NATIONAL UNIVERSITY OF SINGAPORE
Department of Computer Science, School of Computing
IT5100B—Industry Readiness: Stream Processing
Academic Year 2023/2024, Semester 2
ASSIGNMENT 1

GENERIC PROGRAMMING IN JAVA

Due: 9 February 2024, 11.59pm **Score**: [20 marks] (20%)

INSTRUCTIONS

For this assignment, you will need: (1) An IDE or text editor to write, compile and execute Java programs with JDK 8+ and (2) templates, utilities and testing code (bundled together with this document).

This assignment is worth 20% of your overall grade. The objectives of this assignment are:

- To introduce you to the `Stream` and `Optional` APIs.
- To introduce you to declarative programming.
- To help you practice reading and using other APIs.

The assignment is split into two parts, (1) generic programming with Streams (`StreamExercises.java`) and (2) generic programming with Optionals (`Historical.java`). Each part is worth 10% of your overall grade. You should only submit two files: `Historical.java` and `StreamExercises.java`. You can use the `...Test.java` files to run some simple tests. Contact `yongqi@nus.edu.sg` if you have queries.

GENERIC PROGRAMMING WITH STREAMS [10 marks]

The `java.util.stream` Application Programming Interface (API) gives us a nice set of tools to work with streams declaratively. In this part, we will be working with streams, so you should minimize use of for loops and recursion.

Question 1 (Happy Sum) [1 mark]. The happy sum of n is defined to be

$$n^2 + (n-1)^2 + \cdots + 4 + 1 + 4 + \cdots + (n-1)^2 + n^2$$

Complete the `happySum` method that receives `n` and produces its happy sum. You may assume that `n` is positive. You should make use of the `IntStream::rangeClosed` method to generate a stream of integers from 1 to n (inclusive). Example runs follow:

```
System.out.println(happySum(1)); // 1
System.out.println(happySum(5)); // 109
```


Question 2 (Windowing) [1 mark]. A common stream operation that you might see being used is to create a *sliding window*. For example, if we have a stream of integers from 1 to 5, a sliding window of size 3 might look like this (in square brackets):

```
[1 2 3] 4 5
1 [2 3 4] 5
1 2 [3 4 5]
```

thus, turning our stream `[1 2 3 4 5]` into a stream of streams `[[1 2 3] [2 3 4] [3 4 5]]`.

Complete the `window` method which receives a finitely-large stream `stream` and a window size `windowSize`, and produces a stream of sliding windows over `stream` with size `windowSize`. If `windowSize` is not positive, return an empty stream.

This windowing operation need not be lazy. You may wish to collect the entire stream into a list first (see `Stream::collect`) and produce the stream of sliding windows from there. Example runs follow.

```java
public static <T> List<List<T>> streamsToLists(Stream<Stream<T>> s) {
    return s.map(x -> x.collect(Collectors.toList()))
            .collect(Collectors.toList());
}
Stream<Integer> s = IntStream.range(1, 6).boxed();
Stream<Stream<Integer>> ss = window(s, 3);
System.out.println(streamsToLists(ss));
// [[1, 2, 3], [2, 3, 4], [3, 4, 5]]
```

Question 3 (Moving Averages) [1 mark]. Let us try to use our `window` method from earlier. Suppose we have an IoT sensor recording air temperatures every day. What we can do is to transform a stream of temperatures into a stream of n-day moving average temperatures by:

1. Windowing the stream of temperatures with window size n.
2. For each window, obtain the average.

Complete the `movingAverage` method that receives a `DoubleStream`[1] of temperatures and `n`, and produces a stream of `n`-day moving average temperatures. Example runs follow:

```java
DoubleStream temps = DoubleStream.of(1, 2, 3, 4, 5);
List<Double> ls = movingAverage(temps, 3)
        .collect(ArrayList::new, List::add, List::addAll);
System.out.println(ls);
// [2.0, 3.0, 4.0]
```

[1] To convert a `DoubleStream` into a `Stream<Double>` you can use `DoubleStream::boxed`.

Question 4 (Bowling) [2 marks]. In this question, we are going to see how we can perform *sequential folds* over streams. Hopefully, you've played ten-pin bowling before. From Wikipedia: "*Ten-pin bowling is a type of bowling in which a bowler rolls a bowling ball down a wood or synthetic lane toward ten pins positioned in a tetractys (equilateral triangle-based pattern) at the far end of the lane. The objective is to knock down all ten pins on the first roll of the ball (a strike), or failing that, on the second roll (a spare).*"

A bowling game has 10 frames, each frame has up to two rolls to knock down a total of ten pins. Each roll in a bowling game is represented as a character:

1. Characters `'0'` to `'9'` denote the number of pins knocked down in a roll.
2. `'/'` denotes a spare, e.g. if the previous roll (which is the first roll of the current frame) is a `'4'`, then this roll knocked down the remaining 6 pins, thus the current frame is a spare.
3. `'X'` denotes a strike, i.e. the bowler knocked down all ten pins in the first roll of the frame.

For this question, **we are going to use a simplified scoring system**: a strike gives 30 points, a spare gives 20 points, and the score for all other frames is the total number of pins knocked down (which could be zero). Thus, the frame `'3'` `'1'` gives 4 points, the frame `'6'` `'/'` gives 20 points, and the frame `'X'` gives 30 points.

Fortunately, all this computation has been provided for you. The **immutable** `BowlingGameStatistics` class provides functionality for keeping track of a bowling game. The `put` method receives a roll (as a character) and gives a new `BowlingGameStatistics` object with the new total score; it even keeps track of spares! The `get` method gets the current total score of the game tracked so far.

```
BowlingGameStatistics b = new BowlingGameStatistics();
System.out.println(b.put('3').get()); // 3
System.out.println(b.put('3').put('6').get()); // 9
System.out.println(b.put('3').put('/').get()); // 20
System.out.println(b.put('X').put('3').put('0').get()); // 33
```

Your task is to complete the `totalScore` method that receives a sequential stream of characters representing the rolls of a valid[2] bowling game. As output, return the total score of the game. Example runs follow:

```
Stream<Character> game = Stream.of('3', '2', '0', '0', '4', '/',
        'X', 'X', 'X', 'X', 'X', 'X', 'X');
System.out.println(StreamExercises.totalScore(game)); // 235
```

Tip: You might want to use the `<U> U reduce(U, BiFunction<U, ? super T, U>, BinaryOperator<U>)` method since we are doing a general fold over the stream. We are going to ignore parallelism, so your `BinaryOperator<U>` function can simply be `(x, y) -> x`.

[2] Valid meaning that there are ten frames and a frame does not start with `'/'`.

Question 5 (Event Streams) [2 marks]. One of the key ideas taught in this course is that *data stores are aggregates over streams of state changes*. We are going to demonstrate this idea with a simple example on how a stream of state change events of users in a simple online banking system can be aggregated into a datastore. Better still, if we design our operations correctly, our aggregation can be *parallelized*.

You are provided with a `User` class that represents a user of our online banking system. Along with this are the `UserStateChange` abstract class along with two concrete classes: `UserNameChange` and `UserAccountBalanceIncrease`. These classes represent changes in a user's state.

To parallelize our aggregation over a stream of state changes, we require *stateless* operations with *no side-effects*. As such, we shall keep all our classes **immutable**. However, our datastore is in the form of a `Map` (like Python dictionaries) mapping user IDs to `User`s, but `Map`s are mutable. Thus, you are also provided a minimal immutable wrapper around `Map`s (called `ImmutableMap`) that has support for *some* map operations (it has those that you need). Operations that are typically mutable for `Map`s do not cause state change on `ImmutableMap`s; instead, they produce a new `ImmutableMap` that contains the updated state.

To collect the stream of state changes into a datastore, we need to perform a reduction over the stream. Of course, the devil is in the details. Our reduction has three components:

1. The `identity`. This is the empty `ImmutableMap`, which we will add `User`s to as we traverse the stream of state changes. Notice that the identity is indeed an identity of the combiner (see the `java.util.stream` documentation for more details).
2. An `accumulator`. This takes an `ImmutableMap` and processes each `UserStateChange` sequentially (this is our reduction operation). When it encounters a state change on a user, it looks up the target user in the `ImmutableMap` (creating an empty one if it doesn't exist), updates the user (see `UserStateChange::changeUserState`) and 'adds' the user back into the `ImmutableMap`.
3. A `combiner`. This is required because we are performing a parallel reduction. When doing a parallel `reduce`, the stream will be partitioned and each partition will be accumulated separately into `ImmutableMap`s using the `identity` and `accumulator`. This `combiner` is then invoked to combine the results of two partitions into a single result. This operation must be *associative*. For this, given two maps `left` and `right`, we can add all users in `right` into `left`. However, if there a user in `right` that has the same ID as a user in `left` (these are two states of the same user), then we can combine the two user states into one that reflects the latest state using `leftUser.combineWith(rightUser)` (this operation is not commutative!).

Question 5 (i) (Accumulator) [1 mark]. Complete the `changeStateOfUserInMap` method which acts as our accumulator for reduction. It receives the current datastore `map` and a user state change even `u`. As output, it produces a new `ImmutableMap` as if `u` has updated the state of the target user in the map. Example runs follow:

```java
ImmutableMap<String, User> im = ImmutableMap.empty();
im = changeStateOfUserInMap(im, new UserNameChange("1", "Bob"));
System.out.println(im); // {1={id: 1, name: Bob, accountBalance: 0}}
im = changeStateOfUserInMap(im, new UserAccountBalanceIncrease("1", 100));
System.out.println(im); // {1={id: 1, name: Bob, accountBalance: 100}}
```

Question 5 (ii) (Combiner) [1 mark]. Complete the `combineMaps` method which acts as our combiner for reduction. It receives two maps `im1` (left) and `im2` (right), and combines them into a single map containing the most updated states of all users in these maps. One way you can do this is with the `ImmutableMap::reduceEntries` method which performs a reduction over all key-value pairs of an `ImmutableMap`. To combine two users to get the most updated state, use `leftUser.combineWith(rightUser)`. Example runs follow:

```
// left partition
ImmutableMap<String, User> im1 = ImmutableMap.empty();
im1 = changeStateOfUserInMap(im1, new UserNameChange("1", "Bob"));
im1 = changeStateOfUserInMap(im1, new UserAccountBalanceIncrease("1", 500));
// right partition
ImmutableMap<String, User> im2 = ImmutableMap.empty();
im2 = changeStateOfUserInMap(im2, new UserAccountBalanceIncrease("1", -200));
// combine
System.out.println(combineMaps(im1, im2));
// {1={id: 1, name: Bob, accountBalance: 300}}
```

Once you have done these, our `collectToDb` method should be complete! Try creating a stream of user state changes and run the `collectToDb` method. Does it work? (It should, you might want to debug your accumulator and combiner if it doesn't).

Question 6 (Programming Style) [3 marks]. You will receive the remaining three points if your solution is well-written. All your code should be declarative and do not contain loops or recursion.

GENERIC PROGRAMMING WITH OPTIONALS [10 marks]

What better way to learn about generic programming than to write our own generic class! In this section, we are going to program a very simple container class that keeps track of how an object changes with each operation. Think of this class as containing an object with its history (just like our internet browser :O). This allows us to have a look at how an object changes over time, and allows us the possibility of undo-ing an operation, if need be.

You are given a template containing an example implementation with some blanks. If you prefer to implement this class in a different way, feel free to do so. However, to get you comfortable with writing code declaratively (which you should really get used to if you want to do stream processing), we require that your code should, as much as possible, be declarative. Therefore, minimize use of imperative programming constructs like for loops, if-else statements and so forth. Recursion is okay :)

The Historical Class

The `Historical<T>` class is a wrapper class that keeps track of state changes to an object[3]. `Historical` objects are **immutable**. The class mainly supports the following operations:

Initialization and Collapsing Creating a `Historical` object can be done with the `of()` method. We can retrieve the object stored in a `Historical` object with the `get()` method:

```
Historical<Integer> h1 = Historical.of(1);
Historical<String> h2 = Historical.of("hello!");
Historical<?> h3 = Historical.of(null);

System.out.println(h1); // 1
System.out.println(h2); // hello!
System.out.println(h3); // null

Optional<Integer> i = h1.get();
Optional<String> s = h2.get();
Optional<?> n = h3.get();

System.out.println(h1); // Optional[1]
System.out.println(h2); // Optional[hello!]
System.out.println(h3); // Optional.empty
```

Replacement With the `replace` method we can change the value that is contained in the `Historical` object. However, since the `Historical` class is **immutable** and we are keeping track of state changes, by 'replacing' the value in the `Historical` object, what we are really doing is returning a new `Historical` object containing the new value, while still retaining its history:

```
Historical<Integer> h1 = Historical.of(1);
System.out.println(h1); // 1
Historical<Integer> h2 = h1.replace(h1.get().get() + 1);
System.out.println(h2); // 1 -> 2
Historical<String> h3 = h2.replace("hello!");
System.out.println(h3); // 1 -> 2 -> hello!
// Notice h1 and h2 did not change
System.out.println(h2); // 1 -> 2
System.out.println(h1); // 1
```

However, using `replace` on its own is clunky; thus it shall also support common declarative operations like `map`, `filter` and `flatMap`:

[3] The type parameter `T` is the type of the current value of the object. For example, if the current value of the tracked object is `1`, then the `Historical` object that is tracking it should have type `Historical<Integer>`.


```
// just for the flatMap example
Function<String, Historical<Integer>> f = x -> Historical.of(x + " = 1")
        .map(String::length);

Historical<Integer> h = Historical.of(1) // 1
        .map(x -> x + "!")               // 1!
        .filter(x -> x.length() > 3)     // null
        .replace("1!")                   // 1!
        .flatMap(f);                     // 6

System.out.println(h); // 1 -> 1! -> null -> 1! -> 1! = 1 -> 6
```

Undoing Changes Naturally, the `Historical` class is amenable to an `undo()` operation that gives us the previous `Historical` value:

```
Historical<String> h = Historical.of(1).replace("hello!");
Optional<Historical<?>> u1 = h.undo();
System.out.println(u1); // Optional[1]
Optional<Historical<?>> u2 = u1.get().undo();
System.out.println(u2); // Optional.empty
```

Questions

Question 7 (The Basics) [2 marks]. Complete the following methods:

- The class-level `of()` method produces a new `Historical` object that begins tracking changes to that value. This value can be `null`.[4]
- The `get()` method returns the current value of the stored object as an `Optional`; this is because the current value may be `null` (nothing).

Note that if you are using your own implementation of `Historical`, you will need to provide your own `toString` and `equals` implementations.

Example runs can be seen above.

Question 8 (Replacement) [1 mark]. Complete the `replace` method which receives a new value and creates a new `Historical` object whose current value is the new value, while keeping track of its history. The new value can be `null`. If the new value is equal to the current value, `this`[5] is returned, i.e. if there are no changes to the state, nothing is appended to the history (just like how refreshing a

[4] See `Optional::ofNullable` and `Optional::empty`.
[5] You will have to typecast `this` into a `Historical<R>`.

page on your browser doesn't add to the page history). This should be the only part of your solution that contains if-else statements. Example runs follow:

```
Historical<Integer> h1 = Historical.of(1);
Historical<String> h2 = h1.replace("hello!");
Historical<Object> h3 = h2.replace(null);
System.out.println(h1); // 1
System.out.println(h2); // 1 -> hello!
System.out.println(h3); // 1 -> hello! -> null
System.out.println(Historical.of(1)
        .replace(1)
        .replace(null)
        .replace(null)
        .replace(2)
        .replace(null)); // 1 -> null -> 2 -> null
```

Question 9 (Map, Filter and FlatMap) [3 marks]. Complete three methods, `map`, `filter` and `flatMap` which do the following:

- `map` receives a mapping function and uses it to map the current element. If there is no current value, then nothing happens.[6]

```
Historical<String> h1 = Historical.of("hello!");
Historical<Integer> h2 = h1.map(x -> x.length() + 1);
System.out.println(h2); // hello! -> 7
Historical<Object> h3 = Historical.of(null);
System.out.println(h3.map(x -> x.toString()); // null
```

- `filter` receives a predicate, and if the current element passes the predicate, nothing happens, otherwise, it is replaced with `null`. [7]

```
System.out.println(Historical.of(1)
    .filter(x -> x % 2 == 1)); // 1
System.out.println(Historical.of(1)
    .filter(x -> x % 2 == 0)); // 1 -> null
System.out.println(Historical.of(1)
    .filter(x -> x % 2 == 0)); // 1 -> null
    .filter(x -> x == x)); // 1 -> null
```

- `flatMap` receives a mapping function that maps the current element into a `Historical` object and uses it to map the current element, giving us a `Historical<Historical<R>>` for some `R`. To flatten this object into a `Historical<R>`, concatenate the two histories together.

[6]Hint: see `Optional::map`. You should rely on your `replace` method written earlier.
[7]Hint: see `Optional::filter`. You should rely on your `replace` method written earlier.


```java
/* 1 -> 2 flatMapped to
        2 -> 3 -> 6
   becomes
   1 -> 2 -> 3 -> 6
*/
Function<Integer, Historical<Integer>> f = x -> Historical.of(x)
    .map(i -> i + 1)
    .map(i -> i * 2);
System.out.println(Historical.of(1).replace(2).flatMap(f));
// 1 -> 2 -> 3 -> 6
System.out.println(Historical.<Integer>of(null).flatMap(f)); // null
```

Just like `replace`, consecutive duplicates are ignored.

Question 10 (Undo) [1 mark]. Create an `undo` method that produces the previous value of a `Historical` object. This method should be relatively straightforward to implement.

Question 11 (Programming Structure) [3 marks]. You will receive the remaining three points if your solution is well-written and well-designed. Some key points to take note of:

- You should write your code declaratively. Each method should (ideally) be written as a single `return` statement.
- You should avoid using constructs like if-else statements and loops. You should only need one if-else statement (or ternary expression in the form of `cond ? e_if : e_else`) for the `replace` method.

– End of Assignment 1 –

