TCSS143
Fundamentals of Object-Oriented Programming-Theory and Application
Programming Assignment 2

Due: See Programming Assignment 2 link on Canvas.

The purpose of this programming project is to demonstrate a fundamental understanding of inheritance, polymorphism, and interfaces.

REQUIREMENTS

You will submit all source code files (**4 in all**) as a **single zipped file named "Programming2.zip**" through the Programming Assignment 2 Submission link on Canvas.

Not only will you be graded on program correctness (Program executes correctly, proper use of methods, classes, inheritance, etc.) but also, good programming documentation techniques including javadoc, proper indentation, correct locations of braces, meaningful identifier names, preface instance fields with my and method parameter names with the, javadoc prior to methods, specific comments on complex code, etc. NOTE: Documentation is not "always" part of the grading process, but should always be properly done in the event it will be graded and will always be 20% of the overall programming assignment grade.

Do **NOT** use any constructs that have not been presented in the course thus far.

DETAILS

You will create 3 classes related to banking operations for customers, and an Interface that is implemented by all either directly or inherited. Keep in mind, one major purpose of inheritance is to eliminate redundant code. For this reason you should take full advantage of super class code or methods when developing your subclasses.

You will create a basic bank account class "BankAccount.java," a more specific savings account class "SavingsAccount.java," an Interface for accessing and changing a customer's name "NamedAccount.java," a safe deposit box account class for establishing a safe deposit box "SafeDepositBoxAccount.java" which implements the NamedAccount interface, and finally, you will create toString() methods for your classes and modify the BankAccount class to also implement the NamedAccount interface. **Always remember to use final constants in place of any literal constants, sometimes known as "Magic Numbers," (anything other than 0, 1, 2, & -1) you would normally use in your code!** Greater detail of each class follows:

1. BankAccount.java

public class BankAccount {

Represents a basic bank account.

finals: As mentioned above, declare and use in place of literal constants as needed.

Fields: 3 private fields for the customer's name, account balance, and the interest rate (String, double, double and use correct naming conventions). 2 protected fields (the SavingsAccount subclass needs direct access) for myMonthlyWithdrawCount and myMonthlyServiceCharges (int, double).

Methods:

public BankAccount(final String theNameOfOwner, final double theInterestRate)
The constructor initializes the name and interest rate to the values of the passed parameters (**taking care that the interest rate is a legitimate value**) and sets the remaining fields to 0.

public double getBalance( )
Returns the current balance

public boolean processDeposit(final double theAmount)
Adds the specified amount of money to the calling BankAccount object. However, if the "theAmount" of money is zero or negative, the deposit fails (it's not possible to withdraw money by depositing a negative amount), the account balance is unchanged, and the method should return false. Otherwise, if the amount of money is positive, the deposit succeeds and the method should return true.

public boolean processWithdrawal(final double theAmount)
Subtracts the specified amount of money from the calling BankAccount object. However, if the amount of money is not a positive value or is more than the current balance of the account, the withdraw fails, the account balance is unchanged, and the method should return false. Otherwise, the withdrawal succeeds, the record of successful withdrawals made in the current month is incremented, and the method should return true.

public double calculateInterest( )
Returns the monthly interest based on the APR (Annual Percentage Rate). Which is:

$$myBalance * \frac{myInterestRate}{12.0}$$

Do NOT use APY (Annual Percentage Yield). APY is derived when the above (APR) is compounded (daily, monthly, quarterly, etc.) during the course of a year.

public void performMonthlyProcess( )
Subtracts all monthly service charges from the balance, adds the monthly interest accrued to the balance through a call to calculateInterest(), and resets the withdrawal count and service charge amount to 0. Finally, the balance should **not** be set to anything less than 0.

2. SavingsAccount.java

This class represents a more specialized type of bank account, a savings account. In addition to the properties of a bank account, a savings account should record an active/inactive status property (as a private field). A savings account is considered active if the account balance is at least $25, otherwise the account is considered inactive. Withdrawals cannot be successfully completed from an inactive account. The constructor should initialize the owner and interest rate of the SavingsAccount object to the specified values.

public class SavingsAccount extends BankAccount {
finals: As mentioned above, declare and use in place of literal constants as needed.
Fields: A single private boolean field named myStatusIsActive is all that is necessary.

Methods:
public SavingsAccount(final String theNameOfOwner, final double theInterestRate)
Makes an appropriate call to SavingsAccount's super class constructor and also initializes the myStatusIsActive field to an appropriate value.

public boolean processDeposit(final double theAmount)
Adds "theAmount" to the SavingsAccount object's balance following the same rules as the deposit method of the BankAccount class. In addition, if the account balance after a successful deposit is at least $25 dollars, the account status should be active.

public boolean processWithdrawal(final double theAmount)
Subtracts "theAmount" from the SavingsAccount object's balance, as with the processWithdrawal method of the BankAccount class. If the account is inactive, the withdrawal should fail. If the withdrawal succeeds and the resulting balance is less than $25, the account status should become inactive after the money is withdrawn. Finally, the 5th successful withdrawal and any subsequent successful withdrawals each incur a $1 service charge, to be deducted from the account at the end of the month by the performMonthlyProcess method as described next.

public void performMonthlyProcess( )
This method performs the same monthly processing as is done in the super class but also, updates the myStatusIsActive value based on the balance, i.e. it is set to true if the balance is greater than or equal to $25.00 and false otherwise.

3. NamedAccount.java

```
public interface NamedAccount {
  String getAccountHolderName( );
  void setAccountHolderName(final String theNewName);
  }
```

Any classes that implement this interface **"must"** implement the getAccountHolderName and setAccountHolderName methods. The getAccountHolderName method will return the name of the account holder, as appropriate for the implementing class. The setAccountHolderName method will permit the account holder or owner to be changed.

4. SafeDepositBoxAccount.java

This class represents the rental of a bank's safe deposit box to a customer. Because banks do not know what is stored in the safe deposit box, the only information that the class will keep track of is the name of the person who owns the safe deposit box account (as a private field).

You are to write the code that makes up the methods of this class (Remember, you are implementing NamedAccount). Also, don't forget the name or the owner field.

```
public class SafeDepositBoxAccount implements NamedAccount {

    /* You write the code */
}
```

Once you have completed these three classes and the interface, **update the BankAccount class so it can properly implement the NamedAccount interface.**

Next, add a toString() method to each of your three classes. They should return a String which includes the name of the class and then a comma separated list of label/value pairs for each field of the class. This list should be contained in square brackets.

(**As a hint:** a toString() method in a sub-class can call the toString() method of a super class. Also, the String.format() formats a String just as the printf format String, specifiers, and arguments are formatted). Remember, whenever possible, always use methods of the super class to reduce redundancy and also, to maintain the integrity of the object and its fields.

You will use the Driver program "AccountTester.java" to test your classes (Save all classes in the same folder and compile the driver program before execution).

You will submit all source code files (**4 in all**) as a **single zipped file named "Programming2.zip**" through the Programming Assignment 2 Submission link on Canvas (Only zip the files. Do Not zip any folders).

See the sample run on the following page for detailed output. About 1/3 of the way through the output, I had to remove leading spaces on the lines that begin with "number of withdrawals this month…" due to Word's margins. In my formatted Strings I inserted both newline and tab characters which should start these lines with 8 spaces.

Sample Run:

```
----jGRASP exec: java AccountTester

Testing methods of class BankAccount
Creating a bank account for John Doe at 5.00%
BankAccount[owner: John Doe, balance: 0.00, interest rate: 0.05,
        number of withdrawals this month: 0, service charges for this month: 0.00]
Creating a bank account for Sam Smith at -5.00%
BankAccount[owner: Sam Smith, balance: 0.00, interest rate: 0.00,
        number of withdrawals this month: 0, service charges for this month: 0.00]
testAcct1 owner: John Doe
Changing the owner name to Jane Doe:
BankAccount[owner: Jane Doe, balance: 0.00, interest rate: 0.05,
        number of withdrawals this month: 0, service charges for this month: 0.00]

All numerical tests passed!

Testing methods of class SavingsAccount

Creating a savings account for Dan Doe at 5.00%
SavingsAccount[owner: Dan Doe, balance: 0.00, interest rate: 0.05,
 number of withdrawals this month: 0, service charges for this month: 0.00, myStatusIsActive: false]
Depositing 100.00 . . .
Now status should be active.
SavingsAccount[owner: Dan Doe, balance: 100.00, interest rate: 0.05,
 number of withdrawals this month: 0, service charges for this month: 0.00, myStatusIsActive: true]
Withdrawing 15.00 . . .
Status should be active.
Number of withdrawals should be 1
SavingsAccount[owner: Dan Doe, balance: 85.00, interest rate: 0.05,
 number of withdrawals this month: 1, service charges for this month: 0.00, myStatusIsActive: true]
Withdrawing 15.00 6 more times . . .
Balance should be 10.00
Status should not be active.
Number of withdrawals should be 6
Service charge should be 2.00
SavingsAccount[owner: Dan Doe, balance: 10.00, interest rate: 0.05,
 number of withdrawals this month: 6, service charges for this month: 2.00, myStatusIsActive: false]
performing monthly process  . . .
Balance should be 8.03
Status should not be active.
Number of withdrawals should be 0
Service charge should be 0.00
SavingsAccount[owner: Dan Doe, balance: 8.03, interest rate: 0.05,
 number of withdrawals this month: 0, service charges for this month: 0.00, myStatusIsActive: false]

Testing methods of class SafeDepositBoxAccount

Creating a safe deposit box account for John Doe
SafeDepositBoxAccount[owner: John Doe]
Changing the owner name to Jane Doe:
SafeDepositBoxAccount[owner: Jane Doe]

----jGRASP: operation complete.
```