MPP Standardized Programming Exam
June, 2017

This two-hour 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.

In order to pass this programming test, *you must get a score of 70 or higher on both of the problems given here.* Getting a high score on one problem and a low score (below 70) on another will result in a "Fail."

Problem 1. [Lambdas/Streams] In your `prob1` package, you will find a class `Problem1` that contains two static methods:

```
static List<String> elementsInBoth(List<String> list1, List<String> list2)
static List<String> getZipsOfSpecialCustomers(List<Customer> list)
```

The method `elementsInBoth` returns a list of all `Strings` that occur in both of the two input lists. Below is an example of how this method should behave:

Example: If `list1 = {"A", "B", "D"}` and `list2 = {"B", "C", "D"}`, then the return list should be `{"B", "D"}` since both "B" and "D" occur in both lists.

The method `getZipsOfSpecialCustomers` returns a list of the zipcodes, in sorted order, of those Customers who live in a city for which the name of the city contains 6 or more characters, but which does not contain the letter 'e'. Your output list must not contain duplicate elements.

Example: Below are 5 customers.
Customer 1: ["Bob", "11 Adams", "Fairfield", "52556"]
Customer 2: ["Andy", "1000 Channing Ave", "Oskaloosa", "54672"]
Customer 3: ["Zeke", "212 Wilkshire Blvd", "Chicago", "57532"]
Customer 4: ["Tom", "211 Blake Ave", "Oskaloosa", "54672"]
Customer 5: ["Bill", "10 Wolfsen Blvd", "Orkin", "84447"]

When run on this customer list, the method should return the following list of zip codes:

["54672","57532"]

The Fairfield customer was ignored (because 'e' occurs in Fairfield) and the Orkin customer was ignored (because the length of "Orkin" is less than 6). Also, the multiple occurrences of an Oskaloosa zipcode were reduced to just one so that there were no duplicates in the final list. Note that the final zipcode list is in sorted order.

A `main` method has been provided that will help you test your implementations of both of these methods.

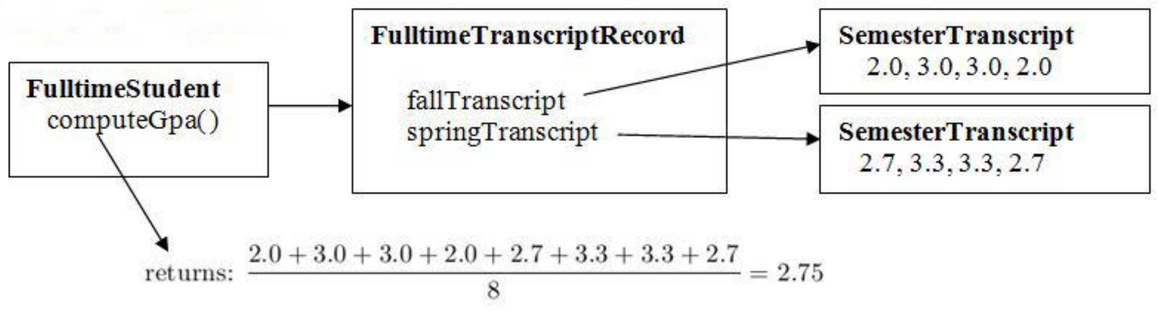
Requirements for Problem 1.

1. Your code may not contain any loops (while loops, for loops).
2. The body of each of the methods `elementsInBoth`, `getZipsOfSpecialCustomers` must be a single `Stream` pipeline. You must not make use of instance variables or local variables declared in the body of either method. (Example of a local variable:
   ```
   int myMethod() {
          int x = //computation
          return x;
    }
   ```
   Here, x is a local variable. Not allowed in this problem.)
3. You may not create auxiliary methods for use in your pipeline.
4. There must not be any compilation errors or runtime errors in the solution that you submit.

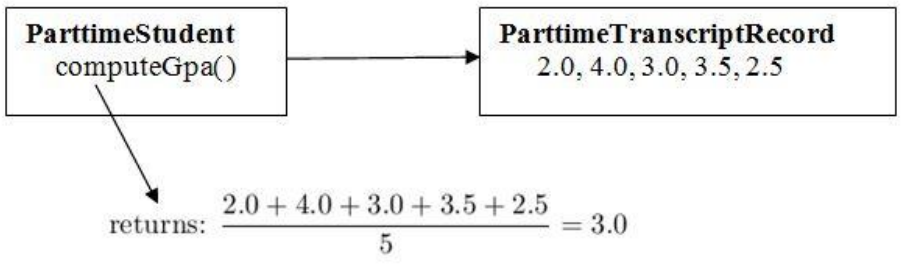
Problem 2. [UML → Code] The administrative office at a particular university keeps records of all student grade point averages (GPAs). Normal record formatting is done for full-time students, but a less detailed type of record formatting is done for part-time students.

In the class diagram below, you will see that there are two types of students: `FulltimeStudent` and `ParttimeStudent`; each of these is a subclass of `Student`, which is an abstract class having an abstract method `computeGpa()`.

Related to `FulltimeStudent` in the diagram are the classes `FulltimeTranscriptRecord` and `SemesterTranscript`. These classes work together to provide information about a full-time student's performance during a particular school year. A `SemesterTranscript` just contains a list of grades (numerical values in the range 0.0 – 4.0). A `FulltimeTranscriptRecord` contains one `fallTranscript` and one `springTranscript`, each of which is an instance of `SemesterTranscript`. And a `FulltimeStudent` has just one `FulltimeTranscriptRecord`, which provides a complete record of a student's grades in both the fall and spring semesters. The `FulltimeStudent` class provides a method `computeGpa` which sums all the grades stored in the student's `FulltimeTranscriptRecord` – adding up the gpa's from both fall and spring semester – and which then computes the average of these and returns it.



Related to `ParttimeStudent` in the diagram is the class `ParttimeTranscriptRecord`. This class works with `ParttimeStudent` to provide information about a part-time student's performance during a school year. A `ParttimeTranscriptRecord` contains a list of grades (numerical values in the range 0.0 – 4.0), and every `ParttimeStudent` has just one `ParttimeTranscriptRecord`. Note that grades for a part-time student are tracked for the whole year, and not considered separately for each semester as they are for full-time students. A `ParttimeStudent` contains a method `computeGpa` which sums all the grades in the student's `ParttimeTranscriptRecord`, computes the average, and returns it.



There are two objectives for this problem. The first is to translate the class diagram shown below into Java code. Shells for the Java classes have been provided for you in the `prob2` package. Your code must accurately display attributes, methods, inheritance, depedendencies, associations and association roles, and multiplicities.

The second objective is to properly implement the following static method in the `Admin` class:

```
static double computeAverageGpa(List<Student> students)
```

The `computeAverageGpa` method uses the input list to polymorphically compute the average gpa of all the students in the input list. It does this by calling the `computeGpa` method on each student in the list, summing, and taking the average. The sequence diagram shown below illustrates how each subclass of `Student` (namely, `FulltimeStudent` and `ParttimeStudent`) implements its own `computeGpa` method.

HINT: You may implement `computeAverageGpa` using Java 8 lambdas and streams or by using loops – either way is allowed.

Besides getter and setter methods in the classes shown in the class diagram below, there are two methods that you need to implement which are not shown in the sequence diagram—namely, two occurrences of the method `insertGrade(double grade)`. These are found in the `SemesterTranscript` class and also in the `ParttimeTranscriptRecord` class. In both cases, the method is implemented by adding the input grade to the `grades` list that is contained in the class.

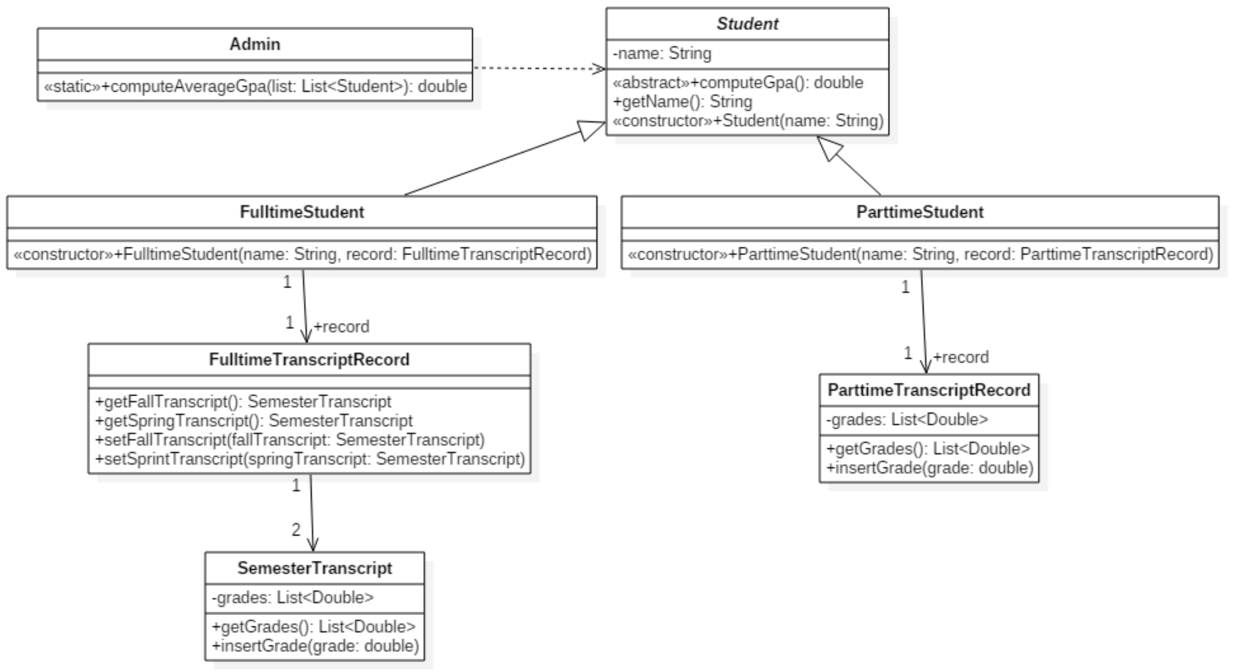
A `main` method has been provided that will allow you to test your code. The expected output for the `computeAverageGpa` method is shown in the comments.

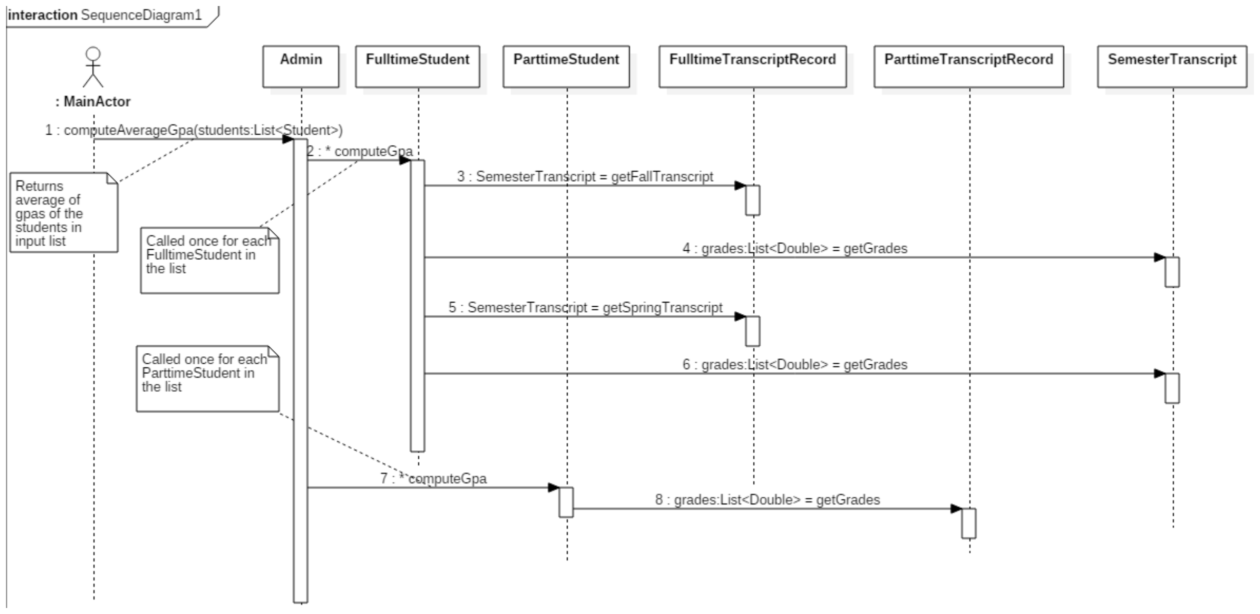
Notes:

(a) Although computation of the average gpa (by the method `computeAverageGpa`) is to be done by polymorphism, the sequence diagram (which shows only runtime objects) does not illustrate how the polymorphism works.
(b) The `Main` class is treated as an actor in the Sequence diagram since it is responsible for initiating the action

Requirements for Problem 2.

(1) You must compute average gpa *using polymorphism.*
(2) Your implementation of `computeAverageGpa` may not check types (using `instanceof` or `getClass()`) in order to `computeGpa` from any of the students in the input list.
(3) Your computation of average gpa must be correct.
(4) There must not be any compilation errors or runtime errors in the solution that you submit.


Admin
«static»+computeAverageGpa(list: List<Student>): double
Student
-name: String
«abstract»+computeGpa(): double
+getName(): String
«constructor»+Student(name: String)
FulltimeStudent
«constructor»+FulltimeStudent(name: String, record: FulltimeTranscriptRecord)
ParttimeStudent
«constructor»+ParttimeStudent(name: String, record: ParttimeTranscriptRecord)
1
1
+record
FulltimeTranscriptRecord
+getFallTranscript(): SemesterTranscript
+getSpringTranscript(): SemesterTranscript
+setFallTranscript(fallTranscript: SemesterTranscript)
+setSprintTranscript(springTranscript: SemesterTranscript)
1
2
SemesterTranscript
-grades: List<Double>
+getGrades(): List<Double>
+insertGrade(grade: double)
1
1
+record
ParttimeTranscriptRecord
-grades: List<Double>
+getGrades(): List<Double>
+insertGrade(grade: double)


interaction SequenceDiagram1
: MainActor
Admin
FulltimeStudent
ParttimeStudent
FulltimeTranscriptRecord
ParttimeTranscriptRecord
SemesterTranscript
1 : computeAverageGpa(students:List<Student>)
2 : * computeGpa
3 : SemesterTranscript = getFallTranscript
4 : grades:List<Double> = getGrades
5 : SemesterTranscript = getSpringTranscript
6 : grades:List<Double> = getGrades
7 : * computeGpa
8 : grades:List<Double> = getGrades
Returns average of gpas of the students in input list
Called once for each FulltimeStudent in the list
Called once for each ParttimeStudent in the list