide Web, and provides the sort of network communication capability that is used by a Web browser when it downloads pages for you to view. The main classes for this style of networking are `java.net.URL` and `java.net.URLConnection`. An object of type *URL* is an abstract representation of a Universal Resource Locator, which is an address for an HTML document or other resource on the Web. A *URLConnection* represents a network connection to such a resource.

The second style of I/O, which is more general and much more important, views the network at a lower level. It is based on the idea of a socket. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called `java.net.Socket` to represent sockets that are used for network communication. The term "socket" presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class *Socket*. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams.

11.4.1 URLs and URLConnections

The *URL* class is used to represent resources on the World-Wide Web. Every resource has an address, which identifies it uniquely and contains enough information for a Web browser to find the resource on the network and retrieve it. The address is called a "url" or "universal resource locator."

An object belonging to the *URL* class represents such an address. Once you have a *URL* object, you can use it to open a *URLConnection* to the resource at that address. A url is ordinarily specified as a string, such as "`http://math.hws.edu/eck/index.html`". There are also relative url's. A relative url specifies the location of a resource relative to the location of another url, which is called the base or context for the relative url. For example, if the context is given by the url `http://math.hws.edu/eck/`, then the incomplete, relative url "`index.html`" would really refer to `http://math.hws.edu/eck/index.html`.

An object of the class *URL* is not simply a string, but it can be constructed from a string representation of a url. A *URL* object can also be constructed from another *URL* object, representing a context, and a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName) throws MalformedURLException
```

Note that these constructors will throw an exception of type *MalformedURLException* if the specified strings don't represent legal url's. The *MalformedURLException* class is a subclass of *IOException*, and it requires mandatory exception handling. That is, you must call the constructor inside a `try..catch` statement that handles the exception or in a subroutine that is declared to throw the exception.

The second constructor is especially convenient when writing applets. In an applet, two methods are available that provide useful URL contexts. The method `getDocumentBase()`, defined in the *Applet* and *JApplet* classes, returns an object of type *URL*. This *URL* represents the location from which the HTML page that contains the applet was downloaded. This allows the applet to go back and retrieve other files that are stored in the same location as that document. For example,

```
URL url = new URL(getDocumentBase(), "data.txt");
```

constructs a *URL* that refers to a file named `data.txt` on the same computer and in the same directory as the source file for the web page on which the applet is running. Another method, `getCodeBase()`, returns a *URL* that gives the location of the applet class file (which is not necessarily the same as the location of the document).

Once you have a valid *URL* object, you can call its `openConnection()` method to set up a connection. This method returns a *URLConnection*. The *URLConnection* object can, in turn, be used to create an *InputStream* for reading data from the resource represented by the URL. This is done by calling its `getInputStream()` method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The `openConnection()` and `getInputStream()` methods can both throw exceptions of type *IOException*. Once the *InputStream* has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as *TextReader*, or using a *Scanner*. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the *URLConnection* class is `getContentType()`, which returns a *String* that describes the type of information available from the URL. The return value can be `null` if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call `getContentType()` after `getInputStream()`. The string returned by `getContentType()` is in a format called a mime type. Mime types include "text/plain", "text/html", "image/jpeg", "image/gif", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "gif". If you are only interested in text data, for example, you can check whether the string returned by `getContentType()` starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and then copies the text onto the screen. Many of the operations in this subroutine can throw exceptions. They are handled by declaring that the subroutine "`throws IOException`" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws IOException {

   /* Open a connection to the URL, and get an input stream
      for reading data from the URL. */

   URL url = new URL(urlString);
   URLConnection connection = url.openConnection();
   InputStream urlData = connection.getInputStream();

   /* Check that the content is some type of text. */

   String contentType = connection.getContentType();
   if (contentType == null || contentType.startsWith("text") == false)
      throw new IOException("URL does not seem to refer to a text file.");

   /* Copy lines of text from the input stream to the screen, until
      end-of-file is encountered  (or an error occurs). */

   BufferedReader in;  // For reading from the connection's input stream.
   in = new BufferedReader( new InputStreamReader(urlData) );

   while (true) {
      String line = in.readLine();
      if (line == null)
         break;
      System.out.println(line);
   }
```

```
      } // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file *ReadURL.java*. When using the program, note that you have to specify a complete url, including the "`http://`" at the beginning. Here is an applet that does much the same thing. The applet lets you enter a URL, which can be either a complete URL or a relative URL. A relative URL will be interpreted relative to the document base of the applet. Error messages or text loaded from the URL will be displayed in the text area of the applet. (The amount of text is limited to 10000 characters.) When the applet starts up, it is configured to load the file ReadURL.java from this book's source code directory; just click the "Load" button:

You can also try to use this applet to look at the HTML source code for this very page. Just type `s4.html` into the input box at the bottom of the applet and then click on the Load button. You might want to experiment with other urls to see what types of errors can occur. For example, entering "`bogus.html`" is likely to generate a *FileNotFoundException*, since no document of that name exists in the directory that contains this page. As another example, you can probably generate a *SecurityException* by trying to connect to *http://www.whitehouse.gov*. (Not because it's an official secret -- any url that does not lead back to the same computer from which the applet was loaded will generate a security exception. To protect you from malicious applets, an applet is allowed to open network connections only back to the computer from which it came.) The source code for the applet is in the file *ReadURLApplet.java*.

11.4.2 TCP/IP and Client/Server

Communication over the Internet is based on a pair of protocols called the Transmission Control Protocol and the Internet Protocol, which are collectively referred to as TCP/IP. (In fact, there is a more basic communication protocol called UDP that can be used instead of TCP in certain applications. UDP is supported in Java, but for this discussion, I'll stick to the full TCP/IP, which provides reliable two-way communication between networked computers.)

For two programs to communicate using TCP/IP, each program must create a socket, as discussed earlier in this section, and those sockets must be connected. Once such a connection is made, communication takes place using input streams and output streams. Each program has its own input stream and its own output stream. Data written by one program to its output stream is transmitted to the other computer. There, it enters the input stream of the program at the other end of the network connection. When that program reads data from its input stream, it is receiving the data that was transmitted to it over the network.

The hard part, then, is making a network connection in the first place. Two sockets are involved. To get things started, one program must create a socket that will wait passively until a connection request comes in from another socket. The waiting socket is said to be listening for a connection. On the other side of the connection-to-be, another program creates a socket that sends out a connection request to the listening socket. When the listening socket receives the connection request, it responds, and the connection is established. Once that is done, each program can obtain an input stream and an output stream for sending data over the connection. Communication takes place through these streams until one program or the other closes the connection.

A program that creates a listening socket is sometimes said to be a server, and the socket is called a server socket. A program that connects to a server is called a client, and the socket that it uses to make a connection is called a client socket. The idea is that the server is out there somewhere on the network, waiting for a connection request from some client. The server can be thought of as offering some kind of service, and the client gets access to that service by connecting to the server. This is called the client/server model of network communication. In many actual applications, a server program can provide connections to several clients at the same time. When a client connects to a server's listening socket, that socket does not stop listening. Instead, it continues listening for additional client connections at the same time that the first client is being serviced. To do this, it is necessary to use threads (Section 8.5). We'll look at how it works in the next section.

The *URL* class that was discussed at the beginning of this section uses a client socket behind the scenes to do any necessary network communication. On the other side of that connection is a server program that accepts a connection request from the *URL* object, reads a request from that object for some particular file on the server computer, and responds by transmitting the contents of that file over the network back to the *URL* object. After transmitting the data, the server closes the connection.

A client program has to have some way to specify which computer, among all those on the network, it wants to communicate with. Every computer on the Internet has an IP address which identifies it uniquely among all the computers on the net. Many computers can also be referred to by domain names such as math.hws.edu or www.whitehouse.gov. (See Section 1.7.) Traditional (or IPv4) IP addresses are 32-bit integers. They are usually written in the so-called "dotted decimal" form, such as `69.9.161.200`, where each of the four numbers in the address represents an 8-bit integer in the range 0 through 255. A new version of the Internet Protocol, IPv6, is currently being introduced. IPv6 addresses are 128-bit integers and are usually written in hexadecimal form (with some colons and maybe some extra information thrown in). In actual use, IPv6 addresses are still fairly rare.

A computer can have several IP addresses, and can have both IPv4 and IPv6 addresses. Usually, one of these is the loopback address, which can be used when a program wants to communicate with another program *on the same computer*. The loopback address has IPv4 address 127.0.0.1 and can also, in general, be referred to using the domain name localhost. In addition, there can be one or more IP addresses associated with physical network connections. Your computer probably has some utility for displaying your computer's IP addresses. I have written a small Java program, *ShowMyNetwork.java*, that does the same thing. When I run `ShowMyNetwork` on my computer, the output is:

```
en1 :  /192.168.1.47  /fe80:0:0:0:211:24ff:fe9c:5271%5
lo0 :  /127.0.0.1  /fe80:0:0:0:0:0:0:1%1  /0:0:0:0:0:0:0:1%0
```

The first thing on each line is a network interface name, which is really meaningful only to the computer's operating system. The output also contains the IP addresses for that interface. In this example, `lo0` refers to the loopback address, which has IPv4 address 127.0.0.1 as usual. The most important number here is `192.168.1.47`, which is the IPv4 address that can be used for communication over the network.

Now, a single computer might have several programs doing network communication at the same time, or one program communicating with several other computers. To allow for this possibility, a network connection is actually identified by a port number in combination with an IP address. A port number is just a 16-bit integer. A server does not simply listen for connections -- it listens for connections *on a particular port*. A potential client must know both the Internet address (or domain name) of the computer on which the server is running and the port number on which the server is listening. A Web server, for example, generally listens for connections on port 80; other standard Internet services also have standard port numbers. (The standard port numbers are all less than 1024, and are reserved for particular services. If you create your own server programs, you should use port numbers greater than 1024.)

11.4.3 Sockets

To implement TCP/IP connections, the `java.net` package provides two classes, *ServerSocket* and *Socket*. A *ServerSocket* represents a listening socket that waits for connection requests from clients. A *Socket* represents one endpoint of an actual network connection. A *Socket* can be a client socket that sends a connection request to a server. But a *Socket* can also be created by a server to handle a connection request from a client. This allows the server to create multiple sockets and handle multiple connections. A *ServerSocket* does not itself participate in connections; it just listens for connection requests and creates *Sockets* to handle the actual connections.

When you construct a `ServerSocket` object, you have to specify the port number on which the server will listen. The specification for the constructor is

```
public ServerSocket(int port) throws IOException
```

The port number must be in the range 0 through 65535, and should generally be greater than 1024. (A value of 0 tells the server socket to listen on any available port.) The constructor might throw a *SecurityException* if a smaller port number is specified. An *IOException* can occur if, for example, the specified port number is already in use.

As soon as a *ServerSocket* is created, it starts listening for connection requests. The `accept()` method in the *ServerSocket* class accepts such a request, establishes a connection with the client, and returns a *Socket* that can be used for communication with the client. The `accept()` method has the form

```
public Socket accept() throws IOException
```

When you call the `accept()` method, it will not return until a connection request is received (or until some error occurs). The method is said to block while waiting for the connection. (While the method is blocked, the thread that called the method can't do anything else. However, other threads in the same program can proceed.) You can call `accept()` repeatedly to accept multiple connection requests. The *ServerSocket* will continue listening for connections until it is closed, using its `close()` method, or until some error occurs, or until the program is terminated in some way.

Suppose that you want a server to listen on port 1728, and suppose that you've written a method `provideService(Socket)` to handle the communication with one client. Then the basic form of the server program would be:

```
try {
   ServerSocket server = new ServerSocket(1728);
   while (true) {
      Socket connection = server.accept();
      provideService(connection);
   }
}
catch (IOException e) {
   System.out.println("Server shut down with error: " + e);
}
```

On the client side, a client socket is created using a constructor in the *Socket* class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs.

Once you have a connected socket, no matter how it was created, you can use the *Socket* methods `getInputStream()` and `getOutputStream()` to obtain streams that can be used for communication over the connection. These methods return objects of type *InputStream* and *OutputStream*, respectively. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
/**
 * Open a client connection to a specified server computer and
 * port number on the server, and then do communication through
 * the connection.
 */
void doClientConnection(String computerName, int serverPort) {
   Socket connection;
   InputStream in;
   OutputStream out;
   try {
      connection = new Socket(computerName,serverPort);
      in = connection.getInputStream();
      out = connection.getOutputStream();
   }
   catch (IOException e) {
      System.out.println(
          "Attempt to create connection failed with error: " + e);
      return;
   }
    .
    .  // Use the streams, in and out, to communicate with server.
    .
   try {
```

```
            connection.close();
                // (Alternatively, you might depend on the server
                //  to close the connection.)
         }
         catch (IOException e) {
         }
      }  // end doClientConnection()
```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail here. However, what I've covered here should give you the basic ideas of network programming, and it is enough to write some simple network applications. Let's look at a few working examples of client/server programming.

11.4.4 A Trivial Client/Server

The first example consists of two programs. The source code files for the programs are *DateClient.java* and *DateServer.java*. One is a simple network client and the other is a matching server. The client makes a connection to the server, reads one line of text from the server, and displays that text on the screen. The text sent by the server consists of the current date and time on the computer where the server is running. In order to open a connection, the client must know the computer on which the server is running and the port on which it is listening. The server listens on port number 32007. The port number could be anything between 1025 and 65535, as long the server and the client use the same port. Port numbers between 1 and 1024 are reserved for standard services and should not be used for other servers. The name or IP number of the computer on which the server is running must be specified as a command-line argument For example, if the server is running on a computer named math.hws.edu, then you would typically run the client with the command "`java DateClient math.hws.edu`". Here is the complete client program:

```
import java.net.*;
import java.io.*;

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  The connection is made to
 * the port specified by LISTENING_PORT.  The program reads one
 * line of text from the connection and then closes the
 * connection.  It displays the text that it read on
 * standard output.  This program is meant to be used with
 * the server program, DateServer, which sends the current
 * date and time on the computer where the server is running.
 */
public class DateClient {

   public static final int LISTENING_PORT = 32007;
```

```
      public static void main(String[] args) {

         String hostName;         // Name of the server computer to connect to.
         Socket connection;       // A socket for communicating with server.
         BufferedReader incoming; // For reading data from the connection.

         /* Get computer name from command line. */

         if (args.length > 0)
            hostName = args[0];
         else {
               // No computer name was given.  Print a message and exit.
            System.out.println("Usage:  java DateClient <server_host_name>");
            return;
         }

         /* Make the connection, then read and display a line of text. */

         try {
            connection = new Socket( hostName, LISTENING_PORT );
            incoming = new BufferedReader( 
                             new InputStreamReader(connection.getInputStream()) );
            String lineFromServer = incoming.readLine();
            if (lineFromServer == null) {
                  // A null from incoming.readLine() indicates that
                  // end-of-stream was encountered.
               throw new IOException("Connection was opened, " + 
                     "but server did not send any data.");
            }
            System.out.println();
            System.out.println(lineFromServer);
            System.out.println();
            incoming.close();
         }
         catch (Exception e) {
            System.out.println("Error:  " + e);
         }

      }  // end main()


   } //end class DateClient
```

Note that all the communication with the server is done in a `try..catch` statement. This will catch the *IOExceptions* that can be generated when the connection is opened or closed and when data is read from the input stream. The connection's input stream is wrapped in a *BufferedReader*, which has a `readLine()` method that makes it easy to read one line of text. (See Subsection 11.1.4.)

In order for this program to run without error, the server program must be running on the computer to which the client tries to connect. By the way, it's possible to run the client and the server program on the same computer. For example, you can open two command windows, start the server in one window and then run the client in the other window. To make things like this easier, most computers will recognize the domain name `localhost` and the IP number `127.0.0.1` as referring to "this computer." This means that the command "`java DateClient localhost`" will tell the *DateClient* program to connect to a server running on the same computer. If that command doesn't work, try "`java DateClient 127.0.0.1`".

The server program that corresponds to the *DateClient* client program is called *DateServer*. The *DateServer* program creates a *ServerSocket* to listen for connection requests on port 32007. After the listening socket is created, the server will enter an infinite loop in which it accepts and processes connections. This will continue until the program is killed in some way -- for example by typing a `CONTROL-C` in the command window where the server is running. When a connection is received from a client, the server calls a subroutine to handle the connection. In the subroutine, any *Exception* that occurs is caught, so that it will not crash the server. Just because a connection to one client has failed for some reason, it does not mean that the server should be shut down; the error might have been the fault of the client. The connection-handling subroutine creates a *PrintWriter* for sending data over the connection. It writes the current date and time to this stream and then closes the connection. (The standard class `java.util.Date` is used to obtain the current time. An object of type *Date* represents a particular date and time. The default constructor, "`new Date()`", creates an object that represents the time when the object is created.) The complete server program is as follows:

```
import java.net.*;
import java.io.*;
import java.util.Date;

/**
 * This program is a server that takes connection requests on
 * the port specified by the constant LISTENING_PORT.  When a
 * connection is opened, the program sends the current time to
 * the connected socket.  The program will continue to receive
 * and process connections until it is killed (by a CONTROL-C,
 * for example).  Note that this server processes each connection
 * as it is received, rather than creating a separate thread
 * to process the connection.
 */
public class DateServer {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      ServerSocket listener;  // Listens for incoming connections.
      Socket connection;      // For communication with the connecting program.

      /* Accept and process connections forever, or until some error occurs.
         (Note that errors that occur while communicating with a connected
         program are caught and handled in the sendDate() routine, so
         they will not crash the server.) */
      try {
         listener = new ServerSocket(LISTENING_PORT);
         System.out.println("Listening on port " + LISTENING_PORT);
         while (true) {
                // Accept next connection request and handle it.
            connection = listener.accept();
            sendDate(connection);
         }
      }
      catch (Exception e) {
         System.out.println("Sorry, the server has shut down.");
         System.out.println("Error:  " + e);
         return;
      }

   }  // end main()


   /**
    * The parameter, client, is a socket that is already connected to another
    * program.  Get an output stream for the connection, send the current time,
    * and close the connection.
    */
   private static void sendDate(Socket client) {
      try {
         System.out.println("Connection from " +
                                     client.getInetAddress().toString() );
         Date now = new Date();  // The current date and time.
         PrintWriter outgoing;   // Stream for sending data.
         outgoing = new PrintWriter( client.getOutputStream() );
         outgoing.println( now.toString() );
         outgoing.flush();  // Make sure the data is actually sent!
         client.close();
      }
      catch (Exception e){
         System.out.println("Error: " + e);
      }
   } // end sendDate()


} //end class DateServer
```

When you run *DateServer* in a command-line interface, it will sit and wait for connection requests and report them as they are received. To make the *DateServer* service permanently available on a computer, the program really should be run as a daemon. A daemon is a program that runs continually on a computer, independently of any user. The computer can be configured to start the daemon automatically as soon as the computer boots up. It then runs in the background, even while the computer is being used for other purposes. For example, a computer that makes pages available on the World Wide Web runs a daemon that listens for requests for pages and responds by transmitting the pages. It's just a souped-up analog of the *DateServer* program! However, the question of how to set up a program as a daemon is not one I want to go into here. For testing purposes, it's easy enough to start the program by hand, and, in any case, my examples are not really robust enough or full-featured enough to be run as serious servers. (By the way, the word "daemon" is just an alternative spelling of "demon" and is usually pronounced the same way.)

Note that after calling `out.println()` to send a line of data to the client, the server program calls `out.flush()`. The `flush()` method is available in every output stream class. Calling it ensures that data that has been written to the stream is actually sent to its destination. You should generally call this function every time you use an output stream to send data over a network connection. If you don't do so, it's possible that the stream will collect data until it has a large batch of data to send. This is done for efficiency, but it can impose unacceptable delays when the client is waiting for the transmission. It is even possible that some of the data might remain untransmitted when the socket is closed, so it is especially important to call `flush()` before closing the connection. This is one of those unfortunate cases where different implementations of Java can behave differently. If you fail to flush your output streams, it is possible that your network application will work on some types of computers but not on others.

11.4.5 A Simple Network Chat

In the *DateServer* example, the server transmits information and the client reads it. It's also possible to have two-way communication between client and server. As a first example, we'll look at a client and server that allow a user on each end of the connection to send messages to the other user. The program works in a command-line interface where the users type in their messages. In this example, the server waits for a connection from a single client and then closes down its listener so that no other clients can connect. After the client and server are connected, both ends of the connection work in much the same way. The user on the client end types a message, and it is transmitted to the server, which displays it to the user on that end. Then the user of the server types a message that is transmitted to the client. Then the client user types another message, and so on. This continues until one user or the other enters "quit" when prompted for a message. When that happens, the connection is closed and both programs terminate. The client program and the server program are very similar. The techniques for opening the connections differ, and the client is programmed to send the first message while the server is programmed to receive the first message. The client and server programs can be found in the files *CLChatClient.java* and *CLChatServer.java*. (The name "CLChat" stands for "command-line chat.") Here is the source code for the server:

```
import java.net.*;
import java.io.*;

/**
 * This program is one end of a simple command-line interface chat program.
 * It acts as a server which waits for a connection from the CLChatClient
 * program.  The port on which the server listens can be specified as a
 * command-line argument.  If it is not, then the port specified by the
 * constant DEFAULT_PORT is used.  Note that if a port number of zero is
 * specified, then the server will listen on any available port.
 * This program only supports one connection.  As soon as a connection is
 * opened, the listening socket is closed down.  The two ends of the connection
 * each send a HANDSHAKE string to the other, so that both ends can verify
 * that the program on the other end is of the right type.  Then the connected
 * programs alternate sending messages to each other.  The client always sends
 * the first message.  The user on either end can close the connection by
 * entering the string "quit" when prompted for a message.  Note that the first
 * character of any string sent over the connection must be 0 or 1; this
 * character is interpreted as a command.
 */
public class CLChatServer {

   /**
    * Port to listen on, if none is specified on the command line.
    */
   static final int DEFAULT_PORT = 1728;

   /**
    * Handshake string. Each end of the connection sends this  string to the
    * other just after the connection is opened.  This is done to confirm that
    * the program on the other side of the connection is a CLChat program.
    */
   static final String HANDSHAKE = "CLChat";

   /**
    * This character is prepended to every message that is sent.
    */
   static final char MESSAGE = '0';

   /**
    * This character is sent to the connected program when the user quits.
    */
   static final char CLOSE = '1';


   public static void main(String[] args) {

      int port;   // The port on which the server listens.

      ServerSocket listener;  // Listens for a connection request.
      Socket connection;      // For communication with the client.

      BufferedReader incoming;  // Stream for receiving data from client.
      PrintWriter outgoing;     // Stream for sending data to client.
      String messageOut;        // A message to be sent to the client.
      String messageIn;         // A message received from the client.

      BufferedReader userInput; // A wrapper for System.in, for reading
                                // lines of input from the user.
```

```
/* First, get the port number from the command line,
   or use the default port if none is specified. */

if (args.length == 0)
   port = DEFAULT_PORT;
else {
   try {
      port= Integer.parseInt(args[0]);
      if (port < 0 || port > 65535)
         throw new NumberFormatException();
   }
   catch (NumberFormatException e) {
      System.out.println("Illegal port number, " + args[0]);
      return;
   }
}

/* Wait for a connection request.  When it arrives, close
   down the listener.  Create streams for communication
   and exchange the handshake. */

try {
   listener = new ServerSocket(port);
   System.out.println("Listening on port " + listener.getLocalPort());
   connection = listener.accept();
   listener.close();
   incoming = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()) );
   outgoing = new PrintWriter(connection.getOutputStream());
   outgoing.println(HANDSHAKE);  // Send handshake to client.
   outgoing.flush();
   messageIn = incoming.readLine();  // Receive handshake from client.
   if (! HANDSHAKE.equals(messageIn) ) {
      throw new Exception("Connected program is not a CLChat!");
   }
   System.out.println("Connected.  Waiting for the first message.");
}
catch (Exception e) {
   System.out.println("An error occurred while opening connection.");
   System.out.println(e.toString());
   return;
}

/* Exchange messages with the other end of the connection until one side
   or the other closes the connection.  This server program waits for
   the first message from the client.  After that, messages alternate
```

```
            strictly back and forth. */

         try {
            userInput = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("NOTE: Enter 'quit' to end the program.\n");
            while (true) {
               System.out.println("WAITING...");
               messageIn = incoming.readLine();
               if (messageIn.length() > 0) {
                      // The first character of the message is a command. If
                      // the command is CLOSE, then the connection is closed.
                      // Otherwise, remove the command character from the
                      // message and proceed.
                  if (messageIn.charAt(0) == CLOSE) {
                     System.out.println("Connection closed at other end.");
                     connection.close();
                     break;
                  }
                  messageIn = messageIn.substring(1);
               }
               System.out.println("RECEIVED:  " + messageIn);
               System.out.print("SEND:      ");
               messageOut = userInput.readLine();
               if (messageOut.equalsIgnoreCase("quit"))  {
                     // User wants to quit.  Inform the other side
                     // of the connection, then close the connection.
                  outgoing.println(CLOSE);
                  outgoing.flush();  // Make sure the data is sent!
                  connection.close();
                  System.out.println("Connection closed.");
                  break;
               }
               outgoing.println(MESSAGE + messageOut);
               outgoing.flush(); // Make sure the data is sent!
               if (outgoing.checkError()) {
                  throw new IOException("Error occurred while transmitting message.");
               }
            }
         }
         catch (Exception e) {
            System.out.println("Sorry, an error has occurred.  Connection lost.");
            System.out.println("Error:  " + e);
            System.exit(1);
         }

      }  // end main()
```

```
} //end class CLChatServer
```

This program is a little more robust than *DateServer*. For one thing, it uses a handshake to make sure that a client who is trying to connect is really a *CLChatClient* program. A handshake is simply information sent between client and server as part of setting up the connection, before any actual data is sent. In this case, each side of the connection sends a string to the other side to identify itself. The handshake is part of the protocol that I made up for communication between *CLChatClient* and *CLChatServer*. A protocol is a detailed specification of what data and messages can be exchanged over a connection, how they must be represented, and what order they can be sent in. When you design a client/server application, the design of the protocol is an important consideration. Another aspect of the `CLChat` protocol is that after the handshake, every line of text that is sent over the connection begins with a character that acts as a command. If the character is 0, the rest of the line is a message from one user to the other. If the character is 1, the line indicates that a user has entered the "quit" command, and the connection is to be shut down.

Remember that if you want to try out this program on a single computer, you can use two command-line windows. In one, give the command "`java CLChatServer`" to start the server. Then, in the other, use the command "`java CLChatClient localhost`" to connect to the server that is running on the same machine.

[Previous Section | Next Section | Chapter Index | Main Index]