UNIVERSITY OF
BIRMINGHAM

EE2E1 2006/2007

Introduction to Java Programming

Programming Exercise 5

Networking, Threads and GUI's

Dr M.Spann

1. Aims and Objectives

This is the last of the assessed Java labs and involves a combination of some things you have already done before (GUI's) with some things that you have not (networking). Specifically, you are to design a networked snakes-and-ladders game that will allow 2-5 players to simultaneously play the well know board game over the network. You will need to write a multi-threaded server to handle client connections and to manage the state of the game. You will also need to write a graphical client to allow each of the players to play the game.

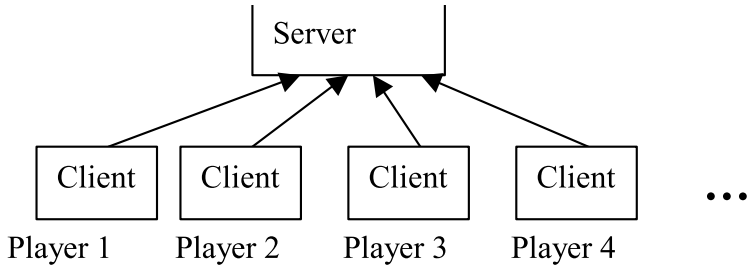

Figure 1: Multiple clients (one per player) connecting to the server.

A class to display the board (on the client), *SnakesAndLaddersGUI*, has already been written for you and **you are expected to use this class in your code**. It will be your responsibility to write the client and server networking code, the code for the snakes-and-ladders game logic, and a simple client user interface to interact with the user.

2. Preparatory Work

One of the main tasks of this assignment is to develop a multi-threading server, and associated client. Networking in Java is about the easiest of any programming language. A simple introduction is given in Appendix 2. Make sure you have read this, and have some understanding of what is going on before you come into the lab. You should also read your notes on Threads and Exceptions, both of which will be useful when writing networking programs.

You will also need to think about how to use Layout Managers to create simple GUI's.

It will also be helpful if you understand the rules of snakes-and-ladders (see Appendix 1).

3. Lab Work

3.1 Introduction

The files *SnakesAndLaddersGUI.class*, *Test.java* and *board.gif* have been zipped in the file *SnakesAndLadders.zip* which you can download from http://www.eee.bham.ac.uk/spannm/Java Stuff\Snakes and Ladders\SnakesAndLadders.zip. Unzip the files, compile and run *Test.java* and you should see something like this:

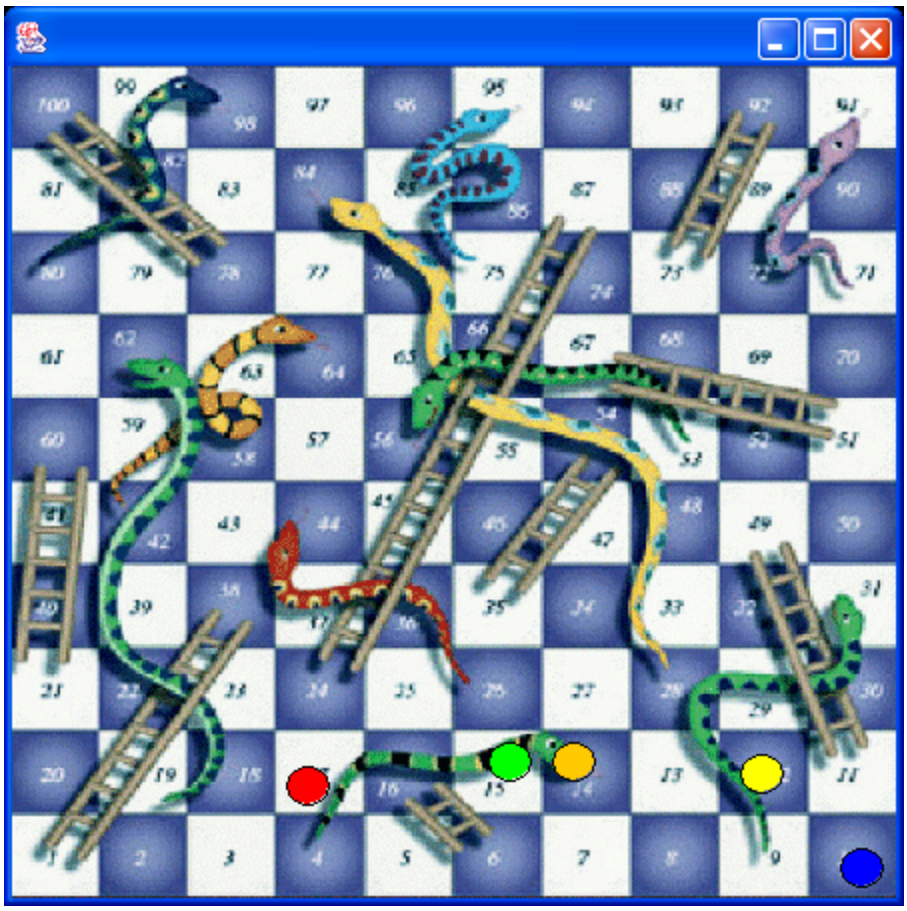

Figure 2: Snakes-and-ladders board – This is what you get as you run *Test.java*.

The class *SnakesAndLaddersGUI* will be used to display the board, and the players' counters. It has been written to make it very easy for you to use. The only two things you need to know about are:

1) It is inherited from *JPanel* (from the *javax.swing* package) – it therefore behaves exactly like a *JPanel* does.

2) It has two interesting public methods:

public void setNumberOfPlayers(int _noofplayers)
//should be obvious from the name....call this first
//before moving counters about

public void setPosition(int playerno,int position)

```
//Moves the counter belonging to 'playerno'
//to square 'position' (NB: if you have 3 players, they
//are numbered 0,1,2 – squares are numbered 1-100,
//squares outside the 1-100 range make the counter invisible)
```

The class *Test* is a small example program that uses a *SnakesAndLaddersGUI* object, and moves the player's counters along in a random manner (ignoring snakes and ladders). Note how it just adds the *SnakesAndLaddersGUI* object (which is of type *JPanel*) to a *JFrame* (which is the thing that you can see in Figure 2). You are not expected to use *Test.java* in your code, it is just an illustration of how you might want to make use of the *SnakesAndLaddersGUI* class (so read it!).

3.2 Networked Snakes-and-Ladders

Now that you have seen how to display the board and move the counters around we come to the real work.

You have 2 separate programs to write:

1) A server,
2) A client.

The Server

The server is made up essentially of two parts. **Firstly** it tracks the current game position, handles the game logic (including going up the ladders and down the snakes) and rolls the dice when asked. **Secondly** the server program is responsible for accepting connections from clients and passing/receiving messages to/from them. An example of how to do networking in Java is given in Appendix 2.

How you decide to structure your code into classes is up to you, but if you structure them into logical classes (with logical names) you will find things are much easier to write and test. For example, does the class that manages the status of the game really need to know that the game is networked?

It should be noted that the server works in two phases. In the first phase it just sits there waiting for the clients to connect. After all of the clients have connected, then we move in to the second phase, that of playing the game.

The Client

In contrast to the server you will be simultaneously running a number of clients, one for each player. Like the server, the client also contains two parts. **Firstly** it must have a networking part which connects to the server and passes/receives messages to/ from it. **Secondly** it must have a graphical display for the user to interact with. This graphical display will show:

1) The snakes-and-ladders board (done for you).
2) Some kind of message box to inform the player of what is happening (e.g. It's your go!).
3) A roll dice button, which only has an effect when it is your go.
4) A box to enter the IP address of the server (the network address of the PC running your server, so that your client can find it – see Appendix 2 for details of networking).
5) A connect button to connect to the server (after the IP address has been entered).

You will need to layout all of these components in a *JFrame* using one or more layout managers.

Note: For testing purposes you can run the server and all of your clients on the same computer. In this case you can enter the IP address of 127.0.0.1 (which means 'on the same computer').

3.3 Test Case

It's important to be clear about which bit is doing what, and when. Essentially you start your server and multiple clients (each player runs a client). The clients then all connect to the server, and the game starts. It might be helpful to consider a typical sequence of events:

Connecting phase

1)Server is started on PC1
2)Server waits for connections
3)Someone starts a client program on PC2
4)User on PC2 enters the IP address of the server (otherwise the client won't know where the server is)
5)User on PC2 pressed 'Connect' button
6)Client on PC2 tries to connect to server on PC1
7)Server on PC1 gets the connection from PC2, starts a new thread (it's a multi-threading server) and goes back to listening for incoming connections
8)Someone starts a client program on PC3, enters IP address and pressed 'Connect'
9)Server on PC1 gets the connection from PC3, starts a new thread and goes back to listening for incoming connections
10)When server has enough players, the game can start.

Game phase

1)The server alerts all clients that the game is starting, and that all counters should be placed on square 0 (off the board and invisible).

2) The server alerts everyone that it is player 1's turn. Nothing happens until player 1 presses the 'throw dice' button. The client of player 1 notifies the server that player 1 wishes to roll the dice.
3) The server informs all of the clients of the dice value and updated counter positions (taking into account all of the snakes and ladders).
4)This is repeated for all of the players until someone wins (gets to square 100).

3.4 Design and Programming Hints

Good separation of the code into suitable classes will make complexity much easier to manage. Think carefully about what you make public in a class. Your public methods define how your class will be used. It should be obvious how to call your public methods and exactly what they do.

Also design, code and test in stages. If you write the whole thing in one go and then test it, you are inviting disaster. Get each bit working separately – for example write a simple client server program that just sends 'hello' and then prints it out. Only when you can do this should you think about adding in game functionality. Also if you have lots of bits that do something then you will have something to show if it doesn't all work together – that's much better than one big program that doesn't compile.

4. Assessment

The overall objective is to produce a multi-player networked application that can play the game snakes-and-ladders. One component of the assessment will be a practical demonstration of your program where you should be able to demonstrate what your program does. If your program does not fully work you should demonstrate the individual elements that do work (e.g. networking, game functionality, GUI, etc.)

The formal report should include the headings outlined in the Introduction to the Java Programming Laboratories document plus any additional headings you want to include. You should include a description of the algorithm you designed and how all of the classes interact with each other.

Appendix 1 : Rules of Snakes and Ladders

1)Every player has a coloured counter
2)At the start of the game all the counters are on square 0 (off the board).
3)Each player takes turns in rolling a dice.
4)The player's counter advances the number of squares shown on the dice.
5)If the counter finishes on the bottom of a ladder it moves to the top of the ladder.
6)If the counter finishes on the top of a snake it moves to the bottom of the snake.
7)If a player reaches square 100 they win.

8) If the current position plus the dice number exceeds 100, they don't move (e.g. on square 98 – rolls a six – player doesn't move)

Appendix 2: Client/Server Networking in Java

Questions and Answers

1)What is a client… and what is a server?

Server: Sits around waiting for a client to connect.
Client: Connects to a server.

2) How does the client find the server?

Every server has an IP address (every PC in the lab has a different one), and a port number. The client must know both of these in order to find the server.

3) So what is my IP address?
Type 'ipconfig' in a command window ('DOS prompt') and it will tell you. Alternatively if your client and server are **on the same machine** you can use the IP address 127.0.0.1 which means 'on this machine'.

4) What port number should I use?

Any number you like above 1023 and below 65536. Just make sure that your client and server are both using the same port. Also be aware that you cannot have two servers listening on the same port.

5) What happens after I connect?

After you connect you use streams to send and receive messages between the client and the server.

6) So how do I write this in Java?

Have a look at this (client and server) – the key lines are in bold.

```
////////////////////////////////////////////////////////////
//Client class TestClient.java
////////////////////////////////////////////////////////////
import java.net.*;
import java.io.*;

public class TestClient{
  private final int PORT=3000; //Port number
  private PrintWriter out=null;
  private BufferedReader in=null;

  public TestClient(){//Constructor
     try{
        //Try to connect to server at IP address/port combination
```

```
        Socket soc=new Socket("127.0.0.1",PORT);
        //Now get the 'streams' - you use these to send messages
        out=new PrintWriter(soc.getOutputStream(),true);
        in=new BufferedReader(new InputStreamReader(soc.getInputStream
()));
        out.println("Hello!\n");
        out.close();
     }
     catch(Exception e){
       e.printStackTrace(); //Displays Error if things go wrong....
     }
  }

  public static void main(String args[]){
     new TestClient(); //Makes object (calls constructor)
  }
}
```

```
//////////////////////////////////////////////////////////////
//Server class TestServer.java
//////////////////////////////////////////////////////////////
import java.net.*;
import java.io.*;

public class TestServer{
  private final int PORT=3000; //Port number
  private PrintWriter out=null;
  private BufferedReader in=null;

  public TestServer(){//Constructor
    try{
        ServerSocket ss = new ServerSocket(PORT);
        Socket soc=ss.accept(); //Waits for client to connect
        //Now get the 'streams'
        out=new PrintWriter(soc.getOutputStream(),true);
        in=new BufferedReader(new InputStreamReader(soc.getInputStream()));

        while(true){//Infinite loop

          String s=in.readLine(); //reads line 'from client'
         if(s==null)break;//Abort if null string
         System.out.println(s); //Prints it out
        }
    }
    catch(Exception e){
        e.printStackTrace(); //Displays Error if things go wrong....
    }

  }

public static void main(String args[]){
  new TestServer(); //Makes object (calls constructor)
}
}
```

6) What about if there are multiple clients?

Good question! The above code will only work for a single client - if you read it you should be able to work out why (hint: the **ss.accept()** line accepts the connection) - You will need a *multi-threaded server* to allow multiple simultaneous connections (because you have multiple clients) - you will need to work out how to do this.