Ex 4: Some Body Once Told Me

Basic Information

- **Deadline:** 8 October 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★

Prerequisite

- Caught up to Unit 27 of Lecture Notes.
- Familiar with CS2030S Java style guide.

Goal

The goal of this exercise is to build a simple *immutable* generic container.

Tasks

Our container is storing *some* value of reference type. We shall call this container `Some`. It is a generic wrapper class with a single type parameter `T` (*i.e.*, `Some<T>`). At the beginning, this will not be a useful container, but do not worry, we will slowly add more functionalities.

Task 1: A Simple Container

Implement a generic class `Some<T>` that

- contains a field of type `T` that is declared `private` and `final` to store the content.
- overrides the `boolean equals(Object)` method from `Object` to compare if two containers are the same in the way described below.
  - Two containers are the same if the contents are equal to each other, as decided by their respective `equals` method.

- overrides the `String toString()` method from `Object` so it returns the string representation of its content, between `[` and `]`.
- provides a class method called `some` that returns a container with the given object.
  - You may assume that no `null` value will be given for now.

Factory Method

The method `some` is 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. The `some` method does not currently reuse instances but this will be rectified in subsequent exercise.

With the availability of the of factory method, `Some<T>` should keep the constructor `private`.

Sample Usage

```
jshell> Some.<Integer>some(4)
$.. ==> [4]
jshell> Some.some(5) // type inference!
$.. ==> [5]

jshell> Some.some(4).equals(Some.some(4))
$.. ==> true
jshell> Some.some(4).equals(4)
$.. ==> false
jshell> Some.some(Some.some(0)).equals(Some.some(Some.some(0)))
$.. ==> true
jshell> Some.some(Some.some(0)).equals(Some.some(0))
$.. ==> false
jshell> Some.some(0).equals(Some.some(Some.some(0)))
$.. ==> false

jshell> Some.some("body once told me")
$.. ==> [body once told me]
jshell> Some.some("4").equals(Some.some(4))
$.. ==> false
```

You can test your `Some<T>` more comprehensively by running:

Test1.java

```
username@pe111:~/ex4-username$ javac -Xlint:rawtypes -Xlint:unchecked
Test1.java
```

```
    username@pe111:~/ex4-username$ java Test1
```

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

Task 2: Transformation

Now, we are going to write an interface (*along with its implementations*) and a method in `Some` that allows a container to be transformed into another container, possibly containing a different type.

Step 1: Transformer Interface

First, create an interface called `Transformer<T, U>` with an abstract method called `transform` that takes in an argument of generic type `T` and returns a value of generic type `U`.

Part 2: Mapping Method

Second, write a method called `map` in the class `Some` that takes in a `Transformer`, and use the given `Transformer` to transform the container (*and the value inside*) into another container of type `Some<U>`. You should leave the original container unchanged.

Sample Usage

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

jshell> Some.some(4).<Integer>map(fn1)
$.. ==> [5]
jshell> Some.some(5).map(fn1)
$.. ==> [6]

jshell> Some<Number> six = Some.some(4).map(fn1).map(fn1)
six ==> [6]
jshell> six.map(fn2)
|  Error: ...
|  six.map(fn2)
|  ^-----^

jshell> Some<String> mod = Some.some("CS2030S")
mod ==> [CS2030S]
jshell> mod.map(fn2)
$.. ==> [7]
jshell> mod
mod ==> [CS2030S]
jshell> mod.map(fn2).map(fn1)
$.. ==> [8]
```

You can test your `Some<T>` more comprehensively by running:

Test2.java

```
username@pe111:~/ex4-username$ javac -Xlint:rawtypes -Xlint:unchecked Test2.java
username@pe111:~/ex4-username$ java Test2
```

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

Part 3: Flexible Method

Make sure that you make the method signature as *flexible* as possible. Follow the PECS principle after you determine which type (*i.e.*, `T` *or* `U`) acts as producer or consumer (*or both?*).

Flexible Usage

```
jshell> /open A.java
jshell> /open B.java
jshell> /open C.java

jshell> class AtoC implements Transformer<A, C> {
   ...>   @Override
   ...>   public C transform(A arg) {
   ...>     return new C(arg.get());
   ...>   }
   ...> }
jshell> class BtoB implements Transformer<B, B> {
   ...>   @Override
   ...>   public B transform(B arg) {
   ...>     return new B(arg.get());
   ...>   }
   ...> }
jshell> class CtoA implements Transformer<C, A> {
   ...>   @Override
   ...>   public A transform(C arg) {
   ...>     return new A(arg.get());
   ...>   }
   ...> }

```

```
jshell> Some<A> someA = Some.some(new A(1))
jshell> Some<B> someB = Some.some(new B(2))
jshell> Some<C> someC = Some.some(new C(3))
jshell> AtoC fn1 = new AtoC()
jshell> BtoB fn2 = new BtoB()
jshell> CtoA fn3 = new CtoA()

jshell> someA.map(fn1)
$.. ==> [C:1]
jshell> someA.map(fn2)
|  Error: ...
|  someA.map(fn2)
|  ^-------^
jshell> someA.map(fn3)
|  Error: ...
|  someA.map(fn3)
|  ^-------^

jshell> someB.map(fn1)
$.. ==> [C:2]
jshell> someB.map(fn2)
$.. ==> [B:2]
jshell> someB.map(fn3)
|  Error: ...
|  someB.map(fn3)
|  ^-------^

jshell> someC.map(fn1)
$.. ==> [C:3]
jshell> someC.map(fn2)
$.. ==> [B:3]
jshell> someC.map(fn3)
$.. ==> [A:3]
```

You can test your `Some<T>` more comprehensively by running:

Test3.java

```
username@pe111:~/ex4-username$ javac -Xlint:rawtypes -Xlint:unchecked Test3.java
username@pe111:~/ex4-username$ java Test3
```

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

Task 3: Jack in the Box

The `Transformer` interface allows us to transform the content of the container from one type into any other type, including a `Some<T>`! You have seen examples above where we have a container inside a container: `Some.some(Some.some(0))`.

Now, implement your own `Transformer` in a class called `JackInTheBox<T>` to transform an item into a `Some` containing the item. The corresponding type `T` is transformed into `Some<T>`. This transformer, when invoked with `map`, results in a new `Some` within the `Some`.

Sample Usage

```
jshell> Some.some(4).map(new JackInTheBox<>())
$.. ==> [[4]]
jshell> Some.some(Some.some(5)).map(new JackInTheBox<>())
$.. ==> [[[5]]]
```

You can test your `JackInTheBox<T>` more comprehensively by running:

Test4.java

```
username@pe111:~/ex4-username$ javac -Xlint:rawtypes -Xlint:unchecked Test4.java
username@pe111:~/ex4-username$ java Test4
```

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

Skeleton for Programming Exercise 4

A set of empty files has been given to you. You should **ONLY** edit these files. You must **NOT** add any additional files.

Do NOT Add

Only edit the given files, do not add any additional files.

Some files (*e.g.*, `Test1.java`, `A.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.

Following CS2030S Style Guide

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

To check for style,

Style Check

```
username@pe111:~/ex4-username$ java -jar ~cs2030s/bin/checkstyle.jar -c ex4_style.xml *.java
```

Suppressing Warnings

If you design your code correctly, you do not need any `@SuppressWarnings`. If you have any, you may want to check your design again.

Further Deductions

Additional deductions may be given for other issues or errors in your code. 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 2. It is, however, always a good practice to include comments to help readers understand your code.