Data Structures
CSC 3280

Fall 2021

Program 1: Mocs Bank

Assigned: Thursday, September 9, 2021
Due: Friday, September 17, 2021 by 11:59 PM

Purpose:

1. The first purpose of this program is for you to REVIEW your Java and OOP.
2. The second purpose of this program is for you to review file I/O (input/output).
3. Finally, you are to implement binary search in the program.

Read Carefully:

- This program is worth 7% of your final grade.
- **WARNING**: This is an individual assignment; you must solve it by yourself. Please review the definition of cheating in the syllabus, the FSC Honor Code, and the serious consequences of those found cheating.
    - **The FSC Honor Code pledge MUST be written as a comment at the top of your program**.
- When is the assignment due? The date is written very clearly above.
    - **Note:** once the clock becomes 11:59 PM, the submission will be closed! Therefore, in reality, you must submit by 11:58 and 59 seconds.
- LATE SUBMISSION: you are allowed to make a late submission according to the rules defined in the syllabus. Please see course syllabus for more information.
- **Canvas Submission**:
    - This assignment must be submitted online via Canvas.
    - Within IntelliJ, your project should be named as follows:
        - `MocsBank`
        - And when you make the project, you will want to name the package as **`mocsbank`** (all lowercase). Please do this exactly as typed.
    - You should submit a single **ZIP** file, which has **ONLY** your Java files inside it.
    - You should NOT submit the entire IntelliJ project.

<u>**Program 1: Mocs Bank**</u>

<u>**Objective**</u>

Review and practice all concepts from CSC 2290, with a focus on making classes and creating objects from these classes. Specifically, you will practice making an array of objects and manipulating/using the data inside the array. Finally, and VERY important, this program requires you to read from a file and write to a file (practice with File I/O). This is very important because all programs during the semester will use File I/O.

<u>**Program Description**</u>

FSC will be opening a bank strictly for members of the FSC community, and, best of all, it'll be housed right here in Weinstein! (not really…this is just the story of the assignment). The purpose of this program is to design a simulation of the FSC Mocs Bank. Students can open accounts at Mocs Bank. They can make withdrawals, deposits, print statements, and more. You will write a program that will simulate normal banking activities at Mocs Bank.

During the program, the following things can occur (just as in a normal bank):

- New accounts can be created. Note: when new accounts are added for the first time, they are initialized with an opening balance (given in the input file). It is guaranteed that newly added accounts will always have a positive balance.
    - Each customer can open <u>one</u> bank account at Mocs Bank
- A customer can request the balance on the account.
- A customer can deposit any amount into his account.
- A customer can withdraw money from their account (if the money is available).
- A customer can transfer amount from their account into another account. In this case the balance of the former customer will be reduced, while the balance of the latter customer will increased accordingly.
- A customer can close their account.
- A list of daily transactions can be printed.

You will use File I/O to read input from a file and then print the output to a file. To be clear, you will read COMMANDS from an input file. Example commands are OPENACCOUNT, DEPOSIT, WITHDRAW, TRANSFER, FINDACCOUNT, CLOSEACCOUNT, TRANSACTIONREPORT etc. Then, depending on the command, you will either add a new account, deposit amount, withdraw amount, transfer amount, close account, etc.

<u>But instead of printing to the console window (screen), you will need to print it to an output file.</u>

**Sample input and output files have been provided for you on the website.*

Input File Specifications

You will read in input from a file, "**`MocsBank.in`**". Have this AUTOMATED. Do not ask the user to enter "**`MocsBank.in`**". You should read in this automatically.

Note: a **`*.in`** file is just a regular text file. Also, a `*.out` file is a regular text file. Do not be scared by this. Instead of calling the input and output file as input.txt and output.txt, it is better for those files to have a good name. Therefore, the input file will be called `MocsBank.in` and the output file will be called `MocsBank.out`.

The first line of the file will be an integer representing the maximum number of accounts in the bank. The second line of the file will be an integer representing the maximum possible number of recorded transactions for any given day of the simulation. The third line of the file will be an integer, d, representing the number of days for the simulation. For each day, there will be an integer k on the first line, followed by k commands, with each command on a new line and followed by appropriate data (and this relevant data will be on the same line as the command).

Output File Specifications

Your program must output to a file, called "**`MocsBank.out`**". **You must follow the program specifications exactly.** You will lose points for formatting errors and spelling.

Sample and Output Files

A sample input file, with corresponding output file, has been provided to you within the zip file you downloaded from Canvas for this assignment.

NOTE: These files do NOT test every possible scenario. That is the job of the programmer to come think up the possible cases and test for them accordingly. You can be sure that your program will be graded with very large input files, which are intended to, ideally, test all possible cases. It is recommended that you spend time thinking of various test cases and build your own input file for testing purposes.

*****WARNING*****

Your program MUST adhere to the EXACT format shown in the sample output file (spacing capitalization, use of dollar signs, periods, punctuation, etc). The output files are very large, dictating the use of text comparison programs to compare your output to the correct output. If, for example, you have two spaces between words in the output, when there should be only one space, this will show up as an error even though you may have the program logic correct. This will have to manually corrected by me in order to effectively grade your program, and these corrections will result in deductions, which is why this is being explained in detail. Minimum deduction will be 10% of the grade, as I will be forced to go to text editing of your program in order to give you an accurate grade. Again, your output must adhere exactly to the formatting depicted in the sample output. Use of a text comparison program makes this very easy for you. While I recommend the downloadable version of WinMerge for PC users, there are numerous online text comparison applications, including Mergely, which many of you used previously.

Implementation

For this program, you will have **four** files. One file will contain main, and the other three files will be classes: `FSCmember`, `MocsBankAccount`, and `MocsBankTransaction`. The class UML diagrams are on the last page of this write-up.

The first two lines of the input file are as follows:
Line 1: the maximum number of accounts that can be in the bank
Line 2: the maximum number of transactions in a given day

You must read these values into appropriate variables (`maxAccounts` and `maxTransactions`).

Next, you will use these new values to make two new arrays:

1. An array of MocsBankAccount object ***references***
2. An array of MocsBankTransaction object ***references***
   - *See pages 72-78 of Chapter 9 slides (from CSC 2290 and included as part of Module 1 on Canvas) for help on array of object references*

To help you, here is the code to do this:

```
MocsBankAccount[] accounts = new MocsBankAccount [maxAccounts];
MocsBankTransaction[] transactions = new MocsBankTransaction[maxTransactions];
```

What do these lines do? They create an array of **references**. Note: each reference has a default value of **null**. Until now, we have NOT created any objects. For example, `MocsBankAccount` objects are only created when we see the command OPENACCOUNT. At that time, a new `MocsBankAccount` object will be created and the reference for that object will be saved in the appropriate location of the array.

Commands to Process (from Input File)

The commands you will have to implement are as follows:

- OPENACCOUNT – Makes a new account which is added to the bank. The command will be followed by the following information all on the same line: ID, an integer representing the ID number of the student (also the account number for this account); firstName, the first name of the customer/student; lastName, the last name of the customer/student; and openingBalance, the initial amount of for this account.

When you read the OPENACCOUNT command, you must make a new object of type `MocsBankAccount`. Next, you must scan the additional information (listed above) and save this information into the correct data members of the new object.

- Note that the firstName, lastName, and ID of the student must be saved into an object of type `FSCmember`. How do you do this? Look at the UML diagram for `MocsBankAccount`. One of the data members is called `customer`, and the data type for `customer` is `FSCmember`. The variable `customer` is a reference variable, but as of now, it is simply pointing to null. Therefore, inside the constructor of `MocsBankAccount`, you should make a new `FSCmember` object and save that reference into the variable student.
- Note also that there is a variable called ID inside the `FSCmember` object, and there is also a variable called `accountNumber` inside the `MocsBankAccount` object. These two variables should have the SAME value…the ID of the customer.
- Each account will be unique. Meaning, the input file is guaranteed to not have duplicate account added to the bank via the OPENACCOUNT command. Finally, you must <u>save the reference of this object</u> inside the appropriate index of the student array.

***<u>Note</u>**: this array **<u>must</u>** stay in <u>sorted order based on the accountNumber</u>. So the account with the smallest accountNumber value will be at index 0, the account with the next smallest at index 1, and so on. Therefore, when you save the object reference into the array, <u>you must first find the correct sorted index</u>. After you find the correct insertion location, if there is no object at that location, then you can easily save the object reference at this index. BUT, IF there is an object at the index that you find, you must **<u>SHIFT</u>** that object, **AND** all objects to the right of it, one space to the right. This is how you make a new space in the array for the new object.

Example: if your array already has 6 account object references, from index 0 to index 5. And now you want to add a new account object reference at index 4 (based on the sorted location). Before you can add the new account object reference at index 4, you must SHIFT the account object references at index 4 and index 5 to index 5 and index 6. Basically you need to move them one position over to the right. Now, you have an empty spot at index 4 for the new account object reference.

HINT: Draw this example out on a piece of paper to visualize the shifting

- PRINTBALANCE– This command will be followed by an integer on the same line. This integer represents the account number of the account you must search for. You must perform a **<u>Binary Search</u>** on your array of `MocsBankAccount` object references. If the account is found, you should print the required information for the account. (see output).

  * There are many possibilities (no accounts at all, this account not found, etc.). See output file for exact scenarios.

  **We will check your code to confirm you are using Binary Search. You will only earn the points for PRINTBALANCE if you use Binary Search.

- DEPOSIT – This command will be followed by an integer and a double on the same line. This integer represents the account number of the account and the double value represents the amount of money you need to deposit in the account.
  - You must search for the correct account in the accounts array and then deposit the money if the account as found.
  - There are many possibilities (no accounts at all, this account not found, etc.). See output file for exact scenarios.

- WITHDRAW – This command will be followed by an integer and a double on the same line. This integer represents the account number of the account and the double value represents the amount of money you need to withdraw from the account.
  - You must search for the correct account in the accounts array and then withrdaw the money if the account as found.
  - There are many possibilities (no accounts at all, this account not found, etc.). See output file for exact scenarios.

- TRANSFER – This command will be followed by two integers and a double on the same line. The first integer represents the account number of the account from which money will be deducted, while the second integer represents the account number to which money will be added. The double value represents the amount of money to be transferred.

  See output for printing examples.

- CLOSEACCOUNT – This command will be followed by an integer. This integer represents the account number to be deleted.

- You must search for the account and delete it from the system. Here, we will delete simply by saving a difference object reference in its place (see below).

*__Note__: this array __must__ stay in sorted order based on the accountNumber. Again, you must take care of shifting!

Example: if your array already has 6 account object references, from index 0 to index 5. And now you want to remove an account object reference at index 3 (based on the sorted location). To remove account object reference at index 3, you must SHIFT the account object references at index 4 to index 3, index 5 to index 4, and index 6 to index 5. Basically you need to move them one position over to the left

See output for printing examples.

- TRANSACTIONREPORT – This command will have no other information on the line. When this command is read, a report of all transactions, for that given day, will be printed to the output file. Please refer to sample output for exact specifications.

  NOTE: When a new day beings, the transaction array must be empty and the variable numTransactions should be reset to zero.

  *NOTE: Not all commands will result in a transaction. Transactions occur only with commands that change the balance (OPENACCOUNT, DEPOSIT, WITHDRAW, TRANSFER, and CLOSEACCOUNT).

Grading Details

Your program will be graded upon the following criteria:

1) Adhering to the implementation specifications listed on this write-up.
2) Your algorithmic design.
3) Correctness.
4) **Use of Classes, Objects, and Arrays of Objects. If your program is missing these elements, you will get a zero. Period.**
5) The frequency and utility of the comments in the code, as well as the use of white space for easy readability. (We're not kidding here. If your code is poorly commented and spaced and works perfectly, you could earn as low as 80-85% on it.)
6) Compatibility to the **newest version** of **IntelliJ IDEA**. (If your program does not compile in IntelliJ, you will get a large deduction from your grade.)

7) Your program should include a header comment with the following information: your name, **email**, course number, section number, assignment title, and date.
8) Your output MUST adhere to the EXACT output format shown in the sample output file.

Deliverables

You should submit a zip file with FOUR files inside:

1. `FSCmember.java`
2. `MocsBankAccount.java`
3. `MocsBankTransaction.java`
4. `MocsBank.java` (this is your main program)

***These four files should all be INSIDE the same package called **mocsbank**. If they are not in this specific package, you will lose points.

NOTE: your name, ID, section number AND EMAIL should be included as comments in all files!

Suggestions:

- Read AND fully understand this document BEFORE starting the program!
- Next, make the project in IntelliJ. The name will be `MocsBank`, and the name you give to the package should be `mocsbank` (all lowercase).
- This is your main program that will read the commands from the file, process the commands, and print to the output file.
- Next, add the three classes to the project:
  - `FSCstudent`, `MocsBankAccount`, and `MocsBankTransaction`
- So make those three classes, with all the data members, constructors, and accessor/mutator methods.
- Use the `IOexample.java` program to help you create your main to read from a file and write to a file.
- Note: the `IOexample.java` program uses a while loop, because that program stopped execution after reading the QUIT command. However, for your program, you will read a specific number of commands, over a specific number of days. You will need TWO for loops to do this…one for loop to iterate over the number of days, and a second for loop to iterate over the number of commands for each day.
- Finally, one by one, implement the commands (OPENACCOUNT, PRINTBALANCE, DEPOSIT, WITHDRAW, etc).

Hope this helps. Final Suggestion: Start early! ☺

UML Diagrams for Three Classes

FSCmember
Data Members private int ID private String firstName private String lastName
Operations/Methods FSCmember() // one or more Constructors ALL getter and setter methods

MocsBankTransaction
Data Members private String transactionType private int accountNumber1 private int accountNumber2 private double amount private double balanceBefore private double balanceAfter private static int numTransactions
Operations/Methods MocsBankTransaction() // one or more Constructors ALL getter and setter methods You should also have a method to increment the numTransactions variable for each time a transaction is added to the transactions array

MocsBankAccount
Data Members private int accountNumber private FSCmember customer private double accountBalance private static int numAccounts
Operations/Methods MocsBankAccount() // one or more Constructors ALL getter and setter methods You should have also have two additional methods: One to increment numAccounts when you create a new account, and one method to decrement numAccounts when you close an account.

NOTE:
Look at the `MocsBankAccount` class. This class has four data members. One of the members is called customer. This variable is of type `FSCmember`. What does this mean? This means that when you create a new object of type `MocsBankAccount` and you are wanting to save the customer data, you must do one more thing first! Inside the constructor for `MocsBankAccount`, you must create a NEW object of type `FSCmember` and then save the reference for this object into the variable customer. Now, you will be able to save the data into the correct variables.