Hassan M. Khan
Operating Systems
Network Programming Project Report

Introduction:

We have all watched Animal Planet/Discovery on the television. Often times they would show some interesting battles, such as lions versus snakes or buffalos versus cheetahs. I tried to recreate this Animal Planet environment in my game, by making a two player game, where each player selects an animal without knowing what the other player selected. Once each player has picked an animal, one player is declared the winner based on the strength of their animal in comparison to their opponents. The players are then asked if they want to replay the game.

The game is played with three animals, lion, cobra, and rabbit; and the game rules followed are the following:

1. Lion eats rabbit. (Lion would be the winner).
2. Rabbit can run from cobra. (Rabbit would be the winner).
3. Cobra can bite lion. (Cobra would be the winner.

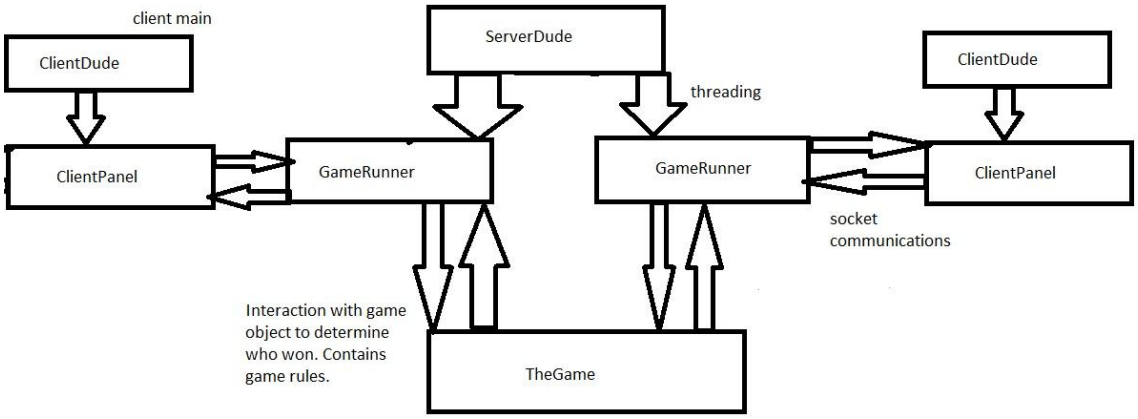
Architecture:

I followed the server, client and threading concept for this game. The server runs first, listening on a port on the host computer for any clients wanting to connect. Once a client sends a request to get connected, using the server's IP address and port number, the server accepts the connection, and waits until another player connects, since this is a two player game. Once both players are received by the server, a thread is started for each client, which sends and receives messages from them. The thread also communicates these messages with a game object, which also takes in two players from the two threads. The game object decides who wins, based on the animals picked by each user, which is forwarded to the game object by the threads for each client.

Once a winner is picked, the game sends this information to both the client threads, which then communicates whether the user has won or lost, back to the socket of the user, where it is received, and displayed in a Graphical User Interface.

The client side has a panel object which maintains the GUI, and also updates it after communicating with the client's respective thread on the server side. The panel object is initiated, packed and run on a frame, declared in a main on the client side.

The following diagram shows all the different classes implemented for the game, and how each class interacts with the other.



Server-side:

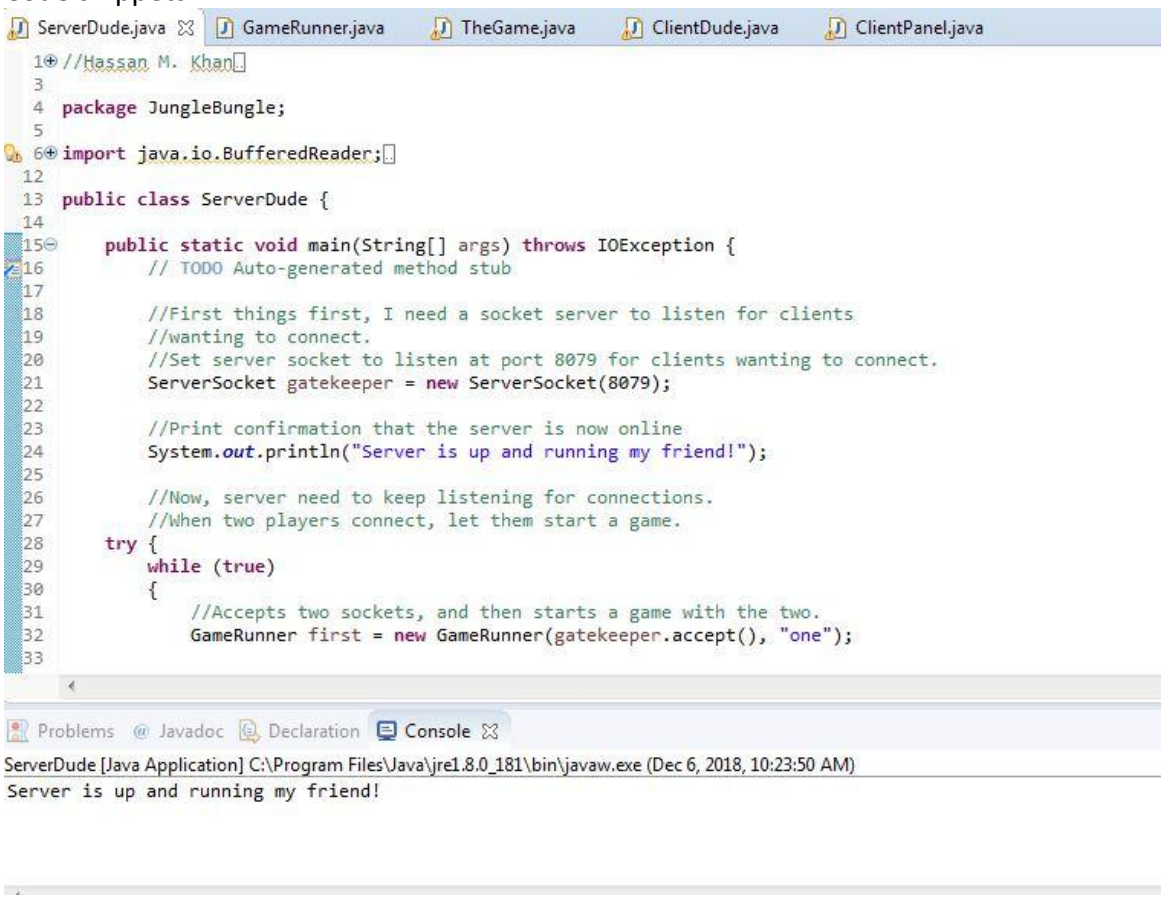
As seen above, the ServerDude class runs the server. It listens for client sockets that want to connect, by using a ServerSocket object.

```
//First things first, I need a socket server to listen for clients
//wanting to connect.
//Set server socket to listen at port 8079 for clients wanting to connect.
ServerSocket gatekeeper = new ServerSocket(8079);

//Print confirmation that the server is now online
System.out.println("Server is up and running my friend!");
```

Once a client has connected, it waits for another one to connect, since we need two players to play the game. Once two clients are connected, it gives each an ID (either “one” or “two”, this is so that they can be differentiated from one another in the game object), and then creates a new game with the two clients. It also passes the address of the game to each client thread, so that the threads and the common game object can exchange information.

Once all this is done, the server then starts each thread, and continues to listen for more clients wanting to connection, by running the “while (true)” loop forever, until it has to close.

Code snippets:



```
//Accepts two sockets, and then starts a game with the two.
GameRunner first = new GameRunner(gatekeeper.accept(), "one");

//Tell the first one, that we are waiting on a player for him.
first.tellUser("We are waiting on a player for you!");

GameRunner second = new GameRunner(gatekeeper.accept(),"two");

System.out.println("Two gamers found!");

//Start the game object
TheGame game = new TheGame(first, second);

System.out.println("New game Started!");

//Set the game object for each gamerunner, which is the thread for each client,
//so that the individual client threads can interact with the game.
first.setGame(game);
second.setGame(game);

//Lets start the game
//Both threads started here, so that at this point we have two players.
first.start();
second.start();
```

The game runner object is the individual thread for each client. It communicates with the client side using the socket address of the client, which was passed to the game runner object from the server.

```
public void getSelection() throws IOException
{
    //Get the client's choice of animal.
    out.println("Welcome to the game! Choose your animal!");
    selection = in.readLine();
}
```

It asks the client to choose an animal, and then receives what animal was chosen by the client, in the input stream. It then passes on this information to the game object, which implements the rules of the game, and checks which client had picked what animal.

```
//Lion and rabbit setup
//Lion beats rabbit.
if (one.selection.equals("lion")&&two.selection.equals("rabbit"))
{
    first.tellUser("You won! Lions eats rabbits!");
    second.tellUser("You lost! Lions eats rabbits!");
}

if (two.selection.equals("lion")&&one.selection.equals("rabbit"))
{
    second.tellUser("You won! Lions eats rabbits!");
    first.tellUser("You lost! Lions eats rabbits!");
}

//Cobra and lion setup
//Cobra beats lion.
if (one.selection.equals("cobra")&&two.selection.equals("lion"))
{
    first.tellUser("You won! Cobras bite lions!");
    second.tellUser("You lost! Cobras bite lions!");
}

if (two.selection.equals("cobra")&&one.selection.equals("lion"))
{
    second.tellUser("You won! Cobras bite lions!");
    first.tellUser("You lost! Cobras bite lions!");
```

Once the game determines the winner, it uses the gameRunner object, which has a connection to the client's socket input stream, to inform the client whether they had won or lost. The telluser function seen above belongs to the gameRunner object. Each game has two gameRunner objects, and hence direct contact with the two threads running for each client. This makes it easy to send information to the client, directly from the game object, via the gameRunner thread.

Client-side:

On the client side, we have a main in the clientDude class, which has the code to create, pack and run a JFrame/GUI for the client side.

```
public static void main(String[] args) throws HeadlessException, UnknownHostException, IOException {
    // TODO Auto-generated method stub

    //Start the client frame, panel and game.
    startClient();

}

public static void startClient() throws HeadlessException, UnknownHostException, IOException
{
    //Start the client's frame.
    JFrame frame = new JFrame("Jungle Bungle!");

    //Initiate the client panel, and add it to a Jframe.
    ClientPanel panel = new ClientPanel(frame);

    //Setting up the frame.
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setPreferredSize(new Dimension(620,520));
    frame.getContentPane().add(panel);
    frame.pack();
    frame.setVisible(true);

    //Lets start the game.
    //Connect with the server.
    //Get the server ip the user wants to connect to.
    panel.connectServer(JOptionPane.showInputDialog("Hello! Enter the IP address of server!:"));
}
```

Although the frame is run from the main of clientDude, the connection of the client to the server-side is established via the clientPanel class, which initializes and runs the client-side socket, and hence has the client side's input and output stream.

```
public void connectServer(String a) throws UnknownHostException, IOException
{
    //Set up the socket for the user to connect.
    //Port number is hardcoded on both the server and the client.
    //This is to make sure they connect on the same port.
    socket = new Socket(a, 8079);

    //Setting up the input and output buffer of the client,
    //so that we can read and write to it.
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    out= new PrintWriter(socket.getOutputStream(), true);

    //Start game
    messageCheck();
}
```

The port number the client and server will connect to are hardcoded, to prevent any anomalies or refused connection possibilities.

The clientPanel class runs the GUI for the client side, by having various features such as buttons which allows the user to select an animal, and a textbox which receives and displays messages from the server.

Code snippet:

```
rabbit = new JButton("");
rabbit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //Setting up the action handler for this button.
        //Change the user's selection side image
        userselect.setIcon(rabbiticon);

        //Lock the other buttons
        lion.setEnabled(false);
        cobra.setEnabled(false);

        //If user chooses this button, send this to the server side thread.
        out.println("rabbit");

        //Go to the next function, and wait for the client to send information
        try {
            messageCheck();
        } catch (IOException e1) {
            // TODO Auto-generated catch block
            //Do nothing
        }
    }
});
```

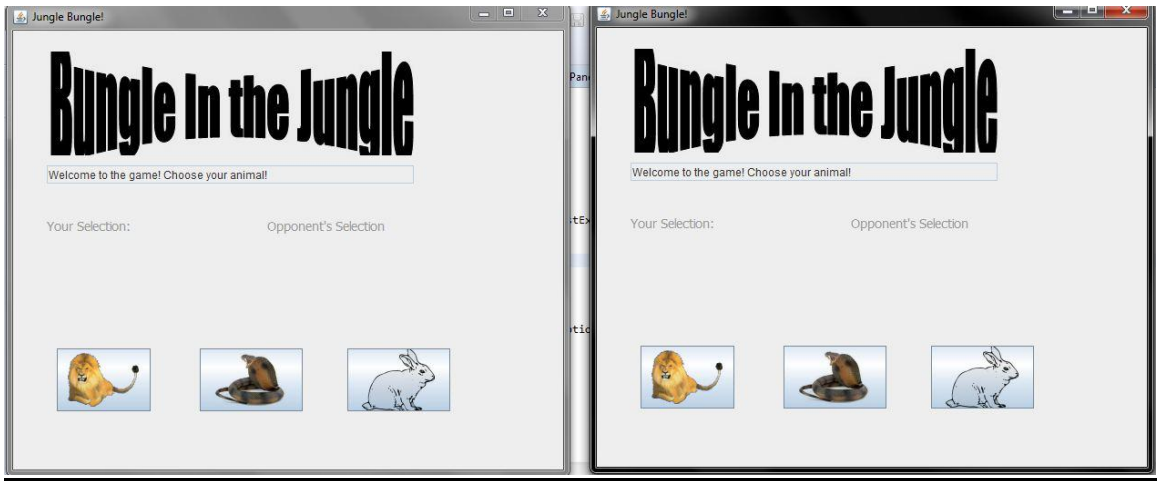
Each button has an actionListener listening on it, which sends the selection of the user to the server-side thread, via the output stream, and also deactivates all the other animal buttons, so that the user cannot pick two animals at the same time.

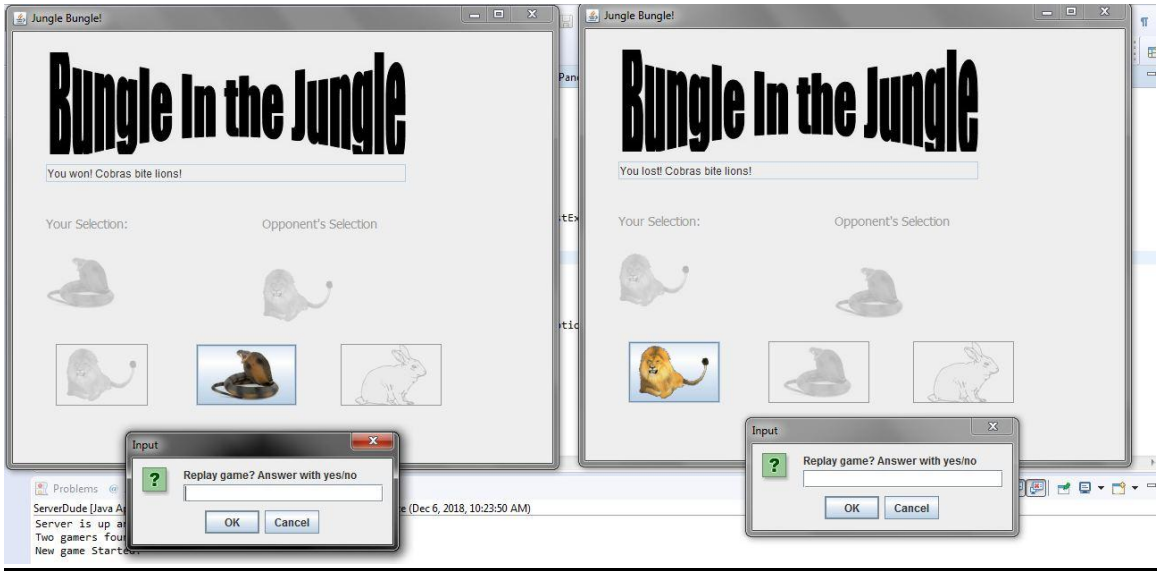
Any messages sent to or received from the server-side thread are picked up here at the clientPanel class, via the client socket. The panel class then displays different things on the GUI based on what the server-side thread has sent from the game object.

Code snippet:

```
//This function is a while loop to keep running while we need input from the server side.
//The server side can send messages to the client, which are then displayed on the clients label.
public void messageCheck() throws IOException
{
    while (true)
    {
        //Display messages from the server side on the label
        String teller = in.readLine();

        //If the server-side sends what the client picked, display picture on label.
        if (teller.equals("lion") || teller.equals("cobra")|| teller.equals("rabbit"))
        {
            if(teller.equals("lion"))
            {
                opponentselect.setIcon(lionicon);
            }

            else if (teller.equals("cobra"))
            {
                opponentselect.setIcon(cobraicon);
            }

            else if (teller.equals("rabbit"))
            {
                opponentselect.setIcon(rabbiticon);
            }
        }
```

GUI snippets of gameplay:





Once the game is played, and the winner and loser are displayed their information on their respective GUIs, the user is then asked whether they want to replay the game. If so, the clientPanel closes the current window, and restarts itself by closing the socket connection, and reconnecting again, by call the restart method startClient(), in the client-side main. If the user wants to quit, the program is exited.

Code snippet:

```
//Check if the user won or lost
//If so, the game has ended at this point.
//Ask the user if they want to replay the game. Break while loop.
//Close the socket.
//If user wants to replay, hide current frame, and start a new game.
if (teller.contains("won")||teller.contains("lost"))
{
    socket.close();

    if(JOptionPane.showInputDialog("Replay game? Answer with yes/no").equals("yes"))
    {
        frame.setVisible(false);
        ClientDude.startClient();
        break;
    }

    //Otherwise, exit system and close program.
    else
        System.exit(0);
}
```

Functional requirements and conclusion:

They key functionality requirements of the program was socket connection, socket communication, threading, and object orientated programming. The server side implemented a different version of the regular socket, which listens to a port for further connection requests from other client sockets. The client side implemented a regular socket with an input output stream, for the client side to be able to communicate with the thread for that respective client running on the server side. The input/output streams of the sockets were used extensively to send and receive data to and from each side. Threading was only implemented on the server-side, for each individual client, because each server has to manage multiple clients. The client-side however only had to manage itself, and the connection to the server, and hence did not require further threading.

Object orientation was a major key to achieving smooth transitions between the server side thread, the game and the client-side thread. Each server-side thread for each individual player was passed into the game object to be able to use the functionality of the server side thread to directly send messages from the game object, instead of having to pass messages back to the server-side thread first before being transferred to the client.
Each server-side thread was also passed the address of the game itself, so that the functions of the game such as checking who won, can be easily achieved.

Similarly, on the client side, the reference to the frame object which contained the client-panel GUI, was passed into the panel itself. This was done so that the panel can control close the old frame and refresh and start a new game is the user wants to, from within itself, instead of having to go back to the main function from the client-panel to restart the process.

Overall this project was immensely fun, as I got to experiment with network programming by myself like never before, and therefore helped me grasp the concepts of sockets, servers, clients and threading, and how to use them.

……