***Chapter in the Advanced Version

IX

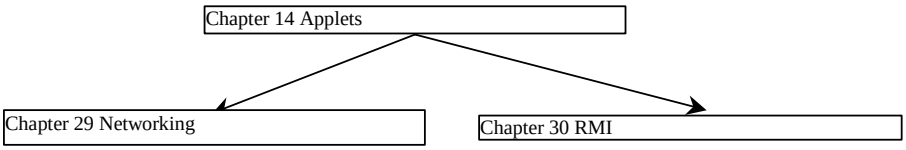
Networking and RMI

This part introduces how to write programs that talk to with each other over the Internet. Networking is embedded in Java. Chapter 29 introduces low-level socket network programming and Chapter 30 introduces high-level remote method invocation.

Chapter 29
Networking
Chapter 30
Remote Method Invocation

***PD: Put this in the center of the back of the Part-opening page. AU

Prerequisites for Part IX



CHAPTER

29

Networking

Objectives

[BL]To comprehend socket-based communication in Java.

[BL]To understand client/server computing.

[BL]To implement Java networking programs using stream sockets.

[BL]To develop servers for multiple clients.

[BL]To send and receive objects on the network.

[BL]To develop applets that communicate with the server.

[BL]To create applications or applets to retrieve files from the network.

[BL]To render HTML files using the JEditorPane class.

[BL]To implement Java networking programs using dartagram sockets.

29.1 Introduction

Networking is tightly integrated in Java. *Socket-based communication* is provided that enables programs to communicate through designated sockets. A *socket* is an abstraction that facilitates communication between a server and a client. 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.

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. Because of this, stream sockets are used in most areas of Java programming, and that is why the most of discussion in this chapter is based on stream sockets. Datagram socket programming is introduced in the last section of this chapter.

29.2 Client/Server Computing

Network programming usually involves a server and one or more clients. The client sends requests to the server, and the server responds to the requests. 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 starts. The server waits for a connection request from a client. The statements needed to create a server and a client are shown in Figure 29.1.

Same as Fig21.1 in Bonus Chapter 21 in introjb3e p40 (Bonus Chapters are in the CD-ROM in introjb3e)

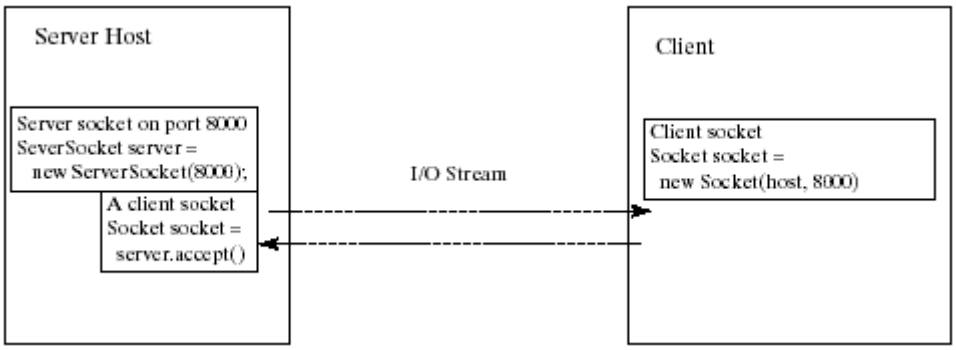


Figure 29.1

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

To establish a server, you need to create a server socket and attach it to a port, which is where the server listens for connections. The port identifies the TCP service on the socket. Port numbers between 0 and 1023 are reserved for privileged processes. For instance, the e-mail 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 any other process. The following statement creates a server socket serverSocket:

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

NOTE

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

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:

```
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 at port 8000 on the client machine to connect to the host drake.armstrong.edu:

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

Alternatively, you can use the IP address to create a socket, as follows:

```
Socket socket = new Socket("130.254.204.36", 8000)
```

An IP address, consisting of four dotted decimal numbers between 0 and 255, such as 130.254.204.36, is a computer's unique identity on the Internet. Since it is not easy to remember so many numbers, they are often mapped to meaningful names called *host names*, such as

```
drake.armstrong.edu.
```

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.

NOTE

There are special servers on the Internet that translate host names into IP addresses. These servers are called Domain Name Servers (DNS). The translation is done behind the scenes. When you create a socket with a host name, the Java Runtime System asks the DNS to translate the host name into the IP address.

After the server accepts the connection, communication between server and client is conducted the same as for I/O streams. The statements needed to create the streams and to exchange data between them are shown in Figure 29.2.

******Same as Fig21.2 in Bonus Chapter 21 in introjb3e p42 (Bonus Chapters are in the CD-ROM in introjb3e)***

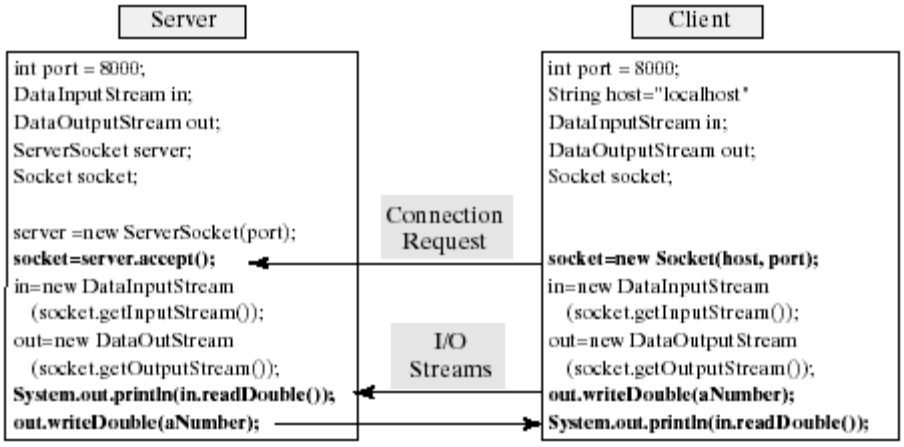


Figure 29.2

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, input, and an OutputStream stream, 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 a DataInputStream stream, input, and a DataOutput stream, output, to read and write primitive data values:

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

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

Example 29.1

A Client/Server Example

Problem

This example presents a client program and a server program. The client sends data to a server. The server receives the data, uses them 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 29.3).

***Same as Fig21.3 in Bonus Chapter 21 in introjb3e p43 (Bonus Chapters are in the CD-ROM in introjb3e)

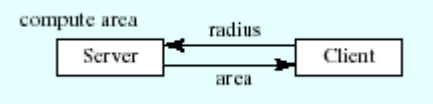


Figure 29.3

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

Solution

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 29.4 (A). 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 29.4 (B). The client and server programs are given below. Figure 29.5 contains a sample run of the server and the client.

***Same as Fig21.4 in Bonus Chapter 21 in introjb3e p43 (Bonus Chapters are in the CD-ROM in introjb3e)

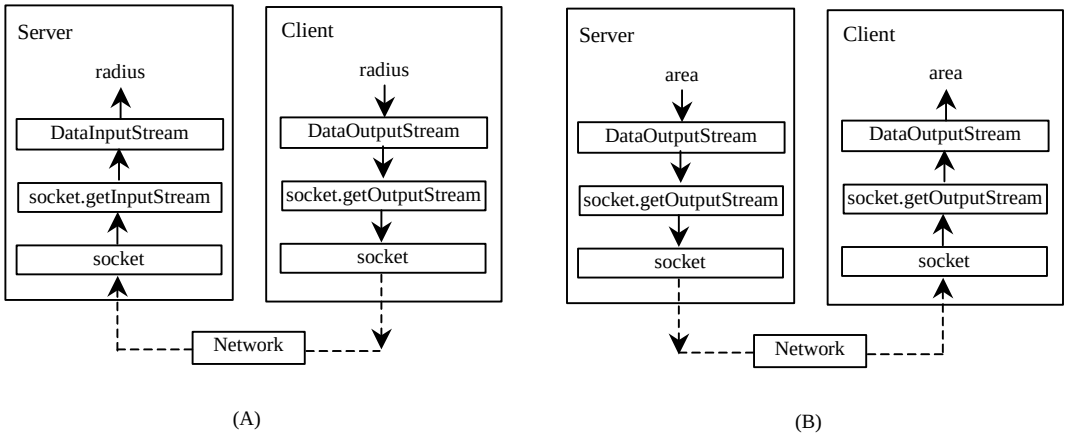


Figure 29.4

(A) The client sends the radius to the server. (B) The server sends the area to the client.

***Same as Fig21.5 in Bonus Chapter 21 in introjb3e p43 (Bonus Chapters are in the CD-ROM in introjb3e)

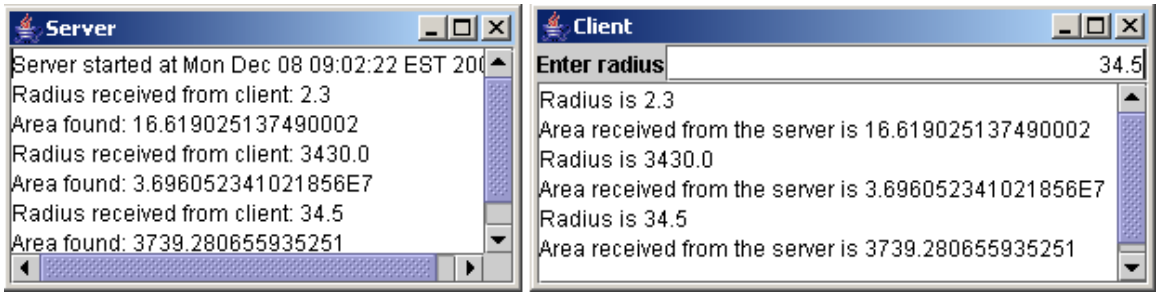


Figure 29.5

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

PD: Please add line numbers in the following code

```
// Server.java: The server accepts data from the client, processes it
// and returns the result back to the client
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

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

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

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

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

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

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

      // 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);
```

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

PD: Please insert a separator line

PD: Please add line numbers in the following code

```
// Client.java: The client sends the input to the server and receives
// result back from the server
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

  // IO streams
  DataOutputStream outputToServer;
  DataInputStream inputFromServer;

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

  public Client() {
    // Panel p to hold the label and text field
    JPanel p = new JPanel();
    p.setLayout(new BorderLayout());
    p.add(new JLabel("Enter radius"), BorderLayout.WEST);
    p.add(jtf, BorderLayout.CENTER);
    jtf.setHorizontalAlignment(JTextField.RIGHT);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(p, BorderLayout.NORTH);
    getContentPane().add(new JScrollPane(jta), BorderLayout.CENTER);

    jtf.addActionListener(this); // Register listener

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

    try {
      // Create a socket to connect to the server
      Socket socket = new Socket("localhost", 8000);
      // Socket socket = new Socket("130.254.204.36", 8000);
      // Socket socket = new Socket("drake.Armstrong.edu", 8000);

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

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

  public void actionPerformed(ActionEvent e) {
    String actionCommand = e.getActionCommand();
    if (e.getSource() instanceof JTextField) {
      try {
        // Get the radius from the text field
```

```
        double radius = Double.parseDouble(jtf.getText().trim());

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

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

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

Review

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. This should be imported when writing Java network programs.

The Server class creates a ServerSocket serverSocket and attaches it to port 8000, using the following statement (Line 29 in Server.java):

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

The server then starts to listen for connection requests, using the following statement (Line 33 in Server.java):

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

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

The Client class uses the following statement to create a socket that will request a connection to the server on the same machine (localhost) at port 8000 (Line 45 in Client.java).

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

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

If the server is not running, the client program terminates with a java.net.ConnectException. After it is connected, the client gets input and output

streams—wrapped by data input and output streams—in order to receive and send data to the server.

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

What happens if the setVisible(true) statement in Line 25 in Server.java is moved after the try/catch block in Line 58 in Server.java? The frame would not be displayed because the while loop in the try/catch block will not finish until the program terminates.

29.3 The InetAddress Class

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 statement shown below to create an instance of InetAddress for the client on a socket.

```
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");
```

29.4 Serving Multiple Clients

Multiple clients are quite often connected to a single server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet may want to 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();
  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.

Example 29.2

Serving Multiple Clients

Problem

Write a server 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 corresponding clients (see Figure 29.6).

******Same as Fig21.6 in Bonus Chapter 21 in introjb3e p48 (Bonus Chapters are in the CD-ROM in introjb3e)***

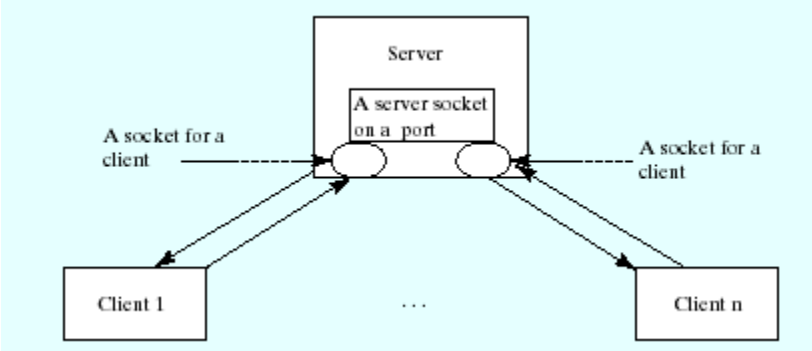


Figure 29.6

Multithreading enables a server to handle multiple independent clients.

Solution

The client program is same as in Example 29.1. The new server program follows. A sample run of the server with two clients is shown in Figure 29.7.

******Same as Fig21.7 in Bonus Chapter 21 in introjb3e p48 (Bonus Chapters are in the CD-ROM in introjb3e)***

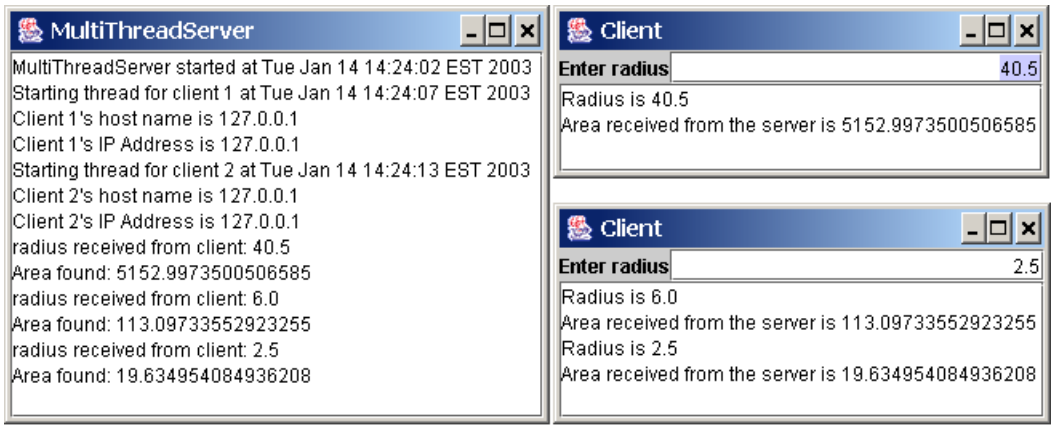


Figure 29.7

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

******PD: Please add line numbers in the following code******

```
// MultiThreadServer.java: The server can communicate with
// multiple clients concurrently using the multiple threads
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

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

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

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

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

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

      // Number a client
      int clientNo = 1;

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

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

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

        // Create a new thread for the connection
        HandleAClient thread = new HandleAClient(socket);
```

```
        // Start the new thread
        thread.start();

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

  // Inner class
  // Define the thread class for handling new connection
  class HandleAClient extends Thread {
    private Socket socket; // A connected socket

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

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

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

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

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

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

Review

The server creates a server socket at port 8000 and waits for a connection. When a connection with a client is established, the server creates a new thread to handle the communication. It then waits for another connection.

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 the client.

This server accepts an unlimited number of clients. To limit the number of concurrent connections, you can use a thread group to monitor the number of active threads and modify the while loop (Lines 35-58), as follows:

```
ThreadGroup group = new ThreadGroup("serving clients");

while (true) {
```

```
  if (group.activeCount() >= maxThreadLimit)
    try {
      Thread.sleep(1000);
    }
    catch (InterruptedException ex) {
    }
  else {
    // Listen for a new connection request
    Socket socket = serverSocket.accept();

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

    // Create a new thread for the connection
    Thread thread = new Thread(group,
      new HandleAClient(socket));

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

    // Increment clientNo to label the next connection
    clientNo++;
  }
}
```

29.5 Applet Clients

Because of security constraints, applets can only connect to the host from which they were loaded. Therefore, the HTML file must be located on the machine on which the server is running. Below is an example of how to use an applet to connect to a server.

Example 29.3

Creating Applet Clients

Problem

This example, which is similar to Example 16.7, "Using Random Access Files," shows how to use an applet to register students. The client collects and sends registration information to the server. The server appends the information to a data file using a random access file stream.

Solution

The server and client programs are given below. A sample run of the server and client is shown in Figure 29.8.

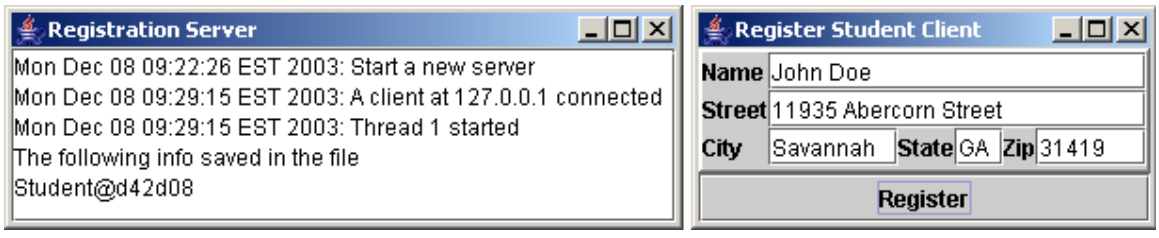


Figure 29.8

The server receives information from a client (name, street, city, state, and zip code) and stores it in a file.

******PD: Please add line numbers in the following code******

```
// RegistrationServer.java: The server for the applet responsible for
// writing on the server side
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.util.Date;

public class RegistrationServer extends JFrame {
  private static JTextArea jtaLog;

  // The file to store the records
  private static RandomAccessFile raf = null;

  /** Main method */
  public static void main(String[] args) {
    new RegistrationServer();
  }

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

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

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

    // Open the local file on the server side
    try {
      // Open the file if the file exists, create a new file
      // if the file does not exist
      raf = new RandomAccessFile("student.dat", "rw");
    }
    catch(IOException ex) {
      jtaLog.append(new Date() + ": Error: " + ex);
      System.exit(0);
    }

    // Establish server socket
    try {
      // Create a server socket
      ServerSocket serverSocket = new ServerSocket(8000);
      jtaLog.append(new Date() + ": Start a new server\n");

      // Count the number of threads started
      int count = 1;

      while (true) {
        // Connect to a client
        Socket socket = serverSocket.accept();
        jtaLog.append(new Date() + ": A client at " +
          socket.getInetAddress().getHostAddress() + " connected\n");

        // Start a new thread to register a client
        new RegistrationThread(socket, count++).start();
      }
    }
    catch (IOException ex) {
      jtaLog.append(new Date() + ": " + ex);
    }
  }

  /** Write student information to the file */
  private synchronized static void writeToFile(Student student) {
    try {
      // Append it to "student.dat"
      raf.seek(raf.length());
      student.writeStudent(raf);
```

```
      // Display data saved
      jtaLog.append("The following info saved in the file\n");
      jtaLog.append(student.toString());
    }
    catch (Exception ex) {
      jtaLog.append(new Date() + ": " + ex);
    }
  }

  // Define a thread to process the client registration
  class RegistrationThread extends Thread {
    // The socket to serve a client
    private Socket socket;

    private int clientNo; // The thread number

    // Buffered reader to get input from the client
    private BufferedReader in;

    // Create a registration thread
    public RegistrationThread(Socket socket, int clientNo) {
      this.socket = socket;
      this.clientNo = clientNo;

      jtaLog.append(new Date() + ": Thread " + clientNo
        + " started\n");

      // Create an input stream to receive data from a client
      try {
        in = new BufferedReader
          (new InputStreamReader(socket.getInputStream()));
      }
      catch(IOException ex) {
        jtaLog.append(new Date() + ": " + ex);
      }
    }

    public void run() {
      String name;
      String street;
      String city;
      String state;
      String zip;

      try {
        // Receive data from the client
        name = new String(in.readLine());
        street = new String(in.readLine());
        city = new String(in.readLine());
        state = new String(in.readLine());
        zip = new String(in.readLine());

        // Create a student instance
        Student student =
          new Student(name, street, city, state, zip);

        writeToFile(student);
      }
      catch (IOException ex) {
        System.out.println(ex);
      }
    }
  }
}
```

PD: Please insert a separator line here

PD: Please add line numbers in the following code

```
// RegistrationClient.java: The applet client for gathering student
// informationthe and passing it to the server
import java.io.*;
import java.net.*;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

public class RegistrationClient extends JApplet
  implements ActionListener {
  // Button for registering a student in the file
```

```
  private JButton jbtRegister = new JButton("Register");

  // Create student information panel
  private StudentPanel studentPanel = new StudentPanel();

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

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

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

    // Register listener
    jbtRegister.addActionListener(this);
  }

  /** Handle button action */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == jbtRegister) {
      try {
        // Establish connection with the server
        Socket socket;
        if (isStandAlone)
          socket = new Socket(host, 8000);
        else
          socket = new Socket(getCodeBase().getHost(), 8000);

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

        // Get text field
        Student s = studentPanel.getStudent();

        // Get data from text fields and send it to the server
        toServer.println(s.getName());
        toServer.println(s.getStreet());
        toServer.println(s.getCity());
        toServer.println(s.getState());
        toServer.println(s.getZip());
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    }
  }

  /** Run the applet as an application */
  public static void main(String[] args) {
    // Create a frame
    JFrame frame = new JFrame("Register Student Client");

    // Create an instance of the applet
    RegistrationClient applet = new RegistrationClient();
    applet.isStandAlone = true;

    // Get host
    if (args.length == 1) applet.host = args[0];

    // Add the applet instance to the frame
    frame.getContentPane().add(applet, BorderLayout.CENTER);

    // Invoke init() and start()
    applet.init();
    applet.start();

    // Display the frame
    frame.pack();
    frame.setVisible(true);
  }
}
```

Review

Let us first review RegistrationServer.java:

The server handles multiple clients. It waits for a connection request from a client in the while loop (Line 55). After a connection with a client is established, the server creates a thread to serve the client. The server then stays in the while loop to listen for the next connection request.

The server passes socket (connection socket) and count (thread number) to the thread (Line 60). The count argument is nonessential; its only use is to identify the thread. The thread receives student information from the client through the BufferedReader stream and appends a student record to **student.dat**, using the random-access file raf.

The StudentPanel and Student classes were defined in Example 16.7, "Using Random Access Files." The statement given below (Lines 128–129) creates an instance of Student:

```
Student student = new Student(name, street, city, state, zip);
```

The next code (Line 130) writes the student record into the file:

```
s.writeStudent(raf);
```

When multiple clients register students simultaneously, data may be corrupted. To avoid this, the writeToFile() method (Line 69) is synchronized and defined as a static method. The synchronization is at the class level, meaning that only one object of the RegistrationThread at a time can execute the writeToFile() method to write a student to the file.

Now let us review RegistrationClient.java:

The client is an applet and can run standalone. When it runs as an applet, it uses getCodeBase().getHost() (Line 41) to return the IP address for the server. When it runs as an application, it passes the URL from the command line. If the URL is not passed from the command line, by default "localhost" is used for the URL.

The data are entered into text fields (name, street, city, state, and zip). When the Register button is clicked, the data from the text fields are collected and sent to the server. Each time the Register button is clicked from a client, the server creates a new thread to store the student information. Obviously, this is not efficient. In Exercise 29.3, you will modify the program to enable one thread to handle all the registration requests from a single client.

29.6 Sending and Receiving Objects

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.

Example 29.4

Passing Objects in Network Programs

Problem

This example rewrites Example 29.3, "Creating Applet Clients," using object streams on the socket. Instead of passing name, street, state, and zip separately, the program passes the student object as a whole object.

Solution

The client sends the Student 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 29.9. The client uses the writeObject method in the ObjectOutputStream class to send a student to the server, and the server receives the student using the readObject method in the ObjectInputStream class. The server and client programs are given as follows:

******Same as Fig21.9 in Bonus Chapter 21 in introjb3e p57 (Bonus Chapters are in the CD-ROM in introjb3e)***

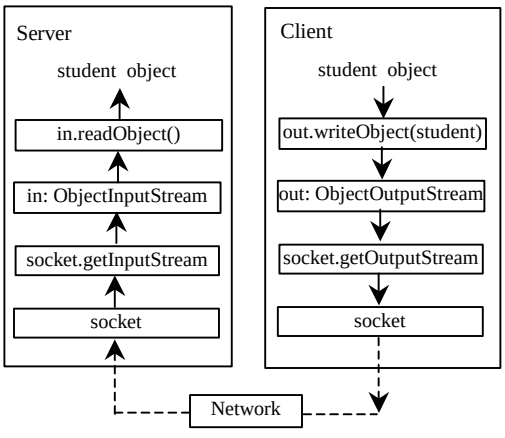


Figure 29.9

The client sends the radius to the server. The server sends the area to the client.

PD: Please add line numbers in the following code

```
// RegistrationServerUsingObjectStream.java: The server for the
// applet responsible for writing on the server side
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.util.Date;

public class RegistrationServerUsingObjectStream extends JFrame {
  private static JTextArea jtaLog;

  // The file to store the records
  private static RandomAccessFile raf = null;

  /** Main method */
  public static void main(String[] args) {
    new RegistrationServerUsingObjectStream();
  }

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

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

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(300, 300);
    setTitle("Registration Server Using Object Streams");
    setVisible(true);

    // Open the local file on the server side
    try {
      // Open the file if the file exists, create a new file
      // if the file does not exist
      raf = new RandomAccessFile("student.dat", "rw");
    }
    catch(IOException ex) {
      jtaLog.append(new Date() + ": Error: " + ex);
      System.exit(0);
    }

    // Establish server socket
    try {
      // Create a server socket
      ServerSocket serverSocket = new ServerSocket(8000);
      jtaLog.append(new Date() + ": Start a new server\n");

      // Count the number of threads started
      int count = 1;

      while (true) {
        // Connect to a client
        Socket socket = serverSocket.accept();
        jtaLog.append(new Date() + ": A client at " +
          socket.getInetAddress().getHostAddress() + " connected\n");

        // Start a new thread to register a client
        new RegistrationThread(socket, count++).start();
      }
    }
    catch (IOException ex) {
      jtaLog.append(new Date() + ": " + ex);
    }
  }

  /** Write student information to the file */
  private synchronized static void writeToFile(Student student) {
    try {
      // Append it to "student.dat"
      raf.seek(raf.length());
      student.writeStudent(raf);

      // Display data saved
      jtaLog.append("The following info is saved in the file\n");
      jtaLog.append(student.toString());
    }
    catch (Exception ex) {
      jtaLog.append(new Date() + ": " + ex);
    }
```

```
  }

  /** Define a thread to process the client registration */
  class RegistrationThread extends Thread {
    // The socket to serve a client
    private Socket socket;

    private int clientNo; // The thread number

    // Object input stream to get input from the client
    private ObjectInputStream in;

    // Create a registration thread
    public RegistrationThread(Socket socket, int clientNo) {
      this.socket = socket;
      this.clientNo = clientNo;

      jtaLog.append(new Date() + ": Thread " + clientNo
        + " started\n");

      // Create an input stream to receive data from a client
      try {
        in = new ObjectInputStream(socket.getInputStream());
      }
      catch(IOException ex) {
        jtaLog.append(new Date() + ": " + ex);
      }
    }

    public void run() {
      try {
        // Receive data from the client
        Student student = (Student)in.readObject();

        writeToFile(student);
      }
      catch (Exception ex) {
        System.out.println(ex);
      }
    }
  }
}
```

Layout: Insert a separator line here. AU

PD: Please add line numbers in the following code

```
// RegistrationClientUsingObjectStream.java: The applet client for
// gathering the student information and passing it to the server
import java.io.*;
import java.net.*;
import java.awt.BorderLayout;
import java.awt.event.*;
import javax.swing.*;

public class RegistrationClientUsingObjectStream extends JApplet
  implements ActionListener {
  // Button for registering a student in the file
  private JButton jbtRegister = new JButton("Register");

  // Create student information panel
  private StudentPanel studentPanel = new StudentPanel();

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

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

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

    // Register listener
    jbtRegister.addActionListener(this);

    // Find the IP address of the Web server
    if (!isStandAlone)
```

```
      host = getCodeBase().getHost();
    }

    /** Handle button action */
    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == jbtRegister) {
        try {
          // Establish connection with the server
          Socket socket = new Socket(host, 8000);

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

          // Get text field
          Student s = studentPanel.getStudent();

          // Get data from text fields and send it to the server
          toServer.writeObject(s);
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
      }
    }

    /** Run the applet as an application */
    public static void main(String[] args) {
      // Create a frame
      JFrame frame = new JFrame("Register Student Client");

      // Create an instance of the applet
      RegistrationClientUsingObjectStream applet =
        new RegistrationClientUsingObjectStream();
      applet.isStandAlone = true;

      // Get host
      if (args.length == 1) applet.host = args[0];

      // Add the applet instance to the frame
      frame.getContentPane().add(applet, BorderLayout.CENTER);

      // Invoke init() and start()
      applet.init();
      applet.start();

      // Display the frame
      frame.pack();
      frame.setVisible(true);
    }
  }
```

Review

The Student class in Example 16.8, "Using Random Access Files," implements the Serializable interface. Therefore, it can be sent and received using the object output and input streams.

On the client side, when the user clicks the Registration button, the client creates a socket to connect to the host (Line 45), creates an ObjectOutputStream on the output stream of the socket (Lines 48-49), and invokes the writeObject method to send the Student object to the server through the object output stream (Line 55).

On the server side, when a client connects to the server, the server creates a thread to process the client registration (Line 64). The thread creates an ObjectOutputStream on the input stream of the socket (Line 108), and invokes the readObject method to receive the Student object through the object input stream (Line 118).

The client collects student information from the text fields and creates an instance of the Student class. The client then sends it to the server using the object output stream through the socket. The server restores the object using the object input stream on the socket.

29.7 Retrieving Files from Web Servers

You developed client/server applications in the previous sections. Java allows you to develop clients that retrieve files on a remote host through a Web server. In this case, you don't have to create a custom server program. The Web server can be used to send the files, as shown in Figure 29.10.

******Same as Fig21.12 in Bonus Chapter 21 in introjb3e p65 (Bonus Chapters are in the CD-ROM in introjb3e)***

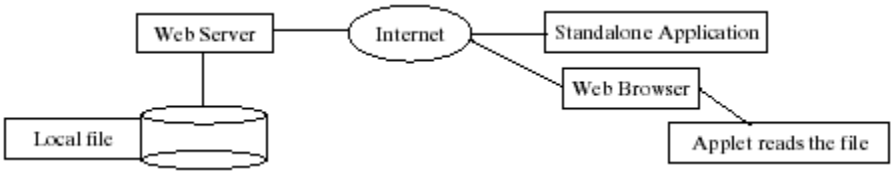


Figure 29.10

The applet client or the application client retrieves files from a Web server.

To retrieve the file, first create a URL object for the file. The java.net.URL class was introduced in Section 20.2, "The URL Class." For example, the following statement creates a URL object for http://www.cs.armstrong.edu/liang/index.html.

```
URL url = new URL("http://www.cs.armstrong.edu/liang/index.html");
```

You can then use the openStream() method defined in the URL class to open an input stream to the file's URL.

```
InputStream inputStream = url.openStream();
```

Now you can read the data from the input stream.

Example 29.5

Retrieving Remote Files

Problem

This example, which is similar to Example 16.5, "Displaying a File in a Text Area," demonstrates how to retrieve a file from a Web server. The program can run as an application or an applet. The user interface includes a text field in which to enter the

URL of the filename, a text area in which to show the file, and a button that can be used to submit an action. A label is added at the bottom of the applet to indicate the status, such as File loaded successfully or Network connection problem. A sample run of the program is shown in Figure 29.11.

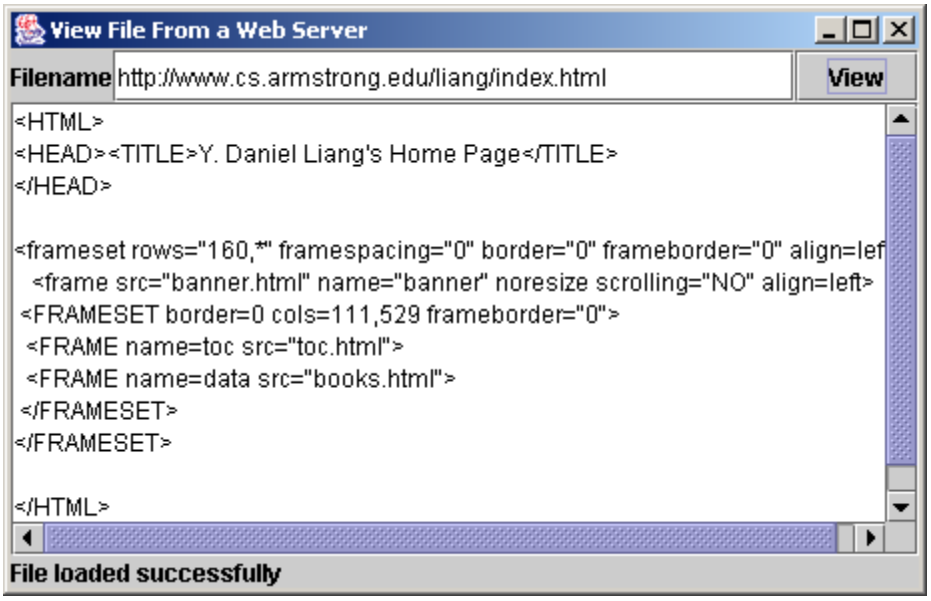


Figure 29.11

The program displays the contents of a specified file on the Web server.

Solution

The following code gives the solution to the problem.

******PD: Please add line numbers in the following code******

```
// ViewRemoteFile.java: Retrieve remote files
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class ViewRemoteFile extends JApplet
  implements ActionListener {
  // Button to view the file
  private JButton jbtView = new JButton("View");

  // Text field to receive file name
  private JTextField jtfURL = new JTextField(12);

  // Text area to store file
  private JTextArea jtaFile = new JTextArea();
```

```
  // Label to display status
  private JLabel jlblStatus = new JLabel();

  /** Initialize the applet */
  public void init() {
    // Create a panel to hold a label, a text field, and a button
    JPanel p1 = new JPanel();
    p1.setLayout(new BorderLayout());
    p1.add(new JLabel("Filename"), BorderLayout.WEST);
    p1.add(jtfURL, BorderLayout.CENTER);
    p1.add(jbtView, BorderLayout.EAST);

    // Place text area and panel p to the applet
    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(new JScrollPane(jtaFile),
      BorderLayout.CENTER);
    getContentPane().add(p1, BorderLayout.NORTH);
    getContentPane().add(jlblStatus, BorderLayout.SOUTH);

    // Register listener
    jbtView.addActionListener(this);
  }

  /** Handle the "View" button */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == jbtView)
      showFile();
  }

  private void showFile() {
    // Declare buffered stream for reading text for the URL
    BufferedReader infile = null;
    URL url = null;

    try {
      // Obtain URL from the text field

      url = new URL(jtfURL.getText().trim());

      // Create a buffered stream
      InputStream is = url.openStream();
      infile = new BufferedReader(new InputStreamReader(is));

      // Get file name from the text field
      String inLine;

      // Read a line and append the line to the text area
      while ((inLine = infile.readLine()) != null) {
        jtaFile.append(inLine + '\n');
      }

      jlblStatus.setText("File loaded successfully");
    }
    catch (FileNotFoundException e) {
      jlblStatus.setText("URL " + url + " not found.");
```

```
    }
    catch (IOException e) {
      jlblStatus.setText(e.getMessage());
    }
    finally {
      try {
        if (infile != null) infile.close();
      }
      catch (IOException ex) {}
    }
  }
}
```

Review

Line 55 new URL(jtfURL.getText().trim()) creates an URL for the filename entered from the text field. Line 59 url.openStream() creates an InputStream from the URL. After the input stream is established, reading data from the remote file is just like reading data locally. A BufferReader object is created from the input stream (Line 60). The text from the file is displayed in the text area (Lines 61-67).

This program can run either as an applet or as an application. In order for it to run as an applet, the user would need to place two files on the Web server:

ViewRemoteFile.class

ViewRemoteFile.html

ViewRemoteFile.html can be browsed from any Java 2-enabled Web browser.

29.8 JEditorPane

Swing provides a GUI component named javax.swing.JEditorPane that can be used to display plain text, HTML, and RTF files automatically. So you don't have to write code to explicit read data from the files. JEditorPane is a subclass of JTextComponent. Thus it inherits all the behavior and properties of JTextComponent.

To display the content of a file, use the setPage(URL) method as follows:

```
public void setPage(URL url) throws IOException
```

JEditorPane generates javax.swing.event.HyperlinkEvent when a hyperlink in the editor pane is clicked. Through this event, you can get the URL of the hyperlink and display it using the setPage(url) method.

Example 29.6

Creating a Web Browser

Problem

Create a simple Web browser to render HTML files. The program lets the user enter an HTML file in a text field and press the Enter key to display it in an editor pane, as shown in Figure 29.12.

***Same as Fig21.14 in Bonus Chapter 21 in introjb3e p69 (Bonus Chapters are in the CD-ROM in introjb3e)

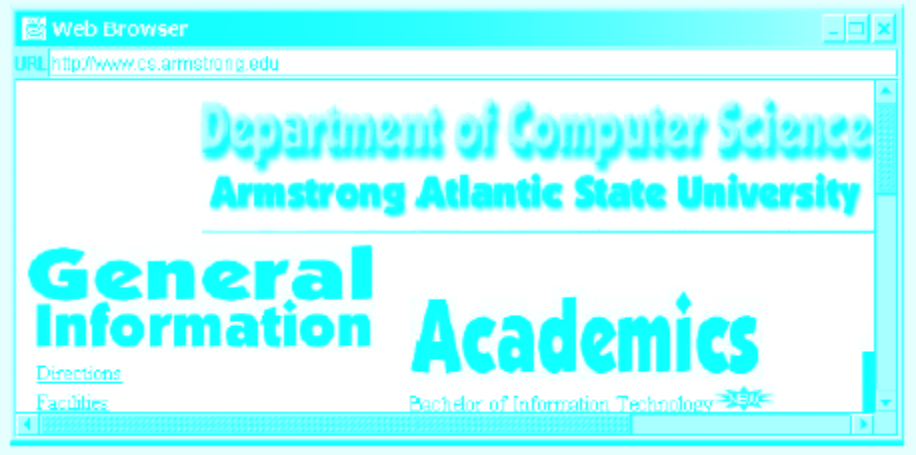


Figure 29.12

You can specify a URL in the text field and display the HTML file in an editor pane.

Solution

The following code gives the solution to the problem.

PD: Please add line numbers in the following code

```
// WebBrowser.java: Display HTML file in JEditorPane
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;
import java.net.URL;
import javax.swing.event.*;
import java.io.*;

public class WebBrowser extends JApplet
  implements ActionListener, HyperlinkListener {
  // JEditor pane to view HTML files
  private JEditorPane jep = new JEditorPane();

  // Label for URL
  private JLabel jlblURL = new JLabel("URL");

  // Text field for entering URL
  private JTextField jtfURL = new JTextField();

  /** Initialize the applet */
  public void init() {
    // Create a panel jpURL to hold the label and text field
    JPanel jpURL = new JPanel();
    jpURL.setLayout(new BorderLayout());
    jpURL.add(jlblURL, BorderLayout.WEST);
    jpURL.add(jtfURL, BorderLayout.CENTER);

    // Create a scroll pane to hold JEditorPane
    JScrollPane jspViewer = new JScrollPane();
```

```
    jspViewer.getViewport().add(jep, null);

    // Place jpURL and jspViewer in the applet
    this.getContentPane().add(jspViewer, BorderLayout.CENTER);
    this.getContentPane().add(jpURL, BorderLayout.NORTH);

    // Set jep noneditable
    jep.setEditable(false);

    // Register listener
    jep.addHyperlinkListener(this);
    jtfURL.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    try {
      // Get the URL from text field
      URL url = new URL(jtfURL.getText().trim());

      // Display the HTML file
      jep.setPage(url);
    }
    catch (IOException ex) {
      System.out.println(ex);
    }
  }

  public void hyperlinkUpdate(HyperlinkEvent e) {
    try {
      jep.setPage(e.getURL());
    }
    catch (IOException ex) {
      System.out.println(ex);
    }
  }
}
```

Review

In this example, a simple Web browser is created using the JEditorPane class. JEditorPane is capable of displaying files in HTML format. To enable scrolling, the editor pane is placed inside a scroll pane.

The user enters a URL of the HTML file in the text field and presses the Enter key to fire an action event to display the URL in the editor pane. To display the URL in the editor pane, simply set the URL in the page property of the editor pane.

The editor pane does not have all the functions of a commercial Web browser, but it is convenient for displaying HTML files, including embedded images.

This program cannot view a local HTML file. To view a remote HTML file, you have to enter a URL beginning with http://. In Exercise 29.8, you will modify the program so that it can also view an HTML file from the local host and accept URLs beginning with either http:// or www.

29.9 Cases Studies (Optional)

In Section 14.9, "Case Studies," you developed an applet for the TicTacToe game that enables two players to play from the same machine. In this section, you will learn how to develop a distributed TicTacToe game using multithreads and networking with socket streams.

Example 29.7

Distributed TicTacToe Game

Problem

Example 14.4, "The TicTacToe Game," lets the user to play the game from the same machine. Create a distributed TicTacToe game that enables users to play on different machines from anywhere on the Internet.

Solution

The example consists of 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 29.13.

******Same as Fig21.15 in Bonus Chapter 21 in introjb3e p71 (Bonus Chapters are in the CD-ROM in introjb3e)***

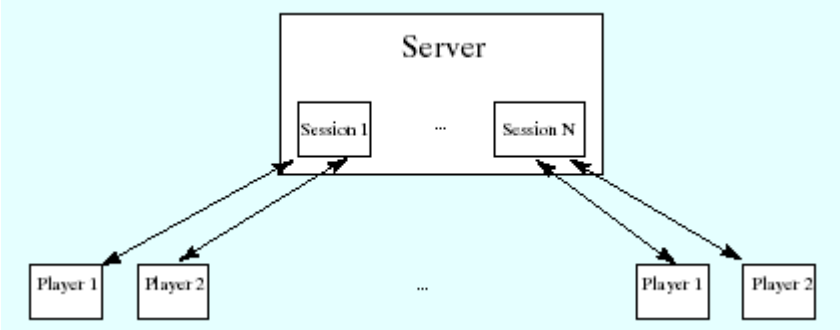


Figure 29.13

The server can create many sessions, each of which facilitates a TicTacToe game for two players.

For each session, the first client connecting to the server is identified as player 1 with token 'X', and the second client connecting to the server 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 29.14.

******Same as Fig21.16 in Bonus Chapter 21 in introjb3e p72 (Bonus Chapters are in the CD-ROM in introjb3e)***

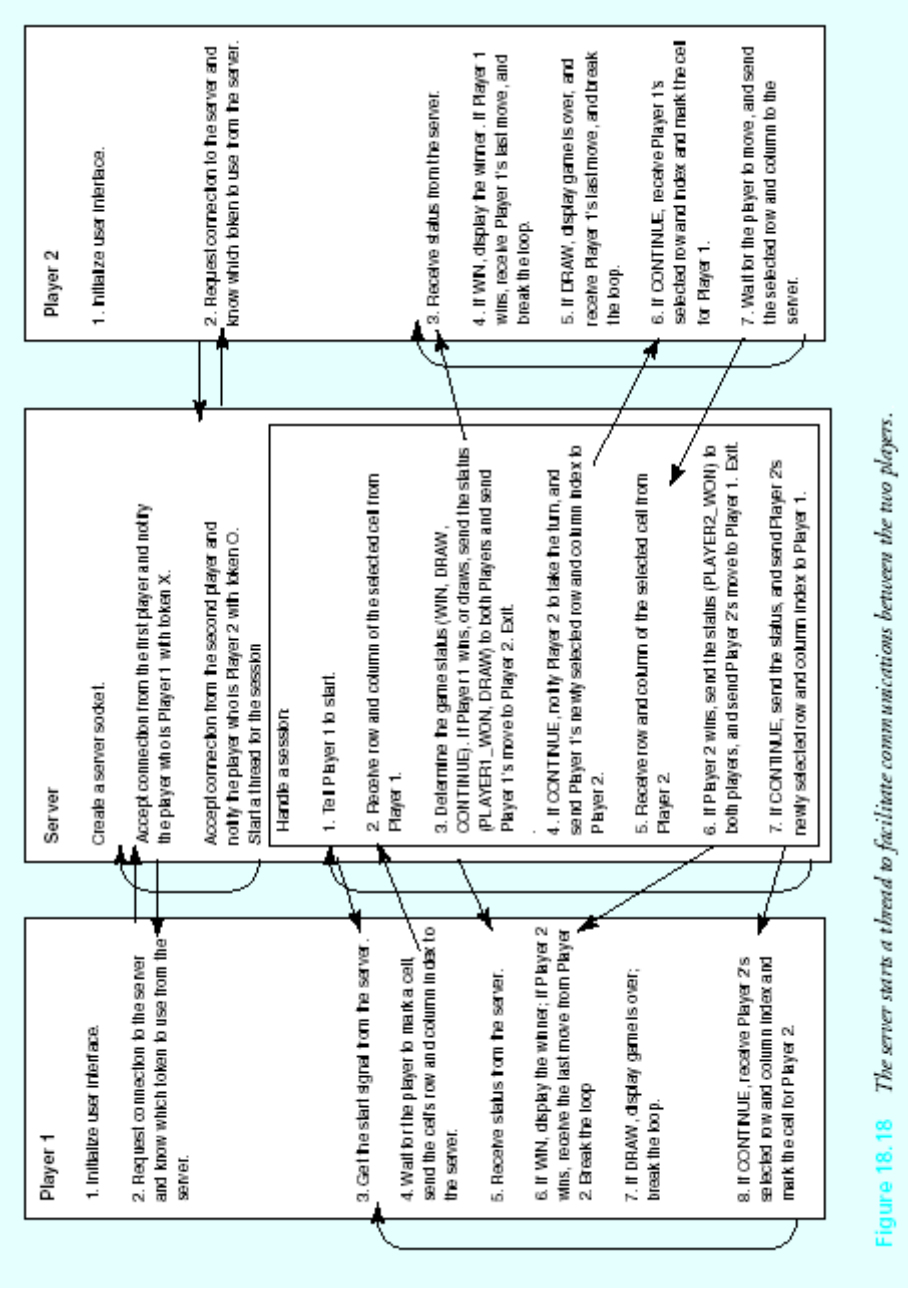


Figure 18.18 *The server starts a thread to facilitate communications between the two players.*

Figure 29.14

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

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

The client is responsible for interacting with the players. It creates a user interface with nine cells,

and displays the game title and status to the players in the labels. The client class is very similar to the TicTacToe class presented in Example 14.4, "The TicTacToe Game." 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 game status from the server.

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

[BL]**TicTacToeServer** serves all the clients.

[BL]**HandleASession** facilitates the game for two players.

[BL]**TicTacToeClient** models a player.

[BL]**Cell** models a cell in the game.

[BL]**TicTacToeConstants** is an interface that defines the constants shared by all the classes in the example.

The relationships of these classes are shown in Figure 29.15.

******Same as Fig21.17 in Bonus Chapter 21 in introjb3e p74 (Bonus Chapters are in the CD-ROM in introjb3e)***

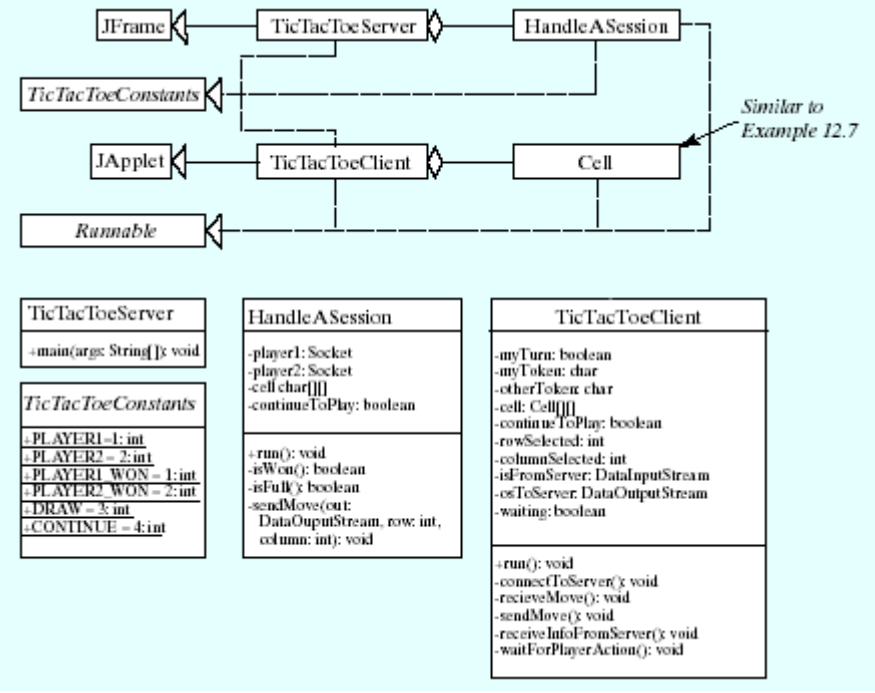


Figure 29.15

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

The program is given as follows:

PD: Please add line numbers in the following code

```
// TicTacToeConstants.java: Define constants for the classes
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
}
```

***Layout: Insert a separator line here. AU

PD: Please add line numbers in the following code

```
// TicTacToeServer.java: Server for the TicTacToe game
import java.io.*;
import java.net.*;
import javax.swing.*;
import java.awt.*;
import java.util.Date;

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

  public TicTacToeServer() {
    JTextArea jtaLog;

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

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

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

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

      // Number a session
      int sessionNo = 1;

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

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

        jtaLog.append(new Date() + ": Player 1 joined session " +
          sessionNo + '\n');
        jtaLog.append("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();

        jtaLog.append(new Date() +
          ": Player 2 joined session " + sessionNo + '\n');
        jtaLog.append("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
        jtaLog.append(new Date() + ": Start a thread for session " +
          sessionNo++ + '\n');

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

        // Start the new thread
        thread.start();
      }
    }
    catch(IOException ex) {
      System.err.println(ex);
    }
  }
}

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

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

  private DataInputStream isFromPlayer1;
  private DataOutputStream osToPlayer1;
  private DataInputStream isFromPlayer2;
  private DataOutputStream osToPlayer2;

  // 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 isFromPlayer1 = new DataInputStream(
        player1.getInputStream());
      DataOutputStream osToPlayer1 = new DataOutputStream(
        player1.getOutputStream());
      DataInputStream isFromPlayer2 = new DataInputStream(
        player2.getInputStream());
      DataOutputStream osToPlayer2 = new DataOutputStream(
        player2.getOutputStream());

      // Write anything to notify player 1 to start
      // This is just to let player 1 know to start
      osToPlayer1.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 = isFromPlayer1.readInt();
        int column = isFromPlayer1.readInt();
        cell[row][column] = 'X';
```

```
        // Check if Player 1 wins
        if (isWon('X')) {
          osToPlayer1.writeInt(PLAYER1_WON);
          osToPlayer2.writeInt(PLAYER1_WON);
          sendMove(osToPlayer2, row, column);
          break; // Break the loop
        }
        else if (isFull()) { // Check if all cells are filled
          osToPlayer1.writeInt(DRAW);
          osToPlayer2.writeInt(DRAW);
          sendMove(osToPlayer2, row, column);
          break;
        }
        else {
          // Notify player 2 to take the turn
          osToPlayer2.writeInt(CONTINUE);

          // Send player 1's selected row and column to player 2
          sendMove(osToPlayer2, row, column);
        }

        // Receive a move from Player 2
        row = isFromPlayer2.readInt();
        column = isFromPlayer2.readInt();
        cell[row][column] = 'O';

        // Check if Player 2 wins
        if (isWon('O')) {
          osToPlayer1.writeInt(PLAYER2_WON);
          osToPlayer2.writeInt(PLAYER2_WON);
          sendMove(osToPlayer1, row, column);
          break;
        }
        else {
          // Notify player 1 to take the turn
          osToPlayer1.writeInt(CONTINUE);

          // Send player 2's selected row and column to player 1
          sendMove(osToPlayer1, row, column);
        }
      }
    }
    catch(IOException ex) {
      System.err.println(ex);
    }
  }

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

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

    // All cells are filled
    return true;
  }

  /** Determine if the player with the specified token wins */
  private boolean isWon(char token) {
    // Check all rows
    for (int i = 0; i < 3; i++)
      if ((cell[i][0] == token)
          && (cell[i][1] == token)
          && (cell[i][2] == token)) {
        return true;
      }

    /** Check all columns */
    for (int j = 0; j < 3; j++)
      if ((cell[0][j] == token)
          && (cell[1][j] == token)
          && (cell[2][j] == token)) {
        return true;
      }

    /** Check major diagonal */
```

```
    if ((cell[0][0] == token)
        && (cell[1][1] == token)
        && (cell[2][2] == token)) {
      return true;
    }

    /** Check subdiagonal */
    if ((cell[0][2] == token)
        && (cell[1][1] == token)
        && (cell[2][0] == token)) {
      return true;
    }

    /** All checked, but no winner */
    return false;
  }
}
```

***Layout: Insert a separator line here. AU

PD: Please add line numbers in the following code

```
// TicTacToeClient.java: Play the TicTacToe game
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.LineBorder;
import java.io.*;
import java.net.*;

public class TicTacToeClient extends JApplet
  implements Runnable, 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 JLabel jlblTitle = new JLabel();

  // Create and initialize a status label
  private JLabel jlblStatus = new JLabel();

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

  // Input and output streams from/to server
  private DataInputStream inputFromServer;
  private DataOutputStream outputToServer;

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

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

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

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

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

    // Connect to the server
    connectToServer();
  }

  private void connectToServer() {
    try {
      // Create a socket to connect to the server
      Socket socket = new Socket("localhost", 8000);

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

      // Create an output stream to send data to the server
      outputToServer = new DataOutputStream(socket.getOutputStream());
    }
    catch (Exception ex) {
      System.err.println(ex);
    }

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

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

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

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

        // The other player has joined
        jlblStatus.setText("Player 2 has joined. I start first");

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

      // Continue to play
      while (continueToPlay) {
        if (player == PLAYER1) {
          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) {
    }
  }

  /** 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 {
    outputToServer.writeInt(rowSelected); // Send the selected row
    outputToServer.writeInt(columnSelected); // Send the selected column
  }

  /** Receive info from the server */
  private void receiveInfoFromServer() throws IOException {
    // Receive game status
    int status = inputFromServer.readInt();

    if (status == PLAYER1_WON) {
      // Player 1 won, stop playing
      continueToPlay = false;
      if (myToken == 'X') {
        jlblStatus.setText("I won! (X)");
      }
      else if (myToken == 'O') {
        jlblStatus.setText("Player 1 (X) has won!");
        receiveMove();
      }
    }
    else if (status == PLAYER2_WON) {
      // Player 2 won, stop playing
      continueToPlay = false;
      if (myToken == 'O') {
        jlblStatus.setText("I won! (O)");
      }
      else if (myToken == 'X') {
        jlblStatus.setText("Player 2 (O) has won!");
        receiveMove();
      }
    }
    else if (status == DRAW) {
      // No winner, game is over
      continueToPlay = false;
      jlblStatus.setText("Game is over, no winner!");

      if (myToken == 'O') {
        receiveMove();
      }
    }
    else {
      receiveMove();
      jlblStatus.setText("My turn");
      myTurn = true; // It is my turn
    }
  }

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

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

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

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

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

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

```
    /** Paint the cell */
    public void paintComponent(Graphics g) {
      super.paintComponent(g);

      if (token == 'X') {
        g.drawLine(10, 10, getWidth() - 10, getHeight() - 10);
        g.drawLine(getWidth() - 10, 10, 10, getHeight() - 10);
      }
      else if (token == 'O') {
        g.drawOval(10, 10, getWidth() - 20, getHeight() - 20);
      }
    }

    /** Handle mouse click on a cell */
    public void mouseClicked(MouseEvent e) {
      // 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;
        jlblStatus.setText("Waiting for the other player to move");
        waiting = false; // Just completed a successful move
      }
    }

    public void mousePressed(MouseEvent e) {
      // TODO: implement this java.awt.event.MouseListener method;
    }

    public void mouseReleased(MouseEvent e) {
      // TODO: implement this java.awt.event.MouseListener method;
    }

    public void mouseEntered(MouseEvent e) {
      // TODO: implement this java.awt.event.MouseListener method;
    }

    public void mouseExited(MouseEvent e) {
      // TODO: implement this java.awt.event.MouseListener method;
    }
  }
}
```

Review

The server can serve any number of sessions. Each session takes care of two players. The client can be a Java applet or a Java application. To run a client as a Java applet from a Web browser, the server must run from a Web server. Figures 29.16 and 29.17 show sample runs of the server and the clients.

******Same as Fig21.18 in Bonus Chapter 21 in introjb3e p83 (Bonus Chapters are in the CD-ROM in introjb3e)***

Figure 29.16

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

Same as Fig21.19 in Bonus Chapter 21 in introjb3e p83 (Bonus Chapters are in the CD-ROM in introjb3e)

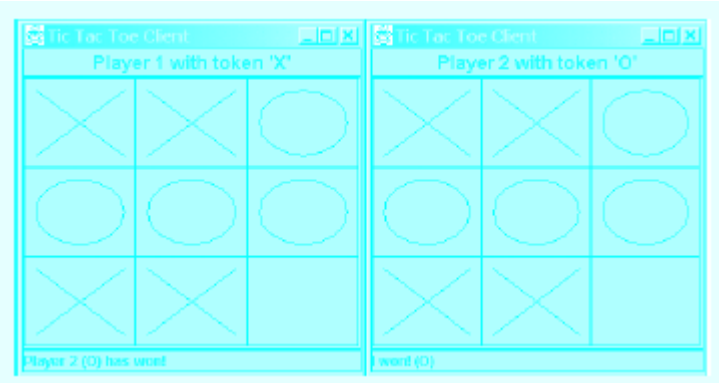


Figure 29.17

TicTacToeClient can run as an applet or an application.

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

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

29.10 Datagram Socket

Clients and servers that communicate via a stream socket have a dedicated point-to-point channel between them. To communicate, they establish a connection, transmit the data, and then close the connection. The stream sockets use TCP (Transmission Control Protocol) for data transmission. Since TCP can detect lost transmissions and resubmit them, transmissions are lossless and reliable. All data sent via a stream socket are received in the same order in which they were sent.

In contrast, clients and servers that communicate via a datagram socket do not have a dedicated point-to-point channel. Data are transmitted using packets. Datagram sockets use UDP (User Datagram Protocol), which cannot guarantee that the packets are not lost, or received in duplicate, or received in the order in which they were sent. A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

In an analogy, a stream socket communication between a client and a server is like a telephone connection with a dedicated link. A datagram communication is like sending a letter through the postal office. Your letter is contained in an envelope (packet). If the letter is too large, it may be sent in several envelopes (packets). There is no guarantee that your letter will arrive, or arrive in the order it was sent. One difference is that the letter will not arrive in duplicate, whereas a datagram packet may arrive in duplicate.

Most applications require reliable transmission between clients and servers. In such cases it is best to use stream socket network communication. Some applications that you write to communicate over the network will not require the reliable, point-to-point channel provided by TCP. In such cases datagram communication is more efficient.

29.10.1 The DatagramPacket and DatagramSocket Classes

The java.net package contains two classes to help you write Java programs that use datagrams to send and receive packets over the network: DatagramPacket and DatagramSocket. An application can send and receive DatagramPackets through a DatagramSocket.

29.10.1.1 The DatagramPacket Class

The DatagramPacket class represents a datagram packet. Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within the packet. Multiple packets sent from one machine to another may be routed differently and may arrive in any order. Packet delivery is not guaranteed.

To create a DatagramPacket for delivery from a client, use the DatagramPacket(byte[] buf, int length, InetAddress host, int port) constructor, and to create all other DatagramPacket, use the DatagramPacket(byte[] buf, int length) constructor, as shown in Figure 29.18. Once a datagram packet is created, you can use the getData and setData methods to obtain and set data in the packet.

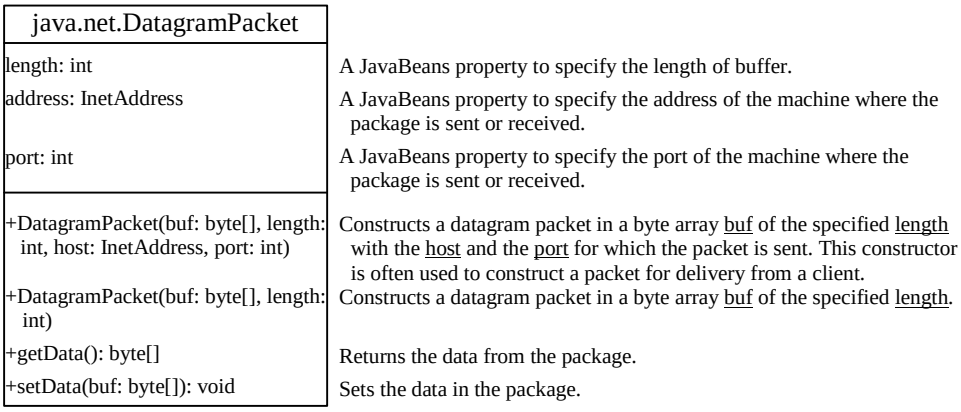


Figure 29.18

The DatagramPacket class contains the data and information about data.

29.10.1.2 DatagramSocket

The DatagramSocket class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

To create a server DatagramSocket, use the constructor DatagramSocket(int port), which binds the socket with the specified port on the local host machine.

To create a client DatagramSocket, use the constructor DatagramSocket(), which binds the socket with any available port on the local host machine.

To send data, you need to create a packet, fill in the contents, specify the Internet address and port number for the receiver, and invoke the send(packet) method on a DatagramSocket.

To receive data, create an empty packet and invoke the receive(packet) method on a DatagramSocket.

29.10.3 Datagram Programming

Datagram programming is different from stream socket programming in the sense that there is no concept of a ServerSocket for datagrams. Both client and server use DatagramSocket to send and receive packets, as shown in Figure 29.19.

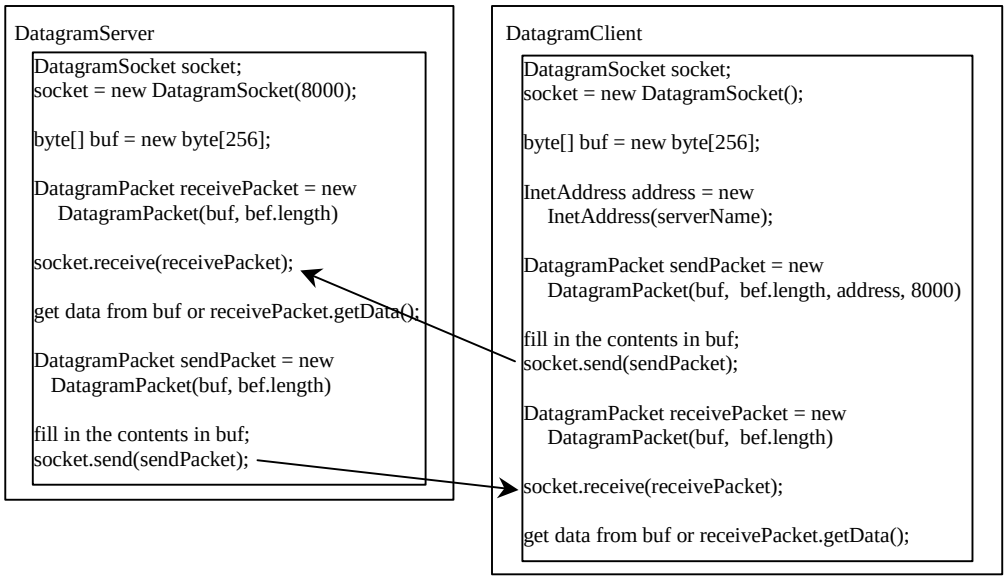


Figure 29.19

The programs send and receive packets via datagram sockets.

Normally, you designate one application as the server and create a DatagramSocket with the specified port using the constructor DatagramSocket(port). A client can create a DatagramSocket without specifying a port number. The port number will be dynamically chosen at runtime. When a client sends a packet to the server, the client's IP address and port number are contained in the packet. The server can retrieve it from the packet and use it to send the packet back to the client.

To demonstrate, let us rewrite Example 29.1, "A Client/Server Example," using datagrams.

Example 29.8

Client/Server Programming Using Datagrams

Problem

Example 29.1 presents a client program and a server program using socket streams. The client sends radius to a server. The server receives the data, uses them to find the area, and then sends the area to the client. Rewrite the program using datagram sockets.

Solution

The server and client programs follow. A sample run of the program is shown in Figure 29.20.

***Same as FigJ.2 in Supplement J in introjb3e p314 (Supplements are in the CD-ROM in introjb3e)

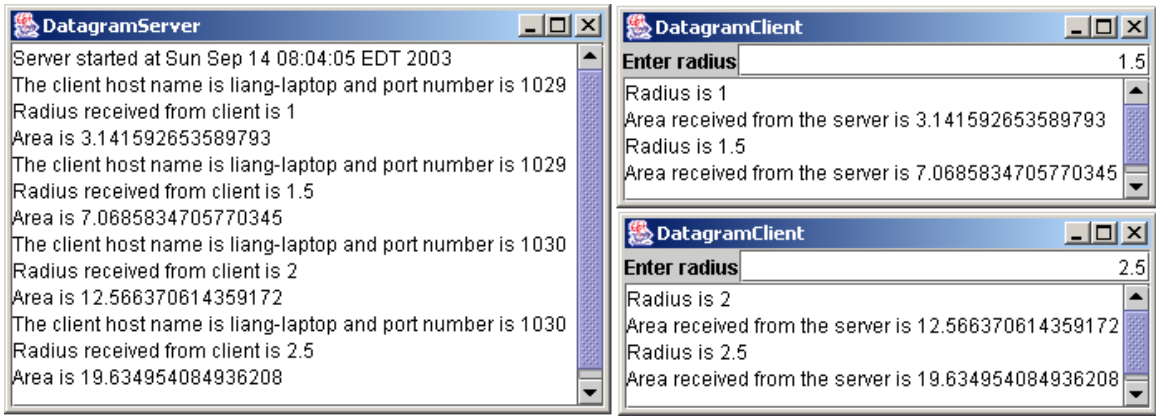


Figure 29.20

The server receives a radius from a client, computes the area, and sends the area to the client. The server can serve multiple clients.

PD: Please add line numbers in the following code

```
// DatagramServer.java: The server accepts data from the client, processes it
// and returns the result back to the client using datagram packet
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

  // The byte array for sending and receiving datagram packets
  private byte[] buf = new byte[256];

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

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

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

    try {
```

```
      // Create a server socket
      DatagramSocket socket = new DatagramSocket(8000);
      jta.append("Server started at " + new Date() + '\n');

      // Create a packet for receiving data
      DatagramPacket receivePacket =
        new DatagramPacket(buf, buf.length);

      // Create a packet for sending data
      DatagramPacket sendPacket =
        new DatagramPacket(buf, buf.length);

      while (true) {
        // Initialize buffer for each iteration
        Arrays.fill(buf, (byte)0);

        // Receive radius from the client in a packet
        socket.receive(receivePacket);
        jta.append("The client host name is " +
          receivePacket.getAddress().getHostName() +
          " and port number is " + receivePacket.getPort() + '\n');
        jta.append("Radius received from client is " +
          new String(buf).trim() + '\n');

        // Compute area
        double radius = Double.parseDouble(new String(buf).trim());
        double area = radius * radius * Math.PI;
        jta.append("Area is " + area + '\n');

        // Send area to the client in a packet
        sendPacket.setAddress(receivePacket.getAddress());
        sendPacket.setPort(receivePacket.getPort());
        sendPacket.setData(new Double(area).toString().getBytes());
        socket.send(sendPacket);
      }
    }
    catch(IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

PD: Please insert a separate line

PD: Please add line numbers in the following code

```
// DatagramClient.java: The client sends the input to the server and receives
// result back from the server using datagram socket
import java.io.*;
import java.net.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class DatagramClient extends JFrame implements ActionListener {
  // Text field for receiving radius
  private JTextField jtf = new JTextField();

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

  // Datagram socket
  private DatagramSocket socket;

  // The byte array for sending and receiving datagram packets
  private byte[] buf = new byte[256];

  // Server InetAddress
  private InetAddress address;

  // The packet sent to the server
  private DatagramPacket sendPacket;

  // The packet received from the server
  private DatagramPacket receivePacket;

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

  public Client() {
    // Panel p to hold the label and text field
    JPanel p = new JPanel();
    p.setLayout(new BorderLayout());
    p.add(new JLabel("Enter radius"), BorderLayout.WEST);
    p.add(jtf, BorderLayout.CENTER);
    jtf.setHorizontalAlignment(JTextField.RIGHT);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(p, BorderLayout.NORTH);
    getContentPane().add(new JScrollPane(jta), BorderLayout.CENTER);

    jtf.addActionListener(this); // Register listener

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

    try {
      // get a datagram socket
      socket = new DatagramSocket();
      address = InetAddress.getByName("localhost");
      sendPacket =
        new DatagramPacket(buf, buf.length, address, 8000);
      receivePacket = new DatagramPacket(buf, buf.length);
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
```

```
  }

  public void actionPerformed(ActionEvent e) {
    String actionCommand = e.getActionCommand();
    if (e.getSource() instanceof JTextField) {
      try {
        // Initialize buffer for each iteration
        Arrays.fill(buf, (byte)0);

        // send radius to the server in a packet
        sendPacket.setData(jtf.getText().trim().getBytes());
        socket.send(sendPacket);

        // receive area from the server in a packet
        socket.receive(receivePacket);

        // Display to the text area
        jta.append("Radius is " + jtf.getText().trim() + "\n");
        jta.append("Area received from the server is "
          +  Double.parseDouble(new String(buf).trim()) + '\n');
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

Review

Since datagrams are connectionless, a DatagramPacket can be sent to multiple clients, and multiple clients can receive a packet from the same server. As shown in this example, you can launch multiple clients. Each client sends the radius to the server, and the server sends the area back to the client.

The server creates a DatagramSocket on port 8000 (Line 35 in Server.java). No DatagramSocket can be created again on the same port. The client creates a DatagramSocket on an available port (Line 59 in Client.java). The port number is dynamically assigned to the socket. You can launch multiple clients simultaneously, and each client's datagram socket will be different.

The client creates a DatagramPacket named sendPacket for delivery to the server (Lines 61-62 in Client.java). The DatagramPacket contains the server address and port number. The client creates another DatagramPacket named receivePacket (Line 63), which is used for receiving packets from the server. This packet does not need to contain any address or port number.

A user enters a radius in the text field in the client. Upon pressing the Enter key on the text field, the radius value in the text field is put into the packet and sent to the server (Lines 78-79 in Client.java). The server receives the packet (Line 51 in Server.java), extracts the data from the byte array buf, and computes the area (Lines 59-60 in Server.java). The server then builds a packet that contains the area value in the buffer, the client's address, and the port number, and sends the packet to the client (Lines 64-67). The client receives the packet (Line 82) and displays the result in the text area.

The data in the packet are stored in a byte array. To send a numerical value, you need to convert it into a string and then store it in the array as bytes, using the getBytes() method in the String class (Line 66 in DatagramServer.java and Line 78 in DatagramClient.java). To convert the array into a number, first convert it into a string, and then convert it into a number using the static parseDouble method in the Double class (Line 59 in DatagramServer.java and Line 87 in DatagramClient.java).

NOTE: The port numbers for the stream socket and the datagram socket are not related. You can use the same port number for a stream socket and a datagram socket simultaneously.

Key Classes and Methods

- **java.net.ServerSocket** A class for creating a server socket. A server socket waits for request to come in over the network using the accept() method.
- **java.net.Socket** A class for creating a socket. A socket is an endpoint for communication between two machines. A client uses new Socket(serverName, port) to connect to a server socket. Use getInputStream() to obtain an InputStream and getOutputStream() to obtain an OutputStream on the socket.
- **java.net.InetAddress** A class that represents an Internet Protocol (IP) address. Use getHost() to get the host name and getHostAddress() to get host IP address.
- **java.net.URL** A class that represents a URL. Use new URL(urlString) to create a URL for an Internet resource. Use openStream() to get an InputStream from the URL. Use getHost() to return the host name of the URL.
- **javax.swing.JEditorPane** A subclass of JTextComponent. that represents a URL. Use new URL(urlString) to create a URL for an Internet resource. Use openStream() to get an InputStream from the URL. Use getHost() to return the host name of the URL.
- **java.net.DatagramPacket** A class that represents a datagram packet for sending and receiving packets. To create a DatagramPacket from a client, use new DatagramPacket(byte[] buf, int length, InetAddress host, int port). To create it

from the server, use new DatagramPacket(byte[] buf, int length). Use getData() and setData(byte[]) to obtain and set data in a packet.

- **java.net.DatagramSocket** A class that represents a datagram socket. Use new DatagramSocket(int port) to create a server socket and use new DatagramSocket() to create a client socket. Use send(DatagramPacket) to send a packet and use receive(DatagramPacket) to receive a packet.

Chapter Summary

- 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.

- 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.

- 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.

- 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.

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

- Java programs can retrieve data from a file on a remote host through a Web server. To do so, first create a URL object using new URL(urlString), then use openStream() to get an InputStream to read the data from the file.

- Swing provides a GUI component named javax.swing.JEditorPane that can be used to display text, HTML, and RTF files automatically without writing the code to read data from the file explicitly.

- Clients and servers that communicate via a datagram socket do not have a dedicated point-to-point channel. Data are transmitted using packets. Datagram sockets use UDP (User Datagram Protocol), which cannot guarantee that the packets are not lost, or received in duplicate, or received in the order in which they were sent. A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

Review Questions

Section 29.2 Client/Server Computing

29.1

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?

29.2

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

29.3

How does a client program initiate a connection?

29.4

How does a server accept a connection?

29.5

How are data transferred between a client and a server?

Sections 29.3-29.4

29.6

How do you find the IP address of a client that connects to a server?

29.7

How do you make a server serve multiple clients?

Sections 29.5-29.6

29.8

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

29.9

How do you find the host name of an applet?

29.10

How do you send and receive an object?

Sections 29.7-29.8

29.11

Can an application retrieve a file from a remote host? Can an application update a file on a remote host?

29.12

How do you retrieve a file from a Web server?

29.13

What types of files can be displayed in a JEditorPane? How do you display a file in a JEditorPane?

Section 29.10 Datagram Socket

29.14

What are the differences between stream sockets and datagram sockets? How do you create a Datagram socket? How do set data in the packet? How do you send and receive packets? How do you find the IP address of the sender?

Programming Exercises

Sections 29.2-29.4

29.1

(Computing loans on a server) Write a server for multiple clients. The client sends loan information (annual interest rate, number of years, and loan amount) to the server (see Figure 29.21). The server computes monthly payment and total payment and sends them back to the client (see Figure 29.18). Name the client Exercise29_1Client and the server Exercise29_1Server.

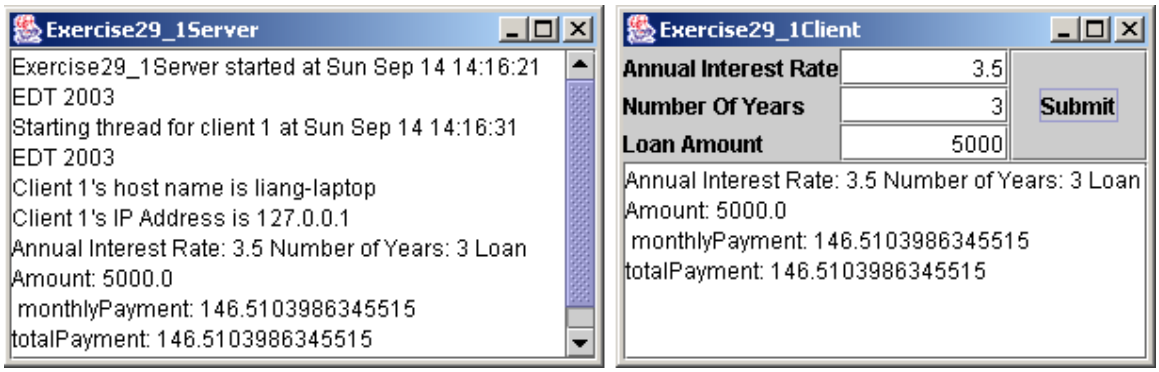


Figure 29.21

The client 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.

29.2 (Revising Example 29.1 "A Client/Server Example") Rewrite Example 29.1 using a buffered reader for input and a print stream for output. Use the readLine() method to read a string from the input stream, and use the Double.parseDouble(string) to convert the string into a double value. Name the client Exercise29_2Client and the server Exercise29_2Server.

Sections 29.5-29.9

29.3 (Revising Example 29.3 "Creating Applet Clients") Modify Example 29.3 by adding the following features:

[BL] Add a View button to the user interface that allows the client to view a record for a specified name. The user will be able to enter a name in the Name field and click the View button to display the record for the student, as shown in Figure 29.22.

[BL] Limit the concurrent connections to two clients.

[BL] Display the status of the submission, such as Successful, Failed, Record found, or Record not found, on a label.

[BL] Use one thread to handle all requests from a client. The thread terminates when the client exits.

Name the client Exercise29_3Client and the server Exercise29_3Server.

Figure 29.22

You can view or register students in this applet.

29.4

(Revising Example 29.6 "Retrieving Remote Files") Rewrite Example 29.6 to use JEditorPane instead of JTextArea.

29.5

(Counting Web site visits) Write an applet that shows the number of visits made to a Web page. The count should be stored in a file on the server side. Every time the page is visited or reloaded, the applet sends a request to the server, and the server increases the count and sends it to the applet. The applet then displays the new count in a message, such as You are visitor number: 1000, as shown in Figure 29.23. Name the client Exercise29_6Client and the server Exercise29_6Server.

HINT

Use a random-access file stream to read or write the count to the file. To improve performance, read the count from the file when the server starts, and save the count to the file when the server exits. When the server is alive, use a variable to store the count. When a client applet starts, it connects to the server; the server increases the count by 1 and sends it to the client for display.

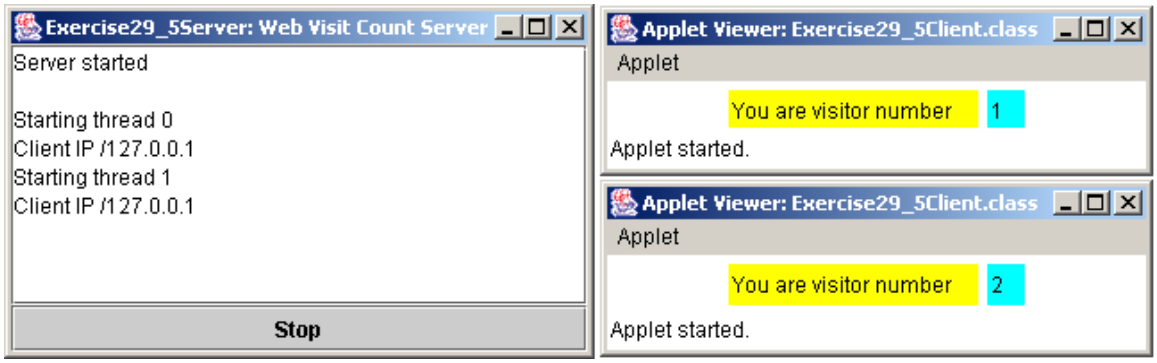


Figure 29.23

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

29.6

(Creating a stock ticker in an applet) Write an applet like the ones in Exercise 19.12. Ensure that the applet gets the stock index from a file stored on the Web server. Enable the applet to run standalone.

29.7

(Revising Example 29.7 “Creating a Web Browser”) Modify Example 29.7 as follows:

[BL]It accepts an HTML file from a local host. Assume that a local HTML filename begins neither with http:// nor with www.

[BL]It accepts a remote HTML file. A remote HTML filename begins with either http:// or www.

Section 29.10, Datagram Socket

29.8 (Datagram socket programming) Rewrite Exercise 29.1 using datagram sockets.