Lab 7

Submission deadline: 2359, Sunday, October 22, 2017.

Prerequisites

This lab assumes that students:

- are familiar with the concept of possibly infinite, lazily evaluated, list
- familiar with the various `Stream` operations
- familiar with creating, passing, storing, and invoking various lambda functions

Learning Objectives

After completing this lab, students should:

- be comfortable with implementing a lazily evaluated list using lambdas.
- be familiar with the concept of memoization in implementing a lazily evaluated list.

Setup

The skeleton code from Lab 7 is available on `cs2030-i` under the directory `~cs2030/lab07`. There are three files, `InfiniteList.java` which you are asked to complete, `LabSeven.java`, which contains simple test code to test the behavior of InfiniteList, and `Pair.java`, a helper class to maintain a pair of items.

Task

For Lab 7, you are asked to implement a bunch of methods for the class `InfiniteList`, a simple list that supports various lambda operations. See Grading section below for a list.

You are still required to

- follows the CS2030 Coding Style [../style/index.html]
- clearly documented with `javadoc` [../javadoc/index.html] (this has been done for you, for free!)

InfiniteList

A `InfiniteList<T>` is a generic list that can store elements of type `T` in order. Duplicates are allowed.

Unlike `LambdaList`, we can only create either an empty list (with `InfiniteList.empty()`) or an infinitely long list (with `generate` or `iterate`). Unlike `LambdaList`, an `InfiniteList` is lazily evaluated -- the actual list element is generated on demand.

An `InfiniteList` is similiar to `Stream` in Java, but a `Stream` can only be traversed once, while an `InfiniteList` can be repeatedly traversed! `InfiniteList` also supports some operations, like `zipWith` and `unzipTo` that is not supported by a `Stream`. While this lab did not require you to implement all methods available to `Stream`, you can easily add them after you complete this lab and unlike `LambdaList`, which you should never use again, the `InfiniteList` class is actually something that you can keep around and use it in your later study / career if needed!

Like in the previous lab, you are *NOT ALLOWED* to solve this lab using the `Stream` interfaces.

You should be familiar with all the operations that you are asked to implement, except two new ones, `zipWith` and `unzipTo`, which are explained below:

A zip operation takes in two lists, and combined them, element-by-element, into a single list. The combination is done through a `zipper` method. The `zipWith` operation on an `InfiniteList` does this. For instance:

```
1  InfiniteList<Integer> list1 = InfiniteList.iterate(0, x -> x + 2)
2  InfiniteList<Integer> list2 = InfiniteList.iterate(1, x -> x + 1)
3  list1.zipWith(list2, (x, y) -> x - y); // -1, 0, 1, 2, 3, ...
```

The unzip operation does the reverse -- given a list, it splits it into two. The `unzipTo` operation on an `InfiniteList` returns a list of pairs, each pair containing two elements `unzipped` from the original list.

```
1  InfiniteList<Integer> list1 = InfiniteList.iterate(0, x -> x + 1)
2  list1.unzipTo(x -> x % 3, x -> x / 3); // split into (0, 0) (1, 0
```

Implementing InfiniteList

You have seen a simple version of `InfiniteList` in class on Monday. We are going to implement a better and more complicated version of that (so that code from Lecture 8 is not directly usable).

The main differences are:

- Lab 7's version of `InfiniteList` is not always infinite. It can be truncated just like a `Stream` with `limit` and `takeWhile` operation. So methods such as `findFirst` need to consider the possibility of a finite list, including an empty list.
- We need to be as lazy as possible and only generate the element (i.e., invoke the `Supplier` 's `get()` method) when necessary. Once we generate an element, we shouldn't generate it again. So, we cache a copy of the value if it has been generated before. This logic has been written in the `head()` and `tail()` method for you.

Debugging Lazy Operations

In addition, to help with debugging the laziness, we have included a counter variable `numOfEvals` that count how many times the head's supplier has been invoked. We can obtain the value of counter with `numOfEvals()` method and reset the counter with `resetStats()`.

Recusions

You might be tempted to implement the terminal operations (those that does not return an `InfiniteList`) recursively. Doing so will cause stack overflows if the list is too long, due to the lack of tail recursion optimization in Java (more on this in the future). To prevent that, you should implement the terminal operations with loops, instead of recursion.

Empty List

We created a special private inner class `Empty` to represent an empty `InfiniteList`. Note that we intentionally violates the Liskov Substitution Principle here -- since the empty list *should* have a different behavior than a non-empty list, and treating an empty `InfiniteList` just like a non-empty one in your code *should* give you an error!

Additional Private Methods

You will find it useful to implement additional private constructors, methods within `Empty`, etc. to solve this lab.

Grading

This lab contributes another 4 marks to your final grade (100 marks). Correctly implementing each of the following item get you 0.4 marks:

- `generate`, `iterate`
- `map`
- `limit`

- `filter`
- `takeWhile`
- `zipWith`
- `unZipWith`
- `reduce`
- `count`, `forEach`
- `findFirst` and `toArray`

You can get -0.5 mark deduction for serious violation of style. Note that "correct" here not only means it gives the correct output, but it should invoked `Supplier` for head as lazily as possible.

Submission

When you are ready to submit your lab, on `cs2030-i`, run the script

```
1  ~cs2030/submit07
```

which will copy all files matching `*.java` (and nothing else) from your `~/lab07` directory on `cs2030-i` to an internal grading directory. We will test compile and test run with a tiny sample input to make sure that your submission is OK.

You can submit multiple times, but only the most recent submission will be graded.

> **Warning**
> Make sure your code are in the right place -- it must be in subdirectory named `lab07`, directly under your home directory, in other words `~/lab07`. If you place it anywhere else, it will not get submitted.