Ex 7: To Infinity and Beyond

Basic Information

- **Deadline:** 29 October 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★★★★

Prerequisite

- Caught up to Unit 33 of Lecture Notes.
- Familiar with CS2030S Java style guide.

Class Files

If you have not finished Programming Exercise 5 and 6, do not worry. We provide `.class` files for the functional interfaces as well as `Maybe<T>` and `Lazy<T>`. Note that the implementation for `Maybe<T>` is badly written not following OOP but it is the correct implementation.

Additionally, the class files were compiled on PE node using Java 21 compiled. If you are not using Java 21 or if you are not working on PE node, you may get different result. It is unlikely, but the possibility is there. Please only do your work on the PE node.

You are required to use the given `.class` files as we will be using our version during testing. In other words, you are not allowed to add methods not specified by Ex 5 and Ex 6.

Maybe Class

The class `Maybe<T>` has the following `public` methods. You cannot use any methods that are not `public` from outside the package.

Method	Description
`static <T> Maybe<T> of(T val)`	Creates a `Maybe<T>` with the given content `val` if `val` is not `null`. Otherwise, returns the shared instance of `None<?>`.
`static <T> Maybe<T> some(T val)`	Creates a `Maybe<T>` with the given content `val` which may be `null`.
`static <T> Maybe<T> none()`	Creates a `Maybe<T>` without any content, this is guaranteed to return the shared instance of `None<?>`.
`String toString()`	Returns the string representation of `Maybe<T>`.
`boolean equals(Object obj)`	• `Maybe<T>`: Returns `true` if the content is equal to the content of `obj`. Otherwise returns `false`. • `None<T>`: Returns `true` if `obj` is also `None<T>`. Otherwise returns `false`.
`<U> Maybe<U> map (Transformer<? super T, ? extends U> fn)`	• `Maybe<T>`: Create a new instance of `Maybe<T>` by applying the transformer `fn` to the content and wrapping it in `Maybe<T>`. It will never return `None<T>` to allow for our `InfiniteList<T>` to contain `null`. • `None<T>`: Returns `None<T>`.
`Maybe<T> filter (BooleanCondition<? super T> pred)`	• `Maybe<T>`: If the content is not `null` and `pred.test(content)` returns `true`, we return the current instance. Otherwise, returns `None<T>`. • `None<T>`: Returns `None<T>`.
`<U> Maybe<U> flatmap (Transformer<? super T, ? extends Maybe<? extends U>> fn)`	• `Maybe<T>`: Create a new instance of `Maybe<T>` by applying the transformer `fn` to the content *without* wrapping it in `Maybe<T>` as `fn` already returns `Maybe<U>`. • `None<T>`: Returns `None<T>`.
`T orElse(Producer<? extends T> prod)`	• `Maybe<T>`: Returns the content (*even if it is* `null`). • `None<T>`: Returns the value produced by the producer `prod`.

Method	Description
`void ifPresent(Consumer<? super T> cons)`	• `Maybe<T>`: Pass the content to the consumer `cons`. • `None<T>`: Do nothing.

Lazy Class

The class `Lazy<T>` has the following `public` methods. You cannot use any methods that are not `public` from outside the package.

Method	Description
`static <T> Lazy<T> of(T val)`	Creates a `Lazy<T>` with the given content `val` already evaluated.
`static <T> Lazy<T> of(Producer<? extends T> prod)`	Creates a `Lazy<T>` with the content not yet evaluated and will be evaluated using the given producer.
`boolean equals(Object obj)`	Returns `true` if the content is equal to the content of `obj`. Otherwise returns `false`. This **forces** evaluation of the content.
`<U> Lazy<U> map (Transformer<? super T, ? extends U> fn)`	Lazily maps the content using the given transformer.
`Lazy<Boolean> filter (BooleanCondition<? super T> pred)`	Lazily test if the value passes the test or not and returns a `Lazy<Boolean>` to indicate the result.
`<U> Lazy<U> flatMap (Transformer<? super T, ? extends Lazy<? extends U>> fn)`	Lazily creates a new instance of `Lazy<T>` by applying the transformer `fn` to the content without wrapping it in another `Lazy<..>`.

Method	Description
`<U, V> Lazy<V> combine (Lazy<? extends U> lazy, Combiner<? super T, ? super U, ? extends V> fn)`	Combine `this` with `lazy` using `Combiner` by invoking `fn.combine(this.get(), lazy.get())`. Then we wrap the result back in `Lazy`.
`T get()`	Evaluates (*if not yet evaluated, otherwise do not evaluate again*) and returns the content.

Infinity

This is a follow-up from Ex 6. In Ex 6, we have constructed a generic class `Lazy<T>` using `Maybe<T>`. Now we are going to combine them into a `Lazy<Maybe<T>>` to build an infinite list. We need the `Lazy<..>` because we want our infinite list to be lazily evaluated. We need the `Maybe<T>` because the value may be present or may be missing due to `filter`. Recap that in the lecture notes we use the `null` value to indicate missing values because:

1. We need a value that all possible generic type `T` has.
2. We need a value that indicates a value should not be included.

The only solution was to use `null` because there is no other value that satisfies these two conditions. Of course, it will be better if we have a second `null` value (*maybe we call it `None` like in Python, heeeyyy wait a minute, that's our `None<T>`*) to indicate this, but unfortunately we do not have such value. That is why we need to use `Maybe<T>`.

Please make sure you are familiar with `Maybe<T>` and `Lazy<T>` before proceeding. You do not have to know the implementation but you should understand the expected behavior.

Constraints

We will recap some of the constraints for our labs so far.

- You are **NOT** allowed to use raw types.
- `@SuppressWarnings` must be used responsibly, in the smallest scope, and commented.

Additionally, you have the following constraints for this lab.

- You are **NOT** allowed to use `java.util.stream.Stream`.
- You are **NOT** allowed to add new classes (*nested or otherwise*).

- You are **NOT** allowed to add new methods in the `InfiniteList` (*not even private methods*).
- You are **NOT** allowed to use conditional statement (*e.g.*, `if`-`else`) or conditional expression (*e.g.*, `?:` *operator*).
  - Unless otherwise stated.
- You are **NOT** allowed to use loops (*e.g.*, `while`-loop or `for`-loop).
  - You may, however, use recursion (*but possibly without conditionals*).
- There must only be a single instance of `Sentinel`.

Relaxation

As a relaxtaion, the type signature in the templates are already the most flexible types. You have suffered enough thinking about more flexible type in the past two labs. The focus here is about laziness.

Basic

You are already given most of the implementations including `head()`, `tail()`, `map(..)`, and `filter(..)`. Please study them carefully. Additionally, to help with debugging, a `toString()` has already been provided for you. Lastly, there are three factory methods for `InfiniteList`, namely `generate(..)`, `iterate(..)`, and `sentinel()`. The last one creates an empty list.

As we also want to limit our infinite list to a potentially finite list, we have provided the `Sentinel` class. This class is rather straightforward as it overrides all of the methods in `InfiniteList<T>`. However, in most cases, it simply returns itself (*through the use of* `InfiniteList.<Object>sentinel()`) or throw an exception.

You can test the initial implementation by running `Test0A.java` to `Test0C.java`.

```
$ javac -Xlint:rawtypes cs2030s/fp/*java
$ javac -Xlint:rawtypes Test0A.java
$ javac -Xlint:rawtypes Test0B.java
$ javac -Xlint:rawtypes Test0C.java
$ java Test0A
$ java Test0B
$ java Test0C
```

Tasks

Task 1: Limit

The `Sentinel` class is not only an indication of an empty list but because our idea of an `InfiniteList` is the value (*or value wrapped in* `Lazy` *and* `Maybe`, *i.e., the head*) and the rest of the list (*which is another* `InfiniteList`), an empty list is also an indicator for the end of the list. Given a `Sentinel`, we can now write two methods:

1. Implement the method `InfiniteList<T> limit(long n)` in `InfiniteList`.
   - The method takes in a number `n`.
   - The method returns a new `InfiniteList` that is lazily computed which is the *truncation* of the `InfiniteList` to a finite list with *at most* `n` elements.
   - The method should not count the elements that are filtered out by `filter` (*if any*).
   - The method is **allowed** to use conditional statement/expression.
2. Override the method `InfiniteList<T> limit(long n)` in `Sentinel`.
   - The method takes in a number `n`.
   - Determine the appropriate behaviour for this.

Sample Usage

```
jshell> import cs2030s.fp.BooleanCondition
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Transformer
jshell> import cs2030s.fp.Producer

jshell> InfiniteList.sentinel().limit(4).isSentinel()
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1).limit(0).isSentinel()
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1).limit(1).isSentinel()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).limit(10).isSentinel()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).limit(-1).isSentinel()
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1).limit(0).isSentinel()
$.. ==> true
jshell> InfiniteList.iterate(1, x -> x + 1).limit(1).isSentinel()
$.. ==> false
jshell> InfiniteList.iterate(1, x -> x + 1).limit(10).isSentinel()
$.. ==> false

jshell> InfiniteList.generate(() -> 1).limit(4)
$.. ==> [? ?]
jshell> InfiniteList.iterate(1, x -> x + 1).limit(4)
$.. ==> [[1] ?]
```

```
jshell> InfiniteList.iterate(1, x -> x + 1).limit(1).head()
$.. ==> 1
jshell> InfiniteList.iterate(1, x -> x + 1).limit(4).head()
$.. ==> 1

jshell> <T> T run(Producer<T> p) {
   ...>   try {
   ...>     return p.produce();
   ...>   } catch (Exception e) {
   ...>     System.out.println(e);
   ...>     return null;
   ...>   }
   ...> }

jshell> run(() -> InfiniteList.iterate(1, x -> x +
1).limit(1).tail().head());
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, x -> x + 1).limit(0).head());
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, x -> x +
1).limit(4).tail().tail().head());
$.. ==> 3
jshell> run(() -> InfiniteList.iterate(1, x -> x +
1).limit(4).limit(1).tail().head());
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, x -> x +
1).limit(1).limit(4).tail().head());
java.util.NoSuchElementException
$.. ==> null

jshell> run(() -> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2
== 0).limit(0).head());
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, x -> x + 1).filter(x -> x % 2
== 0).limit(1).head());
$.. ==> 2
jshell> run(() -> InfiniteList.iterate(1, x -> x + 1).limit(1).filter(x -
> x % 2 == 0).head());
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1, x -> x + 1).limit(2).filter(x -
> x % 2 == 0).head());
$.. ==> 2

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

```
$.. ==> null

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

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test1.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test1.java
java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex7_style.xml cs2030s/fp/*.java
```

Task 2: To List

1. Implement the method `List<T> toList()` in `InfiniteList`.
   - The method takes in no parameter.
   - The method returns a new `List<T>` (*should really just be* `ArrayList<T>`) which is a collection elements of the `InfiniteList` in the same order as they appear in the `InfiniteList`.
2. Override the method `InfiniteList<T> toList()` in `Sentinel`.
   - The method takes in no parameter.
   - Determine the appropriate behaviour for this.

Sample Usage

```
jshell> import cs2030s.fp.BooleanCondition
jshell> import cs2030s.fp.InfiniteList
jshell> import cs2030s.fp.Transformer
jshell> import cs2030s.fp.Producer

jshell> <T> T run(Producer<T> p) {
   ...>   try {
   ...>     return p.produce();
   ...>   } catch (Exception e) {
   ...>     System.out.println(e);
   ...>     return null;
   ...>   }
```

```
    ...> }

jshell> Transformer<Integer, Integer> incr = x -> x + 1;
jshell> BooleanCondition<Integer> isEven = x -> (x % 2 == 0);

jshell> InfiniteList.<String>sentinel().toList()
$.. ==> []
jshell> InfiniteList.iterate("A", s -> s + "Z").map(s ->
s.length()).limit(2).toList()
$.. ==> [1, 2]
jshell> InfiniteList.iterate("A", s -> s + "Z").limit(2).map(s ->
s.length()).toList()
$.. ==> [1, 2]
jshell> InfiniteList.iterate(1, incr).limit(2).filter(isEven).toList()
$.. ==> [2]
jshell> InfiniteList.iterate(1, incr).filter(isEven).limit(2).toList()
$.. ==> [2, 4]
jshell> InfiniteList.iterate(1, x -> x + 1).limit(10).limit(3).toList()
$.. ==> [1, 2, 3]
jshell> InfiniteList.iterate(1, x -> x + 1).limit(3).limit(10).toList()
$.. ==> [1, 2, 3]
jshell> InfiniteList.generate(() -> 4).limit(0).toList()
$.. ==> []
jshell> InfiniteList.generate(() -> 4).limit(2).toList()
$.. ==> [4, 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> java.util.Random rng = new java.util.Random(1);

jshell> InfiniteList.generate(() -> rng.nextInt() % 100).filter(x -> x >
10).limit(4).toList()
$.. ==> [76, 95, 26, 69]
jshell> InfiniteList.<Object>generate(() ->
null).limit(4).limit(1).toList()
$.. ==> [null]
jshell> InfiniteList.<Object>generate(() ->
null).limit(1).limit(4).toList()
$.. ==> [null]
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test2.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test2.java
java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex7_style.xml cs2030s/fp/*.java
```

Task 3: It Takes a While

Now we want to implement the `takeWhile` method.

1. Implement the method `InfiniteList<T> takeWhile(BooleanCondition<? super T> pred)` in `InfiniteList`.
   - The method takes in a `BooleanCondition`.
   - The method returns an `InfiniteList` that is a truncated version of the initial `InfiniteList`.
     - The method truncates the infinite list as soon as it finds an element that evaluates the condition to `false`.
   - The method is **allowed** to use conditional statement/expression.
2. Override the method `InfiniteList<T> takeWhile(BooleanCondition<? super T> pred)` in `Sentinel`.
   - The method takes in a `BooleanCondition`.
   - Determine the appropriate behaviour for this.

Sample Usage

```
jshell> import cs2030s.fp.InfiniteList;
jshell> import cs2030s.fp.Transformer;
jshell> import cs2030s.fp.Producer;
jshell> import cs2030s.fp.BooleanCondition;

jshell> Transformer<Integer, Integer> incr = x -> {
   ...>   System.out.println("    iterate: " + x);
   ...>   return x + 1;
   ...> };
jshell> BooleanCondition<Integer> lessThan0 = x -> {
   ...>   System.out.println("    takeWhile x < 0: " + x);
   ...>   return x < 0;
   ...> };
jshell> BooleanCondition<Integer> lessThan2 = x -> {
   ...>   System.out.println("    takeWhile x < 2: " + x);
   ...>   return x < 2;
   ...> };
jshell> BooleanCondition<Integer> lessThan5 = x -> {
   ...>   System.out.println("    takeWhile x < 5: " + x);
   ...>   return x < 5;
   ...> };
jshell> BooleanCondition<Integer> lessThan10 = x -> {
   ...>   System.out.println("    takeWhile x < 10: " + x);
   ...>   return x < 10;
   ...> };
jshell> BooleanCondition<Integer> isEven = x -> {
   ...>   System.out.println("    filter x % 2 == 0: " + x);
   ...>   return x % 2 == 0;
   ...> };

jshell> <T> T run(Producer<T> p) {
   ...>   try {
   ...>     return p.produce();
```

```
    ...>   } catch (Exception e) {
    ...>     System.out.println(e);
    ...>     return null;
    ...>   }
    ...> }

jshell> InfiniteList.
<Integer>sentinel().takeWhile(lessThan0).isSentinel()
$.. ==> true
jshell> InfiniteList.iterate(1, incr).takeWhile(lessThan0).isSentinel()
$.. ==> false
jshell> InfiniteList.iterate(1, incr).takeWhile(lessThan2).isSentinel()
$.. ==> false
jshell> InfiniteList.iterate(1,
incr).takeWhile(lessThan5).takeWhile(lessThan2).toList()
    takeWhile x < 5: 1
    takeWhile x < 2: 1
    iterate: 1
    takeWhile x < 5: 2
    takeWhile x < 2: 2
$.. ==> [1]
jshell> InfiniteList.iterate(1,
incr).filter(isEven).takeWhile(lessThan10).toList()
    filter x % 2 == 0: 1
    iterate: 1
    filter x % 2 == 0: 2
    takeWhile x < 10: 2
    iterate: 2
    filter x % 2 == 0: 3
    iterate: 3
    filter x % 2 == 0: 4
    takeWhile x < 10: 4
    iterate: 4
    filter x % 2 == 0: 5
    iterate: 5
    filter x % 2 == 0: 6
    takeWhile x < 10: 6
    iterate: 6
    filter x % 2 == 0: 7
    iterate: 7
    filter x % 2 == 0: 8
    takeWhile x < 10: 8
    iterate: 8
    filter x % 2 == 0: 9
    iterate: 9
    filter x % 2 == 0: 10
    takeWhile x < 10: 10
$.. ==> [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());
    takeWhile x < 0: 1
java.util.NoSuchElementException
```

```
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1,
incr).takeWhile(lessThan2).head());
    takeWhile x < 2: 1
$.. ==> 1
jshell> run(() -> InfiniteList.iterate(1,
incr).takeWhile(lessThan2).tail().head());
    takeWhile x < 2: 1
    iterate: 1
    takeWhile x < 2: 2
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1,
incr).takeWhile(lessThan2).takeWhile(lessThan0).head());
    takeWhile x < 2: 1
    takeWhile x < 0: 1
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1,
incr).takeWhile(lessThan0).takeWhile(lessThan2).head());
    takeWhile x < 0: 1
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1,
incr).takeWhile(lessThan5).takeWhile(lessThan2).tail().head());
    takeWhile x < 5: 1
    takeWhile x < 2: 1
    iterate: 1
    takeWhile x < 5: 2
    takeWhile x < 2: 2
java.util.NoSuchElementException
$.. ==> null
jshell> run(() -> InfiniteList.iterate(1,
incr).filter(isEven).takeWhile(lessThan10).head());
    filter x % 2 == 0: 1
    iterate: 1
    filter x % 2 == 0: 2
    takeWhile x < 10: 2
$.. ==> 2
jshell> run(() -> InfiniteList.iterate(1,
incr).filter(isEven).takeWhile(lessThan10).tail().head());
    filter x % 2 == 0: 1
    iterate: 1
    filter x % 2 == 0: 2
    takeWhile x < 10: 2
    iterate: 2
    filter x % 2 == 0: 3
    iterate: 3
    filter x % 2 == 0: 4
    takeWhile x < 10: 4
$.. ==> 4

jshell> InfiniteList<Integer> list = InfiniteList.iterate(1,
incr).takeWhile(lessThan10)

jshell> list.tail().tail().head()
    takeWhile x < 10: 1
```

```
    iterate: 1
    takeWhile x < 10: 2
    iterate: 2
    takeWhile x < 10: 3
$.. ==> 3
jshell> list.head()
$.. ==> 1
jshell> list
list ==> [[1] [[2] [[3] ?]]]

jshell> list.tail().head()
$.. ==> 2
jshell> list.tail().tail().tail().head()
    iterate: 3
    takeWhile x < 10: 4
$.. ==> 4
jshell> list
list ==> [[1] [[2] [[3] [[4] ?]]]]
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test3.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test3.java
java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex7_style.xml cs2030s/fp/*.java
```

Task 4: Folding it Right

In the lecture, we discuss about the behaviour of `reduce` method of `Stream`. `reduce` is actually equivalent to fold left. In a fold left, given a list `[v1, v2, v3, v4]`, an initial value `e` and the operation `+`, the operation performed is as follows:

`(((e + v1) + v2) + v3) + v4`

What we want is the opposite. We want to do a fold but from the right. In a fold left, given a list `[v1, v2, v3, v4]`, an initial value `e` and the operation `+`, the operation performed is as follows:

`(v1 + (v2 + (v3 + (v4 + e))))`

Since the order of operations are different, the result may potentially be different. For `+`, these two are going to produce the same result. But for `-`, the result will be different. Note that this is a *terminal operation* in Java stream.

1. Implement the method `<U> U foldRight(U id, Combiner<? super T, U, U> acc)` in `InfiniteList`.

- The method takes in the initial value `id` and an accumulator `acc`.
  - Note that `Combiner<T, S, R>` is a new interface with the implementation given in `cs2030s/fp/Combiner.java`.
  - It has a single abstract method `R combine(T arg1, S arg2)`.
- The method returns the result of fold right of type `U` performed on the `InfiniteList` with the given accumulator `acc` starting from the initial value `id`.
  - Note that the name `id` is used as typically the initial value is the *identity* operation of the accumulator.

2. Override the method `<U> U foldRight(U id, Combiner<? super T, U, U> acc)` in `Sentinel`.
   - The method takes in the initial value `id` and an accumulator `acc`.
   - Determine the appropriate behaviour for this.

Sample Usage

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

jshell> InfiniteList.<Integer>sentinel().foldRight(0, (x, y) -> x + y)
$.. ==> 0
jshell> InfiniteList.iterate(0, x -> x + 1).limit(5).foldRight(0, (x, y)
-> x + y)
$.. ==> 10
jshell> InfiniteList.iterate(0, x -> x + 1).limit(0).foldRight(0, (x, y)
-> x + y)
$.. ==> 0
jshell> InfiniteList.iterate(1, x -> x + 1).map(x -> x *
x).limit(5).foldRight(1, (x, y) -> x * y)
$.. ==> 14400
jshell> InfiniteList.iterate(1, x -> x + 1).map(x -> x *
x).limit(5).foldRight(1, (x, y) -> x - y)
$.. ==> 14
jshell> // the above is equivalent to (1 - (4 - (9 - (16 - (25 - 1)))))
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test4.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test4.java
java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex7_style.xml cs2030s/fp/*.java
```

Task 5: Documenting Your Code

Now that we are beginning to build our own package that others can use, we should start to produce documentation on our code.

From Ex 7 onwards, you are required to document your classes and methods with Javadoc comments. You have seen examples from the skeleton code earlier exercises. For more details, see the JavaDoc guide. The checkstyle tool now checks for JavaDoc-related style as well.

For Ex 7, you should write javadoc documentation for all methods in `InfiniteList.java`. Documenting the code your wrote previously for Ex 5 and Ex 6 are encouraged but optional. Your task is to document the remaining methods. We have provided some documentations on some of the codes as example. You should also double-check that the provided documentations satisfies the style guide.

JavaDoc

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

Following CS2030S Style Guide

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

Further Deductions

Additional deductions may be given for other issues or errors in your code. These deductions may now be unbounded, up to 5 marks. This include *but not limited to*

- run-time error.
- failure to follow instructions.
- improper designs (*e.g.*, not following good OOP practice).
- not comenting `@SuppressWarnings`.
- misuse of `@SuppressWarnings` (*e.g.*, not necessary, not in smallest scope, *etc*).