```
package net108.util;

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


/**
 * This object handles the io between the server and one client.
 * It has a run method that listens for information and relays it to
 * the server.   It also has a method to send information back to its
 * client.
 */
public class BabySitter implements Runnable
{
    private Server myServer;
    private Thread mySpirit;
    private SocketConnection myConnection;
    private boolean myStopped;


    /**
     * Creates a BabySitter to handle communication with a
     * client.  Spawns a new thread to handle input from client;
     * sends objects on request.
     */
    protected BabySitter (Socket socket, Server server)
    {
        try
        {
            myServer = server;
            myConnection = new SocketConnection(socket);
            mySpirit = new Thread(this);
            mySpirit.start();
        }
        catch (Exception e)
        {
            System.err.println("Server:  socket error in BabySitter");
            myServer.removeBabySitter(this);
        }
    }


    /**
     * Sends object to the client.
     */
    protected void send (Object o)
    {
        try
        {
            Debug.println("Server:  sending '" + o + "' to " + getRepresentedClient());
            myConnection.writeObject(o);
        }
        catch (Exception e)
        {
            System.err.println("Server:  socket error in send");
            myServer.removeBabySitter(this);
        }
    }

    /**
     * Recieves new info from clients.
     *
     * @see Server#sendToAllExcept
     */
    public void run ()
    {
        try
        {
            Debug.println("Server:  BabySitter running for " + getRepresentedClient());
```



```
            while (!myStopped)
            {
                Object o = myConnection.readObject();
                Debug.println("Server:  read '" + o + "' from " + getRepresentedClient(
));
                if (o == null)
                {
                    myServer.removeBabySitter(this);
                    return;
                }
                myServer.sendToAllExcept(o, this);
            }
        }
        catch (Exception e)
        {
            System.err.println("Server:  socket error in run");
            myServer.removeBabySitter(this);
        }
    }


    /**
     * Should only be called by Server.
     * User server.removeBabySittter( BabySitter ) instead.
     */
    protected void stop ()
    {
        Debug.println("Server:  closing connection from " + getRepresentedClient());
        myStopped = true;
        myConnection = null;
    }


    public String getRepresentedClient ()
    {
        return myConnection.getRemoteHostName();
    }
}
```

```
package net108.util;

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


/**
 * A reasonably vanilla network client that works with Server.java.
 * To send objects to server, using method send.
 * To customize how objects are received from server, provide it with
 * an InputHandler (the handleInput method is called).
 */
public class Client implements Runnable
{
    ////////////////////////////////////////////////////////////
    // state
    private SocketConnection myConnection;
    private ArrayList myHandlers;
    private boolean myStopped;
    private Thread mySpirit;


    ////////////////////////////////////////////////////////////
    // constructor
    /**
     * Creates an autonomous client connected to a server running at hostname
     * and listening on port.
     */
    public Client (String hostName, int port)
    {
        try
        {
            myConnection = new SocketConnection(new Socket(hostName, port));
            myHandlers = new ArrayList();
            mySpirit = new Thread(this);
            mySpirit.start();
        }
        catch (Exception e)
        {
            throw new RuntimeException("Client: could not establish connection with " +
                                       hostName + ":" + port);
        }
    }


    ////////////////////////////////////////////////////////////
    // public methods
    /**
     * Reads objects from the server.  Called by this object's Thread.
     * Should not be called otherwise.
     */
    public void run ()
    {
        try
        {
            Debug.println("Client: starting read loop." );
            while (!myStopped)
            {
                Object input = myConnection.readObject();
                Debug.println("Client: read from server " + input);
                notifyHandlers(input);
            }
        }
        catch (Exception e)
        {
            throw new RuntimeException("Client: failed to read");
        }
    }
```



```
    /**
     * Use this method to send an object to the server.
     */
    public void send (Object o)
    {
        try
        {
            Debug.println("Client: sending " + o + " to server.");
            myConnection.writeObject(o);
        }
        catch (Exception e)
        {
            throw new RuntimeException("Client: failed to send " + o);
        }
    }

    /**
     * Add specialized handler to customize how server input is received.
     */
    public void addInputHandler (InputHandler handler)
    {
        myHandlers.add(handler);
    }

    /**
     * Remove server input handler.
     */
    public void removeInputHandler (InputHandler handler)
    {
        myHandlers.remove(handler);
    }

    /**
     *  Closes the socket, stops the thread.
     */
    public void stop ()
    {
        try
        {
            myStopped = true;
            myConnection = null;
        }
        catch (Exception e)
        {}
    }

    private void notifyHandlers (Object o)
    {
        Iterator iter = myHandlers.iterator();
        while (iter.hasNext())
        {
            ((InputHandler)(iter.next())).handleInput(o);
        }
    }
}
```

```
package net108.util;

import java.io.PrintWriter;


/**
 *  This class is a collection of functions for printing
 *  debugging info (i.e., you can turn it on and off
 *  globally).
 *
 *  <p>
 *  Use this class in place of System.out.println like so:
 *  <blockquote>
 *    <tt>Debug.println(foo);</tt>
 *  </blockquote>
 *  where foo is a primitive type or an Object.
 *
 *  <p>
 *  You could do this without making every function
 *  static by expecting users to write:
 *  <blockquote>
 *    <tt>new Debug().println(foo);</tt>
 *  </blockquote>
 *  but I have found C++ programmers have problems with
 *  this style.
 *
 *  <p>
 *  Objects to be printed should implement the function
 *  toString() in a meaningful way.
 */
public class Debug
{
    ////////////////////////////////////////////////////////////
    // state
    private static PrintWriter ourWriter = new PrintWriter(System.out);
    private static boolean ourShouldDebug = false;


    ////////////////////////////////////////////////////////////
    // constructors
    public static void setWriter (PrintWriter out)
    {
        ourWriter = out;
    }

    ////////////////////////////////////////////////////////////
    // public methods
    public static void turnOn ()
    {
        ourShouldDebug = true;
    }

    public static void turnOff ()
    {
        ourShouldDebug = false;
    }


    // yuck :(
    public static void print (boolean b)
    {
        if (ourShouldDebug) { ourWriter.print(b); ourWriter.flush(); }
    }

    public static void print (char c)
    {
        if (ourShouldDebug) { ourWriter.print(c); ourWriter.flush(); }
    }
```



```
    public static void print (int i)
    {
        if (ourShouldDebug) { ourWriter.print(i); ourWriter.flush(); }
    }

    public static void print (long l)
    {
        if (ourShouldDebug) { ourWriter.print(l); ourWriter.flush(); }
    }

    public static void print (float f)
    {
        if (ourShouldDebug) { ourWriter.print(f); ourWriter.flush(); }
    }

    public static void print (double d)
    {
        if (ourShouldDebug) { ourWriter.print(d); ourWriter.flush(); }
    }

    public static void print (String s)
    {
        if (ourShouldDebug) { ourWriter.print(s); ourWriter.flush(); }
    }

    public static void print (Object o)
    {
        if (ourShouldDebug) { ourWriter.print(o); ourWriter.flush(); }
    }


    // yuck yuck :(
    public static void println ()
    {
        if (ourShouldDebug) { ourWriter.println(); ourWriter.flush(); }
    }

    public static void println (boolean b)
    {
        if (ourShouldDebug) { ourWriter.println(b); ourWriter.flush(); }
    }

    public static void println (char c)
    {
        if (ourShouldDebug) { ourWriter.println(c); ourWriter.flush(); }
    }

    public static void println (int i)
    {
        if (ourShouldDebug) { ourWriter.println(i); ourWriter.flush(); }
    }

    public static void println (long l)
    {
        if (ourShouldDebug) { ourWriter.println(l); ourWriter.flush(); }
    }

    public static void println (float f)
    {
        if (ourShouldDebug) { ourWriter.println(f); ourWriter.flush(); }
    }

    public static void println (double d)
    {
        if (ourShouldDebug) { ourWriter.println(d); ourWriter.flush(); }
    }

    public static void println (String s)
    {
```

```
        if (ourShouldDebug) { ourWriter.println(s); ourWriter.flush(); }
    }

    public static void println (Object o)
    {
        if (ourShouldDebug) { ourWriter.println(o); ourWriter.flush(); }
    }
}
```

```
package net108.util;


/**
 * An interface for objects that can handle Objects received from the
 * network by a Client.
 */
public interface InputHandler
{
    public void handleInput (Object o);
}
```

```
package net108.util;

import java.io.PrintWriter;


/**
 * A default Handler.  Just prints things to System.out.
 */
public class PrintInputHandler implements InputHandler
{
    private PrintWriter myWriter;


    public PrintInputHandler ()
    {
        this(new PrintWriter(System.out));
    }

    public PrintInputHandler (PrintWriter writer)
    {
        myWriter = writer;
    }


    /**
     * Just print the object.
     */
    public void handleInput (Object o)
    {
        myWriter.println(o);
        myWriter.flush();
    }
}
```

```
package net108.util;

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


/**
 * Generic Server class:  broadcasts whatever messages it receives to
 * all clients other than the message's originator.  Spawns one thread
 * to listen and another for each client as they connect.
 */
public class Server implements Runnable
{
    ////////////////////////////////////////////////////////////
    // state
    private ArrayList myClients = new ArrayList();
    private ServerSocket mySocket;
    private Thread mySpirit;


    ////////////////////////////////////////////////////////////
    // constructors
    /**
     * Creates a server socket.  Waits for connections.
     */
    public Server (int port)
    {
        listenOn(port);
        mySpirit = new Thread(this);
        mySpirit.start();
    }

    /**
     * Wait loop to service the ServerSocket.  Called by this server's
     * Thread.  Should not be called externally.
     * Each time that  a connection is recieved, forks a BabySitter to
     * handle that connection.
     */
    public void run ()
    {
        try
        {
            Debug.println("Server:  listening for connections");
            while (true)
            {
                spawnBabySitter(mySocket.accept());
            }
        }
        catch (IOException e)
        {
            System.err.println("Server:  abrupt failure on accept attempt.");
            System.err.println("Server:  no longer accepting connections.");
        }
    }


    ////////////////////////////////////////////////////////////
    // helper methods
    /**
     * Helper method to actually open the ServerSocket and intialize
     * other state.  Also spawns Thread to continue listening.
     */
    protected void listenOn (int port)
    {
        try
        {
            Debug.println("Server:  starting up on port " + port);
            mySocket = new ServerSocket(port);
```



```
        }
        catch (IOException e)
        {
            throw new RuntimeException("Server:  failed to listen on port " + port);
        }
    }

    /**
     * Creates a BabySitter with the client socket passed and
     * adds it to the list of BabySitters.
     *
     * @param s the client socket that this BabySitter will handle
     */
    protected void spawnBabySitter (Socket s)
    {
        myClients.add(new BabySitter(s, this));
    }

    /**
     * Removes a BabySitter for the list of babySitters.
     *
     * @param bbst the BabySitter to be remooved from use.
     */
    protected void removeBabySitter (BabySitter sitter)
    {
        myClients.remove(sitter);
        sitter.stop();
    }

    /**
     * Sends the object passed in to all clients accept the one
     * represented by the BabySitter passed in.
     */
    protected synchronized void sendToAllExcept (Object o, BabySitter sitter)
    {
        Iterator iter = myClients.iterator();
        while (iter.hasNext())
        {
            BabySitter current = (BabySitter)iter.next();
            if (current != sitter)
            {
                current.send(o);
            }
        }
    }
}
```

```
package net108.util;

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

/**
 * Networked Wire, Client Side.  Provides readObject, writeObject.
 */
class SocketConnection implements Wire
{
    private Socket mySocket;
    private ObjectInputStream myInput;
    private ObjectOutputStream myOutput;


    /**
     * How to make one, if we know who we want to talk to.
     *
     * @param hostName  the name of the machine that the server is on
     * @param port      the port number on which the server is listening
     */
    public SocketConnection (Socket socket)
    {
        try
        {
            mySocket = socket;
            // socket protocols require that we create Output before Input
            myOutput = new ObjectOutputStream(mySocket.getOutputStream());
            myInput = new ObjectInputStream(mySocket.getInputStream());
        }
        catch (Exception e)
        {
            throw new RuntimeException("");
        }
    }

    /**
     * Use this to read an Object from the Wire.
     *
     * @returns the Object read.
     */
    public Object readObject ()
    {
        try
        {
            return myInput.readObject();
        }
        catch (Exception e)
        {
            throw new RuntimeException("failed to read from " + getRemoteHostName());
        }
    }

    /**
     * Use this method to write an Object to the Wire.
     *
     * @param o The object to be written.
     */
    public void writeObject (Object o)
    {
        try
        {
            myOutput.writeObject(o);
            myOutput.flush();
        }
        catch (IOException e)
        {
            throw new RuntimeException("failed to write to " + getRemoteHostName());
        }
```



```
    }

    public String getRemoteHostName ()
    {
        return mySocket.getInetAddress().getHostName() + ":" + mySocket.getPort(
);
    }


    public String getLocalHostName ()
    {
        return mySocket.getLocalAddress().getHostName() + ":" + mySocket.getLoca
lPort();
    }


    /**
     *  Closes the Socket and Streams.
     */
    public void finalize ()
    {
        try
        {
            myInput.close();
            myOutput.close();
            mySocket.close();
        }
        catch (IOException e)
        {}
    }
}
```

```
package net108.util;


/**
 * A generic interface for stream-like things that can read and write
 * objects.
 */
public interface Wire
{
    /**
     * Read the next object.  Block if no object is available.
     */
    public Object readObject ();

    /**
     * Write an object.  Blocks if no space is available to write to
     */
    public void writeObject (Object obj);
}
```