Lab Assignment - 3

Copyright Policy

Academic Integrity

What is allowed:

- Looking up **syntax** related to Java
- You can refer the code and classwork created in this course

What is **NOT** allowed:

- Searching for partial or full solutions of the main problem description
- Communication with others, either inside or outside the class
- Sharing of resources, including but not limited to code, files, links, computers, etc.

Instructions:

- This assignment is to be done individually.
- Besides implementing the required functionality submissions are required to use the correct coding conventions used in class, professional organization of the code, alignment, clarity of names is all going to be part of the evaluation.

Submission Checklist:

- Once you are done, write your name and student number at the top of each **.java** file.
- Compress your entire project folder into a .zip file. Your .zip file name must be **YourFirstName_Section#_Lab3.zip** such as John_17_Lab3. **Please don't submit .rar or .7zip file.**
- Upload **the zip file** in the submission folder in eCentennial under **Assessments menu -> Assignments** in appropriate folder.

Task:

In IntelliJ Idea, create a new project (Java Application) named **YourFirstName_Section#_Lab3** (where YourFirstName is your first name, such as John_17_Lab3) to accomplish the following task:

The application will allow the user to perform various banking operations such as deposit and withdrawal of money from the bank account. The bank offers two types of accounts: savings account and chequing account.

Consumer class:

This class will represent following consumer information. Both the variables must be private members.

- ***id*** – consumer ID e.g., S102
- ***name*** – name of the consumer

Constructor: Create parameterized constructor to accept necessary parameters and assign to each of the variables.

Methods:

- ***Setter & Getter***: Create necessary setter and getter methods for variables.
- **toString() :** Create toString() function to return a string containing values of variables in appropriate format.

Account class:

This class inherits from Consumer class. It will represent following account information.

- **accountNum** – *protected*- account number e.g., 222210212

Constructor: Create parameterized constructor to accept necessary parameters for the variables of account class and the parameters of the super class.

Methods:

- ***Setter & Getter***: Create necessary setter and getter methods for variables.
- ***withdraw(float amount)*** – this is an **abstract method** which will accept amount to withdraw as parameter
- ***deposit(float amount)*** – this is an **abstract method** which will accept amount to deposit as parameter
- **toString() :** Create toString() function to return a string containing values of variables of account class as well as super class.

SavingsAccount class:

This class inherits from Account class. It will represent following account information. The variable must be having private access.

- ***balance –*** *private - float*

Constructor: Create parameterized constructor to accept necessary parameters for variables of SavingsAccount class and the parameters of the super class. If the balance parameter value is not

provided, use the default value of 0.0. This class will also implement the abstract methods inherited from the super class.

Methods:

- ***Setter & Getter***: Create necessary setter and getter methods for variables.
- ***withdraw(amount : float)***
    - This is an implementation of abstract method inherited from super class. The method will accept amount to withdraw as parameter and try to deduct it from the balance.
    - The method must check if there is sufficient balance in the account; if not, display appropriate message to the user.
    - Also, check if after the amount will be deducted from the balance, $3000 will remain in the balance. If not, display appropriate message to the operation.
    - If all the conditions are met, deduct the amount from the balance and display the updated balance.

- ***deposit(amount : float)***
    - This is an implementation of abstract method inherited from super class. The method will accept amount to deposit as parameter and try to add it to the balance.
    - The method must check if the given amount is not less than zero (0); if not, display appropriate message to the user.
    - If all the conditions are met, add the amount to the balance and display the updated balance.

- ***toString()*** **:** Create toString() function to return a string containing values of variables of SavingsAccount class as well as super class.

ChequingAccount class:

This class inherits from Account class. It will represent following account information. The variable must be having private access.

- ***balance –*** *private - float*

Constructor: Create parameterized constructor to accept necessary parameters for variables of ChequingAccount class and the parameters of the super class. If the balance parameter value is not provided, use the default value of 0.0. This class will also implement the abstract methods inherited from the super class.

Methods:

- ***Setter & Getter***: Create necessary setter and getter methods for variables.
- ***withdraw(amount : float)***

- This is an implementation of abstract method inherited from super class. The method will accept amount to withdraw as parameter and try to deduct it from the balance.
- The ChequingAccount holders can deduct up to $2000 more than the balance. For instance, if the balance is 5000, the consumer can withdraw up to $7000 from the account.
- The method must check if the amount to be deducted is less than (2000 + balance), if not, display appropriate message to the user.
- If all the conditions are met, deduct the amount from the balance and display the updated balance.

- ***deposit(amount : float)***
  - This is an implementation of abstract method inherited from super class. The method will accept amount to deposit as parameter and try to add it to the balance.
  - The method must check if the given amount is not less than zero (0); if not, display appropriate message to the user.
  - If all the conditions are met, add the amount to the balance and display the updated balance.

- ***toString()*** **:** Create toString() function to return a string containing values of variables of ChequingAccount class as well as super class.

Bank class:

This class represents the bank that consist of several accounts. Create the following members of the class:

accountList[] – ArrayList<> - consist of objects of accounts.

Create a static block within the Bank class which will create at least 3 objects of SavingAccount class & 3 objects of ChequingAccount class with hardcoded values. Add the created account objects to the accountList variable. Use the following code snippet for the Constructor code.

```
static
  {
     accountList = new ArrayList<Account>();
     accountList.add(new SavingsAccount("S101", "James Finch", 222210212, 3400.90));
     accountList.add(new SavingsAccount("S102", "Kell Forest", 222214500, 42520.32));
     accountList.add(new SavingsAccount("S103", "Amy Stone", 222212000, 8273.45));
     accountList.add(new ChequingAccount("C104", "Kaitlin Ross", 333315002, 91230.45));
     accountList.add(new ChequingAccount("C105", "Adem First", 333303019, 43987.67));
     accountList.add(new ChequingAccount("C106", "John Doe", 333358927, 34829.76));
  }
```

Methods:

- ***toString()*** **:** Create toString() function that will return a string with all the account details for each account in accountList.

BankTest Class:

To test the above-mentioned classes, create a class named BankTest.java. This class will contain the main() method that should perform the following operations:

- Create at least 2 objects of Bank class
- For each of the objects,
  - call the withdraw() function for Savings and Chequing account types with different parameters and/or values to demonstrate that the methods are performing all the validations.
  - call the deposit() function for Savings and Chequing account types with different parameters and/or values to demonstrate that the methods are performing all the validations.
  - Use hard-coded values for supplying to the function parameters. You don't have to read input from user.