3 April – 5 April 2019
Tutorial 8
Lazy Evaluation

1. Implement a class `LazyInt` that encapsulates a lazily evaluated Integer value. A `LazyInt` is specified by a `Supplier`, such that when the value of the `LazyInt` is needed, the `Supplier` will be evaluated to yield the value. Otherwise, the evaluation is delayed as much as possible.

   `LazyInt` supports the following operations:

   - `map` returns a `LazyInt` consisting of the results of applying the given function to the value of this `LazyInt`.

   - `flatMap` returns a `LazyInt` consisting of the results of replacing the value of this `LazyInt` with the value of a mapped `LazyInt` produced by applying the provided mapping function to the value.

   - `get` returns the value of `LazyInt`.

   As an example, the expression below will return 200.

   ```
   new LazyInt(() -> 10)
       .map(x -> x * x)
       .flatMap(x -> new LazyInt(() -> x * 2))
       .get()
   ```

   Given the skeleton class with `import` statements omitted for brevity, complete the method bodies of `map` and `flatMap`.

   ```
   class LazyInt {
       Supplier<Integer> supplier;

       LazyInt(Supplier<Integer> supplier) {
           this.supplier = supplier;
       }

       int get() {
           return supplier.get();
       }

       LazyInt map(Function<? super Integer, Integer> mapper) {
           // To complete
       }

       LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
           // To complete
       }
   }
   ```

2. Study the following implementation of an infinite list.

```java
public interface IFL<T> {

    public static <T> IFL<T> iterate(T seed, Function<T, T> next) {
        return new IFLImpl<T>() {
            T element = seed;
            Function<T, T> func = x -> {
                func = next;
                return element;
            };

            Optional<T> get() {
                element = func.apply(element);
                return Optional.of(element);
            }
        };
    }

    public <R> IFL<R> map(Function<T, R> mapper);
    public void forEach(Consumer<T> action);
}

abstract class IFLImpl<T> implements IFL<T> {
    public <R> IFL<R> map(Function<T, R> mapper) {
        return new IFLImpl<R>() {
            Optional<R> get() {
                return IFLImpl.this.get().map(mapper);
            }
        };
    }

    public void forEach(Consumer<T> action) {
        Optional<T> curr = get();
        while (curr.isPresent()) {
            action.accept(curr.get());
            curr = get();
        }
    }

    abstract Optional<T> get();
}
```

(a) Modify the `iterate` method such that it now supports a condition to stop iterating.

```
IFL<Integer> if = IFL.iterate(0, i -> i < 2, i -> i + 1);
```

(b) Suppose we call

```
IFL.iterate(0, i -> i < 2, i -> i + 1).map(f).map(g).forEach(c)
```

where `f` and `g` are lambda expressions of type `Function` and `c` is a lambda expression of type `Consumer`. Let `e` be the lambda expression `i -> i + 1` passed to `iterate`. Write down the sequence of which the lambda expressions `e`, `f`, `g`, and `c` that are evaluated. Verify your answer.

(c) Define method `concat` takes in two IFL objects, `ifl1` and `ifl2`, and creates a new IFL whose elements are all the elements of the first list `ifl1` followed by all the elements of the second list `ifl2`.

```
public static <T> LazyList<T> concat(LazyList<T> l1, LazyList<T> l2)
```

The elements in newly concatenated list must be lazily evaluated as well. For example, in

```
IFL<Integer> ifl1 = IFL.iterate(0, i -> i < 2, i -> i + 1);
IFL<Integer> ifl2 = IFL.iterate(5, i -> i < 8, i -> i + 2);
IFL<Integer> ifl3 = IFL.concat(ifl1, ifl2);
ifl3.forEach(x -> System.out.print(x + " "));
```

Being a lazy-evaluated, nothing is evaluated when `ifl3` is created. Thus, `concat` should not result in an infinite loop even if the list `ifl1` infinitely long. The elements are only evaluated when terminal operator `forEach` is called. In the example above, `0 1 5 7` will be printed.

3. The following depicts a classic tail-recursive implementation for finding the sum of values of n (given by $\sum_{i=0}^{n} i$) for $n \geq 0$.

```
static long sum(long n, long result) {
    if (n == 0) {
        return result;
    } else {
        return sum(n - 1, n + result);
    }
}
```

In particular, the implementation above is considered **tail-recursive** because the recursive function is at the tail end of the method, i.e. no computation is done after the recursive call returns. As an example, `sum(100, 0)` gives `5050`.

However, this recursive implementation causes a `java.lang.StackOverflowError` error for large values such as `sum(100000, 0)`.

Although the tail-recursive implementation can be simply re-written in an iterative form using loops, we desire to capture the original intent of the tail-recursive implementation using delayed evaluation via the `Supplier` functional interface.

We represent each recursive computation as a `Compute<T>` object. A `Compute<T>` object can be either:

- a recursive case, represented by a `Recursive<T>` object, that can be recursed, or
- a base case, represented by a `Base<T>` object, that can be evaluated to a value of type `T`.

As such, we can rewrite the above `sum` method as

```
static Compute<Long> sum(long n, long s) {
    if (n == 0) {
        return new Base<>(() -> s);
    } else {
        return new Recursive<>(() -> sum(n - 1, n + s));
    }
}
```

and evaluate the sum of n terms via the `summer` method below:

```
static long summer(long n) {
    Compute<Long> result = sum(n, 0);

    while (result.isRecursive()) {
        result = result.recurse();
    }

    return result.evaluate();
}
```

(a) Complete the program by writing the `Compute`, `Base` and `Recursive` classes.

(b) By making use of a suitable client class `Main`, show how the "tail-recursive" implementation is invoked

(c) Redefine the `Main` class so that it now computes the factorial of n recursively.