Lab 8: Infinite List

- Deadline: 8 November, 2022, Tuesday, 23:59, SST
- Mark: 6%

Prerequisite

- Caught up to Unit 34 of Lecture Notes
- Completed Lab 7

Important Concepts Tested

- ***Memo***: Compute only when needed & do not repeat yourself
- ***Actually***: Possibly error handled by container
- **PECS**: Make your method signature as flexible as possible
- ***JavaDoc***: Documenting your code and generating the documentation

Files

A skeleton for `InfiniteList<T>` is provided for you. Copy the following implementations over before you start with Lab 7:

- `cs2030s.fp.Action`
- `cs2030s.fp.Immutator`
- `cs2030s.fp.Constant`
- `cs2030s.fp.Combiner`
- `cs2030s.fp.Actionable`
- `cs2030s.fp.Immutatorable`
- `cs2030s.fp.Actually`
- `cs2030s.fp.Memo`

The files `Test1.java`, `Test2.java`, etc., as well as `CS2030STest.java`, are provided for testing. You can edit them to add your test cases, but they will not be submitted.

Infinite List

You have seen in class a poorly implemented version of `InfiniteList`. Recall that there are two

issues:

1. It uses `null` to represent a missing value.
    - This design prevents us from having `null` as elements in the list.
2. Produced values are not memoized.
    - This design results in repeated computation of the same value.

You are already given a (*badly written but still correct*) implementation of `Actually<T>` which will solve problem (1), and `Memo<T>` which will solve problem (2). We will use them to build a better version of `InfiniteList` here.

You are required to implement a single `InfiniteList` class as part of the `cs2030s.fp` package with *only two instance fields. No other instance fields are needed and allowed.* You may add one class field (*see below later about* `End`).

```
public class InfiniteList<T> {
  private Memo<Actually<T>> head;
  private Memo<InfiniteList<T>> tail;
}
```

IMPORTANT❗

Take note of the following constraints. Not following these constraints will result in immediate 0 for your lab.

- You are **NOT** allowed to add other instance fields.
- You are **NOT** allowed to use any raw types.
- You are **NOT** allowed to use `java.util.stream.Stream` to solve this lab.
- You are **NOT** allowed to use `unwrap` from `Actually`.

Additionally, you must follow the following constraints or heavy penalty will be given.

- `@SuppressWarnings` must be used responsibly.
- Where possible, use the methods provided by `Actually<T>` to handle the conditions where the value is there or not there, instead of using `if-else` or `try-catch`.

The Basics

Write the static `generate` and `iterate` methods that create an `InfiniteList`. This is different from `generate` in `MemoList` in Lab 7 but this is similar to `generate` introduced in the lecture on Infinite List.

To access the elements of the list, provide the `head` and `tail` method that produces the head and tail of the infinite list. Recap the problem in the lecture about `head()` and `tail()` in relation to `filter` method. You should follow the idea presented in the lecture to avoid modifying `head()` and

`tail()` later on.

To help with debugging, a `toString` method has been provided for you.

```
jshell> import cs2030s.fp.InfiniteList;
jshell> import cs2030s.fp.Immutator;
jshell> import cs2030s.fp.Constant;

jshell> InfiniteList<Integer> one = InfiniteList.generate(() -> 1)
one ==> [? ?]
jshell> one.head()
$.. ==> 1
jshell> one
one ==> [<1> ?]
jshell> one.tail().head()
$.. ==> 1
jshell> one
one ==> [<1> [<1> ?]]

jshell> InfiniteList<Integer> nul = InfiniteList.generate(() -> null)
nul ==> [? ?]
jshell> nul.head()
$.. ==> null
jshell> nul
nul ==> [<null> ?]
jshell> nul.tail().head()
$.. ==> null
jshell> nul
nul ==> [<null> [<null> ?]]

jshell> InfiniteList<String> str = InfiniteList.iterate("A", x -> x + "R")
str ==> [<A> ?]
jshell> str.tail().head()
$.. ==> "AR"
jshell> str
str ==> [<A> [<AR> ?]]
jshell> str.tail().tail().tail().head()
$.. ==> "ARRR"
jshell> str
str ==> [<A> [<AR> [<ARR> [<ARRR> ?]]]]

jshell> Immutator<Integer, Integer> incr = x -> {
   ...>   System.out.println("   " + x + " + 1 = " + (x + 1));
   ...>   return x + 1;
   ...> }
jshell> InfiniteList<Integer> nat = InfiniteList.iterate(1, incr)
nat ==> [<1> ?]

jshell> nat.head()
$.. ==> 1
jshell> nat
nat ==> [<1> ?]

jshell> nat.tail().head()
   1 + 1 = 2
$21 ==> 2
jshell> nat
nat ==> [<1> [<2> ?]]

jshell> nat.tail().head()
$.. ==> 2
jshell> nat
```

```
nat ==> [<1> [<2> ?]]

jshell> nat.tail().tail().head()
   2 + 1 = 3
$.. ==> 3
jshell> nat
nat ==> [<1> [<2> [<3> ?]]]

jshell> nat.tail().head()
$.. ==> 2
jshell> nat
nat ==> [<1> [<2> [<3> ?]]]

jshell> Constant<Integer> zero = () -> {
   ...>   System.out.println("   => 0");
   ...>   return 0;
   ...> }
jshell> InfiniteList<Integer> zeroes = InfiniteList.generate(zero)
zeroes ==> [? ?]

jshell> zeroes.head()
   => 0
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> ?]

jshell> zeroes.tail().head()
   => 0
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> [<0> ?]]

jshell> zeroes.head()
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> [<0> ?]]

jshell> zeroes.tail().head()
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> [<0> ?]]

jshell> zeroes.tail().tail().head()
   => 0
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> [<0> [<0> ?]]]

jshell> zeroes.tail().head()
$.. ==> 0
jshell> zeroes
zeroes ==> [<0> [<0> [<0> ?]]]
```

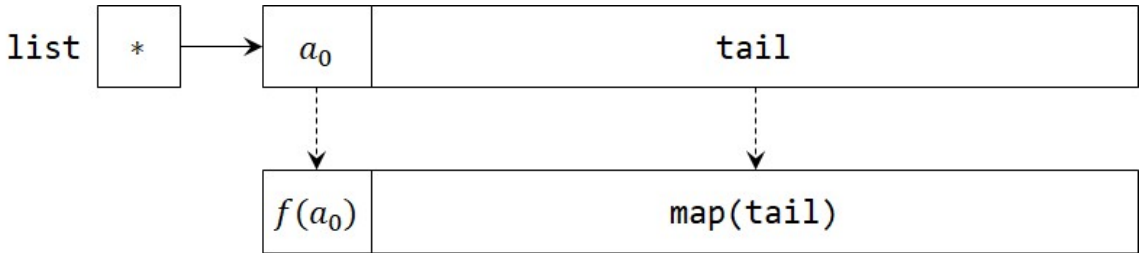
You can test your code by running the `Test1.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test1.java
$ java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml cs2030s/fp
/InfiniteList.java
```

```
$ javadoc -quiet -private -d docs cs2030s/fp/InfiniteList.java
```

`map`

Now let's add the `map` method. The `map` method (*lazily*) applies the given `Immutator` to each element in the list and returns the resulting `InfiniteList`.




```
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.Constant

jshell> InfiniteList<Integer> nat = InfiniteList.iterate(1, x -> x + 1)
nat ==> [<1> ?]
jshell> InfiniteList.generate(() -> 1).map(x -> x * 2)
$.. ==> [? ?]
jshell> nat.map(x -> x * 2)
$.. ==> [? ?]
jshell> InfiniteList.generate(() -> 1).map(x -> x * 2).tail().head()
$.. ==> 2
jshell> nat.map(x -> x * 2).head()
$.. ==> 2
jshell> nat.map(x -> x * 2).tail().head()
$.. ==> 4
jshell> nat.map(x -> x * 2).map(x -> x - 1).head()
$.. ==> 1
jshell> nat.map(x -> x * 2).map(x -> x - 1).tail().head()
$.. ==> 3
jshell> nat.map(x -> x % 2 == 0 ? null : x).tail().head()
$.. ==> null

jshell> Constant<Integer> one = () -> {
   ...>   System.out.println("   => 1");
   ...>   return 1;
   ...> }
jshell> Immutator<Integer, Integer> dbl = x -> {
   ...>   System.out.println("   " + x + " + " + x + " = " + (x + x));
   ...>   return x + x;
   ...> }

jshell> InfiniteList.generate(one).map(dbl).tail().head()
   => 1
   1 + 1 = 2
   => 1
   1 + 1 = 2
$.. ==> 2

jshell> InfiniteList<Integer> ones = InfiniteList.generate(one)
ones ==> [? ?]
jshell> InfiniteList<Integer> twos = ones.map(dbl)
twos ==> [? ?]

```

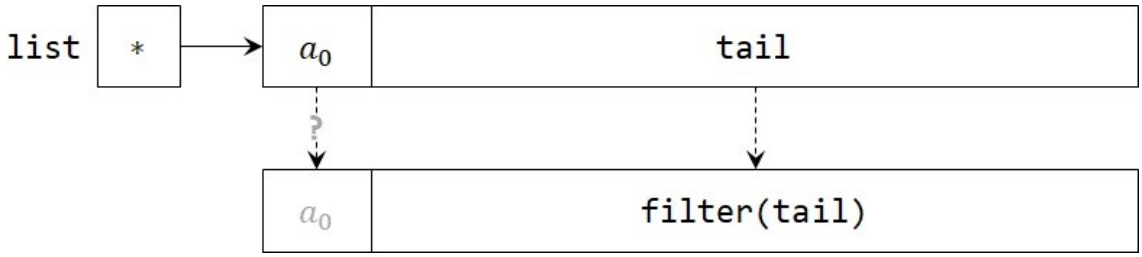
```
jshell> twos.tail().head()
   => 1
   1 + 1 = 2
   => 1
   1 + 1 = 2
$.. ==> 2
jshell> ones
ones ==> [<1> [<1> ?]]
jshell> twos
twos ==> [<2> [<2> ?]]

jshell> twos.head()
$.. ==> 2
jshell> twos.tail().head()
$.. ==> 2
```

You can test your code by running the `Test2.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test2.java
$ java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml cs2030s/fp
/InfiniteList.java
$ javadoc -quiet -private -d docs cs2030s/fp/InfiniteList.java
```

`filter`

Add the `filter` method to filter out elements in the list that fail a given `Immutator<Boolean, T>` (*note: make the type flexible following PECS*). `filter` should mark any filtered (*i.e., removed or missing*) element as `Actually.err()` instead of `null`. The resulting (lazily) filtered `InfiniteList` is returned.



To do this, you may want to understand the method `check` in both `Memo` and `Actually`. Or, if you are using your own implementation, you may want to add a similar method into your own implementation.

Lastly, for this lab, modify `Actually<T>` such that `Failure` class simply returns `<>` on `toString`. This has been done in our implementation of `Actually` (*there is no `Failure` class but we print `<>` on our equivalent concept of failure*) but if you are using your own implementation, you need to modify this.

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

```

```
jshell> InfiniteList<Integer> nat = InfiniteList.iterate(1, x -> x + 1)
nat ==> [<1> ?]
jshell> InfiniteList.generate(() -> 1).filter(x -> x % 2 == 0)
$.. ==> [? ?]
jshell> nat.filter(x -> x % 2 == 0)
$.. ==> [? ?]
jshell> nat.filter(x -> x % 2 == 0).head()
$.. ==> 2
jshell> nat.filter(x -> x % 2 == 0).filter(x -> x > 4).head()
$.. ==> 6

jshell> Immutator<Integer, Integer> incr = x -> {
   ...>   System.out.println("   " + x + " + 1 = " + (x + 1));
   ...>   return x + 1;
   ...> }
jshell> Immutator<Boolean, Integer> isEven = x -> {
   ...>   System.out.println("     " + x + " % 2 = " + (x % 2));
   ...>   return x % 2 == 0;
   ...> }

jshell> InfiniteList.iterate(1, incr).filter(isEven).tail().head()
     1 % 2 = 1
   1 + 1 = 2
     2 % 2 = 0
   2 + 1 = 3
     3 % 2 = 1
   3 + 1 = 4
     4 % 2 = 0
$.. ==> 4

jshell> InfiniteList<Integer> nums = InfiniteList.iterate(1, x -> x + 1)
nums ==> [<1> ?]
jshell> InfiniteList<Integer> evens = nums.filter(x -> x % 2 == 0)
evens ==> [? ?]

jshell> evens.tail().head()
$.. ==> 4
jshell> nums
nums ==> [<1> [<2> [<3> [<4> ?]]]]
jshell> evens // modify Failure::toString for this
evens ==> [<> [<2> [<> [<4> ?]]]]

jshell> nums.tail().head()
$.. ==> 2
jshell> evens.tail().head()
$.. ==> 4

jshell> Immutator<Boolean, Integer> moreThan5 = x -> {
   ...>   System.out.println("     " + x + " > 5 = " + (x > 5));
   ...>   return x > 5;
   ...> }
jshell> Immutator<Integer, Integer> dbl = x -> {
   ...>   System.out.println("   " + x + " + " + x + " = " + (x + x));
   ...>   return x + x;
   ...> }

jshell> InfiniteList.iterate(1, incr).filter(moreThan5).filter(isEven).head()
     1 > 5 = false
   1 + 1 = 2
     2 > 5 = false
   2 + 1 = 3
     3 > 5 = false
   3 + 1 = 4
     4 > 5 = false
```

```
    4 + 1 = 5
      5 > 5 = false
    5 + 1 = 6
      6 > 5 = true
      6 % 2 = 0
$.. ==> 6

jshell> InfiniteList.iterate(1, incr)
   ...>     .map(dbl).filter(moreThan5)
   ...>     .filter(isEven).tail().head()
    1 + 1 = 2
      2 > 5 = false
    1 + 1 = 2
    2 + 2 = 4
      4 > 5 = false
    2 + 1 = 3
    3 + 3 = 6
      6 > 5 = true
      6 % 2 = 0
    3 + 1 = 4
    4 + 4 = 8
      8 > 5 = true
      8 % 2 = 0
$.. ==> 8

jshell> InfiniteList.iterate(1, incr)
   ...>     .filter(isEven).map(dbl)
   ...>     .filter(moreThan5).head()
      1 % 2 = 1
    1 + 1 = 2
      2 % 2 = 0
    2 + 2 = 4
      4 > 5 = false
    2 + 1 = 3
      3 % 2 = 1
    3 + 1 = 4
      4 % 2 = 0
    4 + 4 = 8
      8 > 5 = true
$.. ==> 8
```

You can test your code by running the `Test3.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml cs2030s/fp
/InfiniteList.java
$ javadoc -quiet -private -d docs cs2030s/fp/InfiniteList.java
```

`end` and `isEnd`

We now consider the situation where the list can be finite. In this case, we need to mark the end of the list with a special tail (a.k.a. an *end*). Create a static nested class in `InfiniteList<T>` called `End` to represent a list that contains nothing and is used to mark the end of the list. You may add one final class field `END` (similar to `NONE` from Lab 4) in your `InfiniteList` to cache a single instance of the

end. Override the `toString` method so that an *end* returns `"-"`.

Provide a boolean `isEnd` method that returns true if the list is an instance of `End` and returns false otherwise. Note that `isEnd` is a lazy operation and should not trigger the evaluation of the infinite list.

Provide an `end` method that returns an end.

Provide a `head` and `tail` method that simply throws `java.util.NoSuchElementException()`.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Constant

jshell> InfiniteList.iterate(1, x -> x + 1).isEnd()
$.. ==> false
jshell> InfiniteList.generate(() -> 2).isEnd()
$.. ==> false
jshell> InfiniteList.generate(() -> 2).filter(x -> x % 3 == 0).isEnd()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).map(x -> 2).isEnd()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2 == 0).isEnd()
$.. ==> false

jshell> InfiniteList.end()
$.. ==> -
jshell> InfiniteList.end().isEnd()
$.. ==> true
jshell> InfiniteList.end().map(x -> 2).isEnd()
$.. ==> true
jshell> InfiniteList.end().filter(x -> true).isEnd()
$.. ==> true
jshell> InfiniteList.end().filter(x -> false).isEnd()
$.. ==> true
```

`limit`, `toList`

Now that we have a way to terminate an infinite list into a finite list, write a `limit` method that takes in a value `n` and truncate the `InfiniteList<T>` to a finite list with at most `n` elements. Your `limit` method must not count elements that are filtered out by `filter`, *if any*. Here, the type of `n` should be `long` instead of `int`.

Now, provide a terminal `toList` method that collects the elements in the `InfiniteList<T>` into a `java.util.List`. You may refer to `java.util.ArrayList` for methods that might be useful for implementing this method.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Constant

jshell> InfiniteList.end().limit(4).isEnd()
$.. ==> true
jshell> InfiniteList<Integer> nat = InfiniteList.iterate(1, x -> x + 1)
nat ==> [<1> ?]
```

```
jshell> nat.limit(0).isEnd()
$.. ==> true
jshell> nat.limit(1).isEnd()
$.. ==> false
jshell> nat.limit(10).isEnd()
$.. ==> false
jshell> nat.limit(-1).isEnd()
$.. ==> true
jshell> nat.limit(0).isEnd()
$.. ==> true
jshell> nat.limit(1).isEnd()
$.. ==> false
jshell> nat.limit(10).isEnd()
$.. ==> false

jshell> InfiniteList.generate(() -> 1).limit(4)
$.. ==> [? ?]
jshell> nat.limit(4)
$.. ==> [<1> ?]
jshell> nat.limit(1).head()
$.. ==> 1
jshell> nat.limit(4).head()
$.. ==> 1

jshell> <T> T run(Constant<T> c) {
   ...>   try {
   ...>     return c.init();
   ...>   } catch (Exception e) {
   ...>     System.out.println(e);
   ...>     return null;
   ...>   }
   ...> }
jshell> Immutator<Boolean, Integer> isEven = x -> x % 2 == 0
jshell> Immutator<String, String> zzz = s -> s + "Z"

jshell> run(() -> nat.limit(1).tail().head())
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> nat.limit(0).head())
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> nat.limit(4).tail().tail().head())
$.. ==> 3
jshell> run(() -> nat.limit(4).limit(1).tail().head())
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> nat.limit(1).limit(4).tail().head())
java.util.NoSuchElementException
$.. ==> null

jshell> run(() -> nat.filter(isEven).limit(0).head())
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> nat.filter(isEven).limit(1).head())
$.. ==> 2
jshell> run(() -> nat.limit(1).filter(isEven).head())
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> nat.limit(2).filter(isEven).head())
$.. ==> 2

jshell> run(() -> InfiniteList.iterate("A", zzz).limit(2).map(s -> s.length())
   ...>             .head())
$.. ==> 1
```

```
jshell> run(() -> InfiniteList.iterate("A", zzz).limit(2).map(s -> s.length())
   ...>             .tail().head())
$.. ==> 2
jshell> run(() -> InfiniteList.iterate("A", zzz).limit(2).map(s -> s.length())
   ...>             .tail().tail().head())
java.util.NoSuchElementException
$.. ==> null

jshell> run(() -> InfiniteList.iterate("A", zzz).map(s -> s.length())
   ...>             .limit(2).head())
$.. ==> 1
jshell> run(() -> InfiniteList.iterate("A", zzz).map(s -> s.length())
   ...>             .limit(2).tail().head())
$.. ==> 2
jshell> run(() -> InfiniteList.iterate("A", zzz).map(s -> s.length())
   ...>             .limit(2).tail().tail().head())
java.util.NoSuchElementException
$.. ==> null

jshell> InfiniteList.<String>end().toList()
$.. ==> []
jshell> InfiniteList.iterate("A", zzz).map(s -> s.length()).limit(2).toList()
$.. ==> [1, 2]
jshell> InfiniteList.iterate("A", zzz).limit(2).map(s -> s.length()).toList()
$.. ==> [1, 2]
jshell> nat.limit(2).filter(isEven).toList()
$.. ==> [2]
jshell> nat.filter(isEven).limit(2).toList()
$.. ==> [2, 4]
jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x > 10)
   ...>   .map(x -> x.hashCode() % 30).filter(x -> x < 20).limit(5).toList()
$.. ==> [11, 12, 13, 14, 15]
jshell> Random rng = new Random(1)rng ==> java.util.Random@2b9627bc
jshell> InfiniteList.generate(() -> rng.nextInt() % 100)
   ...>   .filter(x -> x > 10).limit(4).toList()
$.. ==> [76, 95, 26, 69]
jshell> InfiniteList.generate(() -> null).limit(4).limit(1).toList()
$.. ==> [null]
jshell> InfiniteList.generate(() -> null).limit(1).limit(4).toList()
$.. ==> [null]
```

You can test your code by running the `Test4.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml cs2030s/fp
/InfiniteList.java
$ javadoc -quiet -private -d docs cs2030s/fp/InfiniteList.java
```

`takeWhile`

Now, implement the `takeWhile` method. The method takes in an `Immutator<Boolean, T>` (*note: make the type flexible following PECS*), and truncates the list as soon as it finds an element that evaluates the condition to false.

Just like `limit`, the `takeWhile` method should ignore elements that have been filtered out by `filter`.

```
jshell> import cs2030s.fp.Immutator
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Constant

jshell> Immutator<Integer, Integer> incr = x -> {
   ...>   System.out.println("   " + x + " + 1 = " + (x + 1));
   ...>   return x + 1;
   ...> }
jshell> Immutator<Boolean, Integer> lessThan0 = x -> {
   ...>   System.out.println("     " + x + " < 0 = " + (x < 0));
   ...>   return x < 0;
   ...> }
jshell> Immutator<Boolean, Integer> lessThan2 = x -> {
   ...>   System.out.println("     " + x + " < 2 = " + (x < 2));
   ...>   return x < 2;
   ...> }
jshell> Immutator<Boolean, Integer> lessThan5 = x -> {
   ...>   System.out.println("     " + x + " < 5 = " + (x < 5));
   ...>   return x < 5;
   ...> }
jshell> Immutator<Boolean, Integer> lessThan10 = x -> {
   ...>   System.out.println("     " + x + " < 10 = " + (x < 10));
   ...>   return x < 10;
   ...> }
jshell> Immutator<Boolean, Integer> isEven = x -> {
   ...>   System.out.println("     " + x + " % 2 = " + (x % 2));
   ...>   return x % 2 == 0;
   ...> }
jshell> <T> T run(Constant<T> c) {
   ...>   try {
   ...>     return c.init();
   ...>   } catch (Exception e) {
   ...>     System.out.println(e);
   ...>     return null;
   ...>   }
   ...> }

jshell> jshell> InfiniteList.<Integer>end().takeWhile(lessThan0).isEnd()
$.. ==> true
jshell> InfiniteList.iterate(1, incr).takeWhile(lessThan0).isEnd()
$.. ==> false
jshell> InfiniteList.iterate(1, incr).takeWhile(lessThan2).isEnd()
$.. ==> false
jshell> InfiniteList.iterate(1, incr).takeWhile(lessThan5)
   ...>   .takeWhile(lessThan2).toList()
     1 < 5 = true
     1 < 2 = true
   1 + 1 = 2
     2 < 5 = true
     2 < 2 = false
$.. ==> [1]
jshell> InfiniteList.iterate(1, incr).filter(isEven)
   ...>   .takeWhile(lessThan10).toList()
     1 % 2 = 1
   1 + 1 = 2
     2 % 2 = 0
     2 < 10 = true
   2 + 1 = 3
     3 % 2 = 1
```

```
   3 + 1 = 4
     4 % 2 = 0
     4 < 10 = true
   4 + 1 = 5
     5 % 2 = 1
   5 + 1 = 6
     6 % 2 = 0
     6 < 10 = true
   6 + 1 = 7
     7 % 2 = 1
   7 + 1 = 8
     8 % 2 = 0
     8 < 10 = true
   8 + 1 = 9
     9 % 2 = 1
   9 + 1 = 10
     10 % 2 = 0
     10 < 10 = false
$.. ==> [2, 4, 6, 8]

jshell> run(() -> InfiniteList.generate(() -> 2).takeWhile(lessThan0));
$.. ==> [? ?]
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan0));
$.. ==> [? ?]
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan0).head());
     1 < 0 = false
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan2).head());
     1 < 2 = true
$.. ==> 1
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan2)
   ...>             .tail().head());
     1 < 2 = true
   1 + 1 = 2
     2 < 2 = false
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan2)
   ...>             .takeWhile(lessThan0).head());
     1 < 2 = true
     1 < 0 = false
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan0)
   ...>             .takeWhile(lessThan2).head());
     1 < 0 = false
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, incr).takeWhile(lessThan5)
   ...>             .takeWhile(lessThan2).tail().head());
     1 < 5 = true
     1 < 2 = true
   1 + 1 = 2
     2 < 5 = true
     2 < 2 = false
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, incr).filter(isEven)
   ...>             .takeWhile(lessThan10).head());
     1 % 2 = 1
   1 + 1 = 2
     2 % 2 = 0
     2 < 10 = true
```

```
$.. ==> 2
jshell> run(() -> InfiniteList.iterate(1, incr).filter(isEven)
   ...>             .takeWhile(lessThan10).tail().head());
     1 % 2 = 1
   1 + 1 = 2
     2 % 2 = 0
     2 < 10 = true
   2 + 1 = 3
     3 % 2 = 1
   3 + 1 = 4
     4 % 2 = 0
     4 < 10 = true
$.. ==> 4

jshell> InfiniteList<Integer> list = InfiniteList.iterate(1, incr)
   ...>                                 .takeWhile(lessThan10)
list ==> [? ?]
jshell> list.tail().tail().head()
     1 < 10 = true
   1 + 1 = 2
     2 < 10 = true
   2 + 1 = 3
     3 < 10 = true
$.. ==> 3
jshell> list.head()
$.. ==> 1
jshell> list
list ==> [<1> [<2> [<3> ?]]]
jshell> list.tail().head()
$.. ==> 2
jshell> list.tail().tail().tail().head()
   3 + 1 = 4
     4 < 10 = true
$.. ==> 4
jshell> list
list ==> [<1> [<2> [<3> [<4> ?]]]]
```

`reduce` and `count`

Finally, we are going to implement the terminal operations: `count` and `reduce`. To imitate `java.util.stream.Stream`, the `count` method should return a `long`.

Note: In Java, any integral value with suffix `L` is treated as a `long` value. For instance, `123` has the type `int`, but `123L` has the type `long`.

```
jshell> import cs2030s.fp.InfiniteList;

jshell> InfiniteList.<Integer>end().reduce(0, (x, y) -> x + y)
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(5).reduce(0, (x, y) -> x + y)
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).reduce(0, (x, y) -> x + y)
$.. ==> 0
jshell> InfiniteList.iterate(1, x -> x + 1).map(x -> x * x)
   ...>   .limit(5).reduce(1, (x, y) -> x * y)
$.. ==> 14400

jshell> InfiniteList.<Integer>end().count()
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).count()
```

```
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(1).count()
$.. ==> 1

jshell> InfiniteList.iterate(0, x -> x + 1).filter(x -> x % 2 == 1)
   ...>   .limit(10).count()
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).limit(10)
   ...>   .filter(x -> x % 2 == 1).count()
$.. ==> 5
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10)
   ...>   .count()
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).takeWhile(x -> x < 10)
   ...>   .filter(x -> x % 2 == 0).count()
$.. ==> 5
```

You can test your code by running the `Test6.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style and can generate the documentation without error.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test6.java
$ java Test6
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml cs2030s/fp
/InfiniteList.java
$ javadoc -quiet -private -d docs InfiniteList.java
```

Following CS2030S Style Guide

You should make sure that your code follows the given Java style guide.

Grading

This lab is worth 24 marks and contributes 6% to your final grade. The marking scheme is as follows:

- Documentation: 2 marks
- Everything Else: 22 marks

We will deduct 1 mark for each unnecessary use of `@SuppressWarnings` and each raw type. `@SuppressWarnings` should be used appropriately and not abused to remove compilation warnings.

Note that general style marks are no longer awarded will only be awarded for documentation. You should know how to follow the prescribed Java style by now. We will still deduct up to 2 marks if there are serious violations of styles. In other words, if you have no documentation and serious violation of styles, you will get deducted 4 marks.

Submission

Similar to Lab 7, submit the files inside the directory cs2030s/fp along with the other file without the need for folder. Your cs2030s/fp should only contain the following files:

- `Action.java`
- `Actionable.java`
- `Actually.java`
- `Combiner.java`
- `Constant.java`
- `Immutator.java`
- `Immutatorable.java`
- `Memo.java`
- `InfiniteList.java`

Additionally, you ***must*** submit the file `Lab8.h` and `Lab8.java`. Otherwise, you CodeCrunch submission will not run.