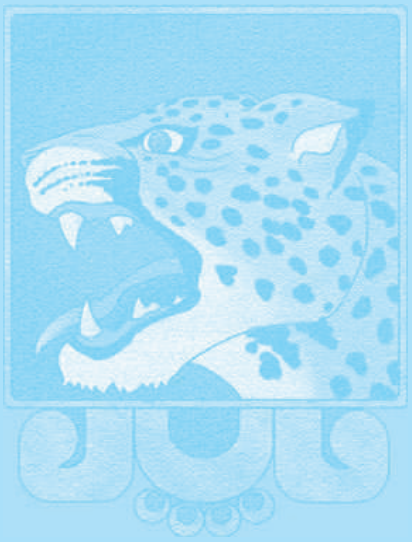
CHAPTER

33

# NETWORKING



## Objectives

- To explain terms: TCP, IP, domain name, domain name server, stream-based communications, and packet-based communications (§33.2).
- To create servers using server sockets (§33.2.1) and clients using client sockets (§33.2.2).
- To implement Java networking programs using stream sockets (§33.2.3).
- To develop an example of a client/server application (§33.2.4).
- To obtain Internet addresses using the `InetAddress` class (§33.3).
- To develop servers for multiple clients (§33.4).
- To develop applets that communicate with the server (§33.5).
- To send and receive objects on a network (§33.6).
- To develop an interactive tic-tac-toe game played on the Internet (§33.7).

## 33.1 Introduction


*Computer networking is to send and receive messages among computers on the Internet.*

To browse the Web or send email, your computer must be connected to the Internet. The *Internet* is the global network of millions of computers. Your computer can connect to the Internet through an Internet Service Provider (ISP) using a dialup, DSL, or cable modem, or through a local area network (LAN).

IP address

When a computer needs to communicate with another computer, it needs to know the other computer's address. An *Internet Protocol* (IP) address uniquely identifies the computer on the Internet. An IP address consists of four dotted decimal numbers between `0` and `255`, such as `130.254.204.33`. Since it is not easy to remember so many numbers, they are often mapped to meaningful names called *domain names*, such as liang.armstrong.edu. Special servers called *Domain Name Servers* (DNS) on the Internet translate host names into IP addresses. When a computer contacts liang.armstrong.edu, it first asks the DNS to translate this domain name into a numeric IP address and then sends the request using the IP address.

domain name

domain name server

TCP

The Internet Protocol is a low-level protocol for delivering data from one computer to another across the Internet in packets. Two higher-level protocols used in conjunction with the IP are the *Transmission Control Protocol* (TCP) and the *User Datagram Protocol* (UDP). TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. UDP is a standard, low-overhead, connectionless, host-to-host protocol that is used over the IP. UDP allows an application program on one computer to send a datagram to an application program on another computer.

stream-based

packet-based

Java supports both stream-based and packet-based communications. *Stream-based communications* use TCP for data transmission, whereas *packet-based communications* use UDP. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission. Stream-based communications are used in most areas of Java programming and are the focus of this chapter. Packet-based communications are introduced in Supplement III.U, Networking Using Datagram Protocol.

## 33.2 Client/Server Computing


*Java provides the* `ServerSocket` *class for creating a server socket and the* `Socket` *class for creating a client socket. Two programs on the Internet communicate through a server socket and a client socket using I/O streams.*

socket

Networking is tightly integrated in Java. The Java API provides the classes for creating sockets to facilitate program communications over the Internet. *Sockets* are the endpoints of logical connections between two hosts and can be used to send and receive data. Java treats socket communications much as it treats I/O operations; thus, programs can read from or write to sockets as easily as they can read from or write to files.

Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client and the server communicate through sockets.

The server must be running when a client attempts to connect to the server. The server waits for a connection request from a client. The statements needed to create sockets on a server and a client are shown in Figure 33.1.

### 33.2.1 Server Sockets

server socket

To establish a server, you need to create a *server socket* and attach it to a *port*, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers range from 0 to 65536, but port numbers 0 to 1024 are reserved for privileged services.

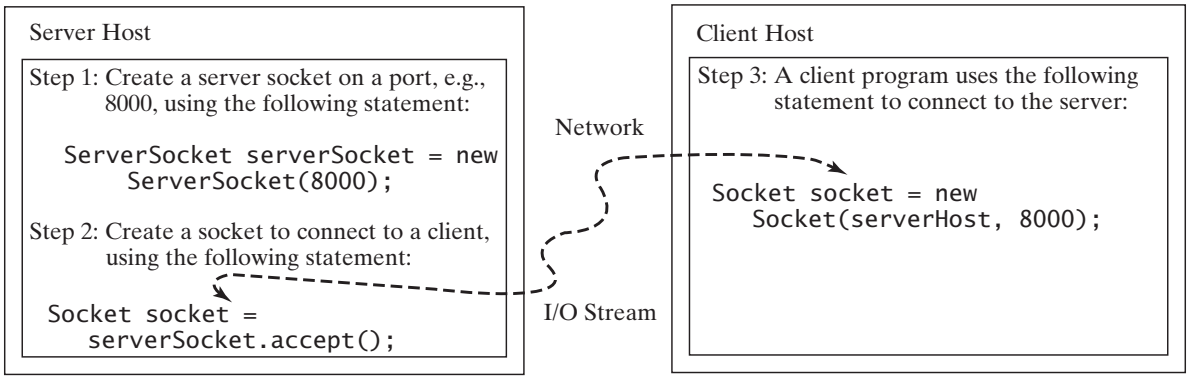



**FIGURE 33.1** The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

For instance, the email server runs on port 25, and the Web server usually runs on port 80. You can choose any port number that is not currently used by other programs. The following statement creates a server socket **serverSocket**:

```
ServerSocket serverSocket = new ServerSocket(port);
```

**Note**

Attempting to create a server socket on a port already in use would cause the **java.net.BindException**.

BindException

## 33.2.2 Client Sockets

After a server socket is created, the server can use the following statement to listen for connections:

```
Socket socket = serverSocket.accept();
```

This statement waits until a client connects to the server socket. The client issues the following statement to request a connection to a server:

connect to client

```
Socket socket = new Socket(serverName, port);
```

This statement opens a socket so that the client program can communicate with the server. ***serverName*** is the server's Internet host name or IP address. The following statement creates a socket on the client machine to connect to the host 130.254.204.33 at port 8000:

client socket
use IP address

```
Socket socket = new Socket("130.254.204.33", 8000)
```

Alternatively, you can use the domain name to create a socket, as follows:

use domain name

```
Socket socket = new Socket("liang.armstrong.edu", 8000);
```

When you create a socket with a host name, the JVM asks the DNS to translate the host name into the IP address.

**Note**

A program can use the host name **localhost** or the IP address **127.0.0.1** to refer to the machine on which a client is running.

localhost

UnknownHostException

**Note**

The `Socket` constructor throws a `java.net.UnknownHostException` if the host cannot be found.

### 33.2.3 Data Transmission through Sockets

After the server accepts the connection, communication between the server and client is conducted the same as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in Figure 33.2.

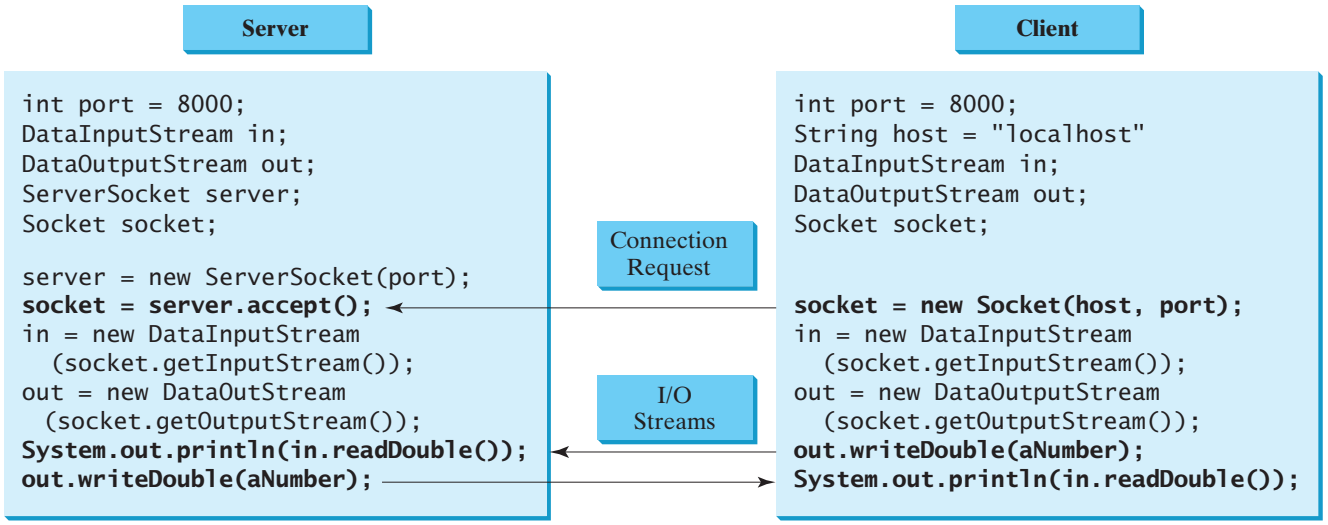



**FIGURE 33.2** The server and client exchange data through I/O streams on top of the socket.

To get an input stream and an output stream, use the `getInputStream()` and `getOutputStream()` methods on a socket object. For example, the following statements create an `InputStream` stream called `input` and an `OutputStream` stream called `output` from a socket:

```
InputStream input = socket.getInputStream();
OutputStream output = socket.getOutputStream();
```

The `InputStream` and `OutputStream` streams are used to read or write bytes. You can use `DataInputStream`, `DataOutputStream`, `BufferedReader`, and `PrintWriter` to wrap on the `InputStream` and `OutputStream` to read or write data, such as `int`, `double`, or `String`. The following statements, for instance, create the `DataInputStream` stream `input` and the `DataOutput` stream `output` to read and write primitive data values:

```
DataInputStream input = new DataInputStream
  (socket.getInputStream());
DataOutputStream output = new DataOutputStream
  (socket.getOutputStream());
```

The server can use `input.readDouble()` to receive a `double` value from the client, and `output.writeDouble(d)` to send the `double` value `d` to the client.

**Tip**

Recall that binary I/O is more efficient than text I/O because text I/O requires encoding and decoding. Therefore, it is better to use binary I/O for transmitting data between a server and a client to improve performance.

### 33.2.4 A Client/Server Example

This example presents a client program and a server program. The client sends data to a server. The server receives the data, uses it to produce a result, and then sends the result back to the client. The client displays the result on the console. In this example, the data sent from the client comprise the radius of a circle, and the result produced by the server is the area of the circle (see Figure 33.3).

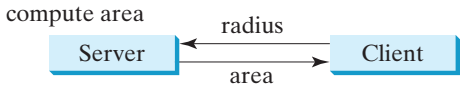



**FIGURE 33.3** The client sends the radius to the server; the server computes the area and sends it to the client.

The client sends the radius through a `DataOutputStream` on the output stream socket, and the server receives the radius through the `DataInputStream` on the input stream socket, as shown in Figure 33.4a. The server computes the area and sends it to the client through a `DataOutputStream` on the output stream socket, and the client receives the area through a `DataInputStream` on the input stream socket, as shown in Figure 33.4b. The server and client programs are given in Listings 33.1 and 33.2. Figure 33.5 contains a sample run of the server and the client.

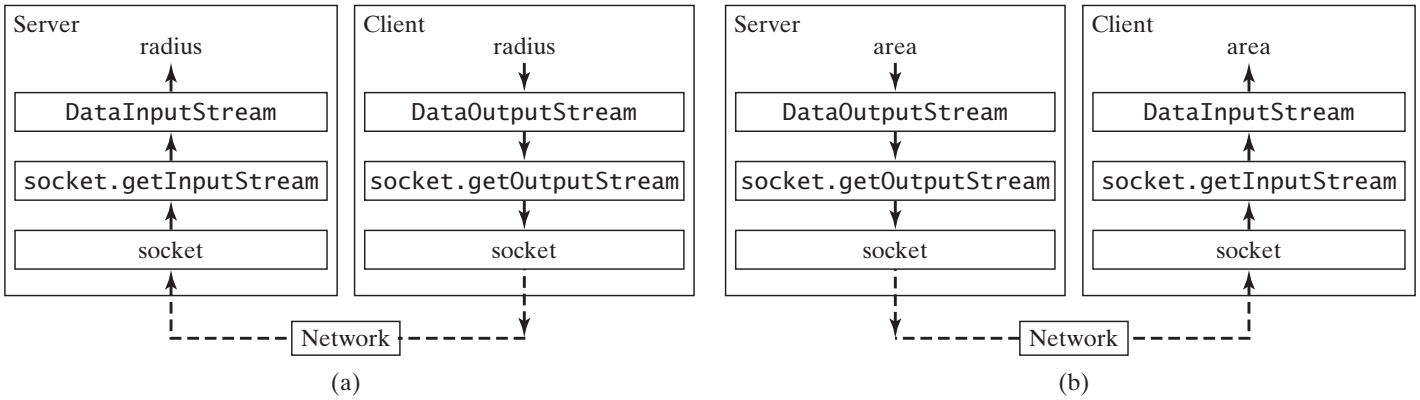



**FIGURE 33.4** (a) The client sends the radius to the server. (b) The server sends the area to the client.

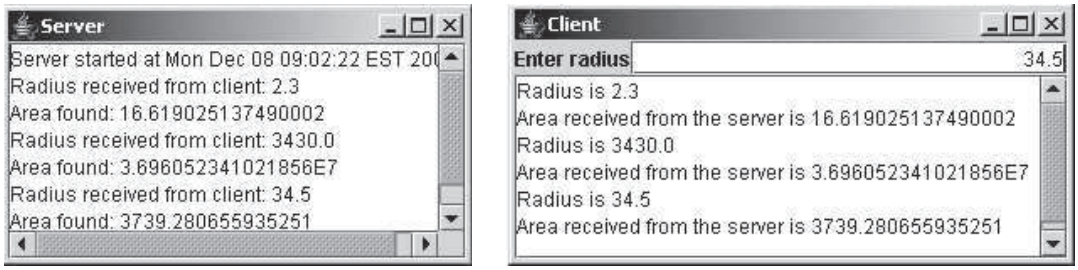



**FIGURE 33.5** The client sends the radius to the server. The server receives it, computes the area, and sends the area to the client.

## LISTING 33.1 Server.java

launch server

server socket

connect client

input from client

output to client

read radius

write area

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4  import java.awt.*;
5  import javax.swing.*;
6
7  public class Server extends JFrame {
8    // Text area for displaying contents
9    private JTextArea jta = new JTextArea();
10
11   public static void main(String[] args) {
12     new Server();
13   }
14
15   public Server() {
16     // Place text area on the frame
17     setLayout(new BorderLayout());
18     add(new JScrollPane(jta), BorderLayout.CENTER);
19
20     setTitle("Server");
21     setSize(500, 300);
22     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     setVisible(true); // It is necessary to show the frame here!
24
25     try {
26       // Create a server socket
27       ServerSocket serverSocket = new ServerSocket(8000);
28       jta.append("Server started at " + new Date() + '\n');
29
30       // Listen for a connection request
31       Socket socket = serverSocket.accept();
32
33       // Create data input and output streams
34       DataInputStream inputFromClient = new DataInputStream(
35         socket.getInputStream());
36       DataOutputStream outputToClient = new DataOutputStream(
37         socket.getOutputStream());
38
39       while (true) {
40         // Receive radius from the client
41         double radius = inputFromClient.readDouble();
42
43         // Compute area
44         double area = radius * radius * Math.PI;
45
46         // Send area back to the client
47         outputToClient.writeDouble(area);
48
49         jta.append("Radius received from client: " + radius + '\n');
50         jta.append("Area found: " + area + '\n');
51       }
52     }
53     catch(IOException ex) {
54       System.err.println(ex);
55     }
56   }
57 }
```

## LISTING 33.2 Client.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class Client extends JFrame {
8    // Text field for receiving radius
9    private JTextField jtf = new JTextField();
10
11   // Text area to display contents
12   private JTextArea jta = new JTextArea();
13
14   // IO streams
15   private DataOutputStream toServer;
16   private DataInputStream fromServer;
17
18   public static void main(String[] args) {
19     new Client();                                          launch client
20   }
21
22   public Client() {
23     // Panel p to hold the label and text field
24     JPanel p = new JPanel();
25     p.setLayout(new BorderLayout());
26     p.add(new JLabel("Enter radius"), BorderLayout.WEST);
27     p.add(jtf, BorderLayout.CENTER);
28     jtf.setHorizontalAlignment(JTextField.RIGHT);
29
30     setLayout(new BorderLayout());
31     add(p, BorderLayout.NORTH);
32     add(new JScrollPane(jta), BorderLayout.CENTER);
33
34     jtf.addActionListener(new TextFieldListener());        register listener
35
36     setTitle("Client");
37     setSize(500, 300);
38     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39     setVisible(true); // It is necessary to show the frame here!
40
41     try {
42       // Create a socket to connect to the server
43       Socket socket = new Socket("localhost", 8000);       request connection
44       // Socket socket = new Socket("130.254.204.33", 8000);
45       // Socket socket = new Socket("liang.armstrong.edu", 8000);
46
47       // Create an input stream to receive data from the server
48       fromServer = new DataInputStream(                    input from server
49         socket.getInputStream());
50
51       // Create an output stream to send data to the server
52       toServer =                                           output to server
53         new DataOutputStream(socket.getOutputStream());
54     }
55     catch (IOException ex) {
56       jta.append(ex.toString() + '\n');
57     }
```

```
58    }
59
60    private class TextFieldListener implements ActionListener {
61      @Override
62      public void actionPerformed(ActionEvent e) {
63        try {
64          // Get the radius from the text field
65          double radius = Double.parseDouble(jtf.getText().trim());
66
67          // Send the radius to the server
68          toServer.writeDouble(radius);
69          toServer.flush();
70
71          // Get area from the server
72          double area = fromServer.readDouble();
73
74          // Display to the text area
75          jta.append("Radius is " + radius + "\n");
76          jta.append("Area received from the server is "
77            + area + '\n');
78        }
79        catch (IOException ex) {
80          System.err.println(ex);
81        }
82      }
83    }
84  }
```

write radius

read radius

You start the server program first, then start the client program. In the client program, enter a radius in the text field and press *Enter* to send the radius to the server. The server computes the area and sends it back to the client. This process is repeated until one of the two programs terminates.

The networking classes are in the package **`java.net`**. You should import this package when writing Java network programs.

The **`Server`** class creates a **`ServerSocket serverSocket`** and attaches it to port 8000, using this statement (line 27 in Server.java):

```
ServerSocket serverSocket = new ServerSocket(8000);
```

The server then starts to listen for connection requests, using the following statement (line 31 in Server.java):

```
Socket socket = serverSocket.accept();
```

The server waits until a client requests a connection. After it is connected, the server reads the radius from the client through an input stream, computes the area, and sends the result to the client through an output stream.

The **`Client`** class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000 (line 43 in Client.java).

```
Socket socket = new Socket("localhost", 8000);
```

If you run the server and the client on different machines, replace **`localhost`** with the server machine's host name or IP address. In this example, the server and the client are running on the same machine.

If the server is not running, the client program terminates with a **`java.net.ConnectException`**. After it is connected, the client gets input and output streams—wrapped by data input and output streams—in order to receive and send data to the server.

If you receive a `java.net.BindException` when you start the server, the server port is currently in use. You need to terminate the process that is using the server port and then restart the server.

What happens if the `setVisible(true)` statement in line 23 in Server.java is moved after the `try-catch` block in line 56 in Server.java? The frame will not be displayed, because the `while` loop in the `try-catch` block will not finish until the program terminates.

**Note**

When you create a server socket, you have to specify a port (e.g., 8000) for the socket. When a client connects to the server (line 43 in Client.java), a socket is created on the client. This socket has its own local port. This port number (e.g., 2047) is automatically chosen by the JVM, as shown in Figure 33.6.

client socket port

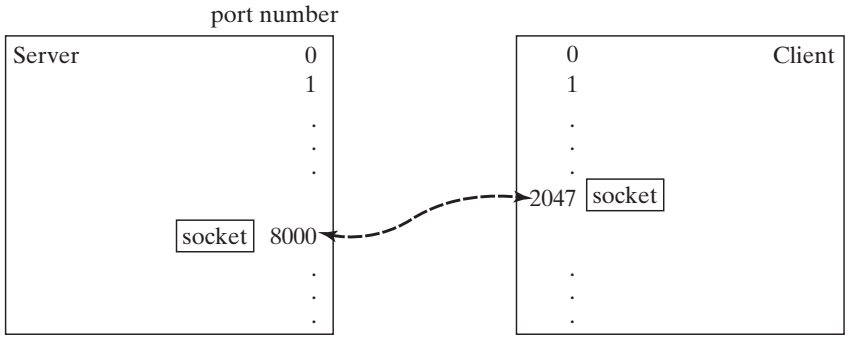



**FIGURE 33.6** The JVM automatically chooses an available port to create a socket for the client.

To see the local port on the client, insert the following statement in line 46 in Client.java.

```
System.out.println("local port: " + socket.getLocalPort());
```

**33.1** How do you create a server socket? What port numbers can be used? What happens if a requested port number is already in use? Can a port connect to multiple clients?

**33.2** What are the differences between a server socket and a client socket?

**33.3** How does a client program initiate a connection?

**33.4** How does a server accept a connection?

**33.5** How are data transferred between a client and a server?




## 33.3 The `InetAddress` Class

*The server program can use the `InetAddress` class to obtain the information about the IP address and host name for the client.*


Occasionally, you would like to know who is connecting to the server. You can use the `InetAddress` class to find the client's host name and IP address. The `InetAddress` class models an IP address. You can use the following statement in the server program to get an instance of `InetAddress` on a socket that connects to the client.

```
InetAddress inetAddress = socket.getInetAddress();
```

Next, you can display the client's host name and IP address, as follows:

```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
```

```
System.out.println("Client's IP Address is " +
  inetAddress.getHostAddress());
```

You can also create an instance of `InetAddress` from a host name or IP address using the static `getByName` method. For example, the following statement creates an `InetAddress` for the host `liang.armstrong.edu`.

```
InetAddress address = InetAddress.getByName("liang.armstrong.edu");
```

Listing 33.3 gives a program that identifies the host name and IP address of the arguments you pass in from the command line. Line 7 creates an `InetAddress` using the `getByName` method. Lines 8–9 use the `getHostName` and `getHostAddress` methods to get the host's name and IP address. Figure 33.7 shows a sample run of the program.


**FIGURE 33.7** The program identifies host names and IP addresses.

## LISTING 33.3 IdentifyHostNameIP.java

```
1  import java.net.*;
2
3  public class IdentifyHostNameIP {
4    public static void main(String[] args) {
5      for (int i = 0; i < args.length; i++) {
6        try {
7          InetAddress address = InetAddress.getByName(args[i]);
8          System.out.print("Host name: " + address.getHostName() + " ");
9          System.out.println("IP address: " + address.getHostAddress());
10       }
11       catch (UnknownHostException ex) {
12         System.err.println("Unknown host or IP address " + args[i]);
13       }
14     }
15   }
16 }
```

get an InetAddress
get host name
get host IP


MyProgrammingLab™

**33.6** How do you obtain an instance of `InetAddress`?

**33.7** What methods can you use to get the IP address and hostname from an `InetAddress`?

## 33.4 Serving Multiple Clients


*A server can serve multiple clients. The connection to each client is handled by one thread.*

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs continuously on a server computer, and clients from all over the Internet can connect to it. You can use threads to handle the server's multiple clients simultaneously—simply

create a thread for each connection. Here is how the server handles the establishment of a connection:

```
while (true) {
  Socket socket = serverSocket.accept(); // Connect to a client
  Thread thread = new ThreadClass(socket);
  thread.start();
}
```

The server socket can have many connections. Each iteration of the `while` loop creates a new connection. Whenever a connection is established, a new thread is created to handle communication between the server and the new client, and this allows multiple connections to run at the same time.

Listing 33.4 creates a server class that serves multiple clients simultaneously. For each connection, the server starts a new thread. This thread continuously receives input (the radius of a circle) from clients and sends the results (the area of the circle) back to them (see Figure 33.8). The client program is the same as in Listing 33.2. A sample run of the server with two clients is shown in Figure 33.9.

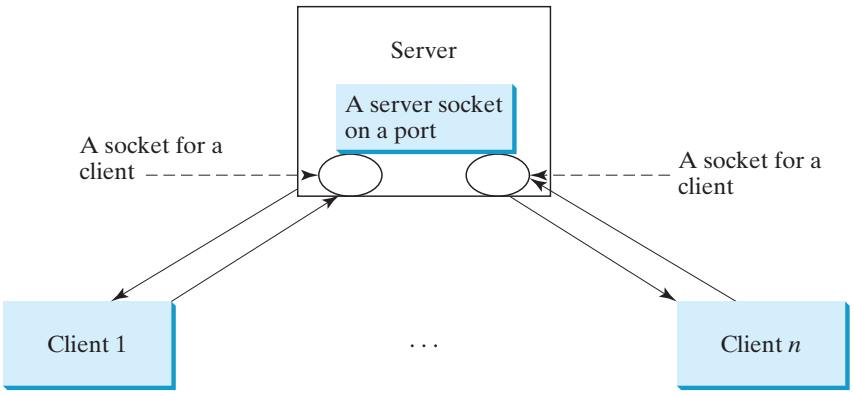



**FIGURE 33.8** Multithreading enables a server to handle multiple independent clients.

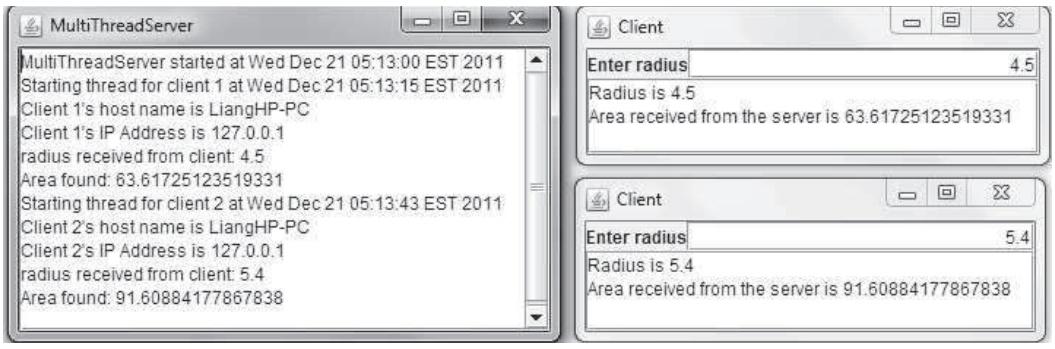



**FIGURE 33.9** The server spawns a thread in order to serve a client.

## LISTING 33.4 MultiThreadServer.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4  import java.awt.*;
5  import javax.swing.*;
```

```
 6
 7  public class MultiThreadServer extends JFrame {
 8    // Text area for displaying contents
 9    private JTextArea jta = new JTextArea();
10
11    public static void main(String[] args) {
12      new MultiThreadServer();
13    }
14
15    public MultiThreadServer() {
16      // Place text area on the frame
17      setLayout(new BorderLayout());
18      add(new JScrollPane(jta), BorderLayout.CENTER);
19
20      setTitle("MultiThreadServer");
21      setSize(500, 300);
22      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23      setVisible(true); // It is necessary to show the frame here!
24
25      try {
26        // Create a server socket
27        ServerSocket serverSocket = new ServerSocket(8000);
28        jta.append("MultiThreadServer started at " + new Date() + '\n');
29
30        // Number a client
31        int clientNo = 1;
32
33        while (true) {
34          // Listen for a new connection request
35          Socket socket = serverSocket.accept();
36
37          // Display the client number
38          jta.append("Starting thread for client " + clientNo +
39            " at " + new Date() + '\n');
40
41          // Find the client's host name and IP address
42          InetAddress inetAddress = socket.getInetAddress();
43          jta.append("Client " + clientNo + "'s host name is "
44            + inetAddress.getHostName() + "\n");
45          jta.append("Client " + clientNo + "'s IP Address is "
46            + inetAddress.getHostAddress() + "\n");
47
48          // Create a new thread for the connection
49          HandleAClient task = new HandleAClient(socket);
50
51          // Start the new thread
52          new Thread(task).start();
53
54          // Increment clientNo
55          clientNo++;
56        }
57      }
58      catch(IOException ex) {
59        System.err.println(ex);
60      }
61    }
62
63    // Inner class
64    // Define the thread class for handling new connection
65    class HandleAClient implements Runnable {
```

server socket

connect client

network information

create task

start thread

task class

```
66      private Socket socket; // A connected socket
67
68      /** Construct a thread */
69      public HandleAClient(Socket socket) {
70        this.socket = socket;
71      }
72
73      @Override /** Run a thread */
74      public void run() {
75        try {
76          // Create data input and output streams
77          DataInputStream inputFromClient = new DataInputStream(          I/O
78            socket.getInputStream());
79          DataOutputStream outputToClient = new DataOutputStream(
80            socket.getOutputStream());
81
82          // Continuously serve the client
83          while (true) {
84            // Receive radius from the client
85            double radius = inputFromClient.readDouble();
86
87            // Compute area
88            double area = radius * radius * Math.PI;
89
90            // Send area back to the client
91            outputToClient.writeDouble(area);
92
93            jta.append("radius received from client: " +
94              radius + '\n');
95            jta.append("Area found: " + area + '\n');
96          }
97        }
98        catch(IOException e) {
99          System.err.println(e);
100       }
101     }
102   }
103 }
```

The server creates a server socket at port 8000 (line 27) and waits for a connection (line 35). When a connection with a client is established, the server creates a new thread to handle the communication (line 49). It then waits for another connection in an infinite `while` loop (lines 33–56).

The threads, which run independently of one another, communicate with designated clients. Each thread creates data input and output streams that receive and send data to a client.

**33.8** How do you make a server serve multiple clients?


MyProgrammingLab™

## 33.5 Applet Clients


*The client can be an applet that connects to the server running on the host from which the applet is loaded.*

Because of security constraints, applets can connect only to the host from which they were loaded. Therefore, the HTML file must be located on the machine on which the server is running. You can obtain the server's host name by invoking `getCodeBase().getHost()` on an applet, so you can write the applet without the host name fixed. The following is an example of how to use an applet to connect to a server.

The applet shows the number of visits made to a Web page. The count should be stored in a file on the server side. Every time the page is visited or reloaded, the applet sends a request to the server, and the server increases the count and sends it to the applet. The applet then displays the new count in a message, such as **You are visitor number 11**, as shown in Figure 33.10. The server and client programs are given in Listings 33.5 and 33.6.

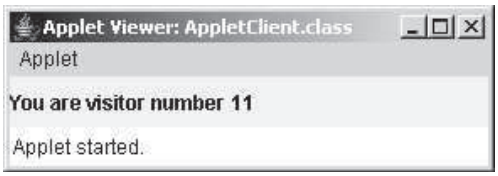



**FIGURE 33.10** The applet displays the access count on a Web page.

## LISTING 33.5 CountServer.java

```
1  import java.io.*;
2  import java.net.*;
3
4  public class CountServer {
5    private RandomAccessFile raf;
6    private int count; // Count the access to the server
7
8    public static void main(String[] args) {
9      new CountServer();
10   }
11
12   public CountServer() {
13     try {
14       // Create a server socket
15       ServerSocket serverSocket = new ServerSocket(8000);
16       System.out.println("Server started ");
17
18       // Create or open the count file
19       raf = new RandomAccessFile("count.dat", "rw");
20
21       // Get the count
22       if (raf.length() == 0)
23         count = 0;
24       else
25         count = raf.readInt();
26
27       while (true) {
28         // Listen for a new connection request
29         Socket socket = serverSocket.accept();
30
31         // Create a DataOutputStream for the socket
32         DataOutputStream outputToClient =
33           new DataOutputStream(socket.getOutputStream());
34
35         // Increase count and send the count to the client
36         count++;
37         outputToClient.writeInt(count);
38
39         // Write new count back to the file
40         raf.seek(0);
```

launch server

server socket

random access file

new file

get count

connect client

send to client

update count

```
41          raf.writeInt(count);
42        }
43      }
44      catch(IOException ex) {
45        ex.printStackTrace();
46      }
47    }
48  }
```

The server creates a `ServerSocket` in line 15 and creates or opens a file using `RandomAccessFile` in line 19. It reads the count from the file in lines 22–33. The server then waits for a connection request from a client (line 29). After a connection with a client is established, the server creates an output stream to the client (lines 32–33), increases the count (line 36), sends the count to the client (line 37), and writes the new count back to the file. This process continues in an infinite `while` loop to handle all clients.

## LISTING 33.6 AppletClient.java

```
1  import java.io.*;
2  import java.net.*;
3  import javax.swing.*;
4
5  public class AppletClient extends JApplet {
6    // Label for displaying the visit count
7    private JLabel jlblCount = new JLabel();
8
9    // Indicate if it runs as application
10   private boolean isStandAlone = false;
11
12   // Host name or IP address
13   private String host = "localhost";
14
15   /** Initialize the applet */
16   public void init() {
17     add(jlblCount);
18
19     try {
20       // Create a socket to connect to the server
21       Socket socket;
22       if (isStandAlone)
23         socket = new Socket(host, 8000);                    for standalone
24       else
25         socket = new Socket(getCodeBase().getHost(), 8000); for applet
26
27       // Create an input stream to receive data from the server
28       DataInputStream inputFromServer =
29         new DataInputStream(socket.getInputStream());
30
31       // Receive the count from the server and display it on label
32       int count = inputFromServer.readInt();              receive count
33       jlblCount.setText("You are visitor number " + count);
34
35       // Close the stream
36       inputFromServer.close();
37     }
38     catch (IOException ex) {
39       ex.printStackTrace();
40     }
41   }
42
```

```
43    /** Run the applet as an application */
44    public static void main(String[] args) {
45      // Create a frame
46      JFrame frame = new JFrame("Applet Client");
47
48      // Create an instance of the applet
49      AppletClient applet = new AppletClient();
50      applet.isStandAlone = true;
51
52      // Get host
53      if (args.length == 1) applet.host = args[0];
54
55      // Add the applet instance to the frame
56      frame.add(applet, java.awt.BorderLayout.CENTER);
57
58      // Invoke init() and start()
59      applet.init();
60      applet.start();
61
62      // Display the frame
63      frame.pack();
64      frame.setVisible(true);
65    }
66  }
```

The client is an applet. When it runs as an applet, it uses `getCodeBase().getHost()` (line 25) to return the IP address for the server. When it runs as an application, it passes the URL from the command line (line 53). If the URL is not passed from the command line, by default `localhost` is used for the URL (line 13).

The client creates a socket to connect to the server (lines 21–25), creates an input stream from the socket (lines 28–29), receives the count from the server (line 32), and displays it in the text field (line 33).

## 33.6 Sending and Receiving Objects


*A program can send and receive objects from another program.*

In the preceding examples, you learned how to send and receive data of primitive types. You can also send and receive objects using `ObjectOutputStream` and `ObjectInputStream` on socket streams. To enable passing, the objects must be serializable. The following example demonstrates how to send and receive objects.

The example consists of three classes: StudentAddress.java (Listing 33.7), StudentClient.java (Listing 33.8), and StudentServer.java (Listing 33.9). The client program collects student information from a client and sends it to a server, as shown in Figure 33.11.


**FIGURE 33.11** The client sends the student information in an object to the server.

The `StudentAddress` class contains the student information: name, street, city, state, and zip. The `StudentAddress` class implements the `Serializable` interface. Therefore, a `StudentAddress` object can be sent and received using the object output and input streams.

## LISTING 33.7 StudentAddress.java

```
1  public class StudentAddress implements java.io.Serializable {
2    private String name;
3    private String street;
4    private String city;
5    private String state;
6    private String zip;
7
8    public StudentAddress(String name, String street, String city,
9      String state, String zip) {
10     this.name = name;
11     this.street = street;
12     this.city = city;
13     this.state = state;
14     this.zip = zip;
15   }
16
17   public String getName() {
18     return name;
19   }
20
21   public String getStreet() {
22     return street;
23   }
24
25   public String getCity() {
26     return city;
27   }
28
29   public String getState() {
30     return state;
31   }
32
33   public String getZip() {
34     return zip;
35   }
36 }
```

serialized

The client sends a `StudentAddress` object through an `ObjectOutputStream` on the output stream socket, and the server receives the `Student` object through the `ObjectInputStream` on the input stream socket, as shown in Figure 33.12. The client uses

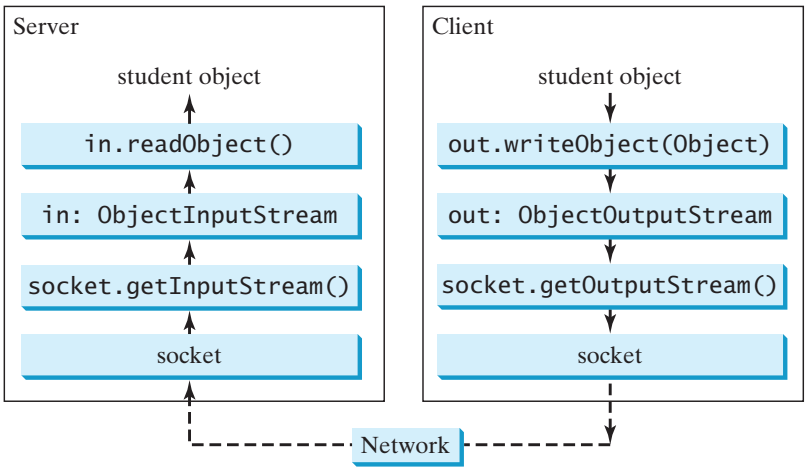



**FIGURE 33.12** The client sends a `StudentAddress` object to the server.

the `writeObject` method in the `ObjectOutputStream` class to send data about a student to the server, and the server receives the student's information using the `readObject` method in the `ObjectInputStream` class. The server and client programs are given in Listings 33.8 and 33.9.

## LISTING 33.8 StudentClient.java

```
1  import java.io.*;
2  import java.net.*;
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.border.*;
7
8  public class StudentClient extends JApplet {
9    private JTextField jtfName = new JTextField(32);
10   private JTextField jtfStreet = new JTextField(32);
11   private JTextField jtfCity = new JTextField(20);
12   private JTextField jtfState = new JTextField(2);
13   private JTextField jtfZip = new JTextField(5);
14
15   // Button for sending a student's address to the server
16   private JButton jbtRegister = new JButton("Register to the Server");
17
18   // Indicate if it runs as application
19   private boolean isStandAlone = false;
20
21   // Host name or IP address
22   String host = "localhost";
23
24   public void init() {
25     // Panel p1 for holding labels Name, Street, and City
26     JPanel p1 = new JPanel();
27     p1.setLayout(new GridLayout(3, 1));
28     p1.add(new JLabel("Name"));
29     p1.add(new JLabel("Street"));
30     p1.add(new JLabel("City"));
31
32     // Panel jpState for holding state
33     JPanel jpState = new JPanel();
34     jpState.setLayout(new BorderLayout());
35     jpState.add(new JLabel("State"), BorderLayout.WEST);
36     jpState.add(jtfState, BorderLayout.CENTER);
37
38     // Panel jpZip for holding zip
39     JPanel jpZip = new JPanel();
40     jpZip.setLayout(new BorderLayout());
41     jpZip.add(new JLabel("Zip"), BorderLayout.WEST);
42     jpZip.add(jtfZip, BorderLayout.CENTER);
43
44     // Panel p2 for holding jpState and jpZip
45     JPanel p2 = new JPanel();
46     p2.setLayout(new BorderLayout());
47     p2.add(jpState, BorderLayout.WEST);
48     p2.add(jpZip, BorderLayout.CENTER);
49
50     // Panel p3 for holding jtfCity and p2
51     JPanel p3 = new JPanel();
52     p3.setLayout(new BorderLayout());
53     p3.add(jtfCity, BorderLayout.CENTER);
```

create UI

```
54      p3.add(p2, BorderLayout.EAST);
55
56      // Panel p4 for holding jtfName, jtfStreet, and p3
57      JPanel p4 = new JPanel();
58      p4.setLayout(new GridLayout(3, 1));
59      p4.add(jtfName);
60      p4.add(jtfStreet);
61      p4.add(p3);
62
63      // Place p1 and p4 into StudentPanel
64      JPanel studentPanel = new JPanel(new BorderLayout());
65      studentPanel.setBorder(new BevelBorder(BevelBorder.RAISED));
66      studentPanel.add(p1, BorderLayout.WEST);
67      studentPanel.add(p4, BorderLayout.CENTER);
68
69      // Add the student panel and button to the applet
70      add(studentPanel, BorderLayout.CENTER);
71      add(jbtRegister, BorderLayout.SOUTH);
72
73      // Register listener
74      jbtRegister.addActionListener(new ButtonListener());          register listener
75
76      // Find the IP address of the Web server
77      if (!isStandAlone)
78        host = getCodeBase().getHost();                             get server name
79    }
80
81    /** Handle button action */
82    private class ButtonListener implements ActionListener {
83      @Override
84      public void actionPerformed(ActionEvent e) {
85        try {
86          // Establish connection with the server
87          Socket socket = new Socket(host, 8000);                   server socket
88
89          // Create an output stream to the server
90          ObjectOutputStream toServer =                             output stream
91            new ObjectOutputStream(socket.getOutputStream());
92
93          // Get text field
94          String name = jtfName.getText().trim();
95          String street = jtfStreet.getText().trim();
96          String city = jtfCity.getText().trim();
97          String state = jtfState.getText().trim();
98          String zip = jtfZip.getText().trim();
99
100         // Create a StudentAddress object and send to the server
101         StudentAddress s =
102           new StudentAddress(name, street, city, state, zip);      send to server
103         toServer.writeObject(s);
104       }
105       catch (IOException ex) {
106         System.err.println(ex);
107       }
108     }
109   }
110
111   /** Run the applet as an application */
112   public static void main(String[] args) {
113     // Create a frame
```

```
114      JFrame frame = new JFrame("Register Student Client");
115
116      // Create an instance of the applet
117      StudentClient applet = new StudentClient();
118      applet.isStandAlone = true;
119
120      // Get host
121      if (args.length == 1) applet.host = args[0];
122
123      // Add the applet instance to the frame
124      frame.add(applet, BorderLayout.CENTER);
125
126      // Invoke init() and start()
127      applet.init();
128      applet.start();
129
130      // Display the frame
131      frame.pack();
132      frame.setVisible(true);
133    }
134  }
```

## LISTING 33.9 StudentServer.java

```
1  import java.io.*;
2  import java.net.*;
3
4  public class StudentServer {
5    private ObjectOutputStream outputToFile;
6    private ObjectInputStream inputFromClient;
7
8    public static void main(String[] args) {
9      new StudentServer();
10   }
11
12   public StudentServer() {
13     try {
14       // Create a server socket
15       ServerSocket serverSocket = new ServerSocket(8000);
16       System.out.println("Server started ");
17
18       // Create an object output stream
19       outputToFile = new ObjectOutputStream(
20         new FileOutputStream("student.dat", true));
21
22       while (true) {
23         // Listen for a new connection request
24         Socket socket = serverSocket.accept();
25
26         // Create an input stream from the socket
27         inputFromClient =
28           new ObjectInputStream(socket.getInputStream());
29
30         // Read from input
31         Object object = inputFromClient.readObject();
32
33         // Write to the file
34         outputToFile.writeObject(object);
35         System.out.println("A new student object is stored");
36       }
```

server socket

output to file

connect to client

input stream

get from client

write to file

```
37      }
38      catch(ClassNotFoundException ex) {
39        ex.printStackTrace();
40      }
41      catch(IOException ex) {
42        ex.printStackTrace();
43      }
44      finally {
45        try {
46          inputFromClient.close();
47          outputToFile.close();
48        }
49        catch (Exception ex) {
50          ex.printStackTrace();
51        }
52      }
53    }
54  }
```

On the client side, when the user clicks the *Register to the Server* button, the client creates a socket to connect to the host (line 87), creates an `ObjectOutputStream` on the output stream of the socket (lines 90–91), and invokes the `writeObject` method to send the `StudentAddress` object to the server through the object output stream (line 103).

On the server side, when a client connects to the server, the server creates an `ObjectInputStream` on the input stream of the socket (lines 27–28), invokes the `readObject` method to receive the `StudentAddress` object through the object input stream (line 31), and writes the object to a file (line 34).

This program can run either as an applet or as an application. To run it as an application, the host name is passed as a command-line argument.

**33.9** Can an applet connect to a server that is different from the machine where the applet is located?

**33.10** How do you find the host name of an applet?

**33.11** How do you send and receive an object?


MyProgrammingLab™

## 33.7 Case Study: Distributed Tic-Tac-Toe Games

*This section develops an applet that enables two players to play the tic-tac-toe game on the Internet.*


In Section 18.9, Case Study: Developing a Tic-Tac-Toe Game, you developed an applet for a tic-tac-toe game that enables two players to play the game on the same machine. In this section, you will learn how to develop a distributed tic-tac-toe game using multithreads and networking with socket streams. A distributed tic-tac-toe game enables users to play on different machines from anywhere on the Internet.

You need to develop a server for multiple clients. The server creates a server socket and accepts connections from every two players to form a session. Each session is a thread that communicates with the two players and determines the status of the game. The server can establish any number of sessions, as shown in Figure 33.13.

For each session, the first client connecting to the server is identified as player 1 with token `X`, and the second client connecting is identified as player 2 with token `O`. The server notifies the players of their respective tokens. Once two clients are connected to it, the server starts a thread to facilitate the game between the two players by performing the steps repeatedly, as shown in Figure 33.14.

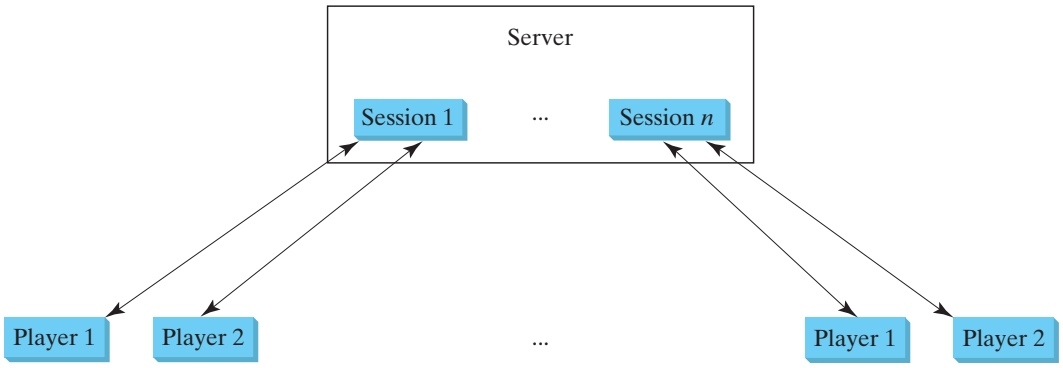



**FIGURE 33.13** The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

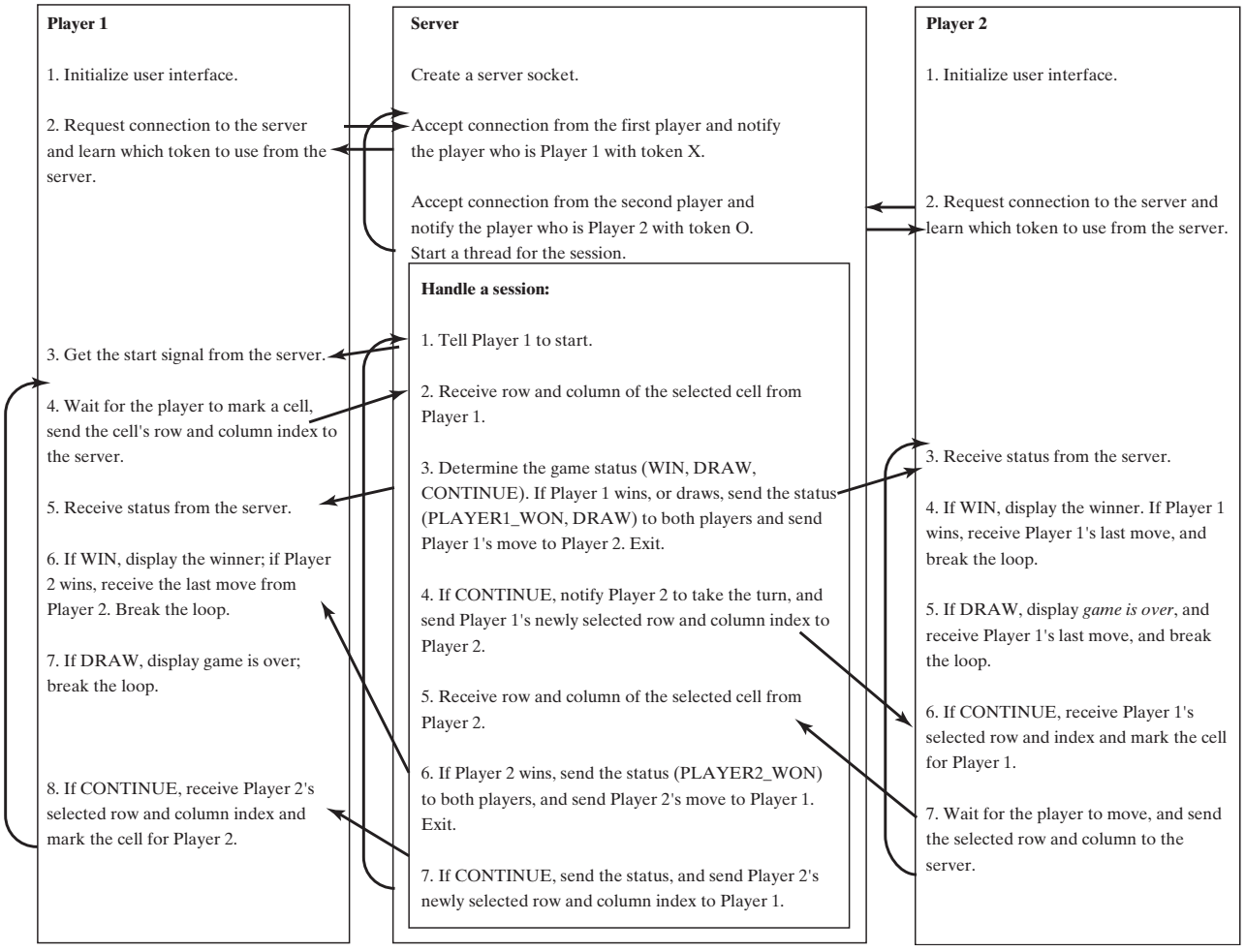



**FIGURE 33.14** The server starts a thread to facilitate communications between the two players.

The server does not have to be a graphical component, but creating it as a frame in which game information can be viewed is user-friendly. You can create a scroll pane to hold a text area in the frame and display game information in the text area. The server creates a thread to handle a game session when two players are connected to the server.

The client is responsible for interacting with the players. It creates a user interface with nine cells, and displays the game title and status to the players in the labels. The client class is very similar to the `TicTacToe` class presented in the case study in Section 18.9. However, the

client in this example does not determine the game status (win or draw); it simply passes the moves to the server and receives the game status from the server.

Based on the foregoing analysis, you can create the following classes:

- `TicTacToeServer` serves all the clients in Listing 33.11.
- `HandleASession` facilitates the game for two players. This class is defined in TicTacToeServer.java.
- `TicTacToeClient` models a player in Listing 33.12.
- `Cell` models a cell in the game. It is an inner class in `TicTacToeClient`.
- `TicTacToeConstants` is an interface that defines the constants shared by all the classes in the example in Listing 33.10.

The relationships of these classes are shown in Figure 33.15.

## LISTING 33.10 TicTacToeConstants.java

```
1  public interface TicTacToeConstants {
2    public static int PLAYER1 = 1; // Indicate player 1
3    public static int PLAYER2 = 2; // Indicate player 2
```

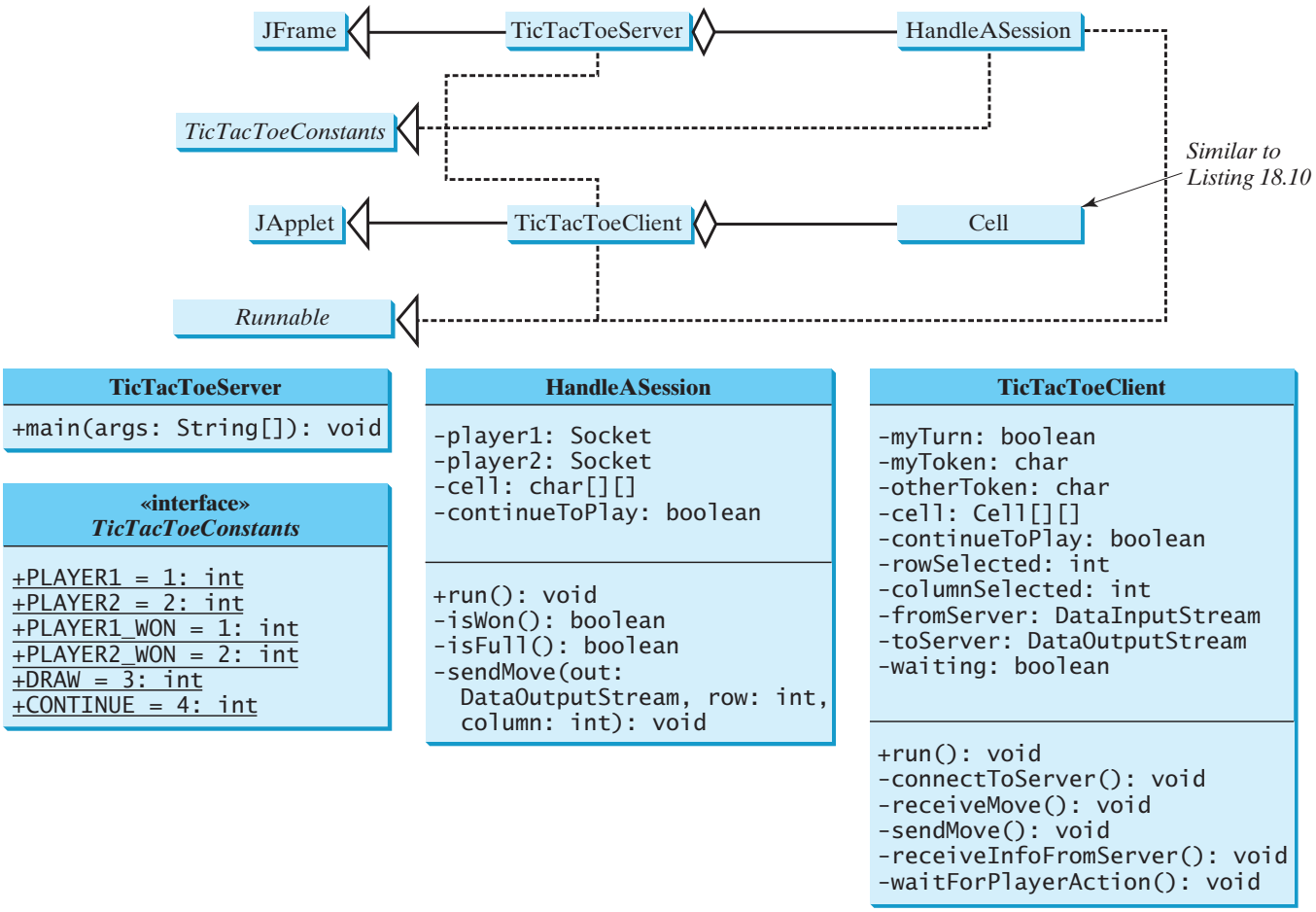



**FIGURE 33.15** `TicTacToeServer` creates an instance of `HandleASession` for each session of two players. `TicTacToeClient` creates nine cells in the UI.

```
4    public static int PLAYER1_WON = 1; // Indicate player 1 won
5    public static int PLAYER2_WON = 2; // Indicate player 2 won
6    public static int DRAW = 3; // Indicate a draw
7    public static int CONTINUE = 4; // Indicate to continue
8  }
```

## LISTING 33.11 TicTacToeServer.java

```
1  import java.io.*;
2  import java.net.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import java.util.Date;
6
7  public class TicTacToeServer extends JFrame
8      implements TicTacToeConstants {
9    public static void main(String[] args) {
10     TicTacToeServer frame = new TicTacToeServer();
11   }
12
13   public TicTacToeServer() {
14     JTextArea jtaLog = new JTextArea();
15
16     // Create a scroll pane to hold text area
17     JScrollPane scrollPane = new JScrollPane(jtaLog);
18
19     // Add the scroll pane to the frame
20     add(scrollPane, BorderLayout.CENTER);
21
22     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     setSize(300, 300);
24     setTitle("TicTacToeServer");
25     setVisible(true);
26
27     try {
28       // Create a server socket
29       ServerSocket serverSocket = new ServerSocket(8000);
30       jtaLog.append(new Date() +
31         ": Server started at socket 8000\n");
32
33       // Number a session
34       int sessionNo = 1;
35
36       // Ready to create a session for every two players
37       while (true) {
38         jtaLog.append(new Date() +
39           ": Wait for players to join session " + sessionNo + '\n');
40
41         // Connect to player 1
42         Socket player1 = serverSocket.accept();
43
44         jtaLog.append(new Date() + ": Player 1 joined session " +
45           sessionNo + '\n');
46         jtaLog.append("Player 1's IP address" +
47           player1.getInetAddress().getHostAddress() + '\n');
48
49         // Notify that the player is Player 1
50         new DataOutputStream(
51           player1.getOutputStream()).writeInt(PLAYER1);
52
```

run server

create UI

server socket

connect to client

to player1

```
53          // Connect to player 2
54          Socket player2 = serverSocket.accept();
55
56          jtaLog.append(new Date() +
57            ": Player 2 joined session " + sessionNo + '\n');
58          jtaLog.append("Player 2's IP address" +
59            player2.getInetAddress().getHostAddress() + '\n');
60
61          // Notify that the player is Player 2
62          new DataOutputStream(                                          to player2
63            player2.getOutputStream()).writeInt(PLAYER2);
64
65          // Display this session and increment session number
66          jtaLog.append(new Date() + ": Start a thread for session " +
67            sessionNo++ + '\n');
68
69          // Create a new thread for this session of two players
70          HandleASession task = new HandleASession(player1, player2);    a session for two players
71
72          // Start the new thread
73          new Thread(task).start();
74        }
75      }
76      catch(IOException ex) {
77        System.err.println(ex);
78      }
79    }
80  }
81
82  // Define the thread class for handling a new session for two players
83  class HandleASession implements Runnable, TicTacToeConstants {
84    private Socket player1;
85    private Socket player2;
86
87    // Create and initialize cells
88    private char[][] cell =  new char[3][3];
89
90    private DataInputStream fromPlayer1;
91    private DataOutputStream toPlayer1;
92    private DataInputStream fromPlayer2;
93    private DataOutputStream toPlayer2;
94
95    // Continue to play
96    private boolean continueToPlay = true;
97
98    /** Construct a thread */
99    public HandleASession(Socket player1, Socket player2) {
100     this.player1 = player1;
101     this.player2 = player2;
102
103     // Initialize cells
104     for (int i = 0; i < 3; i++)
105       for (int j = 0; j < 3; j++)
106         cell[i][j] = ' ';
107   }
108
109   @Override /** Implement the run() method for the thread */
110   public void run() {
111     try {
112       // Create data input and output streams
```

```
113         DataInputStream fromPlayer1 = new DataInputStream(
114           player1.getInputStream());
115         DataOutputStream toPlayer1 = new DataOutputStream(
116           player1.getOutputStream());
117         DataInputStream fromPlayer2 = new DataInputStream(
118           player2.getInputStream());
119         DataOutputStream toPlayer2 = new DataOutputStream(
120           player2.getOutputStream());
121
122         // Write anything to notify player 1 to start
123         // This is just to let player 1 know to start
124         toPlayer1.writeInt(1);
125
126         // Continuously serve the players and determine and report
127         // the game status to the players
128         while (true) {
129           // Receive a move from player 1
130           int row = fromPlayer1.readInt();
131           int column = fromPlayer1.readInt();
132           cell[row][column] = 'X';
133
134           // Check if Player 1 wins
135           if (isWon('X')) {
136             toPlayer1.writeInt(PLAYER1_WON);
137             toPlayer2.writeInt(PLAYER1_WON);
138             sendMove(toPlayer2, row, column);
139             break; // Break the loop
140           }
141           else if (isFull()) { // Check if all cells are filled
142             toPlayer1.writeInt(DRAW);
143             toPlayer2.writeInt(DRAW);
144             sendMove(toPlayer2, row, column);
145             break;
146           }
147           else {
148             // Notify player 2 to take the turn
149             toPlayer2.writeInt(CONTINUE);
150
151             // Send player 1's selected row and column to player 2
152             sendMove(toPlayer2, row, column);
153          }
154
155           // Receive a move from Player 2
156           row = fromPlayer2.readInt();
157           column = fromPlayer2.readInt();
158           cell[row][column] = 'O';
159
160           // Check if Player 2 wins
161           if (isWon('O')) {
162             toPlayer1.writeInt(PLAYER2_WON);
163             toPlayer2.writeInt(PLAYER2_WON);
164             sendMove(toPlayer1, row, column);
165             break;
166           }
167           else {
168             // Notify player 1 to take the turn
169             toPlayer1.writeInt(CONTINUE);
170
171             // Send player 2's selected row and column to player 1
```

```
172             sendMove(toPlayer1, row, column);
173           }
174         }
175       }
176       catch(IOException ex) {
177         System.err.println(ex);
178       }
179     }
180
181     /** Send the move to other player */
182     private void sendMove(DataOutputStream out, int row, int column)
183         throws IOException {
184       out.writeInt(row); // Send row index
185       out.writeInt(column); // Send column index
186     }
187
188     /** Determine if the cells are all occupied */
189     private boolean isFull() {
190       for (int i = 0; i < 3; i++)
191         for (int j = 0; j < 3; j++)
192           if (cell[i][j] == ' ')
193             return false; // At least one cell is not filled
194
195       // All cells are filled
196       return true;
197     }
198
199     /** Determine if the player with the specified token wins */
200     private boolean isWon(char token) {
201       // Check all rows
202       for (int i = 0; i < 3; i++)
203         if ((cell[i][0] == token)
204             && (cell[i][1] == token)
205             && (cell[i][2] == token)) {
206           return true;
207         }
208
209       /** Check all columns */
210       for (int j = 0; j < 3; j++)
211         if ((cell[0][j] == token)
212             && (cell[1][j] == token)
213             && (cell[2][j] == token)) {
214           return true;
215         }
216
217       /** Check major diagonal */
218       if ((cell[0][0] == token)
219           && (cell[1][1] == token)
220           && (cell[2][2] == token)) {
221         return true;
222       }
223
224       /** Check subdiagonal */
225       if ((cell[0][2] == token)
226           && (cell[1][1] == token)
227           && (cell[2][0] == token)) {
228         return true;
229       }
230
```

```
231       /** All checked, but no winner */
232       return false;
233     }
234   }
```

## LISTING 33.12 TicTacToeClient.java

```
1   import java.awt.*;
2   import java.awt.event.*;
3   import javax.swing.*;
4   import javax.swing.border.LineBorder;
5   import java.io.*;
6   import java.net.*;
7
8   public class TicTacToeClient extends JApplet
9       implements Runnable, TicTacToeConstants {
10    // Indicate whether the player has the turn
11    private boolean myTurn = false;
12
13    // Indicate the token for the player
14    private char myToken = ' ';
15
16    // Indicate the token for the other player
17    private char otherToken = ' ';
18
19    // Create and initialize cells
20    private Cell[][] cell =  new Cell[3][3];
21
22    // Create and initialize a title label
23    private JLabel jlblTitle = new JLabel();
24
25    // Create and initialize a status label
26    private JLabel jlblStatus = new JLabel();
27
28    // Indicate selected row and column by the current move
29    private int rowSelected;
30    private int columnSelected;
31
32    // Input and output streams from/to server
33    private DataInputStream fromServer;
34    private DataOutputStream toServer;
35
36    // Continue to play?
37    private boolean continueToPlay = true;
38
39    // Wait for the player to mark a cell
40    private boolean waiting = true;
41
42    // Indicate if it runs as application
43    private boolean isStandAlone = false;
44
45    // Host name or IP address
46    private String host = "localhost";
47
48    @Override /** Initialize UI */
49    public void init() {
50      // Panel p to hold cells
51      JPanel p = new JPanel();
52      p.setLayout(new GridLayout(3, 3, 0, 0));
```

create UI

```
53      for (int i = 0; i < 3; i++)
54        for (int j = 0; j < 3; j++)
55          p.add(cell[i][j] = new Cell(i, j));
56
57      // Set properties for labels and borders for labels and panel
58      p.setBorder(new LineBorder(Color.black, 1));
59      jlblTitle.setHorizontalAlignment(JLabel.CENTER);
60      jlblTitle.setFont(new Font("SansSerif", Font.BOLD, 16));
61      jlblTitle.setBorder(new LineBorder(Color.black, 1));
62      jlblStatus.setBorder(new LineBorder(Color.black, 1));
63
64      // Place the panel and the labels for the applet
65      add(jlblTitle, BorderLayout.NORTH);
66      add(p, BorderLayout.CENTER);
67      add(jlblStatus, BorderLayout.SOUTH);
68
69      // Connect to the server
70      connectToServer();                                          connect to server
71    }
72
73    private void connectToServer() {
74      try {
75        // Create a socket to connect to the server
76        Socket socket;
77        if (isStandAlone)
78          socket = new Socket(host, 8000);                        standalone
79        else
80          socket = new Socket(getCodeBase().getHost(), 8000);     applet
81
82        // Create an input stream to receive data from the server
83        fromServer = new DataInputStream(socket.getInputStream());   input from server
84
85        // Create an output stream to send data to the server
86        toServer = new DataOutputStream(socket.getOutputStream());   output to server
87      }
88      catch (Exception ex) {
89        System.err.println(ex);
90      }
91
92      // Control the game on a separate thread
93      Thread thread = new Thread(this);
94      thread.start();
95    }
96
97    @Override
98    public void run() {
99      try {
100       // Get notification from the server
101       int player = fromServer.readInt();
102
103       // Am I player 1 or 2?
104       if (player == PLAYER1) {
105         myToken = 'X';
106         otherToken = 'O';
107         jlblTitle.setText("Player 1 with token 'X'");
108         jlblStatus.setText("Waiting for player 2 to join");
109
110         // Receive startup notification from the server
111         fromServer.readInt(); // Whatever read is ignored
112
```

```
113           // The other player has joined
114           jlblStatus.setText("Player 2 has joined. I start first");
115
116           // It is my turn
117           myTurn = true;
118         }
119         else if (player == PLAYER2) {
120           myToken = 'O';
121           otherToken = 'X';
122           jlblTitle.setText("Player 2 with token 'O'");
123           jlblStatus.setText("Waiting for player 1 to move");
124         }
125
126         // Continue to play
127         while (continueToPlay) {
128           if (player == PLAYER1) {
129             waitForPlayerAction(); // Wait for player 1 to move
130             sendMove(); // Send the move to the server
131             receiveInfoFromServer(); // Receive info from the server
132           }
133           else if (player == PLAYER2) {
134             receiveInfoFromServer(); // Receive info from the server
135             waitForPlayerAction(); // Wait for player 2 to move
136             sendMove() ; // Send player 2's move to the server
137           }
138         }
139       }
140       catch (Exception ex) {
141       }
142     }
143
144     /** Wait for the player to mark a cell */
145     private void waitForPlayerAction() throws InterruptedException {
146       while (waiting) {
147         Thread.sleep(100);
148       }
149
150       waiting = true;
151     }
152
153     /** Send this player's move to the server */
154     private void sendMove() throws IOException {
155       toServer.writeInt(rowSelected); // Send the selected row
156       toServer.writeInt(columnSelected); // Send the selected column
157     }
158
159     /** Receive info from the server */
160     private void receiveInfoFromServer() throws IOException {
161       // Receive game status
162       int status = fromServer.readInt();
163
164       if (status == PLAYER1_WON) {
165         // Player 1 won, stop playing
166         continueToPlay = false;
167         if (myToken == 'X') {
168           jlblStatus.setText("I won! (X)");
169         }
170         else if (myToken == 'O') {
```

```
171           jlblStatus.setText("Player 1 (X) has won!");
172           receiveMove();
173         }
174       }
175       else if (status == PLAYER2_WON) {
176         // Player 2 won, stop playing
177         continueToPlay = false;
178         if (myToken == 'O') {
179           jlblStatus.setText("I won! (O)");
180         }
181         else if (myToken == 'X') {
182           jlblStatus.setText("Player 2 (O) has won!");
183           receiveMove();
184         }
185       }
186       else if (status == DRAW) {
187         // No winner, game is over
188         continueToPlay = false;
189         jlblStatus.setText("Game is over, no winner!");
190
191         if (myToken == 'O') {
192           receiveMove();
193         }
194       }
195       else {
196         receiveMove();
197         jlblStatus.setText("My turn");
198         myTurn = true; // It is my turn
199       }
200     }
201
202     private void receiveMove() throws IOException {
203       // Get the other player's move
204       int row = fromServer.readInt();
205       int column = fromServer.readInt();
206       cell[row][column].setToken(otherToken);
207     }
208
209     // An inner class for a cell
210     public class Cell extends JPanel {                                    model a cell
211       // Indicate the row and column of this cell in the board
212       private int row;
213       private int column;
214
215       // Token used for this cell
216       private char token = ' ';
217
218       public Cell(int row, int column) {
219         this.row = row;
220         this.column = column;
221         setBorder(new LineBorder(Color.black, 1)); // Set cell's border
222         addMouseListener(new ClickListener());  // Register listener     register listener
223       }
224
225       /** Return token */
226       public char getToken() {
227         return token;
228       }
```

```
229
230       /** Set a new token */
231       public void setToken(char c) {
232         token = c;
233         repaint();
234       }
235
236       @Override /** Paint the cell */
237       protected void paintComponent(Graphics g) {
238         super.paintComponent(g);
239
240         if (token == 'X') {
241           g.drawLine(10, 10, getWidth() - 10, getHeight() - 10);
242           g.drawLine(getWidth() - 10, 10, 10, getHeight() - 10);
243         }
244         else if (token == 'O') {
245           g.drawOval(10, 10, getWidth() - 20, getHeight() - 20);
246         }
247       }
248
249       /** Handle mouse click on a cell */
250       private class ClickListener extends MouseAdapter {
251         @Override
252         public void mouseClicked(MouseEvent e) {
253           // If cell is not occupied and the player has the turn
254           if (token == ' ' && myTurn) {
255             setToken(myToken);  // Set the player's token in the cell
256             myTurn = false;
257             rowSelected = row;
258             columnSelected = column;
259             jlblStatus.setText("Waiting for the other player to move");
260             waiting = false; // Just completed a successful move
261           }
262         }
263       }
264     }
265   }
```

draw X

draw O

mouse listener

main method omitted

The server can serve any number of sessions simultaneously. Each session takes care of two players. The client can be a Java applet or a Java application. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 33.16 and 33.17 show sample runs of the server and the clients.


**FIGURE 33.16** `TicTacToeServer` accepts connection requests and creates sessions to serve pairs of players.

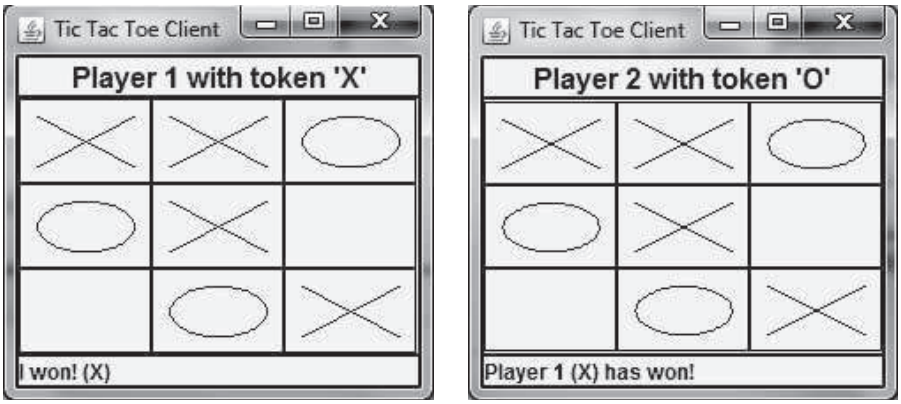



**FIGURE 33.17** `TicTacToeClient` can run as an applet or an application.

The `TicTacToeConstants` interface defines the constants shared by all the classes in the project. Each class that uses the constants needs to implement the interface. Centrally defining constants in an interface is a common practice in Java. For example, all the constants shared by Swing classes are defined in `java.swing.SwingConstants`.

Once a session is established, the server receives moves from the players in alternation. Upon receiving a move from a player, the server determines the status of the game. If the game is not finished, the server sends the status (`CONTINUE`) and the player's move to the other player. If the game is won or a draw, the server sends the status (`PLAYER1_WON`, `PLAYER2_WON`, or `DRAW`) to both players.

The implementation of Java network programs at the socket level is tightly synchronized. An operation to send data from one machine requires an operation to receive data from the other machine. As shown in this example, the server and the client are tightly synchronized to send or receive data.

**33.12** Will the program work if lines 48-49 in Listing 33.12 TicTacToeClient.java

```
@Override /** Initialize UI */
public void init() {
```

is changed to

```
public TicTacToeClient() {
```

Check Point
MyProgrammingLab™

**33.13** If a player does not have the turn, but clicks on an empty cell, will the code in line 254 in Listing 33.12 be executed and will the code in line 255 be executed?

## CHAPTER SUMMARY

1. Java supports stream sockets and datagram sockets. *Stream sockets* use TCP (Transmission Control Protocol) for data transmission, whereas *datagram sockets* use UDP (User Datagram Protocol). Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. UDP, in contrast, cannot guarantee lossless transmission.

2. To create a server, you must first obtain a server socket, using `new ServerSocket(port)`. After a server socket is created, the server can start to listen for connections, using the `accept()` method on the server socket. The client requests a connection to a server by using `new Socket(serverName, port)` to create a client socket.

3. Stream socket communication is very much like input/output stream communication after the connection between a server and a client is established. You can obtain an input stream using the `getInputStream()` method and an output stream using the `getOutputStream()` method on the socket.

4. A server must often work with multiple clients at the same time. You can use threads to handle the server's multiple clients simultaneously by creating a thread for each connection.

5. Applets are good for deploying multiple clients. They can run anywhere with a single copy of the program. However, because of security restrictions, an applet client can connect only to the server where the applet is loaded.

## Test Questions

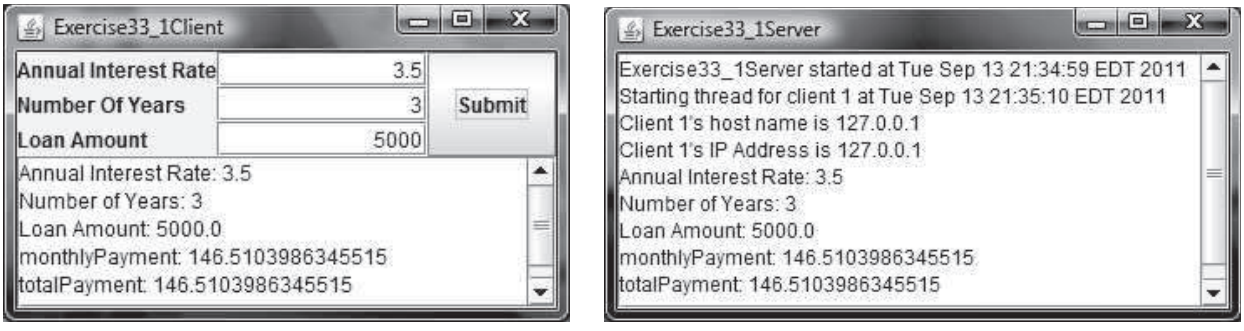
Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

MyProgrammingLab™

## Programming Exercises

### Section 33.2

*33.1 (*Loan server*) Write a server for a client. The client sends loan information (annual interest rate, number of years, and loan amount) to the server (see Figure 33.18a). The server computes monthly payment and total payment and sends them back to the client (see Figure 33.18b). Name the client Exercise33_1Client and the server Exercise33_1Server.




(a) (b)

**Figure 33.18** The client in (a) sends the annual interest rate, number of years, and loan amount to the server and receives the monthly payment and total payment from the server in (b).

33.2 (*Network I/O using `Scanner` and `PrintWriter`*) Rewrite the server and client programs in Listings 33.1 and 33.2 using a `Scanner` for input and a `PrintWriter` for output. Name the server Exercise33_2Server and the client Exercise33_2Client.

### Sections 33.3–33.4

*33.3 (*Loan server for multiple clients*) Revise Exercise 33.1 to write a server for multiple clients.

## Section 33.5

**33.4** (*Web visit count*) Section 33.5, Applet Clients, created an applet that shows the number of visits made to a Web page. The count is stored in a file on the server side. Every time the page is visited or reloaded, the applet sends a request to the server, and the server increases the count and sends it to the applet. The count is stored using a random-access file. When the applet is loaded, the server reads the count from the file, increases it, and saves it back to the file. Rewrite the program to improve its performance. Read the count from the file when the server starts, and save the count to the file when the server stops, using the *Stop* button, as shown in Figure 33.19. When the server is alive, use a variable to store the count. Name the client Exercise33_4Client and the server Exercise33_4Server. The client program should be the same as in Listing 33.6. Rewrite the server as a GUI application with a *Stop* button that exits the server.

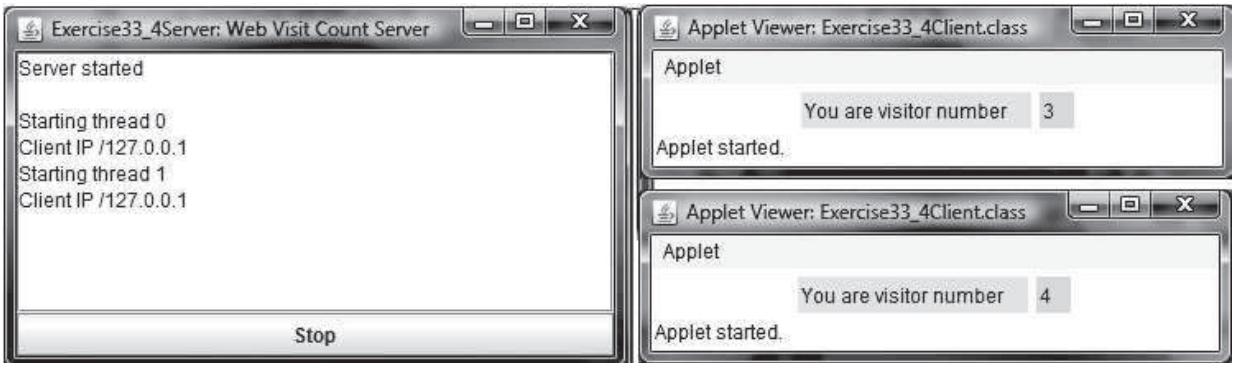



**FIGURE 33.19** The applet displays how many times this Web page has been accessed. The server stores the count.

**33.5** (*Create a stock ticker in an applet*) Write an applet like the one in Programming Exercise 18.16. Assume that the applet gets the stock index from a file named **Ticker.txt** stored on the Web server. Enable the applet to run as a standalone.

## Section 33.6

**33.6** (*Display and add addresses*) Develop a client/server application to view and add addresses, as shown in Figure 33.20.

- Define an `Address` class to hold the name, street, city, state, and zip in an object.
- The user can use the buttons *First*, *Next*, *Previous*, and *Last* to view an address, and the *Add* button to add a new address.
- Limit the concurrent connections to two clients.

Name the client Exercise33_6Client and the server Exercise33_6Server.


**FIGURE 33.20** You can view and add an address in this applet.

*33.7 (*Transfer last 100 numbers in an array*) Programming Exercise 24.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an array. Name the server program Exercise33_7Server and the client program Exercise33_7Client. Assume that the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

*33.8 (*Transfer last 100 numbers in an `ArrayList`*) Programming Exercise 24.12 retrieves the last 100 prime numbers from a file **PrimeNumbers.dat**. Write a client program that requests the server to send the last 100 prime numbers in an `ArrayList`. Name the server program Exercise33_8Server and the client program Exercise33_8Client. Assume that the numbers of the `long` type are stored in **PrimeNumbers.dat** in binary format.

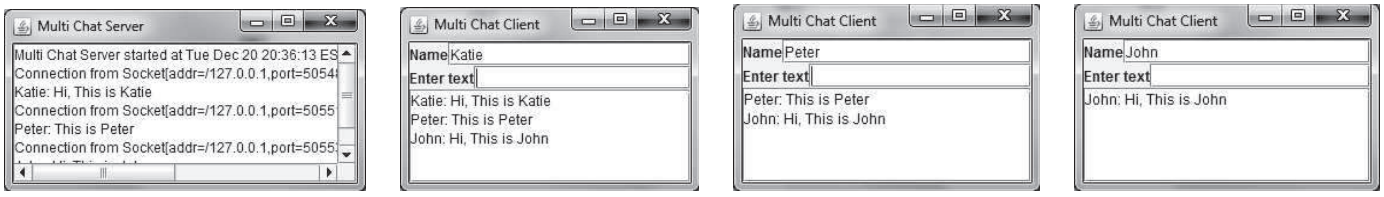
## Section 33.7

**33.9 (*Chat*) Write a program that enables two users to chat. Implement one user as the server (Figure 33.21a) and the other as the client (Figure 33.21b). The server has two text areas: one for entering text and the other (noneditable) for displaying text received from the client. When the user presses the *Enter* key, the current line is sent to the client. The client has two text areas: one (noneditable) for receiving text from the server, and the other for entering text. When the user presses the *Enter* key, the current line is sent to the server. Name the client Exercise33_9Client and the server Exercise33_9Server.


(a) (b)

**FIGURE 33.21** The server and client send text to and receive text from each other.

***33.10 (*Multiple client chat*) Write a program that enables any number of clients to chat. Implement one server that serves all the clients, as shown in Figure 33.22. Name the client Exercise33_10Client and the server Exercise33_10Server.




(a) (b) (c) (d)

**FIGURE 33.22** The server starts in (a) with three clients in (b), (c), and (d).