CST8132 310 - Lab 5: Bank Simulator

In this assignment, you will complete an object-oriented version of the software managing bank customers' Bank Accounts.

Note you may **NOT** use the ArrayList class in Java for this assignment.

We will use this assignment as the base for many following assignments. You must use the classes, methods and non-private data member names as listed in this assignment, but you may add additional helper values long as your solution follows all Object-Oriented Principles.

You will receive partially complete code, which you will need to complete to finish the application.

Requirements

1. Our Bank Simulator will consist of information about customer Bank Accounts. I have simplified the requirements for this assignment to set boundaries on your time.
2. The Bank Simulator consists of a dynamically allocated array of **BankAccounts** of two types – either Savings accounts or Chequing accounts. A bank account consists of an **account number (between 100 and 999),** customer and a double balance. A Savings account has two additional fields - a rate of interest and a minimum balance amount (at the end of the month, the customer is given interest based on their account balance as long as the account balance is more than the minimum balance amount). A Chequing account has one additional field compared to a Bank Account – that is the monthly fee that is deducted.
3. The Simulator will give the user – presumably the user is a bank employee – a menu of choices of actions (ie this is not simulating a customer using a bank machine). These actions will include
    - adding a new Bank Account
    - displaying the information for a specific bank account
    - updating the balance for a specific bank account (withdrawal or deposit)
    - running the monthly update on all accounts
4. Your program should build a Bank "database" – for now this is a dynamically allocated array. The required size of this array is unknown – but we will implement it as a dynamically allocated array of objects of type Bank Account (instantiated with either a Savings Account or a Chequing Account object). How you handle the size of this array is an important part of the assignment.
5. **Your program should handle ALL errors.** It should never stop executing without giving a message about why it is doing so. **Do NOT ever use the command `System.exit()`** – instead you should issue an appropriate message and return to the calling method (ultimately back to method main). If you are having the user enter data – **do not continue until proper data has been entered.** Exit gracefully with messages. This means all your methods which handle data should return a boolean – true if the data was ok – false if it was not – and all calls to these methods should check the return value.
6. You do NOT need to resize the array once you have made it should it become full. You should not allow more entries than you have space for – an error message is sufficient in this situation.
7. **All Java conventions MUST be followed** – ie capital and lower case letters when appropriate, etc. All data members should be private, unless otherwise specified.

8. Do not use ONLY the get/set design pattern in this assignment. In order to emphasize OOP, **all processing of data members in a class to be handled in the class** – and the get/set pattern allows this to be broken. See description for methods for each class below which will help this requirement.

Project: Lab 5

Package: lab5

Class: MainMenu

- **public static void main(String[] args)**
  - Display the main menu of options to the end user (bank employee)
  - Write a **switch** statement to handle user input from **Scanner**.
  - Menu options should be valid if the user enters either the lower-case letter, upper-case letter, or the entire key word of the menu item.
    - Ex: 'a', 'A', 'add', 'Add' should all be considered valid input for the "a: Add new account" menu item.
  - Each menu option should invoke a method in the **Bank** class, however until you code each of these, simply output the method call as a string to test the menu.
    - Ex:

```
case 'a':
      System.out.println("bank.addBankAccount();");
```

  - **Handles any exceptions** that may occur based on invalid user input
- Write this class first and test it thoroughly for invalid entries before proceeding.
- You may add additional helper methods to this class.
- This class should have **no properties**; the only variables you use should be locally declared.

Class: InsufficientFundsException

- Do not edit this class. Use it exactly as provided.
- Use this class as a template for the **OverdrawnAccountException** class.

Class: OverdrawnAccountException

- Using the **InsufficentFundsException** class provided, create this new type of **Exception**.
- **Update the messages** appropriately for the case when a monthly fee withdrawal overdraws the account in question.

Class: Customer

- Implements interface **Comparable<Customer>**
- **4** private instance variables:
  - Strings: **firstName, lastName, email**
  - long: **phoneNum**
- **public Customer(String firstName, String lastName, String email, long phoneNum)**
  - Initializes the properties of the Customer with the parameter values
- **public String getName()**
  - returns the firstName and lastName properties, concatenated with a space in between.
  - Example: "`Angela Giddings`"

- **public String toString()**
  - returns all of the customer information, formatted on 3 lines:
  - **Example ouput:**

```
Name:  Angela Giddings
Email: giddina@algonquincollege.com
Phone: 1231231234
```

- **public int compareTo(Customer customer)**
  - **Returns -1, 0, or 1** depending on how this customer name compares to the name of the customer that is passed as a parameter.
  - **Hint:** you can return the result of using the **String compareTo(String s)** method, comparing **this.getName()** to **customer.getName()**.
  - **Hint:** If this method returns **0** when two names are equal, and not zero for all names which differ, it will work for your program.

Abstract Class: BankAccount

- **3** protected instance variables:
  - int: **accNumber**
  - double: **balance**
  - String: **accType**
  - Customer: **accHolder**
- There is **no constructor** defined for this class. The Java compiler will implicitly invoke the default no-args constructor.
- This class should use the **java.util.Scanner** and **java.text.DecimalFormat** classes.
- **public boolean addBankAccount()**
  - Prompts the user for property values to initialize the bank account
  - Inputs the values from the user, handles any exceptions that may arise, and assigns the input to the appropriate instance variable
  - All account numbers should be unique. Use the static **searchAccounts** method of the **Bank** class to check to see if the new account number entered by the user is unique. If not, be sure to handle this scenario.
  - Prompts the user for property values for the Customer, and then initializes the accHolder property by instantiating a new Customer with this user input.
    - **Example code:**

```
accHolder = new Customer(fName, lName, email, pNum);
```

  - Returns **true** if all values successfully initialized with user input, or **false** if there are exceptions that haven't been handled.
- **public String toString()**
  - returns all of the bank information formatted on **2 lines**, as well as all of the customer information by invoking the **toString()** method of the **Customer** class.
  - **Example output:**

```
Chequing Account 101
Balance: $100.21
Name:  Angela Giddings
Email: giddina@algonquincollege.com
Phone: 1231231234
```

- **public void deposit(double amount)**
    - Increase the balance by the amount in the parameter.
- **public void withdraw(double amount) throws InsufficientFundsException**
    - If the amount requested is strictly greater than the balance, throw a new **InsufficientFundsException**.
        - You can either use the default constructor, or create your own customer message to pass as a parameter to the overloaded constructor.
    - Otherwise, if the amount requested is less than or equal to the balance, decrease the balance by the amount in the parameter.
- **public abstract void calculateAndUpdateBalance();**
    - An abstract method to force the concrete child classes to implement this method in their own custom ways.
- **public int getAccNumber()**
    - Return the value of the **accNumber** instance variable.

Class: ChequingAccount

- This is a sub-class (or child class) of the abstract class BankAccount.
- This class should use the **java.util.Scanner** and **java.text.DecimalFormat** classes.
- 1 protected instance variable:
    - double: **monthlyFee**
- **public boolean addBankAccount()**
    - This method will first invoke the **addBankAccount** method of its superclass.
    - If the **super.addBankAccount()** method call returns **false**, this method should also return **false**.
    - If the **BankAccount** method is true, then this method should continue to prompt the user for the monthly fee, accepting values **between $5.00 and $10.00 (inclusive)**.
    - This method must handle any exceptions that it may encounter while inputting information from the user and return **false** in those scenarios.
    - Return **true** if no exceptions are encountered, and the input is assigned to the **monthlyFee** property successfully.
- **public void calculateAndUpdateBalance()**
    - Withdraw the **monthlyFee** from the **balance**.
    - If the resulting balance is less than $0.00, a new **OverdrawnAccountException** should be thrown. The exception should be caught and handled within this method.
    - Do not use the **getBalance()** method from the **BankAccount** class. Access the **balance** property from the superclass, either implicitly or explicitly.
- **public String toString()**
    - Begin by calling the **toString** method of the **BankAccount** class (or superclass).
    - Append the monthly fee information on a new line.
    - **Example output:**

```
Chequing Account 101
Balance: $100.21
Name:  Angela Giddings
Email: giddina@algonquincollege.com
Phone: 1231231234
```

```
Monthly fee: $7.55
```

Class: SavingsAccount

- This is a sub-class (or child class) of the abstract class BankAccount.
- This class should use the **java.util.Scanner** and **java.text.DecimalFormat** classes.
- **2** protected instance variables:
  - double: **monthlyInterestRate, minBalance**
- **public boolean addBankAccount()**
  - This method will first invoke the **addBankAccount** method of its superclass.
  - If the **super.addBankAccount()** method call returns **false**, this method should also return **false**.
  - If the **BankAccount** method is true, then this method should continue to prompt the user for the monthly interest rate, accepting values **between 0.00 and 1.00 (inclusive)**.
  - Next the user should be prompted for the minimum balance. The minimum balance should be anywhere between **$5.00 and $100.00 (inclusive)**.
  - This method must handle any exceptions that it may encounter while inputting information from the user and return **false** in those scenarios.
  - Return **true** if no exceptions are encountered and the input is assigned to the **minBalance** and **interestRate** properties successfully.
- **public void calculateAndUpdateBalance()**
  - If the value of the **balance** property is greater than the **minBalance** property, add the accumulated interest to the balance. Otherwise, do nothing.
  - Do not use the **getBalance()** method from the **BankAccount** class. Access the **balance** property from the superclass, either implicitly or explicitly.
- **public String toString()**
  - Begin by calling the **toString** method of the **BankAccount** class (or superclass).
  - Append the monthly interest rate and minimum balance information on **2** new lines.
  - **Example output:**

```
Savings Account 102
Balance: $100.21
Name:  Angela Giddings
Email: giddina@algonquincollege.com
Phone: 1231231234
Interest Rate: 0.45%
Minimum Balance: $50.00
```

Class: Bank

- The **Bank** class should handle all of the operations chosen by the user in the **MainMenu** class.
- There should be a method for each of the **MainMenu** options, along with some “helper” methods.
- **1** private instance variable:
  - String: **bankName**
- **3** protected static (class) variables:
  - a Scanner object
  - BankAccount[] : **accounts**

- int: **numAccounts = 0, bankSize = 100**

- **public Bank(String bankName)**
  - Initializes the **bankName** variable with the parameter value.
- **public Bank(String bankName, int bankSize)**
  - Re-initializes the **bankSize** variable with the parameter value (overrides default value of 100).
  - Initializes the **bankName** variable with the corresponding parameter value.
- **public void addBankAccount()**
  - If the number of accounts is greater than or equal to the bank size, output an error message (using your own words) to the user and return.
    - **Example code:**

```
if (numAccounts >= bankSize)
      System.err.println("Error message");
      return;
```

  - Prompt the user to enter the account details for the account number (using the **numAccounts** variable in your ouput).
  - Next prompt them to enter a 'c' to create a new Chequing Account, or 's' to create a new Savings Account.
  - Values of 'c', 'C', 'chequing' or 'Chequing' should all be processed as valid entries to create a new Chequing Account.
  - Values of 's', 'S', 'savings' or 'Savings should all be processed as valid entries to create a new Savings Account.
  - Handle other input appropriately.
  - Depending on the user input, add a new account to the array at the numAccounts position.
  - Next, call the addBankAccount method of the new account. If it returns true, increase the number of accounts.
  - **Example code:**

```
if (accType.equalsIgnoreCase('c'))
      accounts[numAccounts] = new ChequingAccount();
else if (accType.equalsIgnoreCase('s'))
      accounts[numAccounts] = new SavingsAccount();
else
      // handle this possible scenario

if (accounts[numAccounts].addBankAccount())
      numAccounts++;
else
      // handle this scenario here
      // output error message to user
```

- **public void displayAccount()**
  - Call **findAccount()** to search for an account index number.
  - Handle any possible exceptions, including an invalid index number (such as -1).
  - Implicitly call the **toString()** method of the account to print the details.
  - **Example code:**

        - `System.out.println( accounts[ findAccount() ] );`
- **public void updateAccount()**
    - Call **findAccount()** to search for an account index number.
    - Handle any possible exceptions, including an invalid index number (such as -1).
    - Prompt the user to enter an **amount** which will be a **double** value and handle any possible exceptions.
    - If the amount is negative, call the **withdraw** method for the account.
    - If the amount is positive, call the **deposit** method for the account.
    - **Handle every scenario.**
- **public void monthlyUpdate()**
    - Loop through all accounts in the array, from 0 to the number of accounts, calling the **calculateAndUpdateBalance** method of each account.
- **public int findAccount()**
    - Prompt the user for an account number.
    - Use the **Scanner** to input the account number to find and assign it to a local variable.
    - Handle any exceptions that might occur as a result of the user input.
    - Return the result of calling **searchAccounts** with the account number from the user.
    - **Example code:**
        - `return searchAccounts(accToFind);`
- **public static int searchAccounts(int accToFind)**
    - Loop through the accounts array, from 0 to the number of accounts that have been created and compare the **accToFind** parameter value to the account number of each **BankAccount** object, using the **getAccNumber** method of the **BankAccount** class.
    - If the two values are equal, return the index of the classic for loop.
    - Following the loop, if no values matched, return **-1** to indicate that nothing was found.

Submission Requirements

1. Create a new folder named **CST8132_<LabSectionNo>_<LastName>_<FirstName>_Lab5**.
2. In this folder, include (at the very least):
    a. MainMenu.java
    b. Bank.java
    c. Customer.java
    d. BankAccount.java
    e. ChequingAccount.java
    f. SavingsAccount.java
    g. InsufficientFundsException.java
    h. OverdrawnAccountException.java
3. Test Plan (.docx or .xlsx files only).
4. **ZIP** your folder:
    a. **Right-click** on the folder to open the context menu.
    b. Select **Send to > Compressed (zipped) folder**.
5. This should produce a **ZIP** archive file named **CST8132_<LabSectionNo>_<LastName>_<FirstName>_Lab5.zip**.
6. Upload the **ZIP** file to Blackboard.