CS2030S PRACTICAL ASSESSMENT II

AY2021/22 Semester 2

April 2021

INSTRUCTIONS

---

1. This Practical Assessment consists of two questions. Answer ALL questions.
2. The total mark for this assessment is 40. Question 1: 25 marks. Question 2: 15 marks.
3. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
4. You should see the following in your home directory.
5. The files Test1.java, Test2.java... Test5.java, and CS2030STest.java for testing your solution.
6. The skeleton files for Question 1: `cs2030s/fp/Try.java`
7. The skeleton files for Question 2: `Main.java`
8. The following files to solve Question 1 are provided as part of the `cs2030s.fp` package: `Producer.java`, `Consumer.java`, `Transformer.java`, and `Runnable.java`.
9. The following files to solve Question 2 are provided `Circle.java`, `Point.java`, `RandomPoint.java`
10. The file `StreamAPI.md` contains information about the `Stream` class from the `java.util.stream` package.
11. Solve the programming tasks by editing `cs2030s/fp/Try.java` and `Main.java`. Only these two files will be graded.
12. Write your student number on top of the files `cs2030s/fp/Try.java` and `Main.java` as part of the @author tag. Do not write your name.

Important

- `Consumer<T>` is equivalent to `Action<T>`
- `Producer<T>` is equivalent to `Constant<T>`

- `Transformer<T, U>` is equivalent to `Immutator<U, T>`
- There is no equivalent to `Runnable`.

QUESTION 1: Try (25 marks)

Marking Criteria

- Functionality and type correctness (15 marks)
- OO design (5 marks)
- Style (2 marks)
- Documentation (3 marks)

Code that cannot be compiled will receive a 50% penalty for this question.

You need to write the javadoc document for five (and only five) of the methods identified below.

Your Task

In Java, we handle exceptions with `try` and `catch`. For example,

```
Circle c;
try {
  c = new Circle(point, radius);
} catch (IllegalArgumentException e) {
  System.err.println(e.getMessage());
}
```

When we code with the functional paradigm, however, we prefer to chain our operations and keep our functions pure. A more functional way to write this block of code is to use the `Try` monad:

```
Try<Circle> c = Try.of(() -> new Circle(point, radius))
```

The `Try` monad is a way to encapsulate the result of the computation if it is successful, or the reason for failure if the computational failed. We refer to these two possibilities as success and failure respectively. In the example above, the `Try<Circle>` instance would contain the new circle if it is a success, or an `IllegalArgumentException` if it fails.

The reason for failure can be encapsulated as an instance of the `Throwable` class. This class is defined in the `java.lang` package and it is the parent class of `Exception` and `Error`. A `Throwable` instance can be thrown and caught. Note that: - `cs2030s.fp.Producer::produce` and `cs2030s.fp.Runnable::run` now throw a `Throwable`. - You don't need to call any methods or access any fields related to `Throwable` beyond catching, throwing, and converting to string.

Your Task

You will implement the `Try` monad in this question as part of the `cs2030s.fp` package.

We break down the tasks you need to do into several sections. We suggest that you read through the whole question, plan your solution carefully before starting.

Please be reminded of the following:

- You should design your code so that it is extensible to other possible states of computation in the future, beyond just success and failure.
- Your code should be type-safe and catch as many type mismatches as possible during compile time.

Assumption

You can assume that everywhere a method of `Try` accepts a functional interface or a `Throwable` as a parameter, the argument we pass in will not be `null`. When a value is expected, however, there is a possibility that we pass in a `null` as an argument.

The Basics

First, please implement the following methods:

- The `of` factory method, which allows us to create a new `Try` monad with a producer of type `Producer` (imported from the package `cs2030s.fp.Producer`). Returns success if the producer produces a value successfully, or a failure containing the throwable if the producer throws an exception.
- The `success` factory method, which allows us to create a new `Try` monad with a value. A `Try` monad created this way is always a success.
- The `failure` factory method, which allows us to create a new `Try` monad with a `Throwable`. A `Try` monad created this way is always a failure.
- Override the `equals` method in `Object` so that it checks if two different `Try` instances are equal. Two `Try` instances are equal if (i) they are both a success and the values contained in them equal to each other, or (ii) they are both a failure and the `Throwable`s contained in them have the same string representation.
- Implement the method `get()`. `get()` returns the value if the `Try` is a success. It throws the `Throwable` if it is a failure.

Study carefully how these methods can be used in the examples below:

```
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Try

```

```
jshell> Try.success(1).get();
$.. ==> 1

jshell> try {
   ...>   Try.failure(new Error()).get();
   ...> } catch (Error e) {
   ...>   System.out.println(e);
   ...> }
java.lang.Error

jshell> Try.<Number>of((Producer<Integer>) () -> 2).get();
$.. ==> 2

jshell> try {
   ...>   Try<Number> t = Try.of(() -> 4/0);
   ...> } catch (java.lang.ArithmeticException e) {
   ...>   System.out.println(e);
   ...> }

jshell> Try.success(3).equals(Try.success(3))
$.. ==> true
jshell> Try.success(null).equals(Try.success(null))
$.. ==> true
jshell> Try.success(3).equals(Try.success(null))
$.. ==> false
jshell> Try.success(null).equals(Try.success(3))
$.. ==> false
jshell> Try.success(3).equals(Try.success("3"))
$.. ==> false
jshell> Try.success(3).equals(3)
$.. ==> false
jshell> Try.failure(new Error()).equals(new Error())
$.. ==> false
jshell> Try.failure(new Error()).equals(Try.success(3))
$.. ==> false
jshell> Try.failure(new Error()).equals(Try.success(new Error()))
$.. ==> false
jshell> Try.success(new Error()).equals(Try.failure(new Error()))
$.. ==> false
jshell> Try.failure(new ArithmeticException()).equals(Try.failure(new Error()))
$.. ==> false
jshell> Try.failure(new ArithmeticException()).equals(Try.failure(new
ArithmeticException()))
$.. ==> true
```

You can also test your code with `Test1.java`:

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

Write the javadoc documentation for `of`, `success`, and `failure` for `Try`. Note that since we do not require you to write javadoc for every class and methods, `checkstyle` no longer warns about missing javadoc for your class and methods.

map

Now, implement the `map` method so that we can apply a computation on the content of `Try`. If `map` is called on a `Try` instance that is a failure, the same instance of `Try` is returned. Otherwise, if it is a success, the lambda expression is applied to the value contained within `Try`. If this lambda expression throws a `Throwable`, the calling `Try` becomes a failure containing the `Throwable` thrown.

Study carefully how `map` can be used in the examples below:

```
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Transformer
jshell> import cs2030s.fp.Try

jshell> Try.success(4).map(x -> x + 1).get();
$.. ==> 5

jshell> try {
   ...>   Try.failure(new NullPointerException()).map(x -> x.toString()).get();
   ...> } catch (NullPointerException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.NullPointerException

jshell> Try.failure(new IOException()).map(x ->
x.toString()).equals(Try.failure(new IOException()));
$.. ==> true

jshell> Try.success(4).map(x -> 8 / x).map(x -> x + 1).get();
$.. ==> 3

jshell> try {
   ...>   Try.success(0).map(x -> 8 / x).map(x -> x + 1).get();
   ...> } catch (ArithmeticException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.ArithmeticException: / by zero

jshell> Transformer<Object, Integer> hash = x -> x.hashCode();
jshell> Try.success("hello").map(hash).get()
$.. ==> 99162322

jshell> try {
   ...>   Try.<Integer>success(null).map(hash).get();
   ...> } catch (NullPointerException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.NullPointerException
```

You can also test your code with `Test2.java`:

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

flatMap

Now, make `Try` a monad. Implement the `flatMap` method so that we can compose multiple methods

that produce a `Try` together. If `flatMap` is called on a failure, return the failure. Otherwise, if it is a success, apply the lambda expression on the value contained within `Try` and return the result.

Study carefully how `map` can be used in the examples below:

```
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Transformer
jshell> import cs2030s.fp.Try

jshell> Try.success(4).flatMap(x -> Try.success(x + 1)).get();
$.. ==> 5

jshell> try {
   ...>   Try.success(4)
   ...>       .flatMap(x -> Try.failure(new IOException()))
   ...>       .get();
   ...> } catch (IOException e) {
   ...>   System.out.println(e);
   ...> }
java.io.IOException

jshell> try {
   ...>   Try.failure(new NullPointerException())
   ...>       .flatMap(x -> Try.success(x.toString()))
   ...>       .get();
   ...> } catch (NullPointerException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.NullPointerException

jshell> Try.failure(new IOException()).flatMap(x ->
Try.success(x.toString())).equals(Try.failure(new IOException()));
$.. ==> true

jshell> try {
   ...>   Try.failure(new NullPointerException())
   ...>       .flatMap(x -> Try.failure(new IOException()))
   ...>       .get();
   ...> } catch (NullPointerException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.NullPointerException

jshell> Transformer<Object, Try<Integer>> hash = x -> Try.success(x.hashCode());
jshell> Try<Number> t = Try.success("hello").flatMap(hash);
```

You can also test your code with `Test3.java`:

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

Dealing with failures

The methods `map` and `flatMap` apply the given lambda to the value contained within the `Try` monad

where it is a success. Write the following method to deal with failure:

- `onFailure`: Return this instance if the calling `Try` instance is a success. Consume the `Throwable` with a `Consumer` if it is a failure, and then either (i) return this instance if the consumer runs successfully, or (ii) return a failure instance containing the error/exception when consuming the `Throwable`.

For example, we can use `onFailure` to replace this snippet

```
Circle c;
try {
  c = new Circle(point, radius);
} catch (IllegalArgumentException e) {
  System.err.println(e.getMessage());
}
```

with:

```
Try<Circle> c = Try.of(() -> new Circle(point, radius))
                   .onFailure(System.err::println);
```

We can also recover from the failure, by turning the `Try` into a success. Write the following method:

- `recover`: Return this instance if it is a success. If this `Try` instance is a failure. Apply the given `Transformer` to the `Throwable`, if the transformation is a success, return the resulting `Try`, otherwise, return a failure containing the error/exception when transforming the `Throwable`.

For example, we can use `recover` to replace this snippet

```
Circle c;
try {
  c = new Circle(point, radius);
} catch (IllegalArgumentException e) {
  c = new Circle(point, 1);
}
```

with:

```
Try<Circle> c = Try.of(() -> new Circle(point, radius))
                   .recover(e -> new Circle(point, 1));
```

Study carefully how `onFailure` and `recover` can be used in the examples below:

```
jshell> import cs2030s.fp.Consumer
jshell> import cs2030s.fp.Producer
jshell> import cs2030s.fp.Transformer
jshell> import cs2030s.fp.Try

jshell> Try.success(4).onFailure(System.out::println).get()
$.. ==> 4

jshell> try {
   ...>   Try.failure(new IOException()).onFailure(System.out::println).get();
   ...> } catch (IOException e) {
```

```
   ...>   System.out.println(e);
   ...> }
java.io.IOException
java.io.IOException

jshell> try {
   ...>   Try.failure(new IOException()).onFailure(e -> { int x = 1 / 0; }).get();
   ...> } catch (ArithmeticException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.ArithmeticException: / by zero

jshell> Consumer<Object> print = System.out::println
jshell> Try.<Number>success(4).onFailure(print).get()
$.. ==> 4

jshell> Try.success(4).recover(e -> 10).get()
$.. ==> 4
jshell> Try.failure(new IOException()).recover(e -> 10).get();
$.. ==> 10

jshell> try {
   ...>   Try.failure(new IOException()).recover(e -> e.hashCode() / 0).get();
   ...> } catch (ArithmeticException e) {
   ...>   System.out.println(e);
   ...> }
java.lang.ArithmeticException: / by zero

jshell> Transformer<Object, Integer> hash = x -> x.hashCode();
jshell> Try.<Number>success(4).recover(hash).get()
$.. ==> 4
```

You can also test your code with `Test4.java`:

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

Write the javadoc documentation for `onFailure` and `recover` for `Try`.

QUESTION 2: Stream (15 marks)

Marking Criteria

- Functionality and type correctness (12 marks)
- Style (3 marks)

Code that cannot be compiled will receive a 50% penalty for this question.

Your Task

There are five parts to this question that may or may not be dependent on each other. You will need

to write five single line `Stream` pipelines to generate certain `Stream`s and solve certain computations. The last part of this question will get you to re-solve the Lab 0 (Pi Estimation) using only a `Stream` and some supporting classes.

The Stream API is included in the file `StreamAPI.md`.

You are provided with a `Point`, `Circle`, `RandomPoint`, and `Main` class. These `Point`, `Circle` and `RandomPoint` classes are similar to those used in Lab 0. Take a look at them to see what are the methods available.

All of your single line pipelines will be written in the `Main.java` skeleton file. Each method body must contain only a single return statement.

Implement the `pointStream` method.

The method `pointStream` has two arguments: `point` of type `Point` and `f` of type `Function<Point,Point>`. Recall that `Function` is the Java equivalent of our `Transformer` functional interface which has the single abstract method `apply` instead of `transform`. The method should return a `Stream<Point>` which contains the point `p`, followed by `f(p)`, and then `f(f(p))`, and so on. Implement this method body using a single stream pipeline.

Some examples of use are shown below:

```
jshell> pointStream(new Point(0, 0), p -> new Point(p.getX(), p.getY() +
1)).limit(3).forEach(System.out::println)
(0.0, 0.0)
(0.0, 1.0)
(0.0, 2.0)

jshell> pointStream(new Point(0, 0), p -> new Point(p.getX() + 1,
p.getY())).limit(3).forEach(System.out::println)
(0.0, 0.0)
(1.0, 0.0)
(2.0, 0.0)

jshell> pointStream(new Point(0, 0), p -> new Point(p.getX() + 1, p.getY() +
1)).limit(3).forEach(System.out::println)
(0.0, 0.0)
(1.0, 1.0)
(2.0, 2.0)
```

Implement the `generateGrid` method.

The method `generateGrid` has two arguments: `point` of type `Point` and `n` which is of type `int`. This method should return a finite stream of type `Stream<Point>` containing the `n * n` points that define a grid starting from the point `point` and then incrementing both `x` and `y` cordinates by one. For example: a grid of size `3` starting from a point `(0,0)` should look like the following:

```
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

When in the stream they should appear in the order of the row first i.e. `(0,0) (0,1) (0,2) (1,0) (1,1) (1,2) (2,0) (2,1) (2,2)`.

Implement this method body using a single stream pipeline.

Some examples of use are shown below:

```
jshell> generateGrid(new Point(0, 0), 2).forEach(System.out::println)
(0.0, 0.0)
(0.0, 1.0)
(1.0, 0.0)
(1.0, 1.0)

jshell> generateGrid(new Point(0, 0), 3).forEach(System.out::println)
(0.0, 0.0)
(0.0, 1.0)
(0.0, 2.0)
(1.0, 0.0)
(1.0, 1.0)
(1.0, 2.0)
(2.0, 0.0)
(2.0, 1.0)
(2.0, 2.0)

jshell> generateGrid(new Point(-1, 0), 2).forEach(System.out::println)
(-1.0, 0.0)
(-1.0, 1.0)
(0.0, 0.0)
(0.0, 1.0)
```

Implement the `concentricCircles` method.

The method `concentricCircles` has two arguments: `circle` of type `Circle` and `f` which is of type `Function<Double,Double>`. The method should return a `Stream<Circle>` which contains the first circle `circle`, followed by the circle with a radius given by `f(circle.getRadius())`, and then `f(f(circle.getRadius()))`, and so on. In this way, we will have a stream of concentric circles (circles with a common center but with different radii - much like a target in archery).

Implement this method body using a single stream pipeline.

Some examples of use are shown below:

```
jshell> concentricCircles(new Circle(new Point(1, 1), 1.0),x -> x +
1).limit(3).forEach(System.out::println)
{ center: (1.0, 1.0), radius: 1.0 }
{ center: (1.0, 1.0), radius: 2.0 }
{ center: (1.0, 1.0), radius: 3.0 }

jshell> concentricCircles(new Circle(new Point(0, 0), 1.0),x -> x +
0.5).limit(3).forEach(System.out::println)
{ center: (0.0, 0.0), radius: 1.0 }
```

```
    { center: (0.0, 0.0), radius: 1.5 }
    { center: (0.0, 0.0), radius: 2.0 }
```

Implement the `pointStreamFromCircle` method.

The method `pointStreamFromCircle` has one argument: `circles` of type `Stream<Circle>`. The method should return a `Stream<Point>` which contains the centers of all the circles in the `circles` list. Implement this method body using a single stream pipeline.

An example of use is shown below:

```
jshell> pointStreamFromCircle(Stream.of(new Circle(new Point(0, 0), 1), new
Circle(new Point(1, 1), 2), new Circle(new Point(-1, -1),
1))).forEach(System.out::println)
(0.0, 0.0)
(1.0, 1.0)
(-1.0, -1.0)
```

Estimating Pi using Monte Carlo Method

As in the Lab 0, we will use the Monte Carlo method for estimating the value of pi. We have a square of width 2r, and within it, a circle with a radius of r.

We randomly generate k points within the square. We count how many points fall within the circle. Suppose n points out of k fall within the circle.

Since the area of the square is 4r^2 and the area of the circle is pi r^2, the ratio between them is pi/4. The ratio n/k should therefore be pi/4, and pi can be estimated as 4n/k.

`RandomPoint`

`RandomPoint` is a subclass of `Point` that represents a randomly generated point. The random number generator that generates a random point has a default seed of 1. There is a public method `setSeed()` that we can use to update the seed.

This is how it can be used.

To generate a new point,

```
Point p = new RandomPoint(minX, maxX, minY, maxY);
```

`minX`, `minY`, `maxX`, `maxY` represent the minimum and maximum possible x and y values respectively, for each randomly generated point.

To set the random seed,

```
RandomPoint.setSeed(10);
```

Implement the `estimatePi` method.

You will now implement an `estimatePi` method using a single stream pipeline. `estimatePi` takes in three arguments, `circle` of type `Circle` , `supplier` of type `Supplier<RandomPoint>` (`Supplier` is the Java equivalent `Producer`) and `k` of type int. The method should return an estimate of pi as a `double` , using the monte carlo method by generating `k` `RandomPoint` s.

Hint: the `contains` method in `Circle` might come in handy.

Some examples of its use:

```
jshell> RandomPoint.setSeed(10);
jshell> Circle c = new Circle(new Point(0.5, 0.5), 0.5)
jshell> estimatePi(c, () -> new RandomPoint(0, 1, 0, 1), 10000)
$.. ==> 3.1284
jshell> RandomPoint.setSeed(55);
jshell> estimatePi(c, () -> new RandomPoint(0, 1, 0, 1), 10000)
$.. ==> 3.1216
```

```
                                END OF PAPER
```