



Chapter 27

Networking

Java How to Program, 9/e



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
 - 27.2** Manipulating URLs
 - 27.3** Reading a File on a Web Server
 - 27.4** Establishing a Simple Server Using Stream Sockets
 - 27.5** Establishing a Simple Client Using Stream Sockets
 - 27.6** Client/Server Interaction with Stream Socket Connections
 - 27.7** Datagrams: Connectionless Client/Server Interaction
 - 27.8** Client/Server Tic-Tac-Toe Using a Multithreaded Server
 - 27.9** [Web Bonus] Case Study: DeitelMessenger
 - 27.10** Wrap-Up
-



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's fundamental networking capabilities are declared by the classes and interfaces of package `java.net`
- ▶ **Stream-based communications** that enable applications to view networking as streams of data.
- ▶ **Packet-based communications** for transmitting individual **packets** of information—commonly used to transmit data images, audio and video over the Internet.
- ▶ 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.



27.1 Introduction (cont.)

- ▶ Socket-based communications enable applications to view networking as if it were file I/O.
 - The socket is simply a software construct that represents one endpoint of a connection.
- ▶ 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).



27.1 Introduction (cont.)

- ▶ With **datagram sockets**, individual **packets** of information are transmitted.
- ▶ **UDP**—the **User Datagram Protocol**—is a **connectionless service**, and thus does not guarantee that packets arrive in any particular order.
 - Packets can even be lost or duplicated.
- ▶ UDP is most appropriate for network applications that do not require the error checking and reliability of TCP.



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.



27.2 Manipulating URLs

- ▶ HyperText Transfer Protocol (HTTP) forms the basis of the World Wide Web
 - uses URIs (Uniform Resource Identifiers) to identify data on the Internet.
- ▶ URIs that specify the locations of documents are called URLs (Uniform Resource Locators).
- ▶ AppletContext method `showDocument` causes the browser to access and display a resource.



```
1 <html>
2 <head>
3     <title>Site Selector</title>
4 </head>
5 <body>
6     <applet code = "SiteSelector.class" width = "300" height = "75">
7         <param name = "title0" value = "Java Home Page">
8         <param name = "location0"
9             value = "http://www.oracle.com/technetwork/java/">
10        <param name = "title1" value = "Deitel">
11        <param name = "location1" value = "http://www.deitel.com/">
12        <param name = "title2" value = "jGuru">
13        <param name = "location2" value = "http://www.jGuru.com/">
14        <param name = "title3" value = "JavaWorld">
15        <param name = "location3" value = "http://www.javaworld.com/">
16    </applet>
17 </body>
18 </html>
```

Fig. 27.1 | HTML document to load SiteSelector applet.

```
1 // Fig. 27.2: SiteSelector.java
2 // Loading a document from a URL into a browser.
3 import java.net.MalformedURLException;
4 import java.net.URL;
5 import java.util.HashMap;
6 import java.util.ArrayList;
7 import java.awt.BorderLayout;
8 import java.applet.AppletContext;
9 import javax.swing.JApplet;
10 import javax.swing.JLabel;
11 import javax.swing.JList;
12 import javax.swing.JScrollPane;
13 import javax.swing.event.ListSelectionEvent;
14 import javax.swing.event.ListSelectionListener;
15
```

Fig. 27.2 | Loading a document from a URL into a browser. (Part I of 6.)



```
16 public class SiteSelector extends JApplet
17 {
18     private HashMap< String, URL > sites; // site names and URLs
19     private ArrayList< String > siteNames; // site names
20     private JList siteChooser; // list of sites to choose from
21
22     // read parameters and set up GUI
23     public void init()
24     {
25         sites = new HashMap< String, URL >(); // create HashMap
26         siteNames = new ArrayList< String >(); // create ArrayList
27
28         // obtain parameters from HTML document
29         getSitesFromHTMLParameters();
30     }
```

Fig. 27.2 | Loading a document from a URL into a browser. (Part 2 of 6.)



```
31 // create GUI components and lay out interface
32 add( new JLabel( "Choose a site to browse" ), BorderLayout.NORTH );
33
34 siteChooser = new JList( siteNames.toArray() ); // populate JList
35 siteChooser.addListSelectionListener(
36     new ListSelectionListener() // anonymous inner class
37     {
38         // go to site user selected
39         public void valueChanged( ListSelectionEvent event )
40         {
41             // get selected site name
42             Object object = siteChooser.getSelectedValue();
43
44             // use site name to locate corresponding URL
45             URL newDocument = sites.get( object );
46
47             // get applet container
48             AppletContext browser = getAppletContext();
49
50             // tell applet container to change pages
51             browser.showDocument( newDocument );
52         } // end method valueChanged
53     } // end anonymous inner class
54 ); // end call to addListSelectionListener
```

Fig. 27.2 | Loading a document from a URL into a browser. (Part 3 of 6.)



```
55      add( new JScrollPane( siteChooser ), BorderLayout.CENTER );
56  } // end method init
57
58
59 // obtain parameters from HTML document
60 private void getsitesFromHTMLParameters()
61 {
62     String title; // site title
63     String location; // location of site
64     URL url; // URL of location
65     int counter = 0; // count number of sites
66
67     title = getParameter( "title" + counter ); // get first site title
68
```

Fig. 27.2 | Loading a document from a URL into a browser. (Part 4 of 6.)



```
69     // loop until no more parameters in HTML document
70     while ( title != null )
71     {
72         // obtain site location
73         location = getParameter( "location" + counter );
74
75         try // place title/URL in HashMap and title in ArrayList
76         {
77             url = new URL( location ); // convert location to URL
78             sites.put( title, url ); // put title/URL in HashMap
79             siteNames.add( title ); // put title in ArrayList
80         } // end try
81         catch ( MalformedURLException urlException )
82         {
83             urlException.printStackTrace();
84         } // end catch
85
86         ++counter;
87         title = getParameter( "title" + counter ); // get next site title
88     } // end while
89 } // end method getSitesFromHTMLParameters
90 } // end class SiteSelector
```

Fig. 27.2 | Loading a document from a URL into a browser. (Part 5 of 6.)

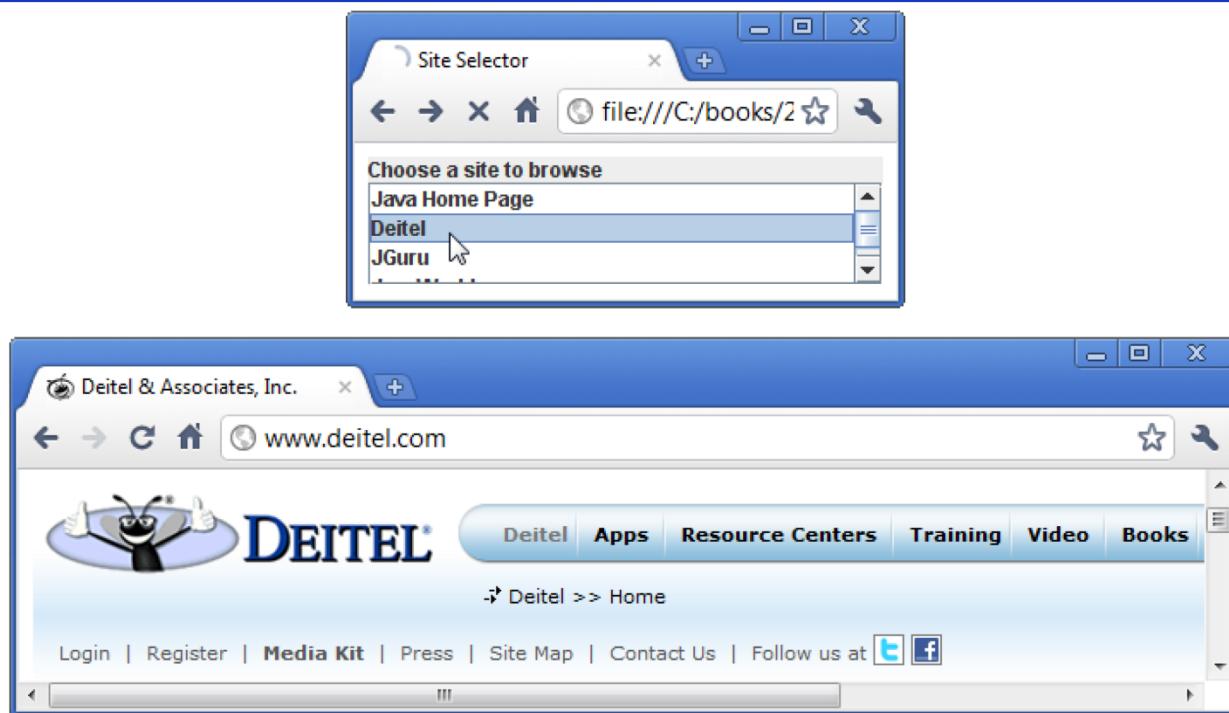


Fig. 27.2 | Loading a document from a URL into a browser. (Part 6 of 6.)



27.2 Manipulating URLs (cont.)

- ▶ This applet takes advantage of **applet parameters** specified in the HTML document that invokes the applet.
- ▶ The HTML document contains eight **applet 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**.
 - Argument is a **String** containing the name of the parameter in the **param** element.



27.2 Manipulating URLs (cont.)

- ▶ The **URL** constructor determines whether its argument represents a properly constructed URL.
 - otherwise, throws a **MalformedURLException**.
- ▶ **Applet** method **getAppletContext** gets a reference to an **AppletContext** object that represents the applet container.
 - Used to invoke method **showDocument** to change web pages.
- ▶ A second version of **AppletContext** method **showDocument** enables an applet to specify the so-called **target frame** in which to display the web resource.
- ▶ Target frames:
 - **_blank** results in a new web browser window.
 - **_self** displays content in the same frame as the applet.
 - **_top** removes the current frames in the browser window, then displays the content 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

- ▶ Swing GUI component **JEditorPane** (from package `javax.swing`) can display the contents of a file on a web server.
- ▶ Class **JEditorPane** can render both plain text and XHTML-formatted text
 - this application acts as a simple web browser.
- ▶ Can process **HyperlinkEvents** when the user clicks a hyperlink in the **JEditorPane**.



```
1 // Fig. 27.3: ReadServerFile.java
2 // Reading a file by opening a connection through a URL.
3 import java.awt.BorderLayout;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.io.IOException;
7 import javax.swing.JEditorPane;
8 import javax.swing.JFrame;
9 import javax.swing.JOptionPane;
10 import javax.swing.JScrollPane;
11 import javax.swing.JTextField;
12 import javax.swing.event.HyperlinkEvent;
13 import javax.swing.event.HyperlinkListener;
14
15 public class ReadServerFile extends JFrame
16 {
17     private JTextField enterField; // JTextField to enter site name
18     private JEditorPane contentsArea; // to display website
19 }
```

Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 1 of 4.)



```
20    // set up GUI
21    public ReadServerFile()
22    {
23        super( "Simple Web Browser" );
24
25        // create enterField and register its listener
26        enterField = new JTextField( "Enter file URL here" );
27        enterField.addActionListener(
28            new ActionListener()
29            {
30                // get document specified by user
31                public void actionPerformed( ActionEvent event )
32                {
33                    getThePage( event.getActionCommand() );
34                } // end method actionPerformed
35            } // end inner class
36        ); // end call to addActionListener
37
38        add( enterField, BorderLayout.NORTH );
```

Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 2 of 4.)



```
39
40     contentsArea = new JEditorPane(); // create contentsArea
41     contentsArea.setEditable( false );
42     contentsArea.addHyperlinkListener(
43         new HyperlinkListener()
44     {
45         // if user clicked hyperlink, go to specified page
46         public void hyperlinkUpdate( HyperlinkEvent event )
47         {
48             if ( event.EventType() ==
49                 HyperlinkEvent.EventType.ACTIVATED )
50                 getThePage( event.getURL().toString() );
51         } // end method hyperlinkUpdate
52     } // end inner class
53 ); // end call to addHyperlinkListener
54
55     add( new JScrollPane( contentsArea ), BorderLayout.CENTER );
56     setSize( 400, 300 ); // set size of window
57     setVisible( true ); // show window
58 } // end ReadServerFile constructor
59
```

Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 3 of 4.)



```
60 // Load document
61 private void getThePage( String location )
62 {
63     try // Load document and display location
64     {
65         contentsArea.setPage( location ); // set the page
66         enterField.setText( location ); // set the text
67     } // end try
68     catch ( IOException ioException )
69     {
70         JOptionPane.showMessageDialog( this,
71             "Error retrieving specified URL", "Bad URL",
72             JOptionPane.ERROR_MESSAGE );
73     } // end catch
74 } // end method getThePage
75 } // end class ReadServerFile
```

Fig. 27.3 | Reading a file by opening a connection through a URL. (Part 4 of 4.)

```
1 // Fig. 27.4: ReadServerFileTest.java
2 // Create and start a ReadServerFile.
3 import javax.swing.JFrame;
4
5 public class ReadServerFileTest
6 {
7     public static void main( String[] args )
8     {
9         ReadServerFile application = new ReadServerFile();
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11    } // end main
12 } // end class ReadServerFileTest
```

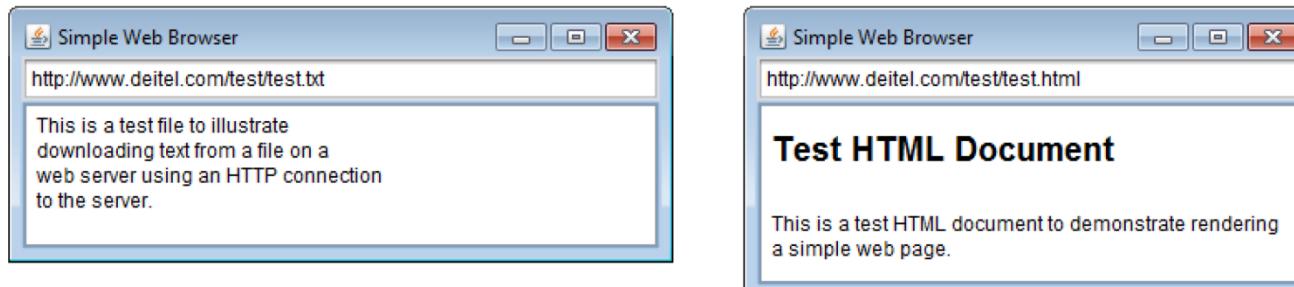


Fig. 27.4 | Test class for ReadServerFile.



27.3 Reading a File on a Web Server (cont.)

- ▶ **JEditorPane** method `setPage` downloads a document and displays it.
 - If there is an error, throws an `IOException`.
 - if an invalid URL, throws a `MalformedURLException`.
- ▶ If a **JEditorPane** contains an XHTML document and the user clicks a hyperlink, the **JEditorPane** generates a **HyperlinkEvent** (package `javax.swing.event`) and notifies all registered **HyperlinkListeners** (package `javax.swing.event`) of that event.
- ▶ When a **HyperlinkEvent** occurs, the program calls method `hyperlinkUpdate` on the registered listener(s).



27.3 Reading a File on a Web Server (cont.)

- ▶ `HyperlinkEvent` method `getEventType` returns the type of the `HyperlinkEvent`.
- ▶ `HyperlinkEvent` nested class `EventType` declares three `static EventType` objects
 - `ACTIVATED` indicates that the user clicked a hyperlink to change web pages
 - `ENTERED` indicates that the user moved the mouse over a hyperlink
 - `EXITED` indicates that the user moved the mouse away from a hyperlink
- ▶ `HyperlinkEvent` method `getURL` obtains the URL represented by the hyperlink.



Look-and-Feel Observation 27.1

A JEditorPane generates HyperlinkEvents only if it's uneditable.



27.4 Establishing a Simple Server Using Stream Sockets

- ▶ Establishing a simple server in Java requires five steps.
- ▶ *Step 1 is to create a **ServerSocket** object.*
- ▶ **ServerSocket** constructor
 - `ServerSocket server = new ServerSocket(
 portNumber, queueLength);`
- ▶ The constructor establishes the **port number** 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**.



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.

11



27.4 Establishing a Simple Server Using Stream Sockets (cont.)

- ▶ 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
 - `Socket connection = server.accept();`
 - returns a **Socket** when a connection with a client is established.
- ▶ The **Socket** allows the server to interact with the client.



27.4 Establishing a Simple Server Using Stream Sockets (cont.)

- ▶ *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 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`.
- ▶ *Step 4* is the processing phase, in which the server and the client communicate via the `OutputStream` and `InputStream` objects.
- ▶ 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.
- ▶ In *Step 1*, the **Socket** constructor establishes a connection to the server.
 - `Socket connection = new Socket(serverAddress, port);`
 - If the connection attempt is successful, this statement returns a **Socket**.
 - A connection attempt that fails throws an instance of a subclass of **IOException**.
 - An **UnknownHostException** occurs when the system is unable to resolve the server name.



27.5 Establishing a Simple Client Using Stream Sockets (cont.)

- ▶ In *Step 2*, the client uses `Socket` methods `getInputStream` and `getOutputStream` to obtain references to the `Socket`'s `InputStream` and `OutputStream`.
- ▶ *Step 3* is the processing phase in which the client and the server communicate via the `InputStream` and `OutputStream` objects.
- ▶ 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`.



27.6 Client/Server Interaction with Stream Socket Connections

- ▶ Figures 27.5 and 27.7 present a simple client-/server chat application.



```
1 // Fig. 27.5: Server.java
2 // Server portion of a client/server stream-socket connection.
3 import java.io.EOFException;
4 import java.io.IOException;
5 import java.io.ObjectInputStream;
6 import java.io.ObjectOutputStream;
7 import java.net.ServerSocket;
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 Server extends JFrame
19 {
20     private JTextField enterField; // inputs message from user
21     private JTextArea displayArea; // display information to user
22     private ObjectOutputStream output; // output stream to client
23     private ObjectInputStream input; // input stream from client
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part I of 9.)



```
24     private ServerSocket server; // server socket
25     private Socket connection; // connection to client
26     private int counter = 1; // counter of number of connections
27
28     // set up GUI
29     public Server()
30     {
31         super( "Server" );
32
33         enterField = new JTextField(); // create enterField
34         enterField.setEditable( false );
35         enterField.addActionListener(
36             new ActionListener()
37             {
38                 // send message to client
39                 public void actionPerformed( ActionEvent event )
40                 {
41                     sendData( event.getActionCommand() );
42                     enterField.setText( "" );
43                 } // end method actionPerformed
44             } // end anonymous inner class
45         ); // end call to addActionListener
46     }
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 2 of 9.)



```
47     add( enterField, BorderLayout.NORTH );
48
49     displayArea = new JTextArea(); // create displayArea
50     add( new JScrollPane( displayArea ), BorderLayout.CENTER );
51
52     setSize( 300, 150 ); // set size of window
53     setVisible( true ); // show window
54 } // end Server constructor
55
56 // set up and run server
57 public void runServer()
58 {
59     try // set up server to receive connections; process connections
60     {
61         server = new ServerSocket( 12345, 100 ); // create ServerSocket
62
63         while ( true )
64         {
65             try
66             {
67                 waitForConnection(); // wait for a connection
68                 getStreams(); // get input & output streams
69                 processConnection(); // process connection
70             } // end try

```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 3 of 9.)



```
71         catch ( EOFException eofException )
72     {
73         displayMessage( "\nServer terminated connection" );
74     } // end catch
75     finally
76     {
77         closeConnection(); // close connection
78         ++counter;
79     } // end finally
80 } // end while
81 } // end try
82 catch ( IOException ioException )
83 {
84     ioException.printStackTrace();
85 } // end catch
86 } // end method runServer
87
88 // wait for connection to arrive, then display connection info
89 private void waitForConnection() throws IOException
90 {
91     displayMessage( "Waiting for connection\n" );
92     connection = server.accept(); // allow server to accept connection
93     displayMessage( "Connection " + counter + " received from: " +
94                     connection.getInetAddress().getHostName() );
95 } // end method waitForConnection
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 4 of 9.)



```
96
97 // get streams to send and receive data
98 private void getStreams() throws IOException
99 {
100     // set up output stream for objects
101     output = new ObjectOutputStream( connection.getOutputStream() );
102     output.flush(); // flush output buffer to send header information
103
104     // set up input stream for objects
105     input = new ObjectInputStream( connection.getInputStream() );
106
107     displayMessage( "\nGot I/O streams\n" );
108 } // end method getStreams
109
110 // process connection with client
111 private void processConnection() throws IOException
112 {
113     String message = "Connection successful";
114     sendData( message ); // send connection successful message
115
116     // enable enterField so server user can send messages
117     setTextFieldEditable( true );
118 }
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 5 of 9.)



```
I19      do // process messages sent from client
I20    {
I21      try // read message and display it
I22    {
I23          message = ( String ) input.readObject(); // read new message
I24          displayMessage( "\n" + message ); // display message
I25      } // end try
I26      catch ( ClassNotFoundException classNotFoundException )
I27      {
I28          displayMessage( "\nUnknown object type received" );
I29      } // end catch
I30
I31      } while ( !message.equals( "CLIENT>> TERMINATE" ) );
I32  } // end method processConnection
I33
I34 // close streams and socket
I35 private void closeConnection()
I36 {
I37     displayMessage( "\nTerminating connection\n" );
I38     setTextFieldEditable( false ); // disable enterField
I39
I40     try
I41     {
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 6 of 9.)



```
142     output.close(); // close output stream
143     input.close(); // close input stream
144     connection.close(); // close socket
145 } // end try
146 catch ( IOException ioException )
147 {
148     ioException.printStackTrace();
149 } // end catch
150 } // end method closeConnection
151
152 // send message to client
153 private void sendData( String message )
154 {
155     try // send object to client
156     {
157         output.writeObject( "SERVER>>> " + message );
158         output.flush(); // flush output to client
159         displayMessage( "\nSERVER>>> " + message );
160     } // end try
161     catch ( IOException ioException )
162     {
163         displayArea.append( "\nError writing object" );
164     } // end catch
165 } // end method sendData
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 7 of 9.)



```
166
167     // manipulates displayArea in the event-dispatch thread
168     private void displayMessage( final String messageToDisplay )
169     {
170         SwingUtilities.invokeLater(
171             new Runnable()
172             {
173                 public void run() // updates displayArea
174                 {
175                     displayArea.append( messageToDisplay ); // append message
176                 } // end method run
177             } // end anonymous inner class
178         ); // end call to SwingUtilities.invokeLater
179     } // end method displayMessage
180
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 8 of 9.)



```
I81 // manipulates enterField in the event-dispatch thread
I82 private void setTextFieldEditable( final boolean editable )
I83 {
I84     SwingUtilities.invokeLater(
I85         new Runnable()
I86         {
I87             public void run() // sets enterField's editability
I88             {
I89                 enterField.setEditable( editable );
I90             } // end method run
I91         } // end inner class
I92     ); // end call to SwingUtilities.invokeLater
I93 } // end method setTextFieldEditable
I94 } // end class Server
```

Fig. 27.5 | Server portion of a client/server stream-socket connection. (Part 9 of 9.)



```
1 // Fig. 27.6: ServerTest.java
2 // Test the Server application.
3 import javax.swing.JFrame;
4
5 public class ServerTest
6 {
7     public static void main( String[] args )
8     {
9         Server application = new Server(); // create server
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        application.runServer(); // run server application
12    } // end main
13 } // end class ServerTest
```

Fig. 27.6 | Test class for Server.



27.6 Client/Server Interaction with Stream Socket Connections (cont.)

- ▶ Method `runServer` sets up the server to receive a connection and processes one connection at a time.
- ▶ Methods `displayMessage` uses the event-dispatch thread to display messages in the application's `JTextArea`.
- ▶ `SwingUtilities` method `invokeLater` receives a `Runnable` object as its argument and places it into the event-dispatch thread for execution.
 - Ensures that we don't modify a GUI component from a thread other than the event-dispatch thread.
 - We use a similar technique in method `setTextFieldEditable`.



Common Programming Error 27.1

Specifying a port that's already in use or specifying an invalid port number when creating a `ServerSocket` results in a `BindException`.



27.6 Client/Server Interaction with Stream Socket Connections (cont.)

- ▶ Socket method `getInetAddress` returns an `InetAddress` (package `java.net`) containing information about the client computer.
- ▶ `InetAddress` method `getHostName` returns the host name of the client computer.
- ▶ IP address (`127.0.0.1`) and host name (`localhost`) are useful for testing networking applications on your local computer
 - known as the `loopback` address



27.6 Client/Server Interaction with Stream Socket Connections (cont.)

- ▶ Note the call to `ObjectOutputStream` method **flush**.
 - 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 ObjectOutputStream 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.



27.6 Client/Server Interaction with Stream Socket Connections (cont.)

- ▶ 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.
- ▶ The following command attempts to connect to the **Server** on the computer with IP address **192.168.1.15**
 - `java client 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
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 1 of 9.)



```
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
27
28     // initialize chatServer and set up GUI
29     public Client( String host )
30     {
31         super( "Client" );
32
33         chatServer = host; // set server to which this client connects
34
35         enterField = new JTextField(); // create enterField
36         enterField.setEditable( false );
37         enterField.addActionListener(
38             new ActionListener()
39             {
40                 // send message to server
41                 public void actionPerformed( ActionEvent event )
42                 {
43                     sendData( event.getActionCommand() );
44                     enterField.setText( "" );
45                 } // end method actionPerformed
46             } // end anonymous inner class
47         ); // end call to addActionListener
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 2 of 9)



```
48     add( enterField, BorderLayout.NORTH );
49
50
51     displayArea = new JTextArea(); // create displayArea
52     add( new JScrollPane( displayArea ), BorderLayout.CENTER );
53
54     setSize( 300, 150 ); // set size of window
55     setVisible( true ); // show window
56 } // end Client constructor
57
58 // connect to server and process messages from server
59 public void runClient()
60 {
61     try // connect to server, get streams, process connection
62     {
63         connectToServer(); // create a Socket to make connection
64         getStreams(); // get the input and output streams
65         processConnection(); // process connection
66     } // end try
67     catch ( EOFException eofException )
68     {
69         displayMessage( "\nClient terminated connection" );
70     } // end catch
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 3 of 9.)



```
71     catch ( IOException ioException )
72     {
73         ioException.printStackTrace();
74     } // end catch
75     finally
76     {
77         closeConnection(); // close connection
78     } // end finally
79 } // end method runClient
80
81 // connect to server
82 private void connectToServer() throws IOException
83 {
84     displayMessage( "Attempting connection\n" );
85
86     // create Socket to make connection to server
87     client = new Socket( InetAddress.getByName( chatServer ), 12345 );
88
89     // display connection information
90     displayMessage( "Connected to: " +
91         client.getInetAddress().getHostName() );
92 } // end method connectToServer
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 4 of 9.)



```
93
94 // get streams to send and receive data
95 private void getStreams() throws IOException
96 {
97     // set up output stream for objects
98     output = new ObjectOutputStream( client.getOutputStream() );
99     output.flush(); // flush output buffer to send header information
100
101    // set up input stream for objects
102    input = new ObjectInputStream( client.getInputStream() );
103
104    displayMessage( "\nGot I/O streams\n" );
105 } // end method getStreams
106
107 // process connection with server
108 private void processConnection() throws IOException
109 {
110     // enable enterField so client user can send messages
111     setTextFieldEditable( true );
112
113     do // process messages sent from server
114     {
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 5 of 9.)



```
115     try // read message and display it
116     {
117         message = ( String ) input.readObject(); // read new message
118         displayMessage( "\n" + message ); // display message
119     } // end try
120     catch ( ClassNotFoundException classNotFoundException )
121     {
122         displayMessage( "\nUnknown object type received" );
123     } // end catch
124
125     } while ( !message.equals( "SERVER>>> TERMINATE" ) );
126 } // end method processConnection
127
128 // close streams and socket
129 private void closeConnection()
130 {
131     displayMessage( "\nClosing connection" );
132     setTextFieldEditable( false ); // disable enterField
133
134     try
135     {
136         output.close(); // close output stream
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 6 of 9.)



```
137     input.close(); // close input stream 1
138     client.close(); // close socket
139 } // end try
140 catch ( IOException ioException )
141 {
142     ioException.printStackTrace();
143 } // end catch
144 } // end method closeConnection
145
146 // send message to server
147 private void sendData( String message )
148 {
149     try // send object to server
150     {
151         output.writeObject( "CLIENT>>> " + message );
152         output.flush(); // flush data to output
153         displayMessage( "\nCLIENT>>> " + message );
154     } // end try
155     catch ( IOException ioException )
156     {
157         displayArea.append( "\nError writing object" );
158     } // end catch
159 } // end method sendData
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 7 of 9.)



```
160
161 // manipulates displayArea in the event-dispatch thread
162 private void displayMessage( final String messageToDisplay )
163 {
164     SwingUtilities.invokeLater(
165         new Runnable()
166         {
167             public void run() // updates displayArea
168             {
169                 displayArea.append( messageToDisplay );
170             } // end method run
171         } // end anonymous inner class
172     ); // end call to SwingUtilities.invokeLater
173 } // end method displayMessage
174
175 // manipulates enterField in the event-dispatch thread
176 private void setTextFieldEditable( final boolean editable )
177 {
178     SwingUtilities.invokeLater(
179         new Runnable()
180     {
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 8 of 9.)



```
181     public void run() // sets enterField's editability
182     {
183         enterField.setEditable( editable );
184     } // end method run
185 } // end anonymous inner class
186 ); // end call to SwingUtilities.invokeLater
187 } // end method setTextFieldEditable
188 } // end class Client
```

Fig. 27.7 | Client portion of a stream-socket connection between client and server.
(Part 9 of 9.)



```
1 // Fig. 27.8: ClientTest.java
2 // Class that tests the Client.
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main( String[] args )
8     {
9         Client application; // declare client application
10
11        // if no command line args
12        if ( args.length == 0 )
13            application = new Client( "127.0.0.1" ); // connect to localhost
14        else
15            application = new Client( args[ 0 ] ); // use args to connect
16
17        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
18        application.runClient(); // run client application
19    } // end main
20 } // end class ClientTest
```

Fig. 27.8 | Class that tests the Client. (Part I of 2.)

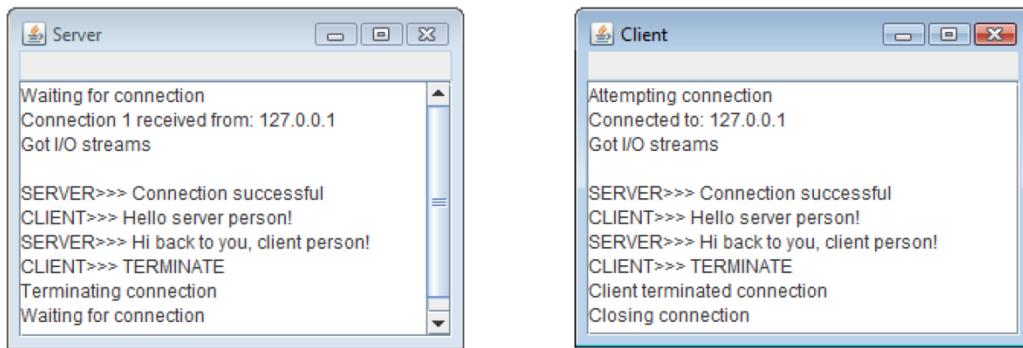


Fig. 27.8 | Class that tests the Client. (Part 2 of 2.)



27.6 Client/Server Interaction with Stream Socket Connections (cont.)

- ▶ `InetAddress static` method `getByName` returns an `InetAddress` object containing the specified IP address
 - 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()`



27.7 Connectionless Client/Server Interaction with Datagrams

- ▶ **Connectionless transmission with datagrams.**
- ▶ 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 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).



27.7 Connectionless Client/Server Interaction with Datagrams (cont.)

- ▶ Class **Server** declares two **DatagramPackets** that the server uses to send and receive information and one **DatagramSocket** that sends and receives the packets.
- ▶ The **DatagramSocket** constructor that takes an integer port-number argument binds 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 **Socket-Exception** 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.



```
1 // Fig. 27.9: Server.java
2 // Server side of connectionless client/server computing with datagrams.
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net DatagramSocket;
6 import java.net.SocketException;
7 import java.awt.BorderLayout;
8 import javax.swing.JFrame;
9 import javax.swing.JScrollPane;
10 import javax.swing.JTextArea;
11 import javax.swing.SwingUtilities;
12
13 public class Server extends JFrame
14 {
15     private JTextArea displayArea; // displays packets received
16     private DatagramSocket socket; // socket to connect to client
17 }
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams.
(Part 1 of 5.)



```
18 // set up GUI and DatagramSocket
19 public Server()
20 {
21     super( "Server" );
22
23     displayArea = new JTextArea(); // create displayArea
24     add( new JScrollPane( displayArea ), BorderLayout.CENTER );
25     setSize( 400, 300 ); // set size of window
26     setVisible( true ); // show window
27
28     try // create DatagramSocket for sending and receiving packets
29     {
30         socket = new DatagramSocket( 5000 );
31     } // end try
32     catch ( SocketException socketException )
33     {
34         socketException.printStackTrace();
35         System.exit( 1 );
36     } // end catch
37 } // end Server constructor
38
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams.
(Part 2 of 5.)



```
39 // wait for packets to arrive, display data and echo packet to client
40 public void waitForPackets()
41 {
42     while ( true )
43     {
44         try // receive packet, display contents, return copy to client
45         {
46             byte[] data = new byte[ 100 ]; // set up packet
47             DatagramPacket receivePacket =
48                 new DatagramPacket( data, data.length );
49
50             socket.receive( receivePacket ); // wait to receive packet
51
52             // display information from received packet
53             displayMessage( "\nPacket received:" +
54                 "\nFrom host: " + receivePacket.getAddress() +
55                 "\nHost port: " + receivePacket.getPort() +
56                 "\nLength: " + receivePacket.getLength() +
57                 "\nContaining:\n\t" + new String( receivePacket.getData(),
58                     0, receivePacket.getLength() ) );
59
60             sendPacketToClient( receivePacket ); // send packet to client
61         } // end try
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams.
(Part 3 of 5.)



```
62         catch ( IOException ioException )
63     {
64         displayMessage( ioException + "\n" );
65         ioException.printStackTrace();
66     } // end catch
67 } // end while
68 } // end method waitForPackets
69
70 // echo packet to client
71 private void sendPacketToClient( DatagramPacket receivePacket )
72 throws IOException
73 {
74     displayMessage( "\n\nEcho data to client..." );
75
76     // create packet to send
77     DatagramPacket sendPacket = new DatagramPacket(
78         receivePacket.getData(), receivePacket.getLength(),
79         receivePacket.getAddress(), receivePacket.getPort() );
80
81     socket.send( sendPacket ); // send packet to client
82     displayMessage( "Packet sent\n" );
83 } // end method sendPacketToClient
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams.
(Part 4 of 5.)

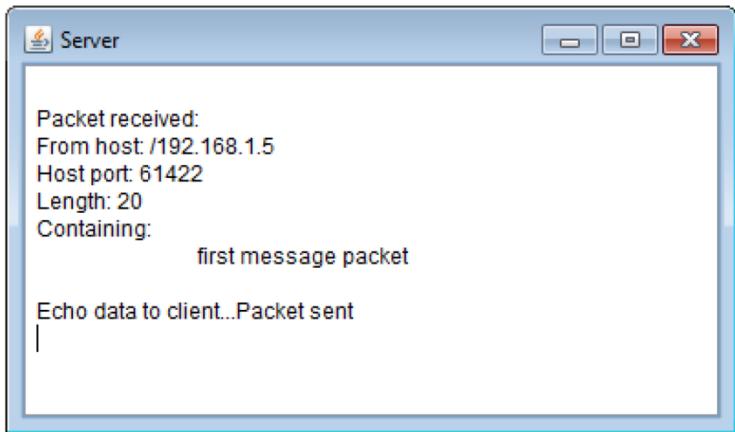
```
84  
85     // manipulates displayArea in the event-dispatch thread  
86     private void displayMessage( final String messageToDisplay )  
87     {  
88         SwingUtilities.invokeLater(  
89             new Runnable()  
90             {  
91                 public void run() // updates displayArea  
92                 {  
93                     displayArea.append( messageToDisplay ); // display message  
94                 } // end method run  
95             } // end anonymous inner class  
96         ); // end call to SwingUtilities.invokeLater  
97     } // end method displayMessage  
98 } // end class Server
```

Fig. 27.9 | Server side of connectionless client/server computing with datagrams.
(Part 5 of 5.)



```
1 // Fig. 27.10: ServerTest.java
2 // Class that tests the Server.
3 import javax.swing.JFrame;
4
5 public class ServerTest
6 {
7     public static void main( String[] args )
8     {
9         Server application = new Server(); // create server
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        application.waitForPackets(); // run server application
12    } // end main
13 } // end class ServerTest
```

Fig. 27.10 | Class that tests the Server. (Part 1 of 2.)



Server window after packet
of data is received from Client

Fig. 27.10 | Class that tests the Server. (Part 2 of 2.)



27.7 Connectionless Client/Server Interaction with Datagrams (cont.)

- ▶ **DatagramSocket** method `receive` waits for a packet to arrive at the Server.
 - 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.
- ▶ **DatagramPacket** method `getAddress` returns an **InetAddress** object containing the IP address of the computer from which the packet was sent.
- ▶ Method `getPort` returns an integer specifying the port number through which the client computer sent the packet.
- ▶ Method `getLength` returns an integer representing the number of bytes of data received.
- ▶ Method `getData` returns a **byte** array containing the data.
- ▶ **DatagramSocket** method `send` throws an **IOException** if an error occurs while sending a packet.



```
1 // Fig. 27.11: Client.java
2 // Client side of connectionless client/server computing with datagrams.
3 import java.io.IOException;
4 import java.net.DatagramPacket;
5 import java.net.DatagramSocket;
6 import java.net.InetAddress;
7 import java.net.SocketException;
8 import java.awt.BorderLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Client extends JFrame
18 {
19     private JTextField enterField; // for entering messages
20     private JTextArea displayArea; // for displaying messages
21     private DatagramSocket socket; // socket to connect to server
22 }
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 1 of 7.)

```
23     // set up GUI and DatagramSocket
24     public Client()
25     {
26         super( "Client" );
27
28         enterField = new JTextField( "Type message here" );
29         enterField.addActionListener(
30             new ActionListener()
31             {
32                 public void actionPerformed( ActionEvent event )
33                 {
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 2 of 7.)



```
34         try // create and send packet
35     {
36         // get message from textfield
37         String message = event.getActionCommand();
38         displayArea.append( "\nSending packet containing: " +
39             message + "\n" );
40
41         byte[] data = message.getBytes(); // convert to bytes
42
43         // create sendPacket
44         DatagramPacket sendPacket = new DatagramPacket( data,
45             data.length, InetAddress.getLocalHost(), 5000 );
46
47         socket.send( sendPacket ); // send packet
48         displayArea.append( "Packet sent\n" );
49         displayArea.setCaretPosition(
50             displayArea.getText().length() );
51     } // end try
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 3 of 7.)



```
52         catch ( IOException ioException )
53     {
54         displayMessage( ioException + "\n" );
55         ioException.printStackTrace();
56     } // end catch
57 } // end actionPerformed
58 } // end inner class
59 ); // end call to addActionListener
60
61 add( enterField, BorderLayout.NORTH );
62
63 displayArea = new JTextArea();
64 add( new JScrollPane( displayArea ), BorderLayout.CENTER );
65
66 setSize( 400, 300 ); // set window size
67 setVisible( true ); // show window
68
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 4 of 7.)

```
69     try // create DatagramSocket for sending and receiving packets
70     {
71         socket = new DatagramSocket();
72     } // end try
73     catch ( SocketException socketException )
74     {
75         socketException.printStackTrace();
76         System.exit( 1 );
77     } // end catch
78 } // end Client constructor
79
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 5 of 7.)



```
80 // wait for packets to arrive from Server, display packet contents
81 public void waitForPackets()
82 {
83     while ( true )
84     {
85         try // receive packet and display contents
86         {
87             byte[] data = new byte[ 100 ]; // set up packet
88             DatagramPacket receivePacket = new DatagramPacket(
89                 data, data.length );
90
91             socket.receive( receivePacket ); // wait for packet
92
93             // display packet contents
94             displayMessage( "\nPacket received:" +
95                             "\nFrom host: " + receivePacket.getAddress() +
96                             "\nHost port: " + receivePacket.getPort() +
97                             "\nLength: " + receivePacket.getLength() +
98                             "\nContaining:\n\t" + new String( receivePacket.getData(),
99                                         0, receivePacket.getLength() ) );
100        } // end try
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.

(Part 6 of 7.)



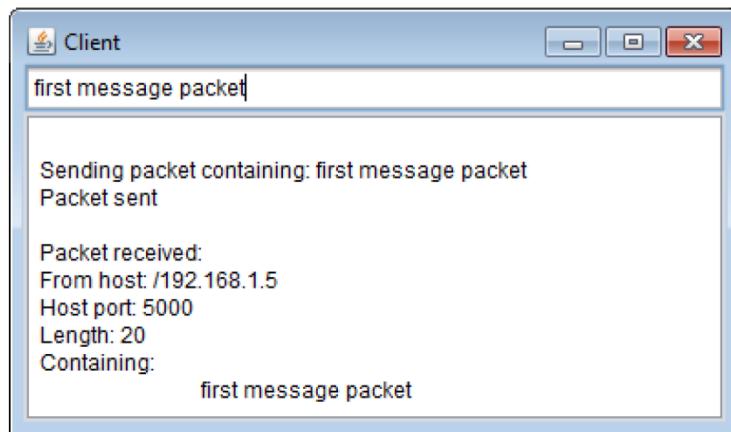
```
101     catch ( IOException exception )
102     {
103         displayMessage( exception + "\n" );
104         exception.printStackTrace();
105     } // end catch
106 } // end while
107 } // end method waitForPackets
108
109 // manipulates displayArea in the event-dispatch thread
110 private void displayMessage( final String messageToDisplay )
111 {
112     SwingUtilities.invokeLater(
113         new Runnable()
114     {
115         public void run() // updates displayArea
116         {
117             displayArea.append( messageToDisplay );
118         } // end method run
119     } // end inner class
120 ); // end call to SwingUtilities.invokeLater
121 } // end method displayMessage
122 } // end class Client
```

Fig. 27.11 | Client side of connectionless client/server computing with datagrams.
(Part 7 of 7.)



```
1 // Fig. 27.12: ClientTest.java
2 // Tests the Client class.
3 import javax.swing.JFrame;
4
5 public class ClientTest
6 {
7     public static void main( String[] args )
8     {
9         Client application = new Client(); // create client
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        application.waitForPackets(); // run client application
12    } // end main
13 } // end class ClientTest
```

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



Client window after sending packet to Server and receiving packet back from Server

Fig. 27.12 | Class that tests the Client. (Part 2 of 2.)



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

- ▶ Tic-Tac-Toe implemented by using client/server techniques with stream sockets.
- ▶ Sample outputs are shown in Fig. 27.17.
- ▶ As the `TicTacToeServer` receives each client connection, it creates an instance of inner-class `Player` 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.



```
1 // Fig. 27.13: TicTacToeServer.java
2 // Server side of client/server Tic-Tac-Toe program.
3 import java.awt.BorderLayout;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6 import java.io.IOException;
7 import java.util.Formatter;
8 import java.util.Scanner;
9 import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13 import java.util.concurrent.locks.Condition;
14 import javax.swing.JFrame;
15 import javax.swing.JTextArea;
16 import javax.swing.SwingUtilities;
17
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part I of 17.)



```
18 public class TicTacToeServer extends JFrame
19 {
20     private String[] board = new String[ 9 ]; // tic-tac-toe board
21     private JTextArea outputArea; // for outputting moves
22     private Player[] players; // array of Players
23     private ServerSocket server; // server socket to connect with clients
24     private int currentPlayer; // keeps track of player with current move
25     private final static int PLAYER_X = 0; // constant for first player
26     private final static int PLAYER_O = 1; // constant for second player
27     private final static String[] MARKS = { "X", "O" }; // array of marks
28     private ExecutorService runGame; // will run players
29     private Lock gameLock; // to lock game for synchronization
30     private Condition otherPlayerConnected; // to wait for other player
31     private Condition otherPlayerTurn; // to wait for other player's turn
32
33     // set up tic-tac-toe server and GUI that displays messages
34     public TicTacToeServer()
35     {
36         super( "Tic-Tac-Toe Server" ); // set title of window
37
38         // create ExecutorService with a thread for each player
39         runGame = Executors.newFixedThreadPool( 2 );
40         gameLock = new ReentrantLock(); // create lock for game
41     }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 2 of 17.)



```
42 // condition variable for both players being connected
43 otherPlayerConnected = gameLock.newCondition();
44
45 // condition variable for the other player's turn
46 otherPlayerTurn = gameLock.newCondition();
47
48 for ( int i = 0; i < 9; i++ )
49     board[ i ] = new String( "" ); // create tic-tac-toe board
50 players = new Player[ 2 ]; // create array of players
51 currentPlayer = PLAYER_X; // set current player to first player
52
53 try
54 {
55     server = new ServerSocket( 12345, 2 ); // set up ServerSocket
56 } // end try
57 catch ( IOException ioException )
58 {
59     ioException.printStackTrace();
60     System.exit( 1 );
61 } // end catch
62
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 3 of 17.)



```
63     outputArea = new JTextArea(); // create JTextArea for output
64     add( outputArea, BorderLayout.CENTER );
65     outputArea.setText( "Server awaiting connections\n" );
66
67     setSize( 300, 300 ); // set size of window
68     setVisible( true ); // show window
69 } // end TicTacToeServer constructor
70
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 4 of 17.)



```
71 // wait for two connections so game can be played
72 public void execute()
73 {
74     // wait for each client to connect
75     for ( int i = 0; i < players.length; i++ )
76     {
77         try // wait for connection, create Player, start runnable
78         {
79             players[ i ] = new Player( server.accept(), i );
80             runGame.execute( players[ i ] ); // execute player runnable
81         } // end try
82         catch ( IOException ioException )
83         {
84             ioException.printStackTrace();
85             System.exit( 1 );
86         } // end catch
87     } // end for
88 }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 5 of 17.)



```
89     gameLock.lock(); // lock game to signal player X's thread
90
91     try
92     {
93         players[ PLAYER_X ].setSuspended( false ); // resume player X
94         otherPlayerConnected.signal(); // wake up player X's thread
95     } // end try
96     finally
97     {
98         gameLock.unlock(); // unlock game after signalling player X
99     } // end finally
100    } // end method execute
101
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 6 of 17.)



```
102 // display message in outputArea
103 private void displayMessage( final String messageToDisplay )
104 {
105     // display message from event-dispatch thread of execution
106     SwingUtilities.invokeLater(
107         new Runnable()
108     {
109         public void run() // updates outputArea
110         {
111             outputArea.append( messageToDisplay ); // add message
112         } // end method run
113     } // end inner class
114 ); // end call to SwingUtilities.invokeLater
115 } // end method displayMessage
116
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 7 of 17.)



```
117 // determine if move is valid
118 public boolean validateAndMove( int location, int player )
119 {
120     // while not current player, must wait for turn
121     while ( player != currentPlayer )
122     {
123         gameLock.lock(); // lock game to wait for other player to go
124
125         try
126         {
127             otherPlayerTurn.await(); // wait for player's turn
128         } // end try
129         catch ( InterruptedException exception )
130         {
131             exception.printStackTrace();
132         } // end catch
133         finally
134         {
135             gameLock.unlock(); // unlock game after waiting
136         } // end finally
137     } // end while
138 }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 8 of 17.)



```
139     // if location not occupied, make move
140     if ( !isOccupied( location ) )
141     {
142         board[ location ] = MARKS[ currentPlayer ]; // set move on board
143         currentPlayer = ( currentPlayer + 1 ) % 2; // change player
144
145         // let new current player know that move occurred
146         players[ currentPlayer ].otherPlayerMoved( location );
147
148         gameLock.lock(); // lock game to signal other player to go
149
150         try
151         {
152             otherPlayerTurn.signal(); // signal other player to continue
153         } // end try
154         finally
155         {
156             gameLock.unlock(); // unlock game after signaling
157         } // end finally
158     }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 9 of 17.)



```
159         return true; // notify player that move was valid
160     } // end if
161     else // move was not valid
162         return false; // notify player that move was invalid
163 } // end method validateAndMove
164
165 // determine whether location is occupied
166 public boolean isOccupied( int location )
167 {
168     if ( board[ location ].equals( MARKS[ PLAYER_X ] ) ||
169         board [ location ].equals( MARKS[ PLAYER_O ] ) )
170         return true; // location is occupied
171     else
172         return false; // location is not occupied
173 } // end method isOccupied
174
175 // place code in this method to determine whether game over
176 public boolean isGameOver()
177 {
178     return false; // this is left as an exercise
179 } // end method isGameOver
180
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 10 of 17.)



```
I81 // private inner class Player manages each Player as a runnable
I82 private class Player implements Runnable
I83 {
I84     private Socket connection; // connection to client
I85     private Scanner input; // input from client
I86     private Formatter output; // output to client
I87     private int playerNumber; // tracks which player this is
I88     private String mark; // mark for this player
I89     private boolean suspended = true; // whether thread is suspended
I90
I91     // set up Player thread
I92     public Player( Socket socket, int number )
I93     {
I94         playerNumber = number; // store this player's number
I95         mark = MARKS[ playerNumber ]; // specify player's mark
I96         connection = socket; // store socket for client
I97     }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 11 of 17.)



```
198     try // obtain streams from Socket
199     {
200         input = new Scanner( connection.getInputStream() );
201         output = new Formatter( connection.getOutputStream() );
202     } // end try
203     catch ( IOException ioException )
204     {
205         ioException.printStackTrace();
206         System.exit( 1 );
207     } // end catch
208 } // end Player constructor
209
210 // send message that other player moved
211 public void otherPlayerMoved( int location )
212 {
213     output.format( "Opponent moved\n" );
214     output.format( "%d\n", location ); // send location of move
215     output.flush(); // flush output
216 } // end method otherPlayerMoved
217
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 12 of 17.)



```
218     // control thread's execution
219     public void run()
220     {
221         // send client its mark (X or O), process messages from client
222         try
223         {
224             displayMessage( "Player " + mark + " connected\n" );
225             output.format( "%s\n", mark ); // send player's mark
226             output.flush(); // flush output
227 }
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 13 of 17.)



```
228     // if player X, wait for another player to arrive
229     if ( playerName == PLAYER_X )
230     {
231         output.format( "%s\n%s", "Player X connected",
232                         "Waiting for another player\n" );
233         output.flush(); // flush output
234
235         gameLock.lock(); // lock game to wait for second player
236
237         try
238         {
239             while( suspended )
240             {
241                 otherPlayerConnected.await(); // wait for player 0
242             } // end while
243         } // end try
244         catch ( InterruptedException exception )
245         {
246             exception.printStackTrace();
247         } // end catch
248         finally
249         {
250             gameLock.unlock(); // unlock game after second player
251         } // end finally
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 14 of 17.)



```
252
253         // send message that other player connected
254         output.format( "Other player connected. Your move.\n" );
255         output.flush(); // flush output
256     } // end if
257     else
258     {
259         output.format( "Player 0 connected, please wait\n" );
260         output.flush(); // flush output
261     } // end else
262
263     // while game not over
264     while ( !isGameOver() )
265     {
266         int location = 0; // initialize move location
267
268         if ( input.hasNext() )
269             location = input.nextInt(); // get move location
270
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 15 of 17.)



```
271     // check for valid move
272     if ( validateAndMove( location, playerNumber ) )
273     {
274         displayMessage( "\nlocation: " + location );
275         output.format( "Valid move.\n" ); // notify client
276         output.flush(); // flush output
277     } // end if
278     else // move was invalid
279     {
280         output.format( "Invalid move, try again\n" );
281         output.flush(); // flush output
282     } // end else
283     } // end while
284 } // end try
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 16 of 17.)



```
285         finally
286     {
287         try
288     {
289             connection.close(); // close connection to client
290         } // end try
291         catch ( IOException ioException )
292     {
293             ioException.printStackTrace();
294             System.exit( 1 );
295         } // end catch
296     } // end finally
297 } // end method run
298
299 // set whether or not thread is suspended
300 public void setSuspended( boolean status )
301 {
302     suspended = status; // set value of suspended
303 } // end method setSuspended
304 } // end class Player
305 } // end class TicTacToeServer
```

Fig. 27.13 | Server side of client/server Tic-Tac-Toe program. (Part 17 of 17.)



```
1 // Fig. 27.14: TicTacToeServerTest.java
2 // Class that tests Tic-Tac-Toe server.
3 import javax.swing.JFrame;
4
5 public class TicTacToeServerTest
6 {
7     public static void main( String[] args )
8     {
9         TicTacToeServer application = new TicTacToeServer();
10        application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11        application.execute();
12    } // end main
13 } // end class TicTacToeServerTest
```

Fig. 27.14 | Class that tests Tic-Tac-Toe server. (Part I of 2.)

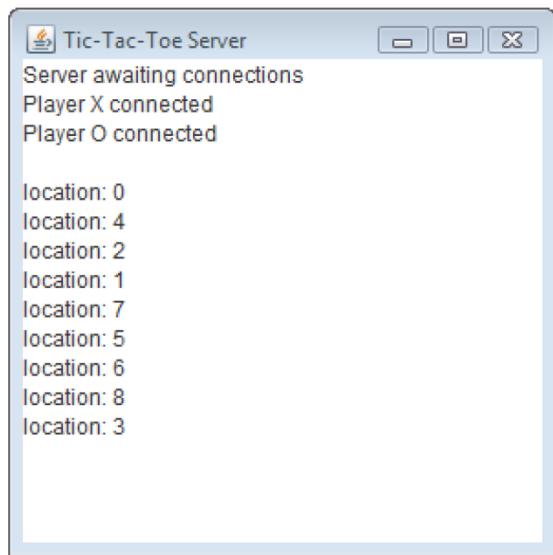


Fig. 27.14 | Class that tests Tic-Tac-Toe server. (Part 2 of 2.)



27.8 Client/Server Tic-Tac-Toe Using a Multithreaded Server (cont.)

- ▶ When a client connects, a new `Player` object is created to manage the connection as a separate thread, and the thread is executed in the `runGame` thread pool.
- ▶ The `Player` constructor receives the `Socket` object representing the connection to the client and gets the associated input and output streams.
- ▶ The `Player`'s `run` method controls the information that is sent to and received from the client.



```
1 // Fig. 27.15: TicTacToeClient.java
2 // Client side of client/server Tic-Tac-Toe program.
3 import java.awt.BorderLayout;
4 import java.awt.Dimension;
5 import java.awt.Graphics;
6 import java.awt.GridLayout;
7 import java.awt.event.MouseAdapter;
8 import java.awt.event.MouseEvent;
9 import java.net.Socket;
10 import java.net.InetAddress;
11 import java.io.IOException;
12 import javax.swing.JFrame;
13 import javax.swing.JPanel;
14 import javax.swing.JScrollPane;
15 import javax.swing.JTextArea;
16 import javax.swing.JTextField;
17 import javax.swing.SwingUtilities;
18 import java.util.Formatter;
19 import java.util.Scanner;
20 import java.util.concurrent.Executors;
21 import java.util.concurrent.ExecutorService;
22
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part I of 14.)



```
23 public class TicTacToeClient extends JFrame implements Runnable
24 {
25     private JTextField idField; // textfield to display player's mark
26     private JTextArea displayArea; // JTextArea to display output
27     private JPanel boardPanel; // panel for tic-tac-toe board
28     private JPanel panel2; // panel to hold board
29     private Square[][] board; // tic-tac-toe board
30     private Square currentSquare; // current square
31     private Socket connection; // connection to server
32     private Scanner input; // input from server
33     private Formatter output; // output to server
34     private String ticTacToeHost; // host name for server
35     private String myMark; // this client's mark
36     private boolean myTurn; // determines which client's turn it is
37     private final String X_MARK = "X"; // mark for first client
38     private final String O_MARK = "O"; // mark for second client
39
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 2 of 14.)



```
40 // set up user-interface and board
41 public TicTacToeClient( String host )
42 {
43     ticTacToeHost = host; // set name of server
44     displayArea = new JTextArea( 4, 30 ); // set up JTextArea
45     displayArea.setEditable( false );
46     add( new JScrollPane( displayArea ), BorderLayout.SOUTH );
47
48     boardPanel = new JPanel(); // set up panel for squares in board
49     boardPanel.setLayout( new GridLayout( 3, 3, 0, 0 ) );
50
51     board = new Square[ 3 ][ 3 ]; // create board
52
53     // loop over the rows in the board
54     for ( int row = 0; row < board.length; row++ )
55     {
56         // loop over the columns in the board
57         for ( int column = 0; column < board[ row ].length; column++ )
58         {
59             // create square
60             board[ row ][ column ] = new Square( ' ', row * 3 + column );
61             boardPanel.add( board[ row ][ column ] ); // add square
62         } // end inner for
63     } // end outer for
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 3 of 14.)



```
64      idField = new JTextField(); // set up textfield
65      idField.setEditable( false );
66      add( idField, BorderLayout.NORTH );
67
68
69      panel2 = new JPanel(); // set up panel to contain boardPanel
70      panel2.add( boardPanel, BorderLayout.CENTER ); // add board panel
71      add( panel2, BorderLayout.CENTER ); // add container panel
72
73      setSize( 300, 225 ); // set size of window
74      setVisible( true ); // show window
75
76      startClient();
77  } // end TicTacToeClient constructor
78
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 4 of 14.)



```
79 // start the client thread
80 public void startClient()
81 {
82     try // connect to server and get streams
83     {
84         // make connection to server
85         connection = new Socket(
86             InetAddress.getByName( ticTacToeHost ), 12345 );
87
88         // get streams for input and output
89         input = new Scanner( connection.getInputStream() );
90         output = new Formatter( connection.getOutputStream() );
91     } // end try
92     catch ( IOException ioException )
93     {
94         ioException.printStackTrace();
95     } // end catch
96
97     // create and start worker thread for this client
98     ExecutorService worker = Executors.newFixedThreadPool( 1 );
99     worker.execute( this ); // execute client
100 } // end method startClient
101
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 5 of 14.)



```
102 // control thread that allows continuous update of displayArea
103 public void run()
104 {
105     myMark = input.nextLine(); // get player's mark (X or O)
106
107     SwingUtilities.invokeLater(
108         new Runnable()
109         {
110             public void run()
111             {
112                 // display player's mark
113                 idField.setText( "You are player \\" + myMark + "\\\" );
114             } // end method run
115         } // end anonymous inner class
116     ); // end call to SwingUtilities.invokeLater
117
118     myTurn = ( myMark.equals( X_MARK ) ); // determine if client's turn
119
120     // receive messages sent to client and output them
121     while ( true )
122     {
123         if ( input.hasNextLine() )
124             processMessage( input.nextLine() );
125     } // end while
126 } // end method run
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 6 of 14.)



```
I27
I28     // process messages received by client
I29     private void processMessage( String message )
I30     {
I31         // valid move occurred
I32         if ( message.equals( "Valid move." ) )
I33         {
I34             displayMessage( "Valid move, please wait.\n" );
I35             setMark( currentSquare, myMark ); // set mark in square
I36         } // end if
I37         else if ( message.equals( "Invalid move, try again" ) )
I38         {
I39             displayMessage( message + "\n" ); // display invalid move
I40             myTurn = true; // still this client's turn
I41         } // end else if
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 7 of 14.)



```
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
149         setMark( board[ row ][ column ],
150                 ( myMark.equals( X_MARK ) ? O_MARK : X_MARK ) ); // mark move
151         displayMessage( "Opponent moved. Your turn.\n" );
152         myTurn = true; // now this client's turn
153     } // end else if
154     else
155         displayMessage( message + "\n" ); // display the message
156 } // end method processMessage
157
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 8 of 14.)



```
158 // manipulate displayArea in event-dispatch thread
159 private void displayMessage( final String messageToDisplay )
160 {
161     SwingUtilities.invokeLater(
162         new Runnable()
163     {
164         public void run()
165         {
166             displayArea.append( messageToDisplay ); // updates output
167         } // end method run
168     } // end inner class
169 ); // end call to SwingUtilities.invokeLater
170 } // end method displayMessage
171
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 9 of 14.)



```
I72 // utility method to set mark on board in event-dispatch thread
I73 private void setMark( final Square squareToMark, final String mark )
I74 {
I75     SwingUtilities.invokeLater(
I76         new Runnable()
I77         {
I78             public void run()
I79             {
I80                 squareToMark.setMark( mark ); // set mark in square
I81             } // end method run
I82         } // end anonymous inner class
I83     ); // end call to SwingUtilities.invokeLater
I84 } // end method setMark
I85
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 10 of 14.)



```
186 // send message to server indicating clicked square
187 public void sendClickedSquare( int location )
188 {
189     // if it is my turn
190     if ( myTurn )
191     {
192         output.format( "%d\n", location ); // send location to server
193         output.flush();
194         myTurn = false; // not my turn any more
195     } // end if
196 } // end method sendClickedSquare
197
198 // set current Square
199 public void setCurrentSquare( Square square )
200 {
201     currentSquare = square; // set current square to argument
202 } // end method setCurrentSquare
203
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 11 of 14.)



```
204 // private inner class for the squares on the board
205 private class Square extends JPanel
206 {
207     private String mark; // mark to be drawn in this square
208     private int location; // location of square
209
210     public Square( String squareMark, int squareLocation )
211     {
212         mark = squareMark; // set mark for this square
213         location = squareLocation; // set location of this square
214
215         addMouseListener(
216             new MouseAdapter()
217             {
218                 public void mouseReleased( MouseEvent e )
219                 {
220                     setCurrentSquare( Square.this ); // set current square
221
222                     // send location of this square
223                     sendClickedSquare( getSquareLocation() );
224                 } // end method mouseReleased
225             } // end anonymous inner class
226         ); // end call to addMouseListener
227     } // end Square constructor
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 12 of 14.)



```
228
229     // return preferred size of Square
230     public Dimension getPreferredSize()
231     {
232         return new Dimension( 30, 30 ); // return preferred size
233     } // end method getPreferredSize
234
235     // return minimum size of Square
236     public Dimension getMinimumSize()
237     {
238         return getPreferredSize(); // return preferred size
239     } // end method getMinimumSize
240
241     // set mark for Square
242     public void setMark( String newMark )
243     {
244         mark = newMark; // set mark of square
245         repaint(); // repaint square
246     } // end method setMark
247
```

Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 13 of 14.)



```
248     // return Square location
249     public int getSquareLocation()
250     {
251         return location; // return location of square
252     } // end method getSquareLocation
253
254     // draw Square
255     public void paintComponent( Graphics g )
256     {
257         super.paintComponent( g );
258
259         g.drawRect( 0, 0, 29, 29 ); // draw square
260         g.drawString( mark, 11, 20 ); // draw mark
261     } // end method paintComponent
262 } // end inner-class Square
263 } // end class TicTacToeClient
```

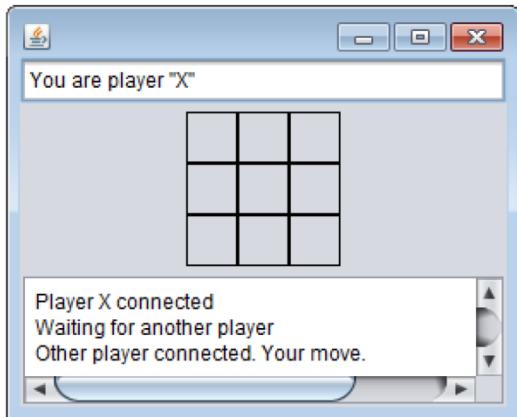
Fig. 27.15 | Client side of client/server Tic-Tac-Toe program. (Part 14 of 14.)



```
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.

a) Player X connected to server.



b) Player O connected to server.

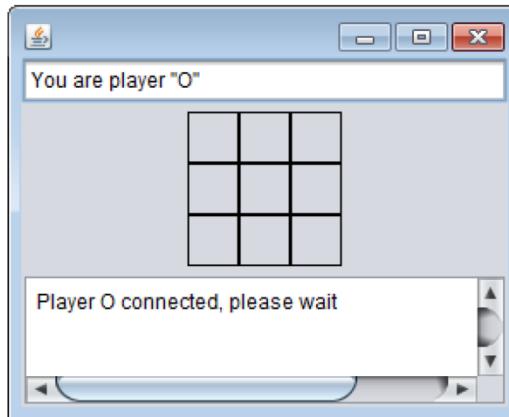
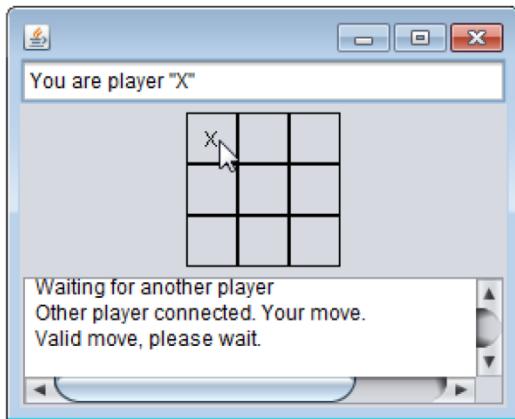


Fig. 27.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part I of 4.)

c) Player X moved.



d) Player O sees Player X's move.

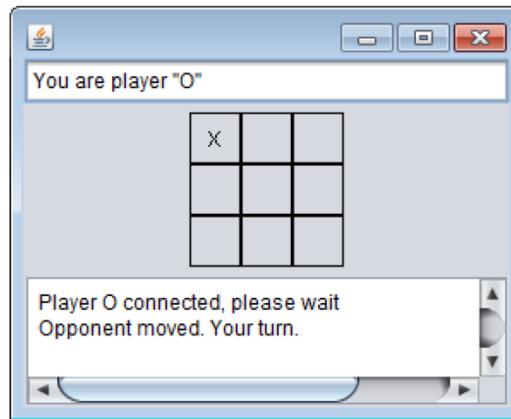
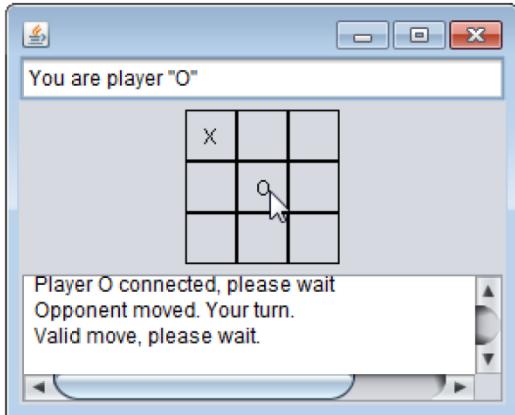


Fig. 27.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 2 of 4.)

e) Player O moved.



f) Player X sees Player O's move.

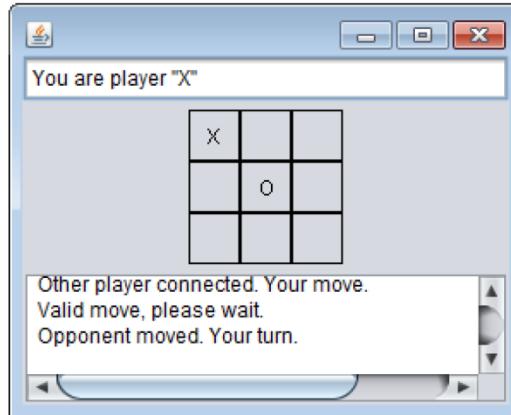
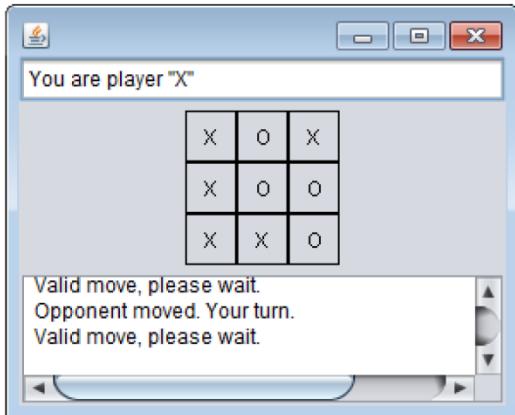


Fig. 27.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 3 of 4.)

g) Player X moved.



h) Player O sees Player X's last move.

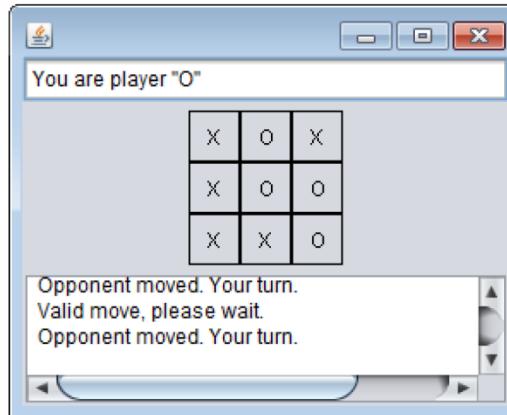


Fig. 27.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 4 of 4.)