FPP Standardized Programming Exam
July, 2017

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 partially implemented class `SinglyLinked`. For this problem, you must implement the following methods

```
public void insertLast(String value)
public void removeLast()
```

The `insertLast` method inserts the input `String` `value` at the end of the current list (as long as `value` is not null) , so that it becomes the last element of the list. On the other hand, if the input String is null, the insertLast method does nothing (and the current list is not modified). The `removeLast()` method removes the last element of the list. If `removeLast()` is called on a list that contains no elements, an `IndexOutOfBoundsException` (which is one of the subclasses of `RuntimeException` in the Java libraries) must be thrown.

A `toString` method, and also a main `method` with test data, have been provided in `SinglyLinked` so you can test your code.

Example. Suppose your list has these values:

```
["Bob","Bill","Tom"]
```

After executing `insertLast("Rich")`, the list should contain these elements (in this order):

```
["Bob", "Bill", "Tom", "Rich"]
```

Example. Suppose your list has these values:

```
["Bob","Bill","Tom"]
```

After executing `removeLast`, the list should contain these elements (in this order):

```
["Bob", "Bill"]
```

Requirements for Problem 1:

(1) Your code must run correctly for lists containing any number of elements, including an empty list. In particular, if `null` is passed into the `insertLast` method, your code must handle this correctly (by ignoring the input).

(2) You may not modify the inner class `Node` that has been provided in your `SinglyLinked` class. In particular, you may not add or modify the constructor provided in this class.

(3) `IndexOutOfBoundsException` (which is a Java library class) must be thrown when `removeLast` is called on your list when it has no elements.

(4) You may not use any of the Collections API classes in the Java libraries, including any implementation of the `List` interface.

(5) In the class `SinglyLinked`, you may not introduce any new instance variables or instance methods, and you may not modify the other instance methods.

(6) There should be no compiler or runtime errors. In particular, no `NullPointerException`s should be thrown during execution.
(7) It should be possible to add any number (not exceeding memory limits of your laptop) of new elements to your list.
(8) The `main` method provides tests and test data that are very similar to the tests that will be run by the evaluators of your exam. Expected outputs are stated in the comments of the `main` method.

Notes:
(A) This singly linked list does *not* have a header. The Node in position 0 is called `startNode`, and it stores data values in the same way as each of the other Nodes in the list. This must always remain in position 0 in your list and should never become null. The list is considered to be empty if the value stored in the startNode is null (see the isEmpty method provided for you).

Problem 2. [Polymorphism] In the `prob2` package of your workspace, there are two subpackages: `prob2.incorrect` and `prob2.solution`. Both packages contain an `Employee` class, as well as classes for three different types of bank accounts (`RetirementAccount, SavingsAccount, CheckingAccount`). An `Employee` has instance variables `id` and `accounts` (which is a list of accounts). Each employee account will be one of the three account types mentioned above. There is also an `AccountManager` class containing a static method `computeAccountBalanceSum`, which takes as input a list of `Employee` objects; for each such `Employee` object, it extracts the balance from each of the accounts in the list of accounts contained in that `Employee`, and adds them to a running `sum` variable; finally, `computeAccountBalanceSum` returns the final value of `sum`.

In the package `prob2.incorrect`, all the code is correct and the total sum of balances that is computed by the `AccountManager` is correct; however, the implementation in this package is of very low quality because polymorphism has not been used.

The objective of this problem is to rewrite this code so that polymorphism is used. All the classes in `prob2.incorrect` have been copied into the package `prob2.solution`. You must make the necessary modifications to the classes you find in `prob2.solution` so that computation of total sum of balances is still correct, but computation is done polymorphically. An abstract class `Account` (unimplemented) has been provided for you to assist in your polymorphic implementation.

In each of the packages `prob2.incorrect` and `prob2.solution`, a `Main` class is provided that can be used to test the `AccountManager.computeAccountBalanceSum` method.

Requirements for this problem.
1. All Lists in your solution package must have an appropriate type (for instance, `List<Employee>` rather than just `List`).
2. Your implementation of `computeAccountBalanceSum` in `AccountManager` must correctly output the sum of the balances of all accounts in all the `Employee` objects passed in as an argument.
3. Your implementation of `computeAccountBalanceSum` must make correct use of polymorphism .
4. *None of the code in the prob2.incorrect package may be modified!*
5. You are allowed to modify declarations of the different bank account classes, but the *final* keyword used in these classes may not be removed.
6. There must not be any compilation errors or runtime errors in the solution that you submit.