PROO 3 - Assignment: *Socket Banking*

Objective

Get familiar with *Sockets* in Java by implementing a simple *client-server*-application for performing bank transactions.

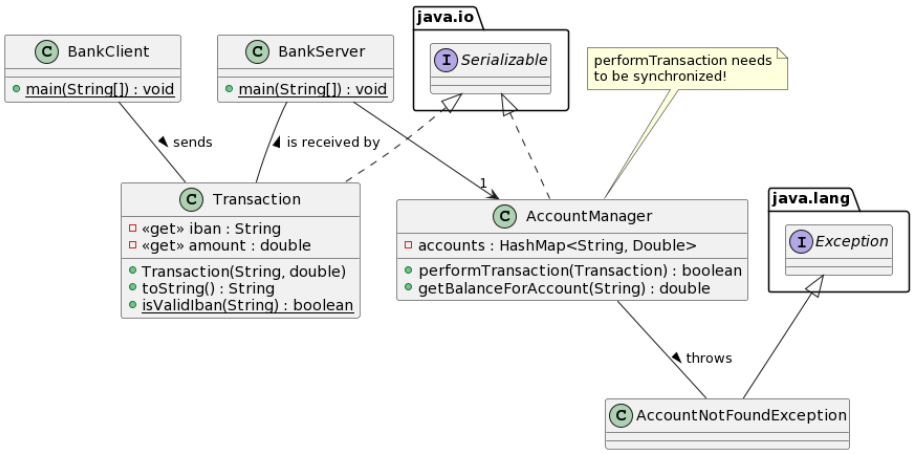
Things To Learn

- Working with `Sockets` and *Streams*.
- *Serializing* objects.
- Working with `Threads` and synchronization.

Submission Guidelines

- Your implemented solution as ***zipped*** *IntelliJ*-project.

Task



In this assignment you'll create a simple, text-based application for very rudimentary *online banking*. Start by preparing the following classes:

- A `Transaction` is represented by an *International Bank Account Number* (*IBAN*) and the transaction amount. Both properties are read-only.
  - An `IllegalArgumentException` is to be thrown, if the *IBAN* is invalid. Use the static *helper* method `isValidIban` for this - the validation algorithm is described in the *Hints* section.
  - Make sure the class is *serializable* so we can pass it using *streams*.
- The `AccountManager` stores all bank accounts - the most minimalistic solution is a *map* of `String` (i.e. *IBAN*) to `Double` (i.e. the balance) values, but you are free to use whatever you like.

  - *Transactions* can be performed using the `performTransaction` method.
    * If the *IBAN* does not exist, a new account with the respective balance is created.
    * Overdrafts are not possible - return `false` if a transaction would lead to a negative belance.
    * Make sure that the method is `synchronized` and think of reasons why this is necessary in the context of a *multi-threaded server*.
  - The balance of an account can be retrieved using `getBalanceForAccount`. If the passed *IBAN* does not exist, a (*checked*) `AccountNotFoundException` is to be thrown.
  - The `AccountManager` should be serializable as well for *persistence* reasons.

Next, prepare a *main* class for the *server*:

- The `BankServer` should listen and accept incoming connections on the *port* `7070` using a `ServerSocket` (of course).
  - Bonus: Create an own *thread* when accepting a connection to enable multiple clients at once.
- Since your server will be receiving serialized objects you should use an `ObjectInputStream`, as well as a `PrintWriter` for text-based answers to the connected client.
  - Make sure to use `try-with-resources`-statements as shown in the *demos*!
- The accounts are stored in an `AccountManager` object.
  - Bonus: Use *serialization* to make sure the accounts are persisted when stopping and starting the server.

Finally, you can create the *client*:

- After asking the user for an *IBAN* (until a valid one is entered!) the client should connect to the server and allow multiple amounts to be transferred until the input `quit`.
- Since your client will be sending serialized objects you should use an `ObjectOutputStream`, as well as a `BufferedReader` for text-based answers from the server.

You can use the following exemplary outputs to guide you through the implementation:

Exemplary Client Output

```
Please enter your IBAN: AT022050302101023600
Valid IBAN entered.
Enter amount for new transaction (or "quit" to quit the application): 10.0
Response from server: Transaction successful. New balance: 10.00.
Enter amount for new transaction (or "quit" to quit the application): haxi
```

```
Number not formatted correctly.
Enter amount for new transaction (or "quit" to quit the application): 5.0
Response from server: Transaction successful. New balance: 15.00.
Enter amount for new transaction (or "quit" to quit the application): -10.0
Response from server: Transaction successful. New balance: 5.00.
Enter amount for new transaction (or "quit" to quit the application): -10.0
Response from server: Transaction unsuccessful.
Enter amount for new transaction (or "quit" to quit the application): 2
Response from server: Transaction successful. New balance: 7.00.
Enter amount for new transaction (or "quit" to quit the application): quit
```

Exemplary Server Output

N.B.: The server output is not really necessary but proves useful for debugging purposes.

```
SERVER: Starting.
SERVER: Listening for a connection.
SERVER: Connection accepted.
SERVER: Listening for a connection.
SERVER-THREAD: Starting.
SERVER-THREAD: Received object "AT022050302101023600: 10.00,-".
SERVER-THREAD: Received object "AT022050302101023600: 5.00,-".
SERVER-THREAD: Received object "AT022050302101023600: -10.00,-".
SERVER-THREAD: Received object "AT022050302101023600: -10.00,-".
SERVER-THREAD: Received object "AT022050302101023600: 2.00,-".
SERVER-THREAD: Received null. Finishing.
SERVER-THREAD: Finished.
```

Hints

Validating *IBANs*

From *Wikipedia*:

> An IBAN is validated by converting it into an integer and performing a basic mod-97 operation (as described in ISO 7064) on it. If the IBAN is valid, the remainder equals 1. The algorithm of IBAN validation is as follows: 1. Move the four initial characters to the end of the string 2. Replace each letter in the string with two digits, thereby expanding the string, where A = 10, B = 11, ..., Z = 35 3. Interpret the string as a decimal integer and compute the remainder of that number on division by 97 If the remainder is 1, the check digit test is passed and the IBAN might be valid.

Example (also from *Wikipedia*):

- IBAN: GB82 WEST 1234 5698 7654 32

- Rearrange: W E S T12345698765432 G B82
- Convert to integer: 3214282912345698765432161182
- Compute remainder: 3214282912345698765432161182 mod 97 = 1

Since you will be dealing with potentially very large numbers you could use the `BigInteger`-class for accomplishing this easily. The *constructor* accepts a `String` and there is also a `mod`-method.