Pre-test Guidelines

I. To qualify for the FPP track, you must have sufficient skill in the following general areas:
   1. Basic procedural programming and general problem solving
   2. Experience in at least one programming language
   3. See https://compro.miu.edu/sample-test/ for sample FPP pretests

II. For the MPP track, a deeper grasp of the the topics listed above is necessary, and the student should understand how to apply them in the context of the Java programming language. In addition, the student should have sufficient understanding of
   1. Data structures (lists and list nodes, stacks, queues, hash tables, trees)
   2. Recursion and iteration
   3. The object-oriented paradigm: working with objects, interfaces, inheritance, and polymorphism in the Java language

   For each of these topics, there will be one question on the MPP pretest that tests your understanding.

III. As a guide to the level of required experience in Java, essential topics are listed below. You can review these topics by reading *Core Java*, Volume 1, Horstmann and Cornell, 9th edition. Chapter references are given beside each of the topics below.

- the String, StringBuffer/StringBuilder classes (for creating and concatenating Strings) (chapter 3, Strings)

- Java methods: return values, parameter passing (chapter 4, see Mutator and Accessor Methods, and Private Methods for example)

- basics of classes and objects: class construction, object references, invoking a method on an object (chapter 4, Object Construction)

- Data Structures

  Be able to use these classes from the Java libraries
  - `ArrayList` (p. 658 and elsewhere)
  - `LinkedList` (p. 664)
  - Stacks and Queues
  - `HashMap` (p. 707-8)
  - Tree (`TreeMap`) (pp. 681, 691)

  Be able to create the following data structures on your own, *without* the help of the Java libraries:
  - `ArrayList` (p. 658 and elsewhere)
  - `LinkedList` (p. 664)
  - Stacks and Queues

  Note: Basic knowledge about how to use generics, like List<Integer>, HashMap<String, String>, will be assumed.

- Using the methods inherited from the Object class – in particular, you need to know how to override `toString` and `equals` as necessary.

- OO concepts: objects, interfaces, passing messages between objects, polymorphism.

Sample MPP Pretest

The purpose of this test is to assess your level of preparation in problem-solving, data structures, basic OO, and the Java programming language. For each of the problems below, write the simplest, clearest solution you can, in the form of a short program. You will be writing your code with the help of a Java compiler and the Eclipse development environment; you will not, however, have access to the internet. Because a compiler has been provided, it is expected that the code you submit for each of the problems will be free of compilation errors and will be a fully functioning program. If a solution that you submit has compilation errors, you will not receive credit for that problem.

Initially, you will receive Java code for each of 3 problems. The instructions below will explain how this code is to be completed to solve certain problems. In each of the packages that you will find in your workspace, a `Main` class has been provided, with an implemented `main` method. This method is provided to you as a way of testing your implementations. Similar test code will be run when your code is tested during evaluation.

Unless instructed otherwise, you must not change methods in the code provided – you must not change the name, the input arguments, the return type, or the visibility qualifier (public, private, etc.) The only exception is that you can change the code in any of the `Main` classes in any way you wish.

To get a passing grade on this Pretest (so that you may go directly to MPP rather than FPP), there are two requirements:

A. You must get full credit for the Polymorphism problem (Problem 3)

B. Your total score needs to be 70% or higher.

Problem 1: [40 %] [Recursion] In the `prob1` package, you will find a class Reverse with an unimplemented method having the following signature and return type:
String reverse(String str)

For this problem, you must provide a *recursive implementation* that will reverse the characters in any input string.

For example, on input "Axyt", the reverse method should ouptut "tyxA".

Use the class Reverse and method reverse that are provided.

Note: If you solve this problem without using recursion, you will receive no credit.

Problem 2. **[40%][Data Structures]** For this problem, you will implement a stack of Integers, using an array in the background. In your `prob2` package, you will find a class named `MyIntStack`. There are no implemented methods in this class. Your task is to provide implementations of all of the methods declared there so that your class behaves as a stack. The methods to be implemented are: `isFull, isEmpty, push, pop`, and `peek`. The comments provided in the code specify the expected behavior of each of these operations.

Your code must meet the following requirements:

1. Pushing, popping, and peeking your stack during test runs must not cause any kind of runtime exception to occur (on the other hand, if a pop or peek is attempted when the stack is empty, a `StackException` should be thrown; note that `StackException` is a checked exception, not a runtime exception). (Beware of `ArrayIndexOutOfBoundsExceptions`.)
2. Your stack should reject `null` inputs – it should not be possible to push a `null Integer` onto your stack.
3. Your stack's top element is always the last element in the underlying array `arr`.
4. The `size` variable, provided in `MyIntStack`, serves as a stack pointer, indicating how many elements are in the stack as well as indicating the position of the next available slot in the underlying array `arr`. Your implementations of pop and push operations must keep the `size` field up to date.

Problem 3. [20%][Polymorphism] In the `prob3` package of your workspace, you are given fully implemented classes `Employee and Manager. Manager` is a subclass of `Employee`. You will also find a class `Statistics` in `prob3` package, with an unimplemented method `computeSumOfSalaries.` Your task is to implement `computeSumOfSalaries`.

Your implementation must first combine the two input arrays of `Employees` and `Managers` into a single list, appropriately typed. Then it must polymorphically compute the sum of all the salaries of all `Employees` and `Managers` in this combined list.

If you do not do this computation polymorphically (for example, by computing the sums of the two lists separately and summing the two results), you will receive no credit. The first step of your implementation *must* combine the two lists into a single list, before any sum of salaries is computed.

You may wish to use the test data provided in the `Main` class in order to test your implementation.