CS2030 Programming Methodology
Semester 1 2022/2023

26 & 27 October 2022
Problem Set #9
Lazy Evaluation

1. Suppose we are given the following `foo` method that represents a *pure function*:

```
int foo() {
    System.out.println("foo method evaluated");
    return 1;
}
```

In order for `foo` to be evaluated lazily, we "wrap" the `foo` method within a `Supplier`:

```
jshell> Supplier<Integer> supplier = () -> foo()
supplier ==> $Lambda$...
```

Notice that the `foo` method will only be evaluated when we invoke the `Supplier`'s `get` method.

```
jshell> supplier.get()
foo method evaluation
$.. ==> 1
```

However, repeated invocations of the supplier's `get` method would result in the `foo` method being re-evaluated despite that the same value will be returned.

```
jshell> supplier.get()
foo method evaluation
$.. ==> 1
```

Let's create a `Lazy` context so as to *cache* the result of the first evaluation. In this way, *referential transparency* ensures that subsequent evaluations would only require the cached value in the `Lazy` context to be returned.

(a) Define a `Lazy<T>` class with the factory method `of(Supplier<T> supplier)`.

```
jshell> Lazy<Integer> lazyint = Lazy.<Integer>of(() -> foo())
lazyint ==> Lazy@3941a79c
```

(b) Write a `get()` method that returns the value within the `Lazy` context. Note that the value is only evaluated once; subsequent calls to the `get` method returns the cached value.
Hint: Declare the cache as a non-final variable of type `Optional<T>`*, so that the first call to the method* `get` *would assign it with the evaluated value. Although it might seem that the* `Lazy` *class is no longer immutable, it is still observably immutable.*

seems like immutable to the client

```
jshell> lazyint.get() // first call to get()
foo method evaluated
$.. ==> 1

jshell> lazyint.get() // subsequent call to get()
$1.. ==> 1
```

(c) Include the `map` and `flatMap` methods.

```
jshell> Lazy<Integer> lazyint = Lazy.<Integer>of(() -> foo())
lazyint ==> Lazy@9807454

jshell> lazyint.map(x -> x + 1)
$.. ==> Lazy@b1bc7ed

jshell> lazyint.map(x -> x + 1).get()
foo method evaluated
$.. ==> 2

jshell> lazyint.map(x -> x * 2).get()
$.. ==> 2

jshell> Lazy<Integer> lazyint = Lazy.<Integer>of(() -> foo())
lazyint ==> Lazy@70177ecd

jshell> Function<Integer, Lazy<Integer>> addOne = x -> Lazy.<Integer>of(() -> x).
   ...> map(y -> y + 1)
addOne ==> $Lambda$...

jshell> lazyint.flatMap(addOne)
$.. ==> Lazy@65b3120a

jshell> lazyint.flatMap(addOne).get()
foo method evaluated
$.. ==> 2

jshell> lazyint.flatMap(addOne).get()
$.. ==> 2

jshell> Lazy<Integer> lazyint = Lazy.<Integer>of(() -> foo())
lazyint ==> Lazy@7e6cbb7a

jshell> Function<Integer, Lazy<Integer>> addFoo = x -> lazyint.map(y -> y + x)
addFoo ==> $Lambda$...

jshell> lazyint.flatMap(addFoo).get()
foo method evaluated
$.. ==> 2
```

```
jshell> lazyint.flatMap(addFoo).get()
$.. ==> 2
```

(d) By including an appropriate overriding `equals` method, demonstrate that the `map` and `flatMap` methods obeys the identity and associativity laws of the Functor and Monad. in the absence of side effects

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

```
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

```
Compute<Long> sum(long n, long s) {
    if (n == 0) {
        return new Base<Long>(() -> s);
    } else {                                the recursive call is wrapped in a object
        return new Recursive<Long>(() -> sum(n - 1, n + s));
    }
}
```

and evaluate the sum via the `summer` method below:

```
long summer(long n) {
    Compute<Long> result = sum(n, 0);
                        only for Recursive<T> not Base<T>
    while (result.isRecursive()) {
        result = result.recurse();
    }                           Recursive<T> method

    return result.evaluate();
}
```

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

(b) Demonstrate how the above classes can be used to find

- the sum of values from 0 to n;
- the factorial of n