Lecture 14

Networking

OBJECTIVES

In this lecture you will learn:

- To understand Java networking with URLs, sockets.
- To implement Java networking applications by using sockets.
- To understand how to implement Java clients and servers that communicate with one another.
- To understand how to implement network-based collaborative applications.
- To construct a multithreaded server.

Outline

24.1 Introduction

- **Networking package is `java.net`**
  - **Stream-based communications**
    - **Applications view networking as streams of data**
    - **Connection-based protocol**
    - **Uses TCP (Transmission Control Protocol**
  - **Packet-based communications**
    - **Individual packets transmitted**
    - **Connectionless service**
    - **Uses UDP (User Datagram Protocol)**

24.1 Introduction (Cont.)

- **Client-server relationship**
  - **Client requests some action be performed**
  - **Server performs the action and responds to client**
  - **Request-response model**
    - **Common implementation: Web browsers and Web servers**

Performance Tip 24.1

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

Portability Tip 24.1

TCP, UDP and related protocols enable a great variety of heterogeneous computer systems (i.e., computer systems with different processors and different operating systems) to intercommunicate.

Error-Prevention Tip 24.1

The applet in Fig. 24.2 must be run from a Web browser, such as Mozilla or Microsoft Internet Explorer, to see the results of displaying another Web page. The `appletviewer` is capable only of executing applets—it ignores all other HTML tags. If the Web sites in the program contained Java applets, only those applets would appear in the `appletviewer` when the user selected a Web site. Each applet would execute in a separate `appletviewer` window.

24.3 Reading a File on a Web Server

- Swing GUI component `JEditorPane`
  - Render both plain text and HTML-formatted text
  - Act as a simple Web browser
    - Retrieves files from a Web server at a given URI
    - `HyperlinkEvent`s
      - Occur when the user clicks a hyperlink
      - Three event types
        - `HyperlinkEvent.EventType.ACTIVATED`
        - `HyperlinkEvent.EventType.ENTERED`
        - `HyperlinkEvent.EventType.EXITED`

```java
// Fig. 24.3: ReadServerFile.java
// Use a JEditorPane to display the contents of a file on a Web server.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
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 Web site

   // set up GUI
   public ReadServerFile()
   {
      super( "Simple Web Browser" );

```

Import `JEditPane` from package `javax.swing`, import `HyperlinkEvent` and `HyperlinkListener` from package `javax.swing.event`

Declare `JEditorPane contentsArea`, which will be used to display the contents of the file

```
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 );
39
40      contentsArea = new JEditorPane(); // create contents
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.getEventType() ==
49                     HyperlinkEvent.EventType.ACTIVATED )
50                  getThePage( event.getURL().toString() );
51            } // end method hyperlinkUpdate
52         } // end inner class
53      ); // end call to addHyperlinkListener
54
```

Create `JEditorPane`

Set `JEditorPane`

Register a `HyperlinkListener` to handle `HyperlinkEvents`, which occur when the user clicks a

Method `hyperlinkUpdate`

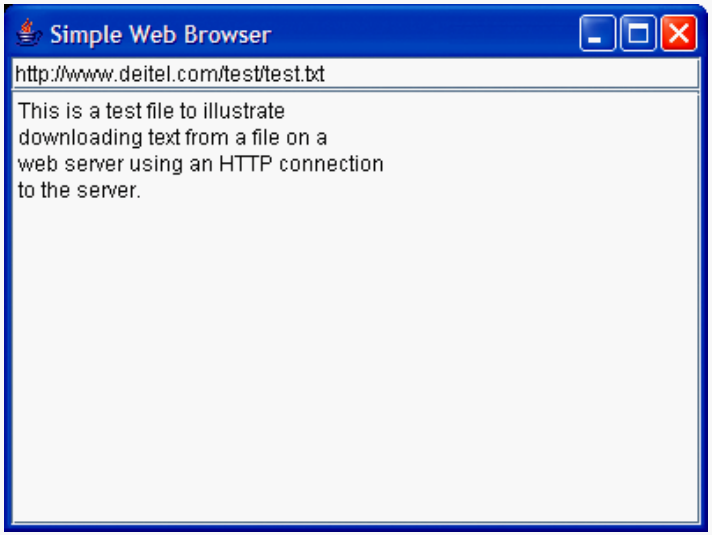
Use `HyperlinkEvent` method `getEventType` to determine the type of the `HyperlinkEvent`

Use `HyperlinkEvent` method `getURL` to obtain the URL represented by the hyperlink

```
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
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
```

Invoke `JEditorPane` method `setPage` to download the document specified by `location` and display it in the `JEditPane`

```java
// Fig. 24.4: 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 );
   } // end main
} // end class ReadServerFileTest
```




Simple Web Browser
http://www.prenhall.com/deitel
Prentice Hall
Computer Science
DEITEL & DEITEL
Sign up for the
DEITEL BUZZ ONLINE
e-mail Newsletter
Products
Books
Downloads
FAQs
Multimedia Products
Visual C++/J++ Options
What's Coming Soon
Value Added Packages
Technical Support Information
BOOKS
Operating Systems, 3/E
(2004)
Catalog Page/More Info
Sample Chapters (in Adobe Acrobat PDF format)
Accompanying Website
Companion Website (w/ Text Support Materials)

Look-and-Feel Observation 24.1

A `JEditorPane` generates `HyperlinkEvents` only if it is uneditable.

24.4 Establishing a Simple Server Using Stream Sockets

- **Five steps to create a simple server in Java**
  - ***Step 1*: Create `ServerSocket` object**
    - `ServerSocket server = new ServerSocket(` *`portNumber`*`, `*`queueLength`*` );`
    - **Register an available port**
    - **Specify a maximum number of clients**
    - **Handshake point**
    - **Binding the server to the port**
      - **Only one client can be bound to a specific port**

Software Engineering Observation 24.1

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.

24.4 Establishing a Simple Server Using Stream Sockets (Cont.)

- **Five steps to create a simple server in Java**
  - ***Step 2*: Server listens for client connection**
    - **Server blocks until client connects**
    - `Socket connection = server.accept();`
  - ***Step 3*: Sending and receiving data**
    - `OutputStream` **to send and** `InputStream` **to receive data**
      - **Method** `getOutputStream` **returns** `Socket`**'s** `OutputStream`
      - **Methods** `getInputstream` **returns** `Socket`**'s** `InputStream`

24.4 Establishing a Simple Server Using Stream Sockets (Cont.)

- **Five steps to create a simple server in Java**
  - ***Step 4*: Process phase**
    - **Server and Client communicate via streams**
  - ***Step 5*: Close streams and connections**
    - **Method `close`**

Software Engineering Observation 24.2

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 from the programmer.

Software Engineering Observation 24.3

With Java's multithreading, we can create multithreaded servers that can manage many simultaneous connections with many clients. This multithreaded-server architecture is precisely what popular network servers use.

Software Engineering Observation 24.4

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 are created. See Chapter 23 for more information on multithreading.

Performance Tip 24.2

In high-performance systems in which memory is abundant, a multithreaded server can be implemented to create a pool of threads that can be assigned quickly to handle network I/O across each new `Socket` as it is created. Thus, when the server receives a connection, it need not incur the overhead of thread creation. When the connection is closed, the thread is returned to the pool for reuse.

24.5 Establishing a Simple Client Using Stream Sockets

- **Four steps to create a simple client in Java**
  - ***Step 1*: Create a `Socket` to connect to server**

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

  - ***Step 2*: Obtain `Socket`'s `InputStream` and `Outputstream`**
  - ***Step 3*: Process information communicated**
  - ***Step 4*: Close streams and connection**

24.6 Client/Server Interaction with Stream Socket Connections

- **Client/server chat application**
  - **Uses stream sockets**
  - **Server waits for a client connection attempt**
  - **Client connects to the server**
    - **Send and receive messages**
  - **Client or server terminates the connection**
  - **Server waits for the next client to connect**

```java
// Fig. 24.5: Server.java
// Set up a Server that will receive a connection from a client, send
// a string to the client, and close the connection.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
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 c
   private ServerSocket server; // server socket
   private Socket connection; // connection to client
   private int counter = 1; // counter of number of connections

```

Import `ServerSocket` and `Socket` from package `java.net`

Declare `ServerSocket server`

Declare `Socket connection` which connects to the client

```
29    // set up GUI
30    public Server()
31    {
32       super( "Server" );
33
34       enterField = new JTextField(); // create enterField
35       enterField.setEditable( false );
36       enterField.addActionListener(
37          new ActionListener()
38          {
39             // send message to client
40             public void actionPerformed( ActionEvent event )
41             {
42                sendData( event.getActionCommand() );
43                enterField.setText( "" );
44             } // end method actionPerformed
45          } // end anonymous inner class
46       ); // end call to addActionListener
47
48       add( enterField, BorderLayout.NORTH );
49
50       displayArea = new JTextArea(); // create displayArea
51       add( new JScrollPane( displayArea ), BorderLayout.CENTER );
52
53       setSize( 300, 150 ); // set size of window
54       setVisible( true ); // show window
55    } // end Server constructor
56
```

```
57    // set up and run server
58    public void runServer()
59    {
60       try // set up server to receive connections; process connections
61       {
62          server = new ServerSocket( 12345, 100 ); // create ServerSocket
63
64          while ( true )
65          {
66             try
67             {
68                waitForConnection(); // wait for a connection
69                getStreams(); // get input & output streams
70                processConnection(); // process connection
71             } // end try
72             catch ( EOFException eofException )
73             {
74                displayMessage( "\nServer terminated connection" );
75             } // end catch
```

Create `ServerSocket` at port 12345 with queue of length 100

Wait for a client

After the connection is

Send the initial connection message to the client and process all messages received from the client

```
76            finally
77            {
78               closeConnection(); // close connection
79               counter++;
80            } // end finally
81         } // end while
82      } // end try
83      catch ( IOException ioException )
84      {
85         ioException.printStackTrace();
86      } // end catch
87   } // end method runServer
88
89   // wait for connection to arrive, then di
90   private void waitForConnection() throws I
91   {
92      displayMessage( "Waiting for connection\n" );
93      connection = server.accept(); // allow server to accept connection
94      displayMessage( "Connection " + counter + " received from: " +
95         connection.getInetAddress().getHostName() );
96   } // end method waitForConnection
97
98   // get streams to send and recei
99   private void getStreams() throws
100  {
101     // set up output stream for objects
102     output = new ObjectOutputStream( connection.getOutputStream() );
103     output.flush(); // flush output buffer to send header information
104
```

Output the host name of the computer that made the connection using `Socket` method `getInetAddress` and `InetAddress` method `getHostName`

Obtain `Socket`'s `OutputStream` and use `ctOutputStream`

Method `flush` empties output buffer and sends header information

```
105      // set up input stream for objects
106      input = new ObjectInputStream( connection.getInputStream() );
107
108      displayMessage( "\nGot I/O streams\n" );
109   } // end method getStreams
110
111   // process connection with client
112   private void processConnection() throws IOException
113   {
114      String message = "Connection successful";
115      sendData( message ); // send connection successful message
116
117      // enable enterField so server user can send messages
118      setTextFieldEditable( true );
119
120      do // process messages sent from client
121      {
122         try // read message and display it
123         {
124            message = ( String ) input.readObject(); // read new message
125            displayMessage( "\n" + message ); // display message
126         } // end try
127         catch ( ClassNotFoundException classNotFoundException )
128         {
129            displayMessage( "\nUnknown object type received" );
130         } // end catch
131
```

Obtain `Socket`'s `InputStream` and use it to initialize `ObjectInputStream`

Use `ObjectInputStream` method `readObject` to read a `String` from client

```
132      } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
133   } // end method processConnection
134
135   // close streams and socket
136   private void closeConnection()
137   {
138      displayMessage( "\nTerminating connection\n" );
139      setTextFieldEditable( false ); // disable enterField
140
141      try
142      {
143         output.close(); // close output stream
144         input.close(); // close input stream
145         connection.close(); // close socket
146      } // end try
147      catch ( IOException ioException )
148      {
149         ioException.printStackTrace();
150      } // end catch
151   } // end method closeConnection
153   // send message to client
154   private void sendData( String message )
155   {
156      try // send object to client
157      {
158         output.writeObject( "SERVER>>> " + message );
159         output.flush(); // flush output to client
160         displayMessage( "\nSERVER>>> " + message );
161      } // end try
```

Method `closeConnection` closes streams and sockets

Invoke `Socket` method `close` to close the socket

Use `ObjectOutputStream` method `writeObject` to send a `String` to client

```
162      catch ( IOException ioException )
163      {
164         displayArea.append( "\nError writing object" );
165      } // end catch
166   } // end method sendData
167
168   // manipulates displayArea in the event-dispatch thread
169   private void displayMessage( final String messageToDisplay )
170   {
171      SwingUtilities.invokeLater(
172         new Runnable()
173         {
174            public void run() // updates displayArea
175            {
176               displayArea.append( messageToDisplay ); // append message
177            } // end method run
178         } // end anonymous inner class
179      ); // end call to SwingUtilities.invokeLater
180   } // end method displayMessage
181
```

```
182   // manipulates enterField in the event-dispatch thread
183   private void setTextFieldEditable( final boolean editable )
184   {
185      SwingUtilities.invokeLater(
186         new Runnable()
187         {
188            public void run() // sets enterField's editability
189            {
190               enterField.setEditable( editable );
191            } // end method run
192         }  // end inner class
193      ); // end call to SwingUtilities.invokeLater
194   } // end method setTextFieldEditable
195} // end class Server
```

```java
// Fig. 24.6: ServerTest.java
// Test the Server application.
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.runServer(); // run server application
   } // end main
} // end class ServerTest
```

Common Programming Error 24.1

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

Software Engineering Observation 24.5

When using an `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 only for networking applications that communicate using `ObjectOutputStream` and `ObjectInputStream`.

Performance Tip 24.3

A computer's input and output components are typically much slower than its memory. Output buffers typically are used to increase the efficiency of an application by sending larger amounts of data fewer times, thus reducing the number of times an application accesses the computer's input and output components.

```java
// Fig. 24.7: Client.java
// Client that reads and displays information sent from a Server.
import java.io.EOFException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
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 Client extends JFrame
{
   private JTextField enterField; // enters information from user
   private JTextArea displayArea; // display information to user
   private ObjectOutputStream output; // output stream to server
   private ObjectInputStream input; // input stream from server
   private String message = ""; // message from server
   private String chatServer; // host server for this application
   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 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 Client constructor

```

```
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
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
93
94   // get streams to send and recei
95   private void getStreams() throws
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
```

Use `InetAddress` static method `getByName` to obtain an `InetAdress` object containing the IP address spec

Display a message

Obtain `Socket`'s `OutputStream` and use it to initialize `ObjectOutputStream`

Method `flu` and sends header information

```
113     do // process messages sent from server
114     {
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
137        input.close(); // close input stream  1
138        client.close(); // close socket
139     } // end try
```

Read a `String` object from server

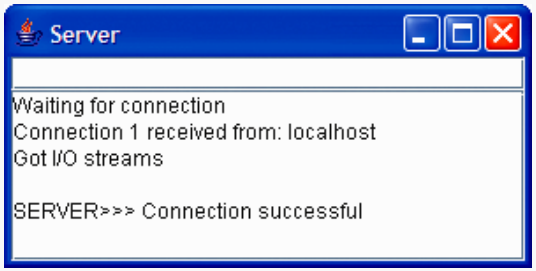
Invoke `Socket` method `close` to close the socket

```
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
160
```

Use `ObjectOutputStream` method `writeObject` to send a `String` to server

```
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         {
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
```

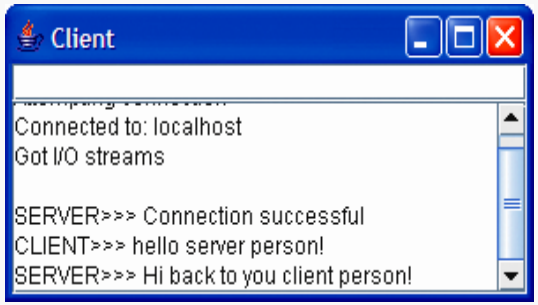
```java
// Fig. 24.8: ClientTest.java
// Test the Client class.
import javax.swing.JFrame;

public class ClientTest
{
   public static void main( String args[] )
   {
      Client application; // declare client application

      // if no command line args
      if ( args.length == 0 )
         application = new Client( "127.0.0.1" ); // connect to localhost
      else
         application = new Client( args[ 0 ] ); // use args to connect

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.runClient(); // run client application
   } // end main
} // end class ClientTest
```

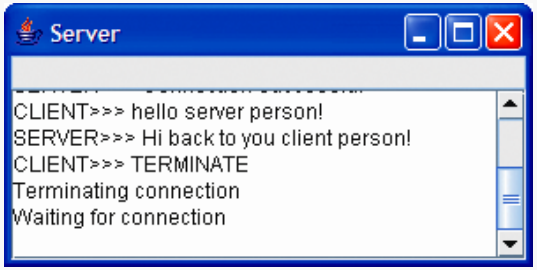


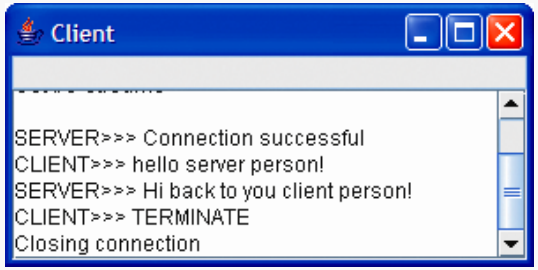

Server
Waiting for connection
Connection 1 received from: localhost
Got I/O streams
SERVER>>> Connection successful
CLIENT>>> hello server person!


Client
Attempting connection
Connected to: localhost
Got I/O streams
SERVER>>> Connection successful
CLIENT>>> hello server person!


Server
Connection 1 received from: localhost
Got I/O streams
SERVER>>> Connection successful
CLIENT>>> hello server person!
SERVER>>> Hi back to you client person!


Client
Connected to: localhost
Got I/O streams
SERVER>>> Connection successful
CLIENT>>> hello server person!
SERVER>>> Hi back to you client person!


Server
CLIENT>>> hello server person!
SERVER>>> Hi back to you client person!
CLIENT>>> TERMINATE
Terminating connection
Waiting for connection


Client
SERVER>>> Connection successful
CLIENT>>> hello server person!
SERVER>>> Hi back to you client person!
CLIENT>>> TERMINATE
Closing connection

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

- **Multiple threads**
  - **Server uses one thread per player**
    - **Allow each player to play game independently**

```java
// Fig. 24.13: TicTacToeServer.java
// This class maintains a game of Tic-Tac-Toe for two clients.
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.util.concurrent.Executors;
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;

```

```
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
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
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
```

Create array `players` with 2 elements

Create `ServerSocket` to listen on port 12345

```
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
89       gameLock.lock(); // lock game to signal player X's thread
90
```

Loop twice, blocking at line 79 each time while waiting for a client connection

Create a new `Player` object to manage the connection as eparate thread

Execute the Player in the `runGame` thread pool

```
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
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
```

```
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
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
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
181   // private inner class Player manages each Player as a runnable
182   private class Player implements Runnable
183   {
184      private Socket connection; // connection to client
185      private Scanner input; // input from client
186      private Formatter output; // output to client
187      private int playerNumber; // tracks which player this is
188      private String mark; // mark for this player
189      private boolean suspended = true; // whether thread is suspended
190
191      // set up Player thread
192      public Player( Socket socket, int number )
193      {
194         playerNumber = number; // store this player's number
195         mark = MARKS[ playerNumber ]; // specify player's mark
196         connection = socket; // store socket for client
197
198         try // obtain streams from Socket
199         {
200            input = new Scanner( connection.getInputStream() );
201            output = new Formatter( connection.getOutputStream() );
202         } // end try
```

Get the streams to send and receive data

```
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
215         output.flush(); // flush output
216      } // end method otherPlayerMoved
217
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
```

Format output notifying the [illegible] of the move

Call `Formatter` method `flush` to force the output to the client

Send player's mark

```
228         // if player X, wait for another player to arrive
229         if ( playerNumber == 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 O
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
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
```

Send message indicating one player connected and waiting for another player to arrive

Begin the game

```
257         else
258         {
259            output.format( "Player O 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
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
```

Send message indicating player O connected

Read a move

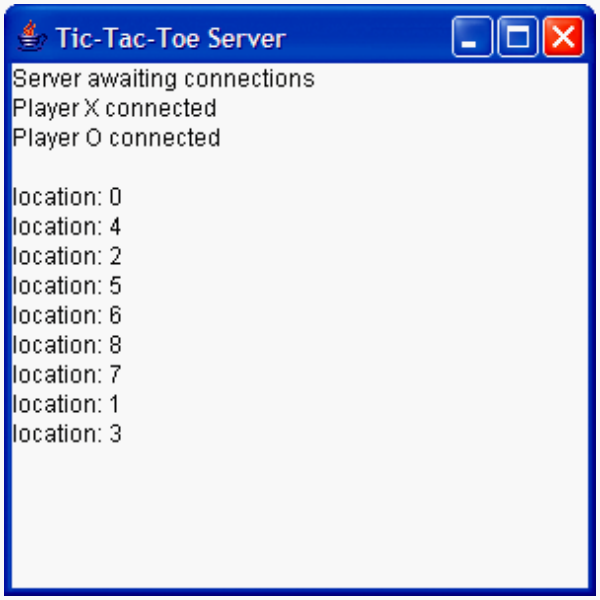
Check the move

Send message indicating the move is valid

```
            else // move was invalid
            {
               output.format( "Invalid move, try again\n" );
               output.flush(); // flush output
            } // end else
         } // end while
      } // end try
      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
```

Send message indicating the move is invalid

```java
// Fig. 24.14: TicTacToeServerTest.java
// Tests the TicTacToeServer.
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
```



```java
// Fig. 24.15: TicTacToeClient.java
// Client that let a user play Tic-Tac-Toe with another across a network.
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.GridLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.net.Socket;
import java.net.InetAddress;
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;
import java.util.Formatter;
import java.util.Scanner;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

```

```
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
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
64
65      idField = new JTextField(); // set up textfield
66      idField.setEditable( false );
67      add( idField, BorderLayout.NORTH );
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
```

```
79    // start the client thread
80    public void startClient()
81    {
82       try // connect to server, get streams and start outputThrea
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
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
```

Connect to the server

Get the streams to send and receive data

Read mark character from server

```
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
127
128   // process messages received by client
129   private void processMessage( String message )
130   {
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
```

Loop continually

Read and process messages from server

If valid move, write message and set mark in square

```
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
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
```

If opponent moves, set mark in square

```
158   // manipulate outputArea 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
172   // utility method to set mark on board in event-dispatch thread
173   private void setMark( final Square squareToMark, final String mark )
174   {
175      SwingUtilities.invokeLater(
176         new Runnable()
177         {
178            public void run()
179            {
180               squareToMark.setMark( mark ); // set mark in square
181            } // end method run
182         } // end anonymous inner class
183      ); // end call to SwingUtilities.invokeLater
184   } // end method setMark
185
```

```
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 anymore
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
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
```

Send the move to the server

```
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
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
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
```

```java
// Fig. 24.16: TicTacToeClientTest.java
// Tests the TicTacToeClient class.
import javax.swing.JFrame;

public class TicTacToeClientTest
{
   public static void main( String args[] )
   {
      TicTacToeClient application; // declare client application

      // if no command line args
      if ( args.length == 0 )
         application = new TicTacToeClient( "127.0.0.1" ); // localhost
      else
         application = new TicTacToeClient( args[ 0 ] ); // use args

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   } // end main
} // end class TicTacToeClientTest
```

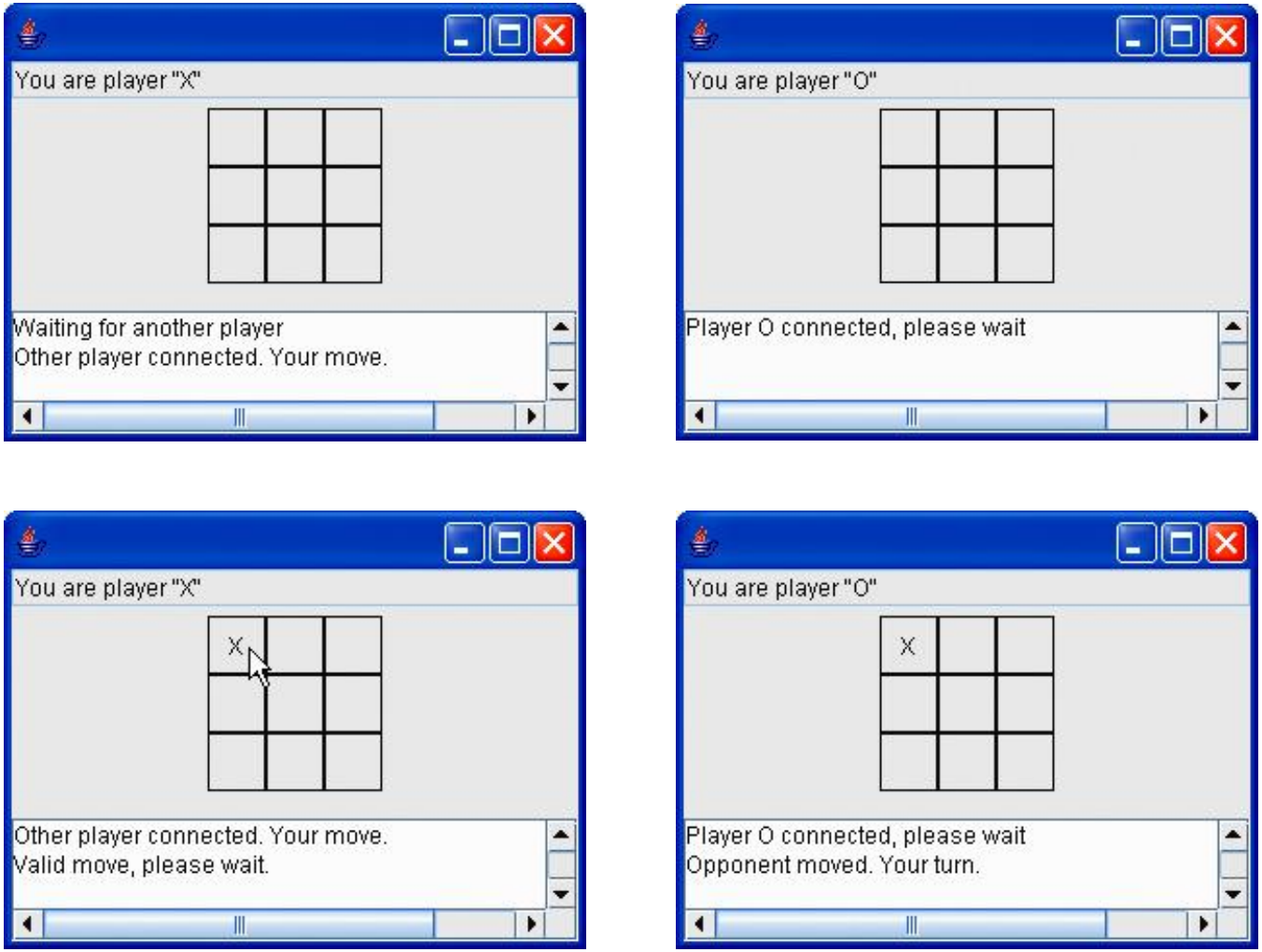


Fig.24.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 1 of 2.)

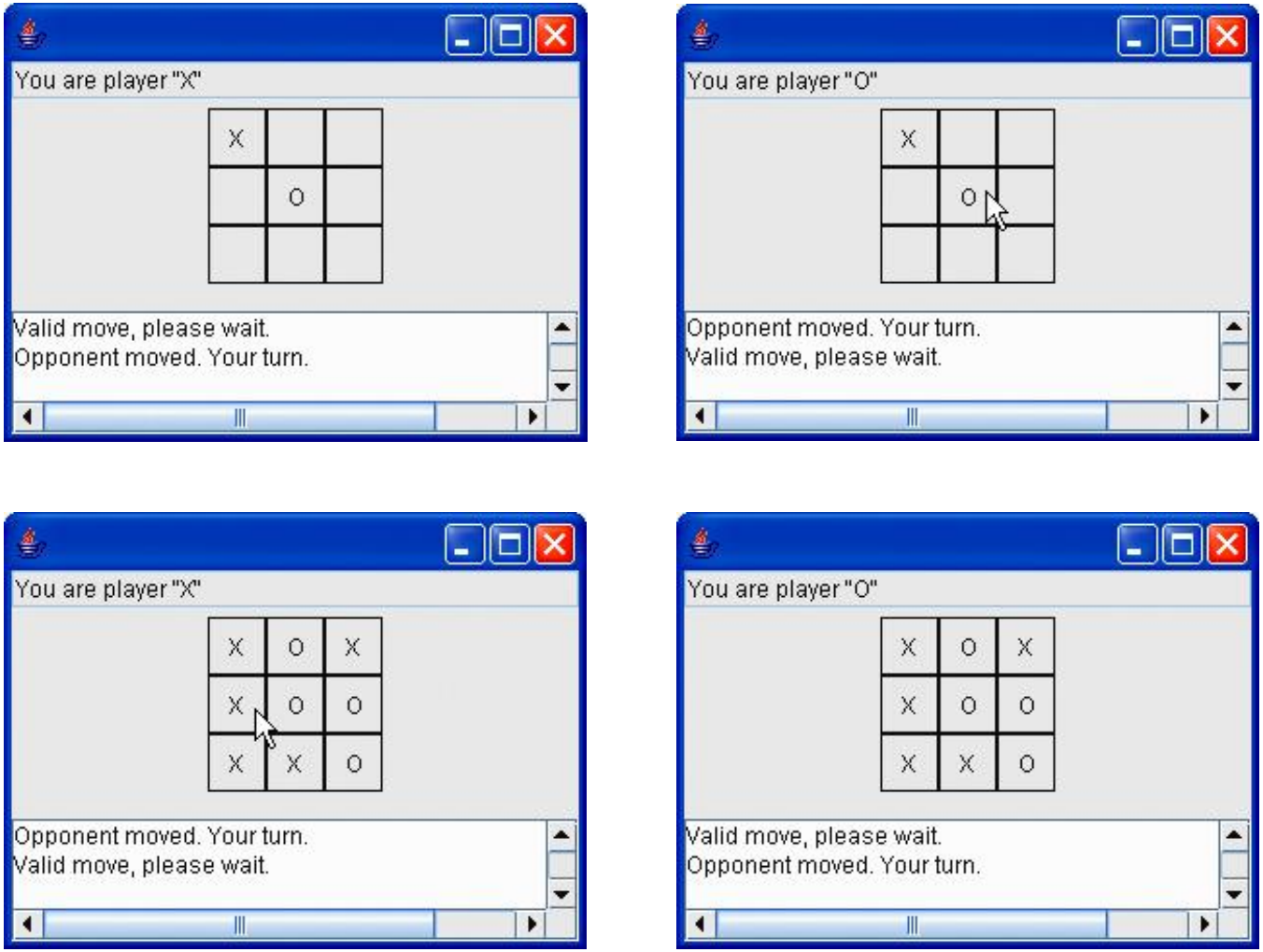


Fig.24.17 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 2 of 2.)

24.9 Security and the Network

- **By default, applets cannot perform file processing**
- **Applets often limited in machine access**
  - **Applets can communicate only with the machine from which it was originally downloaded**
- **Java Security API**
  - **Digitally signed applets**
  - **Applets given more privileges if from trusted source**