MPP Pretest #2
August, 2020

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 problem, which provides skeletons for the solutions to each. Your task is to fill in the details of each of these skeletons. Do not change the names of the methods (though you may add new methods if you want) and do not change their signatures or access modifiers (e.g. `public`)).

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

<u>Problem 1</u>: [40 %] [Recursion] In this problem you will write a recursive solution to the problem of searching for a value in a Stack.

In your `prob1` package, you will find a class `IntStack` for storing Integers; the class contains fully implemented public methods `pop`, `peek`, and `push`. You will also find a `StackInfo` class that stores two instances of `IntStack`: `mainStack` and `tempStack`. The `mainStack` instance provides data for the `StackInfo` class; it is the stack that is accessed when `StackInfo`'s `addValue` method is called. The `tempStack` instance is used for temporary storage during an operation; whenever an operation has completed, the `tempStack` must be empty.

`StackInfo` also contains one unimplemented method

```
public boolean find(Integer num)
```

The `find` method reads the `mainStack` and returns `true` if the `Integer num` is found, `false` otherwise. In case the `Integer` input is `null`, the `find` method returns false.

The states of the two stacks must *not* be altered by the `find` method. This means that the number and order of elements in the `mainStack` *after* `find` is called must be the same as the number and order of elements in the stack *before* `find` is called. And `tempStack` must be empty after the `find` operation has completed.

Example: Suppose the `mainStack` in a `StackInfo` object looks like this:

3
2
6
4

Suppose now that you call `find` with input argument num = 6. The `find` method will return `true` and `mainStack` in the `StackInfo` object should look like this (in other words, `mainStack` is the same as it was before `find` was called):

3
2
6
4

Also, in this example, the `tempStack` will be empty *before* `find` is called, and will be empty *after* the `find` method returns.

To carry out the recursion, use the following strategy:

- Pop the `mainStack`; if the popped `Integer` equals `num` then return `true`
- Else, recursively search `mainStack`
- If not found, return `false`.

Note that in addition to the recursive search, you will need to write code to restore the states of `mainStack` and `tempStack` (remember that after the `find` operation executes, `tempStack` must be empty and `mainStack` must be the same as it was before `find` was called).

The class `StackInfo` contains a `main` method, which you can use to test your `find` method.

Requirements for Problem 1:

(1) Your implementation of the method `find` must use recursion, following the recursive strategy described above.
(2) You may not use any kind of loops (no `for` loops, no `while` loops).
(3) You may not use any auxiliary storage classes (arrays, lists, or other data structures) ; the only storage available is `mainStack` and `tempStack` in `StackInfo`
(4) There should be no compiler or runtime errors. In the same spirit, if your code causes a stack overflow, or does not halt, you will get no credit for this problem.

Problem 2. **[Data Structures]** In your `prob2` package, you will find a fully implemented linked list class called `MyStringLinkedList`, which is a linked list for `String`s.

Your task for this problem is to implement the unimplemented method `findMin`(). The `findMin()` method returns the alphabetically least `String` in the list (using the usual ordering for Strings). If the list is empty, `findMin()` should return `null`.

A main method has been provided in `MyStringLinkedList` that you can use to test your `findMin()` method.

Example. Suppose the method `addLast` in `MyStringLinkedList` is called three times on an instance of `MyStringLinkedList` as follows:

```
MyStringLinkedList list = new MyStringLinkedList();
list.addLast("Harry");
list.addLast("Bob");
list.addLast("Steve");
```

Then, suppose you make the calls

```
String min = list.findMin();
System.out.println(min);
```

The output should be:

```
Bob
```

Requirements for this problem.

(1) You must implement the `findMin()` method in the `MyStringLinkedList` class provided for you in `prob2`, and it must work.
(2) You are *not allowed* to use Java library sorting methods (in particular, you may not make a call to `Collections.sort` or to `Arrays.sort`).
(3) You are *not allowed* to use any Java library data structures, such as `ArrayList`, `LinkedList`, or arrays. In particular, your `findMin()` method *must not* be done by copying elements from the `MyStringLinkedList` class into some other kind of data structure and then performing an operation that finds the min in this other data structure.
(4) There must not be any compilation errors or runtime errors in the solution that you submit.

Problem 3. [Polymorphism] When run, the `main` method in the `Driver` class in package `prob3_old` produces the following output:

```
/\ /\ \/ ||
```

Each of the figures displayed is represented by a separate class. For example, the figure

```
/\
```

is created by the class `HatMaker`. The `main` method in the `Driver` class creates an array of instances of these figure classes and then passes this array to the `make` method, which reads the value in each figure object and assembles them all into an ouput `String` and returns it. The `main` method then prints this output `String` to the console.

The code works but is poorly designed. The code in the `make` method tests the type of each figure class in the input array, and then downcasts to the correct type and calls this particular object's `getFigure` method in order to get the figure from the object.

For this problem, *work in the package* `prob3_new` (which contains the same classes as `prob3_old`) and modify the implementation in the `Driver` class so that polymorphism is used in the make method in order to create an output string. To do this, you will need to add one more class or interface to the `prob3_new` package – a class that generalizes the figure classes. You may make type changes to the `main` method and rework the implementation of the `make` method.

After you have made the necessary code modifications, running the `main` method in the `Driver` class belonging `to prob3_new` should produce the same output as the `Driver` class belonging to `prob3_old`.

NOTE: All your new code for this problem *must be placed in the package* `prob3_new`. Graders of this exam will not have access to the package `prob3_old`, so *do not put your new code in that old package!*

Requirements for this problem.

(1) The make method in your new version of the `Driver` class must use polymorphism to create the output `String` that is passed back to the `main` method.

(2) The input argument for make must be an <u>array</u> consisting of instances of `HatMaker`, `ParallelMaker`, and `VeeMaker`.

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