Ex 6: The Art of Being Lazy

Basic Information

- **Deadline:** 22 October 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★★★★

Prerequisite

- Caught up to Unit 32 of Lecture Notes.
- Familiar with CS2030S Java style guide.

Class Files

If you have not finished Programming Exercise 5, do not worry. We provide `.class` files for the functional interfaces as well as `Maybe<T>`. Note that the implementation for `Maybe<T>` is badly written not following OOP but it is the correct implementation.

Additionally, the class files were compiled on PE node using Java 21 compiled. If you are not using Java 21 or if you are not working on PE node, you may get different result. It is unlikely, but the possibility is there. Please only do your work on the PE node.

Note that since we provide only the `.class` files for `Maybe<T>` and the functional interfaces, you may need to compile `Lazy<T>` with the following command from `ex6-username` directory.

```
1  javac cs2030s/fp/Lazy.java
```

Maybe

Our `Maybe` class has the following methods available. Methods that are not available cannot be used. You should not modify `Maybe` class. You may use the method descriptor as an inspiration for future exercises to make your method more flexible. All the methods below are `public`. No other `public` methods are available.

Method	Description
`static <T> Maybe<T> of (T val)`	Returns a new `Maybe<T>` depending on the value of `val`. • If `val` is `null` returns the singleton `NONE` without any value inside • Otherwise returns a new `Maybe<T>` with the content `val`.
`static <T> Maybe<T> some (T val)`	Returns a new `Maybe<T>` with the content `val` regardless if `val` is `null` or not.
`static <T> Maybe<T> none (T val)`	Returns a the singleton `NONE` without any value inside.
`<U> Maybe<U> map (Transformer<? super T, ? extends U> func)`	Transform `this.val` (*if any*) using `func` and return a new `Maybe<U>`. • If there is no `this.val` returns the singleton `NONE` without any value inside. • Otherwise return a new instance of `Maybe<U>` with `this.val` transformed using `func`.
`Maybe<T> filter (BooleanCondition<? super T> pred)`	Transform `this.val` (*if any*) depending on the result of `pred`. • If there is no `this.val` returns the singleton `NONE` without any value inside. • If `this.val == null` returns the singleton `NONE` without any value inside. • If `this.val` evaluates to `false` when passed into `pred` returns the singleton `NONE` without any value inside. • Otherwise the current instance.
`<U> Maybe<U> flatMap (Transformer<? super T, ? extends Maybe<? extends U>> func)`	Transform `this.val` (*if any*) using `func` and return a new `Maybe<U>`. • If there is no `this.val` returns the singleton `NONE` without any value inside.

Method	Description
	• Otherwise return a new instance of `Maybe<U>` with `this.val` transformed using `func` but without making a nested `Maybe`.
`T orElse (Producer<? extends T> prod)`	Returns `this.val` (*if any*). • If there is no `this.val` produce a new value using `prod`. • Otherwise `this.val`.
`void ifPresent (Consumer<? super T> cons)`	Consumes `this.val` (*if any*). • If there is no `this.val` do nothing. • Otherwise consume `this.val` using `cons`.

Being Lazy

Programming languages such as Scala support lazy values, where the expression that produces a lazy value is not evaluated until the value is needed. Lazy value is useful for cases where producing the value is expensive, but the value might not eventually be used. Java, however, does not provide a similar abstraction. So, you are going to build one.

You are required to design a single `Lazy<T>` class as part of the `cs2030s.fp` package with **two** instance fields and no class fields. You are not allowed to add additional instance/class fields to `Lazy<T>`.

```
1  public class Lazy<T> {
2    private Producer<T> producer;
3    private Maybe<T> value;
4
5     :
6  }
```

While you cannot add new fields, you should make the current field more flexible whenever possible. Furthermore, in all discussion below, the method signature given may not be the most flexible. Your task is to determine if they can be made more flexible. If they can, you should use the most flexible type while minimizing the number of type parameters by using wildcards.

Constraints

You should minimize the use of conditional statements and conditional expressions. In many cases, this can be done by using the appropriate methods from `Maybe<T>`. You are also not allowed to have nested class within `Lazy<T>` to avoid conditional statements/expressions by using **polymorphism**.

If you have done the design correctly, you will have no conditional statements/expressions except for the `boolean equals(Object)` method.

The basic idea is that we can match the concept of `None<T>` to a lazy value that is not yet computed and the concept of `Some<T>` to a lazy value that is already computed. The proper name for this is that they are **isomorphic**.

Tasks

Task 1: Basic

Define a generic `Lazy` class to encapsulate a value with the following operations:

- `static of(T v)` method that initializes the `Lazy` object with the given value.
- `static of(Producer<T> s)` method that takes in a producer that produces the value when needed.
- `get()` method that is called when the value is needed. If the value is already available, return that value; otherwise, compute the value and return it. The computation should only be done once for the same value.
- `toString()` : returns `"?"` if the value is not yet available; returns the string representation of the value otherwise.
  - You are encouraged to use `String.valueOf(obj)` instead of `obj.toString()` to avoid runtime error when `obj` is `null`.

> **Immutable?**
>
> For our `Lazy<T>` to be immutable and to make the memoization of the value transparent, `toString` should call `get()` and should never return `"?"`. We break the rules of immutability and encapsulation here, just so that it is easier to debug and test the laziness of your implementation.

Sample Usage

```
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Lazy

jshell> Lazy<Integer> eight = Lazy.of(8)
jshell> eight
eight ==> 8
jshell> eight.get()
$.. ==> 8

jshell> Producer<String> s = () -> "hello"
jshell> Lazy<Object> hello = Lazy.of(s)
jshell> Lazy<String> hello = Lazy.of(s)
jshell> hello
hello ==> ?
jshell> hello.get()
$.. ==> "hello"

jshell> s = () -> { System.out.println("world!"); return "hello"; }
jshell> Lazy<String> hello = Lazy.of(s)
jshell> hello
hello ==> ?
jshell> hello.get()
world!
$.. ==> "hello"

jshell> // check that "world!" should not be printed again.
jshell> hello.get()
$.. ==> "hello"

jshell> Random rng = new Random(1)
jshell> Producer<Integer> r = () -> rng.nextInt()
jshell> Lazy<Integer> random = Lazy.of(r)

jshell> // check that random value should not be available
jshell> random
random ==> ?

jshell> // check that random value is obtained only once
jshell> random.get().equals(random.get())
$.. ==> true

jshell> // should handle null
jshell> Lazy<Object> n = Lazy.of((Object)null)
jshell> n.toString()
$.. ==> "null"
jshell> n.get()
$.. ==> null

jshell> Lazy<Integer> n = Lazy.of((Producer<Integer>)() -> null)
jshell> n
n ==> ?
jshell> n.get()
$.. ==> null
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test1.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test1.java
java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex6_style.xml cs2030s/fp/*.java
```

Task 2: Map and FlatMap

Now let's add the `map` and `flatMap` method. Remember that `Lazy` should not evaluate anything until `get()` is called, so the function `f` passed into `Lazy` through `map` and `flatMap` should not be evaluated until `get()` is called. Furthermore, they should be evaluated once. That result from `map` and `flatMap`, once evaluated, should be cached (*also called memoized*), so that function must not be called again.

Sample Usage

```
jshell> import cs2030s.fp.Lazy
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Transformer

jshell> Producer<String> s = () -> "123456"
jshell> Lazy<String> lazy = Lazy.of(s)
jshell> lazy.map(str -> str.substring(0, 1))
$.. ==> ?
jshell> lazy
$.. ==> ?
jshell> lazy.map(str -> str.substring(0, 1)).get()
$.. ==> "1"
jshell> lazy.get()
$.. ==> "123456"

jshell> Transformer<String, String> substr = str -> {
   ...>   System.out.println("substring");
   ...>   return str.substring(0, 1);
   ...> }
jshell> lazy = lazy.map(substr)
jshell> lazy.get()
substring
$.. ==> "1"
jshell> lazy.get()
$.. ==> "1"

jshell> Lazy<Integer> lazy = Lazy.of(10)
jshell> lazy = lazy.map(i -> i + 1)
jshell> lazy = lazy.flatMap(j -> Lazy.of(j + 3))
jshell> lazy
lazy ==> ?
jshell> lazy.get()
```

```
$.. ==> 14
jshell> lazy
lazy ==> 14

jshell> Transformer<Object, Integer> hash = x -> x.hashCode();
jshell> Lazy<Number> lazy = Lazy.<String>of("sunday").map(hash);
jshell> Transformer<Object, Lazy<Integer>> hash = x -> Lazy.
<Integer>of(x.hashCode());
jshell> Lazy<Number> lazy = Lazy.<String>of("sunday").flatMap(hash);
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test2.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test2.java
java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex6_style.xml cs2030s/fp/*.java
```

Task 3: Filter and Equality

Write a `filter` method, which takes in a `BooleanCondition` and lazily tests if the value passes the test or not. Returns a `Lazy<Boolean>` object. The `BooleanCondition` must be executed at most once.

Then write an `equals`, which overrides the `equals` method in the `Object` class. `equals` is an **eager** operation that causes the values to be evaluated (*if not already cached*). `equals` should return `true` only if both objects being compared are `Lazy` and the value contains within are equals (*according to their* `equals()` *methods*).

Sample Usage

```
jshell> import cs2030s.fp.Lazy

jshell> Lazy<Integer> fifty = Lazy.of(50)
jshell> Lazy<Boolean> even = fifty.filter(i -> i % 2 == 0)
jshell> even
even ==> ?
jshell> even.get()
$.. ==> true
jshell> even
even ==> true

jshell> // equals
jshell> fifty.equals(Lazy.of(5).map(i -> i * 10))
$.. ==> true
jshell> fifty.equals(50)
$.. ==> false
jshell> fifty.equals(Lazy.of("50"))
$.. ==> false
```

```
jshell> even.equals(Lazy.of(true))
$.. ==> true

jshell> BooleanCondition<String> isHello = s -> {
   ...>   System.out.println(s);
   ...>   return s.equals("hello");
   ...> }
jshell> Lazy<Boolean> same = Lazy.of("hi").filter(isHello)
jshell> same
same ==> ?
jshell> same.get()
hi
$.. ==> false
jshell> same.get()
$.. ==> false

jshell> BooleanCondition<Object> alwaysFalse = s -> false
jshell> Lazy<Boolean> same = Lazy.<String>of("hi").filter(alwaysFalse)

jshell> Producer<String> producer = () -> "123456";
jshell> Lazy<String> oneToSix = Lazy.of(producer);
jshell> oneToSix.toString();
$.. ==> ?
jshell> oneToSix == oneToSix
$.. ==> true
jshell> oneToSix.toString();
$.. ==> ?
jshell> oneToSix.equals(oneToSix)
$.. ==> true
jshell> oneToSix.toString();
$.. ==> 123456
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test3.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test3.java
java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex6_style.xml cs2030s/fp/*.java
```

Task 4: Combine

We have provided an interface called `Combiner<S, T, R>` in `cs2030s.fp`, with a single `combine` method to combine two values, of type `S` and `T` respectively, into a result of type `R`.

Add a method called `combine` into `Lazy`. The `combine` method takes in another `Lazy` object and a `Combiner` implementation to lazily combine the two Lazy objects (*which may contain values of different types*) and return a new `Lazy` object.

```
jshell> import cs2030s.fp.Lazy
jshell> Lazy<Integer> five, ten, fifty, hundred
jshell> ten = Lazy.of(10)
jshell> five = Lazy.of(5)
jshell> // combine (same types)
jshell> Combiner<Integer, Integer, Integer> add = (x, y) -> {
   ...>   System.out.println("combine");
   ...>   return x + y;
   ...> }
jshell> fifty = five.combine(ten, (x, y) -> x * y)
jshell> fifty
fifty ==> ?
jshell> hundred = fifty.combine(fifty, add)
jshell> hundred
hundred ==> ?
jshell> // combine (different types)
jshell> Combiner<Integer,Double,String> f = (x, y) -> Integer.toString(x)
+ " " + Double.toString(y)
jshell> Lazy<String> s = Lazy.of(10).combine(Lazy.of(0.01), f)
jshell> s
s ==> ?
jshell> s.get()
$.. ==> "10 0.01"

jshell> Combiner<Object,Object,Integer> f = (x, y) -> x.hashCode() +
y.hashCode()
jshell> Lazy<Number> n = Lazy.<String>of("hello").combine(Lazy.
<Integer>of(123), f);
```

You can test this more comprehensively by running without compilation warning/error and all tests printing `ok`. Make sure your code follows the CS2030S Java style.

Test4.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test4.java
java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex6_style.xml cs2030s/fp/*.java
```

Skeleton for Programming Exercise 5

We provide `.class` files for the functional interfaces as well as `Maybe<T>`. There are also template files for `Lazy.java` and `Combiner.java` in `cs2030s/fp` directory. Some files (*e.g.*, `Test1.java`, `Test2.java`, `CS2030STest.java`, *etc*) are provided for testing. You may edit them to add your own test cases, but we will be using our own version for testing.

While there is no given public test cases for it, we will test your code with hidden test cases that checks for flexible type. Additionally, minimize the number of type parameter by using wildcards. Lastly, ensure that you use `@SuppressWarnings` as needed.

Following CS2030S Style Guide

You should make sure your code follows the given Java style guide.

Further Deductions

Additional deductions may be given for other issues or errors in your code. These deductions may now be unbounded, up to 5 marks. This include *but not limited to*

- run-time error.
- failure to follow instructions.
- improper designs (*e.g.*, not following good OOP practice).
- not comenting `@SuppressWarnings`.
- misuse of `@SuppressWarnings` (*e.g.*, not necessary, not in smallest scope, *etc*).

Documentation (Optional)

Documenting your code with Javadoc is optional for Programming Exercise 6. It is, however, always a good practice to include comments to help readers understand your code.