FPP Standardized Programming Exam
March, 2018

This 2-hour 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. In order to pass, you must get a score of at least 70% *on each of the two problems*.

Problem 1. **[Data Structures]** In your `prob1` package, you will find a class `MinDoublyLinkedList`. A `MinDoublyLinkedList` is a doubly linked list with header, and one more property: The minimum element is always contained in the node immediately after the header. (We adopt the convention that header is in position -1 and the node immediately following the header is in position 0, etc...). For this problem, all elements of the list will be `Strings`. We will say that a `String t` is "greater than" a `String s` if `t` comes after `s` in the usual dictionary ordering of `Strings`.

For this problem, you must implement the following method:

```
public void add(String item)
```

The strategy for the `add` method is the following:

- If the list is empty, the element to add is placed in position 0.
- If the element to add is greater than the element at position 0, then place the new element in position 1.
- Otherwise, the new element should be placed in position 0, and the element that used to be in position 0 should then be put in position 1.

In all three cases, no changes need to be made to any other nodes.

Examples. Suppose you start with the following list:

`["bill","tom","mike"]`

After executing `add("anne")`, the list should look like this:

`["anne","bill","tom","mike"]`

(The `String` "anne" is placed in position 0 because "anne" comes before "bill" in the `String` ordering.)

Then, after executing `add("chris")`, the list should look like this:

`["anne","chris","bill","tom","mike"]`

(The `String` "chris" is placed in position 1 because "chris" is greater than "anne" in the `String` ordering.)

A `toString` method has been provided so you can test your code.

Requirements for Problem 1:

(1) Your code must run correctly for lists containing any number of elements, including an empty list. (An empty list will still have a header node, but the list will have no nodes containing any data.)
(2) No data may be placed in the header node.
(3) You may not introduce any new instance variables or instance methods, and you may not modify the other methods in `MinDoublyLinkedList`.
(4) The `Node` class contained in `MinDoublyLinkedList` must not be modified.
(5) During execution, each `Node` in your `MinDoublyLinkedList` must have correct values for the `next` and `previous` `Node`s.
(6) There should be no compiler or runtime errors. In particular, no `NullPointerException`s should be thrown during execution.

Problem 2. [Polymorphism] In the `prob2` package of your workspace, you are given a `Main` class and three fully implemented classes: `FirstClass, BusinessClass` and `EconomyClass`. The classes `FirstClass, BusinessClass,` and `EconomyClass` represent different types of customers that an airline service may have. Each contains a method `computeBoardingTime` which computes the boarding time for the current customer. Notice that the `computeBoardingTime` method is implemented in different ways in each of the classes.

The `Main` class is used for accumulating information about multiple customers – in particular, the `Main` class can be used to store several customers and then compute the average boarding time across all of these stored customers, using the method `computeAverageBoardingTime`.

The method `computeAverageBoardingTime` has been provided to you but has been implemented in the wrong way: It checks the runtime type of each customer, casts it to the right type, and then calls the `computeBoardingTime` on each type. Your task for this problem is to rewrite this method so that the computation of average boarding time is done *polymorphically.* This requirement implies that your implementation *does not check* the runtime types of the customers in the object array. To satisfy this requirement, you *will need to create and use an interface* `BoardingTime`, which has been provided for you in your `prob2` package; the interface `BoardingTime` that has been given to you contains no methods; you must add an appropriate method to this interface.

You are allowed to make changes to the type of the object array `objs` in the `main` method, and also to the type of the argument of the method `computeAverageBoardingTime`. You are allowed to make changes to the class declaration of the classes `FirstClass, BusinessClass, EconomyClass`, but you are not allowed to modify the `computeBoardingTime` method in any of these classes.

Test your work by running the `main` method of `Main`. The expected output is provided in the `Main` class.