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).

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.

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.

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.

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

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.
33.2 Client/Server Computing

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`:

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

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

### 33.2.2 Client Sockets

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

```java
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:

```java
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:

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

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

```java
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.
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.

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:

```java
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 `DataOutputStream` stream `output` to read and write primitive data values:

```java
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).

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 33.3](image1)

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

![Figure 33.4](image2)

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

![Figure 33.5](image3)

**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**

```java
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class Server extends JFrame {
    // Text area for displaying contents
    private JTextArea jta = new JTextArea();

    public static void main(String[] args) {
        new Server();
    }

    public Server() {
        // Place text area on the frame
        setLayout(new BorderLayout());
        add(new JScrollPane(jta), BorderLayout.CENTER);

        setTitle("Server");
        setSize(500, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true); // It is necessary to show the frame here!
        try {
            // Create a server socket
            ServerSocket serverSocket = new ServerSocket(8000);
            jta.append("Server started at " + new Date() + '\n');

            // Listen for a connection request
            Socket socket = serverSocket.accept();
            DataInputStream inputFromClient = new DataInputStream(socket.getInputStream());
            double radius = inputFromClient.readDouble();

            // Compute area
            double area = radius * radius * Math.PI;

            // Send area back to the client
            DataOutputStream outputToClient = new DataOutputStream(socket.getOutputStream());
            outputToClient.writeDouble(area);
            jta.append("Radius received from client: " + radius + '\n');
            jta.append("Area found: " + area + '\n');
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```
```java
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Client extends JFrame {
    // Text field for receiving radius
    private JTextField jtf = new JTextField();

    // Text area to display contents
    private JTextArea jta = new JTextArea();

    // IO streams
    private DataOutputStream toServer;
    private DataInputStream fromServer;

    public static void main(String[] args) {
        new Client();
    }

    public Client() {
        // Panel p to hold the label and text field
        JPanel p = new JPanel();
        p.setLayout(new BorderLayout());
        p.add(new JLabel("Enter radius"), BorderLayout.WEST);
        p.add(jtf, BorderLayout.CENTER);
        jtf.setHorizontalAlignment(JTextField.RIGHT);
        setLayout(new BorderLayout());
        add(p, BorderLayout.NORTH);
        add(new JScrollPane(jta), BorderLayout.CENTER);
        jtf.addActionListener(new TextFieldListener());
        setTitle("Client");
        setSize(500, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true); // It is necessary to show the frame here!
        try {
            // Create a socket to connect to the server
            Socket socket = new Socket("localhost", 8000); // localhost
            // Socket socket = new Socket("130.254.204.33", 8000); // 192.168.1.18
            // Socket socket = new Socket("liang.armstrong.edu", 8000);

            // Create an input stream to receive data from the server
            fromServer = new DataInputStream(socket.getInputStream());

            // Create an output stream to send data to the server
            toServer = new DataOutputStream(socket.getOutputStream());
        } catch (IOException ex) {
            jta.append(ex.toString() + 'n');
        }
    }
}
```

**List 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();
- **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** setLayout(new BorderLayout());
- **30** add(p, BorderLayout.NORTH);
- **31** add(new JScrollPane(jta), BorderLayout.CENTER);
- **32** jtf.addActionListener(new TextFieldListener());
- **33** setTitle("Client");
- **34** setSize(500, 300);
- **35** setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
- **36** setVisible(true); // It is necessary to show the frame here!
- **37** try {
- **38** // Create a socket to connect to the server
- **39** Socket socket = new Socket("localhost", 8000); // localhost
- **40** // Socket socket = new Socket("130.254.204.33", 8000);
- **41** // Socket socket = new Socket("liang.armstrong.edu", 8000);
- **42**
- **43** // Create an input stream to receive data from the server
- **44** fromServer = new DataInputStream(socket.getInputStream());
- **45**
- **46** // Create an output stream to send data to the server
- **47** toServer = new DataOutputStream(socket.getOutputStream());
- **48** }
- **49** catch (IOException ex) {
- **50** jta.append(ex.toString() + 'n');
- **51** }
```
private class TextFieldListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    try {
      // Get the radius from the text field
      double radius = Double.parseDouble(jtf.getText().trim());

      // Send the radius to the server
      toServer.writeDouble(radius);
      toServer.flush();

      // Get area from the server
      double area = fromServer.readDouble();

      // Display to the text area
      jta.append("Radius is " + radius + "\n");
      jta.append("Area received from the server is "
        + area + "\n");
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}

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 con-
nection 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.

![Figure 33.6](image)

**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`.

```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.

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

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

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

```
import java.net.*;

public class IdentifyHostNameIP {

    public static void main(String[] args) {

        for (int i = 0; i < args.length; i++) {
            try {
                InetAddress address = InetAddress.getByName(args[i]);
                System.out.print("Host name: "+address.getHostName()+" ");
                System.out.println("IP address: "+address.getHostAddress());
            }
            catch (UnknownHostException ex) {
                System.err.println("Unknown host or IP address "+args[i]);
            }
        }
    }
}
```

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

**Listing 33.3** IdentifyHostNameIP.java

### Check Point

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:

```java
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](image)

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

![Figure 33.9](image)

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

### Listing 33.4 MultiThreadServer.java

```java
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
```
```java
public class MultiThreadServer extends JFrame {
    // Text area for displaying contents
    private JTextArea jta = new JTextArea();

    public static void main(String[] args) {
        new MultiThreadServer();
    }

    public MultiThreadServer() {
        // Place text area on the frame
        setLayout(new BorderLayout());
        add(new JScrollPane(jta), BorderLayout.CENTER);

        setTitle("MultiThreadServer");
        setSize(500, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true); // It is necessary to show the frame here!

        try {
            ServerSocket serverSocket = new ServerSocket(8000);
            jta.append("MultiThreadServer started at " + new Date() + '\n');

            int clientNo = 1;

            while (true) {
                Socket socket = serverSocket.accept();

                jta.append("Starting thread for client " + clientNo + " at " + new Date() + '\n');

                InetAddress inetAddress = socket.getInetAddress();
                jta.append("Client ", clientNo, ": host name is " + inetAddress.getHostAddress() + ", IP Address is " + inetAddress.getHostAddress() + '\n');

                HandleAClient task = new HandleAClient(socket);

                new Thread(task).start();

                clientNo++;
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }

    // Inner class
    class HandleAClient implements Runnable {
```
private Socket socket; // A connected socket

/** Construct a thread */
public HandleAClient(Socket socket) {
    this.socket = socket;
}

@Override /** Run a thread */
public void run() {
    try {
        // Create data input and output streams
        DataInputStream inputFromClient = new DataInputStream(socket.getInputStream());
        DataOutputStream outputToClient = new DataOutputStream(socket.getOutputStream());

        // Continuously serve the client
        while (true) {
            // Receive radius from the client
            double radius = inputFromClient.readDouble();

            // Compute area
            double area = radius * radius * Math.PI;

            // Send area back to the client
            outputToClient.writeDouble(area);

            jta.append("radius received from client: "+radius + '\n');
            jta.append("Area found: "+area + '\n');
        }
        catch (IOException e) {
            System.err.println(e);
        }
    }
}

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?

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.

![Applet Viewer: AppletClient.class](image)

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

**Listing 33.5 CountServer.java**

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

public class CountServer {
    private RandomAccessFile raf;
    private int count; // Count the access to the server

    public static void main(String[] args) {
        new CountServer();
    }

    public CountServer() {
        try {
            // Create a server socket
            ServerSocket serverSocket = new ServerSocket(8000);
            System.out.println("Server started");

            // Create or open the count file
            raf = new RandomAccessFile("count.dat", "rw");

            // Get the count
            if (raf.length() == 0)
                count = 0;
            else
                count = raf.readInt();

            while (true) {
                // Listen for a new connection request
                Socket socket = serverSocket.accept();

                // Create a DataOutputStream for the socket
                DataOutputStream outputToClient = new DataOutputStream(socket.getOutputStream());

                // Increase count and send the count to the client
                count++;
                outputToClient.writeInt(count);

                // Write new count back to the file
                raf.seek(0);
            }
        } catch (IOException e) {
            // Handle exceptions
        }
    }
}
```
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**

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

public class AppletClient extends JApplet {

    private JLabel jlblCount = new JLabel();

    // Indicate if it runs as application
    private boolean isStandAlone = false;

    // Host name or IP address
    private String host = "localhost";

    /** Initialize the applet */
    public void init() {
        add(jlblCount);

        try {
            // Create a socket to connect to the server
            Socket socket;
            if (isStandAlone)
                socket = new Socket(host, 8000); // for standalone
            else
                socket = new Socket(getCodeBase().getHost(), 8000); // for applet

            // Create an input stream to receive data from the server
            DataInputStream inputFromServer =
                    new DataInputStream(socket.getInputStream());

            // Receive the count from the server and display it on label
            int count = inputFromServer.readInt();
            jlblCount.setText("You are visitor number " + count);

            // Close the stream
            inputFromServer.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```
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.

![Register Student Client](image)

**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

```java
public class StudentAddress implements java.io.Serializable {  // serialized
    private String name;
    private String street;
    private String city;
    private String state;
    private String zip;

    public StudentAddress(String name, String street, String city,
                          String state, String zip) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getName() {
        return name;
    }

    public String getStreet() {
        return street;
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

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](image-url) 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**

```java
public class StudentClient extends JApplet {
    private JTextField jtfName = new JTextField(32);
    private JTextField jtfStreet = new JTextField(32);
    private JTextField jtfCity = new JTextField(20);
    private JTextField jtfState = new JTextField(2);
    private JTextField jtfZip = new JTextField(5);

    // Button for sending a student's address to the server
    private JButton jbtRegister = new JButton("Register to the Server");

    // Indicate if it runs as application
    private boolean isStandAlone = false;

    // Host name or IP address
    String host = "localhost";

    public void init() {
        // Panel p1 for holding labels Name, Street, and City
        JPanel p1 = new JPanel();
        p1.setLayout(new GridLayout(3, 1));
        p1.add(new JLabel("Name"));
        p1.add(new JLabel("Street"));
        p1.add(new JLabel("City"));

        // Panel jpState for holding state
        JPanel jpState = new JPanel();
        jpState.setLayout(new BorderLayout());
        jpState.add(new JLabel("State"), BorderLayout.WEST);
        jpState.add(jtfState, BorderLayout.CENTER);

        // Panel jpZip for holding zip
        JPanel jpZip = new JPanel();
        jpZip.setLayout(new BorderLayout());
        jpZip.add(new JLabel("Zip"), BorderLayout.WEST);
        jpZip.add(jtfZip, BorderLayout.CENTER);

        // Panel p2 for holding jpState and jpZip
        JPanel p2 = new JPanel();
        p2.setLayout(new BorderLayout());
        p2.add(jpState, BorderLayout.WEST);
        p2.add(jpZip, BorderLayout.CENTER);

        // Panel p3 for holding jtfCity and p2
        JPanel p3 = new JPanel();
        p3.setLayout(new BorderLayout());
        p3.add(jtfCity, BorderLayout.CENTER);
```
33.6 Sending and Receiving Objects

```java
// Panel p4 for holding jtfName, jtfStreet, and p3
JPanel p4 = new JPanel();
p4.setLayout(new GridLayout(3, 1));
p4.add(jtfName);
p4.add(jtfStreet);
p4.add(p3);

// Place p1 and p4 into StudentPanel
JPanel studentPanel = new JPanel(new BorderLayout());
studentPanel.setBorder(new BevelBorder(BevelBorder.RAISED));
studentPanel.add(p1, BorderLayout.WEST);
studentPanel.add(p4, BorderLayout.CENTER);

// Add the student panel and button to the applet
add(studentPanel, BorderLayout.CENTER);
add(jbtRegister, BorderLayout.SOUTH);

// Register listener
jbtRegister.addActionListener(new ButtonListener());

// Find the IP address of the Web server
if (!isStandAlone)
    host = getCodeBase().getHost();

/** Handle button action */
private class ButtonListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            // Establish connection with the server
            Socket socket = new Socket(host, 8000);

            // Create an output stream to the server
            ObjectOutputStream toServer =
                new ObjectOutputStream(socket.getOutputStream());

            // Get text field
            String name = jtfName.getText().trim();
            String street = jtfStreet.getText().trim();
            String city = jtfCity.getText().trim();
            String state = jtfState.getText().trim();
            String zip = jtfZip.getText().trim();

            // Create a StudentAddress object and send to the server
            StudentAddress s =
                new StudentAddress(name, street, city, state, zip);
            toServer.writeObject(s);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}

/** Run the applet as an application */
public static void main(String[] args) {
    // 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              }
37          }
38      }
33.7 Case Study: Distributed Tic-Tac-Toe Games

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?

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

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

```java
public interface TicTacToeConstants {
    public static int PLAYER1 = 1; // Indicate player 1
    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.
Listing 33.11 TicTacToeServer.java

```java
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.util.Date;

public class TicTacToeServer extends JFrame {
    implements TicTacToeConstants {
        public static void main(String[] args) {
            TicTacToeServer frame = new TicTacToeServer();
        }
    }

    public TicTacToeServer() {
        JTextArea jtaLog = new JTextArea();

        // Create a scroll pane to hold text area
        JScrollPane scrollPane = new JScrollPane(jtaLog);

        // Add the scroll pane to the frame
        add(scrollPane, BorderLayout.CENTER);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 300);
        setTitle("TicTacToeServer");
        setVisible(true);

        try {
            // Create a server socket
            ServerSocket serverSocket = new ServerSocket(8000);
            jtaLog.append(new Date() + "\n" + "\n" + "Server started at socket 8000\n");

            // Number a session
            int sessionNo = 1;

            // Ready to create a session for every two players
            while (true) {
                jtaLog.append(new Date() + "\n" + "Wait for players to join session " + sessionNo + '\n');

                // Connect to player 1
                Socket player1 = serverSocket.accept();

                jtaLog.append(new Date() + "\n" + ": Player 1 joined session " + sessionNo + '\n');
                jtaLog.append("Player 1's IP address" + "\n" + player1.getInetAddress().getHostAddress() + '\n');

                // Notify that the player is Player 1
                new DataOutputStream(player1.getOutputStream()).writeInt(PLAYER1);
            }
        }
    }
}
```
// Connect to player 2
Socket player2 = serverSocket.accept();

jtaLog.append(new Date() + " : Player 2 joined session "+ sessionNo + '
');
jtaLog.append("Player 2's IP address" + player2.getInetAddress().getHostAddress() + '
');

// Notify that the player is Player 2
new DataOutputStream( player2.getOutputStream() ).writeInt(PLAYER2);

// Display this session and increment session number
jtaLog.append(new Date() + " : Start a thread for session "+ sessionNo++ + '
');

// Create a new thread for this session of two players
HandleASession task = new HandleASession(player1, player2);

// Start the new thread
new Thread(task).start();

catch(IOException ex) {
    System.err.println(ex);
}

} // Define the thread class for handling a new session for two players
class HandleASession implements Runnable, TicTacToeConstants {
    private Socket player1;
    private Socket player2;

    // Create and initialize cells
    private char[][] cell = new char[3][3];

    private DataInputStream fromPlayer1;
    private DataOutputStream toPlayer1;
    private DataInputStream fromPlayer2;
    private DataOutputStream toPlayer2;

    // Continue to play
    private boolean continueToPlay = true;

    /** Construct a thread */
    public HandleASession(Socket player1, Socket player2) {
        this.player1 = player1;
        this.player2 = player2;

        // Initialize cells
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                cell[i][j] = ' ';

    }

    @Override /** Implement the run() method for the thread */
    public void run() {
        try {
            // Create data input and output streams
            try {
                fromPlayer1 = new DataInputStream(player1.getInputStream());
            } catch (IOException ex) {
                System.err.println(ex);
            }

            try {
                toPlayer1 = new DataOutputStream(player1.getOutputStream());
            } catch (IOException ex) {
                System.err.println(ex);
            }

            try {
                fromPlayer2 = new DataInputStream(player2.getInputStream());
            } catch (IOException ex) {
                System.err.println(ex);
            }

            try {
                toPlayer2 = new DataOutputStream(player2.getOutputStream());
            } catch (IOException ex) {
                System.err.println(ex);
            }

            // Display this session and increment session number
            jtaLog.append(new Date() + " : Start a thread for session "+ sessionNo++ + '
');

            // Create a new thread for this session of two players
            HandleASession task = new HandleASession(player1, player2);

            // Start the new thread
            new Thread(task).start();

        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
DataInputStream fromPlayer1 = new DataInputStream(player1.getInputStream);
DataOutputStream toPlayer1 = new DataOutputStream(player1.getOutputStream);
DataInputStream fromPlayer2 = new DataInputStream(player2.getInputStream);
DataOutputStream toPlayer2 = new DataOutputStream(player2.getOutputStream);

// Write anything to notify player 1 to start
// This is just to let player 1 know to start
toPlayer1.writeInt(1);

// Continuously serve the players and determine and report
// the game status to the players
while (true) {
    // Receive a move from player 1
    int row = fromPlayer1.readInt();
    int column = fromPlayer1.readInt();
    cell[row][column] = 'X';

    // Check if Player 1 wins
    if (isWon('X')) {
        toPlayer1.writeInt(PLAYER1_WON);
        toPlayer2.writeInt(PLAYER1_WON);
        sendMove(toPlayer2, row, column);
        break; // Break the loop
    } else if (isFull()) { // Check if all cells are filled
        toPlayer1.writeInt(DRAW);
        toPlayer2.writeInt(DRAW);
        sendMove(toPlayer2, row, column);
        break;
    } else {
        // Notify player 2 to take the turn
        toPlayer2.writeInt(CONTINUE);

        // Send player 1's selected row and column to player 2
        sendMove(toPlayer2, row, column);
    }

    // Receive a move from Player 2
    row = fromPlayer2.readInt();
    column = fromPlayer2.readInt();
    cell[row][column] = 'O';

    // Check if Player 2 wins
    if (isWon('O')) {
        toPlayer1.writeInt(PLAYER2_WON);
        toPlayer2.writeInt(PLAYER2_WON);
        sendMove(toPlayer1, row, column);
        break;
    } else {
        // Notify player 1 to take the turn
        toPlayer1.writeInt(CONTINUE);

        // Send player 2's selected row and column to player 1
33.7 Case Study: Distributed Tic-Tac-Toe Games

```java
    sendMove(toPlayer1, row, column);
    }
    }
    catch(IOException ex) {
        System.err.println(ex);
    }
}

/** Send the move to other player */
private void sendMove(DataOutputStream out, int row, int column) throws IOException {
    out.writeInt(row); // Send row index
    out.writeInt(column); // Send column index
}

/** Determine if the cells are all occupied */
private boolean isFull() {
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (cell[i][j] == ' ')
                return false; // At least one cell is not filled
    // All cells are filled
    return true;
}

/** Determine if the player with the specified token wins */
private boolean isWon(char token) {
    // Check all rows
    for (int i = 0; i < 3; i++)
        if ((cell[i][0] == token)
            && (cell[i][1] == token)
            && (cell[i][2] == token)) {
            return true;
    }
    // Check all columns
    for (int j = 0; j < 3; j++)
        if ((cell[0][j] == token)
            && (cell[1][j] == token)
            && (cell[2][j] == token)) {
            return true;
    }
    // Check major diagonal
    if ((cell[0][0] == token)
        && (cell[1][1] == token)
        && (cell[2][2] == token)) {
        return true;
    }
    // Check subdiagonal
    if ((cell[0][2] == token)
        && (cell[1][1] == token)
        && (cell[2][0] == token)) {
        return true;
    }
```
/** All checked, but no winner */
return false;
}

public class TicTacToeClient extends JApplet implements Runnable, TicTacToeConstants {

private boolean myTurn = false;

private char myToken = ' ';

private char otherToken = ' ';

private Cell[][] cell = new Cell[3][3];

private JLabel jlblTitle = new JLabel();

private JLabel jlblStatus = new JLabel();

private int rowSelected;

private int columnSelected;

private DataInputStream fromServer;

private DataOutputStream toServer;

private boolean continueToPlay = true;

private boolean waiting = true;

private boolean isStandAlone = false;

private String host = "localhost";

@Override /** Initialize UI */
public void init() {
      JPanel p = new JPanel();
      p.setLayout(new GridLayout(3, 3, 0, 0));
}
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        p.add(cell[i][j] = new Cell(i, j));

// Set properties for labels and borders for labels and panel
p.setBorder(new LineBorder(Color.black, 1));
jlblTitle.setHorizontalAlignment(JLabel.CENTER);
jlblTitle.setFont(new Font("SansSerif", Font.BOLD, 16));
jlblStatus.setBorder(new LineBorder(Color.black, 1));

// Place the panel and the labels for the applet
add(jlblTitle, BorderLayout.NORTH);
add(p, BorderLayout.CENTER);
add(jlblStatus, BorderLayout.SOUTH);

// Connect to the server
connectToServer();

private void connectToServer() {
    try {
        // Create a socket to connect to the server
        Socket socket;
        if (isStandAlone)
            socket = new Socket(host, 8000);  // standalone
        else
            socket = new Socket(getCodeBase().getHost(), 8000);  // applet

        // Create an input stream to receive data from the server
        fromServer = new DataInputStream(socket.getInputStream());

        // Create an output stream to send data to the server
        toServer = new DataOutputStream(socket.getOutputStream());

        catch (Exception ex) {
            System.err.println(ex);
        }

        // Control the game on a separate thread
        Thread thread = new Thread(this);
        thread.start();
    }

    @Override
    public void run() {
        try {
            // Get notification from the server
            int player = fromServer.readInt();

            // Am I player 1 or 2?
            if (player == PLAYER1) {
                myToken = 'X';
                otherToken = 'O';
                jlblTitle.setText("Player 1 with token 'X'");
                jlblStatus.setText("Waiting for player 2 to join");

                // Receive startup notification from the server
                fromServer.readInt();  // Whatever read is ignored
// The other player has joined
jlblStatus.setText("Player 2 has joined. I start first");

// It is my turn
myTurn = true;
}
else if (player == PLAYER2) {
    myToken = 'O';
    otherToken = 'X';
    jlblTitle.setText("Player 2 with token 'O'");
    jlblStatus.setText("Waiting for player 1 to move");
}

// Continue to play
while (continueToPlay) {
    if (player == PLAYER1) {
        // Wait for player 1 to move
        waitForPlayerAction(); // Wait for player 1 to move
        sendMove(); // Send the move to the server
        receiveInfoFromServer(); // Receive info from the server
    }  
    else if (player == PLAYER2) {
        receiveInfoFromServer(); // Receive info from the server
        waitForPlayerAction(); // Wait for player 2 to move
        sendMove(); // Send player 2's move to the server
    }
}

捕获 (Exception ex) {
}

/** Wait for the player to mark a cell */
private void waitForPlayerAction() throws InterruptedException {
    while (waiting) {
        Thread.sleep(100);
    }
    waiting = true;
}

/** Send this player's move to the server */
private void sendMove() throws IOException {
    toServer.writeInt(rowSelected); // Send the selected row
    toServer.writeInt(columnSelected); // Send the selected column
}

/** Receive info from the server */
private void receiveInfoFromServer() throws IOException {
    // Receive game status
    int status = fromServer.readInt();
    if (status == PLAYER1_WON) {
        // Player 1 won, stop playing
        continueToPlay = false;
        if (myToken == 'X') {
            jlblStatus.setText("I won! (X)!");
        }
    } else if (myToken == 'O') {

}
jlblStatus.setText("Player 1 (X) has won!");
receiveMove();

else if (status == PLAYER2_WON) {
    // Player 2 won, stop playing
    continueToPlay = false;
    if (myToken == '0') {
        jlblStatus.setText("I won! (O)"奇);
    }
    else if (myToken == 'X') {
        jlblStatus.setText("Player 2 (O) has won!");
        receiveMove();
    }
}
else if (status == DRAW) {
    // No winner, game is over
    continueToPlay = false;
    jlblStatus.setText("Game is over, no winner!");
    if (myToken == '0') {
        receiveMove();
    }
}
else {
    receiveMove();
    jlblStatus.setText("My turn");
    myTurn = true; // It is my turn
}

private void receiveMove() throws IOException {
    // Get the other player's move
    int row = fromServer.readInt();
    int column = fromServer.readInt();
    cell[row][column].setToken(otherToken);
}

// An inner class for a cell
public class Cell extends JPanel {
    // Indicate the row and column of this cell in the board
    private int row;
    private int column;

    // Token used for this cell
    private char token = ' ';

    public Cell(int row, int column) {
        this.row = row;
        this.column = column;
        setBorder(new LineBorder(Color.black, 1)); // Set cell's border
        addMouseListener(new ClickListener()); // Register listener
    }

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

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.](image-url)
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

```java
@Override /** Initialize UI */
public void init() {
    is changed to

public TicTacToeClient() {
```

### 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](http://www.cs.armstrong.edu/liang/intro9e/test.html).

**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.

![Exercise33_1Client](image1)

(a)

![Exercise33_1Server](image2)

(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](image)

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](image)

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 (non-editable) 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`.

**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`.

![Figure 33.21](image1) The server and client send text to and receive text from each other.

![Figure 33.22](image2) The server starts in (a) with three clients in (b), (c), and (d).