Java Socket Programming Examples

Although most programmers probably do network programming using a nice library with high-level application protocol (such as HTTP) support built-in, it's still useful to have an understanding of how to code at the socket level. Here are a few complete examples you can compile and run.

Overview

We will look at four network applications, written completely from scratch in Java. We will see that we can write these programs without any knowledge of the technologies under the hood (which include operating system resources, routing between networks, address lookup, physical transmission media, etc.)

Each of these applications use the *client-server* paradigm, which is roughly

1. One program, called the *server* blocks waiting for a client to connect to it
2. A client connects
3. The server and the client exchange information until they're done
4. The client and the server both close their connection

The only pieces of background information you need are:

- Hosts have *ports*, numbered from 0-65535. Servers listen on a port. Some port numbers are reserved so you can't use them when you write your own server.
- Multiple clients can be communicating with a server on a given port. Each client connection is assigned a separate *socket* on that port.

- Client applications get a port and a socket on the client machine when they connect successfully with a server.

The four applications are

- A trivial date server and client, illustrating simple one-way communication. The server sends data to the client only.
- A capitalize server and client, illustrating two-way communication. Since the dialog between the client and server can comprise an unbounded number of messages back and forth, the server is *threaded* to service multiple clients efficiently.
- A two-player tic tac toe game, illustrating a server that needs to keep track of the state of a game, and inform each client of it, so they can each update their own displays.
- A multi-user chat application, in which a server must broadcast messages to all of its clients.

A Date Server and Client

The server

DateServer.java

```
        edu.lmu.cs.networking;

       java.io.            ;
       java.io.            ;
       java.net.             ;
       java.net.       ;
       java.util.     ;

/**
 * A TCP server that runs on port 9090.  When a client connects, it
 * sends the client the current date and time, then closes the
 * connection with that client.  Arguably just about the simplest
 * server you can write.
 */
                        {

    /**
     * Runs the server.
     */
                      main(       [] args)                     {
                    listener =                  (9090);
            {
                  (     ) {
                      socket = listener.accept();
                  {
                                    =
                                       (socket.getOutputStream(),      );
```

```
                    .println(        ().toString());
                }         {
                    socket.close();
                }
            }
        }
                {
            listener.close();
        }
    }
}
```

The client

DateClient.java

```
        edu.lmu.cs.networking;

       java.io.               ;
       java.io.            ;
       java.io.                   ;
       java.net.       ;

       javax.swing.            ;

/**
 * Trivial client for the date server.
 */
                        {

    /**
     * Runs the client as an application.  First it displays a dialog
     * box asking for the IP address or hostname of a host running
     * the date server, then connects to it and displays the date that
     * it serves.
     */
                    main(      [] args)                    {
            serverAddress =             .showInputDialog(
          "Enter IP Address of a machine that is\n" +
          "running the date service on port 9090:");
            s =            (serverAddress, 9090);
                    input =
                           (                      (s.getInputStream()));
            answer = input.readLine();
                .showMessageDialog(     , answer);
          .     (0);
    }
}
```

You can also test the server with `telnet`.

A Capitalization Server and Client

The server

CapitalizeServer.java

```
        edu.lmu.cs.networking;

       java.io.               ;
       java.io.            ;
       java.io.                   ;
       java.io.            ;
       java.net.             ;
       java.net.       ;

/**
 * A server program which accepts requests from clients to
 * capitalize strings.  When clients connect, a new thread is
 * started to handle an interactive dialog in which the client
 * sends in a string and the server thread sends back the
 * capitalized version of the string.
 *
 * The program is runs in an infinite loop, so shutdown in platform
 * dependent.  If you ran it from a console window with the "java"
 * interpreter, Ctrl+C generally will shut it down.
 */
                               {

    /**
     * Application method to run the server runs in an infinite loop
     * listening on port 9898.  When a connection is requested, it
     * spawns a new thread to do the servicing and immediately returns
     * to listening.  The server keeps a unique client number for each
     * client that connects just to show interesting logging
     * messages.  It is certainly not necessary to do this.
     */
                        main(       [] args)                   {
              .    .println("The capitalization server is running.");
            clientNumber = 0;
                     listener =                  (9898);
            {
                  (     ) {
                                (listener.accept(), clientNumber++).start();
            }
        }         {
            listener.close();
        }
    }

    /**
     * A private thread to handle capitalization requests on a particular
     * socket.  The client terminates the dialogue by sending a single line
     * containing only a period.
     */
                                                        {
                        socket;
                    clientNumber;

                             (        socket,      clientNumber) {
                .socket = socket;
                .clientNumber = clientNumber;
            log("New connection with client# " + clientNumber + " at " + socket);
        }

        /**
         * Services this thread's client by first sending the
         * client a welcome message then repeatedly reading strings
         * and sending back the capitalized version of the string.
```

```
         */
                    run() {
                {

                // Decorate the streams so we can send characters
                // and not just bytes.  Ensure output is flushed
                // after every newline.
                                =                    (
                                              (socket.getInputStream()));
                                =                (socket.getOutputStream(),     );

                // Send a welcome message to the client.
                   .println("Hello, you are client #" + clientNumber + ".");
                   .println("Enter a line with only a period to quit\n");

                // Get messages from the client, line by line; return them
                // capitalized
                      (    ) {
                           input =   .readLine();
                       (input ==      || input.equals(".")) {
                             ;
                    }
                       .println(input.toUpperCase());
                }
            }       (             e) {
                log("Error handling client# " + clientNumber + ": " + e);
            }         {
                     {
                    socket.close();
                }       (             e) {
                    log("Couldn't close a socket, what's going on?");
                }
                log("Connection with client# " + clientNumber + " closed");
            }
        }

        /**
         * Logs a simple message.  In this case we just write the
         * message to the server applications standard output.
         */
                    log(       message) {
                .    .println(message);
        }
    }
}
```

The client

CapitalizeClient.java

```
        edu.lmu.cs.networking;

       java.awt.      .            ;
       java.awt.      .               ;
       java.io.               ;
       java.io.            ;
       java.io.                  ;
       java.io.            ;
       java.net.       ;

       javax.swing.       ;
       javax.swing.            ;
       javax.swing.            ;
```

```
        javax.swing.          ;
        javax.swing.           ;

/**
 * A simple Swing-based client for the capitalization server.
 * It has a main frame window with a text field for entering
 * strings and a textarea to see the results of capitalizing
 * them.
 */
                              {

                              ;
                            ;
                  frame =            ("Capitalize Client");
                      dataField =                (40);
                     messageArea =               (8, 60);

    /**
     * Constructs the client by laying out the GUI and registering a
     * listener with the textfield so that pressing Enter in the
     * listener sends the textfield contents to the server.
     */
                           () {

        // Layout GUI
        messageArea.setEditable(     );
        frame.getContentPane().add(dataField, "North");
        frame.getContentPane().add(                (messageArea), "Center");

        // Add Listeners
        dataField.addActionListener(                    () {
            /**
             * Responds to pressing the enter key in the textfield
             * by sending the contents of the text field to the
             * server and displaying the response from the server
             * in the text area.  If the response is "." we exit
             * the whole application, which closes all sockets,
             * streams and windows.
             */
                       actionPerformed(             e) {
                  .println(dataField.getText());
                         response;
                  {
                  response =    .readLine();
                     (response ==       || response.equals("")) {
                               .     (0);
                    }
                }        (             ex) {
                         response = "Error: " + ex;
                }
                messageArea.append(response + "\n");
                dataField.selectAll();
            }
        });
    }

    /**
     * Implements the connection logic by prompting the end user for
     * the server's IP address, connecting, setting up streams, and
     * consuming the welcome messages from the server.  The Capitalizer
     * protocol says that the server sends three lines of text to the
     * client immediately after establishing a connection.
     */
                connectToServer()                    {
```

```
        // Get the server address from a dialog box.
               serverAddress =             .showInputDialog(
            frame,
            "Enter IP Address of the Server:",
            "Welcome to the Capitalization Program",
                        .QUESTION_MESSAGE);

        // Make connection and initialize streams
               socket =            (serverAddress, 9898);
           =                    (
                                       (socket.getInputStream()));
            =                 (socket.getOutputStream(),     );

        // Consume the initial welcoming messages from the server
            (    i = 0; i < 3; i++) {
            messageArea.append(  .readLine() + "\n");
        }
    }

    /**
     * Runs the client application.
     */
                        main(      [] args)                  {
                          client =                      ();
        client.frame.setDefaultCloseOperation(      .EXIT_ON_CLOSE);
        client.frame.pack();
        client.frame.setVisible(    );
        client.connectToServer();
    }
}
```

A Two-Player Networked Tic-Tac-Toe Game

The server

TicTacToeServer.java

```
        edu.lmu.cs.networking;

       java.io.              ;
       java.io.           ;
       java.io.                 ;
       java.io.           ;
       java.net.            ;
       java.net.      ;

/**
 * A server for a network multi-player tic tac toe game.  Modified and
 * extended from the class presented in Deitel and Deitel "Java How to
 * Program" book.  I made a bunch of enhancements and rewrote large sections
 * of the code.  The main change is instead of passing *data* between the
 * client and server, I made a TTTP (tic tac toe protocol) which is totally
 * plain text, so you can test the game with Telnet (always a good idea.)
 * The strings that are sent in TTTP are:
```

```
 *
 *  Client -> Server           Server -> Client
 *  ----------------           ----------------
 *  MOVE <n>  (0 <= n <= 8)    WELCOME <char>  (char in {X, O})
 *  QUIT                       VALID_MOVE
 *                             OTHER_PLAYER_MOVED <n>
 *                             VICTORY
 *                             DEFEAT
 *                             TIE
 *                             MESSAGE <text>
 *
 * A second change is that it allows an unlimited number of pairs of
 * players to play.
 */
                            {

    /**
     * Runs the application. Pairs up clients that connect.
     */
                        main(      [] args)                  {
                       listener =                 (8901);
              .    .println("Tic Tac Toe Server is Running");
            {
                  (    ) {
                     game =          ();
                    .        playerX = game.           (listener.accept(), 'X');
                    .        playerO = game.           (listener.accept(), 'O');
                playerX.setOpponent(playerO);
                playerO.setOpponent(playerX);
                game.currentPlayer = playerX;
                playerX.start();
                playerO.start();
            }
        }         {
            listener.close();
        }
    }
}

/**
 * A two-player game.
 */
           {

    /**
     * A board has nine squares.  Each square is either unowned or
     * it is owned by a player.  So we use a simple array of player
     * references.  If null, the corresponding square is unowned,
     * otherwise the array cell stores a reference to the player that
     * owns it.
     */
                [] board = {
            ,      ,      ,
            ,      ,      ,
            ,      ,     };

    /**
     * The current player.
     */
           currentPlayer;

    /**
     * Returns whether the current state of the board is such that one
     * of the players is a winner.
     */
```

```
                hasWinner() {

        (board[0] !=      && board[0] == board[1] && board[0] == board[2])
      ||(board[3] !=      && board[3] == board[4] && board[3] == board[5])
      ||(board[6] !=      && board[6] == board[7] && board[6] == board[8])
      ||(board[0] !=      && board[0] == board[3] && board[0] == board[6])
      ||(board[1] !=      && board[1] == board[4] && board[1] == board[7])
      ||(board[2] !=      && board[2] == board[5] && board[2] == board[8])
      ||(board[0] !=      && board[0] == board[4] && board[0] == board[8])
      ||(board[2] !=      && board[2] == board[4] && board[2] == board[6]);
}

/**
 * Returns whether there are no more empty squares.
 */
                boardFilledUp() {
        (    i = 0; i < board.length; i++) {
           (board[i] ==     ) {
                        ;
        }
    }
                ;
}

/**
 * Called by the player threads when a player tries to make a
 * move.  This method checks to see if the move is legal: that
 * is, the player requesting the move must be the current player
 * and the square in which she is trying to move must not already
 * be occupied.  If the move is legal the game state is updated
 * (the square is set and the next player becomes current) and
 * the other player is notified of the move so it can update its
 * client.
 */
                             legalMove(    location,         player) {
       (player == currentPlayer && board[location] ==     ) {
        board[location] = currentPlayer;
        currentPlayer = currentPlayer.opponent;
        currentPlayer.otherPlayerMoved(location);
                    ;
    }
                ;
}

/**
 * The class for the helper threads in this multithreaded server
 * application.  A Player is identified by a character mark
 * which is either 'X' or 'O'.  For communication with the
 * client the player has a socket with its input and output
 * streams.  Since only text is being communicated we use a
 * reader and a writer.
 */
                             {
         mark;
           opponent;
           socket;
                    input;
                output;

    /**
     * Constructs a handler thread for a given socket and mark
     * initializes the stream fields, displays the first two
     * welcoming messages.
     */
                 (       socket,      mark) {
```

```
            .socket = socket;
            .mark = mark;
            {
            input =                    (
                                       (socket.getInputStream()));
            output =                  (socket.getOutputStream(),     );
            output.println("WELCOME " + mark);
            output.println("MESSAGE Waiting for opponent to connect");
        }       (             e) {
                  .    .println("Player died: " + e);
        }
    }

    /**
     * Accepts notification of who the opponent is.
     */
                setOpponent(       opponent) {
            .opponent = opponent;
    }

    /**
     * Handles the otherPlayerMoved message.
     */
                otherPlayerMoved(    location) {
        output.println("OPPONENT_MOVED " + location);
        output.println(
            hasWinner() ? "DEFEAT" : boardFilledUp() ? "TIE" : "");
    }

    /**
     * The run method of this thread.
     */
                run() {
            {
            // The thread is only started after everyone connects.
            output.println("MESSAGE All players connected");

            // Tell the first player that it is her turn.
               (mark == 'X') {
                output.println("MESSAGE Your move");
            }

            // Repeatedly get commands from the client and process them.
                  (    ) {
                       command = input.readLine();
                   (command.startsWith("MOVE")) {
                        location =         .parseInt(command.substring(5));
                       (legalMove(location,     )) {
                        output.println("VALID_MOVE");
                        output.println(hasWinner() ? "VICTORY"
                                     : boardFilledUp() ? "TIE"
                                     : "");
                    }      {
                        output.println("MESSAGE ?");
                    }
                }          (command.startsWith("QUIT")) {
                           ;
                }
            }
        }       (             e) {
                  .    .println("Player died: " + e);
        }         {
                {socket.close();}       (             e) {}
        }
    }
```

```
    }
}
```

The client

TicTacToeClient.java

```
        edu.lmu.cs.networking;

       java.awt.      ;
       java.awt.           ;
       java.awt.      .             ;
       java.awt.      .           ;
       java.io.                ;
       java.io.                   ;
       java.io.            ;
       java.net.       ;

       javax.swing.     ;
       javax.swing.          ;
       javax.swing.       ;
       javax.swing.       ;
       javax.swing.            ;
       javax.swing.       ;

/**
 * A client for the TicTacToe game, modified and extended from the
 * class presented in Deitel and Deitel "Java How to Program" book.
 * I made a bunch of enhancements and rewrote large sections of the
 * code.  In particular I created the TTTP (Tic Tac Toe Protocol)
 * which is entirely text based.  Here are the strings that are sent:
 *
 *  Client -> Server           Server -> Client
 *  ----------------           ----------------
 *  MOVE <n>  (0 <= n <= 8)    WELCOME <char>  (char in {X, O})
 *  QUIT                       VALID_MOVE
 *                             OTHER_PLAYER_MOVED <n>
 *                             VICTORY
 *                             DEFEAT
 *                             TIE
 *                             MESSAGE <text>
 *
 */
                               {

                   frame =           ("Tic Tac Toe");
                   messageLabel =           ("");
                      icon;
                      opponentIcon;

                  [] board =            [9];
                   currentSquare;

                       PORT = 8901;
                   socket;
                              ;
                             ;

    /**
     * Constructs the client by connecting to a server, laying out the
     * GUI and registering GUI listeners.
     */
                           (       serverAddress)                   {
```

```
        // Setup networking
        socket =            (serverAddress, PORT);
           =                    (                      (
            socket.getInputStream()));
            =                 (socket.getOutputStream(),     );

        // Layout GUI
        messageLabel.setBackground(     .lightGray);
        frame.getContentPane().add(messageLabel, "South");

               boardPanel =            ();
        boardPanel.setBackground(     .black);
        boardPanel.setLayout(               (3, 3, 2, 2));
            (    i = 0; i < board.length; i++) {
                    j = i;
        board[i] =            ();
        board[i].addMouseListener(                () {
                         mousePressed(           e) {
                 currentSquare = board[j];
                    .println("MOVE " + j);}});
        boardPanel.add(board[i]);
    }
    frame.getContentPane().add(boardPanel, "Center");
}

/**
 * The main thread of the client will listen for messages
 * from the server.  The first message will be a "WELCOME"
 * message in which we receive our mark.  Then we go into a
 * loop listening for "VALID_MOVE", "OPPONENT_MOVED", "VICTORY",
 * "DEFEAT", "TIE", "OPPONENT_QUIT or "MESSAGE" messages,
 * and handling each message appropriately.  The "VICTORY",
 * "DEFEAT" and "TIE" ask the user whether or not to play
 * another game.  If the answer is no, the loop is exited and
 * the server is sent a "QUIT" message.  If an OPPONENT_QUIT
 * message is recevied then the loop will exit and the server
 * will be sent a "QUIT" message also.
 */
            play()                   {
           response;
        {
        response =   .readLine();
           (response.startsWith("WELCOME")) {
                mark = response.charAt(8);
            icon =              (mark == 'X' ? "x.gif" : "o.gif");
            opponentIcon  =              (mark == 'X' ? "o.gif" : "x.gif");
            frame.setTitle("Tic Tac Toe - Player " + mark);
        }
              (    ) {
            response =   .readLine();
               (response.startsWith("VALID_MOVE")) {
                messageLabel.setText("Valid move, please wait");
                currentSquare.setIcon(icon);
                currentSquare.repaint();
            }         (response.startsWith("OPPONENT_MOVED")) {
                 loc =         .parseInt(response.substring(15));
                board[loc].setIcon(opponentIcon);
                board[loc].repaint();
                messageLabel.setText("Opponent moved, your turn");
            }         (response.startsWith("VICTORY")) {
                messageLabel.setText("You win");
                     ;
            }         (response.startsWith("DEFEAT")) {
                messageLabel.setText("You lose");
```

```
                    ;
                }         (response.startsWith("TIE")) {
                    messageLabel.setText("You tied");
                    ;
                }         (response.startsWith("MESSAGE")) {
                    messageLabel.setText(response.substring(8));
                }
            }
               .println("QUIT");
        }
             {
            socket.close();
        }
    }

                wantsToPlayAgain() {
            response =            .showConfirmDialog(frame,
            "Want to play again?",
            "Tic Tac Toe is Fun Fun Fun",
                        .YES_NO_OPTION);
        frame.dispose();
               response ==            .YES_OPTION;
    }

    /**
     * Graphical square in the client window.  Each square is
     * a white panel containing.  A client calls setIcon() to fill
     * it with an Icon, presumably an X or O.
     */
                                      {
            label =           ((    )    );

                  () {
            setBackground(     .white);
            add(label);
        }

                setIcon(     icon) {
            label.setIcon(icon);
        }
    }

    /**
     * Runs the client as an application.
     */
                    main(      [] args)                  {
          (    ) {
               serverAddress = (args.length == 0) ? "localhost" : args[1];
                         client =                     (serverAddress);
            client.frame.setDefaultCloseOperation(      .EXIT_ON_CLOSE);
            client.frame.setSize(240, 160);
            client.frame.setVisible(    );
            client.frame.setResizable(     );
            client.play();
               (!client.wantsToPlayAgain()) {
                  ;
            }
        }
    }
}
```

A Multi-User Chat Application

The server

ChatServer.java

```
        edu.lmu.cs.networking;

       java.io.               ;
       java.io.           ;
       java.io.                 ;
       java.io.           ;
       java.net.            ;
       java.net.      ;
       java.util.       ;

/**
 * A multithreaded chat room server.  When a client connects the
 * server requests a screen name by sending the client the
 * text "SUBMITNAME", and keeps requesting a name until
 * a unique one is received.  After a client submits a unique
 * name, the server acknowledges with "NAMEACCEPTED".  Then
 * all messages from that client will be broadcast to all other
 * clients that have submitted a unique screen name.  The
 * broadcast messages are prefixed with "MESSAGE ".
 *
 * Because this is just a teaching example to illustrate a simple
 * chat server, there are a few features that have been left out.
 * Two are very useful and belong in production code:
 *
 *     1. The protocol should be enhanced so that the client can
 *        send clean disconnect messages to the server.
 *
 *     2. The server should do some logging.
 */
                        {

    /**
     * The port that the server listens on.
     */
                            PORT = 9001;

    /**
     * The set of all names of clients in the chat room.  Maintained
     * so that we can check that new clients are not registering name
     * already in use.
     */
                        <       > names =             <       >();

    /**
     * The set of all the print writers for all the clients.  This
     * set is kept so we can easily broadcast messages.
     */
                        <            > writers =             <            >();

    /**
     * The appplication main method, which just listens on a port and
     * spawns handler threads.
     */
```

```
                    main(       [] args)                    {
            .    .println("The chat server is running.");
                    listener =                  (PORT);
        {
              (     ) {
                        (listener.accept()).start();
        }
    }         {
        listener.close();
    }
}

/**
 * A handler thread class.  Handlers are spawned from the listening
 * loop and are responsible for a dealing with a single client
 * and broadcasting its messages.
 */
                                              {
                    name;
                    socket;
                              ;
                            ;

    /**
     * Constructs a handler thread, squirreling away the socket.
     * All the interesting work is done in the run method.
     */
                 (       socket) {
            .socket = socket;
    }

    /**
     * Services this thread's client by repeatedly requesting a
     * screen name until a unique one has been submitted, then
     * acknowledges the name and registers the output stream for
     * the client in a global set, then repeatedly gets inputs and
     * broadcasts them.
     */
                run() {
           {

           // Create character streams for the socket.
              =                     (                       (
               socket.getInputStream()));
               =                 (socket.getOutputStream(),     );

           // Request a name from this client.  Keep requesting until
           // a name is submitted that is not already used.  Note that
           // checking for the existence of a name and adding the name
           // must be done while locking the set of names.
                 (    ) {
                  .println("SUBMITNAME");
               name =   .readLine();
                  (name ==     ) {
                         ;
               }
                             (names) {
                      (!names.contains(name)) {
                       names.add(name);
                            ;
                }
               }
           }

           // Now that a successful name has been chosen, add the
```

```
                // socket's print writer to the set of all writers so
                // this client can receive broadcast messages.
                   .println("NAMEACCEPTED");
                writers.add(   );

                // Accept messages from this client and broadcast them.
                // Ignore other clients that cannot be broadcasted to.
                      (    ) {
                           input =   .readLine();
                       (input ==     ) {
                              ;
                    }
                        (            writer : writers) {
                        writer.println("MESSAGE " + name + ": " + input);
                    }
                }
            }       (            e) {
                      .   .println(e);
            }         {
                // This client is going down!  Remove its name and its print
                // writer from the sets, and close its socket.
                   (name !=     ) {
                    names.remove(name);
                }
                   (    !=     ) {
                    writers.remove(   );
                }
                    {
                    socket.close();
                }       (            e) {
                }
            }
        }
    }
}
```

The client

ChatClient.java

```
        edu.lmu.cs.networking;

       java.awt.     .           ;
       java.awt.     .               ;
       java.io.              ;
       java.io.           ;
       java.io.                  ;
       java.io.           ;
       java.net.      ;

       javax.swing.      ;
       javax.swing.            ;
       javax.swing.            ;
       javax.swing.          ;
       javax.swing.           ;

/**
 * A simple Swing-based client for the chat server.  Graphically
 * it is a frame with a text field for entering messages and a
 * textarea to see the whole dialog.
 *
 * The client follows the Chat Protocol which is as follows.
 * When the server sends "SUBMITNAME" the client replies with the
```

```
 * desired screen name.  The server will keep sending "SUBMITNAME"
 * requests as long as the client submits screen names that are
 * already in use.  When the server sends a line beginning
 * with "NAMEACCEPTED" the client is now allowed to start
 * sending the server arbitrary strings to be broadcast to all
 * chatters connected to the server.  When the server sends a
 * line beginning with "MESSAGE " then all characters following
 * this string should be displayed in its message area.
 */
                        {

                      ;
                    ;
          frame =            ("Chatter");
              textField =               (40);
             messageArea =              (8, 40);

    /**
     * Constructs the client by laying out the GUI and registering a
     * listener with the textfield so that pressing Return in the
     * listener sends the textfield contents to the server.  Note
     * however that the textfield is initially NOT editable, and
     * only becomes editable AFTER the client receives the NAMEACCEPTED
     * message from the server.
     */
                    () {

        // Layout GUI
        textField.setEditable(     );
        messageArea.setEditable(     );
        frame.getContentPane().add(textField, "North");
        frame.getContentPane().add(               (messageArea), "Center");
        frame.pack();

        // Add Listeners
        textField.addActionListener(                  () {
            /**
             * Responds to pressing the enter key in the textfield by sending
             * the contents of the text field to the server.    Then clear
             * the text area in preparation for the next message.
             */
                        actionPerformed(            e) {
                  .println(textField.getText());
               textField.setText("");
            }
        });
    }

    /**
     * Prompt for and return the address of the server.
     */
                  getServerAddress() {
                         .showInputDialog(
            frame,
            "Enter IP Address of the Server:",
            "Welcome to the Chatter",
                        .QUESTION_MESSAGE);
    }

    /**
     * Prompt for and return the desired screen name.
     */
                  getName() {
                         .showInputDialog(
            frame,
```

```
            "Choose a screen name:",
            "Screen name selection",
                        .PLAIN_MESSAGE);
    }

    /**
     * Connects to the server then enters the processing loop.
     */
                 run()                    {

        // Make connection and initialize streams
                serverAddress = getServerAddress();
                socket =            (serverAddress, 9001);
           =                    (                      (
            socket.getInputStream()));
            =                (socket.getOutputStream(),     );

        // Process all messages from server, according to the protocol.
              (    ) {
                   line =   .readLine();
               (line.startsWith("SUBMITNAME")) {
                   .println(getName());
            }         (line.startsWith("NAMEACCEPTED")) {
                textField.setEditable(    );
            }         (line.startsWith("MESSAGE")) {
                messageArea.append(line.substring(8) + "\n");
            }
        }
    }

    /**
     * Runs the client as an application with a closeable frame.
     */
                       main(       [] args)                   {
                  client =                ();
        client.frame.setDefaultCloseOperation(       .EXIT_ON_CLOSE);
        client.frame.setVisible(    );
        client.run();
    }
}
```