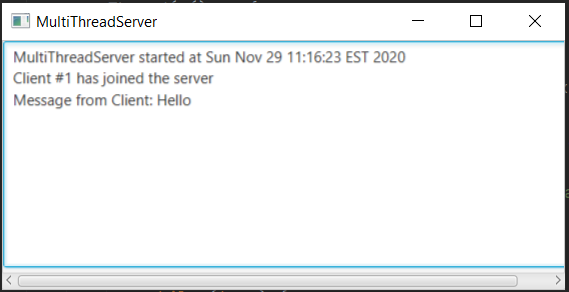
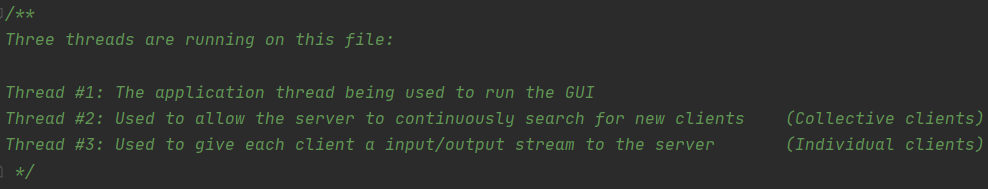
Programming Project #4: Short Report **Project Participant(s)**: Zachary Doctrove

***Prompt****: Discuss how you went about designing and implementing your project, which team members did what part(s). All team member names must be listed on the cover/first page of the report, or they will not receive credit. Only one team member should upload the assignment.*

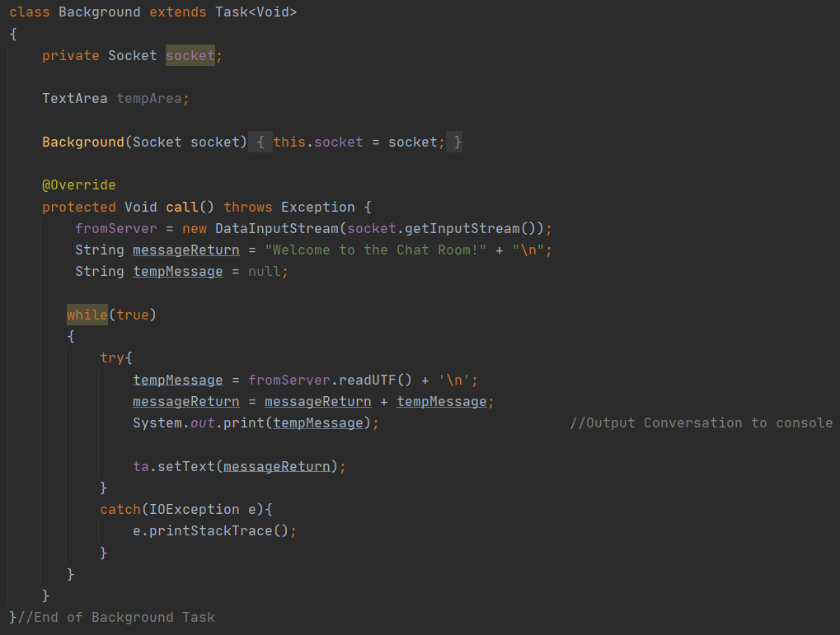
Introduction

I started this final project on the Sunday before thanksgiving break and I completed it on the Sunday after thanksgiving break. I completed it by myself, using the “MultiThreadServer” and “Client” programs from the book as my foundation. This is most obvious from the fact that my java files and the package they are stored in still retain their original names.

The Server

The very first part I worked on was the server. Since the GUI of the original book program was already fully integrated with the socket, I used it as my way of reporting messages sent from the client. GUI aside, the most important/difficult aspect (to me at least) was the threads used to facilitate the program. Due to this, I made a comment at the top of my server file summarizing the three threads I need to manage: The Application thread that runs the GUI, the socket search/connect thread, and the socket input/output thread. All three of these threads were a part of the foundation, making my task one of editing the software to complete tasks in line with what I needed. For instance, the original socket input/output thread found in “HandleAClient” did not originally account for inter-client communication. To address this, I had to put all the clients in an ArrayList and create a method called “outToAll” to systematically message each client instance.

The Client

 The part I struggled the most with was getting the Client GUI to constantly update itself when a new message was sent from any other client. I originally was hoping to repurpose the “HandleAClient” class, as to make its full-time job one of data reception. The problem with that came from my use of the “runLater( () -> { } );” method in a while loop. This is because evoking this method allows non-Application threads to change the GUI while it is not doing anything. It does this by queueing the statements inside “runLater” onto the Application thread, allowing for changes that take mere milliseconds to occur without interrupting the user. The problem I ran into was caused by me queueing an infinite loop onto the GUI, that prevented users from interacting with the GUI and forcing task manager to get involved. Eventually I learned of the “concurrent” class and its “Task” subclass, which I used to get around this issue. No longer using “runLater”, my program now uses the Task “Background” to receive server outputs, print them to console, store the entire Chat App conversation in a string, and update the GUI TextArea with that string. This works because instead of constantly trying to receive inputs that cannot even be typed to begin with, the program now allocates that task to another thread which directly changes the entirety of the TextArea in one assignment. Other than the GUI debacle, the client side of the server was relatively easy in the fact that the foundation already set up most of the work for me.

Conclusion

This project took a while, but I am happy with how it came out. I originally thought the idea of networking by itself would be the difficult part, but the use of threading required to get it to work was a lot harder to wrap my head around. I also wanted to note that I let the server have up to 4 users at once, due to the execution pool being limited to four threads at a time. If you attempt to add a fifth client, that instance will not be able to connect to the server properly and send messages to the other users. Their client window still receives the other client’s messages, as the client file still contains the ability to receive data from the server.

