CST8132 Object Oriented Programming
Lab 4: Bank Account Polymorphism

Due: Start of Lab Sessions in Week 5

Purpose

This is an exercise to practice with the concept of Polymorphism.
Suggested Reading: Chapters 9, 10 of Deitel and Deitel.

Description

This is a polymorphism example, so right from the start we know we will be processing items of a type that is a super-class (maybe abstract) or an interface. Remember the "type" is the word that goes on the left side when we declare a variable (examples, **int**, **Shape**, **Animal**, **Vehicle**, **double**, **String**, etc - but only the ones with capital letters are reference types that can be used with polymorphic processing). The super-class or interface tells us about the list of behaviors of the objects we need to process. We've seen the super-classes **Shape** in the past. When we say "behavior", we mean the methods that will be available for us to use, even though the specifics, the actual implementation, of those methods, can be determined by the sub-class. For example, the **Shape** superclass tells us what all shapes can do: give their area. The specific shapes each calculate their area differently.

In this case for this exercise we have bank accounts. What methods would all bank accounts have? **calculateInterest**? **deposit**? **withdraw**? **getBalance**? **calculateAndUpdateBalance**? What kinds of bank accounts are there? Chequing, Savings, Investment, Credit Card? As programmers by now we have been trained to recognize that there can be some methods that would be done the same way in all cases, implemented with the same Java code (**getBalance**) and there can be some methods that might be done differently depending on what sub-type it is, what type of account it is (**calculateAndUpdateBalance**) Similarly, the attributes (also known as instance variables) can be divided up into two kinds: those that are the same no matter what type of account it is, and those that depend on the subclass or type of account. The actual **balance** could be represented by a **double** in all cases, but the fees might not even exist for some bank account types, so fees would be present as an attribute in only some of the subclasses, perhaps. The basic question is what's the same for all of them, and what is not the same depending on the specific type.

So, based on the above, we have the following summary to use for the present exercise:

(super, **abstract**) class **BankAccount**
attributes: double **balance**
methods: double **getBalance()**
abstract methods: **calculateAndUpdateBalance()**

class **SavingsAccount** extends **BankAccount**
attributes: double **annualInterestRate** (yearly)
overrides **calculateAndUpdateBalance()** (add the interest for the **month**)

class **ChequingAccount** extends **BankAccount**
attributes: double **monthlyFee**
overrides **calculateAndUpdateBalance()** (subtract the fee)

We can now think about the polymorphic processing. Suppose we have thousands of accounts of various types and we need to calculate the updated balance every month. To keep things simple, we will not represent deposits and withdrawals; rather, we'll just add interest earned by a savings account, or charge the monthly fee for a chequing account. Every month we would run a loop that goes through a collection of accounts and invokes the **calculateAndUpdateBalance()** method on each of these accounts. It is polymorphic processing because the type of the objects we are processing is a superclass or an interface (in this case, a superclass). The details of the **calculateAndUpdateBalance** method come from the sub-class.

Steps

- Write down the Class Diagram UML to represent this program.
    - Classes: BankAccount, ChequingAccount, SavingsAccount, BankAccountTest
- Create a package named **lab5**.
- Implement the UML as Java classes, **including Javadoc comments throughout**.
- Make up a monthly fee amount and annual interest rate you think are reasonable. Assign these values to the appropriate properties inside the class constructors.
- Create a **BankAccountTest** class that declares, instantiates, and initializes 4 bank accounts.
    - Add an instance variable that is an array of **BankAccount** objects.
    - In the default **BankAccountTest constructor**, initialize the array with 2 instances of **ChequingAccount** objects and 2 instances of **SavingsAccount** objects. Each **BankAccount constructor** (and subclass constructor) should take a parameter of type double with a **Random** initial balance between 0 – 100.
    - The subclass constructor should call the super constructor, and pass the balance to the **BankAccount** class.
    - Add a method **monthlyProcess(BankAccount[] accounts)** method that takes an array of bank accounts as a parameter and does the monthly balance update for each account by calling **calculateAndUpdateBalance**() for each.
    - Add a **display(BankAccount[] accounts)** method that outputs the balance of each account when passed an array of bank accounts as an argument.
    - Add a **main** method that creates an instance of the **BankAccountTest** class, and calls the **monthlyProcess** and **display** methods 12 times (simulating an entire year).

Submission Requirements

1. Create a new assignment folder and name it:

   CST8132_<SectionNo>_<LastName>_<FirstName>_Lab4.

   Note: Your **SectionNo** should be your **Lab Section** (311, 312, 313).

2. Place copies of your **BankAccount.java, ChequingAccount.java, SavingsAccount.java,** and **BankAccountTest.java** files in the assignment folder.

3. Add a copy of your UML diagram to the folder, named **CST8132_<SectionNo>_<LastName>_<FirstName>_Lab4_UML.[jpg/gif/docx/vsd]**

   Example: CST8132_312_Giddings_Angela_Lab4_UML.jpg

4. Create an **ZIP** file of the assignment, and name it:

   CST8132_<SectionNo>_<LastName>_<FirstName>_Lab4.zip.

5. Upload the **ZIP** file to Blackboard.

Grading Scheme

- Javadoc: 3marks
  - Thorough – every class, method, and non-private property has Javadoc.
  - Complete tags (@author, @version, @param, @return) where necessary.
  - Proper descriptions.
- Java code: 4 marks
  - Correct implementation of abstract superclass and subclasses.
  - Proper use of polymorphic attributes and behaviors in BankAccountTest class.
  - Style – naming conventions, readability, indentation, comments, etc.
  - Submission
- UML: 3 marks
  - Classes
  - Relationships
  - Matches Java code
- Code does not compile: -5 marks