Ex 5: Call Me Maybe

Basic Information

- **Deadline:** 15 October 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★★★★

Prerequisite

- Caught up to Unit 29 of Lecture Notes.
- Familiar with CS2030S Java style guide.

Goal

This is a continuation of Programming Exercise 4. In Exercise 4, we have constructed a generic class `Some<T>`, which is a container for an item of type `T`. Beyond being an exercise for teaching about generics, `Some<T>` is not a very useful type. In Programming Exercises 5 and 6, we are going to modify `Some<T>` into two more useful and general classes. We are going to build our own Java packages using these useful classes.

Java Package

Java package mechanism allows us to group relevant classes and interfaces under a namespace. You have seen two packages so far: `java.util`, where we import `List` and `Arrays` from as well as `java.lang` where we import the `Math` class. These are provided by Java as standard libraries. We can also create our package and put the classes and interfaces into the same package. We (*and the clients*) can then import and use the classes and interfaces that we provide.

Java package provides a higher layer of abstraction barrier. We can designate a class to be used outside a package by prefixing the keyword `class` with the access modifier `public`. This is another advantage of public class. The previous advantage of a public class is that it must have the same name as the file. So java compiler knows where to search. We can further fine-tune which fields and methods are accessible from other classes in the same package using the `protected` access modifier.

You can read more about java packages and the `protected` modifier yourself through Oracle's Java tutorial.

As a summary, the access levels are as follows.

Modifier	**Access from same class**	**Access from same package (*or same directory*)**	**Access from subclass (*even in other directory*)**	**Access from other class (*even in other directory*)**
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
no modifier	✓	✓	✗	✗
`private`	✓	✗	✗	✗

We will create a package named `cs2030s.fp` to be used for this and the next few exercises. First, we need to add the line:

```
package cs2030s.fp;
```

on top of every `.java` file that we would like to include in the package.

Second, the package name is typically written in a hierarchical manner using the "." notation. The name also indicates the location of the `.java` files and the `.class` files. For this reason, you can no longer store the `.java` files under `ex5-username` directly. Instead, you should put them in a subdirectory called `cs2030s/fp` under `ex5-username`. To start, our `cs2030s.fp` package will contain the two interfaces `Transformer` and `BooleanCondition` that you have written in Programming Exercise 4.

If you have not made `Transformer` a `public` class, you should do it now.

```
public class Transfomer<T, R> {
    :
}
```

Finally, to compile your code, under your `ex5-username` directory, run:

```
javac -Xlint:unchecked -Xlint:rawtypes cs2030s/fp/*.java *.java
```

If you have set up everything correctly, you should be able to run the following in JShell from your `ex5-username` directory:

```
jshell> import cs2030s.fp.Transformer;
```

Tasks

Eventually, we will be creating a static nested class `Some<T>` that is nested inside the `Maybe<T>` class. `Maybe<T>` encapsulates the possibility that a value is missing. Our `Maybe<T>` is an *option* type, a common abstraction in programming languages (`java.util.Optional` in *Java*, `option` in *Scala*, `Maybe` in *Haskell*, `Nullable<T>` in *C#*, *etc*) that is a wrapper around a value that might be missing. In other words, it represents either some value, or none.

Task 1: More Interfaces

Now, we are going to add three more interfaces into our package:

- `Producer<T>` is an interface with a single `produce` method that takes in no parameter and returns a value of type `T`.
- `Consumer<T>` is an interface with a single `consume` method that takes in a parameter of type `T` and returns *nothing*.
- `BooleanCondition<T>` is an interface with a single `test` method that takes in a parameter of type `T` and returns a primitive `boolean` value.

If you have set up everything correctly, you should be able to run the following in JShell without errors (*remember to always compile your code first!*).

Sample Usage

```
jshell> import cs2030s.fp.Producer;
jshell> import cs2030s.fp.Consumer;
jshell> import cs2030s.fp.BooleanCondition;

jshell> Producer<String> p;
jshell> p = new Producer<>() {
   ...>   public String produce() { return ""; }
   ...> }
jshell> Consumer<Boolean> c;
jshell> c = new Consumer<>() {
   ...>   public void consume(Boolean b) { }
   ...> }
jshell> BooleanCondition<Integer> b;
jshell> b = new BooleanCondition<>() {
   ...>   public boolean test(Integer x) { return x > 0; }
   ...> }
```

Task 2: Some Packaging

There is minimal amount of code to be added here. We will be mainly be doing a rearrangement of code.

1. Copy your implementation of `Some.java` into `lab5-username/cs2030s/fp` directory if you have not done so.
2. Add `package cs2030s.fp;` as the first line on `Some.java`.
3. Rename `Some.java` into `Maybe.java`. This entails some other changes too:
   - Rename all occurrences of `Some` into `Maybe` including `private` constructor and the return type.
   - Do **NOT** change the name of the factory method `some`.

   If you have done this correctly, your directory structure should look something like the following:

   ```
   labX-username/
   ├─ cs2030s/
   │  └─ fp/
   │     ├─ BooleanCondition.java
   │     ├─ Consumer.java
   │     ├─ Maybe.java
   │     ├─ Producer.java
   │     └─ Transformer.java
   ├─ CS2030STest.java
   ├─ Test1.java
   ├─ Test2.java
   ├─   :
   └─  ...
   ```

4. Change `public class Some<T>` to `private static final class Some<T> extends Maybe<T>`.
   - Then wrap it inside the outer class `public abstract class Maybe<T>`.
   - Move `public static <T> Maybe<T> some(T value)` from `Some<T>` to `Maybe<T>`.

Checkpoint

At this point, `Some<T>` is a static nested class inside `Maybe<T>`. But codes from outside of the package cannot see `Some<T>` and only `Maybe<T>`. Since `Maybe<T>` does not have any known method, we need to add abstract methods.

5. Add abstract method descriptor that appears in `Some<T>` to `Maybe<T>` unless these method descriptor already available in `Object`.
6. Finally, we need to handle `null` values in `Some<T>` and `Maybe<T>`.
   - `public static <T> Maybe<T> some(T value)` accepts `null` and simply store the `null` value in the field.
   - Two `Some<T>` instances are equal (*as decided by their respective* `equals(Object)` *method*) if either one (*or both*) of the following condition is true.
     - The content are both `null`.
     - The content are equal as decided by their respective `equals(Object)` method.

map

There is no need to specially handle `null` in `map`. In particular, if the `Transfomer` in `map` returns `null`, we will simply use the `null` value.

Sample Usage

```
jshell> import cs2030s.fp.Maybe
jshell> import cs2030s.fp.Transformer

jshell> Maybe<Object> m = new Maybe<>()
|  Error:
|  cs2030s.fp.Maybe is abstract; cannot be instantiated
|  Maybe<Object> m = new Maybe<>();
|                    ^-----------^
jshell> Maybe.Some<Object> m
|  Error:
|  cs2030s.fp.Maybe.Some has private access in cs2030s.fp.Maybe
|  Maybe.Some<Object> m;
|  ^--------^
jshell> Maybe.some(0).get()
|  Error:
|  cannot find symbol
|    symbol:   method get()
|  Maybe.some(0).get()
|  ^---------------^

jshell> Maybe.some(null)
```

```
$.. ==> [null]
jshell> Maybe.some(4)
$.. ==> [4]
jshell> Maybe.some("day").equals(Maybe.some("day"))
$.. ==> true
jshell> Maybe.some(null).equals(Maybe.some("day"))
$.. ==> false
jshell> Maybe.some(null).equals(Maybe.some(null))
$.. ==> true
jshell> Maybe.some(null).equals(null)
$.. ==> false

jshell> class AddOne implements Transformer<Integer, Integer> {
   ...>   @Override
   ...>   public Integer transform(Integer t) {
   ...>     return t + 1;
   ...>   }
   ...> }
jshell> class StrLen implements Transformer<String, Integer> {
   ...>   @Override
   ...>   public Integer transform(String t) {
   ...>     return t.length();
   ...>   }
   ...> }
jshell> class Destroyer implements Transformer<Integer, Object> {
   ...>   @Override
   ...>   public Object transform(Integer t) {
   ...>     return null;
   ...>   }
   ...> }
jshell> AddOne fn1 = new AddOne();
jshell> StrLen fn2 = new StrLen();
jshell> Destroyer fn3 = new Destroyer();

jshell> Maybe.some(4).<Integer>map(fn1)
$.. ==> [5]
jshell> Maybe.some(5).map(fn1)
$.. ==> [6]
jshell> Maybe.some("CS2030S").map(fn2)
$.. ==> [7]
jshell> Maybe.some("CS2030S").map(fn2).map(fn1)
$.. ==> [8]

jshell> Maybe<Number> six = Maybe.some(4).map(fn1).map(fn1)
six ==> [6]

jshell> Maybe.some(4).map(fn3)
$.. ==> [null]
jshell> Maybe.some(4).map(fn3) == Maybe.some(null)
$.. ==> false
jshell> Maybe.some(4).map(fn3).equals(Maybe.some(null))
$.. ==> true
```

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 ex5_style.xml cs2030s/fp/*.java
```

Task 3: None Other than You

Now we want to add `None<T>` class as another private static nested class inside `Maybe<T>`. This class is also a subtype of `Maybe<T>`. The types `None<T>` is an internal implementation details of `Maybe<T>` and must not be used directly by the client. Hence, it must be declared private. Here is the requirement for `None<T>`.

- `None<T>` is a generic private inner class that inherits from `Maybe<T>`.
- `None<T>` has no instance field.
- `None<T>` has private constructor that takes in no argument.
- `None<T>` overrides the `equals(Object)` method.
  - Any instance of `None<T>` is equal to any other instance of `None<T>`.
  - Note that `Some<T>` should never be equal to `None<T>`.
- `None<T>` overrides the `toString()` method.
  - It simply prints `[]`.
- `None<T>` overrides the `map` method from `Maybe<T>`.
  - This simply returns itself.
- `None<T>` (and by extension `Maybe<T>`) must be immutable up to `T`.
  - But you do not have to make the class a `final` class.

Additionally, we need to add the following factory methods in `Maybe<T>`.

- Add the factory method `none()` that returns an instance of `None<T>`.
  - There should only be **ONE** instance of `None<T>` such that multiple calls to `none()` should return the same instance.
  - You may add `@SuppressWarnings` here with explanation on why it is safe.
- Add the factory method `of` that returns:
  - an instance of `None<T>` if the input is `null`.
  - an instance of `Some<T>` if the input is not `null`.

Sample Usage

```
jshell> import cs2030s.fp.Maybe
jshell> import cs2030s.fp.Transformer

jshell> Maybe.None m;
|  Error:
|  cs2030s.fp.Maybe.None has private access in cs2030s.fp.Maybe
|  Maybe.None m;
|  ^--------^
jshell> Maybe.none().get()
|  Error:
|  cannot find symbol
|    symbol:   method get()
|  Maybe.none().get()
|  ^--------------^

jshell> Maybe.none()
$.. ==> []

jshell> Maybe.none() == Maybe.none()
$.. ==> true
jshell> Maybe.none().equals(Maybe.none())
$.. ==> true
jshell> Maybe.none().equals(Maybe.some("day"))
$.. ==> false
jshell> Maybe.none().equals(Maybe.some(null))
$.. ==> false
jshell> Maybe.some(null).equals(Maybe.none())
$.. ==> false

jshell> Maybe.of(null).equals(Maybe.none())
$.. ==> true
jshell> Maybe.of(null) == Maybe.none()
$.. ==> true
jshell> Maybe.of(null).equals(Maybe.some(null))
$.. ==> false
jshell> Maybe.of(4).equals(Maybe.none())
$.. ==> false
jshell> Maybe.of(4).equals(Maybe.some(4))
$.. ==> true

jshell> Transformer<Integer, Integer> incr = new Transformer<>() {
   ...>   @Override
   ...>   public Integer transform(Integer x) {
   ...>     return x + 1;
   ...>   }
   ...> };
jshell> Maybe.<Integer>none().map(incr)
$.. ==> []
jshell> Maybe.<Integer>some(null).map(incr)
|  Exception java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "<parameter1>" is null
|        at 1.transform (#15:4)
|        at 1.transform (#15:1)
|        at Maybe$Some.map (Maybe.java:62)
|        at (#17:1)
jshell> Maybe.<Integer>some(1).map(incr)
$.. ==> [2]
```

```

jshell> import java.util.Map;
jshell> Map<String, Integer> map = Map.of("one", 1, "two", 2);
jshell> Transformer<String, Integer> wordToInt = new Transformer<>() {
   ...>   @Override
   ...>   public Integer transform(String x) {
   ...>     return map.get(x);
   ...>   }
   ...> };
jshell> Maybe.<String>none().map(wordToInt)
$.. ==> []
jshell> Maybe.<String>some("").map(wordToInt)
$.. ==> [null]
jshell> Maybe.<String>some("one").map(wordToInt)
$.. ==> [1]

jshell> Transformer<String, Maybe<Integer>> wordToMaybeInt = new
Transformer<>() {
   ...>   @Override
   ...>   public Maybe<Integer> transform(String x) {
   ...>     return Maybe.of(map.get(x));
   ...>   }
   ...> };
jshell> Maybe.<String>none().map(wordToMaybeInt)
$.. ==> []
jshell> Maybe.<String>some("").map(wordToMaybeInt)
$.. ==> [[]]
jshell> Maybe.<String>some("one").map(wordToMaybeInt)
$.. ==> [[1]]
```

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 ex5_style.xml cs2030s/fp/*.java
```

Task 4: Filtering

We now add the method `filter` to `Maybe<T>`.

- Add the abstract method `filter` to `Maybe<T>` that takes in a `BooleanCondition<..>` (*type parameter is omitted*) as a parameter.
- Override the method `filter` in `Some<T>` as follows.
  - If the value is `null` or it failed the test (*i.e., the call to* `test` *returns* `false`), return `None<T>`.
  - Otherwise, leaves the `Maybe<T>` untouched and returns the `Maybe<T>` as it is.

- Override the method `filter` in `None<T>` as follows.
    - Always returns a `None<T>`.

Sample Usage

```
jshell> import cs2030s.fp.BooleanCondition
jshell> import cs2030s.fp.Maybe

jshell> BooleanCondition<Number> isEven = new BooleanCondition<>() {
   ...>   public boolean test(Number x) {
   ...>     return x.shortValue() % 2 == 0;
   ...>   }
   ...> };

jshell> Maybe.<Integer>none().filter(isEven)
$.. ==> []
jshell> Maybe.<Integer>some(null).filter(isEven)
$.. ==> []
jshell> Maybe.<Integer>some(1).filter(isEven)
$.. ==> []
jshell> Maybe.<Integer>some(2).filter(isEven)
$.. ==> [2]
```

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 ex5_style.xml cs2030s/fp/*.java
```

Task 5: flatMap

Consider a `Transformer` that might return a `Maybe<T>` itself (*as* `wordToMaybeInt` *above*). Using `map` on such a `Transformer` would lead to a value wrapped around a `Maybe` twice. We want to add a method that is not doing this.

- Add the abstract method `flatMap` to `Maybe<T>` that takes in a `Transfomer<..>` (*type parameter is omitted*) as a parameter.
- Override the method `flatMap` in `Some<T>` as follows.
    - The `Transformer` object transforms the value of type `T` in `Maybe<T>` into a value of type `Maybe<U>`, for some type `U`.
    - The method `flatMap`, however, returns a value of type `Maybe<U>` (*instead of* `Maybe<Maybe<U>>` *as in the case of* `map`).

    - You may add `@SuppressWarnings` here with explanation on why it is safe.
- Override the method `flatMap` in `None<T>` as follows.
    - Always returns a `None<T>`.

Sample Usage

```
jshell> import cs2030s.fp.BooleanCondition
jshell> import cs2030s.fp.Maybe
jshell> import cs2030s.fp.Transformer

jshell> Map<String, Integer> map = Map.of("one", 1, "two", 2);
jshell> Transformer<String, Maybe<Integer>> wordToMaybeInt = new
Transformer<>() {
   ...>   @Override
   ...>   public Maybe<Integer> transform(String x) {
   ...>     return Maybe.of(map.get(x));
   ...>   }
   ...> };

jshell> Maybe.<String>none().flatMap(wordToMaybeInt)
$.. ==> []
jshell> Maybe.<String>some("").flatMap(wordToMaybeInt)
$.. ==> []
jshell> Maybe.<String>some("one").flatMap(wordToMaybeInt)
$.. ==> [1]
```

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 ex5_style.xml cs2030s/fp/*.java
```

Task 6: Back to T

Since `Maybe<T>` is an abstraction for a possibly missing value (*of type* `T`), it would be useful to provide methods that decide what to do if the value is missing.

- Add the abstract method `orElse` in `Maybe<T>` that takes in a `Producer<..>` (*type parameter is omitted*) as the parameter.
    - Override the method `orElse` in `Some<T>` to return the value inside.
    - Override the method `orElse` in `None<T>` to return the subtype of `T` produced by the producer.

- Add the abstract method `ifPresent` in `Maybe<T>` that takes in a `Consumer<..>` (*type parameter is omitted*) as the parameter.
    - Override the method `ifPresent` in `Some<T>` such that the given consumer consumes the value inside.
    - Override the method `ifPresent` in `None<T>` that does *nothing*.

Sample Usage

```
jshell> import cs2030s.fp.Consumer;
jshell> import cs2030s.fp.Maybe;
jshell> import cs2030s.fp.Producer;
jshell> import java.util.ArrayList;
jshell> import java.util.List;

jshell> Producer<Double> zero = new Producer<>() {
   ...>   @Override
   ...>   public Double produce() {
   ...>     return 0.0;
   ...>   }
   ...> };

jshell> Maybe.<Number>none().orElse(zero)
$.. ==> 0.0
jshell> Maybe.<Number>some(1).orElse(zero)
$.. ==> 1

jshell> List<Object> list = new ArrayList<>();
jshell> Consumer<Object> addToList = new Consumer<>() {
   ...>   @Override
   ...>   public void consume(Object o) {
   ...>     list.add(o);
   ...>   }
   ...> };

jshell> Maybe.<Number>none().ifPresent(addToList)

jshell> list.size()
$.. ==> 0
jshell> list
list ==> []
jshell> Maybe.<Number>some(1).ifPresent(addToList)
jshell> list.get(0)
$.. ==> 1
jshell> list
list ==> [1]
```

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.

Test5.java

```
javac -Xlint:rawtypes -Xlint:unchecked Test5.java
java Test5
$ java -jar ~cs2030s/bin/checkstyle.jar -c ex5_style.xml cs2030s/fp/*.java
```

Skeleton for Programming Exercise 5

You need to copy the files `Some.java` and `Transformer.java` from `ex4-username` to `ex5-username`. Some files (*e.g.*, `Test1.java`, `Test2.java`, `CS2030STest.java`, *etc*) are provided for testing. Do not copy these from `ex4-username`. 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.

To check for style, we may need two commands as there are two directories of interest.

Style Check

```
java -jar ~cs2030s/bin/checkstyle.jar -c ex5_style.xml *.java
java -jar ~cs2030s/bin/checkstyle.jar -c ex5_style.xml cs2030s/fp/*.java
```

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 5. It is, however, always a good practice to include comments to help readers understand your code.