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 theclient-server paradigm, which is roughly

1. One program, called the serverblocks 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 separatesocket 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

```
package edu.lmu.cs.networking;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
/**
 * 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.
 */
public class DateServer {
    /**
     * Runs the server.
     */
    public static void main(String[] args) throws IOException {
        ServerSocket listener = new ServerSocket(9090);
        try {
            while (true) {
                Socket socket = listener.accept();
                try {
                    PrintWriter out =
                        new PrintWriter(socket.getOutputStream(), true);
                    out.println(new Date().toString());
                } finally {
                    socket.close();
                }
            }
        }
        finally {
            listener.close();
        }
    }
}
```

The client

DateClient.java

```
package edu.lmu.cs.networking;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import javax.swing.JOptionPane;
/**
 * Trivial client for the date server.
 */
public class DateClient {
    /**
     * 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.
     */
    public static void main(String[] args) throws IOException {
        String serverAddress = JOptionPane.showInputDialog(
            "Enter IP Address of a machine that is\n" +
            "running the date service on port 9090:");
        Socket s = new Socket(serverAddress, 9090);
        BufferedReader input =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        String answer = input.readLine();
        JOptionPane.showMessageDialog(null, answer);
        System.exit(0);
    }
}
```

You can also test the server withtelnet.

A Capitalization Server and Client

The server

CapitalizeServer.java

```
package edu.lmu.cs.networking;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
/**
 * 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.
 */
public class CapitalizeServer {
    /**
     * 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.
     */
    public static void main(String[] args) throws Exception {
        System.out.println("The capitalization server is running.");
        int clientNumber = 0;
        ServerSocket listener = new ServerSocket(9898);
        try {
            while (true) {
                new Capitalizer(listener.accept(), clientNumber++).start();
            }
        } finally {
            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.
     */
    private static class Capitalizer extends Thread {
        private Socket socket;
        private int clientNumber;
        public Capitalizer(Socket socket, int clientNumber) {
            this.socket = socket;
            this.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.
         */
        public void run() {
            try {
                // Decorate the streams so we can send characters
                // and not just bytes.  Ensure output is flushed
                // after every newline.
                BufferedReader in = new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));
                PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
                // Send a welcome message to the client.
                out.println("Hello, you are client #" + clientNumber + ".");
                out.println("Enter a line with only a period to quit\n");
                // Get messages from the client, line by line; return them
                // capitalized
                while (true) {
                    String input = in.readLine();
                    if (input == null || input.equals(".")) {
                        break;
                    }
                    out.println(input.toUpperCase());
                }
            } catch (IOException e) {
                log("Error handling client# " + clientNumber + ": " + e);
            } finally {
                try {
                    socket.close();
                } catch (IOException 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.
         */
        private void log(String message) {
            System.out.println(message);
        }
    }
}
```

The client

CapitalizeClient.java

```
package edu.lmu.cs.networking;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import javax.swing.JFrame;
```

```
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
/**
 * 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.
 */
public class CapitalizeClient {
    private BufferedReader in;
    private PrintWriter out;
    private JFrame frame = new JFrame("Capitalize Client");
    private JTextField dataField = new JTextField(40);
    private JTextArea messageArea = new JTextArea(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.
     */
    public CapitalizeClient() {
        // Layout GUI
        messageArea.setEditable(false);
        frame.getContentPane().add(dataField, "North");
        frame.getContentPane().add(new JScrollPane(messageArea), "Center");
        // Add Listeners
        dataField.addActionListener(new ActionListener() {
            /**
             * 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.
             */
            public void actionPerformed(ActionEvent e) {
                out.println(dataField.getText());
                   String response;
                try {
                    response = in.readLine();
                    if (response == null || response.equals("")) {
                          System.exit(0);
                      }
                } catch (IOException 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.
     */
    public void connectToServer() throws IOException {
        // Get the server address from a dialog box.
        String serverAddress = JOptionPane.showInputDialog(
            frame,
            "Enter IP Address of the Server:",
```

```
            "Welcome to the Capitalization Program",
            JOptionPane.QUESTION_MESSAGE);
        // Make connection and initialize streams
        Socket socket = new Socket(serverAddress, 9898);
        in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        // Consume the initial welcoming messages from the server
        for (int i = 0; i < 3; i++) {
            messageArea.append(in.readLine() + "\n");
        }
    }
    /**
     * Runs the client application.
     */
    public static void main(String[] args) throws Exception {
        CapitalizeClient client = new CapitalizeClient();
        client.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        client.frame.pack();
        client.frame.setVisible(true);
        client.connectToServer();
    }
}
```

A Two-Player Networked Tic-Tac-Toe Game

The server

TicTacToeServer.java

```
package edu.lmu.cs.networking;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
/**
 * 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.
 */
public class TicTacToeServer {
    /**
     * Runs the application. Pairs up clients that connect.
```

```
     */
    public static void main(String[] args) throws Exception {
        ServerSocket listener = new ServerSocket(8901);
        System.out.println("Tic Tac Toe Server is Running");
        try {
            while (true) {
                Game game = new Game();
                Game.Player playerX = game.new Player(listener.accept(), 'X');
                Game.Player playerO = game.new Player(listener.accept(), 'O');
                playerX.setOpponent(playerO);
                playerO.setOpponent(playerX);
                game.currentPlayer = playerX;
                playerX.start();
                playerO.start();
            }
        } finally {
            listener.close();
        }
    }
}
/**
 * A two-player game.
 */
class 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.
     */
    private Player[] board = {
        null, null, null,
        null, null, null,
        null, null, null};
    /**
     * The current player.
     */
    Player currentPlayer;
    /**
     * Returns whether the current state of the board is such that one
     * of the players is a winner.
     */
    public boolean hasWinner() {
        return
            (board[0] != null && board[0] == board[1] && board[0] == board[2])
          ||(board[3] != null && board[3] == board[4] && board[3] == board[5])
          ||(board[6] != null && board[6] == board[7] && board[6] == board[8])
          ||(board[0] != null && board[0] == board[3] && board[0] == board[6])
          ||(board[1] != null && board[1] == board[4] && board[1] == board[7])
          ||(board[2] != null && board[2] == board[5] && board[2] == board[8])
          ||(board[0] != null && board[0] == board[4] && board[0] == board[8])
          ||(board[2] != null && board[2] == board[4] && board[2] == board[6]);
    }
    /**
     * Returns whether there are no more empty squares.
     */
    public boolean boardFilledUp() {
        for (int i = 0; i < board.length; i++) {
            if (board[i] == null) {
                return false;
            }
        }
        return true;
```

```
}
/**
 * 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.
 */
public synchronized boolean legalMove(int location, Player player) {
    if (player == currentPlayer && board[location] == null) {
        board[location] = currentPlayer;
        currentPlayer = currentPlayer.opponent;
        currentPlayer.otherPlayerMoved(location);
        return true;
    }
    return false;
}
/**
 * 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.
 */
class Player extends Thread {
    char mark;
    Player opponent;
    Socket socket;
    BufferedReader input;
    PrintWriter output;
    /**
     * Constructs a handler thread for a given socket and mark
     * initializes the stream fields, displays the first two
     * welcoming messages.
     */
    public Player(Socket socket, char mark) {
        this.socket = socket;
        this.mark = mark;
        try {
            input = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            output = new PrintWriter(socket.getOutputStream(), true);
            output.println("WELCOME " + mark);
            output.println("MESSAGE Waiting for opponent to connect");
        } catch (IOException e) {
            System.out.println("Player died: " + e);
        }
    }
    /**
     * Accepts notification of who the opponent is.
     */
    public void setOpponent(Player opponent) {
        this.opponent = opponent;
    }
    /**
     * Handles the otherPlayerMoved message.
     */
    public void otherPlayerMoved(int location) {
        output.println("OPPONENT_MOVED " + location);
        output.println(
```

```
                hasWinner() ? "DEFEAT" : boardFilledUp() ? "TIE" : "");
        }
        /**
         * The run method of this thread.
         */
        public void run() {
            try {
                // The thread is only started after everyone connects.
                output.println("MESSAGE All players connected");
                // Tell the first player that it is her turn.
                if (mark == 'X') {
                    output.println("MESSAGE Your move");
                }
                // Repeatedly get commands from the client and process them.
                while (true) {
                    String command = input.readLine();
                    if (command.startsWith("MOVE")) {
                        int location = Integer.parseInt(command.substring(5));
                        if (legalMove(location, this)) {
                            output.println("VALID_MOVE");
                            output.println(hasWinner() ? "VICTORY"
                                         : boardFilledUp() ? "TIE"
                                         : "");
                        } else {
                            output.println("MESSAGE ?");
                        }
                    } else if (command.startsWith("QUIT")) {
                        return;
                    }
                }
            } catch (IOException e) {
                System.out.println("Player died: " + e);
            } finally {
                try {socket.close();} catch (IOException e) {}
            }
        }
    }
}
```

The client

TicTacToeClient.java

```
package edu.lmu.cs.networking;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
/**
 * 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>
 *
 */
public class TicTacToeClient {
    private JFrame frame = new JFrame("Tic Tac Toe");
    private JLabel messageLabel = new JLabel("");
    private ImageIcon icon;
    private ImageIcon opponentIcon;
    private Square[] board = new Square[9];
    private Square currentSquare;
    private static int PORT = 8901;
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    /**
     * Constructs the client by connecting to a server, laying out the
     * GUI and registering GUI listeners.
     */
    public TicTacToeClient(String serverAddress) throws Exception {
        // Setup networking
        socket = new Socket(serverAddress, PORT);
        in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        // Layout GUI
        messageLabel.setBackground(Color.lightGray);
        frame.getContentPane().add(messageLabel, "South");
        JPanel boardPanel = new JPanel();
        boardPanel.setBackground(Color.black);
        boardPanel.setLayout(new GridLayout(3, 3, 2, 2));
        for (int i = 0; i < board.length; i++) {
            final int j = i;
            board[i] = new Square();
            board[i].addMouseListener(new MouseAdapter() {
                public void mousePressed(MouseEvent e) {
                    currentSquare = board[j];
                    out.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.
     */
```

```
public void play() throws Exception {
    String response;
    try {
        response = in.readLine();
        if (response.startsWith("WELCOME")) {
            char mark = response.charAt(8);
            icon = new ImageIcon(mark == 'X' ? "x.gif" : "o.gif");
            opponentIcon  = new ImageIcon(mark == 'X' ? "o.gif" : "x.gif");
            frame.setTitle("Tic Tac Toe - Player " + mark);
        }
        while (true) {
            response = in.readLine();
            if (response.startsWith("VALID_MOVE")) {
                messageLabel.setText("Valid move, please wait");
                currentSquare.setIcon(icon);
                currentSquare.repaint();
            } else if (response.startsWith("OPPONENT_MOVED")) {
                int loc = Integer.parseInt(response.substring(15));
                board[loc].setIcon(opponentIcon);
                board[loc].repaint();
                messageLabel.setText("Opponent moved, your turn");
            } else if (response.startsWith("VICTORY")) {
                messageLabel.setText("You win");
                break;
            } else if (response.startsWith("DEFEAT")) {
                messageLabel.setText("You lose");
                break;
            } else if (response.startsWith("TIE")) {
                messageLabel.setText("You tied");
                break;
            } else if (response.startsWith("MESSAGE")) {
                messageLabel.setText(response.substring(8));
            }
        }
        out.println("QUIT");
    }
    finally {
        socket.close();
    }
}
private boolean wantsToPlayAgain() {
    int response = JOptionPane.showConfirmDialog(frame,
        "Want to play again?",
        "Tic Tac Toe is Fun Fun Fun",
        JOptionPane.YES_NO_OPTION);
    frame.dispose();
    return response == JOptionPane.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.
 */
static class Square extends JPanel {
    JLabel label = new JLabel((Icon)null);
    public Square() {
        setBackground(Color.white);
        add(label);
    }
    public void setIcon(Icon icon) {
        label.setIcon(icon);
    }
}
/**
```

```
     * Runs the client as an application.
     */
    public static void main(String[] args) throws Exception {
        while (true) {
            String serverAddress = (args.length == 0) ? "localhost" : args[1];
            TicTacToeClient client = new TicTacToeClient(serverAddress);
            client.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            client.frame.setSize(240, 160);
            client.frame.setVisible(true);
            client.frame.setResizable(false);
            client.play();
            if (!client.wantsToPlayAgain()) {
                break;
            }
        }
    }
}
```

A Multi-User Chat Application

The server

ChatServer.java

```
package edu.lmu.cs.networking;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashSet;
/**
 * 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.
 */
public class ChatServer {
    /**
     * The port that the server listens on.
     */
    private static final int 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.
     */
    private static HashSet<String> names = new HashSet<String>();
```

```
/**
 * The set of all the print writers for all the clients.  This
 * set is kept so we can easily broadcast messages.
 */
private static HashSet<PrintWriter> writers = new HashSet<PrintWriter>();
/**
 * The appplication main method, which just listens on a port and
 * spawns handler threads.
 */
public static void main(String[] args) throws Exception {
    System.out.println("The chat server is running.");
    ServerSocket listener = new ServerSocket(PORT);
    try {
        while (true) {
            new Handler(listener.accept()).start();
        }
    } finally {
        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.
 */
private static class Handler extends Thread {
    private String name;
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    /**
     * Constructs a handler thread, squirreling away the socket.
     * All the interesting work is done in the run method.
     */
    public Handler(Socket socket) {
        this.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.
     */
    public void run() {
        try {
            // Create character streams for the socket.
            in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
            // 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.
            while (true) {
                out.println("SUBMITNAME");
                name = in.readLine();
                if (name == null) {
                    return;
                }
                synchronized (names) {
                    if (!names.contains(name)) {
                        names.add(name);
                        break;
```

```
                    }
                }
            }
            // 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.
            out.println("NAMEACCEPTED");
            writers.add(out);
            // Accept messages from this client and broadcast them.
            // Ignore other clients that cannot be broadcasted to.
            while (true) {
                String input = in.readLine();
                if (input == null) {
                    return;
                }
                for (PrintWriter writer : writers) {
                    writer.println("MESSAGE " + name + ": " + input);
                }
            }
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            // This client is going down!  Remove its name and its print
            // writer from the sets, and close its socket.
            if (name != null) {
                names.remove(name);
            }
            if (out != null) {
                writers.remove(out);
            }
            try {
                socket.close();
            } catch (IOException e) {
            }
        }
    }
}
```

The client

ChatClient.java

```
package edu.lmu.cs.networking;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
/**
 * 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.
 */
public class ChatClient {
    BufferedReader in;
    PrintWriter out;
    JFrame frame = new JFrame("Chatter");
    JTextField textField = new JTextField(40);
    JTextArea messageArea = new JTextArea(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.
     */
    public ChatClient() {
        // Layout GUI
        textField.setEditable(false);
        messageArea.setEditable(false);
        frame.getContentPane().add(textField, "North");
        frame.getContentPane().add(new JScrollPane(messageArea), "Center");
        frame.pack();
        // Add Listeners
        textField.addActionListener(new ActionListener() {
            /**
             * 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.
             */
            public void actionPerformed(ActionEvent e) {
                out.println(textField.getText());
                textField.setText("");
            }
        });
    }
    /**
     * Prompt for and return the address of the server.
     */
    private String getServerAddress() {
        return JOptionPane.showInputDialog(
            frame,
            "Enter IP Address of the Server:",
            "Welcome to the Chatter",
            JOptionPane.QUESTION_MESSAGE);
    }
    /**
     * Prompt for and return the desired screen name.
     */
    private String getName() {
        return JOptionPane.showInputDialog(
            frame,
            "Choose a screen name:",
            "Screen name selection",
            JOptionPane.PLAIN_MESSAGE);
    }
    /**
```

```
     * Connects to the server then enters the processing loop.
     */
    private void run() throws IOException {
        // Make connection and initialize streams
        String serverAddress = getServerAddress();
        Socket socket = new Socket(serverAddress, 9001);
        in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        // Process all messages from server, according to the protocol.
        while (true) {
            String line = in.readLine();
            if (line.startsWith("SUBMITNAME")) {
                out.println(getName());
            } else if (line.startsWith("NAMEACCEPTED")) {
                textField.setEditable(true);
            } else if (line.startsWith("MESSAGE")) {
                messageArea.append(line.substring(8) + "\n");
            }
        }
    }
    /**
     * Runs the client as an application with a closeable frame.
     */
    public static void main(String[] args) throws Exception {
        ChatClient client = new ChatClient();
        client.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        client.frame.setVisible(true);
        client.run();
    }
```