If the presence of electricity can be made visible in any part of a circuit, I see no reason why intelligence may not be transmitted instantaneously by electricity.
—Samuel F. B. Morse

Protocol is everything.
—Francois Giuliani

What networks of railroads, highways and canals were in another age, the networks of telecommunications, information and computerization … are today.
—Bruno Kreisky

The port is near, the bells I hear, the people all exulting.
—Walt Whitman

Objectives
In this chapter you’ll learn:

■ Java networking with URLs, sockets and datagrams.
■ To implement Java networking applications by using sockets and datagrams.
■ To implement Java clients and servers that communicate with one another.
■ To implement network-based collaborative applications.
■ To construct a simple multithreaded server.
27.1 Introduction

Java provides a number of built-in networking capabilities that make it easy to develop Internet-based and web-based applications. Java can enable programs to search the world for information and to collaborate with programs running on other computers internationally, nationally or just within an organization (subject to security constraints).

Java’s fundamental networking capabilities are declared by the classes and interfaces of package `java.net`, through which Java offers **stream-based communications** that enable applications to view networking as streams of data. The classes and interfaces of package `java.net` also offer **packet-based communications** for transmitting individual packets of information—commonly used to transmit data images, audio and video over the Internet. In this chapter, we show how to communicate with packets and streams of data.

We focus on both sides of the **client/server relationship**. The client requests that some action be performed, and the server performs the action and responds to the client. A common implementation of the request-response model is between web browsers and web servers. When a user selects a website to browse through a browser (the client application), a request is sent to the appropriate web server (the server application). The server normally responds to the client by sending an appropriate web page to be rendered by the browser.

We introduce Java’s **socket-based communications**, which enable applications to view networking as if it were file I/O—a program can read from a socket or write to a socket as simply as reading from a file or writing to a file. The socket is simply a software construct that represents one endpoint of a connection. We show how to create and manipulate stream sockets and datagram sockets.

With **stream sockets**, a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams. Stream sockets are said to provide a connection-oriented service. The protocol used for transmission is the popular TCP (Transmission Control Protocol).

With **datagram sockets**, individual packets of information are transmitted. The protocol used—UDP, the User Datagram Protocol—is a connectionless service and does not guarantee that packets arrive in any particular order. With UDP, packets can even be lost or duplicated. Significant extra programming is required on your part to deal with these problems (if you choose to do so). UDP is most appropriate for network applications that...
do not require the error checking and reliability of TCP. Stream sockets and the TCP protocol will be more desirable for the vast majority of Java networking applications.

Performance Tip 27.1
Connectionless services generally offer greater performance but less reliability than connection-oriented services.

Portability Tip 27.1
TCP, UDP and related protocols enable heterogeneous computer systems (i.e., those with different processors and different operating systems) to intercommunicate.

On the web at www.deitel.com/books/jhtp9/, we present a case study that implements a client/server chat application similar to popular instant-messaging services. The application introduces multicasting, in which a server can publish information and many clients can subscribe to it. When the server publishes information, all subscribers receive it.

27.2 Manipulating URLs

The Internet offers many protocols. The HyperText Transfer Protocol (HTTP), which forms the basis of the web, uses URIs (Uniform Resource Identifiers) to identify data on the Internet. URIs that specify the locations of websites and web pages are called URLs (Uniform Resource Locators). Common URLs refer to files or directories and can reference objects that perform complex tasks, such as database lookups and Internet searches. If you know the URL of a publicly available web page, you can access it through HTTP.

Java makes it easy to manipulate URLs. When you use a URL that refers to the exact location of a resource (e.g., a web page) as an argument to the showDocument method of interface AppletContext, the browser in which the applet is executing will access and display that resource. The applet in Figs. 27.1–27.2 demonstrates simple networking capabilities. It enables the user to select a web page from a JList and causes the browser to display the corresponding page. In this example, the networking is performed by the browser.

Processing Applet Parameters

This applet takes advantage of applet parameters specified in the HTML document that invokes the applet. When browsing the web, you’ll often come across applets that are in the public domain—you can use them free of charge on your own web pages (normally in exchange for crediting the applet’s creator). Many applets can be customized via parameters supplied from the HTML file that invokes the applet. For example, Fig. 27.1 contains the HTML that invokes the applet SiteSelector in Fig. 27.2.

```html
<html>
<head>
<title>Site Selector</title>
</head>
<body>
  <applet code = "SiteSelector.class" width = "300" height = "75">
    <param name = "title0" value = "Java Home Page">
    <param name = "location0"
</body>
</html>
```

Fig. 27.1 | HTML document to load SiteSelector applet. (Part 1 of 2.)
The HTML document contains eight parameters specified with the `param` element—these lines must appear between the starting and ending `applet` tags. The applet can read these values and use them to customize itself. Any number of `param` elements can appear between the starting and ending `applet` tags. Each parameter has a unique `name` and a `value`. Applet method `getParameter` returns the value associated with a specific parameter name as a `String`. The argument passed to `getParameter` is a `String` containing the name of the parameter in the `param` element. In this example, parameters represent the title and location of each website the user can select. Parameters specified for this applet are named `title#`, where the value of # starts at 0 and increments by 1 for each new title. Each title should have a corresponding location parameter of the form `location#`, where the value of # starts at 0 and increments by 1 for each new location. The statement

```java
String title = getParameter("title0");
```

gets the value associated with parameter "title0" and assigns it to reference `title`. If there's no `param` tag containing the specified parameter, `getParameter` returns `null`.

### Storing the Website Names and URLs

The applet (Fig. 27.2) obtains from the HTML document (Fig. 27.1) the choices that will be displayed in the applet's `JList`. Class `SiteSelector` uses a `HashMap` (package `java.util`) to store the website names and URLs. In this example, the `key` is the `String` in the `JList` that represents the website name, and the `value` is a `URL` object that stores the location of the website to display in the browser.

```java
// Fig. 27.2: SiteSelector.java
// Loading a document from a URL into a browser.
import java.net.MalformedURLException;
import java.net.URL;
import java.util.HashMap;
import java.util.ArrayList;
import java.awt.BorderLayout;
import java.awt.applet.AppletContext;
import javax.swing.JApplet;
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing JScrollPane;
import javax.swing.JList;
import javax.swing.JScrollPane;
```
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class SiteSelector extends JApplet {

private HashMap< String, URL > sites; // site names and URLs
private ArrayList< String > siteNames; // site names
private JList siteChooser; // list of sites to choose from

// read parameters and set up GUI
public void init() {
    sites = new HashMap< String, URL >(); // create HashMap
    siteNames = new ArrayList< String >(); // create ArrayList

    // obtain parameters from HTML document
    getSitesFromHTMLParameters();

    // create GUI components and lay out interface
    add( new JLabel( "Choose a site to browse" ), BorderLayout.NORTH );

    siteChooser = new JList( siteNames.toArray() ); // populate JList
    siteChooser.addListSelectionListener(
        new ListSelectionListener() // anonymous inner class
            {
                // go to site user selected
                public void valueChanged( ListSelectionEvent event ) {
                    // get selected site name
                    Object object = siteChooser.getSelectedValue();

                    // use site name to locate corresponding URL
                    URL newDocument = sites.get( object );

                    // get applet container
                    AppletContext browser = getAppletContext();

                    // tell applet container to change pages
                    browser.showDocument( newDocument );
                } // end method valueChanged
            } // end anonymous inner class
    ); // end call to addListSelectionListener

    add( new JScrollPane( siteChooser ), BorderLayout.CENTER );
} // end method init

// obtain parameters from HTML document
private void getSitesFromHTMLParameters() {
    String title; // site title
    String location; // location of site
    URL url; // URL of location
    int counter = 0; // count number of sites

    Fig. 27.2 | Loading a document from a URL into a browser. (Part 2 of 3.)
class SiteSelector
also contains an ArrayList (package java.util) in which the site names are placed so that they can be used to initialize the JList (one version of the JList constructor receives an array of Objects which is returned by ArrayList's toArray method). An ArrayList is a dynamically resizable array of references. Class ArrayList provides method add to add a new element to the end of the ArrayList. (ArrayList and HashMap were discussed in Chapter 20.)
Lines 25–26 in the applet’s init method (lines 23–57) create a HashMap object and an ArrayList object. Line 29 calls our utility method getSitesFromHTMLParameters (declared at lines 60–89) to obtain the HTML parameters from the HTML document that invoked the applet.

Method getSitesFromHTMLParameters uses Applet method getParameter (line 67) to obtain a website title. If the title is not null, lines 73–87 execute. Line 73 uses Applet method getParameter to obtain the website location. Line 77 uses the location as the value of a new URL object. The URL constructor determines whether its argument represents a valid URL. If not, the URL constructor throws a MalformedURLException. The URL constructor must be called in a try block. If the URL constructor generates a MalformedURLException, the call to printStackTrace (line 83) causes the program to output a stack trace to the Java console. On Windows machines, the Java console can be viewed by right clicking the Java icon in the notification area of the taskbar. On a Mac, go to Applications > Utilities and launch the Java Preferences app. Then on the Advanced tab under Java console, select Show console. On other platforms, this is typically accessible through a desktop icon. Then the program attempts to obtain the next website title. The program does not add the site for the invalid URL to the HashMap, so the title will not be displayed in the JList.

For a proper URL, line 78 places the title and URL into the HashMap, and line 79 adds the title to the ArrayList. Line 87 gets the next title from the HTML document. When the call to getParameter at line 87 returns null, the loop terminates.

**Building the Applet’s GUI**

When method getSitesFromHTMLParameters returns to init, lines 32–56 construct the applet’s GUI. Line 32 adds the JLabel “Choose a site to browse” to the NORTH of the JApplet’s BorderLayout. Line 34 creates JList siteChooser to allow the user to select a web page to view. Lines 35–54 register a ListSelectionListener to handle the JList’s events. Line 56 adds siteChooser to the CENTER of the JFrame’s BorderLayout.

**Processing a User Selection**

When the user selects a website in siteChooser, the program calls method valueChanged (lines 39–52). Line 42 obtains the selected site name from the JList. Line 45 passes the selected site name (the key) to HashMap method get, which locates and returns a reference to the corresponding URL object (the value) that’s assigned to reference newDocument.

Line 48 uses Applet method getAppletContext to get a reference to an AppletContext object that represents the applet container. Line 51 uses this reference to invoke method showDocument, which receives a URL object as an argument and passes it to the AppletContext (i.e., the browser). The browser displays in the current browser window the resource associated with that URL. In this example, all the resources are HTML documents.

**Specifying the Target Frame for Method showDocument**

A second version of AppletContext method showDocument enables an applet to specify the target frame in which to display the web resource. This takes as arguments a URL object specifying the resource to display and a String representing the target frame. There are some special target frames that can be used as the second argument. The target frame _blank results in a new web browser window to display the content from the specified URL. The target frame _self specifies that the content from the specified URL should be displayed in the same frame as the applet (the applet’s HTML page is replaced in this case).
27.3 Reading a File on a Web Server

The target frame _top specifies that the browser should remove the current frames in the browser window, then display the content from the specified URL in the current window.

Error-Prevention Tip 27.1

The applet in Fig. 27.2 must be run from a web browser to show the results of displaying another web page. The appletviewer is capable only of executing applets—it ignores all other HTML tags. If the websites in the program contained Java applets, only those applets would appear in the appletviewer when the user selected a website. Each applet would execute in a separate appletviewer window.

27.3 Reading a File on a Web Server

The application in Fig. 27.3 uses Swing GUI component JEditorPane (from package javax.swing) to display the contents of a file on a web server. The user enters a URL in the JTextField at the top of the window, and the application displays the corresponding document (if it exists) in the JEditorPane. Class JEditorPane is able to render both plain text and basic HTML-formatted text, as illustrated in the two screen captures (Fig. 27.4), so this application acts as a simple web browser. The application also demonstrates how to process HyperlinkEvents when the user clicks a hyperlink in the HTML document. The techniques shown in this example can also be used in applets. However, an applet is allowed to read files only on the server from which it was downloaded. [Note: This program might not work if your web browser must access the web through a proxy server. If you create a JNLP document for this program and use Java Web Start to launch it, Java Web Start will use the proxy server settings from your default web browser. See Chapters 23–24 for more information on Java Web Start.]

```java
// Fig. 27.3: ReadServerFile.java
// Reading a file by opening a connection through a URL.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.JEditorPane;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

public class ReadServerFile extends JFrame {
    private JTextField enterField; // JTextField to enter site name
    private JEditorPane contentsArea; // to display website

    // set up GUI
    public ReadServerFile() {
    }

    // Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 1 of 2.)
```
super( "Simple Web Browser" );

// create enterField and register its listener
enterField = new JTextField( "Enter file URL here" );
enterField.addActionListener(  
    new ActionListener()  
    {  
        // get document specified by user
        public void actionPerformed( ActionEvent event )  
        {  
            getThePage( event.getActionCommand() );  
        }  
    }  
);  

add( enterField, BorderLayout.NORTH );

contentsArea = new JEditorPane();  
contentsArea.setEditable( false );
contentsArea.addHyperlinkListener(  
    new HyperlinkListener()  
    {  
        // if user clicked hyperlink, go to specified page
        public void hyperlinkUpdate( HyperlinkEvent event )  
        {  
            if ( event.getEventType() == HyperlinkEvent.EventType.ACTIVATED )  
                getThePage( event.getURL().toString() );  
        }  
    }  
);  

add( new JScrollPane( contentsArea ), BorderLayout.CENTER );

// load document
private void getThePage( String location )  
{  
    try // load document and display location
    {  
        contentsArea.setPage( location );  
    }  
    catch ( IOException ioException )  
    {  
        JOptionPane.showMessageDialog( this,  
            "Error retrieving specified URL",  
            "Bad URL",  
            JOptionPane.ERROR_MESSAGE );  
    }  
}  

// end class ReadServerFile

Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 2 of 2.)
27.3 Reading a File on a Web Server

The application class `ReadServerFile` contains `JTextField` `enterField`, in which the user enters the URL of the file to read and `JEditorPane` `contentsArea` to display the file's contents. When the user presses the `Enter` key in `enterField`, the application calls method `actionPerformed` (lines 31–34). Line 33 uses `ActionEvent` method `getActionCommand` to get the String the user input in the `JTextField` and passes the String to utility method `getThePage` (lines 61–74).

Line 65 invokes `JEditorPane` method `setPage` to download the document specified by `location` and display it in the `JEditorPane`. If there's an error downloading the document, method `setPage` throws an `IOException`. Also, if an invalid URL is specified, a `MalformedURLException` (a subclass of `IOException`) occurs. If the document loads successfully, line 66 displays the current location in `enterField`.

Typically, an HTML document contains `hyperlinks` that, when clicked, provide quick access to another document on the web. If a `JEditorPane` contains an HTML document and the user clicks a hyperlink, the `JEditorPane` generates a `HyperlinkEvent` (package `javax.swing.event`) and notifies all registered `HyperlinkListener` (package `javax.swing.event`) of that event. Lines 42–53 register a `HyperlinkListener` to handle `HyperlinkEvents`. When a `HyperlinkEvent` occurs, the program calls method `hyperlinkUpdate` (lines 46–51). Lines 48–49 use `HyperlinkEvent` method `getEventType` to determine the type of the `HyperlinkEvent`. Class `HyperlinkEvent` contains a public nested class called `EventType` that declares three static `EventType` objects, which represent the hyperlink event types. `ACTIVATED` indicates that the user clicked a hyperlink to change web pages, `ENTERED` indicates that the user moved the mouse over a hyperlink and `EXITED` indicates that the user moved the mouse away from a hyperlink. If a hyperlink was `ACTIVATED`, line 50 uses `HyperlinkEvent` method `getURL` to obtain the `URL` represented by the hyperlink. Method `toString` converts the returned `URL` to a `String` that can be passed to utility method `getThePage`.

```java
// Fig. 27.4: ReadServerFileTest.java
// Create and start a ReadServerFile.
import javax.swing.JFrame;

public class ReadServerFileTest {
    public static void main( String[] args ) {
        ReadServerFile application = new ReadServerFile();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    }
}
```

Fig. 27.4 | Test class for ReadServerFile.
27.4 Establishing a Simple Server Using Stream Sockets

The two examples discussed so far use high-level Java networking capabilities to communicate between applications. In the examples, it was not your responsibility to establish the connection between a client and a server. The first program relied on the web browser to communicate with a web server. The second program relied on a JEditorPane to perform the connection. This section begins our discussion of creating your own applications that can communicate with one another.

**Step 1: Create a ServerSocket**

Establishing a simple server in Java requires five steps. Step 1 is to create a ServerSocket object. A call to the ServerSocket constructor, such as

```java
ServerSocket server = new ServerSocket(portNumber, queueLength);
```

registers an available TCP port number and specifies the maximum number of clients that can wait to connect to the server (i.e., the queue length). The port number is used by clients to locate the server application on the server computer. This is often called the handshake point. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients—a process known as binding the server to the port. Each client will ask to connect to the server on this port. Only one application at a time can be bound to a specific port on the server.

**Software Engineering Observation 27.1**

Port numbers can be between 0 and 65,535. Most operating systems reserve port numbers below 1024 for system services (e.g., e-mail and World Wide Web servers). Generally, these ports should not be specified as connection ports in user programs. In fact, some operating systems require special access privileges to bind to port numbers below 1024.

**Step 2: Wait for a Connection**

Programs manage each client connection with a Socket object. In Step 2, the server listens indefinitely (or blocks) for an attempt by a client to connect. To listen for a client connection, the program calls ServerSocket method accept, as in

```java
Socket connection = server.accept();
```

which returns a Socket when a connection with a client is established. The Socket allows the server to interact with the client. The interactions with the client actually occur at a different server port from the handshake point. This allows the port specified in Step 1 to be used again in a multithreaded server to accept another client connection. We demonstrate this concept in Section 27.8.

**Step 3: Get the Socket’s I/O Streams**

Step 3 is to get the OutputStream and InputStream objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to
the client via an OutputStream and receives information from the client via an InputStream. The server invokes method `getOutputStream` on the Socket to get a reference to the Socket's OutputStream and invokes method `getInputStream` on the Socket to get a reference to the Socket's InputStream.

The stream objects can be used to send or receive individual bytes or sequences of bytes with the `OutputStream`'s method `write` and the `InputStream`'s method `read`, respectively. Often it's useful to send or receive values of primitive types (e.g., `int` and `double`) or `Serializable` objects (e.g., `String`s or other serializable types) rather than sending bytes. In this case, we can use the techniques discussed in Chapter 17 to wrap other stream types (e.g., `ObjectOutputStream` and `ObjectInputStream`) around the `OutputStream` and `InputStream` associated with the Socket. For example,

```
ObjectInputStream input = new ObjectInputStream( connection.getInputStream() );
ObjectOutputStream output = new ObjectOutputStream( connection.getOutputStream() );
```

The beauty of establishing these relationships is that whatever the server writes to the `ObjectOutputStream` is sent via the `OutputStream` and is available at the client's `InputStream`, and whatever the client writes to its `OutputStream` (with a corresponding `ObjectOutputStream`) is available via the server's `InputStream`. The transmission of the data over the network is seamless and is handled completely by Java.

**Step 4: Perform the Processing**

Step 4 is the processing phase, in which the server and the client communicate via the `OutputStream` and `InputStream` objects.

**Step 5: Close the Connection**

In Step 5, when the transmission is complete, the server closes the connection by invoking the `close` method on the streams and on the Socket.

---

**Software Engineering Observation 27.2**

With sockets, network I/O appears to Java programs to be similar to sequential file I/O. Sockets hide much of the complexity of network programming.

**Software Engineering Observation 27.3**

A multithreaded server can take the Socket returned by each call to `accept` and create a new thread that manages network I/O across that Socket. Alternatively, a multithreaded server can maintain a pool of threads (a set of already existing threads) ready to manage network I/O across the new Sockets as they're created. These techniques enable multithreaded servers to manage many simultaneous client connections.

**Performance Tip 27.2**

In high-performance systems in which memory is abundant, a multithreaded server can create a pool of threads that can be assigned quickly to handle network I/O for new Sockets as they're created. Thus, when the server receives a connection, it need not incur thread-creation overhead. When the connection is closed, the thread is returned to the pool for reuse.
27.5 Establishing a Simple Client Using Stream Sockets

Establishing a simple client in Java requires four steps.

**Step 1: Create a Socket to Connect to the Server**

In Step 1, we create a Socket to connect to the server. The Socket constructor establishes the connection. For example, the statement

```
Socket connection = new Socket( serverAddress, port );
```

uses the Socket constructor with two arguments—the server’s address (serverAddress) and the port number. If the connection attempt is successful, this statement returns a Socket. A connection attempt that fails throws an instance of a subclass of IOException, so many programs simply catch IOException. An UnknownHostException occurs specifically when the system is unable to resolve the server name specified in the call to the Socket constructor to a corresponding IP address.

**Step 2: Get the Socket’s I/O Streams**

In Step 2, the client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket’s InputStream and OutputStream. As we mentioned in the preceding section, we can use the techniques of Chapter 17 to wrap other stream types around the InputStream and OutputStream associated with the Socket. If the server is sending information in the form of actual types, the client should receive the information in the same format. Thus, if the server sends values with an ObjectOutputStream, the client should read those values with an ObjectInputStream.

**Step 3: Perform the Processing**

Step 3 is the processing phase in which the client and the server communicate via the InputStream and OutputStream objects.

**Step 4: Close the Connection**

In Step 4, the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket. The client must determine when the server is finished sending information so that it can call close to close the Socket connection. For example, the InputStream method read returns the value –1 when it detects end-of-stream (also called EOF—end-of-file). If an ObjectInputStream reads information from the server, an EOFException occurs when the client attempts to read a value from a stream on which end-of-stream is detected.

27.6 Client/Server Interaction with Stream Socket Connections

Figures 27.5 and 27.7 use stream sockets, ObjectInputStream and ObjectOutputStream to demonstrate a simple client/server chat application. The server waits for a client connection attempt. When a client connects to the server, the server application sends the client a String object (recall that Strings are Serializable objects) indicating that the connection was successful. Then the client displays the message. The client and server applications each provide text fields that allow the user to type a message and send it to the other application. When the client or the server sends the String “TERMINATE”, the con-
connection terminates. Then the server waits for the next client to connect. The declaration of class Server appears in Fig. 27.5. The declaration of class Client appears in Fig. 27.7. The screen captures showing the execution between the client and the server are shown in Fig. 27.8.

**Server Class**

Server's constructor (Fig. 27.5, lines 30–55) creates the server's GUI, which contains a JTextField and a JTextArea. Server displays its output in the JTextArea. When the main method (lines 6–11 of Fig. 27.6) executes, it creates a Server object, specifies the window's default close operation and calls method runServer (Fig. 27.5, lines 57–86).

```
// Fig. 27.5: Server.java
// Server portion of a client/server stream-socket connection.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

public class Server extends JFrame
{
    private JTextField enterField; // inputs message from user
    private JTextArea displayArea; // display information to user
    private ObjectOutputStream output; // output stream to client
    private ObjectInputStream input; // input stream from client
    private int counter = 1; // counter of number of connections

    // set up GUI
    public Server()
    {
        super( "Server" );
        enterField = new JTextField(); // create enterField
        enterField.setEditable( false );
        enterField.addActionListener(
        new ActionListener()
        {
            // send message to client
            public void actionPerformed( ActionEvent event )
            {
            }
        });
        // Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 1 of 4.)
```
sendData( event.getActionCommand() );
enterField.setText( "" );
} // end method actionPerformed
} // end anonymous inner class
); // end call to addActionListener
add( enterField, BorderLayout.NORTH );
displayArea = new JTextArea(); // create displayArea
add( new JScrollPane( displayArea ), BorderLayout.CENTER );
setSize( 300, 150 ); // set size of window
setVisible( true ); // show window
} // end Server constructor

// set up and run server
public void runServer()
{
  try // set up server to receive connections; process connections
  {
    server = new ServerSocket( 12345, 100 ); // create ServerSocket
    while ( true )
    {
      try
      {
        waitForConnection(); // wait for a connection
        getStreams(); // get input & output streams
        processConnection(); // process connection
      } // end try
      catch ( EOFException eofException )
      {
        displayMessage( "\nServer terminated connection" );
      } // end catch
    } // end while
    closeConnection(); // close connection
    ++counter;
  } // end finally
} // end try
catch ( IOException ioException )
{
  ioException.printStackTrace();
} // end catch
} // end method runServer

// wait for connection to arrive, then display connection info
private void waitForConnection() throws IOException
{
  displayMessage( "Waiting for connection\n" );
  connection = server.accept(); // allow server to accept connection

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 2 of 4.)
displayMessage( "Connection " + counter + " received from: " +
connection.getInetAddress().getHostName() );
} // end method waitForConnection

// get streams to send and receive data
private void getStreams() throws IOException
{
    // set up output stream for objects
    output = new ObjectOutputStream( connection.getOutputStream() );
    output.flush(); // flush output buffer to send header information
    // set up input stream for objects
    input = new ObjectInputStream( connection.getInputStream() );

displayMessage( "\nGot I/O streams\n" );
} // end method getStreams

// process connection with client
private void processConnection() throws IOException
{
    String message = "Connection successful";
    sendData( message ); // send connection successful message

    // enable enterField so server user can send messages
    setTextFieldEditable( true );

    do // process messages sent from client
    {
        try // read message and display it
        {
            message = ( String ) input.readObject(); // read new message
            displayMessage( "\n" + message ); // display message
        } // end try
        catch ( ClassNotFoundException classNotFoundException )
        {
            displayMessage( "\nUnknown object type received\n" );
        } // end catch
    } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
} // end method processConnection

// close streams and socket
private void closeConnection()
{
    displayMessage( "\nTerminating connection\n" );
    setTextFieldEditable( false ); // disable enterField

    try
    {
        output.close(); // close output stream
        input.close(); // close input stream
        connection.close(); // close socket
    } // end try

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 3 of 4.)
catch ( IOException ioException )
{
    ioException.printStackTrace();
} // end catch

// end method closeConnection

// send message to client
private void sendData( String message )
{
    try // send object to client
    {
        output.writeObject( "SERVER>>> " + message );
        output.flush(); // flush output to client
displayMessage( "\nSERVER>>> " + message );
    } // end try
    catch ( IOException ioException )
    {
        displayArea.append( "\nError writing object" );
    } // end catch
} // end method sendData

// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay )
{
    SwingUtilities.invokeLater( new Runnable()
    {
        public void run() // updates displayArea
        {
            displayArea.append( messageToDisplay ); // append message
        } // end method run
    }); // end call to SwingUtilities.invokeLater
} // end method displayMessage

// manipulates enterField in the event-dispatch thread
private void setTextFieldEditable( final boolean editable )
{
    SwingUtilities.invokeLater( new Runnable()
    {
        public void run() // sets enterField's editability
        {
            enterField.setEditable( editable );
        } // end method run
    }); // end call to SwingUtilities.invokeLater
} // end method setTextFieldEditable

Fig. 27.5  |  Server portion of a client/server stream-socket connection. (Part 4 of 4.)
Method \textit{runServer}

Method \textit{runServer} (Fig. 27.5, lines 57–86) sets up the server to receive a connection and processes one connection at a time. Line 61 creates a \texttt{ServerSocket} called \texttt{server} to wait for connections. The \texttt{ServerSocket} listens for a connection from a client at port 12345. The second argument to the constructor is the number of connections that can wait in a queue to connect to the server (100 in this example). If the queue is full when a client attempts to connect, the server refuses the connection.

Line 67 calls method \textit{waitForConnection} (declared at lines 89–95) to wait for a client connection. After the connection is established, line 68 calls method \textit{getStreams} (declared at lines 98–108) to obtain references to the connection’s streams. Line 69 calls method \textit{processConnection} (declared at lines 111–132) to send the initial connection message to the client and to process all messages received from the client. The \texttt{finally} block (lines 75–79) terminates the client connection by calling method \textit{closeConnection} (lines 135–150), even if an exception occurs. These methods call \textit{displayMessage} (lines 168–179), which uses the event-dispatch thread to display messages in the application’s \texttt{JTextArea}.

\texttt{SwingUtilities} method \textit{invokeLater} receives a \texttt{Runnable} object as its argument and places it into the event-dispatch thread for execution. This ensures that we don’t modify a GUI component from a thread other than the event-dispatch thread, which is important since \textit{Swing GUI components are not thread safe}. We use a similar technique in method \textit{setTextFieldEditable} (lines 182–193), to set the editability of \texttt{enterField}. For more information on interface \texttt{Runnable}, see Chapter 26.

\textbf{Common Programming Error 27.1}

Specifying a port that’s already in use or specifying an invalid port number when creating a \texttt{ServerSocket} results in a \texttt{BindException}.

Method \textit{waitForConnection}

Method \textit{waitForConnection} (lines 89–95) uses \texttt{ServerSocket} method \textit{accept} (line 92) to wait for a connection from a client. When a connection occurs, the resulting \texttt{Socket} is assigned to \texttt{connection}. Method \textit{accept} blocks until a connection is received (i.e., the thread in which \textit{accept} is called stops executing until a client connects). Lines 93–94 output the host name of the computer that made the connection. \texttt{Socket} method \textit{getInetAddress}...
Address returns an InetAddress (package java.net) containing information about the client computer. InetAddress method getHostName returns the host name of the client computer. For example, a special IP address (127.0.0.1) and host name (localhost) are useful for testing networking applications on your local computer (this is also known as the loopback address). If getHostName is called on an InetAddress containing 127.0.0.1, the corresponding host name returned by the method will be localhost.

**Method getStreams**
Method getStreams (lines 98–108) obtains the Socket’s streams and uses them to initialize an ObjectOutputStream (line 101) and an ObjectInputStream (line 105), respectively. Note the call to ObjectOutputStream method flush at line 102. This statement causes the ObjectOutputStream on the server to send a stream header to the corresponding client’s ObjectInputStream. The stream header contains such information as the version of object serialization being used to send objects. This information is required by the ObjectInputStream so that it can prepare to receive those objects correctly.

**Software Engineering Observation 27.4**
When using ObjectOutputStream and ObjectInputStream to send and receive data over a network connection, always create the ObjectOutputStream first and flush the stream so that the client’s ObjectInputStream can prepare to receive the data. This is required for networking applications that communicate using ObjectOutputStream and ObjectInputStream.

**Performance Tip 27.3**
A computer’s I/O components are typically much slower than its memory. Output buffers are used to increase the efficiency of an application by sending larger amounts of data fewer times, reducing the number of times an application accesses the computer’s I/O components.

**Method processConnection**
Line 114 of method processConnection (lines 111–132) calls method sendData to send “SERVER>>> Connection successful” as a String to the client. The loop at lines 119–131 executes until the server receives the message “CLIENT>>> TERMINATE”. Line 123 uses ObjectInputStream method readObject to read a String from the client. Line 124 invokes method displayMessage to append the message to the JTextArea.

**Method closeConnection**
When the transmission is complete, method processConnection returns, and the program calls method closeConnection (lines 135–150) to close the streams associated with the Socket and close the Socket. Then the server waits for the next connection attempt from a client by continuing with line 67 at the beginning of the while loop.

Server receives a connection, processes it, closes it and waits for the next connection. A more likely scenario would be a Server that receives a connection, sets it up to be processed as a separate thread of execution, then immediately waits for new connections. The separate threads that process existing connections can continue to execute while the Server concentrates on new connection requests. This makes the server more efficient, because multiple client requests can be processed concurrently. We demonstrate a multithreaded server in Section 27.8.
Processing User Interactions

When the user of the server application enters a String in the text field and presses the Enter key, the program calls method actionPerformed (lines 39–43), which reads the String from the text field and calls utility method sendData (lines 153–165) to send the String to the client. Method sendData writes the object, flushes the output buffer and appends the same String to the text area in the server window. It’s not necessary to invoke displayMessage to modify the text area here, because method sendData is called from an event handler—thus, sendData executes as part of the event-dispatch thread.

Client Class

Like class Server, class Client’s constructor (Fig. 27.7, lines 29–56) creates the GUI of the application (a JTextField and a JTextArea). Client displays its output in the text area. When method main (lines 7–19 of Fig. 27.8) executes, it creates an instance of class Client, specifies the window’s default close operation and calls method runClient (Fig. 27.7, lines 59–79). In this example, you can execute the client from any computer on the Internet and specify the IP address or host name of the server computer as a command-line argument to the program. For example, the command

```
java Client 192.168.1.15
```

attempts to connect to the Server on the computer with IP address 192.168.1.15.

---

```
1 // Fig. 27.7: Client.java
2 // Client portion of a stream-socket connection between client and server.
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.InetAddress;
8 import java.net.Socket;
9 import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Client extends JFrame
19 {
20   private JTextField enterField; // enters information from user
21   private JTextArea displayArea; // display information to user
22   private ObjectOutputStream output; // output stream to server
23   private ObjectInputStream input; // input stream from server
24   private String message = ""; // message from server
25   private String chatServer; // host server for this application
26   private Socket client; // socket to communicate with server
```
// initialize chatServer and set up GUI
public Client( String host )
{
    super( "Client" );
    chatServer = host; // set server to which this client connects
    enterField = new JTextField(); // create enterField
    enterField.setEditable( false );
    enterField.addActionListener( new ActionListener()
    {
        // send message to server
        public void actionPerformed( ActionEvent event )
        {
            sendData( event.getActionCommand() );
            enterField.setText( "" );
        } // end method actionPerformed
    } ); // end call to addActionListener
    add( enterField, BorderLayout.NORTH );
    displayArea = new JTextArea(); // create displayArea
    add( new JScrollPane( displayArea ), BorderLayout.CENTER );
    setSize( 300, 150 ); // set size of window
    setVisible( true ); // show window
} // end Client constructor

// connect to server and process messages from server
public void runClient()
{
    try // connect to server, get streams, process connection
    {
        connectToServer(); // create a Socket to make connection
        getStreams(); // get the input and output streams
        processConnection(); // process connection
    } // end try
    catch ( EOFException eofException )
    {
        displayMessage( "\nClient terminated connection" );
    } // end catch
    catch ( IOException ioException )
    {
        ioException.printStackTrace();
    } // end catch
    finally
    {
        closeConnection(); // close connection
    } // end finally
} // end method runClient

Fig. 27.7 | Client portion of a stream-socket connection between client and server. (Part 2 of 5.)
// connect to server
private void connectToServer() throws IOException {
    displayMessage("Attempting connection\n");
    // create Socket to make connection to server
    client = new Socket(InetAddress.getByName(chatServer), 12345);
    // display connection information
    displayMessage("Connected to: " + client.getInetAddress().getHostName());
} // end method connectToServer

// get streams to send and receive data
private void getStreams() throws IOException {
    // set up output stream for objects
    output = new ObjectOutputStream(client.getOutputStream());
    output.flush(); // flush output buffer to send header information
    // set up input stream for objects
    input = new ObjectInputStream(client.getInputStream());
    displayMessage("\nGot I/O streams\n");
} // end method getStreams

// process connection with server
private void processConnection() throws IOException {
    // enable enterField so client user can send messages
    setTextFieldEditable(true);
    do // process messages sent from server
    {
        try // read message and display it
        {
            message = (String) input.readObject(); // read new message
            displayMessage("\n" + message); // display message
        } // end try
        catch (ClassNotFoundException classNotFoundException)
        {
            displayMessage("\nUnknown object type received\n");
        } // end catch
    } while (!message.equals("SERVER>>> TERMINATE"));
} // end method processConnection

// close streams and socket
private void closeConnection()
{
    displayMessage("\nClosing connection\n");
    setTextFieldEditable(false); // disable enterField
}

Fig. 27.7 | Client portion of a stream-socket connection between client and server. (Part 3 of 5.)
```java
try {
    output.close(); // close output stream
    input.close(); // close input stream
    client.close(); // close socket
} // end try

catch ( IOException ioException ) {
    ioException.printStackTrace();
} // end catch

// end method closeConnection

// send message to server
private void sendData( String message ) {
    try {
        output.writeObject( "CLIENT>>> " + message );
        output.flush(); // flush data to output
        displayMessage( "\nCLIENT>>> " + message );
    } // end try
    catch ( IOException ioException ) {
        displayArea.append( "\nError writing object" );
    } // end catch
} // end method sendData

// manipulates displayArea in the event-dispatch thread
private void displayMessage( final String messageToDisplay ) {
    SwingUtilities.invokeLater( new Runnable() {
        public void run() // updates displayArea
        {
            displayArea.append( messageToDisplay );
        } // end anonymous inner class
    } ); // end call to SwingUtilities.invokeLater

// manipulates enterField in the event-dispatch thread
private void setTextFieldEditable( final boolean editable ) {
    SwingUtilities.invokeLater( new Runnable() {
        public void run() // sets enterField's editability
        {
            enterField.setEditable( editable );
        } // end anonymous inner class
    } ); // end call to SwingUtilities.invokeLater
```

Fig. 27.7  |  Client portion of a stream-socket connection between client and server. (Part 4 of 5.)
Method runClient

Client method runClient (Fig. 27.7, lines 59–79) sets up the connection to the server, processes messages received from the server and closes the connection when communication is complete. Line 63 calls method connectToServer (declared at lines 82–92) to perform the connection. After connecting, line 64 calls method getStreams (declared at lines 95–105) to obtain references to the Socket’s stream objects. Then line 65 calls method processConnection (declared at lines 108–126) to receive and display messages sent from the server. The finally block (lines 75–78) calls closeConnection (lines 129–144) to close the streams and the Socket even if an exception occurred. Method displayMessage (lines 162–173) is called from these methods to use the event-dispatch thread to display messages in the application’s text area.
Method connectToServer
Method connectToServer (lines 82–92) creates a Socket called client (line 87) to establish a connection. The arguments to the Socket constructor are the IP address of the server computer and the port number (12345) where the server application is awaiting client connections. In the first argument, InetAddress static method getByName returns an InetAddress object containing the IP address specified as a command-line argument to the application (or 127.0.0.1 if none was specified). Method getByName can receive a String containing either the actual IP address or the host name of the server. The first argument also could have been written other ways. For the localhost address 127.0.0.1, the first argument could be specified with either of the following expressions:

```
InetAddress.getByName( "localhost" )
InetAddress.getLocalHost()
```

Other versions of the Socket constructor receive the IP address or host name as a String. The first argument could have been specified as the IP address "127.0.0.1" or the host name "localhost". We chose to demonstrate the client/server relationship by connecting between applications on the same computer (localhost). Normally, this first argument would be the IP address of another computer. The InetAddress object for another computer can be obtained by specifying the computer’s IP address or host name as the argument to InetAddress method getByName. The Socket constructor’s second argument is the server port number. This must match the port number at which the server is waiting for connections (called the handshake point). Once the connection is made, lines 90–91 display a message in the text area indicating the name of the server computer to which the client has connected.

The Client uses an ObjectOutputStream to send data to the server and an ObjectInputStream to receive data from the server. Method getStreams (lines 95–105) creates the ObjectOutputStream and ObjectInputStream objects that use the streams associated with the client socket.

Methods processConnection and closeConnection
Method processConnection (lines 108–126) contains a loop that executes until the client receives the message "SERVER>>> TERMINATE". Line 117 reads a String object from the server. Line 118 invokes displayMessage to append the message to the text area. When the transmission is complete, method closeConnection (lines 129–144) closes the streams and the Socket.

Processing User Interactions
When the client application user enters a String in the text field and presses Enter, the program calls method actionPerformed (lines 41–45) to read the String, then invokes utility method sendData (147–159) to send the String to the server. Method sendData writes the object, flushes the output buffer and appends the same String to the client window’s JTextArea. Once again, it’s not necessary to invoke utility method displayMessage to modify the text area here, because method sendData is called from an event handler.

27.7 Datagrams: Connectionless Client/Server Interaction
We’ve been discussing connection-oriented, streams-based transmission. Now we consider connectionless transmission with datagrams.
Connection-oriented transmission is like the telephone system in which you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for your phone call, even when you're not talking.

Connectionless transmission with datagrams is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate pieces that you place in sequentially numbered envelopes. All of the letters are then mailed at once. The letters could arrive in order, out of order or not at all (the last case is rare). The person at the receiving end reassembles the pieces into sequential order before attempting to make sense of the message.

If your message is small enough to fit in one envelope, you need not worry about the “out-of-sequence” problem, but it's still possible that your message might not arrive. One advantage of datagrams over postal mail is that duplicates of datagrams can arrive at the receiving computer.

Figures 27.9–27.12 use datagrams to send packets of information via the User Datagram Protocol (UDP) between a client application and a server application. In the Client application (Fig. 27.11), the user types a message into a text field and presses Enter. The program converts the message into a byte array and places it in a datagram packet that’s sent to the server. The Server (Figs. 27.9–27.10) receives the packet and displays the information in it, then echoes the packet back to the client. Upon receiving the packet, the client displays the information it contains.

Server Class
Class Server (Fig. 27.9) declares two **DatagramPackets** that the server uses to send and receive information and one **DatagramSocket** that sends and receives the packets. The constructor (lines 19–37), which is called from main (Fig. 27.10, lines 7–12), creates the GUI in which the packets of information will be displayed. Line 30 creates the DatagramSocket in a try block. Line 30 in Fig. 27.9 uses the DatagramSocket constructor that takes an integer port-number argument (5000 in this example) to bind the server to a port where it can receive packets from clients. Clients sending packets to this Server specify the same port number in the packets they send. A **SocketException** is thrown if the DatagramSocket constructor fails to bind the DatagramSocket to the specified port.

**Common Programming Error 27.2**
Specifying a port that’s already in use or specifying an invalid port number when creating a DatagramSocket results in a SocketException.

```java
// Fig. 27.9: Server.java
// Server side of connectionless client/server computing with datagrams.
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.SocketException;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams. (Part 1 of 3.)
import javax.swing.SwingUtilities;

public class Server extends JFrame {
    private JTextArea displayArea; // displays packets received
    private DatagramSocket socket; // socket to connect to client

    // set up GUI and DatagramSocket
    public Server() {
        super( "Server" );
        displayArea = new JTextArea(); // create displayArea
        add( new JScrollPane( displayArea ), BorderLayout.CENTER );
        setSize( 400, 300 ); // set size of window
        setVisible( true ); // show window

        try // create DatagramSocket for sending and receiving packets
            { // end try
                socket = new DatagramSocket( 5000 );
            } // end try catch ( SocketException socketException )
            { // end catch
                socketException.printStackTrace();
                System.exit( 1 );
            } // end catch
    } // end Server constructor

    // wait for packets to arrive, display data and echo packet to client
    public void waitForPackets() {
        while ( true ) {
            try // receive packet, display contents, return copy to client
                { // end try
                    byte[] data = new byte[ 100 ]; // set up packet
                    DatagramPacket receivePacket =
                        new DatagramPacket( data, data.length );
                    socket.receive( receivePacket ); // wait to receive packet

                    displayMessage( "\nPacket received:" +
                        "\nFrom host: " + receivePacket.getAddress() +
                        "\nHost port: " + receivePacket.getPort() +
                        "\nLength: " + receivePacket.getLength() +
                        "\nContaining:\n\t" + new String( receivePacket.getData(), 0, receivePacket.getLength() ) );

                    sendPacketToClient( receivePacket ); // send packet to client
                } // end try catch ( IOException ioException )
    }

    // display information from received packet
    private String displayMessage( String output ) {
        return output;
    }

    // send packet to client
    private void sendPacketToClient( DatagramPacket packet ) {
    }

    // Fig. 27.9 | Server side of connectionless client/server computing with datagrams. (Part 2 of 3.)
Datagrams: Connectionless Client/Server Interaction

```java
64    displayMessage( ioException + "\n" );
65    ioException.printStackTrace();
66 } // end catch
67 } // end while
68 } // end method waitForPackets
69 // echo packet to client
70 private void sendPacketToClient( DatagramPacket receivePacket )
71 throws IOException
72 {
73    displayMessage( "\n\nEcho data to client..." );
74
75    // create packet to send
76    DatagramPacket sendPacket = new DatagramPacket(
77        receivePacket.getData(), receivePacket.getLength(),
78        receivePacket.getAddress(), receivePacket.getPort() );
79
80    socket.send( sendPacket ); // send packet to client
81    displayMessage( "Packet sent\n" );
82 } // end method sendPacketToClient
83
84 // manipulates displayArea in the event-dispatch thread
85 private void displayMessage( final String messageToDisplay )
86 {
87    SwingUtilities.invokeLater(
88        new Runnable()
89        {
90            public void run() // updates displayArea
91            {
92                displayArea.append( messageToDisplay ); // display message
93            } // end method run
94        } // end anonymous inner class
95    ); // end call to SwingUtilities.invokeLater
96 } // end method displayMessage
97 } // end class Server
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams. (Part 3 of 3.)

```java
// Fig. 27.10: ServerTest.java
// Class that tests the Server.
import javax.swing.JFrame;

public class ServerTest
{
    public static void main( String[] args )
    {
        Server application = new Server(); // create server
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        application.waitForPackets(); // run server application
    } // end main
} // end class ServerTest
```

Fig. 27.10 | Class that tests the Server. (Part 1 of 2.)
Method waitForPackets
Server method waitForPackets (Fig. 27.9, lines 40–68) uses an infinite loop to wait for packets to arrive at the Server. Lines 47–48 create a DatagramPacket in which a received packet of information can be stored. The DatagramPacket constructor for this purpose receives two arguments—a byte array in which the data will be stored and the length of the array. Line 50 uses DatagramSocket method receive to wait for a packet to arrive at the Server. Method receive blocks until a packet arrives, then stores the packet in its DatagramPacket argument. The method throws an IOException if an error occurs while receiving a packet.

Method displayMessage
When a packet arrives, lines 53–58 call method displayMessage (declared at lines 86–97) to append the packet’s contents to the text area. DatagramPacket method getAddress (line 54) returns an InetAddress object containing the IP address of the computer from which the packet was sent. Method getPort (line 55) returns an integer specifying the port number through which the client computer sent the packet. Method getLength (line 56) returns an integer representing the number of bytes of data received. Method getData (line 57) returns a byte array containing the data. Lines 57–58 initialize a String object using a three-argument constructor that takes a byte array, the offset and the length. This String is then appended to the text to display.

Method sendPacketToClient
After displaying a packet, line 60 calls method sendPacketToClient (declared at lines 71–83) to create a new packet and send it to the client. Lines 77–79 create a DatagramPacket and pass four arguments to its constructor. The first argument specifies the byte array to send. The second argument specifies the number of bytes to send. The third argument specifies the client computer’s IP address, to which the packet will be sent. The fourth argument specifies the port where the client is waiting to receive packets. Line 81 sends the packet over the network. Method send of DatagramSocket throws an IOException if an error occurs while sending a packet.

Client Class
The Client (Figs. 27.11–27.12) works similarly to class Server, except that the Client sends packets only when the user types a message in a text field and presses the Enter key.
When this occurs, the program calls method `actionPerformed` (Fig. 27.11, lines 32–57), which converts the `String` the user entered into a `byte` array (line 41). Lines 44–45 create a `DatagramPacket` and initialize it with the `byte` array, the length of the `String` that was entered by the user, the IP address to which the packet is to be sent (`InetAddress.getLocalHost()` in this example) and the port number at which the `Server` is waiting for packets (5000 in this example). Line 47 sends the packet. The client in this example must know that the server is receiving packets at port 5000—otherwise, the server will not receive the packets.

The `DatagramSocket` constructor call (Fig. 27.11, line 71) in this application does not specify any arguments. This no-argument constructor allows the computer to select the next available port number for the `DatagramSocket`. The client does not need a specific port number, because the server receives the client’s port number as part of each `DatagramPacket` sent by the client. Thus, the server can send packets back to the same computer and port number from which it receives a packet of information.

```java
public class Client extends JFrame {

    private JTextField enterField; // for entering messages
    private JTextArea displayArea; // for displaying messages
    private DatagramSocket socket; // socket to connect to server

    public Client() {
        super("Client");

        enterField = new JTextField("Type message here");
        enterField.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent event ) {
                try // create and send packet
                {
```
// get message from textfield
String message = event.getActionCommand();
displayArea.append("Sending packet containing: " +
        message + \"\n\" );

byte[] data = message.getBytes(); // convert to bytes

// create sendPacket
DatagramPacket sendPacket = new DatagramPacket( data,
data.length, InetAddress.getLocalHost(), 5000 );
socket.send( sendPacket ); // send packet
displayArea.append("Packet sent\n");
displayArea.setCaretPosition(
        displayArea.getText().length() );
} // end try
catch ( IOException ioException )
{
    displayMessage( ioException + \"\n\" );
    ioException.printStackTrace();
} // end catch
} // end actionPerformed
} // end inner class
); // end call to add ActionListener
add( enterField, BorderLayout.NORTH );
displayArea = new JTextArea();
add( new JScrollPane( displayArea ), BorderLayout.CENTER );
setSize( 400, 300 ); // set window size
setVisible( true ); // show window
try // create DatagramSocket for sending and receiving packets
{
    socket = new DatagramSocket();
} // end try
catch ( SocketException socketException )
{
    socketException.printStackTrace();
    System.exit( 1 );
} // end catch
} // end Client constructor
// wait for packets to arrive from Server, display packet contents
public void waitForPackets()
{
    while ( true )
    {
        try // receive packet and display contents
        {
            byte[] data = new byte[ 100 ]; // set up packet
            Fig. 27.11  |  Client side of connectionless client/server computing with datagrams. (Part 2 of 3.)
DatagramPacket receivePacket = new DatagramPacket(data, data.length);
socket.receive(receivePacket); // wait for packet

// display packet contents
displayMessage("Packet received:" +
"\nFrom host:" + receivePacket.getAddress() +
"\nHost port:" + receivePacket.getPort() +
"\nLength:" + receivePacket.getLength() +
"\nContaining:\n\t" + new String(receivePacket.getData(),
0, receivePacket.getLength()));
}
} catch (IOException exception)
{

displayMessage(exception + "\n");
exception.printStackTrace();
}
// end method waitForPackets

// manipulates displayArea in the event-dispatch thread
private void displayMessage(final String messageToDisplay)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()
        // updates displayArea
        {
            displayArea.append(messageToDisplay);
        }
    });
} // end method displayMessage

Fig. 27.11 | Client side of connectionless client/server computing with datagrams. (Part 3 of 3.)

// Fig. 27.12: ClientTest.java
// Tests the Client class.
import javax.swing.JFrame;

public class ClientTest
{
    public static void main(String[] args)
    {
        Client application = new Client(); // create client
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        application.waitForPackets(); // run client application
    }
} // end class ClientTest

Fig. 27.12 | Class that tests the Client. (Part 1 of 2.)
Method `waitForPackets`

Client method `waitForPackets` (lines 81–107) uses an infinite loop to wait for packets from the server. Line 91 blocks until a packet arrives. This does not prevent the user from sending a packet, because the GUI events are handled in the event-dispatch thread. It only prevents the `while` loop from continuing until a packet arrives at the `Client`. When a packet arrives, line 91 stores it in `receivePacket`, and lines 94–99 call method `displayMessage` (declared at lines 110–121) to display the packet's contents in the text area.

### 27.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

This section presents the popular game Tic-Tac-Toe implemented by using client/server techniques with stream sockets. The program consists of a `TicTacToeServer` application (Figs. 27.13–27.14) that allows two `TicTacToeClient` applications (Figs. 27.15–27.16) to connect to the server and play Tic-Tac-Toe. Sample outputs are shown in Fig. 27.17.

#### `TicTacToeServer` Class

As the `TicTacToeServer` receives each client connection, it creates an instance of inner-class `Player` (Fig. 27.13, lines 182–304) to process the client in a separate thread. These threads enable the clients to play the game independently. The first client to connect to the server is player X and the second is player O. Player X makes the first move. The server maintains the information about the board so it can determine if a player's move is valid.

```java
// Fig. 27.13: TicTacToeServer.java
// Server side of client/server Tic-Tac-Toe program.
import java.awt.BorderLayout;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;
import java.util.Formatter;
import java.util.Scanner;
import java.util.concurrent.ExecutorService;
import java.io.IOException;
import java.util.concurrent.Executors;
```
27.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class TicTacToeServer extends JFrame
{
    private String[] board = new String[9]; // tic-tac-toe board
    private JTextArea outputArea; // for outputting moves
    private Player[] players; // array of Players
    private ServerSocket server; // server socket to connect with clients
    private int currentPlayer; // keeps track of player with current move
    private final static int PLAYER_X = 0; // constant for first player
    private final static int PLAYER_O = 1; // constant for second player
    private final static String[] MARKS = { "X", "O" }; // array of marks
    private ExecutorService runGame; // will run players
    private Lock gameLock; // to lock game for synchronization
    private Condition otherPlayerConnected; // to wait for other player
    private Condition otherPlayerTurn; // to wait for other player's turn

    // set up tic-tac-toe server and GUI that displays messages
    public TicTacToeServer()
    {
        super( "Tic-Tac-Toe Server" ); // set title of window

        // create ExecutorService with a thread for each player
        runGame = Executors.newFixedThreadPool(2);
        gameLock = new ReentrantLock(); // create lock for game

        // condition variable for both players being connected
        otherPlayerConnected = gameLock.newCondition();

        // condition variable for the other player's turn
        otherPlayerTurn = gameLock.newCondition();

        for ( int i = 0; i < 9; i++ )
            board[ i ] = new String( "" ); // create tic-tac-toe board

        players = new Player[2]; // create array of players
        currentPlayer = PLAYER_X; // set current player to first player

        try
        {
            server = new ServerSocket( 12345, 2 ); // set up ServerSocket
        } // end try
        catch ( IOException ioException )
        {
            ioException.printStackTrace();
            System.exit( 1 );
        } // end catch

        outputArea = new JTextArea(); // create JTextArea for output

    }
}

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 2 of 7.)
add(outputArea, BorderLayout.CENTER);
outputArea.setText("Server awaiting connections\n");

setSize(300, 300); // set size of window
isVisible(true); // show window
}

// wait for two connections so game can be played
public void execute()
{
    // wait for each client to connect
    for (int i = 0; i < players.length; i++)
    {
        try // wait for connection, create Player, start runnable
        {
            players[i] = new Player(server.accept(), i);
            runGame.execute(players[i]); // execute player runnable
        } // end try
        catch (IOException ioException)
        {
            ioException.printStackTrace();
            System.exit(1);
        } // end catch
    } // end for

gameLock.lock(); // lock game to signal player X's thread

try
{
    players[PLAYER_X].setSuspended(false); // resume player X
    otherPlayerConnected.signal(); // wake up player X's thread
} // end try
finally
{
    gameLock.unlock(); // unlock game after signalling player X
} // end finally
} // end method execute

// display message in outputArea
private void displayMessage(final String messageToDisplay)
{
    // display message from event-dispatch thread of execution
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run() // updates outputArea
        {
            outputArea.append(messageToDisplay); // add message
        } // end method run
    } // end inner class
); // end call to SwingUtilities.invokeLater

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 3 of 7.)
// determine if move is valid
public boolean validateAndMove( int location, int player )
{
    // while not current player, must wait for turn
    while ( player != currentPlayer )
    {
        gameLock.lock(); // lock game to wait for other player to go
        try
        {
            otherPlayerTurn.await(); // wait for player's turn
        } // end try
        catch ( InterruptedException exception )
        {
            exception.printStackTrace();
        } // end catch
        finally
        {
            gameLock.unlock(); // unlock game after waiting
        } // end finally
    } // end while
    // if location not occupied, make move
    if ( !isOccupied( location ) )
    {
        board[ location ] = MARKS[ currentPlayer ]; // set move on board
        currentPlayer = ( currentPlayer + 1 ) % 2; // change player
        // let new current player know that move occurred
        players[ currentPlayer ].otherPlayerMoved( location );
        gameLock.lock(); // lock game to signal other player to go
        try
        {
            otherPlayerTurn.signal(); // signal other player to continue
        } // end try
        finally
        {
            gameLock.unlock(); // unlock game after signaling
        } // end finally
        return true; // notify player that move was valid
    } // end if
    else // move was not valid
        return false; // notify player that move was invalid
} // end method validateAndMove

// determine whether location is occupied
public boolean isOccupied( int location )
{
if ( board[ location ].equals( MARKS[ PLAYER_X ] ) ||
    board[ location ].equals( MARKS[ PLAYER_O ] ) )
    return true; // location is occupied
else
    return false; // location is not occupied
} // end method isOccupied

// place code in this method to determine whether game over
public boolean isGameOver()
{
    return false; // this is left as an exercise
} // end method isGameOver

// private inner class Player manages each Player as a runnable
private class Player implements Runnable
{
    private Socket connection; // connection to client
    private Scanner input; // input from client
    private Formatter output; // output to client
    private int playerNumber; // tracks which player this is
    private String mark; // mark for this player
    private boolean suspended = true; // whether thread is suspended

    // set up Player thread
    public Player( Socket socket, int number )
    {
        playerNumber = number; // store this player's number
        mark = MARKS[ playerNumber ]; // specify player's mark
        connection = socket; // store socket for client

        try // obtain streams from Socket
        {
            input = new Scanner( connection.getInputStream() );
            output = new Formatter( connection.getOutputStream() );
        } // end try
        catch ( IOException ioException )
        {
            ioException.printStackTrace();
            System.exit( 1 );
        } // end catch
    } // end Player constructor

    // send message that other player moved
    public void otherPlayerMoved( int location )
    {
        output.format( "Opponent moved\n" );
        output.format( "%d\n", location ); // send location of move
        output.flush(); // flush output
    } // end method otherPlayerMoved

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 5 of 7.)
public void run()
{
    // send client its mark (X or O), process messages from client
    try {
        displayMessage("Player " + mark + " connected\n");
        output.format("%s\n", mark); // send player's mark
        output.flush(); // flush output

        // if player X, wait for another player to arrive
        if (playerNumber == PLAYER_X)
        {
            output.format("%s\n", "Player X connected",
                "Waiting for another player\n");
            output.flush(); // flush output

            gameLock.lock(); // lock game to wait for second player
            try {
                otherPlayerConnected.await(); // wait for player O
            } catch (InterruptedException exception)
            {
                exception.printStackTrace();
            } finally {
                gameLock.unlock(); // unlock game after second player
            }
        } else {
            output.format("Player O connected, please wait\n");
            output.flush(); // flush output
        }

        // while game not over
        while (!isGameOver())
        {
            int location = 0; // initialize move location
            if (input.hasNext())
            {
                location = input.nextInt(); // get move location
            }
        }
    }
}
// check for valid move
if ( validateAndMove( location, playerNumber ) )
{
    displayMessage( \nlocation: " + location );
    output.format( "Valid move.\n" ); // notify client
    output.flush(); // flush output
} // end if
else // move was invalid
{
    output.format( "Invalid move, try again\n" );
    output.flush(); // flush output
} // end else
} // end while
finally
{
    try
    {
        connection.close(); // close connection to client
    } // end try
    catch ( IOException ioException )
    {
        ioException.printStackTrace();
        System.exit( 1 );
    } // end catch
} // end finally
} // end method run

// set whether or not thread is suspended
public void setSuspended( boolean status )
{
    suspended = status; // set value of suspended
} // end method setSuspended
} // end class Player
} // end class TicTacToeServer

---

// Fig. 27.14: TicTacToeServerTest.java
// Class that tests Tic-Tac-Toe server.
import javax.swing.JFrame;

public class TicTacToeServerTest
{
    public static void main( String[] args )
    {
        TicTacToeServer application = new TicTacToeServer();
        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        application.execute();
    } // end main
} // end class TicTacToeServerTest

---

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 7 of 7.)

Fig. 27.14 | Class that tests Tic-Tac-Toe server. (Part 1 of 2.)
We begin with a discussion of the server side of the Tic-Tac-Toe game. When the TicTacToeServer application executes, the main method (lines 7–12 of Fig. 27.14) creates a TicTacToeServer object called application. The constructor (Fig. 27.13, lines 34–69) attempts to set up a ServerSocket. If successful, the program displays the server window, then main invokes the TicTacToeServer method execute (lines 72–100). Method execute loops twice, blocking at line 79 each time while waiting for a client connection. When a client connects, line 79 creates a new Player object to manage the connection as a separate thread, and line 80 executes the Player in the runGame thread pool.

When the TicTacToeServer creates a Player, the Player constructor (lines 192–208) receives the Socket object representing the connection to the client and gets the associated input and output streams. Line 201 creates a Formatter (see Chapter 17) by wrapping it around the output stream of the socket. The Player’s run method (lines 219–297) controls the information that’s sent to and received from the client. First, it passes to the client the character that the client will place on the board when a move is made (line 225). Line 226 calls Formatter method flush to force this output to the client. Line 241 suspends player X’s thread as it starts executing, because player X can move only after player O connects.

When player O connects, the game can be played, and the run method begins executing its while statement (lines 264–283). Each iteration of this loop reads an integer (line 269) representing the location where the client wants to place a mark (blocking to wait for input, if necessary), and line 272 invokes the TicTacToeServer method validateAndMove (declared at lines 118–163) to check the move. If the move is valid, line 275 sends a message to the client to this effect. If not, line 280 sends a message indicating that the move was invalid. The program maintains board locations as numbers from 0 to 8 (0 through 2 for the first row, 3 through 5 for the second row and 6 through 8 for the third row).

Method validateAndMove (lines 118–163 in class TicTacToeServer) allows only one player at a time to move, thereby preventing them from modifying the state information of the game simultaneously. If the Player attempting to validate a move is not the current player (i.e., the one allowed to make a move), it’s placed in a wait state until its turn to move. If the position for the move being validated is already occupied on the board,
validMove returns false. Otherwise, the server places a mark for the player in its local representation of the board (line 142), notifies the other Player object (line 146) that a move has been made (so that the client can be sent a message), invokes method signal (line 152) so that the waiting Player (if there is one) can validate a move and returns true (line 159) to indicate that the move is valid.

**TicTacToeClient Class**

Each TicTacToeClient application (Figs. 27.15–27.16; sample outputs in Fig. 27.17) maintains its own GUI version of the Tic-Tac-Toe board on which it displays the state of the game. The clients can place a mark only in an empty square. Inner class Square (Fig. 27.15, lines 205–261) implements each of the nine squares on the board. When a TicTacToeClient begins execution, it creates a JTextArea in which messages from the server and a representation of the board using nine Square objects are displayed. The startClient method (lines 80–100) opens a connection to the server and gets the associated input and output streams from the Socket object. Lines 85–86 make a connection to the server. Class TicTacToeClient implements interface Runnable so that a separate thread can read messages from the server. This approach enables the user to interact with the board (in the event-dispatch thread) while waiting for messages from the server. After establishing the connection to the server, line 99 executes the client with the worker ExecutorService. The run method (lines 103–126) controls the separate thread of execution. The method first reads the mark character (X or O) from the server (line 105), then loops continuously (lines 121–125) and reads messages from the server (line 124). Each message is passed to the processMessage method (lines 129–156) for processing.

```java
public class TicTacToeClient extends JFrame implements Runnable {
    // Fig. 27.15: TicTacToeClient.java
    // Client side of client/server Tic-Tac-Toe program.

    // Client side of client/server Tic-Tac-Toe program.
```
private JTextField idField; // textfield to display player's mark
private JTextArea displayArea; // JTextArea to display output
private JPanel boardPanel; // panel for tic-tac-toe board
private JPanel panel2; // panel to hold board
private Square[][] board; // tic-tac-toe board
private Square currentSquare; // current square
private Socket connection; // connection to server
private Scanner input; // input from server
private Formatter output; // output to server
private String ticTacToeHost; // host name for server
private String myMark; // this client's mark
private boolean myTurn; // determines which client's turn it is
private final String X_MARK = "X"; // mark for first client
private final String O_MARK = "O"; // mark for second client

// set up user-interface and board
public TicTacToeClient(String host)
{
    ticTacToeHost = host; // set name of server
displayArea = new JTextArea(4, 30); // set up JTextArea
displayArea.setEditable(false);
add(new JScrollPane(displayArea), BorderLayout.SOUTH);
boardPanel = new JPanel(); // set up panel for squares in board
boardPanel.setLayout(new GridLayout(3, 3, 0, 0));
board = new Square[3][3]; // create board

// loop over the rows in the board
for (int row = 0; row < board.length; row++)
{
    // loop over the columns in the board
    for (int column = 0; column < board[row].length; column++)
    {
        // create square
        board[row][column] = new Square(' ', row * 3 + column);
        boardPanel.add(board[row][column]); // add square
    } // end inner for
} // end outer for

idField = new JTextField(); // set up textfield
idField.setEditable(false);
add(idField, BorderLayout.NORTH);

panel2 = new JPanel(); // set up panel to contain boardPanel
panel2.add(boardPanel, BorderLayout.CENTER); // add board panel
add(panel2, BorderLayout.CENTER); // add container panel

setSize(300, 225); // set size of window
setVisible(true); // show window

startClient();
} // end TicTacToeClient constructor

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 2 of 6.)
```java
// start the client thread
public void startClient()
{
    try // connect to server and get streams
    {
        // make connection to server
        connection = new Socket(
            InetAddress.getByName( ticTacToeHost ), 12345);

        // get streams for input and output
        input = new Scanner( connection.getInputStream() );
        output = new Formatter( connection.getOutputStream() );
    } // end try
    catch ( IOException ioException )
    {
        ioException.printStackTrace();
    } // end catch

    // create and start worker thread for this client
    ExecutorService worker = Executors.newFixedThreadPool( 1);
    worker.execute( this ); // execute client
} // end method startClient

// control thread that allows continuous update of displayArea
public void run()
{
    SwingUtilities.invokeLater(
        new Runnable()
        {
            public void run()
            {
                // display player's mark
                idField.setText( "You are player \"" + myMark + "\"" );
            } // end method run
        } // end anonymous inner class
    ); // end call to SwingUtilities.invokeLater

    myTurn = ( myMark.equals( X_MARK ) ); // determine if client's turn

    // receive messages sent to client and output them
    while ( true )
    {
        if ( input.hasNextLine() )
            processMessage( input.nextLine() );
    } // end while
} // end method run

private void processMessage( String message )
{
    // process messages received by client
}
```

---

**Fig. 27.15** | Client side of client/server Tic-Tac-Toe program. (Part 3 of 6.)
27.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

```java
131    // valid move occurred
132    if ( message.equals( "Valid move." ) )
133        {
134          displayMessage( "Valid move, please wait.\n" );
135          setMark( currentSquare, myMark ); // set mark in square
136        } // end if
137    else if ( message.equals( "Invalid move, try again" ) )
138        {
139          displayMessage( message + "\n" ); // display invalid move
140          myTurn = true; // still this client's turn
141        } // end else if
142    else if ( message.equals( "Opponent moved" ) )
143        {
144          int location = input.nextInt(); // get move location
145          input.nextLine(); // skip newline after int location
146          int row = location / 3; // calculate row
147          int column = location % 3; // calculate column
148          setMark( board[ row ][ column ],
149                    ( myMark.equals( X_MARK ) ? O_MARK : X_MARK ) ); // mark move
150          displayMessage( "Opponent moved. Your turn.\n" );
151          myTurn = true; // now this client's turn
152        } // end else if
153    else
t154      displayMessage( message + "\n" ); // display the message
155  } // end method processMessage
156
157  // manipulate displayArea in event-dispatch thread
158  private void displayMessage( final String messageToDisplay )
159  {
160      SwingUtilities.invokeLater( new Runnable()
161      {
162        public void run()
163        {
164          displayArea.append( messageToDisplay ); // updates output
165        } // end inner class
166      } ); // end method displayMessage
167  }
168
169  // utility method to set mark on board in event-dispatch thread
170  private void setMark( final Square squareToMark, final String mark )
171  {
172      SwingUtilities.invokeLater( new Runnable()
173      {
174        public void run()
175        {
176          squareToMark.setMark( mark ); // set mark in square
177        } // end method run
178      } ); // end method call to SwingUtilities.invokeLater
179  }
180
Fig. 27.15  |  Client side of client/server Tic-Tac-Toe program. (Part 4 of 6.)
send message to server indicating clicked square
public void sendClickedSquare( int location )
{
    if ( myTurn )
    {
        output.format( "%d\n", location ); // send location to server
        output.flush();
        myTurn = false; // not my turn any more
    }
} // end method sendClickedSquare

// set current Square
public void setCurrentSquare( Square square )
{
    currentSquare = square; // set current square to argument
} // end method setCurrentSquare

// private inner class for the squares on the board
private class Square extends JPanel
{
    private String mark; // mark to be drawn in this square
    private int location; // location of square

    public Square( String squareMark, int squareLocation )
    {
        mark = squareMark; // set mark for this square
        location = squareLocation; // set location of this square

        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased( MouseEvent e )
                {
                    setCurrentSquare( Square.this ); // set current square
                    // send location of this square
                    sendClickedSquare( getSquareLocation() );
                } // end method mouseReleased
            } // end anonymous inner class
        ); // end call to addMouseListener
    } // end Square constructor

    // return preferred size of Square
    public Dimension getPreferredSize()
    {
        return new Dimension( 30, 30 ); // return preferred size
    } // end method getPreferredSize

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 5 of 6.)
27.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server

```java
234     // return minimum size of Square
235     public Dimension getMinimumSize()
236     {
237         return getPreferredSize(); // return preferred size
238     } // end method getMinimumSize
239
240     // set mark for Square
241     public void setMark( String newMark )
242     {
243         mark = newMark; // set mark of square
244         repaint(); // repaint square
245     } // end method setMark
246
247     // return Square location
248     public int getSquareLocation()
249     {
250         return location; // return location of square
251     } // end method getSquareLocation
252
253     // draw Square
254     public void paintComponent( Graphics g )
255     {
256         super.paintComponent( g );
257
258         g.drawRect( 0, 0, 29, 29 ); // draw square
259         g.drawString( mark, 11, 20 ); // draw mark
260     } // end method paintComponent
261 } // end inner-class Square
262
263 } // end class TicTacToeClient

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 6 of 6.)

```

1 // Fig. 27.16: TicTacToeClientTest.java
2 // Test class for Tic-Tac-Toe client.
3 import javax.swing.JFrame;
4
5 public class TicTacToeClientTest
6 {
7     public static void main( String[] args )
8     {
9         TicTacToeClient application; // declare client application
10
11         // if no command line args
12         if ( args.length == 0 )
13             application = new TicTacToeClient( "127.0.0.1" ); // localhost
14         else
15             application = new TicTacToeClient( args[ 0 ] ); // use args
16
17         application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
18     } // end main
19 } // end class TicTacToeClientTest

Fig. 27.16 | Test class for Tic-Tac-Toe client.
If the message received is "Valid move.", lines 134–135 display the message "Valid move, please wait." and call method setMark (lines 173–184) to set the client's mark in the current square (the one in which the user clicked), using SwingUtilities method invokeLater to ensure that the GUI updates occur in the event-dispatch thread. If the message received is "Invalid move, try again.", line 139 displays the message so that the user can click a different square. If the message received is "Opponent moved.", line 144 reads an integer from the server indicating where the opponent moved, and lines 149–150 place a mark in that square of the board (again using SwingUtilities method invokeLater to ensure that the GUI updates occur in the event-dispatch thread). If any other message is received, line 155 simply displays the message.

Fig. 27.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 1 of 2.)
This case study is available at www.deitel.com/books/jhtp9/. Chat rooms provide a central location where users can chat with each other via short text messages. Each participant can see all the messages that the other users post, and each user can post messages. This case study integrates many of the Java networking, multithreading and Swing GUI features you’ve learned thus far to build an online chat system. We also introduce multicasting, which enables an application to send DatagramPackets to groups of clients.

The DeitelMessenger case study is a significant application that uses many intermediate Java features, such as networking with Sockets, DatagramPackets and MulticastSockets, multithreading and Swing GUI. The case study also demonstrates good software engineering practices by separating interface from implementation and enabling developers to support different network protocols and provide different user interfaces. After reading this case study, you’ll be able to build more significant networking applications.

27.10 Wrap-Up

In this chapter, you learned the basics of network programming in Java. We began with a simple applet and application in which Java performed the networking for you. You then learned two different methods of sending data over a network—streams-based networking using TCP/IP and datagrams-based networking using UDP. We showed how to build simple client/server chat programs using both streams-based and datagram-based networking. You then saw a client/server Tic-Tac-Toe game that enables two clients to play by interacting with a multithreaded server that maintains the game’s state and logic. In the next chapter, you’ll learn basic database concepts, how to interact with data in a database using SQL and how to use JDBC to allow Java applications to manipulate database data.

Summary

Section 27.1 Introduction

• Java provides stream sockets and datagram sockets (p. 1119). With stream sockets (p. 1119), a process establishes a connection (p. 1119) to another process. While the connection is in place, data flows between the processes in streams. Stream sockets are said to provide a connection-ori-
Chapter 27 Networking

The protocol used for transmission is the popular TCP (Transmission Control Protocol; p. 1119).

- With datagram sockets (datagram socket), individual packets of information are transmitted. UDP (User Datagram Protocol; p. 1119) is a connectionless service that does not guarantee that packets will not be lost, duplicated or arrive out of sequence.

Section 27.2 Manipulating URLs

- The HTTP protocol (p. 1120) uses URIs (p. 1120) to locate data on the Internet. Common URIs represent files or directories and can represent complex tasks such as database lookups and Internet searches. A URI that represents a website or web page is called a URL (p. 1120).

- Applet method `getAppletContext` (p. 1124) returns an AppletContext (p. 1120) that represents the browser in which the applet is executing. AppletContext method `showDocument` (p. 1120) receives a URL and passes it to the AppletContext, which displays the corresponding web resource. A second version of `showDocument` enables an applet to specify the target frame (p. 1124) in which to display a web resource.

Section 27.3 Reading a File on a Web Server

- `JEditorPane` (p. 1125) method `setPage` (p. 1127) downloads the document specified by its argument and displays it.

- Typically, an HTML document contains hyperlinks that link to other documents on the web. If an HTML document is displayed in an uneditable `JEditorPane` and the user clicks a hyperlink (p. 1127), a `HyperlinkEvent` (p. 1125) occurs and the `HyperlinkListener`s are notified.

- `HyperlinkEvent` method `getEventType` (p. 1127) determines the event type. `HyperlinkEvent` contains nested class `EventType` (p. 1127), which declares event types `ACTIVATED`, `ENTERED` and `EXITED`. `HyperlinkEvent` method `getURL` (p. 1127) obtains the URL represented by the hyperlink.

Section 27.4 Establishing a Simple Server Using Stream Sockets

- Stream-based connections (p. 1119) are managed with `Socket` objects (p. 1128).

- A `ServerSocket` object (p. 1128) establishes the port (p. 1128) where a server (p. 1119) waits for connections from clients (p. 1119). `ServerSocket` method `accept` (p. 1128) waits indefinitely for a connection from a client and returns a `Socket` object when a connection is established.

- `Socket` methods `getOutputStream` and `getInputStream` (p. 1129) get references to a `Socket`’s `OutputStream` and `InputStream`, respectively. Method `close` (p. 1129) terminates a connection.

Section 27.5 Establishing a Simple Client Using Stream Sockets

- A server name and port number (p. 1128) are specified when creating a `Socket` object to enable it to connect a client to the server. A failed connection attempt throws an `IOException`.

- `InetAddress` method `getByName` (p. 1142) returns an `InetAddress` object (p. 1136) containing the IP address of the specified computer. `InetAddress` method `getLocalHost` (p. 1142) returns an `InetAddress` object containing the IP address of the local computer executing the program.

Section 27.7 Datagrams: Connectionless Client/Server Interaction

- Connection-oriented transmission is like the telephone system—you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you aren’t talking.

- Connectionless transmission (p. 1142) with datagrams is similar to mail carried via the postal service. A large message that will not fit in one envelope can be broken into separate message pieces that are placed in separate, sequentially numbered envelopes. All the letters are then mailed at once. They could arrive in order, out of order or not at all.
• DatagramPacket objects store packets of data that are to be sent or that are received by an application. DatagramSockets send and receive DatagramPackets.

• The DatagramSocket constructor that takes no arguments binds the DatagramSocket to a port chosen by the computer executing the program. The one that takes an integer port-number argument binds the DatagramSocket to the specified port. If a DatagramSocket constructor fails to bind the DatagramSocket to a port, a SocketException occurs (p. 1143). DatagramSocket method receive (p. 1146) blocks (waits) until a packet arrives, then stores the packet in its argument.

• DatagramPacket method getAddress (p. 1146) returns an InetAddress object containing information about the computer from or to which the packet was sent. Method getPort (p. 1146) returns an integer specifying the port number (p. 1128) through which the DatagramPacket was sent or received. Method getLength (getLength) returns the number of bytes of data in a DatagramPacket. Method getData (p. 1146) returns a byte array containing the data.

• The DatagramPacket constructor for a packet to be sent takes four arguments—the byte array to be sent, the number of bytes to be sent, the client address to which the packet will be sent and the port number where the client is waiting to receive packets.

• DatagramSocket method send (p. 1146) sends a DatagramPacket out over the network.

• If an error occurs when receiving or sending a DatagramPacket, an IOException occurs.

Self-Review Exercises

27.1 Fill in the blanks in each of the following statements:
   a) Exception _______ occurs when an input/output error occurs when closing a socket.
   b) Exception _______ occurs when a hostname indicated by a client cannot be resolved to an address.
   c) If a DatagramSocket constructor fails to set up a DatagramSocket properly, an exception of type _______ occurs.
   d) Many of Java’s networking classes are contained in package _______.
   e) Class _______ binds the application to a port for datagram transmission.
   f) An object of class _______ contains an IP address.
   g) The two types of sockets we discussed in this chapter are _______ and _______.
   h) The acronym URL stands for _______.
   i) The acronym URI stands for _______.
   j) The key protocol that forms the basis of the World Wide Web is _______.
   k) AppletContext method _______ receives a URL object as an argument and displays in a browser the World Wide Web resource associated with that URL.
   l) Method getLocalHost returns a(n) _______ object containing the local IP address of the computer on which the program is executing.
   m) The URL constructor determines whether its String argument is a valid URL. If so, the URL object is initialized with that location. If not, a(n) _______ exception occurs.

27.2 State whether each of the following is true or false. If false, explain why.
   a) UDP is a connection-oriented protocol.
   b) With stream sockets a process establishes a connection to another process.
   c) A server waits at a port for connections from a client.
   d) Datagram packet transmission over a network is reliable—packets are guaranteed to arrive in sequence.

Answers to Self-Review Exercises

27.1 a) IOException. b) UnknownHostException. c) SocketException. d) java.net. e) DatagramSocket. f) InetAddress. g) stream sockets, datagram sockets. h) Uniform Resource Locator.
1168 Chapter 27 Networking

i) Uniform Resource Identifier. j) HTTP. k) showDocument. l) InetAddress. m) MalformedURLException.

27.2 a) False; UDP is a connectionless protocol and TCP is a connection-oriented protocol. b) True. c) True. d) False; packets can be lost, arrive out of order or be duplicated.

Exercises

27.3 Distinguish between connection-oriented and connectionless network services.

27.4 How does a client determine the hostname of the client computer?

27.5 Under what circumstances would a SocketException be thrown?

27.6 How can a client get a line of text from a server?

27.7 Describe how a client connects to a server.

27.8 Describe how a server sends data to a client.

27.9 Describe how to prepare a server to receive a stream-based connection from a single client.

27.10 How does a server listen for streams-based socket connections at a port?

27.11 What determines how many connect requests from clients can wait in a queue to connect to a server?

27.12 As described in the text, what reasons might cause a server to refuse a connection request from a client?

27.13 Use a socket connection to allow a client to specify a file name of a text file and have the server send the contents of the file or indicate that the file does not exist.

27.14 Modify Exercise 27.13 to allow the client to modify the contents of the file and send the file back to the server for storage. The user can edit the file in a JTextArea, then click a save changes button to send the file back to the server.

27.15 Modify the program in Fig. 27.2 to allow users to add their own sites to the list and remove sites from the list.

27.16 (Multithreaded Server) Multithreaded servers are quite popular today, especially because of the increasing use of multicore servers. Modify the simple server application presented in Section 27.6 to be a multithreaded server. Then use several client applications and have each of them connect to the server simultaneously. Use an ArrayList to store the client threads. ArrayList provides several methods to use in this exercise. Method size determines the number of elements in an ArrayList. Method get returns the element in the location specified by its argument. Method add places its argument at the end of the ArrayList. Method remove deletes its argument from the ArrayList.

27.17 (Checkers Game) In the text, we presented a Tic-Tac-Toe program controlled by a multithreaded server. Develop a checkers program modeled after the Tic-Tac-Toe program. The two users should alternate making moves. Your program should mediate the players’ moves, determining whose turn it is and allowing only valid moves. The players themselves will determine when the game is over.

27.18 (Chess Game) Develop a chess-playing program modeled after Exercise 27.17.

27.19 (Blackjack Game) Develop a blackjack card game program in which the server application deals cards to each of the clients. The server should deal additional cards (per the rules of the game) to each player as requested.

27.20 (Poker Game) Develop a poker game in which the server application deals cards to each client. The server should deal additional cards (per the rules of the game) to each player as requested.

27.21 (Modifications to the Multithreaded Tic-Tac-Toe Program) The programs in Figs. 27.13 and 27.15 implemented a multithreaded, client/server version of the game of Tic-Tac-Toe. Our goal in developing this game was to demonstrate a multithreaded server that could process multiple connections from clients at the same time. The server in the example is really a mediator between
the two client applets—it makes sure that each move is valid and that each client moves in the proper order. The server does not determine who won or lost or whether there was a draw. Also, there’s no capability to allow a new game to be played or to terminate an existing game.

The following is a list of suggested modifications to Figs. 27.13 and 27.15:

a) Modify the `TicTacToeServer` class to test for a win, loss or draw after each move. Send a message to each client that indicates the result of the game when the game is over.

b) Modify the `TicTacToeClient` class to display a button that when clicked allows the client to play another game. The button should be enabled only when a game completes. Both class `TicTacToeClient` and class `TicTacToeServer` must be modified to reset the board and all state information. Also, the other `TicTacToeClient` should be notified that a new game is about to begin so that its board and state can be reset.

c) Modify the `TicTacToeClient` class to provide a button that allows a client to terminate the program at any time. When the user clicks the button, the server and the other client should be notified. The server should then wait for a connection from another client so that a new game can begin.

d) Modify the `TicTacToeClient` class and the `TicTacToeServer` class so that the winner of a game can choose game piece X or O for the next game. Remember: X always goes first.

e) If you’d like to be ambitious, allow a client to play against the server while the server waits for a connection from another client.

27.22 (3-D Multithreaded Tic-Tac-Toe) Modify the multithreaded, client/server Tic-Tac-Toe program to implement a three-dimensional 4-by-4-by-4 version of the game. Implement the server application to mediate between the two clients. Display the three-dimensional board as four boards containing four rows and four columns each. If you’re ambitious, try the following modifications:

a) Draw the board in a three-dimensional manner.

b) Allow the server to test for a win, loss or draw. Beware! There are many possible ways to win on a 4-by-4-by-4 board!

27.23 (Networked Morse Code) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (e.g., period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems. Separation between words is indicated by a space or, simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short time during which no sound is transmitted. The international version of the Morse code appears in Fig. 27.18.

<table>
<thead>
<tr>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
<th>Character</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>.-</td>
<td>J</td>
<td>.---</td>
<td>S</td>
<td>...</td>
<td>1</td>
<td>..----</td>
</tr>
<tr>
<td>B</td>
<td>-...</td>
<td>K</td>
<td>-.</td>
<td>T</td>
<td>-</td>
<td>2</td>
<td>..----</td>
</tr>
<tr>
<td>C</td>
<td>-.</td>
<td>L</td>
<td>..-</td>
<td>U</td>
<td>..-</td>
<td>3</td>
<td>...--</td>
</tr>
<tr>
<td>D</td>
<td>.</td>
<td>M</td>
<td>--</td>
<td>V</td>
<td>...-</td>
<td>4</td>
<td>.....</td>
</tr>
<tr>
<td>E</td>
<td>.</td>
<td>N</td>
<td>-.</td>
<td>W</td>
<td>.--</td>
<td>5</td>
<td>.....</td>
</tr>
<tr>
<td>F</td>
<td>.--</td>
<td>O</td>
<td>---</td>
<td>X</td>
<td>-.--</td>
<td>6</td>
<td>.....</td>
</tr>
<tr>
<td>G</td>
<td>--.</td>
<td>P</td>
<td>.-.</td>
<td>Y</td>
<td>.--</td>
<td>7</td>
<td>.....</td>
</tr>
<tr>
<td>H</td>
<td>....</td>
<td>Q</td>
<td>--.</td>
<td>Z</td>
<td>.--</td>
<td>8</td>
<td>--..</td>
</tr>
<tr>
<td>I</td>
<td>..</td>
<td>R</td>
<td>.-</td>
<td></td>
<td></td>
<td>9</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>-----</td>
</tr>
</tbody>
</table>

Fig. 27.18 | Letters and digits in international Morse code.
Write a client/server application in which two clients can send Morse-code messages to each other through a multithreaded server application. The client application should allow the user to type English-language phrases in a JTextArea. When the user sends the message, the client application encodes the text into Morse code and sends the coded message through the server to the other client. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word. When messages are received, they should be decoded and displayed as normal characters and as Morse code. The client should have one JTextField for typing and one JTextArea for displaying the other client’s messages.