CSE 216 – Homework II

Instructor: Dr. Ritwik Banerjee

NOTE: This homework document consists of 4 pages. Carefully read the entire document before you start coding. 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 Parametric and subtype polymorphism [45 points]

In this first section, an incomplete codebase is given to you with many method bodies simply marked with a TODO tag. Your task is to follow the documentation provided in these methods and complete the codebase. A few things to keep in mind during this code completion:

- You will probably have to read up on two interfaces used in sorting: `Comparable` and `Comparator`. The assignment itself 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. Note that setter methods only make sense for attributes that are modifiable.
- You may, in general, add more methods to the classes if you feel the need. But *do not modify/remove any methods or fields already implemented* in the codebase. Also *do not modify any interface*.

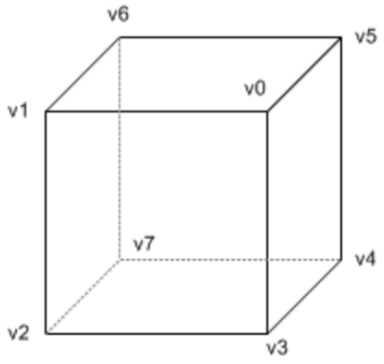

Figure 1: The expected vertex-ordering for `Cuboid`. Note that v_0 through v_3 form one face in counterlockwise order and v_4 through v_7 form the exact opposite face, where v_4 is the corner directly connected to v_3.

1. Complete the parts marked by "TODO" in the codebase.
   (a) `Circle#setPosition(List)` (2)
   (b) `Cuboid#volume()` and `Cuboid#center()` (3)
   (c) Add the method missing from `Cuboid`, defined in an interface it implements. (3)
   (d) `Quadrilateral#getSideLengths()` (3)
   (e) `Quadrilateral#setPosition(List)` (3)
   (f) `Rectangle#center()`, `Rectangle#isMember(List)`, and `Rectangle#area()` (6)

(g) Resolve the issue with the constructor in `Square`. Note that you are *not* allowed to modify the constructor itself. (2)

(h) `Square#snap()` (3)

(i) Complete the `ThreeDPoint` class as marked by the two "TODO" comments. (3)

(j) Complete the `TwoDPoint` class as marked by the two "TODO" comments. (3)

(k) Carefully consider the codes of `Cuboid` and `Circle`, and create a class `Sphere` that implements the `ThreeDShape` interface. (5)

(l) Complete the code in the class `Ordering.java`. The `main(String[])` method in this class 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)[1]. (5)

2. Create two static methods as follows:

(a) `Cuboid#random()`, which will generate a cuboid with random vertices. Please note that in spite of that randomness, the object must actually be a geometric cuboid shape. (2)

(b) `Sphere#random()`, which will generate a sphere with a random center and radius. (2)

2 Functional Programming: Streams [20 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)

[1]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.

```
/**
 * 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)
```

3 Functional Programming: Higher-order functions [30 points]

Code for this section 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.

1. 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)

2. 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.

3. Write a function called `zip` as follows: (9)

```
/**
 * 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);
```

4. Based on the above `zip` function, think about what a function composition would look like. Write a static inner class called `FunctionComposition` that is parameterized by three type parameters. This class should have no methods, and no constructor. It should only have a single `BiFunction` called `composition`, which takes in two functions and provides their composition as the output function. Function composition should be consistent with the types – if there is a function `f: char -> String`, and another function `g: String -> int`, the output of composition should be a function `h: char -> int`. For example, if `f` concatenates a `char` some number of times (say, 'b' yields "bb", 'c' yields "ccc", 'd' yields "dddd", etc.), and `g` converts a string to its length, then `composition(f, g)` should output a function that maps 'z' to 26. (9)

4 Version Control [5 points]

As with the first homework before this, you must use the Bitbucket account for version control. For this, create a **new repository** on Bitbucket with the name `CSE216HW2`, and give access to your grading TA. Yyour grading TA will be allocated based on your last name but the allocation may be different for this homework, so *do not* invite a TA to your repository until the grading TA allocations are announced (on Blackboard) for this assignment. Access must be given to your grading TA before the homework deadline. Otherwise, this component of the homework will not be graded. We strongly encourage you to commit and push your code to this remote repository every time you finish solving any question, and even more frequently for the longer questions. Your grading TA will check for the following:

1. There must be at least 4 distinct code pushes to the remote repository, with each push safely committing the solution to at least one new solution to a homework question. (2)
2. At least 2 code pushes to the remote repository must be made on separate days. (1)
3. At least 2 code pushes must include meaningul commit messages describing what code has been contributed since the previous version. (2)

NOTES:

- **Late submissions** or **uncompilable code** will not be graded.
- 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 (i) the completed codebase, (ii) `StreamUtils.java`, and (iii) `HigherOrderUtils.java`. **This assignment will be graded by a script, so be absolutely sure that the submission follows this structure**.

Submission Deadline: March 23, 2020, 11:59 pm