```
1  <APPLET CODE= "SiteSelector.class" WIDTH=300 HEIGHT=75>
2  <PARAM NAME="title0" VALUE="Java Home Page">
3  <PARAM NAME="location0" VALUE="http://java.sun.com/">
4  <PARAM NAME="title1" VALUE="Deitel">
5  <PARAM NAME="location1" VALUE="http://www.deitel.com/">
6  <PARAM NAME="title2" VALUE="Gamelan">
7  <PARAM NAME="location2" VALUE="http://www.gamelan.com/">
8  <PARAM NAME="title3" VALUE="JavaWorld">
9  <PARAM NAME="location3" VALUE="http://www.javaworld.com/">
10 </APPLET>
```

Fig. 21.1 Loading a document from a URL into a browser (part 1 of 3).

```
11  // Fig. 21.1: SiteSelector.java
12  // This program uses a button to load a document from a URL.
13  import java.net.*;
14  import java.util.*;
15  import javax.swing.*;
16  import javax.swing.event.*;
17  import java.awt.*;
18  import java.applet.AppletContext;
19
20  public class SiteSelector extends JApplet {
21     private Hashtable sites;
22     private Vector siteNames;
23
24     public void init()
25     {
26        sites = new Hashtable();
27        siteNames = new Vector();
28
29        getSitesFromHTMLParameters();
30
31        Container c = getContentPane();
32        c.add( new JLabel( "Choose a site to browse" ),
33               BorderLayout.NORTH );
34
35        final JList siteChooser = new JList( siteNames );
36        siteChooser.addListSelectionListener(
37           new ListSelectionListener() {
38              public void valueChanged( ListSelectionEvent e )
39              {
40                 Object o = siteChooser.getSelectedValue();
41                 URL newDocument = (URL) sites.get( o );
42                 AppletContext browser = getAppletContext();
43                 browser.showDocument( newDocument );
44              }
45           }
46        );
47        c.add( new JScrollPane( siteChooser ),
48               BorderLayout.CENTER );
49     }
50
51     private void getSitesFromHTMLParameters()
52     {
53        // look for applet parameters in the HTML document
54        // and add sites to Hashtable
55        String title, location;
56        URL url;
57        int counter = 0;
58
59        while ( true ) {
60           title = getParameter( "title" + counter );
61
62           if ( title != null ) {
63              location = getParameter( "location" + counter );
```

Fig. 21.1 Loading a document from a URL into a browser (part 2 of 3).

```
64
65             try {
66                url = new URL( location );
67                sites.put( title, url );
68                siteNames.addElement( title );
69             }
70             catch ( MalformedURLException e ) {
71                e.printStackTrace();
72             }
73          }
74          else
75             break;
76
77          ++counter;
78       }
79    }
80 }
```

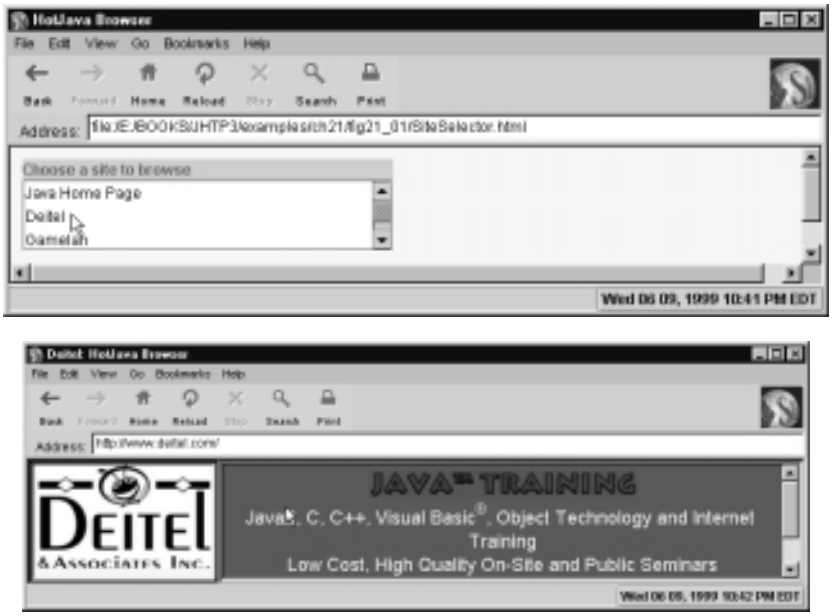


Fig. 21.1 Loading a document from a URL into a browser (part 3 of 3).

```
1  // Fig. 21.2: ReadServerFile.java
2  // This program uses a JEditorPane to display the
3  // contents of a file on a Web server.
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.net.*;
7  import java.io.*;
8  import javax.swing.*;
9  import javax.swing.event.*;
10
11 public class ReadServerFile extends JFrame {
12    private JTextField enter;
13    private JEditorPane contents;
14
15    public ReadServerFile()
16    {
17       super( "Simple Web Browser" );
18
19       Container c = getContentPane();
20
21       enter = new JTextField( "Enter file URL here" );
22       enter.addActionListener(
23          new ActionListener() {
24             public void actionPerformed( ActionEvent e )
25             {
26                getThePage( e.getActionCommand() );
27             }
28          }
29       );
30       c.add( enter, BorderLayout.NORTH );
31
32       contents = new JEditorPane();
33       contents.setEditable( false );
34       contents.addHyperlinkListener(
35          new HyperlinkListener() {
36             public void hyperlinkUpdate( HyperlinkEvent e )
37             {
38                if ( e.getEventType() ==
39                     HyperlinkEvent.EventType.ACTIVATED )
40                   getThePage( e.getURL().toString() );
41             }
42          }
43       );
```

Fig. 21.2 Reading a file through a URL connection (part 1 of 3).

```
44          c.add( new JScrollPane( contents ),
45                 BorderLayout.CENTER );
46
47          setSize( 400, 300 );
48          show();
49       }
50
51       private void getThePage( String location )
52       {
53          setCursor( Cursor.getPredefinedCursor(
54                        Cursor.WAIT_CURSOR ) );
55
56          try {
57             contents.setPage( location );
58             enter.setText( location );
59          }
60          catch ( IOException io ) {
61             JOptionPane.showMessageDialog( this,
62                "Error retrieving specified URL",
63                "Bad URL",
64                JOptionPane.ERROR_MESSAGE );
65          }
66
67          setCursor( Cursor.getPredefinedCursor(
68                        Cursor.DEFAULT_CURSOR ) );
69       }
70
71       public static void main( String args[] )
72       {
73          ReadServerFile app = new ReadServerFile();
74
75          app.addWindowListener(
76             new WindowAdapter() {
77                public void windowClosing( WindowEvent e )
78                {
79                   System.exit( 0 );
80                }
81             }
82          );
83       }
84    }
```

Fig. 21.2 Reading a file through a URL connection (part 2 of 3).

Fig. 21.2 Reading a file through a URL connection (part 3 of 3).

```
1  // Fig. 21.3: Server.java
2  // Set up a Server that will receive a connection
3  // from a client, send a string to the client,
4  // and close the connection.
5  import java.io.*;
6  import java.net.*;
7  import java.awt.*;
8  import java.awt.event.*;
9  import javax.swing.*;
10
11 public class Server extends JFrame {
12    private JTextField enter;
13    private JTextArea display;
14    ObjectOutputStream output;
15    ObjectInputStream input;
16
17    public Server()
18    {
19       super( "Server" );
20
21       Container c = getContentPane();
22
23       enter = new JTextField();
24       enter.setEnabled( false );
25       enter.addActionListener(
26          new ActionListener() {
27             public void actionPerformed( ActionEvent e )
28             {
29                sendData( e.getActionCommand() );
30             }
31          }
32       );
33       c.add( enter, BorderLayout.NORTH );
34
35       display = new JTextArea();
36       c.add( new JScrollPane( display ),
37              BorderLayout.CENTER );
38
39       setSize( 300, 150 );
40       show();
41    }
```

Fig. 21.3 Server portion of a client/server stream socket connection (part 1 of 3).

```
42
43    public void runServer()
44    {
45       ServerSocket server;
46       Socket connection;
47       int counter = 1;
48
49       try {
50          // Step 1: Create a ServerSocket.
51          server = new ServerSocket( 5000, 100 );
52
53          while ( true ) {
54             // Step 2: Wait for a connection.
55             display.setText( "Waiting for connection\n" );
56             connection = server.accept();
57
58             display.append( "Connection " + counter +
59                " received from: " +
60                connection.getInetAddress().getHostName() );
61
62             // Step 3: Get input and output streams.
63             output = new ObjectOutputStream(
64                         connection.getOutputStream() );
65             output.flush();
66             input = new ObjectInputStream(
67                        connection.getInputStream() );
68             display.append( "\nGot I/O streams\n" );
69
70             // Step 4: Process connection.
71             String message =
72                "SERVER>>> Connection successful";
73             output.writeObject( message );
74             output.flush();
75             enter.setEnabled( true );
76
77             do {
78                try {
79                   message = (String) input.readObject();
80                   display.append( "\n" + message );
81                   display.setCaretPosition(
82                      display.getText().length() );
83                }
84                catch ( ClassNotFoundException cnfex ) {
85                   display.append(
86                      "\nUnknown object type received" );
87                }
88             } while ( !message.equals( "CLIENT>>> TERMINATE" ) );
89
90             // Step 5: Close connection.
91             display.append( "\nUser terminated connection" );
92             enter.setEnabled( false );
93             output.close();
94             input.close();
```

Fig. 21.3 Server portion of a client/server stream socket connection (part 2 of 3).

```
95             connection.close();
96
97             ++counter;
98          }
99       }
100      catch ( EOFException eof ) {
101         System.out.println( "Client terminated connection" );
102      }
103      catch ( IOException io ) {
104         io.printStackTrace();
105      }
106   }
107
108   private void sendData( String s )
109   {
110      try {
111         output.writeObject( "SERVER>>> " + s );
112         output.flush();
113         display.append( "\nSERVER>>>" + s );
114      }
115      catch ( IOException cnfex ) {
116         display.append(
117            "\nError writing object" );
118      }
119   }
120
121   public static void main( String args[] )
122   {
123      Server app = new Server();
124
125      app.addWindowListener(
126         new WindowAdapter() {
127            public void windowClosing( WindowEvent e )
128            {
129               System.exit( 0 );
130            }
131         }
132      );
133
134      app.runServer();
135   }
136 }
```

Fig. 21.3 Server portion of a client/server stream socket connection (part 3 of 3).

```
1  // Fig. 21.4: Client.java
2  // Set up a Client that will read information sent
3  // from a Server and display the information.
4  import java.io.*;
5  import java.net.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 public class Client extends JFrame {
11    private JTextField enter;
12    private JTextArea display;
13    ObjectOutputStream output;
14    ObjectInputStream input;
15    String message = "";
16
17    public Client()
18    {
19       super( "Client" );
20
21       Container c = getContentPane();
22
23       enter = new JTextField();
24       enter.setEnabled( false );
25       enter.addActionListener(
26          new ActionListener() {
27             public void actionPerformed( ActionEvent e )
28             {
29                sendData( e.getActionCommand() );
30             }
31          }
32       );
33       c.add( enter, BorderLayout.NORTH );
34
35       display = new JTextArea();
36       c.add( new JScrollPane( display ),
37              BorderLayout.CENTER );
38
39       setSize( 300, 150 );
40       show();
41    }
42
43    public void runClient()
44    {
45       Socket client;
46
47       try {
48          // Step 1: Create a Socket to make connection.
49          display.setText( "Attempting connection\n" );
50          client = new Socket(
51             InetAddress.getByName( "127.0.0.1" ), 5000 );
52
```

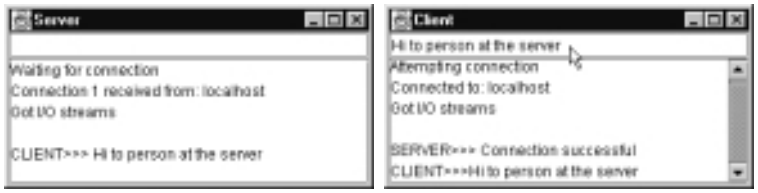
Fig. 21.4 Demonstrating the client portion of a stream socket connection between a client and a server (part 1 of 4).

```
53          display.append( "Connected to: " +
54             client.getInetAddress().getHostName() );
55
56          // Step 2: Get the input and output streams.
57          output = new ObjectOutputStream(
58                       client.getOutputStream() );
59          output.flush();
60          input = new ObjectInputStream(
61                      client.getInputStream() );
62          display.append( "\nGot I/O streams\n" );
63
64          // Step 3: Process connection.
65          enter.setEnabled( true );
66
67          do {
68             try {
69                message = (String) input.readObject();
70                display.append( "\n" + message );
71                display.setCaretPosition(
72                   display.getText().length() );
73             }
74             catch ( ClassNotFoundException cnfex ) {
75                display.append(
76                   "\nUnknown object type received" );
77             }
78          } while ( !message.equals( "SERVER>>> TERMINATE" ) );
79
80          // Step 4: Close connection.
81          display.append( "Closing connection.\n" );
82          input.close();
83          output.close();
84          client.close();
85       }
86       catch ( EOFException eof ) {
87          System.out.println( "Server terminated connection" );
88       }
89       catch ( IOException e ) {
90          e.printStackTrace();
91       }
92    }
93
94    private void sendData( String s )
95    {
96       try {
97          message = s;
98          output.writeObject( "CLIENT>>> " + s );
99          output.flush();
100         display.append( "\nCLIENT>>>" + s );
101      }
```

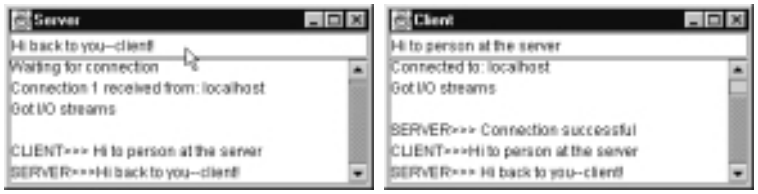
Fig. 21.4 Demonstrating the client portion of a stream socket connection between a client and a server (part 2 of 4).

```
102        catch ( IOException cnfex ) {
103           display.append(
104              "\nError writing object" );
105        }
106     }
107
108     public static void main( String args[] )
109     {
110        Client app = new Client();
111
112        app.addWindowListener(
113           new WindowAdapter() {
114              public void windowClosing( WindowEvent e )
115              {
116                 System.exit( 0 );
117              }
118           }
119        );
120
121        app.runClient();
122     }
123  }
```

The **Server** and **Client** windows after the **Client** connects to the **Server**

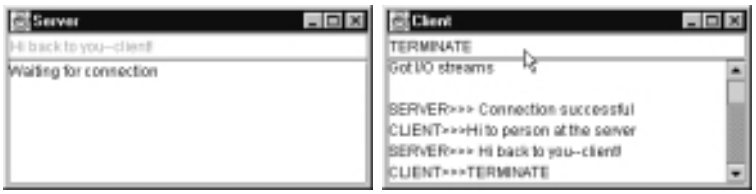


The **Server** and **Client** windows after the **Client** sends a message to the **Server**



The **Server** and **Client** windows after the **Server** sends a message to the **Client**

Fig. 21.4 Demonstrating the client portion of a stream socket connection between a client and a server (part 3 of 4).



The **`Server`** and **`Client`** windows after the **`Client`** terminates the connection

Fig. 21.4 Demonstrating the client portion of a stream socket connection between a client and a server (part 4 of 4).

```
1  // Fig. 21.5: Server.java
2  // Set up a Server that will receive packets from a
3  // client and send packets to a client.
4  import java.io.*;
5  import java.net.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 public class Server extends JFrame {
11    private JTextArea display;
12
13    private DatagramPacket sendPacket, receivePacket;
14    private DatagramSocket socket;
15
16    public Server()
17    {
18       super( "Server" );
19
20       display = new JTextArea();
21       getContentPane().add( new JScrollPane( display),
22                             BorderLayout.CENTER );
23       setSize( 400, 300 );
24       show();
25
26       try {
27          socket = new DatagramSocket( 5000 );
28       }
29       catch( SocketException se ) {
30          se.printStackTrace();
31          System.exit( 1 );
32       }
33    }
34
35    public void waitForPackets()
36    {
37       while ( true ) {
38          try {
39             // set up packet
40             byte data[] = new byte[ 100 ];
41             receivePacket =
42                new DatagramPacket( data, data.length );
43
44             // wait for packet
45             socket.receive( receivePacket );
46
47             // process packet
48             display.append( "\nPacket received:" +
49                "\nFrom host: " + receivePacket.getAddress() +
50                "\nHost port: " + receivePacket.getPort() +
51                "\nLength: " + receivePacket.getLength() +
52                "\nContaining:\n\t" +
```

Fig. 21.5 Demonstrating the server side of connectionless client/server computing with datagrams (part 1 of 2).

```
53                new String( receivePacket.getData(), 0,
54                            receivePacket.getLength() ) );
55
56             // echo information from packet back to client
57             display.append( "\n\nEcho data to client...");
58             sendPacket =
59                new DatagramPacket( receivePacket.getData(),
60                                    receivePacket.getLength(),
61                                    receivePacket.getAddress(),
62                                    receivePacket.getPort() );
63             socket.send( sendPacket );
64             display.append( "Packet sent\n" );
65             display.setCaretPosition(
66                display.getText().length() );
67          }
68          catch( IOException io ) {
69             display.append( io.toString() + "\n" );
70             io.printStackTrace();
71          }
72       }
73    }
74
75    public static void main( String args[] )
76    {
77       Server app = new Server();
78
79       app.addWindowListener(
80          new WindowAdapter() {
81             public void windowClosing( WindowEvent e )
82             {
83                System.exit( 0 );
84             }
85          }
86       );
87
88       app.waitForPackets();
89    }
90 }
```

The **Server** window after the client sends a packet of data

Fig. 21.5 Demonstrating the server side of connectionless client/server computing with datagrams (part 2 of 2).

```
1  // Fig. 21.6: Client.java
2  // Set up a Client that will send packets to a
3  // server and receive packets from a server.
4  import java.io.*;
5  import java.net.*;
6  import java.awt.*;
7  import java.awt.event.*;
8  import javax.swing.*;
9
10 public class Client extends JFrame implements ActionListener {
11    private JTextField enter;
12    private JTextArea display;
13
14    private DatagramPacket sendPacket, receivePacket;
15    private DatagramSocket socket;
16
17    public Client()
18    {
19       super( "Client" );
20
21       enter = new JTextField( "Type message here" );
22       enter.addActionListener( this );
23       getContentPane().add( enter, BorderLayout.NORTH );
24       display = new JTextArea();
25       getContentPane().add( new JScrollPane( display ),
26                             BorderLayout.CENTER );
27       setSize( 400, 300 );
28       show();
29
30       try {
31          socket = new DatagramSocket();
32       }
33       catch( SocketException se ) {
34          se.printStackTrace();
35          System.exit( 1 );
36       }
37    }
38
```

Fig. 21.6 Demonstrating the client side of connectionless client/server computing with datagrams (part 1 of 3).

```
39      public void waitForPackets()
40      {
41         while ( true ) {
42            try {
43               // set up packet
44               byte data[] = new byte[ 100 ];
45               receivePacket =
46                  new DatagramPacket( data, data.length );
47
48               // wait for packet
49               socket.receive( receivePacket );
50
51               // process packet
52               display.append( "\nPacket received:" +
53                  "\nFrom host: " + receivePacket.getAddress() +
54                  "\nHost port: " + receivePacket.getPort() +
55                  "\nLength: " + receivePacket.getLength() +
56                  "\nContaining:\n\t" +
57                  new String( receivePacket.getData(), 0,
58                              receivePacket.getLength() ) );
59               display.setCaretPosition(
60                  display.getText().length() );
61            }
62            catch( IOException exception ) {
63               display.append( exception.toString() + "\n" );
64               exception.printStackTrace();
65            }
66         }
67      }
68
69      public void actionPerformed( ActionEvent e )
70      {
71         try {
72            display.append( "\nSending packet containing: " +
73                            e.getActionCommand() + "\n" );
74
75            String s = e.getActionCommand();
76            byte data[] = s.getBytes();
77
78            sendPacket = new DatagramPacket( data, data.length,
79               InetAddress.getLocalHost(), 5000 );
80            socket.send( sendPacket );
81            display.append( "Packet sent\n" );
82            display.setCaretPosition(
83               display.getText().length() );
84
85         }
86         catch ( IOException exception ) {
87            display.append( exception.toString() + "\n" );
88            exception.printStackTrace();
89         }
90      }
```

Fig. 21.6 Demonstrating the client side of connectionless client/server computing with datagrams (part 2 of 3).

```
91
92      public static void main( String args[] )
93      {
94         Client app = new Client();
95
96         app.addWindowListener(
97            new WindowAdapter() {
98               public void windowClosing( WindowEvent e )
99               {
100                 System.exit( 0 );
101              }
102           }
103        );
104
105        app.waitForPackets();
106     }
107  }
```

The **Client** window after sending a packet to the server and receiving the packet back from the server

Fig. 21.6 Demonstrating the client side of connectionless client/server computing with datagrams (part 3 of 3).

```
1  // Fig. 21.7: TicTacToeServer.java
2  // This class maintains a game of Tic-Tac-Toe for two
3  // client applets.
4  import java.awt.*;
5  import java.awt.event.*;
6  import java.net.*;
7  import java.io.*;
8  import javax.swing.*;
9
10 public class TicTacToeServer extends JFrame {
11    private byte board[];
12    private boolean xMove;
13    private JTextArea output;
14    private Player players[];
15    private ServerSocket server;
16    private int currentPlayer;
17
18    public TicTacToeServer()
19    {
20       super( "Tic-Tac-Toe Server" );
21
22       board = new byte[ 9 ];
23       xMove = true;
24       players = new Player[ 2 ];
25       currentPlayer = 0;
26
27       // set up ServerSocket
28       try {
29          server = new ServerSocket( 5000, 2 );
30       }
```

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 1 of 6).

```
31          catch( IOException e ) {
32             e.printStackTrace();
33             System.exit( 1 );
34          }
35
36          output = new JTextArea();
37          getContentPane().add( output, BorderLayout.CENTER );
38          output.setText( "Server awaiting connections\n" );
39
40          setSize( 300, 300 );
41          show();
42       }
43
44       // wait for two connections so game can be played
45       public void execute()
46       {
47          for ( int i = 0; i < players.length; i++ ) {
48             try {
49                players[ i ] =
50                   new Player( server.accept(), this, i );
51                players[ i ].start();
52             }
53             catch( IOException e ) {
54                e.printStackTrace();
55                System.exit( 1 );
56             }
57          }
58
59          // Player X is suspended until Player O connects.
60          // Resume player X now.
61          synchronized ( players[ 0 ] ) {
62             players[ 0 ].threadSuspended = false;
63             players[ 0 ].notify();
64          }
65
66       }
67
68       public void display( String s )
69       {
70          output.append( s + "\n" );
71       }
72
73       // Determine if a move is valid.
74       // This method is synchronized because only one move can be
75       // made at a time.
76       public synchronized boolean validMove( int loc,
77                                              int player )
78       {
79          boolean moveDone = false;
80
```

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 2 of 6).

```
81          while ( player != currentPlayer ) {
82             try {
83                wait();
84             }
85             catch( InterruptedException e ) {
86                e.printStackTrace();
87             }
88          }
89
90          if ( !isOccupied( loc ) ) {
91             board[ loc ] =
92                (byte) ( currentPlayer == 0 ? 'X' : 'O' );
93             currentPlayer = ( currentPlayer + 1 ) % 2;
94             players[ currentPlayer ].otherPlayerMoved( loc );
95             notify();    // tell waiting player to continue
96             return true;
97          }
98          else
99             return false;
100      }
101
102      public boolean isOccupied( int loc )
103      {
104         if ( board[ loc ] == 'X' || board [ loc ] == 'O' )
105             return true;
106         else
107             return false;
108      }
109
110      public boolean gameOver()
111      {
112         // Place code here to test for a winner of the game
113         return false;
114      }
115
116      public static void main( String args[] )
117      {
118         TicTacToeServer game = new TicTacToeServer();
119
120         game.addWindowListener( new WindowAdapter() {
121           public void windowClosing( WindowEvent e )
122               {
123                  System.exit( 0 );
124               }
125            }
126         );
127
128         game.execute();
129      }
130   }
131
```

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 3 of 6).

```
132 // Player class to manage each Player as a thread
133 class Player extends Thread {
134    private Socket connection;
135    private DataInputStream input;
136    private DataOutputStream output;
137    private TicTacToeServer control;
138    private int number;
139    private char mark;
140    protected boolean threadSuspended = true;
141
142    public Player( Socket s, TicTacToeServer t, int num )
143    {
144       mark = ( num == 0 ? 'X' : 'O' );
145
146       connection = s;
147
148       try {
149          input = new DataInputStream(
150                     connection.getInputStream() );
151          output = new DataOutputStream(
152                     connection.getOutputStream() );
153       }
154       catch( IOException e ) {
155          e.printStackTrace();
156          System.exit( 1 );
157       }
158
159       control = t;
160       number = num;
161    }
162
163    public void otherPlayerMoved( int loc )
164    {
165       try {
166          output.writeUTF( "Opponent moved" );
167          output.writeInt( loc );
168       }
169       catch ( IOException e ) { e.printStackTrace(); }
170    }
171
172    public void run()
173    {
174       boolean done = false;
175
176       try {
177          control.display( "Player " +
178             ( number == 0 ? 'X' : 'O' ) + " connected" );
179          output.writeChar( mark );
180          output.writeUTF( "Player " +
181             ( number == 0 ? "X connected\n" :
182                             "O connected, please wait\n" ) );
183
```

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 4 of 6).

```
184          // wait for another player to arrive
185          if ( mark == 'X' ) {
186             output.writeUTF( "Waiting for another player" );
187
188             try {
189                synchronized( this ) {
190                   while ( threadSuspended )
191                      wait();
192                }
193             }
194             catch ( InterruptedException e ) {
195                e.printStackTrace();
196             }
197
198             output.writeUTF(
199                "Other player connected. Your move." );
200          }
201
202          // Play game
203          while ( !done ) {
204             int location = input.readInt();
205
206             if ( control.validMove( location, number ) ) {
207                control.display( "loc: " + location );
208                output.writeUTF( "Valid move." );
209             }
210             else
211                output.writeUTF( "Invalid move, try again" );
212
213             if ( control.gameOver() )
214                done = true;
215          }
216
217          connection.close();
218       }
219       catch( IOException e ) {
220          e.printStackTrace();
221          System.exit( 1 );
222       }
223    }
224 }
```

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 5 of 6).

Fig. 21.7 Server side of client/server Tic-Tac-Toe program (part 6 of 6).

```
1  // Fig. 21.8: TicTacToeClient.java
2  // Client for the TicTacToe program
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.net.*;
6  import java.io.*;
7  import javax.swing.*;
8
9  // Client class to let a user play Tic-Tac-Toe with
10 // another user across a network.
11 public class TicTacToeClient extends JApplet
12                            implements Runnable {
13    private JTextField id;
14    private JTextArea display;
15    private JPanel boardPanel, panel2;
16    private Square board[][], currentSquare;
17    private Socket connection;
18    private DataInputStream input;
19    private DataOutputStream output;
20    private Thread outputThread;
21    private char myMark;
22    private boolean myTurn;
23
24    // Set up user-interface and board
25    public void init()
26    {
27       display = new JTextArea( 4, 30 );
28       display.setEditable( false );
29       getContentPane().add( new JScrollPane( display ),
30                             BorderLayout.SOUTH );
31
32       boardPanel = new JPanel();
33       GridLayout layout = new GridLayout( 3, 3, 0, 0 );
34       boardPanel.setLayout( layout );
35
```

Fig. 21.8 Client side of client/server Tic-Tac-Toe program (part 1 of 5).

```
36          board = new Square[ 3 ][ 3 ];
37
38          // When creating a Square, the location argument to the
39          // constructor is a value from 0 to 8 indicating the
40          // position of the Square on the board. Values 0, 1,
41          // and 2 are the first row, values 3, 4, and 5 are the
42          // second row. Values 6, 7, and 8 are the third row.
43          for ( int row = 0; row < board.length; row++ )
44          {
45             for ( int col = 0;
46                      col < board[ row ].length; col++ ) {
47                board[ row ][ col ] =
48                   new Square( ' ', row * 3 + col );
49                board[ row ][ col ].addMouseListener(
50                   new SquareListener(
51                      this, board[ row ][ col ] ) );
52
53                boardPanel.add( board[ row ][ col ] );
54             }
55          }
56
57          id = new JTextField();
58          id.setEditable( false );
59
60          getContentPane().add( id, BorderLayout.NORTH );
61
62          panel2 = new JPanel();
63          panel2.add( boardPanel, BorderLayout.CENTER );
64          getContentPane().add( panel2, BorderLayout.CENTER );
65       }
66
67       // Make connection to server and get associated streams.
68       // Start separate thread to allow this applet to
69       // continually update its output in text area display.
70       public void start()
71       {
72          try {
73             connection = new Socket(
74                InetAddress.getByName( "127.0.0.1" ), 5000 );
75             input = new DataInputStream(
76                            connection.getInputStream() );
77             output = new DataOutputStream(
78                            connection.getOutputStream() );
79          }
80          catch ( IOException e ) {
81             e.printStackTrace();
82          }
83
84          outputThread = new Thread( this );
85          outputThread.start();
86       }
87
```

Fig. 21.8 Client side of client/server Tic-Tac-Toe program (part 2 of 5).

```
88     // Control thread that allows continuous update of the
89     // text area display.
90     public void run()
91     {
92        // First get player's mark (X or O)
93        try {
94           myMark = input.readChar();
95           id.setText( "You are player \"" + myMark + "\"" );
96           myTurn = ( myMark == 'X' ? true  : false );
97        }
98        catch ( IOException e ) {
99           e.printStackTrace();
100       }
101
102       // Receive messages sent to client
103       while ( true ) {
104          try {
105             String s = input.readUTF();
106             processMessage( s );
107          }
108          catch ( IOException e ) {
109             e.printStackTrace();
110          }
111       }
112    }
113
114    // Process messages sent to client
115    public void processMessage( String s )
116    {
117       if ( s.equals( "Valid move." ) ) {
118          display.append( "Valid move, please wait.\n" );
119          currentSquare.setMark( myMark );
120          currentSquare.repaint();
121       }
122       else if ( s.equals( "Invalid move, try again" ) ) {
123          display.append( s + "\n" );
124          myTurn = true;
125       }
126       else if ( s.equals( "Opponent moved" ) ) {
127          try {
128             int loc = input.readInt();
129
130             board[ loc / 3 ][ loc % 3 ].setMark(
131                   ( myMark == 'X' ? 'O' : 'X' ) );
132             board[ loc / 3 ][ loc % 3 ].repaint();
133
134             display.append(
135                "Opponent moved. Your turn.\n" );
136             myTurn = true;
137          }
```

Fig. 21.8 Client side of client/server Tic-Tac-Toe program (part 3 of 5).

```
138          catch ( IOException e ) {
139             e.printStackTrace();
140          }
141       }
142       else
143          display.append( s + "\n" );
144
145       display.setCaretPosition(
146          display.getText().length() );
147    }
148
149    public void sendClickedSquare( int loc )
150    {
151       if ( myTurn )
152          try {
153             output.writeInt( loc );
154             myTurn = false;
155          }
156          catch ( IOException ie ) {
157             ie.printStackTrace();
158          }
159    }
160
161    public void setCurrentSquare( Square s )
162    {
163       currentSquare = s;
164    }
165 }
166
167 // Maintains one square on the board
168 class Square extends JPanel {
169    private char mark;
170    private int location;
171
172    public Square( char m, int loc)
173    {
174       mark = m;
175       location = loc;
176       setSize ( 30, 30 );
177
178       setVisible(true);
179    }
180
181    public Dimension getPreferredSize() {
182       return ( new Dimension( 30, 30 ) );
183    }
184
185    public Dimension getMinimumSize() {
186       return ( getPreferredSize() );
187    }
188
189    public void setMark( char c ) { mark = c; }
```

Fig. 21.8 Client side of client/server Tic-Tac-Toe program (part 4 of 5).

```
190
191    public int getSquareLocation() { return location; }
192
193    public void paintComponent( Graphics g )
194    {
195       super.paintComponent( g );
196       g.drawRect( 0, 0, 29, 29 );
197       g.drawString( String.valueOf( mark ), 11, 20 );
198    }
199 }
200
201 class SquareListener extends MouseAdapter {
202    private TicTacToeClient applet;
203    private Square square;
204
205    public SquareListener( TicTacToeClient t, Square s )
206    {
207       applet = t;
208       square = s;
209    }
210
211    public void mouseReleased( MouseEvent e )
212    {
213       applet.setCurrentSquare( square );
214       applet.sendClickedSquare( square.getSquareLocation() );
215    }
216 }
```

Fig. 21.8 Client side of client/server Tic-Tac-Toe program (part 5 of 5).

Fig. 21.9 Sample outputs from the client/server Tic-Tac-Toe program (part 1 of 2).

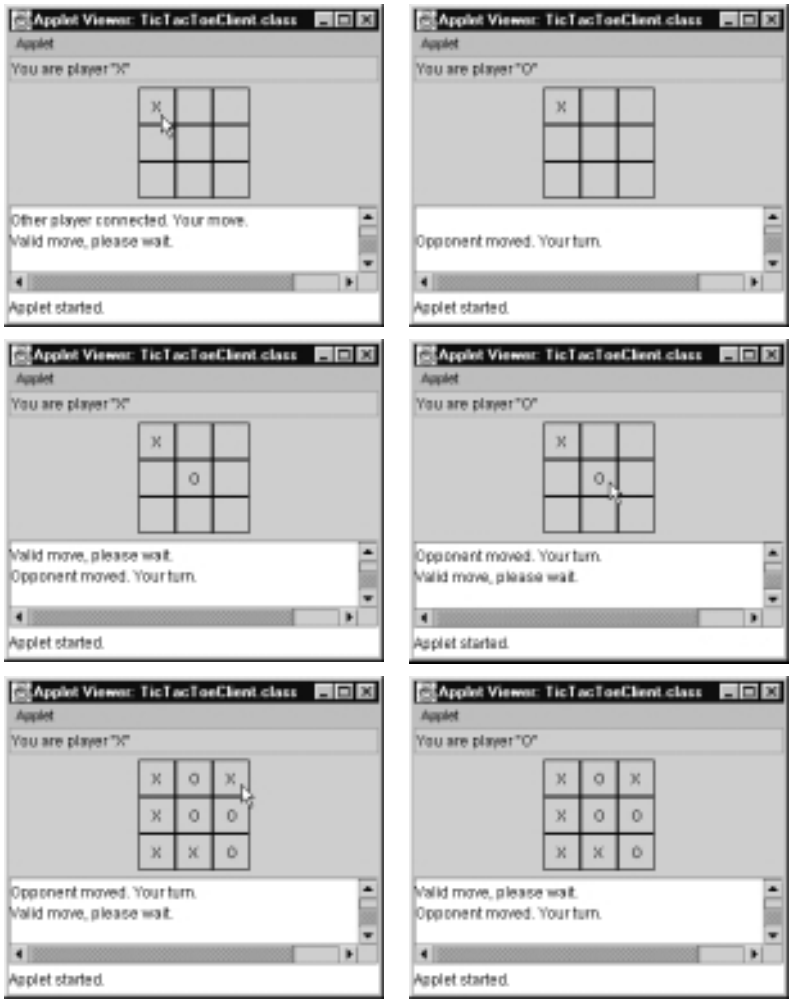


Fig. 21.9 Sample outputs from the client/server Tic-Tac-Toe program (part 2 of 2).