CHAPTER

33

Networking

Objectives

- To explain the 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 send and receive objects on a network (§33.5).
- To develop an interactive tic-tac-toe game played on the Internet (§33.6).

33.1 Introduction

Key Point

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

To browse the Web or send an 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 then sends the request using the IP address.

domain name
domain name server

TCP
UDP

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 communication
packet-based communication

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.P, Networking Using Datagram Protocol.

33.2 Client/Server Computing

Key Point

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.

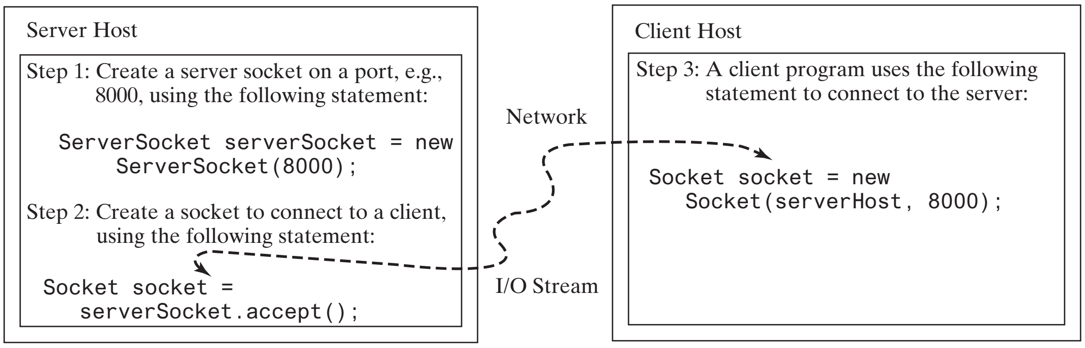
Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds. The client begins by attempting to establish a connection to the server. The server can accept or deny the connection. Once a connection is established, the client 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 the client. The statements needed to create sockets on a server and on a client are shown in Figure 33.1.

33.2.1 Server Sockets

server socket
port

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.



Step 1: Create a server socket on a port, e.g., 8000, using the following statement:

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

Step 2: Create a socket to connect to a client, using the following statement:

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

Step 3: A client program uses the following statement to connect to the server:

```
Socket socket = new
   Socket(serverHost, 8000);
```

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 a `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`

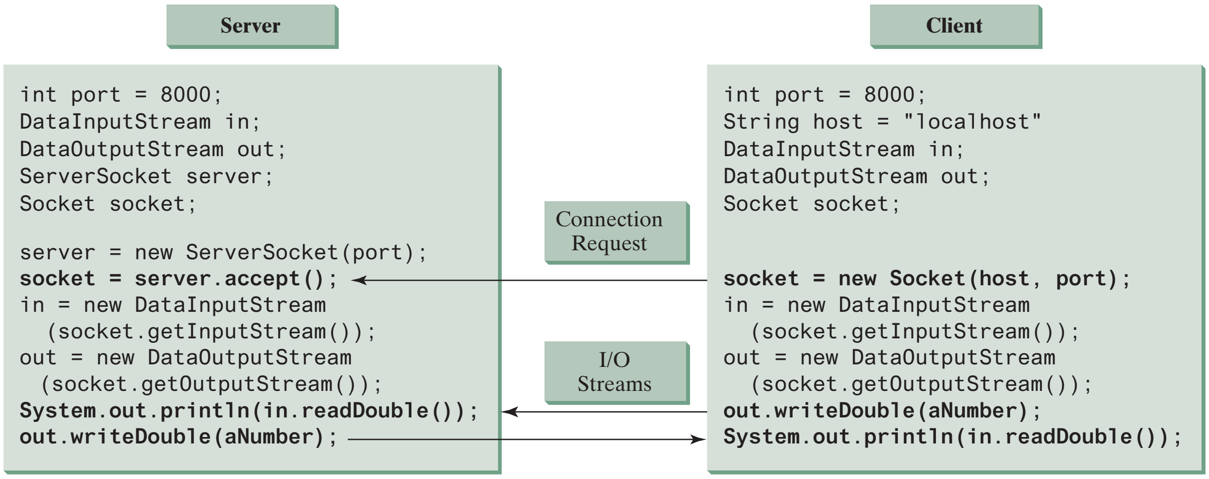
UnknownHostException

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



```
int port = 8000;
DataInputStream in;
DataOutputStream out;
ServerSocket server;
Socket socket;

server = new ServerSocket(port);
socket = server.accept();
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
System.out.println(in.readDouble());
out.writeDouble(aNumber);
```

```
int port = 8000;
String host = "localhost"
DataInputStream in;
DataOutputStream out;
Socket socket;


socket = new Socket(host, port);
in = new DataInputStream
  (socket.getInputStream());
out = new DataOutputStream
  (socket.getOutputStream());
out.writeDouble(aNumber);
System.out.println(in.readDouble());
```

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 `DataOutputstream 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).

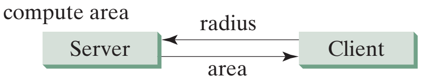


FIGURE 33.3 The client sends the radius to the server; the server computes the area and sends it to the client.

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.

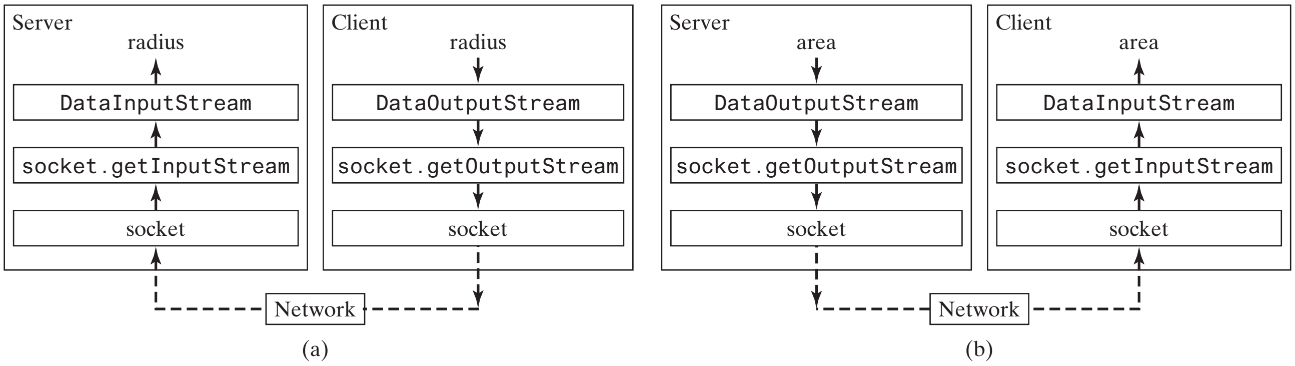
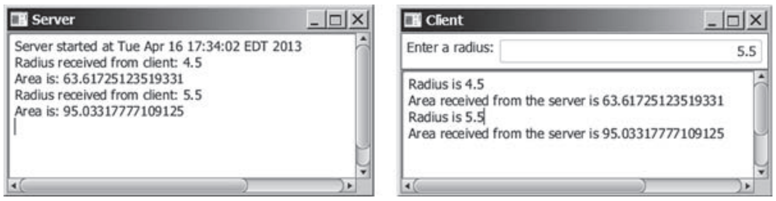


FIGURE 33.4 (a) The client sends the radius to the server. (b) The server sends the area to the client.



```
Server started at Tue Apr 16 17:34:02 EDT 2013
Radius received from client: 4.5
Area is: 63.61725123519331
Radius received from client: 5.5
Area is: 95.03317777109125
```

Enter a radius: 5.5

```
Radius is 4.5
Area received from the server is 63.61725123519331
Radius is 5.5
Area received from the server is 95.03317777109125
```

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.Date;
```

create server UI

server socket

update UI

connect client

input from client

output to client

read radius

write area

update UI

```java
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class Server extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Text area for displaying contents
    TextArea ta = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("Server"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread(() -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        Platform.runLater(() ->
          ta.appendText("Server started at " + new Date() + '\n'));

        // Listen for a connection request
        Socket socket = serverSocket.accept();

        // Create data input and output streams
        DataInputStream inputFromClient = new DataInputStream(
          socket.getInputStream());
        DataOutputStream outputToClient = new DataOutputStream(
          socket.getOutputStream());

        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);

          Platform.runLater(() -> {
            ta.appendText("Radius received from client: "
              + radius + '\n');
            ta.appendText("Area is: " + area + '\n');
          });
        }
      }
      catch(IOException ex) {
        ex.printStackTrace();
      }
    }).start();
 }
}
```

Listing 33.2 Client.java

```
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Client extends Application {
  // IO streams
  DataOutputStream toServer = null;
  DataInputStream fromServer = null;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Panel p to hold the label and text field
    BorderPane paneForTextField = new BorderPane();
    paneForTextField.setPadding(new Insets(5, 5, 5, 5));
    paneForTextField.setStyle("-fx-border-color: green");
    paneForTextField.setLeft(new Label("Enter a radius: "));

    TextField tf = new TextField();
    tf.setAlignment(Pos.BOTTOM_RIGHT);
    paneForTextField.setCenter(tf);

    BorderPane mainPane = new BorderPane();
    // Text area to display contents
    TextArea ta = new TextArea();
    mainPane.setCenter(new ScrollPane(ta));
    mainPane.setTop(paneForTextField);

    // Create a scene and place it in the stage
    Scene scene = new Scene(mainPane, 450, 200);
    primaryStage.setTitle("Client"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    tf.setOnAction(e -> {
      try {
        // Get the radius from the text field
        double radius = Double.parseDouble(tf.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
        ta.appendText("Radius is " + radius + "\n");
        ta.appendText("Area received from the server is "
          + area + '\n');
```

create UI

handle action event

read radius

write radius

read area

request connection

input from server

output to server

```
59        }
60        catch (IOException ex) {
61          System.err.println(ex);
62        }
63      });
64
65      try {
66        // Create a socket to connect to the server
67        Socket socket = new Socket("localhost", 8000);
68        // Socket socket = new Socket("130.254.204.36", 8000);
69        // Socket socket = new Socket("drake.Armstrong.edu", 8000);
70
71        // Create an input stream to receive data from the server
72        fromServer = new DataInputStream(socket.getInputStream());
73
74        // Create an output stream to send data to the server
75        toServer = new DataOutputStream(socket.getOutputStream());
76      }
77      catch (IOException ex) {
78        ta.appendText(ex.toString() + '\n');
79      }
80    }
81  }
```

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 26 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 the 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 `ServerSocket accept()` method takes time to execute. It is not appropriate to run this method in the JavaFX application thread. So, we place it in a separate thread (lines 23–59). The statements for updating GUI needs to run from the JavaFX application thread using the `Platform.runLater` method (lines 27–28, 49–53).

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 67 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 then restart the server.

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 67 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

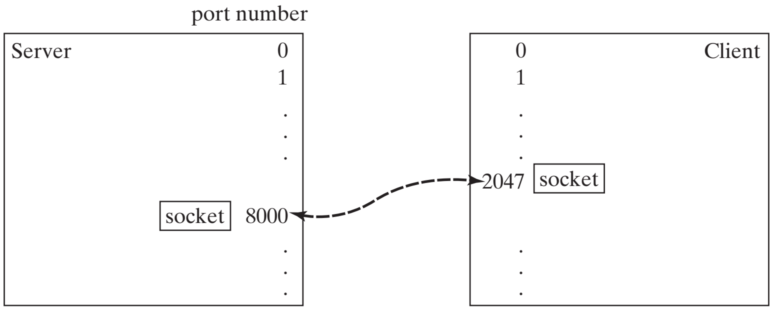


Figure 33.6 The JVM automatically chooses an available port to create a socket for the client.

To see the local port on the client, insert the following statement in line 70 in Client.java.

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

Check Point

33.2.1 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?

33.2.2 What are the differences between a server socket and a client socket?

33.2.3 How does a client program initiate a connection?

33.2.4 How does a server accept a connection?

33.2.5 How are data transferred between a client and a server?

33.3 The InetAddress Class

Key Point

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:

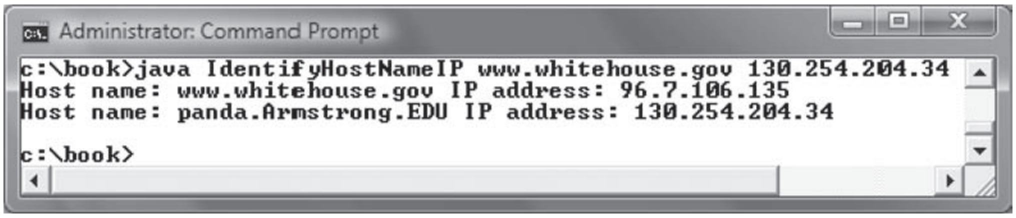
```
System.out.println("Client's host name is " +
  inetAddress.getHostName());
```

```
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 and 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.



```
c:\book>java IdentifyHostNameIP www.whitehouse.gov 130.254.204.34
Host name: www.whitehouse.gov IP address: 96.7.106.135
Host name: panda.Armstrong.EDU IP address: 130.254.204.34

c:\book>
```

FIGURE 33.7 The program identifies host names and IP addresses.

LISTING 33.3 IdentifyHostNameIP.java

get an `InetAddress`
get host name
get host IP

```
 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  }
```

Check Point

33.3.1 How do you obtain an instance of `InetAddress`?

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

33.4 Serving Multiple Clients

Key Point

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.

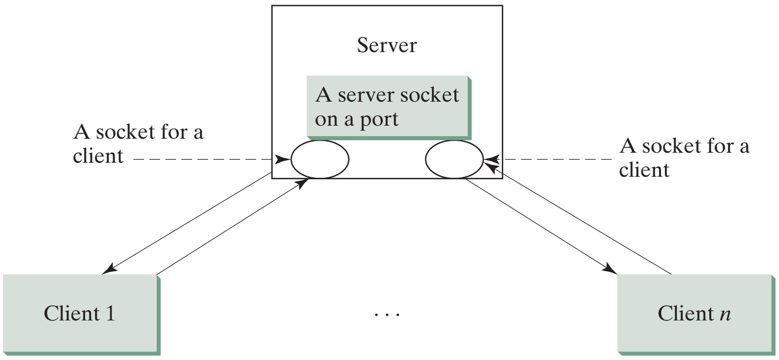
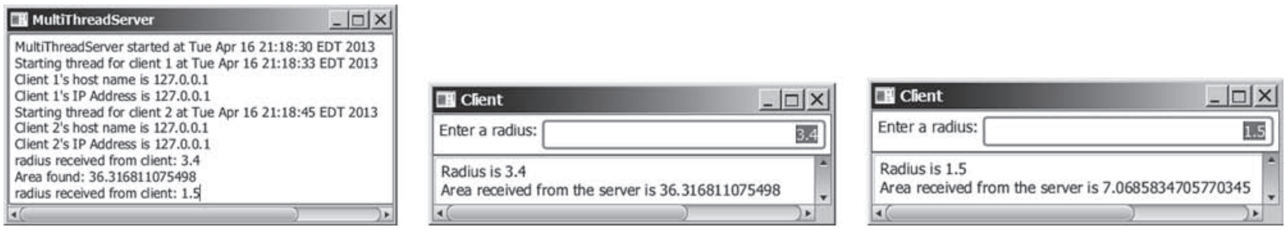


FIGURE 33.8 Multithreading enables a server to handle multiple independent clients.



```
MultiThreadServer started at Tue Apr 16 21:18:30 EDT 2013
Starting thread for client 1 at Tue Apr 16 21:18:33 EDT 2013
Client 1's host name is 127.0.0.1
Client 1's IP Address is 127.0.0.1
Starting thread for client 2 at Tue Apr 16 21:18:45 EDT 2013
Client 2's host name is 127.0.0.1
Client 2's IP Address is 127.0.0.1
radius received from client: 3.4
Area found: 36.316811075498
radius received from client: 1.5
```

Enter a radius: 3.4

```
Radius is 3.4
Area received from the server is 36.316811075498
```

Enter a radius: 1.5

```
Radius is 1.5
Area received from the server is 7.0685834705770345
```

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

LISTING 33.4 MultiThreadServer.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

```

```
public class MultiThreadServer extends Application {
  // Text area for displaying contents
  private TextArea ta = new TextArea();

  // Number a client
  private int clientNo = 0;

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(ta), 450, 200);
    primaryStage.setTitle("MultiThreadServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
        ServerSocket serverSocket = new ServerSocket(8000);
        ta.appendText("MultiThreadServer started at "
          + new Date() + '\n');

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

          // Increment clientNo
          clientNo++;

          Platform.runLater( () -> {
            // Display the client number
            ta.appendText("Starting thread for client " + clientNo +
              " at " + new Date() + '\n');

            // Find the client's host name, and IP address
            InetAddress inetAddress = socket.getInetAddress();
            ta.appendText("Client " + clientNo + "'s host name is "
              + inetAddress.getHostName() + "\n");
            ta.appendText("Client " + clientNo + "'s IP Address is"
              + inetAddress.getHostAddress() + "\n");
          });

          // Create and start a new thread for the connection
          new Thread(new HandleAClient(socket)).start();
        }
      }
      catch(IOException ex) {
        System.err.println(ex);
      }
    }).start();
  }

  // Define the thread class for handling new connection
  class HandleAClient implements Runnable {
    private Socket socket; // A connected socket

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

```

server socket

connect client

update GUI

network information

create task

start thread

task class

```
 72      /** Run a thread */
 73      public void run() {
 74        try {
 75          // Create data input and output streams
 76          DataInputStream inputFromClient = new DataInputStream(
 77            socket.getInputStream());
 78          DataOutputStream outputToClient = new DataOutputStream(
 79            socket.getOutputStream());
 80
 81          // Continuously serve the client
 82          while (true) {
 83            // Receive radius from the client
 84            double radius = inputFromClient.readDouble();
 85
 86            // Compute area
 87            double area = radius * radius * Math.PI;
 88
 89            // Send area back to the client
 90            outputToClient.writeDouble(area);
 91
 92            Platform.runLater(() -> {
 93              ta.appendText("radius received from client: " +
 94                radius + '\n');
 95              ta.appendText("Area found: " + area + '\n');
 96            });
 97          }
 98        }
 99        catch(IOException ex) {
100          ex.printStackTrace();
101        }
102      }
103    }
104  }
```

I/O

update GUI

The server creates a server socket at port 8000 (line 29) 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 54). It then waits for another connection in an infinite `while` loop (lines 33–55).

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.

33.4.1 How do you make a server serve multiple clients?

Check Point

33.5 Sending and Receiving Objects

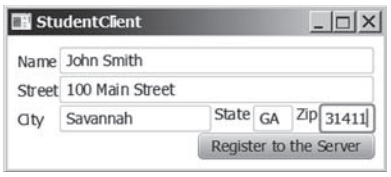
Key Point

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 33.5), StudentClient.java (Listing 33.6), and StudentServer.java (Listing 33.7). The client program collects student information from the client and sends it to a server, as shown in Figure 33.10.

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.



Name John Smith
Street 100 Main Street
City Savannah State GA Zip 31411

Figure 33.10 The client sends the student information in an object to the server.

Listing 33.5 StudentAddress.java

serialized

```java
public class StudentAddress implements java.io.Serializable {
  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 33.11. The client uses 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 33.6 and 33.7.

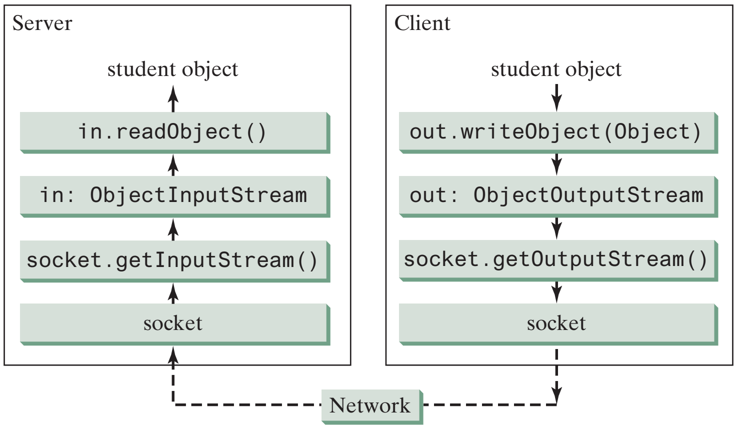


FIGURE 33.11 The client sends a `StudentAddress` object to the server.

LISTING 33.6 StudentClient.java

```java
import java.io.*;
import java.net.*;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class StudentClient extends Application {
  private TextField tfName = new TextField();
  private TextField tfStreet = new TextField();
  private TextField tfCity = new TextField();
  private TextField tfState = new TextField();
  private TextField tfZip = new TextField();

  // Button for sending a student to the server
  private Button btRegister = new Button("Register to the Server");

  // Host name or ip
  String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    GridPane pane = new GridPane();
    pane.add(new Label("Name"), 0, 0);
    pane.add(tfName, 1, 0);
    pane.add(new Label("Street"), 0, 1);
    pane.add(tfStreet, 1, 1);
    pane.add(new Label("City"), 0, 2);

```

create UI

```java
    HBox hBox = new HBox(2);
    pane.add(hBox, 1, 2);
    hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
      new Label("Zip"), tfZip);
    pane.add(btRegister, 1, 3);
    GridPane.setHalignment(btRegister, HPos.RIGHT);

    pane.setAlignment(Pos.CENTER);
    tfName.setPrefColumnCount(15);
    tfStreet.setPrefColumnCount(15);
    tfCity.setPrefColumnCount(10);
    tfState.setPrefColumnCount(2);
    tfZip.setPrefColumnCount(3);

```

register listener

```java
    btRegister.setOnAction(new ButtonListener());

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 450, 200);
    primaryStage.setTitle("StudentClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Handle button action */
  private class ButtonListener implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
      try {
        // Establish connection with the server
```

server socket

```java
        Socket socket = new Socket(host, 8000);

        // Create an output stream to the server
```

output stream

```java
        ObjectOutputStream toServer =
          new ObjectOutputStream(socket.getOutputStream());

        // Get text field
        String name = tfName.getText().trim();
        String street = tfStreet.getText().trim();
        String city = tfCity.getText().trim();
        String state = tfState.getText().trim();
        String zip = tfZip.getText().trim();

        // Create a Student object and send to the server
        StudentAddress s =
          new StudentAddress(name, street, city, state, zip);
```

send to server

```java
        toServer.writeObject(s);
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Listing 33.7 StudentServer.java

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

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     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 }
```

server socket

output to file

connect to client

input stream

get from client

write to file

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 67), creates an `ObjectOutputStream` on the output stream of the socket (lines 70 and 71), and invokes the `writeObject` method to send the `StudentAddress` object to the server through the object output stream (line 83).

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 and 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).

Check Point

33.5.1 How does a server receive connection from a client? How does a client connect to a server?

33.5.2 How do you find the host name of a client program from the server?

33.5.3 How do you send and receive an object?

33.6 Case Study: Distributed Tic-Tac-Toe Games

Key Point

This section develops a program that enables two players to play the tic-tac-toe game on the Internet.

In Section 16.12, Case Study: Developing a Tic-Tac-Toe Game, you developed a program 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 33.12.

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 33.13.

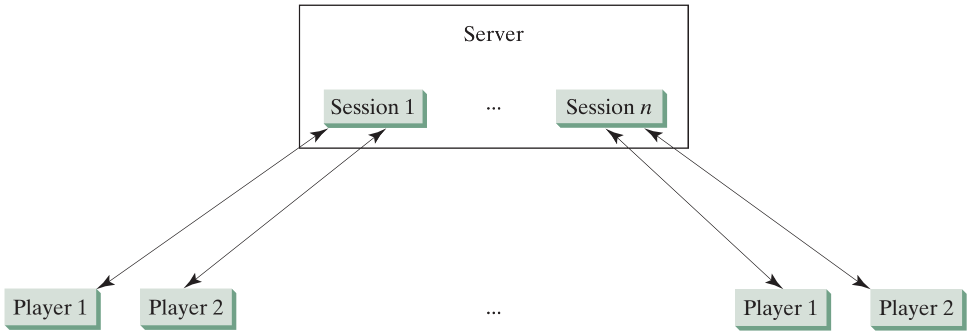


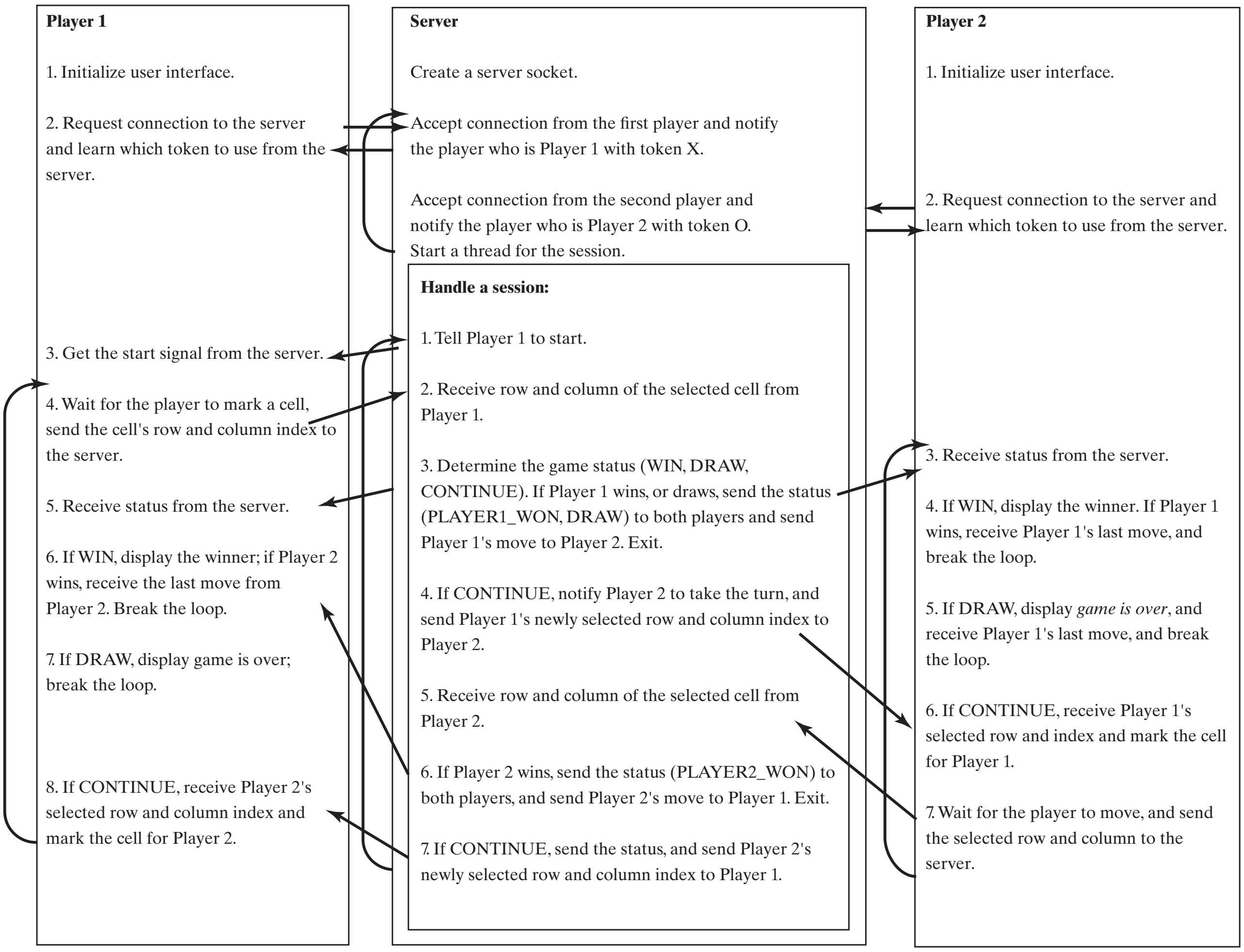
FIGURE 33.12 The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

The server does not have to be a graphical component, but creating it in a GUI in which game information can be viewed is user friendly. You can create a scroll pane to hold a text area in the GUI 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 Listing 16.13. 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 33.9.
- `HandleASession` facilitates the game for two players. This class is defined in Listing 33.9, TicTacToeServer.java.



Player 1

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Get the start signal from the server.

4. Wait for the player to mark a cell, send the cell's row and column index to the server.

5. Receive status from the server.

6. If WIN, display the winner; if Player 2 wins, receive the last move from Player 2. Break the loop.

7. If DRAW, display game is over; break the loop.

8. If CONTINUE, receive Player 2's selected row and column index and mark the cell for Player 2.

Server

Create a server socket.

Accept connection from the first player and notify the player who is Player 1 with token X.

Accept connection from the second player and notify the player who is Player 2 with token O. Start a thread for the session.

Handle a session:

1. Tell Player 1 to start.

2. Receive row and column of the selected cell from Player 1.

3. Determine the game status (WIN, DRAW, CONTINUE). If Player 1 wins, or draws, send the status (PLAYER1_WON, DRAW) to both players and send Player 1's move to Player 2. Exit.

4. If CONTINUE, notify Player 2 to take the turn, and send Player 1's newly selected row and column index to Player 2.

5. Receive row and column of the selected cell from Player 2.

6. If Player 2 wins, send the status (PLAYER2_WON) to both players, and send Player 2's move to Player 1. Exit.

7. If CONTINUE, send the status, and send Player 2's newly selected row and column index to Player 1.

Player 2

1. Initialize user interface.

2. Request connection to the server and learn which token to use from the server.

3. Receive status from the server.

4. If WIN, display the winner. If Player 1 wins, receive Player 1's last move, and break the loop.

5. If DRAW, display *game is over*, and receive Player 1's last move, and break the loop.

6. If CONTINUE, receive Player 1's selected row and index and mark the cell for Player 1.

7. Wait for the player to move, and send the selected row and column to the server.

FIGURE 33.13 The server starts a thread to facilitate communications between the two players.

- `TicTacToeClient` models a player in Listing 33.10.
- `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 33.8.

The relationships of these classes are shown in Figure 33.14.

LISTING 33.8 TicTacToeConstants.java

```java
public interface TicTacToeConstants {
  public static int PLAYER1 = 1; // Indicate player 1
  public static int PLAYER2 = 2; // Indicate player 2
  public static int PLAYER1_WON = 1; // Indicate player 1 won
  public static int PLAYER2_WON = 2; // Indicate player 2 won
  public static int DRAW = 3; // Indicate a draw
  public static int CONTINUE = 4; // Indicate to continue
}
```

LISTING 33.9 TicTacToeServer.java

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

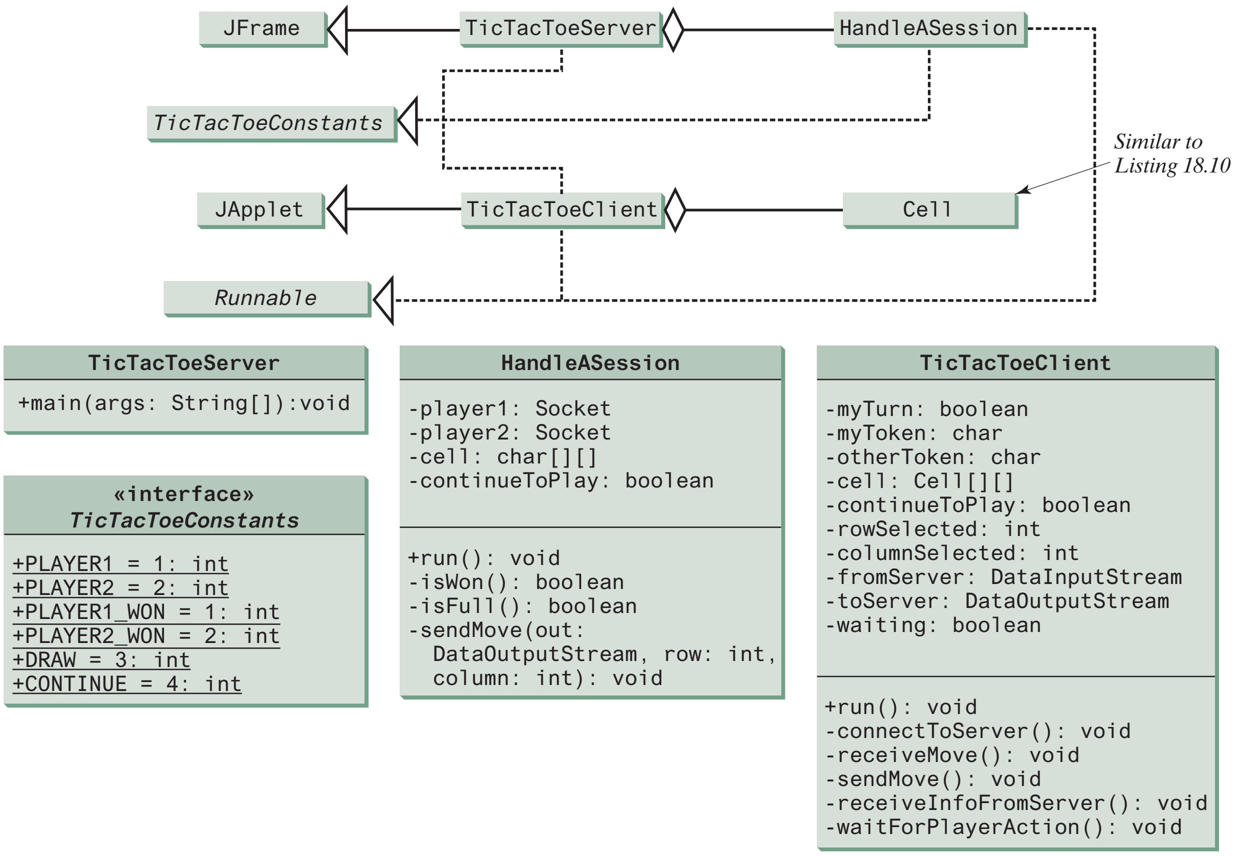


Figure 33.14 TicTacToeServer creates an instance of HandleASession for each session of two players. TicTacToeClient creates nine cells in the UI.

```
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.stage.Stage;

public class TicTacToeServer extends Application
    implements TicTacToeConstants {
  private int sessionNo = 1; // Number a session

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    TextArea taLog = new TextArea();

    // Create a scene and place it in the stage
    Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
    primaryStage.setTitle("TicTacToeServer"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    new Thread( () -> {
      try {
        // Create a server socket
```

create UI

```
      ServerSocket serverSocket = new ServerSocket(8000);
      Platform.runLater(() -> taLog.appendText(new Date() +
        ": Server started at socket 8000\n"));

      // Ready to create a session for every two players
      while (true) {
        Platform.runLater(() -> taLog.appendText(new Date() +
          ": Wait for players to join session " + sessionNo + '\n'));

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

        Platform.runLater(() -> {
          taLog.appendText(new Date() + ": Player 1 joined session "
            + sessionNo + '\n');
          taLog.appendText("Player 1's IP address" +
            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();

        Platform.runLater(() -> {
          taLog.appendText(new Date() +
            ": Player 2 joined session " + sessionNo + '\n');
          taLog.appendText("Player 2's IP address" +
            player2.getInetAddress().getHostAddress() + '\n');
        });

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

        // Display this session and increment session number
        Platform.runLater(() ->
          taLog.appendText(new Date() +
            ": Start a thread for session " + sessionNo++ + '\n'));

        // Launch a new thread for this session of two players
        new Thread(new HandleASession(player1, player2)).start();
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }).start();
}

// 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;
```

server socket

connect to client

to player1

connect to client

to player2

a session for two players

```
    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] = ' ';
    }

    /** Implement the run() method for the thread */
    public void run() {
      try {
        // Create data input and output streams
        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);

```

IO streams

X won?

Is full?

```
            // 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
            sendMove(toPlayer1, row, column);
          }
        }
      }
      catch(IOException ex) {
        ex.printStackTrace();
      }
    }

    /** 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++)
```

O won?

send a move

```
        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;
    }
  }
}
```

Listing 33.10 TicTacToeClient.java

```
import java.io.*;
import java.net.*;
import java.util.Date;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class TicTacToeClient extends Application
    implements TicTacToeConstants {
  // Indicate whether the player has the turn
  private boolean myTurn = false;

  // Indicate the token for the player
  private char myToken = ' ';

  // Indicate the token for the other player
  private char otherToken = ' ';

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

  // Create and initialize a title label
  private Label lblTitle = new Label();

  // Create and initialize a status label
  private Label lblStatus = new Label();

  // Indicate selected row and column by the current move
  private int rowSelected;
  private int columnSelected;

  // Input and output streams from/to server
  private DataInputStream fromServer;
  private DataOutputStream toServer;

  // Continue to play?
  private boolean continueToPlay = true;

  // Wait for the player to mark a cell
  private boolean waiting = true;

  // Host name or ip
  private String host = "localhost";

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Pane to hold cell
    GridPane pane = new GridPane();
    for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
        pane.add(cell[i][j] = new Cell(i, j), j, i);

    BorderPane borderPane = new BorderPane();
    borderPane.setTop(lblTitle);
    borderPane.setCenter(pane);
    borderPane.setBottom(lblStatus);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 320, 350);
    primaryStage.setTitle("TicTacToeClient"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    // Connect to the server
    connectToServer();
  }

  private void connectToServer() {
    try {
      // Create a socket to connect to the server
      Socket socket = new Socket(host, 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 (Exception ex) {
      ex.printStackTrace();
    }
```

create UI

connect to server

input from server

output to server

```

    // Control the game on a separate thread
    new Thread(() -> {
      try {
        // Get notification from the server
        int player = fromServer.readInt();

        // Am I player 1 or 2?
        if (player == PLAYER1) {
          myToken = 'X';
          otherToken = 'O';
          Platform.runLater(() -> {
            lblTitle.setText("Player 1 with token 'X'");
            lblStatus.setText("Waiting for player 2 to join");
          });

          // Receive startup notification from the server
          fromServer.readInt(); // Whatever read is ignored

          // The other player has joined
          Platform.runLater(() ->
            lblStatus.setText("Player 2 has joined. I start first"));

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

        // Continue to play
        while (continueToPlay) {
          if (player == PLAYER1) {
            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
          }
        }
      }
      catch (Exception ex) {
        ex.printStackTrace();
      }
    }).start();
  }

  /** 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') {
        Platform.runLater(() -> lblStatus.setText("I won! (X)"));
      }
      else if (myToken == 'O') {
        Platform.runLater(() ->
          lblStatus.setText("Player 1 (X) has won!"));
        receiveMove();
      }
    }
    else if (status == PLAYER2_WON) {
      // Player 2 won, stop playing
      continueToPlay = false;
      if (myToken == 'O') {
        Platform.runLater(() -> lblStatus.setText("I won! (O)"));
      }
      else if (myToken == 'X') {
        Platform.runLater(() ->
          lblStatus.setText("Player 2 (O) has won!"));
        receiveMove();
      }
    }
    else if (status == DRAW) {
      // No winner, game is over
      continueToPlay = false;
      Platform.runLater(() ->
        lblStatus.setText("Game is over, no winner!"));

      if (myToken == 'O') {
        receiveMove();
      }
    }
    else {
      receiveMove();
      Platform.runLater(() -> lblStatus.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();
    Platform.runLater(() -> cell[row][column].setToken(otherToken));
  }

  // An inner class for a cell
```

model a cell

```
  public class Cell extends Pane {
    // 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;
      this.setPrefSize(2000, 2000); // What happens without this?
      setStyle("-fx-border-color: black"); // Set cell's border
```

register listener

```
      this.setOnMouseClicked(e -> handleMouseClick());
    }

    /** Return token */
    public char getToken() {
      return token;
    }

    /** Set a new token */
    public void setToken(char c) {
      token = c;
      repaint();
    }

    protected void repaint() {
      if (token == 'X') {
```

draw X

```
        Line line1 = new Line(10, 10,
          this.getWidth() - 10, this.getHeight() - 10);
        line1.endXProperty().bind(this.widthProperty().subtract(10));
        line1.endYProperty().bind(this.heightProperty().subtract(10));
        Line line2 = new Line(10, this.getHeight() - 10,
          this.getWidth() - 10, 10);
        line2.startYProperty().bind(
          this.heightProperty().subtract(10));
        line2.endXProperty().bind(this.widthProperty().subtract(10));

        // Add the lines to the pane
        this.getChildren().addAll(line1, line2);
      }
      else if (token == 'O') {
```

draw O

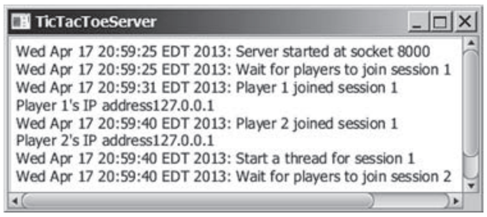
```
        Ellipse ellipse = new Ellipse(this.getWidth() / 2,
          this.getHeight() / 2, this.getWidth() /  2 - 10,
          this.getHeight() / 2 - 10);
        ellipse.centerXProperty().bind(
          this.widthProperty().divide(2));
        ellipse.centerYProperty().bind(
          this.heightProperty().divide(2));
        ellipse.radiusXProperty().bind(
          this.widthProperty().divide(2).subtract(10));
        ellipse.radiusYProperty().bind(
          this.heightProperty().divide(2).subtract(10));
        ellipse.setStroke(Color.BLACK);
        ellipse.setFill(Color.WHITE);

        getChildren().add(ellipse); // Add the ellipse to the pane
      }
    }

    /* Handle a mouse click event */
    private void handleMouseClick() {
      // If cell is not occupied and the player has the turn
      if (token == ' ' && myTurn) {
        setToken(myToken); // Set the player's token in the cell
        myTurn = false;
        rowSelected = row;
        columnSelected = column;
        lblStatus.setText("Waiting for the other player to move");
        waiting = false; // Just completed a successful move
      }
    }
  }
}
```

mouse clicked handler

The server can serve any number of sessions simultaneously. Each session takes care of two players. The client can be deployed to run as a Java applet. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 33.15 and 33.16 show sample runs of the server and the clients.



```
Wed Apr 17 20:59:25 EDT 2013: Server started at socket 8000
Wed Apr 17 20:59:25 EDT 2013: Wait for players to join session 1
Wed Apr 17 20:59:31 EDT 2013: Player 1 joined session 1
Player 1's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Player 2 joined session 1
Player 2's IP address127.0.0.1
Wed Apr 17 20:59:40 EDT 2013: Start a thread for session 1
Wed Apr 17 20:59:40 EDT 2013: Wait for players to join session 2
```

FIGURE 33.15 `TicTacToeServer` accepts connection requests and creates sessions to serve pairs of players.

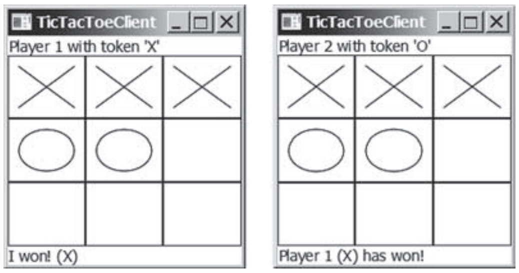


FIGURE 33.16 `TicTacToeClient` can run as an applet or standalone.

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.

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.

Check Point

33.6.1 What would happen if the preferred size for a cell is not set in line 227 in Listing 33.10?

33.6.2 If a player does not have the turn but clicks on an empty cell, what will the client program in Listing 33.10 do?

Key Terms

client socket	33-3	packet-based communication	33-2
domain name	33-2	server socket	33-2
domain name server	33-2	socket	33-2
localhost	33-3	stream-based communication	33-2
IP address	33-2	TCP	33-2
port	33-2	UDP	33-2

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.

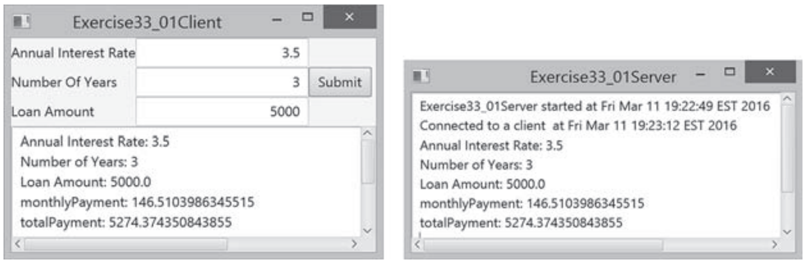
Quiz

Answer the quiz for this chapter online at book Companion Website.

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 33.17a). The server computes monthly payment and total payment, and sends them back to the client (see Figure 33.17b). Name the client Exercise33_01Client and the server Exercise33_01Server.



Annual Interest Rate 3.5
Number Of Years 3
Loan Amount 5000

```
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

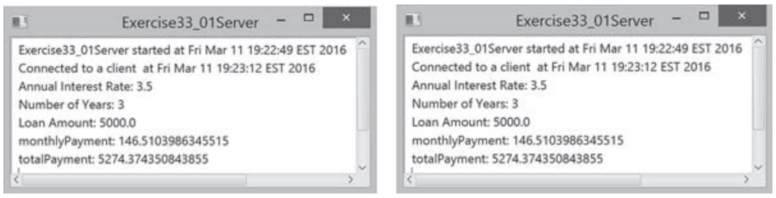
(a)

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(b)

Figure 33.17 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** (*BMI server*) Write a server for a client. The client sends the weight and height for a person to the server (see Figure 33.18a). The server computes BMI (Body Mass Index) and sends back to the client a string that reports the BMI (see Figure 33.18b). See Section 3.8 for computing BMI. Name the client Exercise33_02Client and the server Exercise33_02Server.



```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(a)

```
Exercise33_01Server started at Fri Mar 11 19:22:49 EST 2016
Connected to a client  at Fri Mar 11 19:23:12 EST 2016
Annual Interest Rate: 3.5
Number of Years: 3
Loan Amount: 5000.0
monthlyPayment: 146.5103986345515
totalPayment: 5274.374350843855
```

(b)

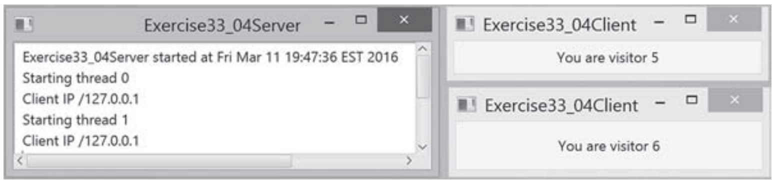
Figure 33.18 The client in (a) sends the weight and height of a person to the server and receives the BMI from the server in (b).

Sections 33.3 and 33.4

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

Section 33.5

33.4 (*Count clients*) Write a server that tracks the number of the clients connected to the server. When a new connection is established, the count is incremented by 1. The count is stored using a random-access file. Write a client program that receives the count from the server and displays a message, such as "You are visitor number 11", as shown in Figure 33.19. Name the client Exercise33_04Client and the server Exercise33_04Server.



Exercise33_04Server

```
Exercise33_04Server started at Fri Mar 11 19:47:36 EST 2016
Starting thread 0
Client IP /127.0.0.1
Starting thread 1
Client IP /127.0.0.1
```

Exercise33_04Client

You are visitor 5

Exercise33_04Client

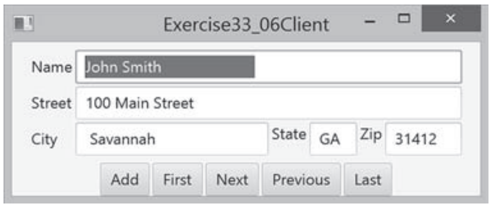
You are visitor 6

FIGURE 33.19 The client displays how many times the server has been accessed. The server stores the count.

33.5 (*Send loan information in an object*) Revise Exercise 33.1 for the client to send a loan object that contains annual interest rate, number of years, and loan amount and for the server to send the monthly payment and total payment.

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.



Exercise33_06Client

Name: John Smith
Street: 100 Main Street
City: Savannah State: GA Zip: 31412

FIGURE 33.20 You can view and add an address.

- Use the `StudentAddress` class defined in Listing 33.5 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.

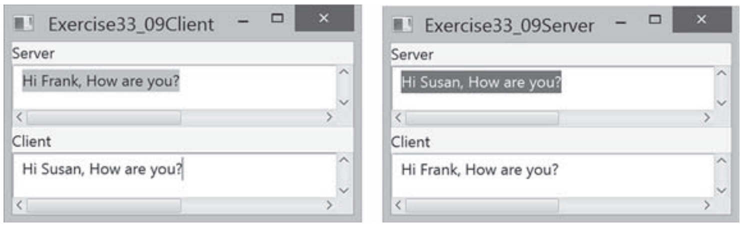
Name the client Exercise33_06Client and the server Exercise33_6Server.

***33.7** (*Transfer last 100 numbers in an array*) Programming Exercise 22.12 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_07Server and the client program Exercise33_07Client. Assume 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_08Server and the client program Exercise33_08Client. Assume 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 (see Figure 33.21a) and the other as the client (see 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 displaying 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_09Client and the server Exercise33_09Server.



Hi Frank, How are you?

Hi Susan, How are you?

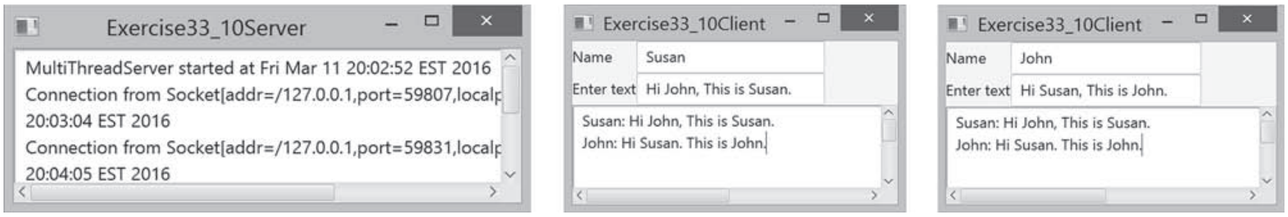
Hi Susan, How are you?

Hi Frank, How are you?

(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.



```
MultiThreadServer started at Fri Mar 11 20:02:52 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59807,localp
20:03:04 EST 2016
Connection from Socket[addr=/127.0.0.1,port=59831,localp
20:04:05 EST 2016
```

Name: Susan
Enter text: Hi John, This is Susan.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

Name: John
Enter text: Hi Susan, This is John.

```
Susan: Hi John, This is Susan.
John: Hi Susan. This is John.
```

(a) (b) (c)

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