CSE 216 : Programming Abstractions – Homework III

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This homework document consists of 3 pages, and focuses on functional programming using Java. While doing this assignment, you will notice that to use the functional programming paradigm with Java, you will also need to be very careful about the use of parameterized types.

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 and beyond all have additional language features that will not compile with a Java 8 compiler.*

1 Simple functional programming in a single method chain

```
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.** (12)

```
/**
 * @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.** (12)

```
/**
 * 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 something extra (your final code should still be a single method chain) to convert it to an object of type `T` (handling any potential exceptions). (12)

```
/**
 * 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.** (12)

```
/**
 * Flattens a map to a stream of <code>String</code>s, where each element in the list
 * is formatted as "key -> value" (i.e., each key-value pair is converted to a string
 * with this format).
 *
 * @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)
```

2 Higher-order functions

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.

Note: be very careful with the parameterized types in this section, and pay attention to the warnings issued by your IDE regarding the raw type.

5. First, write a static nested interface in `HigherOrderUtils` called `NamedBiFunction`. This interface must extend 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. (8)

6. Next, create `public static NamedBiFunction` instances as follows: (16)
   (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: (16)

```
/**
 * 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 = [1,2,6,6,1.5]
*
* @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);
```

8. 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. (12)

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 consisting of the two Java files `StreamUtils.java` and `HigherOrderUtils.java`. *In this assignment, your entire code must be in these two files only. No additional files should be there in the submitted code..*

Submission Deadline: Nov 28, 2021, 11:59 pm