MPP Standardized Programming Exam
March, 2018, Exam #2

This 2-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. To pass this test, you must get a score of at least 70% on Problem 1 and also a score of at least 70% on Problem 2.

Problem 1. **[Lambdas/Streams]** In your `prob1` package, you will find classes `Customer, Address`, and `Account`, along with a class `Problem1`. `Problem1` contains two static methods that you must implement, and a `main` method that will allow you to test your code.

The classes `Customer`, `Address`, and `Account` are *read-only.* They are used for reference during the test and *must not be modified.*

Here are the requirements for implementing the two methods in the `Problem1` class:

`List<Customer> specialCustomers(List<Account> accts)` This method returns a list of `Customers` whose checking account id has exactly the same length (as a string) as the city name of that customer. (Note: You are not required to sort your final list of Customers.)

Example: Suppose a `Customer's` checking account id is "1060422" and the `Customer`'s city is Hedrick. Since the length of "Hedrick" as a string is 7 and since the account id also has 7 characters, this `Customer` should be included in the final output.

`List<Customer> specialAccounts(List<Customer> custs)` This method returns a list of `Customer`s, sorted by last name, whose checking account balance is less than 100 but greater than 50, and whose first name begins with the letter 'A'.

Note: The `main` method in `Problem1` allows you to test your solutions, and expected outputs are displayed in the comments.

Other requirements for this problem.

(1) You are not allowed to change the signature of either of the two functions `specialCustomers`, `specialAccounts` (if you change the input type or return type of either of these, you will not receive any credit for that part of the problem).
(2) Each of the methods you implement, in the class `Problem1`, 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) There must not be any compilation errors or runtime errors in the solution that you submit.

Problem 2. [UML → Code] The class diagram shown below (next page) is a model for a simplified student registration system. Each `Student` has a `PlanOfStudy`, which lists the courses he/she needs to take in order to complete requirements for a particular major. Each `Student` also has a `Transcript`, which is a record of the `Student`'s performance in each class he/she has taken, listing each course taken along with the grade received. The `Transcript` is formed as a list of `TranscriptEntries`; a `TranscriptEntry` contains information about a single course and the grade received.

For this problem, there are two tasks.

(1) Implement the diagram in code. In your workspace, all the classes in the diagram have been created for you, but they need to be implemented. You must include attributes and methods and the associations shown in the diagram (including multiplicities) must be accurately reflected in your code.

(2) You must implement the method

```
List<Course> coursesTaughtByValentine(List<Student> allStudents)
```

that is contained in the `Main` class (located in the package `prob2.admin`). This method accepts as input a list of all `Student`s in the system and returns a list of all courses that were taken by these `Student`s and for which the primary professor was Valentine. The class diagram will clarify how to navigate to obtain the necessary course information so that you can implement this method.

Note: The `main` method of the `Main` class loads test data and calls your method. When you have finished implementing your method `coursesTaughtByValentine`, you can run the `main` method to test your code. The expected output is shown in the comments for your method.

Note: You are *not required* to use lambdas or streams in your implementation of `coursesTaughtByValentine` (you are allowed to use these but not required to do so).

Requirements for this problem.

(1) You must not modify the signature of the method `coursesTaughtByValentine`.

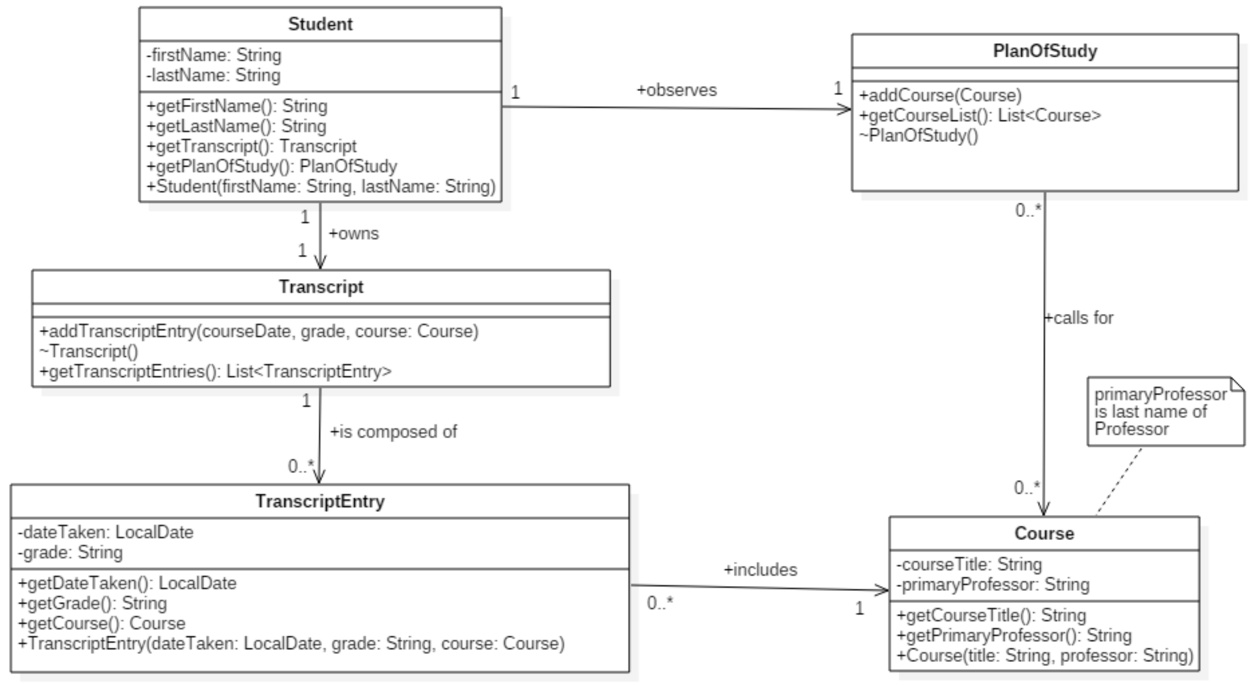
(2) All classes, attributes, operations, and relationships shown in the class diagram must be accurately reproduced in your code (be careful about spelling).

(3) *No duplicates!* The output of your method `coursesTaughtByValentine` must not contain duplicates; each course in your output list must occur only once. (Hint: Think about overriding equals somewhere.)

(4) You must implement constructors of classes as shown in the class diagram. Especially note the *visibility qualifiers*. (The symbol '+' means "public"; the symbol '~' means "package-level".)

(5) Your submitted code must not have compiler errors or runtime exceptions when executed.

Warning! Be careful about your implementation of the constructor for `Student`. If you are not careful, your code will not work. Pay attention to the associations that `Student` is involved in.


Student
-firstName: String
-lastName: String
+getFirstName(): String
+getLastName(): String
+getTranscript(): Transcript
+getPlanOfStudy(): PlanOfStudy
+Student(firstName: String, lastName: String)
1
+observes
1
PlanOfStudy
+addCourse(Course)
+getCourseList(): List<Course>
~PlanOfStudy()
1
+owns
1
Transcript
+addTranscriptEntry(courseDate, grade, course: Course)
~Transcript()
+getTranscriptEntries(): List<TranscriptEntry>
1
+is composed of
0..*
TranscriptEntry
-dateTaken: LocalDate
-grade: String
+getDateTaken(): LocalDate
+getGrade(): String
+getCourse(): Course
+TranscriptEntry(dateTaken: LocalDate, grade: String, course: Course)
0..*
+includes
1
0..*
+calls for
0..*
primaryProfessor is last name of Professor
Course
-courseTitle: String
-primaryProfessor: String
+getCourseTitle(): String
+getPrimaryProfessor(): String
+Course(title: String, professor: String)