Lab 6

Submission deadline: 2359, Sunday, October 15, 2017.

Prerequisites

This lab assumes that students:

- understand what is immutable function
- familiar with the `Function` inteface
- are familiar with the syntax and semantics of lambda expression and method reference.

Learning Objectives

After completing this lab, students should:

- be familiar with eager versions of operators `map` , `reduce` , `filter` , `generate` , `peek` , `findFirst` , `forEach`
- be familiar with `Predicate` , `Supplier` , `Consumer` , and `BiFunction` interfaces provided by Java 8.
- appreciate the usefulness of `Optional` objects
- be familiar with writing methods with variable number of arguments.

Setup

The skeleton code from Lab 6 is available on `cs2030-i` under the directory `~cs2030/lab06`. There are two files, `LambdaList.java` which you are asked to complete, and `LabSix.java`, which contains simple test code to test the behavior of LambdaList.

Task

For Lab 6, you are asked to implement eight methods for the class `LambdaList`, a simple list that supports various lambda operations.

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!)

LambdaList

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

`LambdaList` does not implement Java's `List` interface though. The only similarity to an `ArrayList` is that it supports the `add` method. `LambdaList`, however, is *immutable* and is written in functional-style. You cannot change a given list -- every time you add something to the list, a new list is returned.

```
LambdaList<Integer> list = new LambdaList<>();
list.add(4);
System.out.println(list); // prints [] instead of [4]
```

This means that we can add things to a list using a chain of method calls:

```
list = list.add(4).add(5).add(0);
```

We can also create a new `LambdaList` using constructor:

```java
LambdaList<Integer> intList = new LambdaList<>(1, 2, 3, 4, 5);
LambdaList<Integer> strList = new LambdaList<>("raw", "power");
```

Methods with Varargs
This is the first time we see a method with variable number of arguments in CS2030. Take a look at the method definition of the constructor to see how to declare and use varargs (variable number of arguments) in Java.

Internally (privately), `LambdaList` maintains the list using an `ArrayList` object. A private constructor for `LambdaList` takes in an `ArrayList` object as input -- this constructor will be useful for you.

A `LambdaList` supports eight methods that take in lambda expressions and operates on the list elements.

`LambdaList` operates *eagerly* on the elements. This means that the lambdas and method references are evaluated as soon as they are called. There is a better way to do the same operations using `Stream` interfaces -- which we will visit next lecture. For this reason, you should forget about `LambdaList` after this lab is over, and never use it again!

Since the goal of this lab is to get you familiarize with the semantics of these methods and functional interfaces in Java 8, you are *NOT ALLOWED* to solve this lab using the `Stream` interfaces.

Here are the eight methods that you needs to implement:

1. `generate`

`generate` is a static method that returns a new list, where each element in the list is generated with a given Supplier [https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html]. A `Supplier` is a function interface, just like `Function`, but it takes no argument and returns an element. To invoke a supplier, we call `get()`.

For example:

```java
Random r = new Random();
```

```java
Supplier<Double> s = r::nextDouble;
s.get(); // returns a random double
```

`generate` also takes in an `int` argument for the number of elements in the list to generate.

For instance, you can call:

```java
LambdaList.generate(10, r::nextInt);
```

to generate a list with 10 random integers.

2. `map`

`map` is similar to the method I showed in Lecture 7. It takes in a function f and applies it to each element x in the list, and returns a list of $f(x)$.

For example:

```java
LambdaList<Integer> intList = new LambdaList<>(1, 2, 3, 4, 5);
intList.map(x -> x + 1); // returns [2, 3, 4, 5, 6]
```

3. `filter`

`filter` takes in a `Predicate` [https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html]. A `Predicate` is a function interface with a method `test` that takes in an element and returns a boolean.

For example:

```java
Predicate<String> hasTwoLetters = x -> x.length() == 2;
hasTwoLetters.test("yes"); // returns false
```

`filter` returns a new list, containing only elements in the list that pass the predicate test (i.e., the predicate returns true).

Example:

```
LambdaList<String> list = new LambdaList<>("show", "me", "my","p
list.filter(x -> x.length() == 2); // returns [me, my, in]
```

4. `countIf`

`countIf` is so called a `reduction` operation, which reduces a list to a single value. It also takes a predicate, and returns the number of elements that passes the predicate test.

Example:

```
list.countIf(hasTwoLetters); // returns 3
```

5. `reduce`

`reduce` is a generalized `reduction` operation. It takes in a `BiFunction` [https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html]. A `BiFunction` is just like `Function`, but it takes in two arguments and return a result.

Here is an example of `BiFunction`:

```
BiFunction<Integer, CustomerQueue, Integer> addLength = (sum, q)
```

`reduce` takes in a `BiFunction` that is called the *accumulator* -- it basically goes through the list, and accumulate all the values into one. The accumulator requires an initial value to start with. This initial value is the *identity* of the `BiFunction` (in mathematical notation, for identity i, $f(i, x) = x$ for any x).

For instance, the following computes the total number of customers in the list of customer queues `list`:

```
list.reduce(0, (sum, q) -> sum + q.length());
```

Here is another example:

```java
LambdaList<Integer> list = new LambdaList<>(4, 3, 2, 1);
list.reduce(1, (prod, x) -> prod * x); // returns 24
```

Note that you can implement `countIf` by combining `filter` and `reduce` .

6. `findFirst`

`findFirst` also takes in a predicate, and finds the first element in the list that pass the predicate.

The `findFirst` method returns an `Optional<T>` object. `Optional` [https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html] is a wrapper around a reference to `T` , or `null` . It is introduced in Java 8 and is a neat way to avoid checking for `null` .

Let's first see how we can create a new `Optional` object -- if you want to wrap a non- `null` value in an `Optional` , just call `Optional.of(value)` . Otherwise, if you want to wrap it in a `null` , call `Optional.empty()` .

Alternatively, if you do not want to check if value is null or not, call `Optional.ofNullable(value)` which will return one of the above appropriately for you.

How is returning an `Optional` useful? For instance, you can do the following:

```java
serverList.findFirst(s -> s.isIdle())
          .orElse(shop.getRandomServer());
```

7. `peek`

The `peek` method exists mainly for debugging purposes. It returns the original list, and simply call a given `Consumer` [https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html], another function interface, to consume each element. Here is an example `Consumer` :

```java
Consumer<String> c = System.out::println
c.accept("aaaaaahhh!");
```

The most common usage of `peek` is to stick it in between a chain of operations to print out the intermediate elements of the list.

```
list.map(x -> x.length()).peek(System.out::println).reduce(0, (x,
```

8. `forEach`

`forEach` is just like `peek`, except that it does not return a list.

Grading

This lab contributes another 4 marks to your final grade (100 marks).

Each correct implementation of eight methods above will earn you 0.5 marks.

You can get -0.5 mark deduction for serious violation of style.

Submission

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

```
~cs2030/submit06
```

which will copy all files matching `*.java` (and nothing else) from your `~/lab06` 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.