MPP Standardized Programming Exam
March, 2017

This 90-minute programming test measures the success of your MPP course by testing your new skill level in two core areas of the MPP curriculum: (1) Lambdas and streams, and (2) Implementation of UML in Java.You will need to demonstrate a basic level of competency in these areas in order to move past MPP.

Your test will be evaluated with marks "Pass" or "Fail." A "Pass" means that you have completed this portion of evaluation only; your professor will evaluate your work over the past month to determine your final grade in your MPP course, taking into account your work on exams and assignments. A "Fail" means you will need to repeat MPP, with your professor's approval.

There are two programming problems to solve on this test. You will use the Java classes that have been provided for you in an Eclipse workspace. You will complete the necessary coding in these classes, following the instructions provided below.

Problem 1. **[Lambdas/Streams]** We have a list of undergraduate students. Students have the following attributes:

double gpa;
String major;

In this problem, you will create a `Stream` pipeline to output the list of all `Student`s from the original list that have a gpa greater than 3.0 and have a major of "Computer Science". You will use the start-up code provided in your Eclipse workspace to implement your solution.

Specifically, you must write your pipeline solution in the body of the static method

```
List<Student> Admin.obtainHonorRoll(List<Student>)
```

A `Student` class has already been provided for you. The `Majors` class is a class containing constants which name the possible majors. The `Main` class contains a `main` method that you may use to test your code.

Requirements for this problem.

(1) Your solution, implemented in the body of the `Admin.obtainHonorRoll` method, must be a single `Stream` pipeline. You may not make use of instance variables or local variables declared in the body of `Admin.obtainHonorRoll`.
(2) You may not modify the `Student` or `Majors` classes in any way. Also, you may not modify the signature of `Admin.obtainHonorRoll`, or change any of its qualifiers (`public, static`).
(3) There must not be any compilation errors or runtime errors in the solution that you submit.

Problem 2. [UML → Code] In a company, employees may have multiple bank accounts: zero or more savings accounts and zero or more checking accounts. Each checking account has an account id, a balance, and a monthly fee. Each savings account has an account id, a balance, and an interest rate associated with the particular type of savings account. It is possible to read the current balance in any of these accounts, but it is also possible to determine the balance after interest or monthly fee is applied by calling the `computeUpdatedBalance` method on the account.

An administrator has access to all employee records and from time to time computes the total balance across all employee-owned accounts; for each account, the balance that is needed in this computation is the *updated balance*. This computation is performed in the static method

`computeUpdatedBalanceSum`

in the `Admin` class.

Below is a class diagram showing the classes involved and relationships between them. A sequence diagram for the operation `computeUpdatedBalanceSum` is also provided. Your task in this problem is to write Java code that implements the classes and relationships shown in the diagram. Shells for the `Admin` and the `Employee` classes have been provided in your workspace. Also, a `Main` class (with a `main` method) has been provided for you to test your code – the code in the `main` method has been commented out; when you are ready to test your code, you can uncomment it.

The method `computeUpdatedBalance` in `CheckingAccount` does the following computation to obtain the return value:

```
balance – monthlyFee.
```

The method `computeUpdatedBalance` in `SavingsAccount` performs the following computation to obtain the return value:

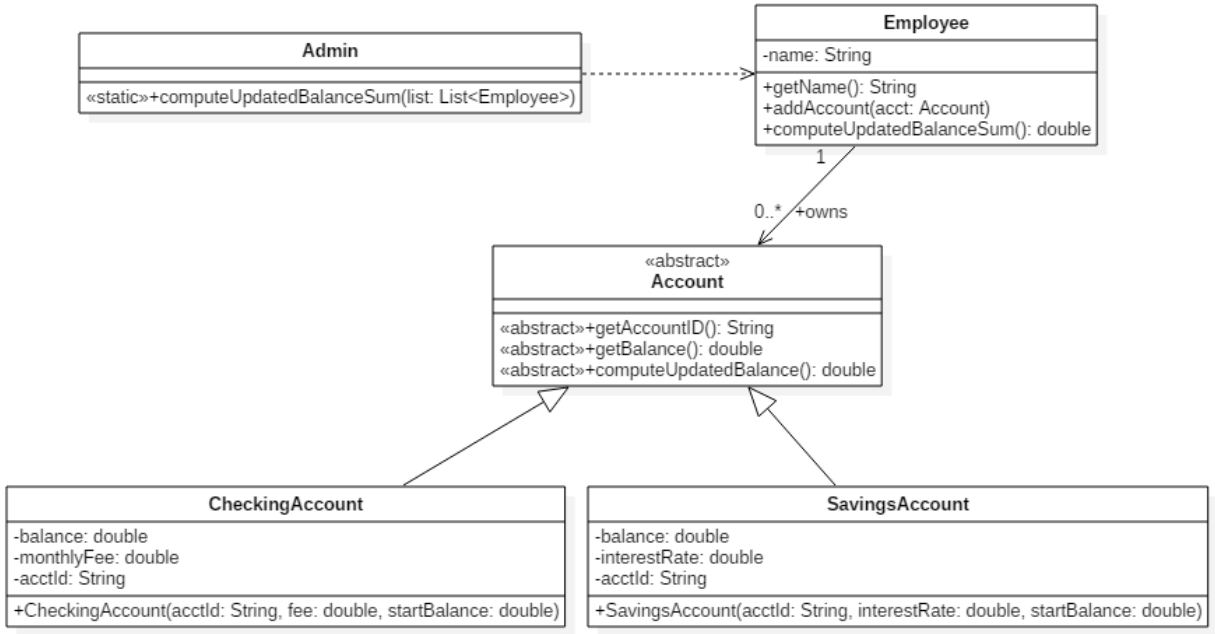
```
balance + (interestRate * balance).
```

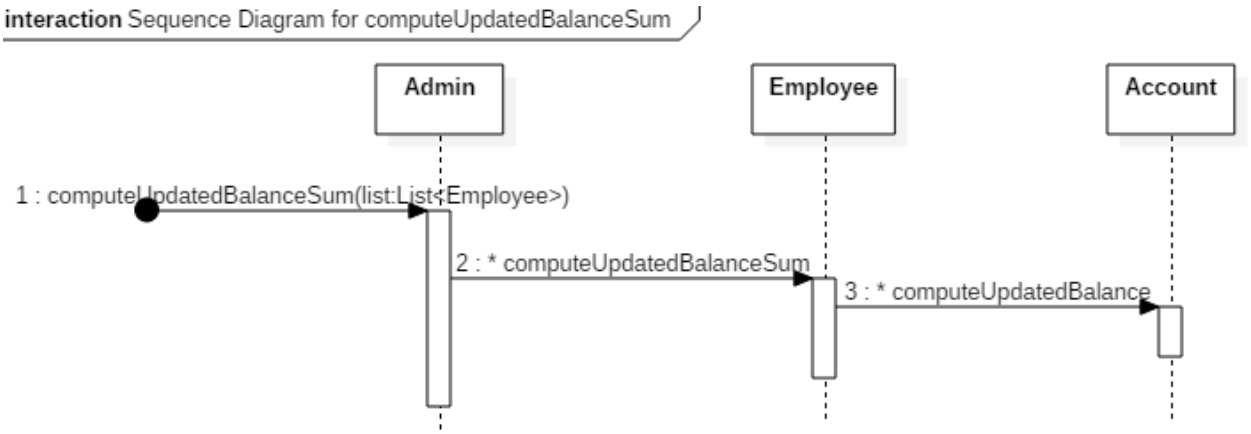
Points to notice about the class diagram:

1. `CheckingAccount` and `SavingsAccount` are subclasses of `Account`. Also, `Account` has several abstract methods which must be implemented in its subclasses.
2. There is a one-way association from `Employee` to `Account`. It is important that your code reflects and maintains this association.
3. The diagram has a mix of dependencies and associations; make sure your code distinguishes between these properly.

Start-up code for this problem can be found in the `prob2` package in your workspace, providing shells for the classes `Admin` and `Employee`. A `Main` class, with a `main` method that launches the program, is provided in the package `prob2.launch`, which you may use to test your code. You may not modify the signatures or qualifiers of the methods contained in the `Admin` or `Employee` classes that have been provided. You will need to create all the other classes mentioned in the diagrams.

Note: Your submitted code must accurately implement the UML models provided and must have no compiler or runtimte errors.


Admin
«static»+computeUpdatedBalanceSum(list: List<Employee>)
Employee
-name: String
+getName(): String
+addAccount(acct: Account)
+computeUpdatedBalanceSum(): double
1
0..*
+owns
«abstract»
Account
«abstract»+getAccountID(): String
«abstract»+getBalance(): double
«abstract»+computeUpdatedBalance(): double
CheckingAccount
-balance: double
-monthlyFee: double
-acctId: String
+CheckingAccount(acctId: String, fee: double, startBalance: double)
SavingsAccount
-balance: double
-interestRate: double
-acctId: String
+SavingsAccount(acctId: String, interestRate: double, startBalance: double)


interaction Sequence Diagram for computeUpdatedBalanceSum
Admin
Employee
Account
1 : computeUpdatedBalanceSum(list:List<Employee>)
2 : * computeUpdatedBalanceSum
3 : * computeUpdatedBalance