Lab 5: Actually

- Deadline: 11 October, 2022, Tuesday, 23:59, SST
- Mark: 3%

Prerequisite:

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

This is a follow-up from Lab 4. In Lab 4, we have constructed a generic class `Probably<T>`, which is a container for an item of type `T`. Beyond being an exercise for teaching about generics, `Probably<T>` is not a very useful type. We also have several interfaces. An important interface for us is the `Immutator<R,P>` interface that abstract away the behaviour of a method `R invoke(P p)`. In Lab 5 and 6, we are going to modify `Probably<T>` as well as add useful `Immutator<R,P>`. 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`, `Arrays`, and `ArrayList` from as well as `java.lang` where we import the `Math` class from[1]. 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`. 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.

We will create a package named `cs2030s.fp` to be used for this and the next few labs.

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.

The package name is typically written in a hierarchical manner using the "." notations. 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 `labX-username` directly. Instead, you should put them in a subdirectory called `cs2030s/fp` under `labX-username`.

To start, our `cs2030s.fp` package will contain the following interfaces from Lab 4: `Action`, `Actionable`, `Immutator`, `Immutatorable`. For now, we ignore `Applicable` and `Probably`.

If you have set up everything correctly, you should be able to run the following in `jshell` (*remember to always compile your code first*!) from your `labX-username` directory:

```
jshell> import cs2030s.fp.Action;
jshell> import cs2030s.fp.Immutator;
```

without error.

More Interfaces

Now, we are going to add one more interface into our package:

- `Constant<T>` is an interface with a single `init` method that takes in no parameter and returns a value of type `T`.

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*!)

```
jshell> import cs2030s.fp.Constant;
jshell> import cs2030s.fp.Action;
jshell> Constant<String> emp;
jshell> emp = new Constant<>() {
   ...>   public String init() { return ""; }
   ...> }
jshell> Action<Boolean> pass;
jshell> pass = new Action<>() {
   ...>   public void call(Boolean b) { }
   ...> }
```

Actually

Now, we are going implement a type called `Actually<T>` in the `cs2030s.fp` package. Our `Actually<T>` is also called a *result type*, a common abstraction in programming languages (*e.g.*, `Either` in Haskell and Scala[2], `enum Result` in Rust, and simply a try-catch syntax in Java) that is a wrapper around the idea that we think maybe the function call is successful but actually a failure (*i.e.*, *throws an exception*). In other words, it represents either a successful computation, or a failure. Here, we represent the failure as an `Exception`.

Inner Classes and Factory Methods

Write an abstract class called `Actually<T>` in a file called `Actually.java`. Make sure this class is a public class. This class should have two concrete, static, nested classes, named `Success<T>` and `Failure`.

Later on, this class need to implement `Immutatorable<T>`. But for now, we can keep it as it is first. The sample run below is before `Actually<T>` implements `Immutatorable<T>`.

```
jshell> import cs2030s.fp.Actually

jshell> Actually<Object> a = new Actually<>()
|  Error:
|  cs2030s.fp.Actually is abstract; cannot be instantiated
|  Actually<Object> a = new Actually<>();
|
```

- Both `Success<T>` and `Failure` are declared inside the class `Actually<T>`.
- Both `Success<T>` and `Failure` inherits from `Actually<T>`. Note that `Failure` is *not* a generic class so you need to specify `Object` as the type argument to `Actually<T>`.
- `Success<T>` and `Failure` must be immutable.
- The types `Success<T>` and `Failure` are *internal* implementation details of `Actually<T>` and must not be used directly. For instance, clients must not be able to declare a variable of type `Actually.Success<T>`.

```
jshell> Actually.Success<Object> s
|  Error:
|  cs2030s.fp.Actually.Success is not public in cs2030s.fp.Actually;
cannot be accessed from outside package
|  Actually.Success<Object> s;
|  ^--------------^

jshell> Actually.Failure f
|  Error:
|  cs2030s.fp.Actually.Failure is not public in cs2030s.fp.Actually;
cannot be accessed from outside package
|  Actually.Failure f;
|  ^--------------^
```

`Actually<T>` has two static factory methods:

- `ok(T res)` returns an instance of `Success<T>`. The method takes in a value `res` and returns an instance of `Success<T>` wrapped around `res`. Here, `res` may be `null` and that's fine, we also had a method that returns `null` in Lab 1 - 3.
- `err(Exception exception)` returns an instance of `Failure`. The method takes in a value `exc` and returns an instance of `Failure` wrapped around `exc`. Here, `exc` will never be `null` as a failure is always accompanied with an exception.

Implement a `Success::toString` method that always returns the string representation of the content between `<` and `>` (*i.e., similar to* `Probably<T>`). Additionally, implement a `Failure::toString` method that always returns the exception class name (*i.e.,* `getClass`) between `[` and `]` followed by a whitespace and lastly followed by the message in the exception (*i.e.,* `getMessage()`).

Here are some examples of how the factory methods might be used (*remember to always compile your code first!*).

```
jshell> Actually<String> success = Actually.ok("success")
success ==> <success>

jshell> Actually<Integer> none = Actually.ok(null)
none ==> <null>

jshell> Actually<Integer> four = Actually.ok(4)
four ==> <4>

jshell> Actually<Object> div0 = Actually.err(new
ArithmeticException("Divide by 0"))
div0 ==> [java.lang.ArithmeticException] Divide by 0
```

Implement the equal method such that two `Success<T>` instances are equals if their contents are equal (*is it similar to* `Probably<T>`?) and two `Failure` instances are equals if

they have the equal messages (*i.e.*, `getMessage()`).

```
jshell> Actually.err(new
ArithmeticException("Err")).equals(Actually.err(new Exception("Err")))
$.. ==> true
jshell> Actually.err(new
ArithmeticException("Err")).equals(Actually.err(new Exception("Error")))
$.. ==> false
jshell> Actually.err(new
ArithmeticException("Err")).equals(Actually.ok(null))
$.. ==> false
jshell> Actually.err(new
ArithmeticException(null)).equals(Actually.ok(null))
$.. ==> false
jshell> Actually.err(new
ArithmeticException("Err")).equals(Actually.ok("Err"))
$.. ==> false

jshell> Actually.ok("Err").equals(Actually.ok("Err"))
$.. ==> true
jshell> Actually.ok("Err").equals(Actually.err(new Exception("Err")))
$.. ==> false
jshell> Actually.ok("Err").equals("Err")
$.. ==> false

jshell> Actually.ok(null).equals(Actually.ok("Err"))
$.. ==> false
jshell> Actually.ok(null).equals(Actually.ok(null))
$.. ==> false
jshell> Actually.ok(null).equals("Err")
$.. ==> false
jshell> Actually.ok(null).equals(null)
$.. ==> false
```

You can test your code by running the `Test1.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes Test1.java
$ java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

So far, what we have done is to create `Success<T>` and `Failure` with their methods. What has this got to do with `Actually<T>`? Well, both `Success<T>` and `Failure` inherits from `Actually<T>`. Also, `Actually<T>` is an abstract class. So, we can actually add abstract methods into `Actually<T>` to ensure that whether the run-time type is `Success<T>` or `Failure`, we can invoke the method. Of course the implementation for any abstract methods we add into `Actually<T>` has to be in `Success<T>` and `Failure`.

Since `Actually<T>` is an abstraction of the result of a computation --*which may be present or actually an exception*-- we want to be able to get the result in a safe way. That is going

to be your first set of tasks.

Secondly, you want to be able to perform computations on the value. This can be done using our `Immutator` and `Action`. That will be your second set of tasks, make `Actually<T>` implements `Immutatorable<T>` and `Actionable<T>`.

Lastly, since our `Immutator` is quite limited, we want to be able to create more complex operations from simpler operations. To do that, we need to be able to chain functions together[3]. We call this a `Transformer`. Another special kind of `Immutator` is an `Immutator` that automatically wrap the result in `Actually<T>`. We call this a constructor.

In all cases, remember to apply PECS in your method signature so that all the methods are as flexible as possible in the type that it accepts. As usual, the test cases given may not be complete and there may be other test cases used.

Safe Result

Here we try to get the result safely. The first method `unwrap` is the only unsafe method as it is guaranteed to throws exception when we try to `Unwrap` a `Failure`.

- Add a `public` abstract method into `Actually<T>` called `unwrap` that accepts no parameter with return type `T`.
  - Implement `unwrap` in `Success<T>` such that it returns the value contained inside.
  - Implement `unwrap` in `Failure` such that it throws the stored exception.
- Add a `public` abstract method into `Actually<T>` called `except` that accepts a single parameter of type `Constant` with return type `T`.
  - Implement `except` in `Success<T>` such that it returns the value contained inside.
  - Implement `except` in `Failure` such that it returns a value that is a subtype of `T` from the result of invoking `init` from the `Constant`.
- Add a `public` abstract method into `Actually<T>` called `finish` that accepts a single parameter of type `Action` and does not return anything.
  - Implement `finish` in `Success<T>` such that it invokes `call` from `Action` using the value contained inside.
  - Implement `finish` in `Failure` such that it does nothing.
- Add a `public` abstract method into `Actually<T>` called `unless` that accepts a single parameter of type that is a subtype of `T` and returns a value that is a subtype of `T`.
  - Implement `unless` in `Success<T>` such that it returns the value contained inside.
  - Implement `unless` in `Failure` such that it returns a given value that is a subtype

of `T`.

```
jshell> Actually.<Number>ok(0).unwrap()
$.. ==> 0
jshell> Actually.<Integer>ok(9).finish(print)
$.. ==> 9
jshell> Actually.<Integer>err(new Exception("Err")).finish(print)
$.. ==>
jshell> Actually.<Number>ok(9).except(zero)
$.. ==> 9
jshell> Actually.<Number>err(new ArithmeticException("div by
0")).except(zero)
$.. ==> 0
jshell> Actually.<Number>err(new ArithmeticException("div by
0")).unless(4)
$.. ==> 4
jshell> Actually.<Number>ok(0).unless(4)
$.. ==> 0
```

You can test your code by running the `Test2.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes Test2.java
$ java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Immutatorable and Actionable

- Modify `Actually<T>` to implement `Immutatorable<T>`.
  - Calling `transform` on `Failure` should propagate the exception contained as a new `Failure`.
  - Calling `transform` on `Success<T>` should attempt to return a new `Success<T>` with the value inside transformed by the `Immutator` instance. However, if an exception occurs, then a new `Failure` wrapping the exception is returned instead.
- Modify `Actually<T>` to implement `Actionable<T>`.
  - Calling `act` on `Failure` does nothing.
  - Calling `act` on `Success<T>` should invoke the `call` from the `Action` using the value inside.

```
jshell> Actually.<Integer>ok(0).transform(inc)
$.. ==> <1>
jshell> Actually.<Integer>ok(0).transform(inv)
$.. ==> [java.lang.ArithmeticException] / by zero
```

```
jshell> Actually.ok(0).transform(inc)
$.. ==> <1>
jshell> Actually.ok(0).transform(inv)
$.. ==> [java.lang.ArithmeticException] / by zero
jshell> Actually.<Integer>ok(0).transform(incNum)
$.. ==> <1>
jshell> Actually.<Integer>ok(0).transform(invNum)
$.. ==> [java.lang.ArithmeticException] / by zero
jshell> Actually.ok(0).transform(incNum)
$.. ==> <1>
jshell> Actually.ok(0).transform(invNum)
$.. ==> [java.lang.ArithmeticException] / by zero
```

You can test your code by running the `Test3.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Transformer

We will now create an abstract class for a special `Immutator`. This `Immutator` can be chained. Since there are two ways to chain an `Immutator`, we are going to do both. Mathematically, two functions can be composed, written as `(f ∘ g)(x)` as either `f(g(x))` or `g(f(x))`. Here, we are going to do both. We implement

- `f(g(x))` as `f.after(g).invoke(x)`
- `g(f(x))` as `f.before(g).invoke(x)`

How are we going to do this? We had defined a class inside another class above. So now, we are going to do something even crazier, we are going to define a class inside a method! We call this local class. Here's the deal, if you define a class inside a method, you have access to the argument. Why do we need access to the argument? Well, consider `f.after(g)`. The result of this should also be an `Immutator`. Which means, the result has an `invoke` method. But invoking the result is equivalent to `f.invoke(g.invoke(..))`.

If we look at `f.after(g)`, then the argument is `g` and we can invoke `g.invoke(..)` inside the local class. What we still need is to be able to use the result of `g.invoke(..)` as argument to `f.invoke`. We cannot really invoke `f.invoke` using `this.invoke` because we are in the local class so the keyword `this` is bound to this local class and not the original `f` instance. An easy solution is to create a temporary variable called `f` assigned to `this`. The limitation in Java is that you cannot change the value of `f`. In other words, it is kind of like there is a `final` keyword used on `f`.

Given the explanation above, we can now create this abstract class for a special

`Immutator`. We call this `Transformer<R,P>` and it should implement `Immutator<R,P>`. `Transformer` have two non-abstract methods

- The method `after` such that `f.after(g).invoke(x)` is equivalent to `f(g(x))`
  - The method accepts an `Transformer<P,N>` as an argument and returns a `Transfomer<R,N>`. In other words, we chain `Transformer<R,P>` and `Transformer<P,N>` to form `Transformer<R,N>`.
- The method `before` such that `f.before(g).invoke(x)` is equivalent to `g(f(x))`
  - The method accepts an `Transformer<T,R>` as an argument and returns a `Transfomer<T,P>`. In other words, we chain `Transformer<T,R>` and `Transformer<R,P>` to form `Transformer<T,P>`.

```
jshell> sqrPlusOneA.invoke(2)
$.. ==> 5
jshell> sqrPlusOneB.invoke(2)
$.. ==> 5
jshell> plusOneSqrA.invoke(2)
$.. ==> 9
jshell> plusOneSqrB.invoke(2)
$.. ==> 9
```

You can test your code by running the `Test4.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Constructor

Let's look at the other kind of special `Immutator` that automatically wraps the result in `Actually<T>`. We call this special `Immutator` as constructor. This simplifies our task in `Actually` since we do not have to wrap the result in another `Actually` and can simply let the constructor do the job for us. But first we need to

- Add an abstract method `next` in `Actually<T>` that takes in an `Immutator<..>` as the parameter. The `Immutator` object transforms the value of type `T` into a value of type `Actually<R>`, for some type `R`. In other words, it accepts `T` and returns `Actually<R>`.
  - Implement `next` in `Success<T>` such that it returns `Actually<R>` (*instead of* *`Actually<Actually<R>>`*) unless there is an exception. If there is an exception, then it returns a `Failure`.

- Implement `next` in `Failure` such that it propagates the exception contained as a new `Failure`.

The use of constructor allows us to chain easily. We first create an `Actually` using a constructor and then simply chain using `next`. At the end, we may `unwrap` if we believe there will not be an error or we use `except` / `unless` if think there may be an error that we want to restart with fresh value.

```
jshell> make.invoke(0).next(inc).next(inc).next(half)
$.. ==> <1>
jshell> make.invoke(0).next(inc).next(half).next(inc)
$.. ==> [java.lang.Exception] odd number
jshell> make.invoke(0).next(inc).next(inc).next(half).except(zero)
$.. ==> 1
jshell> make.invoke(0).next(inc).next(half).next(inc).except(zero)
$.. ==> 0
```

You can test your code by running the `Test5.java` provided. The following should compile without errors or warnings. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes Test5.java
$ java Test5
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Using `Actually`

Now that we have our `Actually` class, let's try to use it to do something more meaningful.

It is a common idiom (*although not a good one*) for a method to return a value if successful and return a `null` otherwise. It is up to the caller to check and make sure that the return value is not `null` before using it, to prevent receiving a run-time `NullPointerException`. In some cases, it may also be simpler (*although still not good*) to simply enclose the `NullPointerException` in a try-catch so that there is no need to check for any `null` value.

One example of this is the `Map<K,V>` implements in Java. The method `Map::get` returns `null` if the key that you are looking for does not exist. This may cause confusion if we are actually mapping some key to `null`. In any case, if the result is `null`, then using this for subsequent method invocation will result in `NullPointerException`.

We have given you a program `Lab5.java` that uses multiple layers of `Map` to store information about modules, the students in the module, and their assessment grades. There is a method `getGrade` that, given this map, a student, a module, and an assessment, look up the corresponding grade. There are multiple checks if a returned value is `null` in

this method.

Our new `Actually<T>` class provides a good abstraction for the chained operation involving the return value from `Map::get` since if there is an error, our `Actually<T>` will simply propagate the error. As such, there is no need to check if the return value is `null` or to put the code inside try-catch block. If the return value is indeed `null`, then we will simply propagate the exception until the end.

Your final task is to modify `getGrade` so that it uses `Actually<T>` instead:

- Declare and initialize two `Constant` instances using anonymous classes.
  a. One to wrap the `db` in `Actually<T>`.
  b. One to produce the string `"No such entry"`.
- Declare and initialize three `Immutator` instances using anonymous classes.
  a. One for the map from `get(student)`.
  b. One for the map from `get(module)`.
  c. One for the string representation of `get(assessment)`.
- Use the two `Constant`, three `Immutator` as well as `Constant::init`, `Actually::next`, and `Actually::except` to achieve the same functionality as the given `getGrade` in a *single return statement*. In other words, your `getGrade` should consists of six Java statements: two to create two `Constant`, three to create three `Immutator`, and one `return` statement. The skeleton has been given.
- Your code should not have any more conditional statements or references to `null` or using any try-catch.

Files

A set of empty files have been given to you. You should only edit these files. You must not add any additional files. Your folder structure should look like the following:

```
<your-lab5-root>
\--- cs2030s
|    \--- fp
|         \--- Action.java
|         +--- Actionable.java
|         +--- Actually.java
|         +--- Constant.java
|         +--- Immutator.java
|         +--- Immutatorable.java
|         +--- Transformer.java
+--- CS2030STest.java
+--- Lab5.h
```

```
+--- Lab5.java
+--- Lab5.pdf
+--- Test1.java
+--- Test2.java
+--- Test3.java
+--- Test4.java
+--- Test5.java
```

The files `Test1.java`, `Test2.java`, etc., as well as `CS2030STest.java` and `Lab5.h`, are provided for testing. You can edit them to add your test cases, but they will not be submitted. You *must* also submit the file `Lab5.h` (*in reality, this is a bash file but CodeCrunch does not allow submission of bash file*) along with your files.

Since CodeCrunch does not allow submission of folder or zip file, you are to submit the files inside the directory cs2030s/fp along with the other file without the need for folder.

Following CS2030S Style Guide

You should make sure that your code follows the given Java style guide. You are not required to correct the styling error for the `Test1.java`, `Test2.java`, etc., as well as `CS2030STest.java`.

Grading

This lab is worth 12 marks and contributes 3% to your final grade. The marking scheme is as follows:

- Style: 2 marks
- Everything Else: 10 marks

We will deduct 1 mark for each unnecessary use of `@SuppressWarnings` and each raw type. `@SuppressWarnings` should be used appropriately and not abused to remove compilation warnings. Furthermore, there should not be any warnings when compiled with `-Xlint:unchecked` and/or `-Xlint:rawtypes`.

Note that the style marks are conditioned on the evidence of efforts in solving Lab 5.

WARNING ❗

We would like to remind you of the following:

- We will take the latest submission only. If you have submitted your work and you

resubmit the same work after the deadline, late submission penalty will apply.

- We will no longer accept submission two days after the deadline.
    - This also applies to all previous labs to ease grading of really late submission and let the TA focus on their assessments.

---

1. In fact, `java.lang` is automatically imported by JVM.
2. Can be implemented using this but not actually this.
3. The more proper term is function composition where `(f ∘ g)(x)` is defined as either `f(g(x))` or `g(f(x))` depending on mathematicians/programmers you are talking to.