Lab 4: Probably

- Deadline: 27 September, 2022, Tuesday, 23:59
- 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>` stores either (1) none[1] or (2) just a value of type `T`[2]. So, it is probably just some value of type `T` or it maybe nothing. It:

- contains a `private final` field of type `T` to store the value of reference type `T`.
- provides a private constructor.
- provides a class method called `none()` that returns nothing[1].
- provides a class method called `just(T value)` that returns something that contains just the value[2].
  - Since there is a possibility that `value` is equal to `null`, in such case, we also return nothing.
- 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 `>`.

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.

`Probably<T>` is also made to be *immutable*. Once created, the value of the field `value` cannot be modified! This is achieved by:

- making the field `final`.
- provide no getter and setter.

Relevant part of the code:

```
class Probably<T> {
  private final T value;
  private static final Probably<?> NONE = new Probably<>(null);

  private Probably(T value) {
    this.value = value;
  }

  public static <T> Probably<T> none() {
    @SuppressWarnings("unchecked")
    Probably<T> res = (Probably<T>) NONE;
    return res;
  }
  public static <T> Probably<T> just(T value) {
    if (value == null) {
      return none();
    }
    return (Probably<T>) new Probably<>(value);
  }

  @Override
  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }
    if (obj instanceof Probably<?>) {
      Probably<?> some = (Probably<?>) obj;
      if (this.value == some.value) {
        return true;
      }
      if (this.value == null || some.value == null) {
        return false;
      }
      return this.value.equals(some.value);
    }
    return false;
  }
```

```
38    @Override
39    public String toString() {
40      if (this.value == null) {
41        return "<>";
42      } else {
43        return "<" + this.value.toString() + ">";
44      }
45    }
46  }
```

Shared Object

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.

Additionally, the factory method as well as having `Probably<T>` immutable allows us to share a common object safely. The most common object here is the concept of nothing[1]. The factory methods return nothing when:

- the input argument to `just(T value)` is `null`, or
- the factory method `none()` is invoked

In both cases, we return a static instance called `NONE`. The sequence below shows how we can use a `Probably` using the methods above.

Testing `Probably<T>`

The following sample run shows the current capability of `Probably<T>`. You should also test using your own test cases to further understand `Probably<T>`.

```
1   jshell> Probably.just(4)
2   $.. ==> <4>
3   jshell> Probably.just(Probably.just(0))
4   $.. ==> <<0>>
5   jshell> Probably.just(Probably.just(Probably.just("null")))
6   $.. ==> <<<null>>>
7   jshell> Probably.just(Probably.just(Probably.none()))
8   $.. ==> <<<>>>
9   jshell> Probably.just(Probably.just(null))
10  $.. ==> <<>>
11  jshell> Probably.just(4).equals(Probably.just(4))
12  $.. ==> true
13  jshell> Probably.just(4).equals(4)
14  $.. ==> false
15  jshell> Probably.just(Probably.just(0)).equals(Probably.just(0))
16  $.. ==> 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

Because `Probably<T>` is immutable, it is not useful for us. Once created, we cannot modify the value and we cannot even get the value back. To make the class more useful, we want to be able to *act* on the object. A simple act can be just printing using `System.out.println`.

One way for us to print the content of the value is to simply add a method called `print`. However, this is not useful as we have to add a new method for each action that we want `Probably<T>` to allow. One way to do this is to abstract a computation. Think of *higher-order function.*

Action Interface

One way to abstract a computation is to imagine having a class with a single method called `call` that perform an action that we want. If we receive an instance of this class, we can invoke the method `call` to perform the action. Thus, we can say that a *collection* of all classes with a single method called `call` is an abstraction of an action.

You have 2 tasks in this section:

1. Create an interface `Action<T>` with a single abstract method called `call` inside the file `Action.java`.
    - The method takes in a single parameter of generic type parameter `T`[3].
    - The method does not return any type.
2. Create a non-generic class `Print` that implements `Action` with the method `call` that prints the string representation of the input argument into the standard output (*i.e.*, `System.out.println`) in the file `Print.java`.

Testing Action

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
jshell> new Print().call(17)
$.. ==> 17
jshell> new Print().call("string")
$.. ==> string
```

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`.

Actionable Interface

With the `Action` interface, we can now perform a custome action. We can do this by adding in the `Probably<T>` class, a method that can accept an `Action`. Since this is a good behaviour to have, we want to create an interface representing all classes that can accept an `Action`. Let's call this interface as `Actionable<T>`.

You have 2 tasks in this section:

1. Create an interface `Actionable<T>` with a single abstract method called `act` inside the file `Actionable.java`.
    - The method takes in a single parameter of type `Action`. This `Action` should accept a type `T`.

- The method does not return any type.

2. Modify `Probably<T>` to implement the interface `Actionable`. You need to implement the method `act` appropriately:
   - if the value in `Probably<T>` is not `null`, we invoke `call` with the value.
   - otherwise, do nothing.

Testing Actionable

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
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 Test2.java
java Test2
```

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

Immutating the Value

So now we can create a container called `Probably<T>` and we can print the content (*with appropriate classes implementing `Action<T>` we may also write the string representation of the value into a file, etc*). This is still not very useful since we cannot mutate the value. In the first place, we can not mutate the value because there is no mutator and the field `value` is declared with `final`. Instead, what we want to do is whenever there is a mutation, we want to create a new instance of this class. This is the essence of an immutable class.

Immutator Interface

How can we do this. Again, the simplest way is to simply add a method to return a new instance with some changes. However, due to type erasure, type `T` is treated as if it is of

type `Object`. There is not much we can do with `Object`.

Similar to how we manage to add a custom action to `Probably<T>`, we want to add a custom mutator to the class. But since the class is immutable, let us calls this concept as immutation instead with the relevant interface called `Immutator` and `Immutatorable`.

An `Immutator` is similar to the `Action<T>`. However, unlike `Action<T>` that does not return anything, we want `Immutator` to return something. Consider any method with single parameter. We can write this method signature as:

```
1   R method(P param) { .. }
```

where `R` is the return type and `P` is the type of the parameter. By invoking this method, we can change a reference type of some value of type `P` into another value of type `R`. This is the basis of our `Immutator`. Such a powerful concept, not only can we change the value, we can also change the type!

You have 2 tasks in this section:

1. Create an interface `Immutator<R,P>` with a single abstract method called `invoke` inside the file `Immutator.java`.
    - The method takes in a single parameter of generic type parameter `P`.
    - The method returns a single value of generic type `R`.
    - In other words, `Immutator<R,P>` is an abstraction of the method `R method(P param)`.
2. Create a generic class `Improbable<T>` that implements `Immutator` in the file `Immutator.java` with the method `invoke` that:
    - takes in a single parameter of type `T`.
    - returns a value of type `Probably<T>`.
    - the method simply creates a `Probably<T>` from `T` regardless of what the type `T` is (*including even when type `T` is already `Probably<T>`, but in this case what will be the actual return type?*).

Testing Immutator

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
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> new Incr().invoke(4)
$.. ==> 5
jshell> new Incr().invoke(new Incr().invoke(4))
$.. ==> 6
jshell> new Length().invoke("string")
$.. ==> 6
jshell> new Incr().invoke(new Length().invoke("string"))
$.. ==> 7
jshell> new Improbable<>().invoke(1)
$.. ==> <1>
jshell> new Improbable<String>().invoke(null)
$.. ==> <>
jshell> new Improbable<Integer>().invoke(1).transform(new Incr())
$.. ==> <2>
jshell> new Improbable<>().invoke(new Improbable<>().invoke(1))
$.. ==> <<1>>
```

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`.

Immutatorable Interface

Similar to before, we want to add a new method to `Probably<T>` but we also want to create an interface to represent all classes that can accept an `Immutator`. Let's call this method as `transform`.

Remember that `Immutator<R,P>` has a method called `invoke` that can produce a value of type `R` when given a value of type `P`. Our final aim in the end is to transform `Probably<T>` to `Probably<R>`. For that, we want an immutator that transform `T` to `R`. Note that in this case, `T` corresponds to `P` in `Immutator<R,P>`.

You have 2 tasks in this section:

1. Create an interface `Immutatorable<T>` with a single abstract method called `transform`

inside the file `Immutatorable.java`.

- The method takes in a single parameter of type `Immutator`. This `Immutator` should accept a type `T` and return a type `R`.
- The method returns a single value of type `Immutatorable<R>`.

2. Modify `Probably<T>` to implement the interface `Immutatorable`. You need to implement the method `transform` appropriately: local
   - if the value in `Probably<T>` is not `null`, we invoke `invoke` with the value and return `Probably<R>`.
   - otherwise, return `NONE`.

Testing Immutatorable

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
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).transform(new Incr())
$.. ==> <5>
jshell> Probably.just(4).transform(new Incr()).transform(new Incr())
$.. ==> <6>
jshell> Probably.just("string").transform(new Length())
$.. ==> <6>
jshell> Probably.just("string").transform(new Length()).transform(new
Incr())
$.. ==> <7>
jshell> Probably.<Integer>none().transform(new Incr())
$.. ==> <>
jshell> Probably.<String>none().transform(new Length())
$.. ==> <>
jshell> Probably.<String>just(null).transform(new Length()).transform(new
Incr())
$.. ==> <>
```

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

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

```
2  java Test4
```

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

Question

First, recap what we can do. We can create `Probably<T>` such that we can perform an action on it and mutate the value by creating a new instance each time the value changes. The next step is to ask questions regarding the value. The kind of questions we want to ask is a simple yes/no question.

Since at this point we have already created so many interfaces:

- `Action`
- `Actionable`
- `Immutator`
- `Immutatorable`

we do not want to create more interface. Instead, note that this is a special case of `Immutator`. This is simply an `Immutator` that returns a boolean.

Special Immutator

You have 2 tasks in this section:

1. Create a non-generic class `IsModEq` that implements this special `Immutator` that returns boolean values in the file `IsModEq.java`.
    - The class has a public constructor that takes in two positive integer parameters `div` and `check`.
    - The class implements the `invoke` inherited from `Immutator`.
        - The method accepts a single integer parameter `val`.
        - The method returns true if the remainder when `val` is divided by `div` is equal to `check`.
2. Add the method `check` in `Probably<T>`.
    - The method takes in a single parameter of type of the special `Immutator` introduced in this section.

- The method returns a single value of type `Probably`.
- The behaviour of the method can be captured by the table below:

value in `Probably<T>`	yes/no	result
`null`	yes (`true`)	nothing
`null`	no (`false`)	nothing
non-`null`	yes (`true`)	`this`
non-`null`	no (`false`)	nothing

Testing Question

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
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).transform(new Incr()).check(new IsModEq(3,2))
// 17 % 3 is not equal to 2
$.. ==> <17>
jshell> Probably.just("string").transform(new Length()).check(new
IsModEq(3,2))
$.. ==> <8>
jshell> Probably.<Integer>just(null).check(new IsModEq(0,2))
$.. ==> <>
```

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

```
javac -Xlint:rawtypes Test5.java
java Test5
```

There shouldn't be any compilation warning or error when you compile `Test5.java` and all tests should prints `ok`.

It is also good to check that the following code should throw `ArithmeticException` due to divide by zero. However, note that this is the similar to the last line above. The difference is that in above we have `null` so the instance of `IsModEq` is not even used.

```
Probably.<>just(2030).check(new IsModEq(0,2))
```

Applicable

In this last section, we want to show some of the power of `Probably<T>` given our changes to it. Recap that we have at least 4 interfaces.

- `Action`
- `Actionable`
- `Immutator`
- `Immutatorable`

It's Probably an Immutator

Since `Probably<T>` can store any reference type `T` and `Immutator` is a reference type, we can store `Immutator` inside `Probably<T>`. What can we do with this `Immutator` inside `Probably<T>`?

We have two cases here:

- if there is indeed an `Immutator` (*i.e., it just*[2] *some immutator*), then we can use the immutator to mutate the value and the result of this really depends on the value.
- if there is no `Immutator` (*i.e., it is nothing*[1]), then we simply return nothing.

Basically:

> nothing in, nothing out.

You have 2 tasks in this section:

1. Create an interface `Applicable<T>` with a single abstract method `apply` inside the file `Applicable.java`.
    - The method takes in a single parameter of type `Immutator` inside `Probably<T>`. This `Immutator` should accept a generic type `T` and return a generic type `R`.
    - The method returns a single value of type `Probably<R>`.
    - The method performs the operation described above.
2. Modify `Probably<T>` to implement the interface `Applicable`. You need to implement the method `apply` appropriately:
    - if the value in `Probably<T>` is not `null` and the value `inside the` Probably `parameter is not null, we invoke the` Immutator `with the value and return` Probably`.
    - otherwise, return `NONE`.

Testing Applicable

We recommend to first design your solution, figure out the type needed which may or may not involve wildcards. Afterwards, you can test your solution with the following code. You should also *test with your own test cases* as the given test case may not be complete.

```
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 Test6.java
java Test6
```

There shouldn't be any compilation warning or error when you compile `Test6.java` and all tests should prints `ok`.

Hints

- This lab is more about the type rather than the code.
- You should think about the types that are required by each class and methods. In particular, you should think carefully about the generic type and wildcards if needed.

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 and they are not to be submitted. You can edit them to add your own test cases.

Lastly, the file `Lab4.java` is given for you and it should not be modified except for correcting any style problems detected by our style checker. This file will be the main entry point for our testing on CodeCrunch.

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.

---

1. We will refer to this as none, nothing, or `NONE`. ↩↩↩↩
2. We will refer to this as just, some, or something. ↩↩↩
3. Here, a single type `T` includes the possibility that it uses wildcards involving `T`. ↩