FPP Standardized Programming Exam
December, 2016

This 90-minute programming test measures the success of your FPP course by testing your new skill level in two core areas of the FPP curriculum: OO programming (specifically, polymorphism) and data structures. You will need to demonstrate a basic level of competency in these areas in order to move on to 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 FPP course, taking into account your work on exams and assignments. A "Fail" means you will need to repeat FPP, 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. **[Data Structures]** In your `prob1` package, you will find two classes, `Employee` and `EmployeeAdmin`. A `Main` class is also provided that will make it convenient to test your code.

The `Employee` class has been fully implemented. It has three fields: `name`, `salary`, and `ssn` (which stores a social security number). `Employee` provides getters and setters for each of these fields.

The `EmployeeAdmin` class is intended to provide reports about `Employee`s. For this problem, the `EmployeeAdmin` class has just one static method, `prepareReport`, which accepts a `HashMap table` and a `List socSecNums` as arguments. The `HashMap` matches employee social security numbers with `Employee` objects. The `List` contains some employee social security numbers, represented as `String`s.

Your method `prepareReport` must produce a list of all `Employee`s in the input `table` whose social security number is in the input list `socSecNums` and whose `salary` is greater than $80,000. In addition, this list of `Employee`s must be sorted by social security number, in ascending order (from numerically smallest to numerically largest).

The `main` method in the `Main` class provides test data that you can use to test your code.

Here is an example of how the method `prepareReport` should behave: In the input `table`, you see four entries: The first entry associates with the `ssn` "`223456789`" the `Employee` object ["Jim", 90000, "`223456789`"]. There are three additional entries in table. The list `socSecNums`, also provided, contains four social security numbers.

```
table:
   "223456789"  →  ["Jim", 90000, "223456789"]
   "100456789"  →  ["Tom", 88000, "100456789"]
   "630426389"  →  ["Don", 60000, "630426389"]
   "777726389"  →  ["Obi", 60000, "777726389"]

socSecNums:
    "630426389",  "223456789"  , "929333111",  "100456789"
```

When we scan the list `socSecNums`, and use these values to read the table, we find only three of the employees: Jim, Tom, Don. We also notice that only Jim and Tom have

salaries greater than $80000, so only these two Employees will be returned in our final list. We then sort this list of two `Employee`s (Jim and Tom) according to the order of their social security numbers. The final output should be :

```
["Tom", 88000, "100456789"],  ["Jim", 90000, "223456789"]
```

Requirements for this problem.

(1) Your list of `Employee`s must be sorted using a sorting method in Java's `Collections` class.

(2) Ordering of `Employee`s must be determined by a `Comparator`, which you must define yourself (and include in your workspace). Your `Comparator` should follow this rule: Given `Employee`s `e1` and `e2`, `e1` should be considered "less than" `e2` if the social security number of `e1` precedes the social security number of `e2` (in the natural ordering of `String`s). *Note*: You may assume that no two employees provided in the input table have the same social security number.

(3) Your return list of `Employee`s must not contain `Employee` objects whose social security number is not on the input list `socSecNums`. Note also that there may be social security numbers in the input list `socSecNums` that do not belong to any of the `Employee` objects in the table.

(4) Your list of `Employees` must not contain any `null`s.

(5) You may not modify the `Employee` class in any way.

(6) There must not be any compilation errors or runtime errors in the solution that you submit.

Problem 2. [Polymorphism] In the `prob2` package of your workspace, you are given fully implemented classes `Staff` and `Teacher`. You will also find a class `Statistics` in the `prob2` package, with an unimplemented method `computeSumOfSalaries`. Your task is to implement `computeSumOfSalaries`.

The `main` method of the `Main` class must first combine the two input lists of `Staff` and `Teacher` objects into a single list (using the `combine` method provided). You may find the interface `EmployeeData` useful for this purpose; this interface is provided in the `prob2` package, but you will need to implement it yourself.

The combined list should be passed into `computeSumOfSalaries`, which must then *polymorphically* compute the sum of all the salaries of all `Staff` and `Teacher` objects in this combined list, by calling each object's `getSalary()` method; it must then return this computed value.

Requirements for this problem.

(1) You must compute the sum of all salaries using polymorphism. (For instance, if you obtain the sum of all salaries by first computing the sums of the salaries in each list separately, you will receive no credit.)

(2) Your implementation of `computeSumOfSalaries` may not check types (using `instanceof` or `getClass()`).

(3) You must use parametrized lists, not "raw" lists. (Example: This is a parametrized list: `List<Duck> list`. This is a "raw" list: `List list`.)

(4) You must add a proper `List` type in your implementations in the `Main` and `Statistics` classes.

(5) You must implement the `combine` method provided in the `Main` class for the purpose of combining the `Staff` and `Teacher` lists.
(6) You are allowed to modify declarations of `Staff` and `Teacher` (to support inheritance or interface implementation), but you *must not* remove the `final` keyword from either of these class declarations.
(7) There must not be any compilation errors or runtime errors in the solution that you submit.