CSE 595 : Programming Abstractions – Homework II

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This homework document consists of 4 pages, and consists of two sections.

The first section focuses on polymorphism, and does not require you to write a lot of new code. Instead, write the correct amount of code carefully within a given framework. The concepts being explored in this assignment are the different kinds of abstraction that you should be familiar with in Java, together with the use of subtype polymorphism and parametric polymorphism.

Carefully read the entire document and the given codebase before you start coding. A lot of relevant information is provided in the codebase in the form of comments and documentation. This homework is designed in a way where the best implementation requires you to start with the big picture and only then begin to implement the details. If you start with implementing specific parts right away, you may end up in a situation where the complete codebase in the end does not work as intended.

Development Environment: It is highly recommended that you use IntelliJ IDEA. You can avail it from `https://www.jetbrains.com/idea/` by either downloading the free community edition, or create a student account and get access to the Ultimate edition for free for one year (after that, you can renew it if you want).

You can, if you really want, use a different IDE, but if things go wrong there, you may be on your own.

Programming Language: Starting with this homework, and for the remainder of this course, all Java code *must* be JDK 1.8 compliant. That is, you may have a higher version of Java installed, but the "language level" must be set to Java 8. This can be easily done in IntelliJ IDEA by going to "Project Structure" and selecting the appropriate "Project language level". *This is a very important requirement, since Java 9, 10, and 11 all have additional language features that will not compile with a Java 8 compiler.*

1 Polymorphism [60 points]

There is some amount of code already given to you, and a large part of this is very similar to the code examples we covered in the lectures on polymorphism. In the given code, you will find the following abstract classes or interfaces (shown in italics)

- *AbstractPrinter*, *Point*, *Positionable*, *TwoDShape*,

and classes

- Printer, Circle, Triangle, Quadrilateral, TwoDPoint, ThreeDPoint, Ordering.

The main driver class is `Ordering`, where a sample main method is provided. This code is extensively commented for you, and divided into three main sections. Please read this very carefully. This method also serves as a test scenario for your complete code.

The given code is incomplete, and several places are marked with a "TODO" tag. Your task is to follow the documentation and the comments to complete the codebase. A few things to keep in mind during this code completion:

- The assignment uses the `Comparable` and `Comparator` interfaces. The comments in the main driver class will illustrate how they are used, and why both are needed.
- Any fields you need/want to add to the classes must be private, with the corresponding `getter()` and `setter()` methods added by you.
- You may, in general, add more methods or fields to the classes if you feel the need. But *do not modify any class or interface* where the comment specifies not to change the code.

1. Completion of `Circle` code:`Circle#setPosition(List)`, and `Circle#getPosition()`, `Circle#area()`, and `Circle#perimeter()`. (2)

2. Completion of `Triangle` code:
   (a) Triangle constructor (1)
   (b) `Triangle#setPosition(List)` (2)
   (c) `Triangle#getPosition()` (2)
   (d) `Triangle#area()` (2)
   (e) `Triangle#perimeter()` (2)
   (f) `Triangle#snap()` (2)
   (g) `Triangle#isMember()` (2)

3. Completion of `Quadrilateral` code:
   (a) Triangle constructor (1)
   (b) `Quadrilateral#setPosition(List)` (2)
   (c) `Quadrilateral#getPosition()` (2)
   (d) `Quadrilateral#area()` (2)
   (e) `Quadrilateral#perimeter()` (2)
   (f) `Quadrilateral#snap()` (2)
   (g) `Quadrilateral#isMember()` (2)

4. Completion of `TwoDPoint` code: TwoDPoint constructor, `TwoDPoint#coordinates()`, and `TwoDPoint#ofDoubles(double...)`. (3)

5. Completion of `ThreeDPoint` code: ThreeDPoint constructor, and `ThreeDPoint#coordinates()`. (2)

6. Fix the definitions of the two lists in the `main(String[])` method: `shapes` and `points`. (2)

7. Completion of the portion required to run "SECTION 1":
   (a) Fix the `Ordering#copy(List, List)` method so that it works with `note-1` in the given code. (2)
   (b) Complete the `XLocationShapeComparator` implementation as per the given documentation, so that the call to `shapes.sort(new XLocationShapeComparator())` works as expected. (2)
   (c) Finish the required implementation in the codebase so that the call to `Collections.sort(shapes)` works as expected. The expected behavior of this sorting is provided in the "TODO" comment. (2)
   (d) The call to `points.sort(new XLocationPointComparator())` should work as expected. The expected behavior of this comparator is provided in the "TODO" comment. (2)
   (e) Implement the natural ordering so that the call to `Collections.sort(points)` works. The natural ordering is described in the "TODO" comment. (2)

8. Completion of the portion required to run "SECTION 2". For this, you must fix the `Ordering#copy(List, List)` method so that it works with `note-2`, `note-3`, and `note-4` as well. (3)

9. Completion of the portion required to run "SECTION 3":
   (a) What methods do you need to override in each class such that the printer automatically prints the human-readable strings for each object? Implement this for the `Circle`, `Triangle`, and `Quandrilateral` classes. The expected string representations are explained in the comments provided with the code. (6)
   (b) What must you implement so that the `printAllAndReturnLeast(List, AbstractPrinter)` method can make sense of the notion of "least"? Think carefully about where you must implement this so that call to `printAllAndReturnLeast(lst, new Printer())` in the main method works. (4)

(c) The `printAllAndReturnLeast(lst, new Printer())` will compile after your above implementation, but there are other aspects of this method that need to be fixed. This has to do with introducing the correct generic parameters. Once you do this, the method will run properly. (2)

The `Ordering#main(String[])` method serves as a test code of sorts. If, after following the instructions in the comments, your code does not compile or run properly, then it indicates a definite error in your implementation(s). Keep in mind, though, that a complete suite of tests is not provided here. You are, of course, free to add your own tests to check whether or not your completed code is running as expected.

2 Functional Programming [40 points]

```
return sequence.stream()
              .intermediate_operation_1(...)
              .intermediate_operation_2(...)
              .intermediate_operation_3(...).terminal_operation();
```

Example 1: A Java function implemented as a single method chain.

Each function implementation must be done using a single method chain (as shown in example 1 above), and all the functions must be implemented in a file named `StreamUtils.java`.

1. **Capitalized strings.** (5)

```
/**
 * @param  strings: the input collection of <code>String</code>s.
 * @return          a collection of those <code>String</code>s in the input collection
                    that start with a capital letter.
 */
public static Collection<String> capitalized(Collection<String> strings);
```

2. **The longest string.** (5)

```
/**
 * Find and return the longest <code>String</code> in a given collection of <code>String</code>s.
 *
 * @param strings:    the given collection of <code>String</code>s.
 * @param from_start: a <code>boolean</code> flag that decides how ties are broken.
                      If <code>true</code>, then the element encountered earlier in
 *                    the iteration is returned, otherwise the later element is returned.
 * @return            the longest <code>String</code> in the given collection,
 *                    where ties are broken based on <code>from_start</code>.
 */
public static String longest(Collection<String> strings, boolean from_start);
```

3. **The least element.** In this function, the single method chain can return a `java.util.Optional<T>`. So you must write additional code to convert it to an object of type `T` (handling any potential exceptions). (5)

```
/**
 * Find and return the least element from a collection of given elements that are comparable.
 *
 * @param items:      the given collection of elements
 * @param from_start: a <code>boolean</code> flag that decides how ties are broken.
 *                    If <code>true</code>, the element encountered earlier in the
 *                    iteration is returned, otherwise the later element is returned.
 * @param <T>:        the type parameter of the collection (i.e., the items are all of type T).
 * @return            the least element in <code>items</code>, where ties are
 *                    broken based on <code>from_start</code>.
 */
public static <T extends Comparable<T>> T least(Collection<T> items, boolean from_start);
```

4. **Flatten a map.** (5)

```
/**
 * Flattens a map to a stream of <code>String</code>s, where each element in the list
 * is formatted as "key -> value".
```

```
     *
     * @param aMap the specified input map.
     * @param <K>  the type parameter of keys in <code>aMap</code>.
     * @param <V>  the type parameter of values in <code>aMap</code>.
     * @return the flattened list representation of <code>aMap</code>.
     */
    public static <K, V> List<String> flatten(Map<K, V> aMap)
```

Code for the following questions must be written in a file named `HigherOrderUtils.java`. You may also have to consult some of the official Java documentation and/or the reference text on Functional Programming in Java. The remaining answers need NOT be written as a single method chain.

5. First, write a nested interface in `HigherOrderUtils` called `NamedBiFunction` that extends the interface `java.util.Function.BiFunction`. The interface should just have one method declaration: `String name();`, i.e., a class implementing this interface must provide a "name" for every instance of that class. (4)

6. Next, create `public static NamedBiFunction` instances as follows: (8)
   (a) `add`, with the name "add", to perform addition of two `Double`s.
   (b) `subtract`, with the name "diff", to perform subtraction of one `Double` from another.
   (c) `multiply`, with the name "mult", to perform multiplication of two `Double`s.
   (d) `divide`, with the name "div", to divide one `Double` by another. This operation should throw a `java.lang.ArithmeticException` if there is a division by zero being attempted.

7. Write a function called `zip` as follows: (8)

```
    /**
     * Applies a given list of bifunctions -- functions that take two arguments of a certain type
     * and produce a single instance of that type -- to a list of arguments of that type. The
     * functions are applied in an iterative manner, and the result of each function is stored in
     * the list in an iterative manner as well, to be used by the next bifunction in the next
     * iteration. For example, given
     *   List<Double> args = Arrays.asList(1d, 1d, 3d, 0d, 4d), and
     *   List<NamedBiFunction<Double, Double, Double>> bfs = [add, multiply, add, divide],
     * <code>zip(args, bfs)</code> will proceed iteratively as follows:
     *   - index 0: the result of add(1,1) is stored in args[1] to yield args = [1,2,3,0,4]
     *   - index 1: the result of multiply(2,3) is stored in args[2] to yield args = [1,2,6,0,4]
     *   - index 2: the result of add(6,0) is stored in args[3] to yield args = [1,2,6,6,4]
     *   - index 3: the result of divide(6,4) is stored in args[4] to yield args = [1,2,6,6,1]
     *
     * @param args:        the arguments over which <code>bifunctions</code> will be applied.
     * @param bifunctions: the list of bifunctions that will be applied on <code>args</code>.
     * @param <T>:         the type parameter of the arguments (e.g., Integer, Double)
     * @return             the item in the last index of <code>args</code>, which has the final
     *                     result of all the bifunctions being applied in sequence.
     */
     public static <T> T zip(List<T> args, List<NamedBiFunction<T, T, T>> bifunctions);
```

NOTES:

- **Uncompilable code** will not be accepted.
- Please remember to verify what you are submitting. Make sure you are, indeed, submitting what you think you are submitting!
- What to submit? A single `.zip` file comprising the completed codebase, and the two Java files `StreamUtils.java` and `HigherOrderUtils.java`. *Some parts of this assignment will be graded by a script, so be absolutely sure that the submission follows this structure.*

Submission Deadline: April 19, 2021, 11:59 pm