**CS2030 Programming Methodology**
Semester 1 2023/2024

Week of 7 – 11 October 2024
Problem Set #6
**Lazy Evaluation**

1. Suppose you are given the following `foo` method:

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

To evaluate `foo` lazily (i.e. only when necessary), we "wrap" the `foo` method within a `Supplier`:

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

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

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

However, repeated invocations of the `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
```

Ideally, the result of the first evaluation should be *cached* so that subsequent calls to `get` will return the cache value.

```
jshell> lazy = Lazy.<Integer>of(() -> foo())
$.. ==> Lazy@..

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

jshell> lazy.get()
$.. ==> 1
```

You are given the following `Lazy` class:

```
class Lazy<T> implements Supplier<T> {
    private final Supplier<? extends T> supplier;

    Lazy(Supplier<? extends T> supplier) {
        this.supplier = supplier;
    }

    public T get() {
        return this.supplier.get();
    }
}
```

(a) We include a *non-final* property `private Optional<T> cache` into the `Lazy` class,

```
class Lazy<T> implements Supplier<T> {
    private final Supplier<? extends T> supplier;
    private Optional<T> cache; // cannot be final
    ...
```

Rewrite the constructor and `get` method such that the first invocation of `get` will call the `Supplier's get` method to perform the first evaluation and cache the result in `cache`. Subsequent `get` invocations will simply return the cached value.

(b) Is the `Lazy` class immutable?

(c) Include the `map` method to map a lazy value. Observe how `map` should behave.

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

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

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

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

(d) Lastly, include the `flatMap` method and test the laziness behaviour.

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

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

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

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@..

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
```