INSTRUCTIONS

1. This Mock Practical Assessment consists of two question.
2. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
3. You should see the following in your home directory.
    - The files `Test1.java`, `Test2.java`, ..., and `CS2030STest.java` for testing your solution.
    - The files `Test1.jsh`, `Test2.jsh`, ... for additional testing on jshell.
    - The skeleton files for Question 1: `cs2030s.fp.Reader.java`.
    - The following files to solve Question 1 is provided as part of the `cs2030s.fp` package: `Immutator.java`.
    - The skeleton files for Question 2: `Main.java`.
    - The following files to solve Question 2 is provided for you: `Utilities.java`.
4. Solve the programming tasks by editing the given file. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you created or edited as part of the @author tag. Do not write your name.
7. Important: Make sure all the code you have written compiles. If one of the Java files you have written causes any compilation error, you will receive 0 marks for that question.

API Reference

`List` and `ArrayList`

You may need to use the interface `List<T>` and its implementation `ArrayList<T>` for both Question 1 and Question 2. Some useful methods are:

- `boolean add(T item)`: append the item to the end of the list
- `boolean addAll(T item)`: inserts all of the elements in the specified collection into this list at the specified position
- `boolean contains(T item)`: returns if an item is in the list
- `T get(int index)`: retrieve the item at the specified index without removing the item. The first

element has an index of 0.

- `T remove(int index)`: retrieve the item at the specified index and remove the item from the list. The first element has an index of 0.
- `boolean remove(Object item)`: removes the passed item from the list.
- `int size()`: return the number of elements in the list.

You can create a new empty `ArrayList` with `new ArrayList<T>()`. You can import `List` using `import java.util.ArrayList;` and `ArrayList` using `import java.util.ArrayList;`.

Stream

You need to use the interface `Stream<T>` and its implementation `IntStream<T>` or `LongStream<T>` for Question 2. Some useful methods are:

- Data Source:
  - `static <T> Stream<T> of(T t)`: returns a sequential `Stream` containing a single element
  - `static <T> Stream<T> of(T... values)`: returns a sequential ordered `Stream` whose elements are the specified values
  - `static <T> generate(Supplier<? extends T> s)`: returns an infinite sequential unordered `Stream` where each element is generated by the provided `Supplier`
  - `static <T> iterate(T seed, UnaryOperator<T> f)`: returns an infinite sequential ordered `Stream` produced by iterative application of a function `f` to an initial element seed, producing a `Stream` consisting of `seed, f(seed), f(f(seed)),` etc
  - `static <T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)`: returns a sequential ordered `Stream` produced by iterative application of the given `next` function to an initial element `seed`, conditioned on satisfying the given `hasNext` predicate
- Intermediate:
  - `<R> Stream<R> map(Function<? super T,? extends R> mapper)`: returns a stream consisting of the results of applying the given function to the elements of this stream
  - `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`: returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element
  - `Stream<T> filter(Predicate<? super T> predicate)`: returns a stream consisting of the elements of this stream that match the given predicate
  - `Stream<T> limit(long maxSize)`: returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length
  - `Stream<T> takeWhile(Predicate<? super T> predicate)`: returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given predicate

  - `Stream<T> dropWhile(Predicate<? super T> predicate)` : returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate
- Terminal:
  - `void forEach(Consumer<? super T> action)` : performs an action for each element of this stream
  - `boolean allMatch(Predicate<? super T> predicate)` : returns whether all elements of this stream match the provided predicate
  - `boolean anyMatch(Predicate<? super T> predicate)` : returns whether any elements of this stream match the provided predicate
  - `boolean noneMatch(Predicate<? super T> predicate)` : returns whether no elements of this stream match the provided predicate
  - `long count()` : returns the count of elements in this stream
  - `<R, A> R collect(Collector<? super T,A,R> collector)` : performs a mutable reduction operation on the elements of this stream using a `Collector`
    - In particular, `Collectors.toList()` will be useful
    - To use `Collectors`, you need to import `java.util.stream.Collectors`
  - `T reduce(T identity, BinaryOperator<T> accumulator)` : performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value
  - `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)` : performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions

Question 1: Reader

We implemented a `Loggable` class in lecture before. `Loggable` is actually classified as a writer monad since it writes into the log. Now, we are going to make a reader monad. In fact, if you combine both reader and writer monads, you get the IO monad of Haskell.

In this question, we simply require `Reader<T>` to be immutable. Since `Reader<T>` may contain a field of type `T`, we do not require the type `T` to be immutable as well.

The Basic

Create an *immutable* `Reader<T>` class that encapsulates a sequence of "inputs" to be read as part of the `cs2030s.fp` package. We put quotes around inputs as it is not a real user input. But you can imagine that we can wrap a `Scanner` class (out of syllabus) to do this instead.

We want to differentiate between an "empty" reader and a "non-empty" reader. An empty reader has no inputs while non-empty reader has some inputs that can be read.

- A `Reader<T>` object can be created using the static factory `of` method, passing in a sequence of inputs as a variable number of arguments. We always read the inputs from the leftmost arguments first.
    - If the `of` method has an input argument, a non-empty reader will be created.
    - If the `of` method has no input argument, an empty reader will be created.
- Implement a `read()` method to retrieve the first value to be read. If there is no such value, the method should throw `java.util.NoSuchElementException`.
- Implement a `hasNext()` method that returns `true` if it is a non-empty reader and `false` if it is an empty reader.
- Implement a `toString()` method that returns a `String` "Reader" if it is a non-empty reader and "EOF" if it is an empty reader.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Reader

jshell> Reader<Integer> intInput = Reader.of(2, 0, 3, 0)
intInput ==> Reader
jshell> intInput.read()
$.. ==> 2
jshell> intInput.read() // always the same
$.. ==> 2
jshell> intInput.hasNext()
$.. ==> true

jshell> Reader<Object> noInput = Reader.of()
noInput ==> EOF
jshell> noInput.read()
|  Exception java.util.NoSuchElementException
jshell> noInput.hasNext()
$.. ==> false

jshell> Reader<String> strInput = Reader.of("CS", "2030", "S")
strInput ==> Reader
jshell> strInput.read()
$.. ==> "CS"
jshell> strInput.read() // always the same
$.. ==> "CS"
jshell> strInput.hasNext()
$.. ==> true
```

You will need to make `of` accepts a variable number of arguments (*commonly known as varargs*). Recall that using generic type with varargs will lead to a compiler warning about heap pollution. If you are sure that you are handling the generic type correctly, you can suppress this warning with `@SafeVarargs` annotation.

You can also test your code with `Test1.java`:

```
$ javac Reader.java
$ javac Test1.java
$ java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
Reader.java
$ javadoc -quiet -private -d docs Reader.java
```

Consuming Input

Now we want to be able to consume the input.

- Implement a `consume()` method to read past the first value. This method should return a new `Reader<T>` with the inputs starting from the next input while keeping the current instance *unchanged*. If there is no such value, the method should throw `java.util.NoSuchElementException`.
- Add an `equals` method to compare if two `Reader` objects are equals. Two non-empty `Reader`s are always equal and two empty `Reader`s are always equal. On the other hand, a non-empty reader is not equal to an empty reader.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Reader

jshell> Reader<Integer> intInput = Reader.of(2, 0, 3, 0)
intInput ==> Reader
jshell> intInput.consume().read()
$.. ==> 0
jshell> intInput.consume().consume().read()
$.. ==> 3
jshell> intInput.consume().consume().consume().read()
$.. ==> 0
jshell> intInput.consume().consume().consume().consume().read()
|  Exception java.util.NoSuchElementException
jshell> intInput.consume().consume().consume().consume().consume()
|  Exception java.util.NoSuchElementException

jshell> Reader<Object> noInput = Reader.of()
noInput ==> EOF
jshell> noInput.consume()
|  Exception java.util.NoSuchElementException

jshell> Reader.of("CS", "2030", "S").equals(Reader.of(2, 0, 3, 0))
$.. ==> true
jshell> Reader.of("CS", "2030", "S").equals(Reader.of())
$.. ==> false
jshell> Reader.of().equals(Reader.of(2, 0, 3, 0))
$.. ==> false
jshell> Reader.of().equals(Reader.of())
$.. ==> true
```

You can also test your code with `Test2.java`:

```
$ javac Reader.java
$ javac Test2.java
$ java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
Reader.java
$ javadoc -quiet -private -d docs Reader.java
```

Map

Now that we have a reader and its inputs (*or empty*) encapsulated within `Reader` class, let's add the ability to manipulate the value that we can read. Add a `map` method that takes in an `Immutator` to update all its input read.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Reader

jshell> Reader<String> strInput = Reader.of("CS", "2030", "S")
strInput ==> Reader
jshell> Reader<Integer> intInput = strInput.map(s -> s.length())
intInput ==> Reader

jshell> intInput.read()
$.. ==> 2
jshell> intInput.consume().read()
$.. ==> 4
jshell> intInput.consume().consume().read()
$.. ==> 1
jshell> intInput.consume().consume().consume().read()
|  Exception java.util.NoSuchElementException
```

Now make sure your `map` method is flexible enough to handle functions other than those that take in `T` and returns `T`.

You can also test your code with `Test3.java`:

```
$ javac Reader.java
$ javac Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
Reader.java
$ javadoc -quiet -private -d docs Reader.java
```

flatMap

Now that we have a rather useful `Reader` class, we can write a method that builds up its own `Reader` object. We want to use the front of the current input as an input argument to the `Immutator`. This will then create a new `Reader` object with some additional input. We use this additional input returned by the `Immutator` to *replace* the front of the current input with this additional input. However, we will flatten it and not have a nested input.

For instance, if the current input is `[1, 2, 3, 4]`. We use `1` as an input argument to the `Immutator`. If this creates something like `[10, 20, 30]`, then the final result should have `[10, 20, 30, 2, 3, 4]` as inputs.

Note in the sample run below what happens when the function taken in by `flatMap` returns an empty input.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Reader

jshell> Reader<Integer> intInput = Reader.of(1, 2)
intInput ==> Reader
jshell> intInput = intInput.flatMap(x -> Reader.of(x * 10, x * 20, x * 30))
intInput ==> Reader

jshell> intInput.read()
$.. ==> 10
jshell> intInput.consume().read()
$.. ==> 20
jshell> intInput.consume().consume().read()
$.. ==> 30
jshell> intInput.consume().consume().consume().read()
$.. ==> 2
jshell> intInput.consume().consume().consume().consume().read()
|  Exception java.util.NoSuchElementException
jshell> intInput.consume().consume().consume().consume().consume()
|  Exception java.util.NoSuchElementException
```

Your `flatMap` method only need to handle those functions that take in `T` and returns `Reader<T>`. However, it should be flexible enough to handle functions that returns `Reader<U>` where `U` is a subtype of `T`. Otherwise, we cannot merge the inputs.

You can also test your code with `Test4.java`:

```
$ javac Reader.java
$ javac Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml Reader.java
$ javadoc -quiet -private -d docs Reader.java
```

Question 2: Stream

Your Task

In this question, you are to write two `Stream` methods. Each method should only contain a single `return` statement involving `Stream` pipeline. Nothing more. No local variables or classes can be defined.

You may call the methods you create when solving other parts of this questions.

Palindrome

A palindrome is defined as a string that reads the same forward and backward. For instance, `"anna"`, `"eve"`, and `"racecar"` are palindromes but `"adam"` and `"adi"` are not. In this question, you may assume that we only use lowercase alphabets. There will be no punctuations or other characters besides those listed below:

```
abcdefghijklmnopqrstuvwxyz
```

You are given a static method `split(String s)` to split the string `s` into its individual characters and return a `Stream` containing each individual characters. This method is inside the class `Utilities` in the file `Utilities.java`. You are not allowed to modify this method.

Write two static methods using only streams:

- `public static String reverse(String s)`: returns a reverse of the string `s`
  - You must use the method `split(String s)` to transform the string `s` into a `Stream<Character>`
- `public static Stream<String> palindrome(Stream<String> stream)`: returns a stream containing only palindromes

Study carefully how this method can be used in the examples below:

```
jshell> /open Utilities.java
jshell> /open Main.java

jshell> Main.reverse("adam")
$.. ==> "mada"
jshell> Main.reverse("eve")
$.. ==> "eve"
jshell> Main.reverse("adi")
$.. ==> "ida"
jshell> Main.reverse("racecar")
$.. ==> "racecar"

jshell> List<String> words = List.of("adam", "eve", "adi", "racecar", "madam",
"anna")
words ==> [adam, eve, adi, racecar, madam, anna]
jshell> Main.palindrome(words.stream()).forEach(System.out::println)
eve
racecar
madam
anna
```

You can also test your code with `Test5.java`:

```
$ javac Main.java
$ javac Test5.java
$ java Test5
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml Main.java
```