Portfolio 1

Donovan Brooks

Matthew Wall

Trivalries is a server/client trivia based game written in Java.

Download the source code, host or join a game (using your hosts IP),

and be the person with the most points after 10 rounds and you win.

TABLE OF CONTENTS

PROJECT OVERVIEW

For our project, we decided to work a little more with multithreading. We felt like a solid foundation in Java Threads would be really good to know since it was our first lab. That being said, similar to lab one– where we made a chat application between a server and multiple clients– our game provides an experience where one would play against others while still having certain elements controllled by a server. Now it wasn't so easy breezy,; we did have a few questions off the get go. One being, how are we going to handle all of these different events that are happening? And also, how can we determine what needs to be handled on the server side vs. the client side? These were an apparent struggle, and continued to be throughout the project. I'll go into how we solved those in the latter pages.

A little fun side note about our project is it actually fetches questions from a trivia website we found. There's a library online called JSoup (which I had heard of before, but never messed around with it) which we thought would be cool to implement. So we decided, "why not?" This being said– and it's also reported on github in the README file provided– our game, as it's currently written, requires an internet connection to play. Now we did handle the case where you try to play without a connection and it catches the error, but we wanted to have an offline version but unfortunately ran out of time. Now we'd messed around a little with accessing the DOM of a website and through JSoup we could do just that. I won't get into much detail here, but on page five you'll find more material about that there.

Another thing we'd never really messed around with was the Window Builder eclipse plugin. We were struggling on tedious activities such as resizing elements when someone in the Coover TLA told us about Window Builder. WOW! What a game changer. This wasn't a big part of our project, but it did serve it's purpose between the give GUIs we had. It was also nice to get accompanied to it in case we need to use it again in the near future.

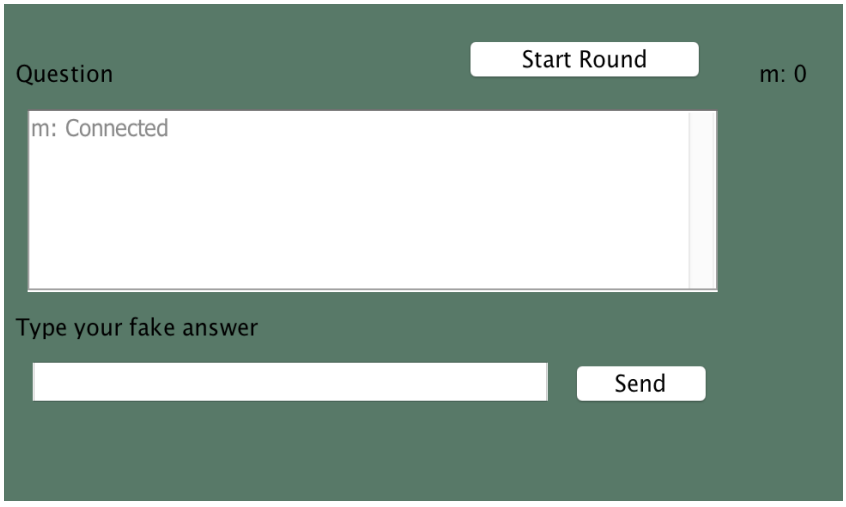
I'll provide a brief description of how to play the game for those of you who may be confused. When you first run Login it'll prompt you to enter a(n): username, password, email, and choose a color. We didn't get around to saving these like we planned. With the username we wanted to save the password that corresponded with your username and/or email. Anyway, once you click continue, you'll see a screen where you're prompted to enter the IP of your host or create your lobby based off your IP. I'll go through the first case first. If you're not the leader of the party, you can't start the round. So you're pretty much stuck until your party leader decides to play. If you're the host, you click "Start the Round," and the round begins. In both cases, you'll be required to enter a believable answer that might trick your opponents. After the timer is up, you'll be redirected to another GUI where you'll answer the question. Score works as follows: 1 point for each client that guesses your fake answer, 2 points for each answer you guess correctly. The game ends after 10 rounds and the person with the most points wins.

SERVER & CLIENT RELATIONS

The basis of our whole game is based around the idea where we have multiple clients threaded to a server thread. This allowed us to keep different tabs on certain elements in our game. If you're the host of the game, loosely connected to the server.

```
EventQueue.invokeLater(new Runnable() {
    public void run() {
        try {
            server = new Server(1222);
            Client client = new Client("localhost", username, password, email, color, 1222, true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
});
```

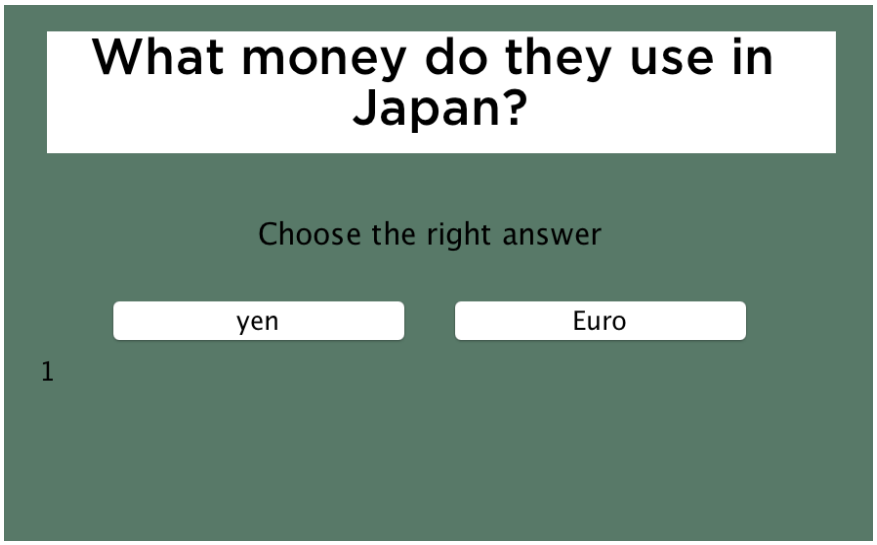
As you can see, in the event the user clicks (or just presses enter) "Start Game" they launch a new server and create themselves a client thread as well. If you're connected to an already launched server, it would just create your client thread. Once this is done, you're added to an array of clients at index clientNum. Inside of the Client class, we handle all the messages that the sever sends us with the function handleChat(Object msg). Based on which type the 'msg' is, we do different things with it. Vice versa, when we send messages back to the server, it's all handled through the handle(String[] input) method. This is the basis of the game, where we're continuously passing information to and from the server and clients. Below is a screenshot of the ClientGUI.



It displays the question once you click "Start Round," reads in your answer input, and keeps track of the score for each user to the right (displayed in this picture as m:0). Once the timer runs out for inputting the fake answer, a new gui will pop up.

SERVER & CLIENT RELATIONS (cont.)

The new GUI that pops up is based upon the input each client provides. For all the client's, we got rid of their own answer, and in it's place, put the correct answer. This prevents each client from selecting their own fake answer. Here's what the QuestionGUI looks like:



The timer is displayed (as a '1' here) and each client's fake answer as well as the correct answer. In this instance, there are two clients playing. As mentioned before in the overview, the client's will get 2 points for guessing the correct answer. As well as that, if someone chooses your fake answer you'll receive 1 point on top of that. To us, customization was a big thing. We provided 4 options for changing the UI's color (green, blue, red, and orange) so you don't have to keep looking at the same interface over and over again.

As with any server and client application, we have our required functions. These include: start(), stop(), run(), etc. And with each class, we have it's corresponding Thread.

> "A thread is an independent path of execution within a program."

In our case, the Server thread is always running and listening for incoming connections. When a connection is made, it sends a signal to the client thread that tells it to start(). Upon starting, the client thread does a few things. Obviously it connects back to the server socket, but it also assigns the client thread a port number. Ports are huge in our project. We identify each client and what needs to happen to each client by their corresponding port number.

```
streamIn = new ObjectInputStream(socket.getInputStream());
streamOut = new ObjectOutputStream(socket.getOutputStream());
```

The client's job is to listen for server calls, as well as talk back to it through I/O Streams.

JSOUP IMPLEMENTATION

Another fun addition to our project (which came very late in the process) was the idea to fetch trivia questions from a source. We originally had the idea to just make an ArrayList of 100 or so questions and use those randomly to display. This would be an awesome solution to our offline problem, but again, we didn't implement a fail-safe method. All this code is inside of Question.java

Here's how it works. First we have to setup a connection to the website. Now we have a document we can get and post to (in our case, get).

```
static Document setupConnection(String site) throws IOException{
    return Jsoup.connect(site).timeout(10*1000).get();
}
```

Next we have to do a few different things. In this case, the questions were easily stored in the "pre" tag on the website. So we build an ArrayList to grab all the elements inside of that tag. From that, we parse the data into our questions Object which we store– once again– in an ArrayList of Questions. Here's a visual example of what I just said:

```
public void fillQuestionList () {
    String site = "http://www.classicweb.com/usr/jseng/trivi.htm";
    try {
        //get the site and set the timeout to 10 seconds
        Document doc = setupConnection(site);

        //build the Arraylist based on the "pre" element
        ArrayList<String> list = buildList(doc, "pre");

        //parse the category, question, and answer into a
        //Question object and store it in list
        parseData(list);

    } catch (IOException e){
        System.out.println("Cannot fetch html from " + e.getMessage());
    }
}
```

To us, we felt like this was a huge contribution to our project. We had never fetched data from websites before inside of Eclipse so that was huge. Accessing the DOM of websites was a cool addition to what we had in mind, and actually made it a little easier on us since we didn't have to write a hundred random trivia questions. I think this was a good example of using programming to make life easier. Since there are already TONS of trivia questions and applications out there, we wanted to write something that could pull from those in existence and utilize them. Since we did it in a way that you could specify which element you wanted to pull from, you could essentially do this for multiple websites, each having their questions stored in different tags.

BLOOM'S TAXONOMY ANALYSIS

Looking back on this project, it was a heck of a lot of fun we must say. But as with every fun project, comes every fun projects side-effects. Briefly, we'll try to describe what we'd like to change if we could go back and do things again.

Number 1: **Reducing use of global variables**

We have an excessive number of global variables. I know generally this isn't a problem because of how much memory computers have today. But what if this was a larger application? That would be a lot of wasted memory we could reduce so easily.

Number 2: **Split a lot of busy code into more functions**

This was definitely apparent in our server side code, in particular the handle function. It's not terribly ugly, but a lot of it could've been reduce if we created different functions for things. One example of this would be a timer function that takes two paramters: an Integer paramater called duration and a String direction which executes the duration in the direction given.

Number 3: **Finish adding different functionalities**

For the login process, we wanted to add a way to cache the user information from last login. If the user has logged in using the same username or email before, it would autofill the password and chosen color that the user entered last login time. We also thought it would be cool if we made this on a mobile platform. That way, users could connect to each other and play with each other on the go, much similar to the game "Words With Friends." We also wanted to spruce up the UI quite a bit more. There were cases where the UI felt lacking when we wanted it to do something, but this could've been fixed by maybe doing it in a different platform.

Number 4: **Combine GUI classes into an abstract class**

I think if we had made all the GUI classes into an abstract class, it would've made the whole GUI implementation a heck of a lot neater. In addition to this, it would've been nice to have all the GUIs open in one frame, and have it switch tabs with progression. I don't think how we did the interface is bad by any means, I just believe there is genuine room for improvement with how we designed each one. My favorite design we used is the QuestionUI. I love how we used a different font for the question field and used buttons for selecting the answers. It's simple, yet very clean. Anyway, if we used abstract classes we could reduce a lot of code that is redundant. Alongside this, we could grab caching using the DocumentListener library.

BLOOM'S TAXONOMY EVALUATION

As aforementioned, a lot of thought process had to go into evaluating what processes needed to be handled on the client's side vs the server's side. For example, in our Server class we started out not having a deep understanding of how we were supposed to handle having a chat system as well as being able to interpret when those messages were supposed to be for the fake answer. It turns out it was as simple as wrapping the code wanted to run during the round in a global variable that toggles after the Timer runs out.

```
if(roundStarted == true){
        //do something
}
```

Another huge issue we got hung on for quite some time was the idea of copying an ArrayList of objects. Now I believe in 228 we learned the difference between shallow and deep copying, but man did we forget that. We were getting such weird to happen and honestly couldn't figure it out for a good while. We finally solved that by creating copy constructors in our Answer and Score class.

```
public Answer(Answer a) {
        this(a.answer, a.port);
}

public Score(Score s) {
        this(s.port, s.username, s.score);
}
```

Almost all the interfaces have some sort of Action tied to them, so figuring out when it was best to handle those was also a headache. For example, each GUI has a Mouse Listener attached to every button it has, as well as (in most cases) a Key Listener. Looking closer, let's take a look at our ClientGUI class. It has a Mouse Listener on the "Send" button (also on "Start Round" for host), a Key Listener on the text field. Knowing when to attach these and what they were doing was a big part of our project. Since we'd never used Window Builder before, it was awesome to be able to implement Actions directly in that interface.

Also we should note that there were a lot of base cases we had to cover. Example: What if the user didn't enter a fake answer? Do we add an answer for them, or do we add nothing? In our case we came to the conclusion that the only thing we could do was not show the buttons corresponding to their answer on the QuestionUI. This was a simple and dirty solution to someone accidentally (or maybe on purpose) not guessing an answer.

BLOOM'S TAXONOMY CREATION

This project was loosely based around the demo code we had from Lab 1, but we'll list out the major parts we threw together to make it what it is now.

- Our Question class was completely written from scratch
- Created 4 GUIs in addition to the ServerGUI with customization for users to have a more pleasant experience
- Created 2 new objects– Score & Answer in which we can store and access while the game is running
- In addition to creating the objects, we handled the stream of data that is passed to the clients (apparent in Client>handleChat)
- Generated new threads to handle client/server relation
- Stored clients in a list for access to their elements after creation
- A fully functional, beta version of a trivia game!

Conclusion

So, what did we learn? Well, mainly– we gained a better understanding of how servers and clients interact with each other. It was our main goal to really get a deep feel for how we can manipulate data on one side and recognize it on the other. We did this through the chat handling funtions which you'll see in our source code. This project also gave us a little more understanding of git and how we can work together even when we're not with each other (which is very important when you're both very busy). But maybe the most important thing that was learned is that it takes a lot of work to make even the most simplest of games. So we gained the knowledge it takes to make these types of games, and an appreciation for those who've made some that are already out there. We also learned to comment our code and create JavaDoc when possible, because that'll not only help you, but your workmates in the long run. Overall, this project was a good time and we're definite fans of Server/Client relations.