Name: Tai-Juan Rennie

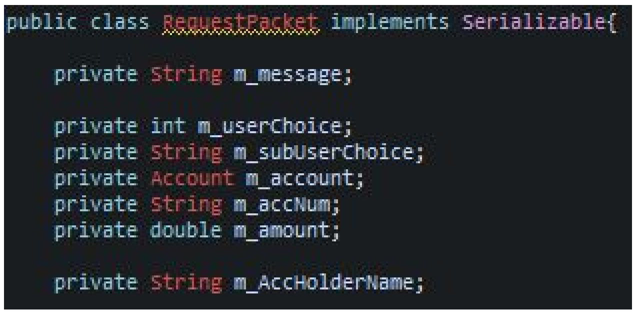
ID : 101359172

BTP400 Final Assessment: Multi-Threaded Bank Server

When I first programmed the financial app, I went out of my way to make things extremely modular. Actions as simple as getting a user's input for an account number or account type were divvied up into separate discrete functions. Due to this extreme approach to modularity, dividing my original financial app into server and client compartments became more manageable.

The general plan of implementation was having the client side represent the UI side of the original financial app. It would request and store user input and send it off to the server. Then I would have the server side have the original CRUD logic of the financial app which would use the inputs sent from a client to perform the requested actions.

The first design hurdle was deciding on a format that would be used for the server and client to communicate to one another. I decided on the ObjectInputStream and ObjectOutputStream classes to send over my own custom RequestPacket & ResponsePacket objects between the client and server. The RequestPacket was strictly made for the client compartment. Its sole purpose was to categorically store any inputs from the user. The one exception to this rule was the Account type member variable because I found it much simpler code wise to send an Account object over to the server verses storing all the inputs needed to create an account. If I had chosen to store all the inputs needed to create an account, then I would end up with an abundance of single-purpose member variables. After the RequestedPacket is finished storing all the user's inputs it is then sent to the server where it's getters can be used to extract the appropriate data.



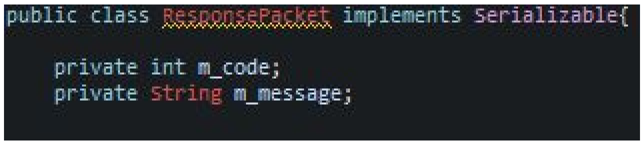
```
public class RequestPacket implements Serializable{

    private String m_message;

    private int m_userChoice;
    private String m_subUserChoice;
    private Account m_account;
    private String m_accNum;
    private double m_amount;

    private String m_AccHolderName;
```

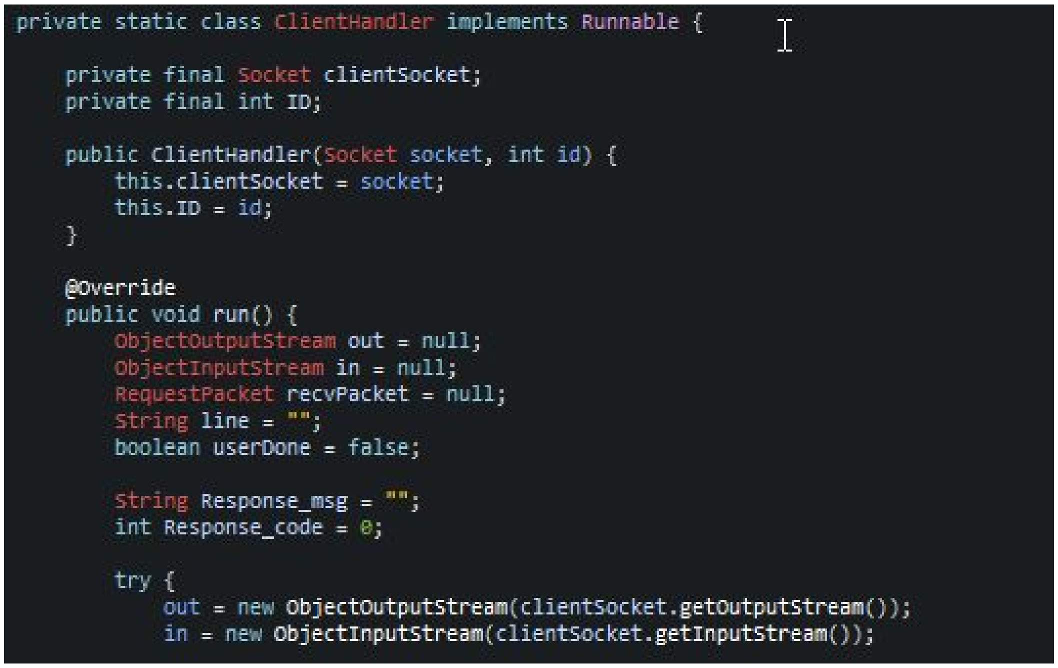
As for the ResponsePacket class it was made server compartment. Its purpose was to store any message or information it wanted the client to print out. It also contain a status code that tells the client whether the server ran into an error. Once the ResponsePacket is completed it sent off to the server where its contents are printed by the client.


```java
public class ResponsePacket implements Serializable{

    private int m_code;
    private String m_message;
```

As far as recycling classes go all of the classes for the financial app such as Account and Bank were used in the server-side code while only Account and its children were reused in the client side.

My biggest challenge with this assignment was my general lack of experience when It came to convert a single threaded java server into multi-threaded server. However hours of research later provide me with clarity and greater understanding. The logic used for the server side was simply put into a custom class that implemented Runnable. Hence, anytime a client connected to that server I would initiate a new thread with that class's object.


```java
private static class ClientHandler implements Runnable {

    private final Socket clientSocket;
    private final int ID;

    public ClientHandler(Socket socket, int id) {
        this.clientSocket = socket;
        this.ID = id;
    }

    @Override
    public void run() {
        ObjectOutputStream out = null;
        ObjectInputStream in = null;
        RequestPacket recvPacket = null;
        String line = "";
        boolean userDone = false;

        String Response_msg = "";
        int Response_code = 0;

        try {
            out = new ObjectOutputStream(clientSocket.getOutputStream());
            in = new ObjectInputStream(clientSocket.getInputStream());
```

The final complication with this assignment was simply making my code thread safe. It was very important to avoid race conditions as it could leave my data inconsistent and unpredictable. I utilized the Atomic family of classes to achieve this. The atomic class is specifically built to make variables thread safe.

Any member variable in the Account class family or in the Bank class that were susceptible to change were wrapped in an atomic reference or integer. These variables included Bank's account ArrayList, Chequing's Transaction history and Account's balance. After that it was just modifying existing logic to utilize Atomic's thread safety mechanisms such as compareAndSet. I also took the extra setup to make the remove and add account functions in Bank synchronized so only one thread could access them at one time.