# Introduction

This project is once again a continuation of the great bank concept initially created by Daryl in his first project. With this project, we will add multithreading capabilities to the bank application as well as begin storing data in a database with the bank application. In order to demonstrate the multithreading in a realistic way, a client/server implementation was added to the application that ended up requiring much more work than simply incorporating multithreading itself. As per usual, we will first discuss some key concepts related to this project, then dive into the program itself.

# Key Concepts

There are a few key concepts introduced in this report that need a brief introduction before we get into how the code works: Multithreading and Client/Server Connections

## Multithreading

Most people are already familiar with multitasking, the ability to do multiple chores or tasks all at the same time. This is applied to computers as well. A CPU is able to multitask by running multiple processes at the same time. Multithreading is taking it a step further. Individual programs can run multiple tasks at the same time by using multiple threads in the same program.

The applications of this concept are limitless, but tend to make the biggest difference with tasks that take a long time to run. While a program is executing one task, the rest of the program can keep on running without having to wait for that task to be completed. Picture a GUI that gets locked into performing a task and so none of the buttons work until the task is completed. This can be extremely frustrating. We fix this by running that task on its own thread, and the rest of the GUI runs on the original thread. Buttons can be pressed and the user can continue working with the program while the greedy task runs on its own thread in the background.

There are two main methods of using multiple threads within a program: implementing the runnable interface and extending the thread class. When you implement the runnable interface, you can simply create a class to be run inside of a thread. The interface has only one method to override: run. All code that you intend to run on the separate thread goes here. Of course, you can still use helper methods within the new class to help organize your code. In order to use the class, instantiate your object as a runnable, then create a thread and pass the thread your runnable in the constructor. Finally, call the start method on your thread and all of your code in the runnable class will now execute on a separate thread.

For the second method, create your own class that extends the thread class. Once again, override the run method and put in the code you would like to run in a separate thread. Then, instantiate your class in the main code and call the start method.

## Client/Server Connections

There are many ways to implement a client and server state between programs. Essentially, a server is hosted in a specific location and has an address to that location. Clients can use the address to begin communication with the server. The client is always the one to initiate communication. The server can be set up to accept either only one client or multiple clients. Both the client and server can talk to each other by sending packets of information back and forth. What stream they use can be determined by the user in code.

We will be doing this in our project using the Socket and ServerSocket classes provided by Java. The ServerSocket class will create a server that listens to a specific port. When a client, implemented with the Socket class, makes a request over that port, the server evaluates the request, the responds. For the purposes of this project, we will be using a localhost on port 4444. Both the server and clients need to know the port number to operate on.

# The Basic Bank Program

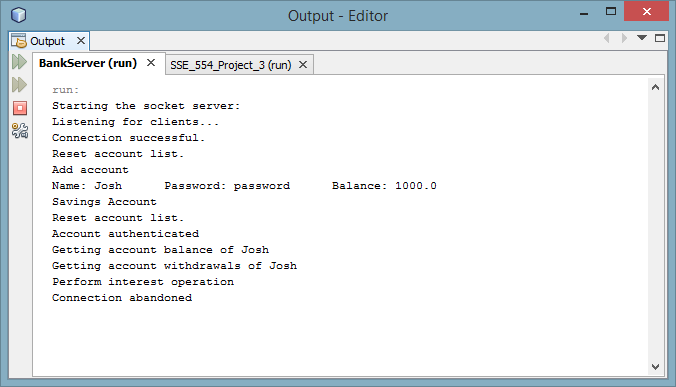
This application is building off of the original idea by Daryl Ebanks in his SSE 554 Project 1, where the application is intended to simulate a basic banking application. The user will be able to both create and access their accounts through the application and perform certain functions such as setting a password, withdrawing or depositing, and performing an interest calculation on all of the accounts. All of these operations will be performed through the graphical user interface.

This project aims at adding to the program by creating a client/server connection method and allowing multiple clients to connect to the server. On the back end, the server will store all data for the bank in a database.

## The Client/Server Implementation

The goal for this implementation was to split up the application into the combination of a client-side application, which only focuses on giving and receiving data from a user through a graphical user interface, and a server-side application, which performs calculations and operations and stores the data. In order to do this, two executables were created. The BankServer class contains the executable for the server and the BankGUI contains the executable for the client.

In order to run the program, the BankServer class must be started first. This class gets the server up and running and listening on the right port. Inside this class, we listen for a client to attempt to connect to the server. When a client is found, we launch a new thread to handle all interactions with that client. The server keeps listening and will interface with every client program that talks to the server on that port. Within each thread, we create an object stream to send information to and from the client. First, we ask for a request number. This number corresponds to what the client wants from the server. With each number, we perform the necessary operation and send either data, a confirmation, or both back to the client. In the figure below, we show the output to console of the bank server. It connected with a single client and performed a few operations for the client before the client disconnected.

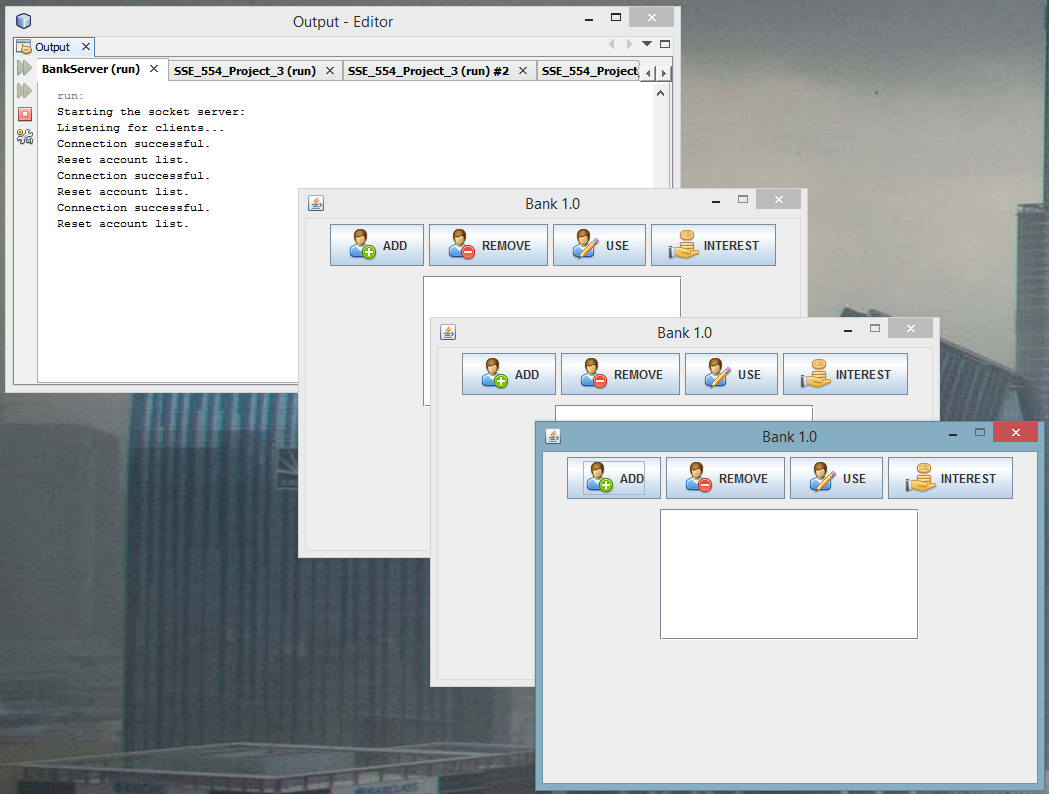


Running the client application will have a familiar effect. The GUI remains unchanged from the project 1 implementation and all actions taken should be the same. The intent is for the client to not realize where or how the actions are taking place. The code for the GUI was modified to use the proper data types for the situation, however, the implementation itself remains unchanged. In order to talk to the server, the bank client class was created. This class connects to the server and sends requests through the object streams created. All requests initially stem from user input to through the GUI. Below can be seen several figures of the GUI in action.

// Insert images here

## Multithreading

Our goal was to have the capacity to support multiple clients connecting to the server at the same time. To do this, we used a multithreading approach. As mentioned earlier, each time a client requests to communication with the server, a new thread is created for that client. The server keeps listening for new clients on the main thread. This is necessary because on each client thread, a continuous while loop is listening for input from the client on any new requests the client has. The server cannot do this and listen for more clients to connect at the same time, thus multiple threads were needed. Whenever a client disconnects, the thread notices and closes accordingly. The image below shows three clients connected to the server at the same time.



All three clients have successfully connected to the server are now communicating with their own unique thread on the server, and the server is still listening for more clients. Obviously, continuing to add more clients will eventually impede server speed because of all the processing power, however, many more clients have been added successfully without crashing the application.

# Conclusion

In conclusion, the project was a success. We learned a great deal about multithreading and by extension, more than ever expected about developing a client/server connection between programs. In addition, the database works well and is an extended solution to storing the data between runs of the server program. From the client perspective, everything still seems to work the same way despite the massive restructuring behind the scenes.

If I were to start over and build the client and server implementation again, I would make all objects passed through between the client and the server arraylists of objects. I would then create separate classes that would know how to interact with these arraylists and pull the proper information out in order to use with the program. This would make the client and server interactions much more reusable and not require near as much rework if any each time some sort of change was made to the banking program. We would simply need to work with the hopefully much simpler interface classes.