

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



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).

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.

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.P, 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 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

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

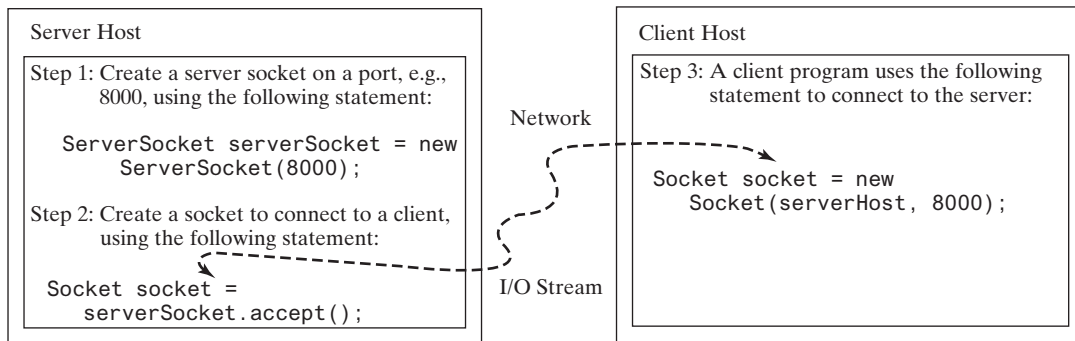


FIGURE 33.1 The server creates a server socket and, once a connection to a client is established, connects to the client with a client socket.

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

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



Note

Attempting to create a server socket on a port already in use would cause 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.

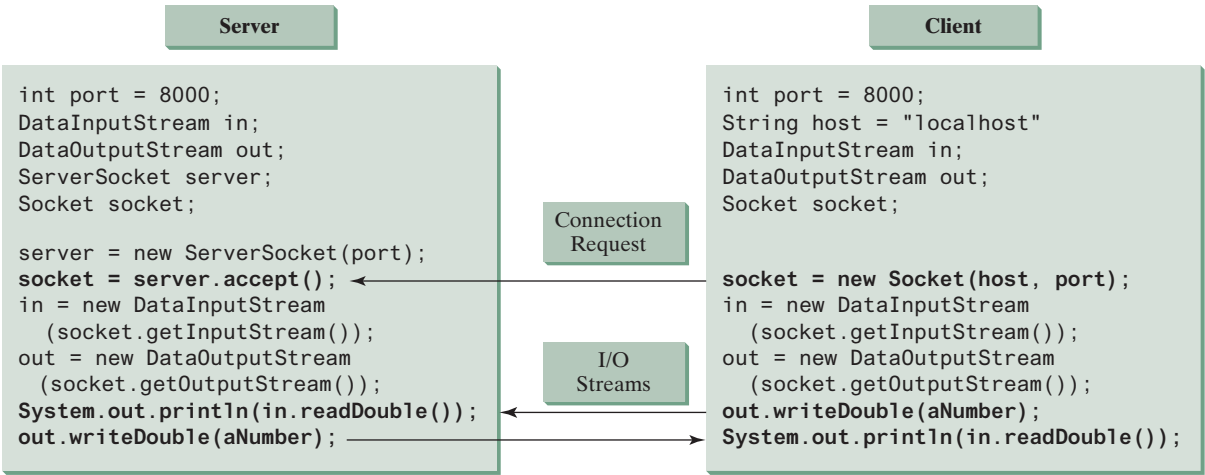


FIGURE 33.2 The server and client exchange data through I/O streams on top of the socket.

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).

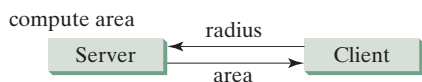


FIGURE 33.3 The client sends the radius to the server; the server computes the area and sends it to the client.

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.

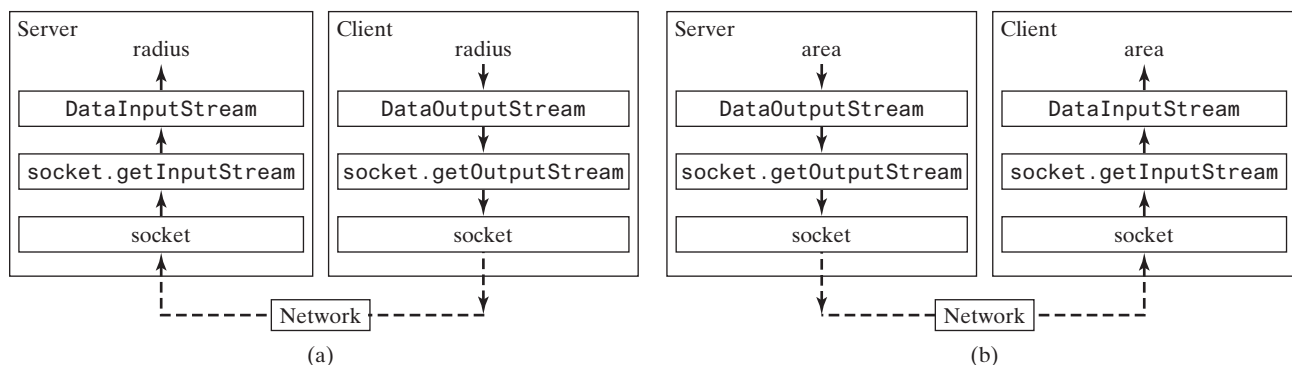


FIGURE 33.4 (a) The client sends the radius to the server. (b) The server sends the area to the client.

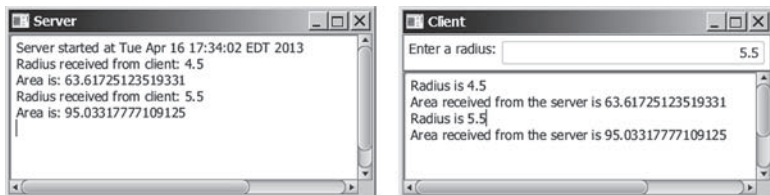


FIGURE 33.5 The client sends the radius to the server. The server receives it, computes the area, and sends the area to the client.

LISTING 33.1 Server.java

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.Date;

```

33-6 Chapter 33 Networking

```

4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.ScrollPane;
8  import javafx.scene.control.TextArea;
9  import javafx.stage.Stage;
10
11  public class Server extends Application {
12      @Override // Override the start method in the Application class
13      public void start(Stage primaryStage) {
14          // Text area for displaying contents
15          TextArea ta = new TextArea();
16
17          // Create a scene and place it in the stage
18          Scene scene = new Scene(new ScrollPane(ta), 450, 200);
19          primaryStage.setTitle("Server"); // Set the stage title
20          primaryStage.setScene(scene); // Place the scene in the stage
21          primaryStage.show(); // Display the stage
22
23          new Thread(() -> {
24              try {
25                  // Create a server socket
26                  ServerSocket serverSocket = new ServerSocket(8000);
27                  Platform.runLater(() ->
28                      ta.appendText("Server started at " + new Date() + '\n'));
29
30                  // Listen for a connection request
31                  Socket socket = serverSocket.accept();
32
33                  // Create data input and output streams
34                  DataInputStream inputFromClient = new DataInputStream(
35                      socket.getInputStream());
36                  DataOutputStream outputToClient = new DataOutputStream(
37                      socket.getOutputStream());
38
39                  while (true) {
40                      // Receive radius from the client
41                      double radius = inputFromClient.readDouble();
42
43                      // Compute area
44                      double area = radius * radius * Math.PI;
45
46                      // Send area back to the client
47                      outputToClient.writeDouble(area);
48
49                      Platform.runLater(() -> {
50                          ta.appendText("Radius received from client: "
51                              + radius + '\n');
52                          ta.appendText("Area is: " + area + '\n');
53                      });
54                  }
55              }
56              catch(IOException ex) {
57                  ex.printStackTrace();
58              }
59          }).start();
60      }
61  }
```

create server UI

server socket

update UI

connect client

input from client

output to client

read radius

write area

update UI

LISTING 33.2 Client.java

```

1  import java.io.*;
2  import java.net.*;
3  import javafx.application.Application;
4  import javafx.geometry.Insets;
5  import javafx.geometry.Pos;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.ScrollPane;
9  import javafx.scene.control.TextArea;
10 import javafx.scene.control.TextField;
11 import javafx.scene.layout.BorderPane;
12 import javafx.stage.Stage;
13
14 public class Client extends Application {
15     // IO streams
16     DataOutputStream toServer = null;
17     DataInputStream fromServer = null;
18
19     @Override // Override the start method in the Application class
20     public void start(Stage primaryStage) {
21         // Panel p to hold the label and text field
22         BorderPane paneForTextField = new BorderPane();           create UI
23         paneForTextField.setPadding(new Insets(5, 5, 5, 5));
24         paneForTextField.setStyle("-fx-border-color: green");
25         paneForTextField.setLeft(new Label("Enter a radius: "));
26
27         TextField tf = new TextField();
28         tf.setAlignment(Pos.BOTTOM_RIGHT);
29         paneForTextField.setCenter(tf);
30
31         BorderPane mainPane = new BorderPane();
32         // Text area to display contents
33         TextArea ta = new TextArea();
34         mainPane.setCenter(new ScrollPane(ta));
35         mainPane.setTop(paneForTextField);
36
37         // Create a scene and place it in the stage
38         Scene scene = new Scene(mainPane, 450, 200);
39         primaryStage.setTitle("Client"); // Set the stage title
40         primaryStage.setScene(scene); // Place the scene in the stage
41         primaryStage.show(); // Display the stage
42
43         tf.setOnAction(e -> {                                     handle action event
44             try {
45                 // Get the radius from the text field
46                 double radius = Double.parseDouble(tf.getText().trim());   read radius
47
48                 // Send the radius to the server
49                 toServer.writeDouble(radius);                               write radius
50                 toServer.flush();
51
52                 // Get area from the server
53                 double area = fromServer.readDouble();                     read area
54
55                 // Display to the text area
56                 ta.appendText("Radius is " + radius + "\n");
57                 ta.appendText("Area received from the server is "
58                     + area + '\n');

```



```

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 }

```

request connection

input from server

output to server

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.


```
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.



FIGURE 33.7 The program identifies host names and IP addresses.

LISTING 33.3 IdentifyHostNameIP.java

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

get an `InetAddress`
get host name
get host IP



- 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

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



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

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

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

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

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

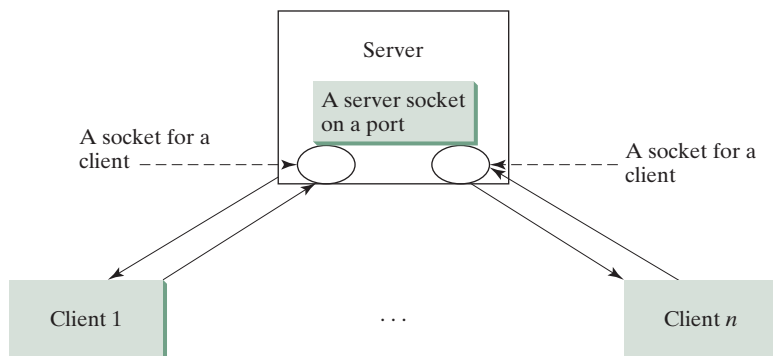


FIGURE 33.8 Multithreading enables a server to handle multiple independent clients.

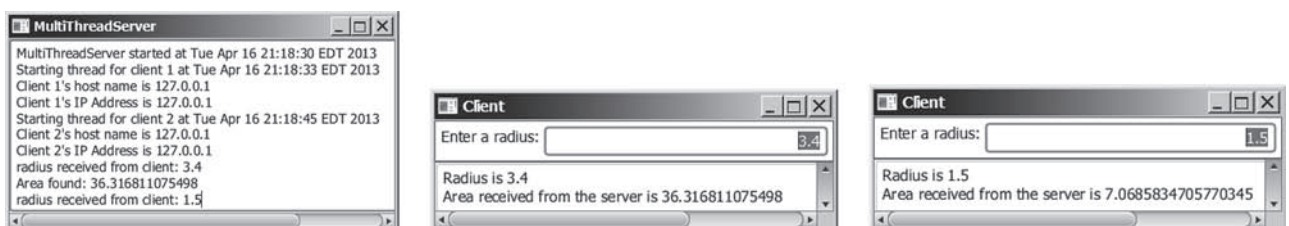


FIGURE 33.9 The server spawns a thread in order to serve a client.

LISTING 33.4 MultiThreadServer.java

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.Date;
4 import javafx.application.Application;
5 import javafx.application.Platform;
6 import javafx.scene.Scene;
7 import javafx.scene.control.ScrollPane;
8 import javafx.scene.control.TextArea;
9 import javafx.stage.Stage;
10
```

33-12 Chapter 33 Networking

```
11 public class MultiThreadServer extends Application {
12     // Text area for displaying contents
13     private TextArea ta = new TextArea();
14
15     // Number a client
16     private int clientNo = 0;
17
18     @Override // Override the start method in the Application class
19     public void start(Stage primaryStage) {
20         // Create a scene and place it in the stage
21         Scene scene = new Scene(new ScrollPane(ta), 450, 200);
22         primaryStage.setTitle("MultiThreadServer"); // Set the stage title
23         primaryStage.setScene(scene); // Place the scene in the stage
24         primaryStage.show(); // Display the stage
25
26         new Thread( () -> {
27             try {
28                 // Create a server socket
29                 ServerSocket serverSocket = new ServerSocket(8000);
30                 ta.appendText("MultiThreadServer started at "
31                     + new Date() + '\n');
32
33                 while (true) {
34                     // Listen for a new connection request
35                     Socket socket = serverSocket.accept();
36
37                     // Increment clientNo
38                     clientNo++;
39
40                     Platform.runLater( () -> {
41                         // Display the client number
42                         ta.appendText("Starting thread for client " + clientNo +
43                             " at " + new Date() + '\n');
44
45                         // Find the client's host name, and IP address
46                         InetAddress inetAddress = socket.getInetAddress();
47                         ta.appendText("Client " + clientNo + "'s host name is "
48                             + inetAddress.getHostName() + '\n');
49                         ta.appendText("Client " + clientNo + "'s IP Address is"
50                             + inetAddress.getHostAddress() + '\n');
51                     });
52
53                     // Create and start a new thread for the connection
54                     new Thread(new HandleAClient(socket)).start();
55                 }
56             }
57             catch(IOException ex) {
58                 System.err.println(ex);
59             }
60         }).start();
61     }
62
63     // Define the thread class for handling new connection
64     class HandleAClient implements Runnable {
65         private Socket socket; // A connected socket
66
67         /** Construct a thread */
68         public HandleAClient(Socket socket) {
69             this.socket = socket;
70         }
71     }
```

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(I/O
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 }

```

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?



33.5 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 33.5), **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.





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

LISTING 33.5 StudentAddress.java

serialized

```

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

```

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.

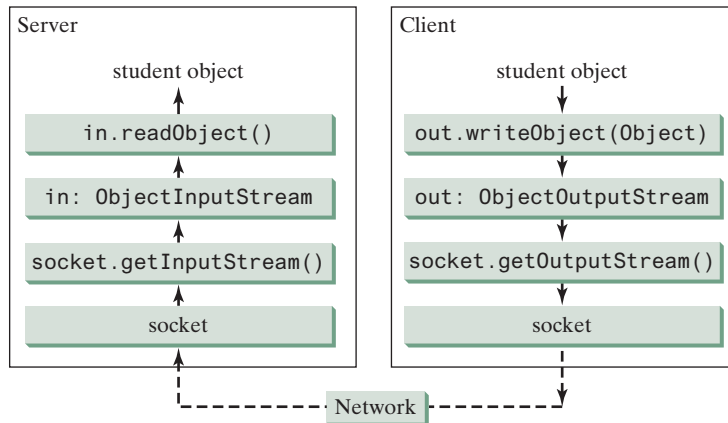


FIGURE 33.11 The client sends a `StudentAddress` object to the server.

LISTING 33.6 StudentClient.java

```

1  import java.io.*;
2  import java.net.*;
3  import javafx.application.Application;
4  import javafx.event.ActionEvent;
5  import javafx.event.EventHandler;
6  import javafx.geometry.HPos;
7  import javafx.geometry.Pos;
8  import javafx.scene.Scene;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TextField;
12 import javafx.scene.layout.GridPane;
13 import javafx.scene.layout.HBox;
14 import javafx.stage.Stage;
15
16 public class StudentClient extends Application {
17     private TextField tfName = new TextField();
18     private TextField tfStreet = new TextField();
19     private TextField tfCity = new TextField();
20     private TextField tfState = new TextField();
21     private TextField tfZip = new TextField();
22
23     // Button for sending a student to the server
24     private Button btRegister = new Button("Register to the Server");
25
26     // Host name or ip
27     String host = "localhost";
28
29     @Override // Override the start method in the Application class
30     public void start(Stage primaryStage) {
31         GridPane pane = new GridPane();
32         pane.add(new Label("Name"), 0, 0);
33         pane.add(tfName, 1, 0);
34         pane.add(new Label("Street"), 0, 1);
35         pane.add(tfStreet, 1, 1);
36         pane.add(new Label("City"), 0, 2);
37         create UI
  
```



```

38     HBox hBox = new HBox(2);
39     pane.add(hBox, 1, 2);
40     hBox.getChildren().addAll(tfCity, new Label("State"), tfState,
41         new Label("Zip"), tfZip);
42     pane.add(btRegister, 1, 3);
43     GridPane.setHalignment(btRegister, HPos.RIGHT);
44
45     pane.setAlignment(Pos.CENTER);
46     tfName.setPrefColumnCount(15);
47     tfStreet.setPrefColumnCount(15);
48     tfCity.setPrefColumnCount(10);
49     tfState.setPrefColumnCount(2);
50     tfZip.setPrefColumnCount(3);
51
52     btRegister.setOnAction(new ButtonListener());
53
54     // Create a scene and place it in the stage
55     Scene scene = new Scene(pane, 450, 200);
56     primaryStage.setTitle("StudentClient"); // Set the stage title
57     primaryStage.setScene(scene); // Place the scene in the stage
58     primaryStage.show(); // Display the stage
59 }
60
61 /** Handle button action */
62 private class ButtonListener implements EventHandler<ActionEvent> {
63     @Override
64     public void handle(ActionEvent e) {
65         try {
66             // Establish connection with the server
67             Socket socket = new Socket(host, 8000);
68
69             // Create an output stream to the server
70             ObjectOutputStream toServer =
71                 new ObjectOutputStream(socket.getOutputStream());
72
73             // Get text field
74             String name = tfName.getText().trim();
75             String street = tfStreet.getText().trim();
76             String city = tfCity.getText().trim();
77             String state = tfState.getText().trim();
78             String zip = tfZip.getText().trim();
79
80             // Create a Student object and send to the server
81             StudentAddress s =
82                 new StudentAddress(name, street, city, state, zip);
83             toServer.writeObject(s);
84         }
85         catch (IOException ex) {
86             ex.printStackTrace();
87         }
88     }
89 }
90 }

```

register listener

server socket

output stream

send to server

LISTING 33.7 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);           server socket
16         System.out.println("Server started ");
17
18         // Create an object output stream
19         outputToFile = new ObjectOutputStream(                       output to file
20             new FileOutputStream("student.dat", true));
21
22         while (true) {
23             // Listen for a new connection request
24             Socket socket = serverSocket.accept();                   connect to client
25
26             // Create an input stream from the socket
27             inputFromClient =                                         input stream
28                 new ObjectInputStream(socket.getInputStream());
29
30             // Read from input
31             Object object = inputFromClient.readObject();            get from client
32
33             // Write to the file
34             outputToFile.writeObject(object);                        write to file
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 }

```

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).



- 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

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.

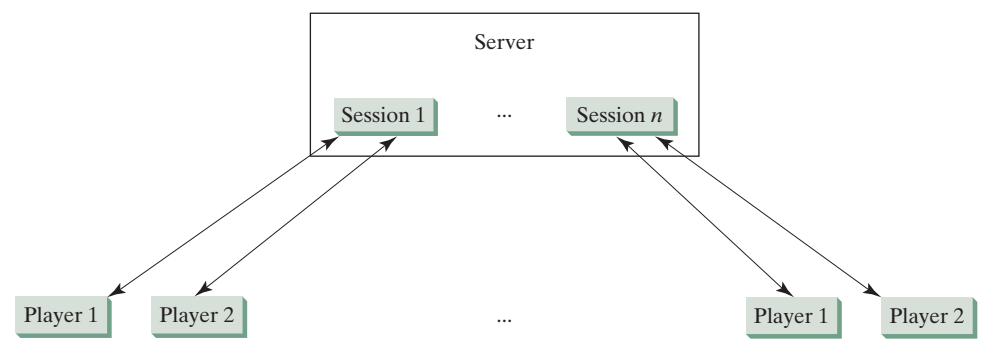


FIGURE 33.12 The server can create many sessions, each of which facilitates a tic-tac-toe game for two players.

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**.

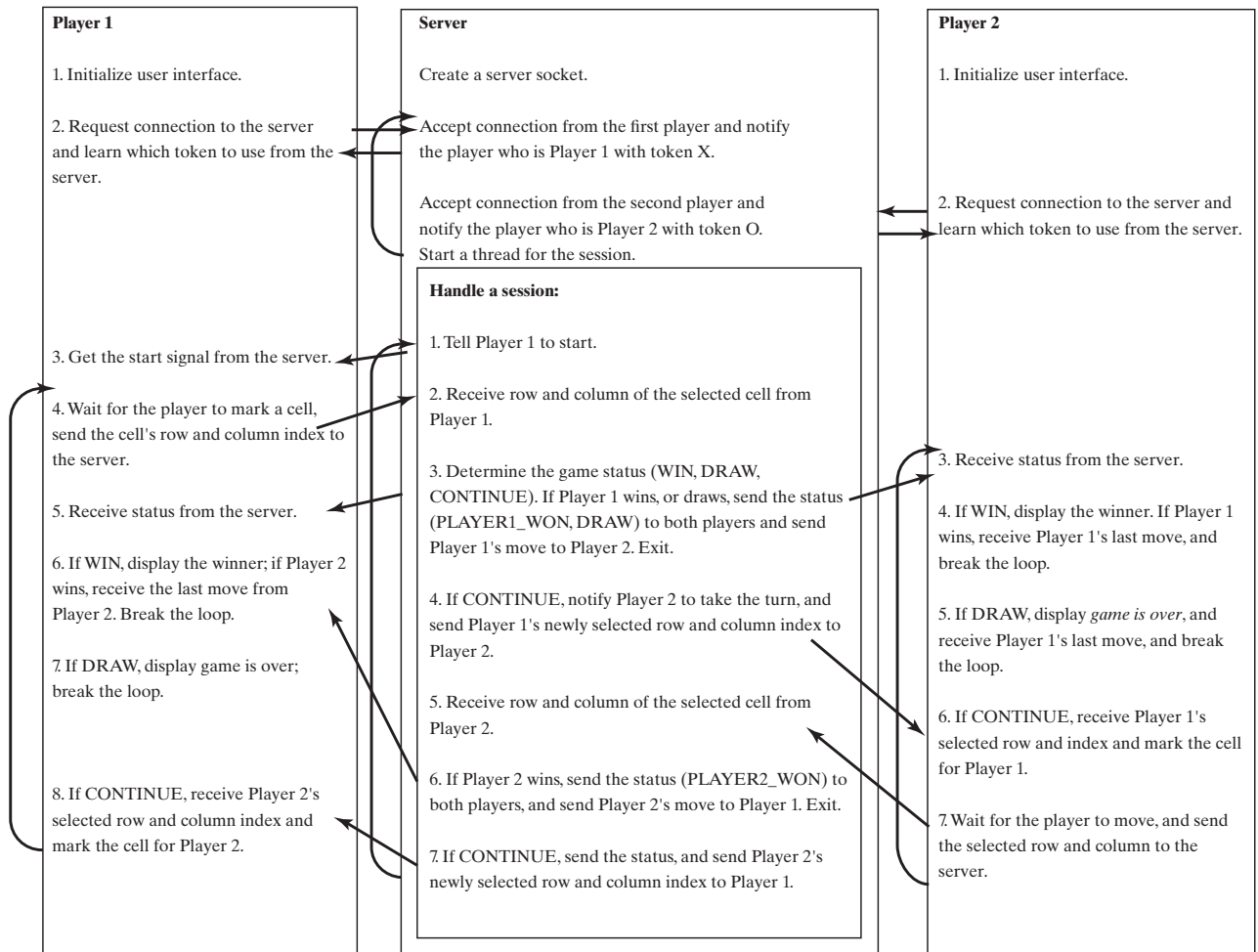


FIGURE 33.13 The server starts a thread to facilitate communications between the two players.

- **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

```

1 public interface TicTacToeConstants {
2     public static int PLAYER1 = 1; // Indicate player 1
3     public static int PLAYER2 = 2; // Indicate player 2
4     public static int PLAYER1_WON = 1; // Indicate player 1 won
5     public static int PLAYER2_WON = 2; // Indicate player 2 won
6     public static int DRAW = 3; // Indicate a draw
7     public static int CONTINUE = 4; // Indicate to continue
8 }
  
```

LISTING 33.9 TicTacToeServer.java

```

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

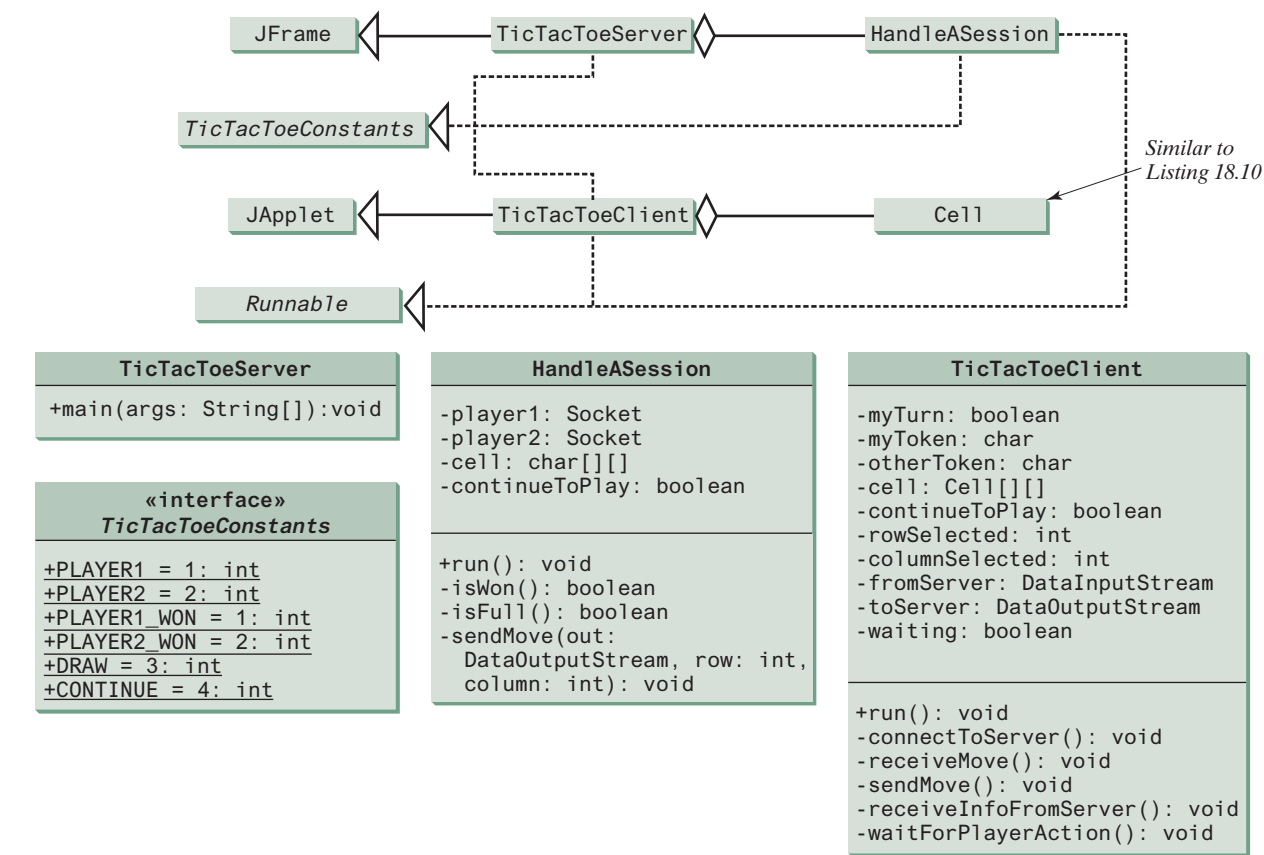


FIGURE 33.14 **TicTacToeServer** creates an instance of **HandleASession** for each session of two players. **TicTacToeClient** creates nine cells in the UI.

create UI

```
3 import java.util.Date;
4 import javafx.application.Application;
5 import javafx.application.Platform;
6 import javafx.scene.Scene;
7 import javafx.scene.control.ScrollPane;
8 import javafx.scene.control.TextArea;
9 import javafx.stage.Stage;
10
11 public class TicTacToeServer extends Application
12     implements TicTacToeConstants {
13     private int sessionNo = 1; // Number a session
14
15     @Override // Override the start method in the Application class
16     public void start(Stage primaryStage) {
17         TextArea taLog = new TextArea();
18
19         // Create a scene and place it in the stage
20         Scene scene = new Scene(new ScrollPane(taLog), 450, 200);
21         primaryStage.setTitle("TicTacToeServer"); // Set the stage title
22         primaryStage.setScene(scene); // Place the scene in the stage
23         primaryStage.show(); // Display the stage
24
25         new Thread( () -> {
26             try {
27                 // Create a server socket
```

```

28  ServerSocket serverSocket = new ServerSocket(8000);           server socket
29  Platform.runLater(() -> taLog.appendText(new Date() +
30      ": Server started at socket 8000\n"));
31
32  // Ready to create a session for every two players
33  while (true) {
34      Platform.runLater(() -> taLog.appendText(new Date() +
35          ": Wait for players to join session " + sessionNo + '\n'));
36
37      // Connect to player 1
38      Socket player1 = serverSocket.accept();                 connect to client
39
40      Platform.runLater(() -> {
41          taLog.appendText(new Date() + ": Player 1 joined session "
42              + sessionNo + '\n');
43          taLog.appendText("Player 1's IP address" +
44              player1.getInetAddress().getHostAddress() + '\n');
45      });
46
47      // Notify that the player is Player 1
48      new DataOutputStream(                                to player1
49          player1.getOutputStream()).writeInt(PAYER1);
50
51      // Connect to player 2
52      Socket player2 = serverSocket.accept();                 connect to client
53
54      Platform.runLater(() -> {
55          taLog.appendText(new Date() +
56              ": Player 2 joined session " + sessionNo + '\n');
57          taLog.appendText("Player 2's IP address" +
58              player2.getInetAddress().getHostAddress() + '\n');
59      });
60
61      // Notify that the player is Player 2
62      new DataOutputStream(                                to player2
63          player2.getOutputStream()).writeInt(PAYER2);
64
65      // Display this session and increment session number
66      Platform.runLater(() ->
67          taLog.appendText(new Date() +
68              ": Start a thread for session " + sessionNo++ + '\n'));
69
70      // Launch a new thread for this session of two players
71      new Thread(new HandleASession(player1, player2)).start();  a session for two players
72  }
73  }
74  catch(IOException ex) {
75      ex.printStackTrace();
76  }
77  }).start();
78  }
79
80  // Define the thread class for handling a new session for two players
81  class HandleASession implements Runnable, TicTacToeConstants {
82      private Socket player1;
83      private Socket player2;
84
85      // Create and initialize cells
86      private char[][] cell = new char[3][3];
87
88      private DataInputStream fromPlayer1;

```

IO streams

X won?

Is full?

```

89     private DataOutputStream toPlayer1;
90     private DataInputStream fromPlayer2;
91     private DataOutputStream toPlayer2;
92
93     // Continue to play
94     private boolean continueToPlay = true;
95
96     /** Construct a thread */
97     public HandleASession(Socket player1, Socket player2) {
98         this.player1 = player1;
99         this.player2 = player2;
100
101         // Initialize cells
102         for (int i = 0; i < 3; i++)
103             for (int j = 0; j < 3; j++)
104                 cell[i][j] = ' ';
105     }
106
107     /** Implement the run() method for the thread */
108     public void run() {
109         try {
110             // Create data input and output streams
111             DataInputStream fromPlayer1 = new DataInputStream(
112                 player1.getInputStream());
113             DataOutputStream toPlayer1 = new DataOutputStream(
114                 player1.getOutputStream());
115             DataInputStream fromPlayer2 = new DataInputStream(
116                 player2.getInputStream());
117             DataOutputStream toPlayer2 = new DataOutputStream(
118                 player2.getOutputStream());
119
120             // Write anything to notify player 1 to start
121             // This is just to let player 1 know to start
122             toPlayer1.writeInt(1);
123
124             // Continuously serve the players and determine and report
125             // the game status to the players
126             while (true) {
127                 // Receive a move from player 1
128                 int row = fromPlayer1.readInt();
129                 int column = fromPlayer1.readInt();
130                 cell[row][column] = 'X';
131
132                 // Check if Player 1 wins
133                 if (isWon('X')) {
134                     toPlayer1.writeInt(PLAYER1_WON);
135                     toPlayer2.writeInt(PLAYER1_WON);
136                     sendMove(toPlayer2, row, column);
137                     break; // Break the loop
138                 }
139                 else if (isFull()) { // Check if all cells are filled
140                     toPlayer1.writeInt(DRAW);
141                     toPlayer2.writeInt(DRAW);
142                     sendMove(toPlayer2, row, column);
143                     break;
144                 }
145                 else {
146                     // Notify player 2 to take the turn
147                     toPlayer2.writeInt(CONTINUE);
148

```



```

149         // Send player 1's selected row and column to player 2
150         sendMove(toPlayer2, row, column);
151     }
152
153     // Receive a move from Player 2
154     row = fromPlayer2.readInt();
155     column = fromPlayer2.readInt();
156     cell[row][column] = 'O';
157
158     // Check if Player 2 wins
159     if (isWon('O')) {                                O won?
160         toPlayer1.writeInt(PLAYER2_WON);
161         toPlayer2.writeInt(PLAYER2_WON);
162         sendMove(toPlayer1, row, column);
163         break;
164     }
165     else {
166         // Notify player 1 to take the turn
167         toPlayer1.writeInt(CONTINUE);
168
169         // Send player 2's selected row and column to player 1
170         sendMove(toPlayer1, row, column);
171     }
172 }
173
174 catch(IOException ex) {
175     ex.printStackTrace();
176 }
177
178
179 /** Send the move to other player */
180 private void sendMove(DataOutputStream out, int row, int column)    send a move
181     throws IOException {
182     out.writeInt(row); // Send row index
183     out.writeInt(column); // Send column index
184 }
185
186 /** Determine if the cells are all occupied */
187 private boolean isFull() {
188     for (int i = 0; i < 3; i++)
189         for (int j = 0; j < 3; j++)
190             if (cell[i][j] == '')
191                 return false; // At least one cell is not filled
192
193     // All cells are filled
194     return true;
195 }
196
197 /** Determine if the player with the specified token wins */
198 private boolean isWon(char token) {
199     // Check all rows
200     for (int i = 0; i < 3; i++)
201         if ((cell[i][0] == token)
202             && (cell[i][1] == token)
203             && (cell[i][2] == token)) {
204             return true;
205         }
206
207     // Check all columns */
208     for (int j = 0; j < 3; j++)

```

```

209         if ((cell[0][j] == token)
210             && (cell[1][j] == token)
211             && (cell[2][j] == token)) {
212             return true;
213         }
214
215         /** Check major diagonal */
216         if ((cell[0][0] == token)
217             && (cell[1][1] == token)
218             && (cell[2][2] == token)) {
219             return true;
220         }
221
222         /** Check subdiagonal */
223         if ((cell[0][2] == token)
224             && (cell[1][1] == token)
225             && (cell[2][0] == token)) {
226             return true;
227         }
228
229         /** All checked, but no winner */
230         return false;
231     }
232 }
233 }

```

LISTING 33.10 TicTacToeClient.java

```

1  import java.io.*;
2  import java.net.*;
3  import java.util.Date;
4  import javafx.application.Application;
5  import javafx.application.Platform;
6  import javafx.scene.Scene;
7  import javafx.scene.control.Label;
8  import javafx.scene.control.ScrollPane;
9  import javafx.scene.control.TextArea;
10 import javafx.scene.layout.BorderPane;
11 import javafx.scene.layout.GridPane;
12 import javafx.scene.layout.Pane;
13 import javafx.scene.paint.Color;
14 import javafx.scene.shape.Ellipse;
15 import javafx.scene.shape.Line;
16 import javafx.stage.Stage;
17
18 public class TicTacToeClient extends Application
19     implements TicTacToeConstants {
20     // Indicate whether the player has the turn
21     private boolean myTurn = false;
22
23     // Indicate the token for the player
24     private char myToken = ' ';
25
26     // Indicate the token for the other player
27     private char otherToken = ' ';
28
29     // Create and initialize cells
30     private Cell[][] cell = new Cell[3][3];
31

```

```

32 // Create and initialize a title label
33 private Label lblTitle = new Label();
34
35 // Create and initialize a status label
36 private Label lblStatus = new Label();
37
38 // Indicate selected row and column by the current move
39 private int rowSelected;
40 private int columnSelected;
41
42 // Input and output streams from/to server
43 private DataInputStream fromServer;
44 private DataOutputStream toServer;
45
46 // Continue to play?
47 private boolean continueToPlay = true;
48
49 // Wait for the player to mark a cell
50 private boolean waiting = true;
51
52 // Host name or ip
53 private String host = "localhost";
54
55 @Override // Override the start method in the Application class
56 public void start(Stage primaryStage) {
57     // Pane to hold cell
58     GridPane pane = new GridPane();
59     for (int i = 0; i < 3; i++)
60         for (int j = 0; j < 3; j++)
61             pane.add(cell[i][j] = new Cell(i, j), j, i);
62
63     BorderPane borderPane = new BorderPane();
64     borderPane.setTop(lblTitle);
65     borderPane.setCenter(pane);
66     borderPane.setBottom(lblStatus);
67
68     // Create a scene and place it in the stage
69     Scene scene = new Scene(borderPane, 320, 350);
70     primaryStage.setTitle("TicTacToeClient"); // Set the stage title
71     primaryStage.setScene(scene); // Place the scene in the stage
72     primaryStage.show(); // Display the stage
73
74     // Connect to the server
75     connectToServer();
76 }
77
78 private void connectToServer() {
79     try {
80         // Create a socket to connect to the server
81         Socket socket = new Socket(host, 8000);
82
83         // Create an input stream to receive data from the server
84         fromServer = new DataInputStream(socket.getInputStream());
85
86         // Create an output stream to send data to the server
87         toServer = new DataOutputStream(socket.getOutputStream());
88     }
89     catch (Exception ex) {
90         ex.printStackTrace();
91     }

```

create UI

connect to server

input from server

output to server

```

92
93 // Control the game on a separate thread
94 new Thread(() -> {
95     try {
96         // Get notification from the server
97         int player = fromServer.readInt();
98
99         // Am I player 1 or 2?
100        if (player == PLAYER1) {
101            myToken = 'X';
102            otherToken = 'O';
103            Platform.runLater(() -> {
104                lblTitle.setText("Player 1 with token 'X'");
105                lblStatus.setText("Waiting for player 2 to join");
106            });
107
108            // Receive startup notification from the server
109            fromServer.readInt(); // Whatever read is ignored
110
111            // The other player has joined
112            Platform.runLater(() ->
113                lblStatus.setText("Player 2 has joined. I start first"));
114
115            // It is my turn
116            myTurn = true;
117        }
118        else if (player == PLAYER2) {
119            myToken = 'O';
120            otherToken = 'X';
121            Platform.runLater(() -> {
122                lblTitle.setText("Player 2 with token 'O'");
123                lblStatus.setText("Waiting for player 1 to move");
124            });
125        }
126
127        // Continue to play
128        while (continueToPlay) {
129            if (player == PLAYER1) {
130                waitForPlayerAction(); // Wait for player 1 to move
131                sendMove(); // Send the move to the server
132                receiveInfoFromServer(); // Receive info from the server
133            }
134            else if (player == PLAYER2) {
135                receiveInfoFromServer(); // Receive info from the server
136                waitForPlayerAction(); // Wait for player 2 to move
137                sendMove(); // Send player 2's move to the server
138            }
139        }
140    }
141    catch (Exception ex) {
142        ex.printStackTrace();
143    }
144    }).start();
145 }
146
147 /** Wait for the player to mark a cell */
148 private void waitForPlayerAction() throws InterruptedException {
149     while (waiting) {
150         Thread.sleep(100);
151     }

```

```

152
153     waiting = true;
154 }
155
156 /** Send this player's move to the server */
157 private void sendMove() throws IOException {
158     toServer.writeInt(rowSelected); // Send the selected row
159     toServer.writeInt(columnSelected); // Send the selected column
160 }
161
162 /** Receive info from the server */
163 private void receiveInfoFromServer() throws IOException {
164     // Receive game status
165     int status = fromServer.readInt();
166
167     if (status == PLAYER1_WON) {
168         // Player 1 won, stop playing
169         continueToPlay = false;
170         if (myToken == 'X') {
171             Platform.runLater(() -> lblStatus.setText("I won! (X)"));
172         }
173         else if (myToken == 'O') {
174             Platform.runLater(() ->
175                 lblStatus.setText("Player 1 (X) has won!"));
176             receiveMove();
177         }
178     }
179     else if (status == PLAYER2_WON) {
180         // Player 2 won, stop playing
181         continueToPlay = false;
182         if (myToken == 'O') {
183             Platform.runLater(() -> lblStatus.setText("I won! (O)"));
184         }
185         else if (myToken == 'X') {
186             Platform.runLater(() ->
187                 lblStatus.setText("Player 2 (O) has won!"));
188             receiveMove();
189         }
190     }
191     else if (status == DRAW) {
192         // No winner, game is over
193         continueToPlay = false;
194         Platform.runLater(() ->
195             lblStatus.setText("Game is over, no winner!"));
196
197         if (myToken == 'O') {
198             receiveMove();
199         }
200     }
201     else {
202         receiveMove();
203         Platform.runLater(() -> lblStatus.setText("My turn"));
204         myTurn = true; // It is my turn
205     }
206 }
207
208 private void receiveMove() throws IOException {
209     // Get the other player's move
210     int row = fromServer.readInt();
211     int column = fromServer.readInt();

```

model a cell

register listener

draw X

draw O

```

212     Platform.runLater(() -> cell[row][column].setToken(otherToken));
213 }
214
215 // An inner class for a cell
216 public class Cell extends Pane {
217     // Indicate the row and column of this cell in the board
218     private int row;
219     private int column;
220
221     // Token used for this cell
222     private char token = ' ';
223
224     public Cell(int row, int column) {
225         this.row = row;
226         this.column = column;
227         this.setPrefSize(2000, 2000); // What happens without this?
228         setStyle("-fx-border-color: black"); // Set cell's border
229         this.setOnMouseClicked(e -> handleMouseClicked());
230     }
231
232     /** Return token */
233     public char getToken() {
234         return token;
235     }
236
237     /** Set a new token */
238     public void setToken(char c) {
239         token = c;
240         repaint();
241     }
242
243     protected void repaint() {
244         if (token == 'X') {
245             Line line1 = new Line(10, 10,
246                 this.getWidth() - 10, this.getHeight() - 10);
247             line1.endXProperty().bind(this.widthProperty().subtract(10));
248             line1.endYProperty().bind(this.heightProperty().subtract(10));
249             Line line2 = new Line(10, this.getHeight() - 10,
250                 this.getWidth() - 10, 10);
251             line2.startYProperty().bind(
252                 this.heightProperty().subtract(10));
253             line2.endXProperty().bind(this.widthProperty().subtract(10));
254
255             // Add the lines to the pane
256             this.getChildren().addAll(line1, line2);
257         }
258         else if (token == 'O') {
259             Ellipse ellipse = new Ellipse(this.getWidth() / 2,
260                 this.getHeight() / 2, this.getWidth() / 2 - 10,
261                 this.getHeight() / 2 - 10);
262             ellipse.centerXProperty().bind(
263                 this.widthProperty().divide(2));
264             ellipse.centerYProperty().bind(
265                 this.heightProperty().divide(2));
266             ellipse.radiusXProperty().bind(
267                 this.widthProperty().divide(2).subtract(10));
268             ellipse.radiusYProperty().bind(
269                 this.heightProperty().divide(2).subtract(10));
270             ellipse.setStroke(Color.BLACK);
271             ellipse.setFill(Color.WHITE);

```

```

272
273     getChildren().add(ellipse); // Add the ellipse to the pane
274 }
275 }
276
277 /* Handle a mouse click event */
278 private void handleClick() {
279     // If cell is not occupied and the player has the turn
280     if (token == ' ' && myTurn) {
281         setToken(myToken); // Set the player's token in the cell
282         myTurn = false;
283         rowSelected = row;
284         columnSelected = column;
285         lblStatus.setText("Waiting for the other player to move");
286         waiting = false; // Just completed a successful move
287     }
288 }
289 }
290 }

```

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.



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

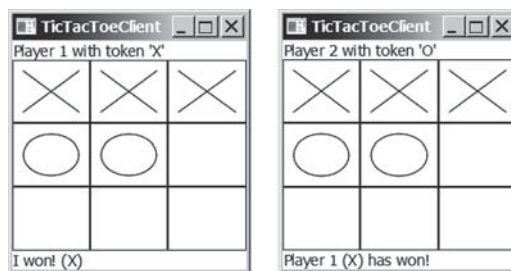


FIGURE 33.16 TicTacToeClient can run as an applet or standalone.

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.



- 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`.

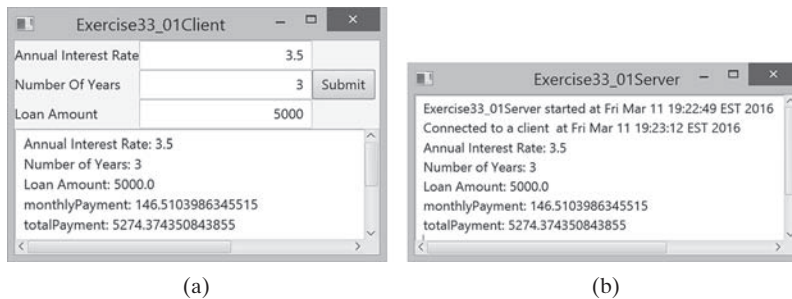


FIGURE 33.17 The client in (a) sends the annual interest rate, number of years, and loan amount to the server and receives the monthly payment and total payment from the server in (b).

- *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`.

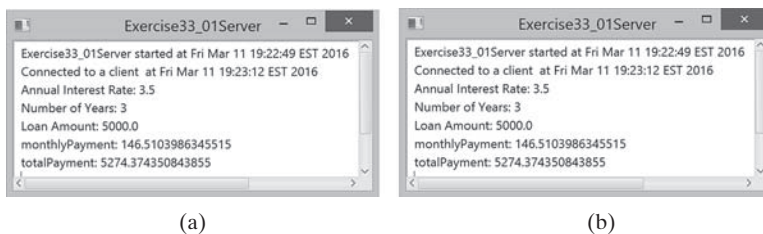


FIGURE 33.18 The client in (a) sends the weight and height of a person to the server and receives the BMI from the server in (b).

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`.

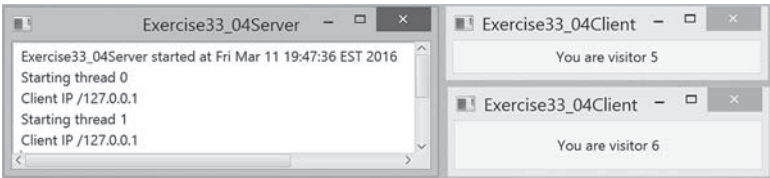


FIGURE 33.19 The client displays how many times the server has been accessed. The server stores the count.

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.



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`.

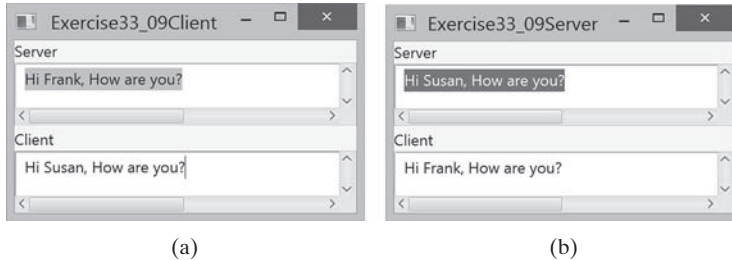


FIGURE 33.21 The server and client send text to and receive text from each other.

*****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`.

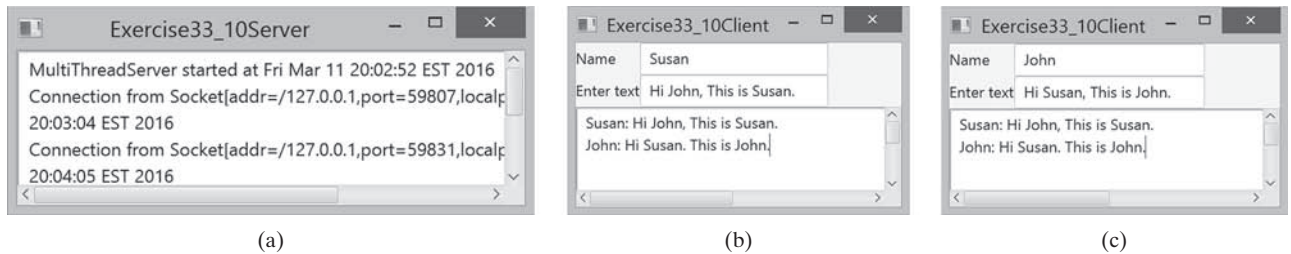


FIGURE 33.22 The server starts in (a) with three clients in (b) and (c).