```java
package dismembery.client;

import java.net.*;
import java.io.*;
import java.util.*;

import dismembery.common.*;

/**
 * Client connects to server then process
 * client->server and server->client requests/messages.
 * <P>
 * The current implementation uses a CommandFactory to
 * map protocal strings to an object to handle further communication
 * with the server. Each message from the server is identified by
 * a unique string which clients use to determine how to process
 * the server's sent command.
 * <P>
 * See {@link CommandFactory CommandFactory} for details on
 * specifying Command objects for each protocol string. Each
 * string should have an associated Command object/class which
 * will be obtained from the CommandFactory.
 * <P>
 * Usage: create a client with a name, then connect. After
 * connecting, use methods indicated by the API for communicating
 * with the server. All server->client commands are handled
 * by commands obtained from the CommandFactory.
 * <P>
 * @author Owen Astrachan
 *
 */
public class Client implements IClient
{
    protected ObjectOutputStream myServerOut;   // stream to server
    protected ObjectInputStream  myServerIn;    // stream from server

    protected BufferedReader stdIn;             // standard input stream
    protected PrintWriter    stdOut;            // standard output stream

    protected String myName;
    protected Player myPlayer;

    protected static
        CommandFactory ourCommands = new CommandFactory("commands.properties");

    protected IClient myRealClient;

    public Client(String name)
    {
        myName = name;
        myRealClient = this;
    }

    public void setClient(IClient client)
    {
        myRealClient = client;
    }

    public void connect(String host, int port)
    {
        Socket sock = null;
        try{
            sock = new Socket(host,port);
        }
        catch(Exception e){
            Log.println("Server connect error "+e);
        }
        try{
            Log.println("server connection made");

            // must open input and output from client in
            // different order from server, or we can get
            // some kind of deadlock -- here we do output/then input
```



```java
            // server output
            myServerOut = new ObjectOutputStream(sock.getOutputStream());
            myServerIn = new ObjectInputStream(sock.getInputStream());

            stdIn = new BufferedReader(new InputStreamReader(System.in));
            stdOut = new PrintWriter(System.out,true);
        }
        catch (Exception e){
            Log.println("Error client side on connection "+e);
        }
    }

    public void move(int amount)
    {
        try{
            myServerOut.writeObject(Globals.MOVE);
            myServerOut.writeObject(new Integer(amount));
        }
        catch (IOException e) {
            Log.println("trouble joining " +myName+" "+ e);
        }
    }

    /**
     * Joins the game, success of join determined
     * by processing Globals.JOINSUCCESS command.
     */
    private void joinGame()
    {
        try{
            myServerOut.writeObject(Globals.JOIN);
            myServerOut.writeObject(myName);
        }
        catch (IOException e) {
            Log.println("trouble joining " +myName+" "+ e);
        }
    }

    public void doTick(long timeLeft)
    {
        System.out.println("time remaining "+timeLeft);
    }

    public void showPlayers(GameState state)
    {
        Iterator it = state.getPlayers();
        while (it.hasNext()){
            Player p = (Player) it.next();
            System.out.println(p + " " + state.getScore(p));
        }
    }

    public void showPlayers(ArrayList list)
    {
        System.out.println("\tplayers:");
        for(int k=0; k < list.size(); k++){
            System.out.println("\t\t"+list.get(k));
        }
    }

    public void setPlayer(Player p)
    {
        myPlayer = p;
        System.out.println("I am player: "+myPlayer);
    }

    private void sendQuit()
    {
        try {
            myServerOut.writeObject(Globals.QUIT);
            stdOut.println("client finished");
```

```java
            // we finished, close stream to server
            myServerOut.close();
        }
        catch (IOException e) {
            Log.println("trouble quitting " + e);
        }
    }

    private void sendGarbage(String s)
    {
        try {
            myServerOut.writeObject(s);
        }
        catch (IOException e) {
            Log.println("trouble quitting " + e);
        }
    }

    /**
     * Get messages from server as long as we can.
     *
     */
    public void run()
    {
        joinGame();

        while (true) {
            try{
                String s = (String) myServerIn.readObject();
                ICommand command = ourCommands.getCommand(s);
                if (command != null){
                    command.execute(myServerIn, myRealClient);
                }
            }
            catch (IOException e) {
                Log.println("IO trouble in client "+e);
                break;
            }
            catch (ClassNotFoundException cnfe) {
                Log.println("class/object problem " + cnfe);
                break;
            }
        }
    }

    public static void main(String args[])
    {
        String name = "fred";
        if (args.length > 0){
            name = args[0];
        }
        Client c = new Client(name);
        c.connect(Globals.HOST, Globals.PORT);
        c.run();
    }
}
```



```java
package dismembery.client;

import java.net.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.Font;
import java.awt.event.*;
import java.awt.Color;
import java.awt.Graphics;

import dismembery.common.*;

/**
 * Client connects to server then process
 * client->server and server->client requests/messages.
 * <P>
 * The current implementation uses a CommandFactory to
 * map protocal strings to an object to handle further communication
 * with the server. Each message from the server is identified by
 * a unique string which clients use to determine how to process
 * the server's sent command.
 * <P>
 * See {@link CommandFactory CommandFactory} for details on
 * specifying Command objects for each protocol string. Each
 * string should have an associated Command object/class which
 * will be obtained from the CommandFactory.
 * <P>
 * This is a GUIclient, created with a Client to which
 * this one forwards many requests just to show how
 * things could work. A real gui client would show things
 * graphically.

 * <P>
 * @author Owen Astrachan
 *
 */
public class ClientGui extends JFrame implements IClient
{
    private Client myClient;
    private JLabel myTimeLabel;
    private JLabel myNameLabel;
    private MemberyPanel myPanel;

    public ClientGui(String name)
    {
                myClient = new Client(name);
                setTitle("DISMembery");

                setContentPane(getPanel());
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                pack();
                setSize(300,300);
                setVisible(true);

                myClient.setClient(this);
                myClient.connect(Globals.HOST, Globals.PORT);
                myClient.run();
    }

    private JPanel getPanel()
    {
                JPanel panel = new JPanel(new BorderLayout());
                myPanel = new MemberyPanel();
                panel.add(myPanel, BorderLayout.CENTER);
                myTimeLabel = new JLabel("    ", SwingConstants.LEFT);
                myTimeLabel.setFont(new Font("SansSerif",Font.BOLD,24));
                myNameLabel = new JLabel("    ", SwingConstants.RIGHT);

                JPanel sub = new JPanel();
                sub.add(myTimeLabel);
```

```java
                sub.add(myNameLabel);

                panel.add(sub, BorderLayout.NORTH);
                JButton button = new JButton("move");
                button.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent e){
                            doMove();
                        }
                    });
                panel.add(button, BorderLayout.SOUTH);
                return panel;
        }

        public void doMove()
        {
                myClient.move(1);
        }

        public void setPlayer(Player p)
        {
                myClient.setPlayer(p);
                myNameLabel.setText(p.getName());
        }

        public void doTick(long timeLeft)
        {
                long secs = timeLeft / 1000;
                long decs = timeLeft % 1000;
                final String display = ""+secs+"."+decs;
                SwingUtilities.invokeLater(new Runnable(){
                        public void run(){
                            myTimeLabel.setText(display);
                        }
                    });
        }

        public void showPlayers(ArrayList list)
        {
                myClient.showPlayers(list);
        }

        public void showPlayers(GameState state)
        {
                myClient.showPlayers(state);
        }

        public static void main(String args[])
        {
                String name = "fred";
                if (args.length > 0){
                    name = args[0];
                }
                ClientGui c = new ClientGui(name);
        }

        /**
         * Place holder for a panel with functionality
         */
        class MemberyPanel extends JPanel
        {
                public MemberyPanel()
                {

                }
        }
    }
```



```java
package dismembery.client;


import java.io.ObjectInputStream;

/**
 * Interface for all client Commands. Client programs
 * execute commands by passing the stream from the server
 * to the client into the execute function. Clients pass
 * themselves as well (or a proxy).
 * <P>
 * @author Owen Astrachan
 */


public interface Command
{
    /**
     * Executes a command on the client, may read from
     * the server stream.
     * @param in is the stream from the server
     * @param client is the client executing this command
     */
    void execute(ObjectInputStream in, IClient client);
}
```

```java
package dismembery.client;

import java.io.IOException;
import java.util.Properties;
import java.util.HashMap;

/**
 * The command factory provides a point-of-entry for
 * client programs to obtain a Command corresponding
 * to a string. In the protocol of this client-server
 * architecture, each string from the server corresponds
 * to a command. The commands are specified in a properties
 * file which provides a map from string to command names.
 * <P>
 * When the client requests a command corresponding to
 * a protocol string (see {@link dismembery.common.Globals
 * dismembery.common.Globals}) a Command object corresponding
 * the string is returned. Only one command object for the string
 * is created, the first time such an object is requested.
 * Created objects are cached in a map and retrieved when
 * requested.
 * <P>
 * @author Owen Astrachan
 */

public class CommandFactory
{
    private Properties myProperties = new Properties();
    private HashMap myMap = new HashMap();
    private String myName;

    private void loadProperties()
    {
        try {
            myProperties.load(getClass().getResource(myName).openStream());
        }
        catch (IOException e) {
            System.err.println("could not load properties "+myName);
            e.printStackTrace();
        }
    }

    /**
     * Create a factory from the specified properties file.
     */
    public CommandFactory(String propertiesName)
    {
        myName = propertiesName;
        loadProperties();
    }

    /**
     * Get the command that corresponds to a specified protocol
     * string. Only one Command per protocol is created and then
     * cached.
     * @param name is the name of the protocol
     * @return the command corresponding to the protocol string
     */
    public ICommand getCommand (String name)
    {
        ICommand command = (ICommand) myMap.get(name);
        if (command == null){

            // make the command, store it, return it;

            String className = myProperties.getProperty(name);
            if (className != null) {
                try {
                    command = (ICommand) Class.forName(className).newInstance();
                }
                catch (Exception e) {
                    System.err.println("*** badly named class: " + className);
                }
                myMap.put(name,command);
            }
        }
        return command;
    }
}
```

```java
package dismembery.client;

/**
 * Interface for client objects. Methods
 * here are callbacks for Commands to use when
 * executing.
 * <P>
 * @author Owen Astrachan
 */

import dismembery.common.*;
import java.util.ArrayList;
import java.util.HashMap;

public interface IClient
{
    /**
     * Sets the player for this client.
     */

    public void setPlayer(Player p);

    /**
     * Shows a list of players (list has Player objects).
     */

    public void showPlayers(ArrayList list);
    /**
     * Shows game state.
     */

    public void showPlayers(GameState state);

    /**
     * Ticks a clock with timeLeft milliseconds.
     */

    public void doTick(long timeLeft);
}
```



```java
package dismembery.client;


import java.io.ObjectInputStream;

/**
 * Interface for all client Commands. Client programs
 * execute commands by passing the stream from the server
 * to the client into the execute function. Clients pass
 * themselves as well (or a proxy).
 * <P>
 * @author Owen Astrachan
 */


public interface ICommand
{
    /**
     * Executes a command on the client, may read from
     * the server stream.
     * @param in is the stream from the server
     * @param client is the client executing this command
     */
    void execute(ObjectInputStream in, IClient client);
}
```

```java
package dismembery.client;

import java.io.ObjectInputStream;
import dismembery.common.Player;

public class JoinSuccessCommand implements ICommand
{
    public void execute(ObjectInputStream in, IClient client)
    {
        try{
            Player player = (Player) in.readObject();
            client.setPlayer(player);
        }
        catch(Exception e){

        }
    }
}
```



```java
package dismembery.client;

import java.io.ObjectInputStream;
import dismembery.common.Player;
import dismembery.common.Log;

public class LeaveCommand implements ICommand
{
    public void execute(ObjectInputStream in, IClient client)
    {
            try{
                Player player = (Player) in.readObject();
                Log.println(" *** player left game "+player);
            }
            catch(Exception e){
                Log.println("error in LeaveCommand");
            }
    }
}
```

```java
package dismembery.client;

import java.io.ObjectInputStream;
import java.util.ArrayList;

public class ListCommand implements ICommand
{
    public void execute(ObjectInputStream in, IClient client)
    {
        try{
            ArrayList list = (ArrayList) in.readObject();
            client.showPlayers(list);
        }
        catch(Exception e){

        }
    }
}
```



```java
package dismembery.client;

import java.io.ObjectInputStream;
import dismembery.common.*;

/**
 * Executes on each clock tick.
 * Currently this commands reads entire GameState on each
 * tick and then gets info from this as needed to pass
 * to client.
 */
public class TickCommand implements ICommand
{
    public void execute(ObjectInputStream in, IClient client)
    {
        try{
            GameState state = (GameState) in.readObject();
            client.doTick(state.getTimeLeft());
            client.showPlayers(state);
        }
        catch(Exception e){
            System.out.println("tick trouble");
        }
    }
}
```

```java
package dismembery.common;

/**
 * Simple counter, starts at 0, can be
 * incremented and queried for value.
 * @author Owen Astrachan
 */
public class Counter implements java.io.Serializable
{
    private int myValue = 0;

    public Counter()
    {

    }
    public void incr()
    {
        incr(1);
    }
    public void incr(int amount)
    {
        myValue += amount;
    }
    public int value()
    {
        return myValue;
    }

    public String toString()
    {
        return ""+myValue;
    }
}
```



```java
package dismembery.common;

import java.util.*;

/**
 * The state of a game. Possible to send
 * this to all clients rather than individual
 * objects. Client can get information needed
 * since all info included here.
 * <P>
 * @author Owen Astrachan
 */

public class GameState implements java.io.Serializable
{
    private HashMap myMap;
    private long myTimeLeft;

    public GameState(HashMap scores, long timeLeft)
    {
        myMap = scores;
        myTimeLeft = timeLeft;
    }

    public int size()
    {
        return myMap.size();
    }

    public Iterator getPlayers()
    {
        return myMap.keySet().iterator();
    }

    public int getScore(Player p)
    {
        return ((Counter) myMap.get(p)).value();
    }

    public void setTimeLeft(long time)
    {
        myTimeLeft = time;
    }

    public long getTimeLeft()
    {
        return myTimeLeft;
    }

    public String toString()
    {
        return ""+getTimeLeft();
    }
}
```

Nov 16, 05 9:59 **Globals.java** Page 1/1

```java
package dismembery.common;

public class Globals
{
    public static final int PORT = 8188;
    public static final String HOST = "127.0.0.1";

    public static final String JOIN = "join";
    public static final String GET = "get";
    public static final String QUIT = "quit";
    public static final String PLAYERS = "players";
    public static final String TICK = "tick";
    public static final String LEAVE = "leave";
    public static final String JOINSUCCESS = "joinsuccess";
    public static final String MOVE = "move";
}
```

Nov 16, 05 9:59 **Log.java** Page 1/1

```java
package dismembery.common;

public class Log
{
    public static void print(String s)
    {
            System.out.print(s);
    }

    public static void println(String s)
    {
            System.out.println(s);
    }
}
```

```java
package dismembery.common;

/**
 * Currently stores name and id of player
 * so that name+id identifies uniquely a player.
 * No players can share the same id (guaranteed by
 * server) so the name+id string is unique per player.
 * <P>
 * This object can be put into HashMaps or TreeMaps
 * <P>
 * @author Owen Astrachan
 */
public class Player implements java.io.Serializable, Comparable
{
    private String myName;
    private String myId;
    private String myHash;

    public Player(String name, String id)
    {
            myName = name;
            myId = id;
            myHash = name+id;
    }

    public boolean equals(Object o)
    {
            Player p = (Player) o;
            return myHash.equals(p.myHash);
    }

    public int hashCode()
    {
            return myHash.hashCode();
    }

    public String toString()
    {
            return myHash;
    }

    public int compareTo(Object o)
    {
            Player p = (Player) o;
            return myHash.compareTo(p.myHash);
    }

    public String getName()
    {
            return myName;
    }
}
```



```java
package dismembery.common;

import java.util.*;

/**
 * Simple example of Timer that reports
 * progress to TimerListeners. This is a count-down
 * timer.
 * <P>
 * For all listeners, calls timerStarted when timer is starting,
 * calls timerTicked on each tick, and calls timerStopped when
 * the timer is stopped.
 * <P>
 * @author Owen Astrachan
 */

public class Timer implements Runnable
{

    private boolean myRunning;
    private long    myDuration;
    private long    myStartTime;
    private long    myStopTime;
    private Thread  myThread;
    private int     myTickPause;

    private List<TimerListener> myListeners;

    /**
     * Create timer with default duration of 3 seconds
     */
    public Timer()
    {
            this(3);
    }

    /**
     * Create timer with specified duration in seconds
     * @param duration is number of seconds to countdown
     */
    public Timer(int duration)
    {
            myListeners = new ArrayList<TimerListener>();
            myRunning = false;
            myThread = null;
            myTickPause = 1000;
            myDuration = duration*1000;
    }

    /**
     * Sets pause in milliseconds, this is rate
     * at which timer fires
     * @param pause is # milliseconds between timer firings
     */

    public void setPause(int pause)
    {
            myTickPause = pause;
    }

    /**
     * Add a listener to receive timer events
     * @param tl is the listener added
     */
    public void addListener(TimerListener tl)
    {
            myListeners.add(tl);
    }

    /**
     * Remove a listener (error if not listening)
     * @param tl is the listener removed
     */
```

```java
        public void removeListener(TimerListener tl)
        {
                myListeners.remove(tl);
        }

        /**
         * Starts this timer, notify all listeners
         */
        public void start()
        {
                myThread = new Thread(this);
                myThread.setDaemon(true);
                myStartTime = System.currentTimeMillis();
                myThread.start();
                for(int k=0; k < myListeners.size(); k++) {
                    myListeners.get(k).timerStarted(myDuration);
                }
        }

        /**
         * Stops timer, notifies listeners that timer has stopped
         */
        public void stop()
        {
                myRunning = false;
                myStopTime = System.currentTimeMillis();
                for(int k=0; k < myListeners.size(); k++) {
                    ( (TimerListener) myListeners.get(k)).timerStopped(this);
                }
        }

        /**
         * Does work, notifies listeners when ticking occurs
         */
        public void run()
        {
                myRunning = true;

                while (myRunning) {

                    try {
                        pause();
                    }
                    catch (InterruptedException e) {
                        myRunning = false;
                        return;
                    }

                    handleTicks();

                    if (isExpired()) {
                        stop();
                    }
                }
        }

        /**
         * Returns true if timer has expired.
         * @return true iff this timer has expired
         */

        public boolean isExpired()
        {
                myStopTime = System.currentTimeMillis();
                return timeRemaining() <= 0;
        }

        /**
         * Returns # milliseconds left before timer expires.
         * @return milliseconds remaining before isExpired() returns true
         */
        public long timeRemaining()
        {
```



```java
                myStopTime = System.currentTimeMillis();
                long secs = myStopTime - myStartTime;
                secs += myTickPause/2;
                secs = 1000* (secs/1000);
                long value = myDuration - secs;
                return value < 0 ? 0 : value;
        }

        private void pause() throws InterruptedException
        {
                Thread.sleep(myTickPause);
        }

        /**
         * Helper function to notify listeners
         */
        private void handleTicks()
        {
                for(int k=0; k < myListeners.size(); k++) {
                    ( (TimerListener) myListeners.get(k)).timerTicked(timeRemain
ing());
                }
        }
}
```

```java
package dismembery.common;

/**
 * Standard convenience adapter class.
 * Empty implementations of each TimerListener method.
 * @author Owen Astrachan
 */
public class TimerAdapter implements TimerListener
{
    public void timerStarted(long timeLeft)
    {

    }
    public void timerStopped(Timer timer)
    {

    }
    public void timerTicked(long timeLeft)
    {

    }
}
```



```java
package dismembery.common;

/**
 * Listen to Timer ticks and other events.
 * @author Owen Astrachan
 */
public interface TimerListener
{
    public void timerStarted(long timeLeft);
    public void timerStopped(Timer timer);
    public void timerTicked(long timeLeft);
}
```

```java
package dismembery.server;

import java.io.*;
import java.net.*;
import java.util.*;
import dismembery.common.*;

/**
 * Simple example of a Game (that currently responds
 * to client requests, but has no model).
 * <P>
 * Here the Game is a TimerAdapter
 * (see {@link dismembery.common.TimerAdapter
 * dismembery.common.TimerAdapter}) for simplicity.
 * The TimerAdapter functionality (only ticking here)
 * should probably be factored into a nested class.
 *<P>
 * The Game handles all server->client communication
 * though client->server communication is handled by
 * the per-client threaded GameHandler class.
 * <P>
 * @author Owen Astrachan
 */

public class Game extends TimerAdapter
{
    private HashMap myMap;
    private HashMap myScores;

    private dismembery.common.Timer myTimer;
    private GameState myState;

    /**
     * Creates a new game. Currently lasts for 500 seconds
     * and then stops forwarding requests to clients.
     */
    public Game()
    {
        myMap = new HashMap();
        myScores = new HashMap();
        myState = new GameState(myScores,0);
        myTimer = new dismembery.common.Timer(500);
        myTimer.addListener(this);
        myTimer.start();
    }

    /**
     * Process a player joining the game.
     * @param p is the player joining the game
     * @param out is the server->client stream for the joining player
     */
    public synchronized void join(Player p,ObjectOutputStream out)
    {
        myMap.put(p,out);
        myScores.put(p, new Counter());


        broadcast(Globals.PLAYERS,new ArrayList(myMap.keySet()));
        Log.println("player added "+p);
    }

    public synchronized void update(Player p, int score)
    {
        ((Counter) myScores.get(p)).incr(score);
    }

    /**
     * Called by timer, notify clients that time has ticked
     * @param timeLeft is the time left in this game
     */

    public synchronized void timerTicked(long timeLeft)
    {
```



```java
        Log.println("game ticked at "+timeLeft);
        myState = new GameState(myScores,timeLeft);
        broadcast(Globals.TICK, myState);
    }

    /**
     * Remove a list of players from the game, notifying
     * each client for each removed players.
     * @param players is a list of players leaving the game
     */
    public synchronized void remove(ArrayList players)
    {
        for(int k=0; k < players.size(); k++){
            Player p = (Player) players.get(k);
            myMap.remove(p);
            myScores.remove(p);
            broadcast(Globals.LEAVE, p);
            broadcast(Globals.PLAYERS,new ArrayList(myMap.keySet()));
            Log.println("removing player "+p);
        }
    }

    /**
     * Broadcast an object to all clients. The protocol
     * is used by clients to know what to expect, the object
     * is sent to each client and must implement Serializable.
     * @param protocol signals clients what to expect
     * @param o is the object sent to all clients
     */
    public synchronized void broadcast(String protocol, Object o)
    {
        Iterator it = myMap.keySet().iterator();
        ArrayList list = new ArrayList();         // dead players

        while (it.hasNext()){
            Player player = (Player) it.next();
            ObjectOutputStream out =
                (ObjectOutputStream) myMap.get(player);
            try{
                out.writeObject(protocol);
                out.writeObject(o);
                out.reset();
            }
            catch (Exception e){
                list.add(player);
            }
        }
        remove(list);
    }

}
```

```java
package dismembery.server;

import java.io.*;
import java.net.*;
import java.util.*;

import dismembery.common.*;


/**
 * The GameHandler class interacts with clients and process
 * incoming commands.  This current version is not object
 * oriented, but could be refactored using the techniques
 * in {@link dismembery.client.CommandFactory
 * dismembery.client.CommandFactory}
 * <P>
 * In the current architecture there is a per-Client
 * GameHandler object on the server side to handle
 * incoming client->server requests.
 *
 * @author Owen Astrachan
 */


public class GameHandler extends Thread
{
    private Socket mySocket;
    private Player myPlayer;
    private Game myGame;

    private static int ourTotal = 0;
    private static int ourPlayerCount = 0;
    private boolean myRunning = false;

    private ObjectInputStream myInput;
    private ObjectOutputStream myOutput;
    private ArrayList myList = new ArrayList();    // for removing myself

    /**
     * Construct an echo handler.
     * @param sock is the socket for the connection
     * @param db is where data is stored for this handler
     */

    public GameHandler(Socket sock, Game game)
    {
        mySocket = sock;

        myGame = game;

        // order of opening input and output here must be
        // different from order at Client side or potential
        // deadlock results. Here we use input/ouput, client
        // therefore must open output/input

        try{
            myInput = new ObjectInputStream(mySocket.getInputStream());
            myOutput = new ObjectOutputStream(mySocket.getOutputStream());
        }
        catch (Exception e){
            Log.println("Server error on streams "+e);
            return;
        }
    }

    public void run()
    {

        Log.println("handler loop starting");
        myRunning = true;
```



```java
        while (myRunning) {

            String s = null;
            try{
                s = (String) myInput.readObject();
            }
            catch (IOException ioe){
                Log.print("error in server read, handler shut down: ");
                Log.println("for player "+myPlayer);
                break;
            }
            catch (ClassNotFoundException cnfe){
                Log.println("error class read, shut down "+cnfe);
                break;
            }

            Log.println("Serviced Request "+ourTotal+" on "+s);
            ourTotal++;

            if (s == null) {
                Log.println("null string error on server");
                break;
            }
            else if (doRequest(s)){
                break; // out of loop
            }
        }
        try{
            myInput.close();
            myOutput.close();
        }
        catch (IOException e){
            Log.println("error closing object streams "+e);
            e.printStackTrace();
        }
    }

    /**
     * Handles request and returns true if we should quit
     * otherwise returns false
     */
    public boolean doRequest(String s)
    {
        if (s.equals(Globals.JOIN)){
            String name = null;
            try{
                name = (String) myInput.readObject();
                myPlayer = new Player(name,""+ourPlayerCount++);
                myGame.join(myPlayer,myOutput);
                myList.add(myPlayer);
                send(Globals.JOINSUCCESS,myPlayer);
            }
            catch (Exception e){
                Log.println("error reading player "+e);
            }
            return false;
        }
        else if (s.equals(Globals.MOVE)){
            try{
                Integer score = (Integer) myInput.readObject();
                myGame.update(myPlayer,score.intValue());
            }
            catch (Exception e){
                Log.println("error reading player "+e);
            }
            return false;
        }
        else if (s.equals(Globals.QUIT)) {
            return true;
        }
        else {
            Log.println("unknown command " + s);
            return false;
```

```java
            }
        }

        /**
         * Send an object to this client only with specified protocol.
         * @param protocol is the string identifying the message
         * @param o is the object (Serializable) written to this client
         */
        public synchronized void send(String protocol, Object o)
        {
            try{
                myOutput.writeObject(protocol);
                myOutput.writeObject(o);
                myOutput.reset();
            }
            catch (Exception e){
                Log.println("removing player "+myPlayer);
                myGame.remove(myList);
                myRunning = false;
            }
        }

    }
```



```java
package dismembery.server;

import java.io.*;
import java.net.*;

import dismembery.common.*;

/**
 *
 * The ServerStart class starts a per-client thread,
 * GameHandler, to handle communication with Clients/Views.
 * <P>
 * The GameHandler currently uses one game, created by
 * this class, ServerStart. It should be possible to create
 * new games when a game is full, for example, or when enough
 * time has elapsed since the first Client was added to the game.
 * <P>
 * All Client->Server communication is handled by the per-Client
 * GameHandler class which has the input-stream associated with
 * each Client.  Outgoing communication is handled by the Game
 * object which eithe broadcasts information to all Clients or
 * sends per-Client information as appropriate. The Game class
 * maintains a map of Client->OutputStream pairs for this
 * Server->Client communication.
 * <P>
 * Note that there is then one thread per client on the
 * server side, though all
 * handler threads share some common resources such
 * as a game.
 *
 *@author Owen Astrachan
 *
 */


public class ServerStart extends Thread
{
    private int myRequestCount;              // # requests handled
    private ServerSocket myServerSocket;     // client request source

    private Game myGame;                     // currently the only game

    public ServerStart(int port)
    {
        myRequestCount = 0;
        myGame = new Game();
        try{
            myServerSocket = new ServerSocket(port);
            Log.println("server started");
        }
        catch(Exception e){
            Log.println("Server error "+e);
        }
    }

    public void run()
    {
        while (true) {

            try {
                // accept blocks until request comes over socket
                Socket intoServer = myServerSocket.accept();

                Thread t = new GameHandler(intoServer,myGame);
                t.start();
                Log.println("Serviced client "+myRequestCount);
                myRequestCount++;
            }
            catch (Exception e){
                Log.println("trouble with server socket "+e);
            }
        }
    }
```

```
    public static void main(String args[])
    {
        ServerStart server = new ServerStart(Globals.PORT);
        server.start();
    }
}
```