CS2030 Programming Methodology
Semester 1 2023/2024

1 & 2 November 2023
Problem Set #9
Lazy Evaluation

1. Study the following implementation of the Lazy class.

```
import java.util.function.Supplier;
import java.util.function.Function;

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

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

    static <T> Lazy<T> of(Supplier<? extends T> supplier) {
        return new Lazy<T>(supplier);
    }

    static <T> Lazy<T> of(T t) {
        return new Lazy<T>(() -> t);
    }

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

    <R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
        Supplier<R> supplier = () -> mapper.apply(this.get());
        return Lazy.<R>of(supplier);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        } else if (obj instanceof Lazy<?> other) {
            return this.get().equals(other.get());
        } else {
            return false;
        }
    }
}
```

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

```
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` and pass it to `Lazy.of`:

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

jshell> lazy = Lazy.<Integer>of(supplier)
$.. ==> Lazy@6f7fd0e6
```

The `foo` method will only be evaluated when we invoke the `Lazy`'s `get` method.

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

Since pure functions are referentially transparent (i.e. we can always replace them with the resulting value), `Lazy` should *cache* the result of the first evaluation and return this cached value during subsequent calls of `get`.

```
jshell> lazy = Lazy.<Integer>of(() -> foo())
$.. ==> Lazy@6f7fd0e6

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

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

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

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

re-write the `get` method such that the first invocation of `Lazy::get` will call the `Supplier::get` to perform the first evaluation and cache the result in `cache`. Subsequent `Lazy::get` invocations will simply return the cached value.

(b) Is the `Lazy` class immutable?

(c) Include the `flatMap` method and demonstrate how `Lazy` obeys the identity and associativity laws of the `map` (`Lazy` is a Functor) and `flatMap` (`Lazy` is a Monad).

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 {
        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);

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

    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