Lab 4: Box

- Deadline: 1 March, 2022, Tuesday, 23:59, SST
- Mark: 4%

Prerequisite:

- Caught up to Unit 26 of Lecture Notes
- Familiar with CS2030S Java style guide

Probably Just a Value but Maybe Nothing?

In this lab, you are given our own generic wrapper class, a `Probably<T>`. This is a wrapper class that can be used to store a value of any reference type. For now, our `Probably<T>` is not going to be a very useful abstraction. Not to worry. we will slowly add more functionalities to it.

Please read the following explanation on what the class `Probably<T>` is.

The Basics

The class `Probably<T>`:

- contains a `private final` field of type `T` to store the value stored inside.
- overrides the `equals` method from `Object` to compare if the two values inside are the same. Two values are the same according to their respective `equals` method.
- overrides the `toString` method so it returns the string representation of its values, between `<` and `>`.
- provides a class method in `Probably<T>` called `none()` that returns nothing. This nothing is a common nothing and there is exactly one nothing in the program.
- provides a class method in `Probably<T>` called `just(T value)` that returns something that contains just the value unless the value is null then this returns the shared nothing.

The method `none` and `just` are called a *factory method*. A factory method is a method provided by a class for the creation of an instance of the class. Using a public constructor to create an instance necessitates calling `new` and allocating a new object on the heap every time. A factory method, on the other hand, allows the flexibility of reusing the same instance.

With the availability of the factory methods, `Probably<T>` should keep the constructor private.

The sequence below shows how we can use a `Probably` using the methods above.

```
jshell> Probably.just(4)
$.. ==> <4>
jshell> Probably.just(Probably.just(0))
$.. ==> <<0>>
jshell> Probably.just(Probably.just(Probably.just("null")))
$.. ==> <<<null>>>
jshell> Probably.just(Probably.just(Probably.none()))
$.. ==> <<<>>>
jshell> Probably.just(Probably.just(null))
$.. ==> <<>>
jshell> Probably.just(4).equals(Probably.just(4))
$.. ==> true
jshell> Probably.just(4).equals(4)
$.. ==> false
jshell> Probably.just(Probably.just(0)).equals(Probably.just(0))
$.. ==> false
jshell>
Probably.just(Probably.just(0)).equals(Probably.just(Probably.just(0)))
$.. ==> true
jshell> Probably.just("string")
$.. ==> <string>
jshell> Probably.just("string").equals(Probably.just(4))
$.. ==> false
jshell> Probably.just("string").equals(Probably.just("null"))
$.. ==> false
jshell> Probably.just(null)
$.. ==> <>
jshell> Probably.none()
$.. ==> <>
jshell> Probably.none().equals(Probably.just(null))
$.. ==> true
jshell> Probably.none() == Probably.just(null)
$.. ==> true
```

You can check that our `Probably<T>` is correct by running:

```
javac -Xlint:rawtypes TestProbably.java
java TestProbably
```

There shouldn't be any compilation warning or error when you compile

`TestProbably.java` and all tests should prints `ok`.

Acting on the Value

The given class `Probably<T>` currently is what we call an *immutable object* because we can create it but we can not mutate the value. Note that we make sure that this is the case by making the value `final`. Also, to avoid any complications, we disallow retrieving the value too.

So what is it good for? How can we do something on it? This is where you come in. You will be writing a few interfaces and classes to act on something that is probably a value but maybe it is also nothing.

Since we cannot access the value directly, we need to be able to give the class the function. Unfortunately, we cannot just put methods as arguments. We need to do something else.

Action

An action is simply a method that accept just some value but does not return anything. First, we create an interface called `Action<T>` with an abstract method called `call` that takes in an argument of generic type `T` and does not return anything.

Second, create an interface called `Actionable<T>` with an abstract method called `act` that takes in an `Action` and does not return anything. Since you cannot use rawtype, what generic type should be added for `Action` in `Actionable`? In particular, do you have to use any bounds?

Now that we have the two interfaces, we will now use this interface. First, we will create the simplest of the action which is to print any value that is given to it. Call this class `Print` and this class should just simply print the string representation of the value contained.

Next, have `Probably<T>` implements the interface `Actionable<T>`. This will allow us to act on the value directly without actually getting the value ourselves. Since the value contained inside `Probably<T>` can be just a value but maybe also nothing, when it is actually nothing, we should not even do anything! Thus, we have prevent `null` value from escaping (*In this case, `null` does not escape from `Probably<T>` to the parameter of the input argument*).

```
jshell> Probably.just(4).act(new Print())
$.. ==> 4
```

```
jshell> Probably.just("string").act(new Print())
$.. ==> string
jshell> Probably.none().act(new Print())
$.. ==>
```

You can test the additions to `Probably<T>` above more comprehensively by running:

```
javac -Xlint:rawtypes Test1.java
java Test1
```

There shouldn't be any compilation warning or error when you compile `Test1.java` and all tests should prints `ok`.

Immutator

Now, we are going to write two other interfaces (along with their implementations) to allow us to mutate the value that is contained inside. However, since we need to make sure that the value is still immutable, we cannot really mutate the value. Instead, we will create a new `Probably<T>` whenever we mutate the value. So, any mutation not only change the value but also change `Probably<T>`. In particular, the type may also be mutated!

Since we are trying to mutate the value while keeping it immutable, we simply call this interface `Immutator<T2, T1>` with an abstract method called `invoke` that takes in an argument of generic type `T1` and returns a value of generic type `T2`. In fact, what we are trying to abstract out is exactly the method with the following method signature:

```
T2 invoke(T1 t1); // T1 change to T2
```

First, have `Probably<T>` also implements `Immutatorable<T>`. This will allow us to change the value but at the expense of keep on getting a new class whenever something changed. Since the value contained inside `Probably<T>` can be just a value but maybe also nothing, when it is actually nothing, we need to return the equivalent of nothing. So, nothing in means nothing out.

Before you embark on this quest, think very carefully about what the return type of `func` method should be. In particular, is it fixed or can we change this? (*Hint: didn't we have a recitation about this?*).

Next, create your own special `Immutator`. This `Immutator` is special because when invoked, it accepts a `Probably<T1>` and returns `Probably<T2>`. What would the method signature of the `invoke` be? Given the method signature for `invoke`, you should then derive the generic type parameter as well as how it will implement `Immutator` (*i.e., what type arguments should be given to* `Immutator`).

Once you have figured that out, you can then create this special `Immutator` called `Improbable<T2, T1>` in the following way:

- adds a private final `Immutator<T2, T1>` field.
- adds a constructor that accepts and initialise this `Immutator<T2, T1>`.
- implements the method `invoke` inherited from `Immutator`.

You can check the usage in the sample run below.

```
jshell> class Incr implements Immutator<Integer,Integer> {
   ...>   public Integer invoke(Integer t1) {
   ...>     return t1 + 1;
   ...> }
jshell> class Length implements Immutator<Integer,String> {
   ...>   public Integer invoke(String t1) {
   ...>     return t1.length();
   ...> }
jshell> Probably.just(4).func(new Incr())
$.. ==> <5>
jshell> Probably.just(4).func(new Incr()).func(new Incr())
$.. ==> <6>
jshell> Probably.just("string").func(new Length())
$.. ==> <6>
jshell> Probably.just("string").func(new Length()).func(new Incr())
$.. ==> <7>
jshell> Probably.<Integer>none().func(new Incr())
$.. ==> <>
jshell> Probably.<String>none().func(new Length())
$.. ==> <>
jshell> Probably.<String>just(null).func(new Length()).func(new Incr())
$.. ==> <>
jshell> new Improbable<Integer,Integer>(new
Incr()).invoke(Probably.just(4))
$.. ==> <5>
jshell> new Improbable<Integer,String>(new
Length()).invoke(Probably.just("null"))
$.. ==> <4>
jshell> new Improbable<Integer,String>(new
Length()).invoke(Probably.just(null))
$.. ==> <>
```

You can test your additions to `Probably<T>` more comprehensively by running:

```
javac -Xlint:rawtypes Test2.java
java Test2
```

There shouldn't be any compilation warning or error when you compile `Test2.java` and all tests should prints `ok`.

Question

Now, we are going to add a method to allow us to check some properties about the value that is contained inside `Probably<T>`. We only need to know about yes/no result of this property. However, we do not want to just create another interface like the `Immutator` to capture this operation. Instead, we will only be using a special case of `Immutator`.

This new method to add into `Probably<T>` should accept an `Immutator` that returns boolean values (*since it is yes/no*) and it should accept a type `T`. We will call this method as `check`. Once you figure out the method descriptor, you can then write its behaviour as follows:

- if the value inside `Probably<T>` is `null`, then simply return the `NONE`.
- if the value is not `null`, then we can invoke the argument of type `Immutator`. This will return either true or false.
    - if true, then we simply return the current object.
    - if false, then we simply return `NONE`.

Write a simple example of an `Immutator` (*i.e., implements* `Immutator`) called `IsModEq` that can be given as arguments to `check` method.

- The constructor for `IsModEq` accepts two positive integers `div` and `mod`.
- The `invoke` method inherited from `Immutator` accepts an integer `val` and returns true if the remainder when `val` is divided by `div` is equal to `mod`.

```
jshell> class Incr implements Immutator<Integer,Integer> {
   ...>   public Integer invoke(Integer t1) {
   ...>     return t1 + 1;
   ...> }
jshell> class Length implements Immutator<Integer,String> {
   ...>   public Integer invoke(String t1) {
   ...>     return t1.length();
   ...> }
jshell> Probably.just(17).check(new IsModEq(3,2)) // 17 % 3 is equal to 2
$.. ==> <17>
jshell> Probably.just(18).check(new IsModEq(3,2)) // 18 % 3 is not equal
to 2
$.. ==> <>
jshell> Probably.just(16).func(new Incr()).check(new IsModEq(3,2)) // 17
% 3 is not equal to 2
$.. ==> <17>
jshell> Probably.just("string").func(new Length()).check(new
IsModEq(3,2))
$.. ==> <8>
```

You can test your additions to `Probably<T>` more comprehensively by running:

```
javac -Xlint:rawtypes Test3.java
java Test3
```

There shouldn't be any compilation warning or error when you compile `Test3.java` and all tests should prints `ok`.

Applicable

Our `Probably<T>` is actually quite powerful in that it could probably just store an `Immutator` *(or it maybe nothing like usual)*! Our `Probably<T>` can already perform an action depending on whether the value it contains is `null` or not. But it must be supplied with an `Immutator` that match the generic type `T`.

To put it more formally, if we are given a method `T1 -> T2` and we are given `Probably<T1>` then we can produce `Probably<T2>`. As a summary, this is often written as a rectangle:

```
    T1 -------------> T2
    ^                 |
    |                 |
    |                 v
Probably<T1> ---> Probably<T2>
```

Our current extension is to put this `Immutator` into `Probably<T>` to form `Probably<Immutator<T2, T1>>`. Note that this cannot be supplied to the framework above because we need `Immutator<T2, T1>` to really be present. However, in `Probably<Immutator<T2, T1>>`, the `Immutator` itself may probably be nothing! If there is nothing that can be used to operate, then we cannot perform any operation so we just return the ever present `NONE`.

Luckily, this is where the `Applicable<T>` interface will be useful. You should first create this `Applicable<T>` interface as follows:

- it contains a single abstract method called `apply`
- the method `apply` takes in a single parameter called `p` of the type `Probably<Immutator<T2, T>>`
- the method `apply` returns the value of the type `Probably<T2>`

You are advised to really think hard about the type first. The code itself is rather short for all of these questions.

```
jshell> class Incr implements Immutator<Integer,Integer> {
   ...>   public Integer invoke(Integer t1) {
   ...>     return t1 + 1;
   ...> }
jshell> class Length implements Immutator<Integer,String> {
   ...>   public Integer invoke(String t1) {
   ...>     return t1.length();
   ...> }
jshell> Probably<Immutator<Integer,Integer>> justIncr = Probably.just(new
Incr());
jshell> Probably<Immutator<Integer,String>> justLength =
Probably.just(new Length());
jshell> Probably<Immutator<Integer,Integer>> noIncr = Probably.none();
jshell> Probably<Immutator<Integer,String>> noLength = Probably.none();
jshell> Probably.just(17).<Integer>apply(justIncr)
$.. ==> <18>
jshell> Probably.<Integer>none().<Integer>apply(justIncr)
$.. ==> <>
jshell> Probably.just(17).<Integer>apply(noIncr)
$.. ==> <>
jshell> Probably.<Integer>none().<Integer>apply(noIncr)
$.. ==> <>
jshell> Probably.just("string").<Integer>apply(justLength)
$.. ==> <6>
jshell> Probably.<String>none().<Integer>apply(justLength)
$.. ==> <>
jshell> Probably.just("string").<Integer>apply(noLength)
$.. ==> <>
jshell> Probably.<String>none().<Integer>apply(noLength)
$.. ==> <>
```

You can test your additions to `Probably<T>` more comprehensively by running:

```
javac -Xlint:rawtypes Test4.java
java Test4
```

There shouldn't be any compilation warning or error when you compile `Test4.java` and all tests should prints `ok`.

Files

A set of empty files have been given to you. You should only edit these files. You must not add any additional files.

The files `Test1.java`, `Test2.java`, etc., as well as `CS2030STest.java`, are provided for testing. You can edit them to add your own test cases, but they will not be submitted.

Following CS2030S Style Guide

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

Grading

This lab is worth 16 marks and contributes 4% to your final grade. The marking scheme is as follows:

- Style: 2 marks
- Correctness: 14 marks

We will deduct 1 mark for each abuse or unnecessary use of `@SuppressWarnings` and for each raw type.

Note that the style marks are conditioned on the evidence of efforts in solving Lab 4.