INSTRUCTIONS

1. This Practical Assessment consists of two questions.
2. The total mark for this assessment is 40.
3. 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).
4. You should see the following in your home directory.
   - The files `Test1.java`, `Test2.java`, ..., and `CS2030STest.java` for testing your solution.
   - The skeleton files for Question 1: `cs2030s/fp/Saveable.java`.
   - The following files to solve Question 1 are provided as part of the `cs2030s.fp` package: `Immutator.java`.
   - The skeleton files for Question 2: `Streaming.java`.
   - The following files to solve Question 2 are provided: `Pair.java`.
5. 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.
6. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
7. Write your student number on top of EVERY FILE you created or edited as part of the @author tag. Do not write your name.
8. 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(Collection<? extends E> c)` : appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation)
- `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;`.

For Question 2, you may want to generate a stream from a `List`. This can be done using: - `Stream<T> stream()` : returns a sequential `Stream` with this list as its source

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: Saveable (25 marks)

Marking Criteria

- functionality and type correctness (15 marks)
- OO design (5 marks)
- style (2 marks)
- documentation (3 marks)

Note that you need to write javadoc document for five of the methods identified below. You don't have to write javadoc for other methods besides those identified below. Additionally, you are not required to use inner/nested classes for this.

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

Motivation

We would like you to build an abstraction for a variable that keeps track of its history so that we can revert (undo) the variable to one of its previous values. We can also revert the undo with a redo operation.

We call this abstraction `Saveable<T>`. Here is an example of how it can be used:

```
Saveable<Integer> i;
i = Saveable.of(8);          // i contains value 8
i = i.map(x -> x + 1);       // i now contains value 9
i = i.undo();                // i now contains value 8 again
i = i.redo();                // i contains 9 again
```

Your Task

We break down the tasks you need to do into three sections. We suggest that you read through the whole question, and plan your solution carefully before starting.

The Basics

Create an immutable class named `Saveable<T>` as part of the cs2030s.fp package.

- Implement the `of` static factory method, which allows us to create a `Saveable<T>` containing a value of type `T`.
- Override the `toString` method from `Object` so that it returns a string of the pattern "Saveable[...]", with ... replaced with the string representation of the value stored inside the `Saveable`.
- Implement the `map` method on a `Saveable<T>`. The `map` method takes in a lambda expression (i.e., `Immutator`) and returns a new `Saveable<T>` with the value transformed by the lambda expression.
- Implement the `undo` method on a `Saveable<T>`. The `undo` method reverts the value of the `Saveable<T>` to its previous value (i.e., it "unmap" the most recent `map`).
  - If there is no `map` operation done on the `Saveable<T>`, the exception `java.util.NoSuchElementException` should be thrown.
- Implement the `redo` method on a `Saveable<T>` The `redo` method reverts the most recent undo operation.
  - If there is no `undo` operation done on the `Saveable<T>`, the exception `java.util.NoSuchElementException` should be thrown.

Write the javadoc documentation for `of`, `redo`, and `undo` in `Saveable<T>`. Since we do not require you to write javadoc for every class and method, `checkstyle` no longer warns about missing javadoc for your class and methods. If you wrote some javadoc and did not format it properly, however, `checkstyle` will still warn you to help you write proper `javadoc` comments.

Study carefully how these methods can be used in the examples below:

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

jshell> Saveable<String> u = Saveable.of("PE2")
jshell> u.map(x -> x + "!")
$.. ==> Saveable[PE2!]
```

```
jshell> u.map(x -> x + "!").map(x -> x + "?")
$.. ==> Saveable[PE2!?]

jshell> u.map(x -> x + "!").undo()
$.. ==> Saveable[PE2]
jshell> u.map(x -> x + "!").undo().map(x -> x + "?")
$.. ==> Saveable[PE2?]
jshell> u.map(x -> x + "!").map(x -> x + "?").undo()
$.. ==> Saveable[PE2!]
jshell> u.map(x -> x + "!").undo().map(x -> x + "?").undo()
$.. ==> Saveable[PE2]
jshell> u.map(x -> x + "!").map(x -> x + "?").undo().undo()
$.. ==> Saveable[PE2]
jshell> u.map(x -> x + "!").map(x -> x + "?").undo().undo().undo()
|  Exception java.util.NoSuchElementException
jshell> u.undo()

jshell> u.map(x -> x + "!").undo()
$.. ==> Saveable[PE2]
jshell> u.map(x -> x + "!").undo().redo()
$.. ==> Saveable[PE2!]
jshell> u.map(x -> x + "!").undo().redo().map(x -> x + "?")
$.. ==> Saveable[PE2!?]
jshell> u.map(x -> x + "!").undo().map(x -> x + "?").redo()
|  Exception java.util.NoSuchElementException

jshell> u.map(x -> x + "!").map(x -> x + "?").undo().redo()
$.. ==> Saveable[PE2!?]
jshell> u.map(x -> x + "!").undo().map(x -> x + "?").undo().redo()
$.. ==> Saveable[PE2?]
jshell> u.map(x -> x + "!").map(x -> x + "?").undo().undo().redo()
$.. ==> Saveable[PE2!]
jshell> u.map(x -> x + "!").map(x -> x + "?").undo().undo().redo().redo()
$.. ==> Saveable[PE2!?]
jshell> u.map(x -> x + "!").undo().redo().redo()
|  Exception java.util.NoSuchElementException

jshell> Immutator<Integer,Object> hash = o -> o.hashCode()
jshell> Saveable<Number> x = Saveable.<Number>of(4).map(hash) // should
compile
```

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

```
$ javac cs2030s/fp/Saveable.java
$ javac Test1.java
$ java Test1
$ java -jar checkstyle.jar -c cs2030_checks.xml cs2030s/fp/Saveable.java
$ javadoc -quiet -private -d docs cs2030s/fp/Saveable.java
```

equals

Override the `equals` method in `Object` so that we can compare two `Saveable<T>` and check if they are the same. Two `Saveable<T>` are equals if their values are equal.

Study carefully how these methods can be used in the examples below:

```
jshell> import cs2030s.fp.Saveable

jshell> Saveable<Integer> u = Saveable.of(10)
jshell> u.equals(u)
$.. ==> true
jshell> u.map(x -> x + 4).equals(u.map(x -> x + 4))
$.. ==> true
jshell> u.map(x -> x + 4).undo().equals(Saveable.of(14))
$.. ==> false
jshell> u.map(x -> x + 4).undo().equals(u.map(x -> x + 4).undo())
$.. ==> true
jshell> u.map(x -> x + 4).undo().redo().equals(u.map(x -> x + 4))
$.. ==> true
jshell> u.map(x -> x + 4).equals(Saveable.of(4).map(x -> x + 10))
$.. ==> true

jshell> Saveable.of("hi").equals(Saveable.of(new String("hi")));
$.. ==> true
```

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

```
$ javac cs2030s/fp/Saveable.java
$ javac Test2.java
$ java Test2
$ java -jar checkstyle.jar -c cs2030_checks.xml cs2030s/fp/Saveable.java
$ javadoc -quiet -private -d docs cs2030s/fp/Saveable.java
```

flatMap

Now, we would like to extend `Saveable<T>` with `flatMap`. `flatMap` takes in a lambda expression that returns a `Saveable<T>`.
Let's say we have a `Saveable<T>` u and a function f. Consider the expression:

```
Saveable<T> v = u.flatMap(x -> f(x));
```

Which versions of history should v inherit? From u or f(x)?

You will provide two versions of `flatMap` for `Saveable<T>`, called `flatMap1` and `flatMap2`.

`flatMap1` keeps the history from `f(x)` and ignores the history from `u`; `flatMap2` keeps the history from `u` and ignores the history from `f(x)`.

For example,

```
Saveable.of(0).map(x -> x + 1).flatMap1(x -> Saveable.of(x))
```

would give us a value 1 but with no history (i.e., `undo()` would throw `NoSuchElementException`) because `Saveable.of(x)` has not been transformed with a `map`.

On the other hand,

```
Saveable.of(0).map(x -> x + 1).flatMap2(x -> Saveable.of(x))
```

would give us a value of 1 with a previous value of 0. Calling `undo()` would revert the value to 0.

Write the javadoc documentation for `flatMap1` and `flatMap2`.

Since we do not require you to write javadoc for every class and method, `checkstyle` no longer warns about missing javadoc for your class and methods. If you wrote some javadoc and did not format it properly, however, `checkstyle` will still warn you to help you write proper `javadoc` comments.

Study carefully how these methods can be used in the examples below:

```
jshell> import cs2030s.fp.Immutator;
jshell> import cs2030s.fp.Saveable;

jshell> Immutator<Saveable<Integer>, Integer> f = x ->
Saveable.of(x).map(y -> y + 1).map(y -> y + 10);
jshell> Saveable<Integer> zero = Saveable.of(0)
jshell> zero.flatMap1(f)
$.. ==> Saveable[11]
jshell> zero.flatMap1(f).flatMap1(f)
$.. ==> Saveable[22]

jshell> zero.flatMap1(f).undo()
$.. ==> Saveable[1]
jshell> zero.flatMap1(f).undo().undo()
$.. ==> Saveable[0]
jshell> zero.flatMap1(f).flatMap1(f).undo()
$.. ==> Saveable[12]
jshell> zero.flatMap1(f).flatMap1(f).undo().undo()
$.. ==> Saveable[11]
jshell> zero.flatMap1(f).flatMap1(f).undo().undo().undo()
|  Exception java.util.NoSuchElementException

jshell> zero.flatMap2(f)
$.. ==> Saveable[11]
jshell> zero.flatMap2(f).flatMap2(f)
$.. ==> Saveable[22]

jshell> zero.flatMap2(f).undo()
$.. ==> Saveable[0]
jshell> zero.flatMap2(f).undo().undo()
```

```
|  Exception java.util.NoSuchElementException
jshell> zero.flatMap2(f).flatMap2(f).undo()
$.. ==> Saveable[11]
jshell> zero.flatMap2(f).flatMap2(f).undo().undo()
$.. ==> Saveable[0]
jshell> zero.flatMap2(f).flatMap2(f).undo().undo().undo()
|  Exception java.util.NoSuchElementException

jshell> Immutator<Saveable<Integer>, Object> hash = o -> Saveable.
<Integer>of(o.hashCode() + 10);
jshell> Saveable.<Number>of(4).flatMap1(hash); // should compile
$.. ==> Saveable[14]
jshell> Saveable.<Number>of(4).flatMap2(hash); // should compile
$.. ==> Saveable[14]
```

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

```
$ javac cs2030s/fp/Saveable.java
$ javac Test3.java
$ java Test3
$ java -jar checkstyle.jar -c cs2030_checks.xml cs2030s/fp/Saveable.java
$ javadoc -quiet -private -d docs cs2030s/fp/Saveable.java
```

Checkpoint

Make sure you try to make your methods as flexible as possible in their types.

QUESTION 2: Stream (15 marks)

Marking Criteria

- correctness (13 marks)
- style (2 marks)

Your Task

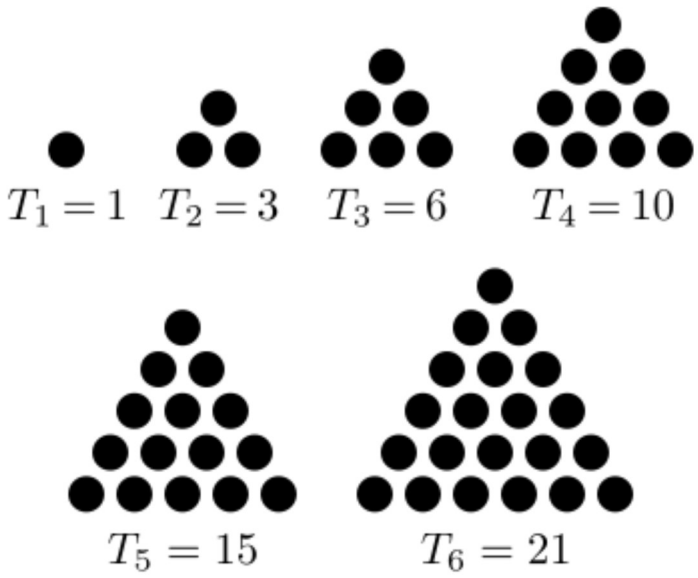
In this question, you are to write four `Stream` methods. Each method should only contain a single `return` statement involving `Stream` pipeline. Nothing more. No local variables or classes can be defined.

You may call the methods you create when solving other parts of this questions. However, you are not allowed to create additional methods to help solve the problem. Additionally, you are given the standard mutable `Pair` class that you can use.

Note that you do not have to worry about PECS for this question.

Even Triangular Number

A triangular number is a number of dots in the triangular arrangement of dots. The image below shows the first 6 triangular numbers:



We can find the n^{th} triangular number using the formula:

$$T_n = \frac{n(n+1)}{2}$$

Notice that not all triangular numbers are even. We want to find only the first n even triangular numbers. Recap that even numbers are numbers that are *fully divisible* by 2.

Write a static method using only streams:

- `public static List<Integer> evenTriangular(int n)` : returns a list of the first `n` triangular numbers

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

```
jshell> /open Pair.java
jshell> /Open Streaming.java

```

```
jshell> Streaming.evenTriangular(5).count()
$.. ==> 5
jshell> Streaming.evenTriangular(5).forEach(System.out::println)
6
10
28
36
66
jshell> Streaming.evenTriangular(5).toArray()
$.. ==> Object[5] { 6, 10, 28, 36, 66 }
jshell> Streaming.evenTriangular(10).forEach(System.out::println)
6
10
28
36
66
78
120
136
190
210
```

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

```
$ javac Streaming.java
$ javac Test4.java
$ java Test4
$ java -jar checkstyle.jar -c cs2030_checks.xml Streaming.java
```

Perfect Number

A number n is called a **perfect number** if:

> n is a positive integer (*i.e.*, $n > 0$) and it is equal to the sum of its unique positive divisors *excluding* itself

For instance, consider $n = 6$. Its unique positive divisors are 1, 2, and 3 (*exclude itself*). Since $1 + 2 + 3 = 6$, then 6 is a perfect number.

Consider $n = 4$. Its unique positive divisors are 1 and 2 (*exclude itself*). Since $1 + 2 = 3$, then 4 is not a perfect number.

Write two static methods using only streams:

- `public static boolean isPerfect(int n)`: returns `true` if `n` is a perfect number and `false` otherwise
- `public static List<Integer> allPerfect(int start, int end)`: returns a list of perfect numbers between `start` (*inclusive*) and `end` (*inclusive*)

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

```
jshell> /open Pair.java
jshell> /Open Streaming.java

jshell> Streaming.isPerfect(4)
$.. ==> false
jshell> Streaming.isPerfect(5)
$.. ==> false
jshell> Streaming.isPerfect(6)
$.. ==> true
jshell> Streaming.isPerfect(7)
$.. ==> false
jshell> Streaming.isPerfect(8)
$.. ==> false

jshell> Streaming.isPerfect(26)
$.. ==> false
jshell> Streaming.isPerfect(27)
$.. ==> false
jshell> Streaming.isPerfect(28)
$.. ==> true
jshell> Streaming.isPerfect(29)
$.. ==> false
jshell> Streaming.isPerfect(30)
$.. ==> false

jshell> Streaming.allPerfect(1, 10)
$.. ==> [6]
jshell> Streaming.allPerfect(1, 100)
$.. ==> [6, 28]
```

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

```
$ javac Streaming.java
$ javac Test5.java
$ java Test5
$ java -jar checkstyle.jar -c cs2030_checks.xml Streaming.java
```

Run-Length Decoding

In this question, you may assume that the given `stream` will be sequential. The next question requires you to modify the static methods such that it works for parallel streams. You are **NOT** allowed to use `sequential()` to force the `stream` to be sequential.

Consider a sequence of element *(can be of any type)*, for instance:

```
A A A B B A A A A C A A A A
```

We can encode this sequence into something more compact by recording the number of

consecutive elements together with the element. We say that two elements are equal if the `equals` method returns `true`.

The above sequence can first be split into several consecutive segments:

Consecutive Segment	Encoded Segment
`A A A`	`(3, A)`
`B B`	`(2, B)`
`A A A A`	`(4, A)`
`C`	`(1, C)`
`A A A A`	`(4, A)`

Each element is now a pair. So the run-length encoding is:

```
(3, A), (2, B), (4, A), (1, C), (4, A)
```

This is typically shorter than the original sequence. The decoding is simply the reverse process where we get back `A A A B B A A A A C A A A A` from `(3, A), (2, B), (4, A), (1, C), (4, A)`. What we want to do is the decoding process.

Write a static method using only streams:

- `public static List<T> decode(Stream<Pair<Integer, T>> stream)`: returns a list that is the result of decoding the given stream

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

```
jshell> /open Pair.java
jshell> /open Streaming.java

jshell> List<Integer> intList1 = List.of(
   ...>   1, 1, 1, 2, 2, 1, 1, 1, 1, 4, 3, 3
   ...> )
jshell> List<Integer> intList2 = List.of(
   ...>   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
   ...> )

jshell> List<String> strList1 = List.of(
   ...>   "A", "A", "A", "B", "B", "A",
   ...>   "A", "A", "A", "R", "Z", "Z"
   ...> )
```

```
jshell> List<String> strList2 = List.of(
   ...>   "A", "A", "A", "A", "A", "A",
   ...>   "A", "A", "A", "A", "A", "A"
   ...> )

jshell> List<Pair<Integer,Integer>> pairLst1 = List.of(
   ...>   new Pair<Integer, Integer>(3, 1),
   ...>   new Pair<Integer, Integer>(2, 2),
   ...>   new Pair<Integer, Integer>(4, 1),
   ...>   new Pair<Integer, Integer>(1, 4),
   ...>   new Pair<Integer, Integer>(2, 3)
   ...> )
jshell> List<Pair<Integer,Integer>> pairLst2 = List.of(
   ...>   new Pair<Integer, Integer>(12, 1)
   ...> )
jshell> List<Pair<Integer,String>> pairLst3 = List.of(
   ...>   new Pair<Integer, String>(3, "A"),
   ...>   new Pair<Integer, String>(2, "B"),
   ...>   new Pair<Integer, String>(4, "A"),
   ...>   new Pair<Integer, String>(1, "R"),
   ...>   new Pair<Integer, String>(2, "Z")
   ...> )
jshell> List<Pair<Integer,String>> pairLst4 = List.of(
   ...>   new Pair<Integer, String>(12, "A")
   ...> )

jshell> Streaming.decode(pairLst1.stream())
$.. ==> [1, 1, 1, 2, 2, 1, 1, 1, 1, 4, 3, 3]
jshell> Streaming.decode(pairLst2.stream())
$.. ==> [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
jshell> Streaming.decode(pairLst3.stream())
$.. ==> [A, A, A, B, B, A, A, A, A, R, Z, Z]
jshell> Streaming.decode(pairLst4.stream())
$.. ==> [A, A, A, A, A, A, A, A, A, A, A, A]

jshell> Streaming.decode(pairLst1.stream()).equals(intList1)
$.. ==> true
jshell> Streaming.decode(pairLst2.stream()).equals(intList2)
$.. ==> true
jshell> Streaming.decode(pairLst3.stream()).equals(strList1)
$.. ==> true
jshell> Streaming.decode(pairLst4.stream()).equals(strList2)
$.. ==> true

jshell> Streaming.decode(pairLst1.stream()).equals(intList2)
$.. ==> false
jshell> Streaming.decode(pairLst2.stream()).equals(intList1)
$.. ==> false
jshell> Streaming.decode(pairLst3.stream()).equals(strList2)
$.. ==> false
jshell> Streaming.decode(pairLst4.stream()).equals(strList1)
$.. ==> false
```

You can also test your code with `Test6.java`:

```
$ javac Streaming.java
```

```
$ javac Test6.java
$ java Test6
$ java -jar checkstyle.jar -c cs2030_checks.xml Streaming.java
```

Make it Parallelizable

You do not have to do anything if your code for `decode(Stream<Pair<Integer, T>> stream)` can already accept parallel stream without calling `sequential()`. You will not receive any mark for this question if you are using `sequential()` anywhere in your code.

Please be careful when attempting this question as incorrect solution will also invalidate your sequential attempt.

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

```
jshell> /open Pair.java
jshell> /open Streaming.java

jshell> List<Integer> intList1 = List.of(
   ...>   1, 1, 1, 2, 2, 1, 1, 1, 1, 4, 3, 3
   ...> )
jshell> List<Integer> intList2 = List.of(
   ...>   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
   ...> )

jshell> List<String> strList1 = List.of(
   ...>   "A", "A", "A", "B", "B", "A",
   ...>   "A", "A", "A", "R", "Z", "Z"
   ...> )
jshell> List<String> strList2 = List.of(
   ...>   "A", "A", "A", "A", "A", "A",
   ...>   "A", "A", "A", "A", "A", "A"
   ...> )

jshell> List<Pair<Integer,Integer>> pairLst1 = List.of(
   ...>   new Pair<Integer, Integer>(3, 1),
   ...>   new Pair<Integer, Integer>(2, 2),
   ...>   new Pair<Integer, Integer>(4, 1),
   ...>   new Pair<Integer, Integer>(1, 4),
   ...>   new Pair<Integer, Integer>(2, 3)
   ...> )
jshell> List<Pair<Integer,Integer>> pairLst2 = List.of(
   ...>   new Pair<Integer, Integer>(12, 1)
   ...> )
jshell> List<Pair<Integer,String>> pairLst3 = List.of(
   ...>   new Pair<Integer, String>(3, "A"),
   ...>   new Pair<Integer, String>(2, "B"),
   ...>   new Pair<Integer, String>(4, "A"),
   ...>   new Pair<Integer, String>(1, "R"),
   ...>   new Pair<Integer, String>(2, "Z")
   ...> )
jshell> List<Pair<Integer,String>> pairLst4 = List.of(
   ...>   new Pair<Integer, String>(12, "A")
```

```
   ...> )

jshell> Streaming.decode(pairLst1.stream().parallel())
$.. ==> [1, 1, 1, 2, 2, 1, 1, 1, 1, 4, 3, 3]
jshell> Streaming.decode(pairLst2.stream().parallel())
$.. ==> [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
jshell> Streaming.decode(pairLst3.stream().parallel())
$.. ==> [A, A, A, B, B, A, A, A, A, R, Z, Z]
jshell> Streaming.decode(pairLst4.stream().parallel())
$.. ==> [A, A, A, A, A, A, A, A, A, A, A, A]

jshell> Streaming.decode(pairLst1.stream().parallel()).equals(intList1)
$.. ==> true
jshell> Streaming.decode(pairLst2.stream().parallel()).equals(intList2)
$.. ==> true
jshell> Streaming.decode(pairLst3.stream().parallel()).equals(strList1)
$.. ==> true
jshell> Streaming.decode(pairLst4.stream().parallel()).equals(strList2)
$.. ==> true

jshell> Streaming.decode(pairLst1.stream().parallel()).equals(intList2)
$.. ==> false
jshell> Streaming.decode(pairLst2.stream().parallel()).equals(intList1)
$.. ==> false
jshell> Streaming.decode(pairLst3.stream().parallel()).equals(strList2)
$.. ==> false
jshell> Streaming.decode(pairLst4.stream().parallel()).equals(strList1)
$.. ==> false
```

You can also test your code with `Test7.java`:

```
$ javac Streaming.java
$ javac Test7.java
$ java Test7
$ java -jar checkstyle.jar -c cs2030_checks.xml Streaming.java
```