INSTRUCTIONS

1. This Practical Assessment consists of two questions. Answer ALL questions.
2. The total mark for this assessment is 40. Answer ALL questions.
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`, `Test3.java`, `Test4.java`, and `CS2030STest.java` for testing your solution.
6. The skeleton files for Question 1: `Either.java` which is part of the `cs2030s.fp` package.
7. The following files to solve Question 1 are provided as part of the `cs2030s.fp` package: `Transformer.java`, and `BooleanCondition.java`.
8. The skeleton file for Question 2: `Query.java`
9. An abridged Stream API is provided in the file `StreamAPI.md`
10. Solve the programming tasks by creating any necessary files and editing them. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
11. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
12. Write your student number on top of EVERY FILE you created or edited as part of the @author tag. Do not write your name.
13. IMPORTANT: Make sure all the code you have written compiles. If one of the Java files you have written causes any compilation error, you will receive 0 marks for that question.

Important

- `Transformer<T, U>` is equivalent to `Immutator<U, T>`
- `BooleanCondition<T>` is equivalent to `Immutator<Boolean, T>`

QUESTION 1: Either (25 marks)

Marking Criteria

- functionality and type correctness (15 marks)
- OO design (5 marks)
- style (2 marks)
- documentation (3 marks)

Note that you need to write javadoc document for five of the methods identified below. You don't have to write javadoc for other methods besides those identified below.

Motivation

In languages such as Javascript, we can write functions or methods such as

```
function foo(x) {
  if (x == 0) {
    return "zero";
  }
  return 1;
}
```

where the return value can be either a string or an integer. In Java, however, the return value of a method can only be of a single type. We cannot write the code equivalent to the example above as we are forced to choose either to return a `String` or return an `int`.

To get around this limitation, other strongly typed languages such as Scala provides a monad called `Either` that can encapsulate a value where the type is one of two possibilities. Java does not provide this abstraction, so you will build one in this question.

Write an abstract class `Either<L, R>` that encapsulates two possible results of computation with possible differing types `L` and `R` in the package `cs2030s.fp`. We refer to these two possibilities as `left` and `right` respectively. One of these possibilities must be true -- i.e., we are guaranteed that there is always a value encapsulated.

We can now rewrite the code above into Java using `Either` monad

```
Either<String, Integer> foo(int x) {
  if (x == 0) {
    return Either.left("zero");
  }
  return Either.right(1);
}
```

Your Task

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

The Basics

First, please implement the following methods:

- the `left` factory method, which allows us to create a left, i.e., a new `Either` monad with a left value.
- the `right` factory method, which allows us to create a right, i.e., a new `Either` monad with a right value.
- `isLeft()`, which will return the value true if the `Either` is a left, false otherwise.
- `isRight()`, which will return the value true if the `Either` is a right, false otherwise.
- `getLeft()`, which will return the value if it is a `left` and throw a `NoSuchElementException` otherwise.
- `getRight()`, which will return the value if it is a `right` and throw a `NoSuchElementException` otherwise.

and overrides the following methods from `Object`: - `equals(Object o)`, which will return the true if two `either` are equals; false otherwise. Two `Either` are equals if (i) either they are both left or both right; and (ii) the value contained inside are equals. - `toString()`, which will return a string of the pattern "Left[...]" if it is a left, or the pattern "Right[...]" if it is a right, with ... replaced with the string representation of the corresponding value stored inside the `Either`.

The `NoSuchElementException` can be found in the package `java.util`.

Write the javadoc documentation for `left` and `right` in the `Either` class. Since we do not require you to write javadoc for every class and methods, `checkstyle` does not warn about missing javadoc for your class and methods. If you wrote some javadoc and did not format it properly, however, `checkstyle` will warn you to help you write proper `javadoc` comments.

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

```
jshell> import cs2030s.fp.Either;
jshell> // Expect error
jshell> new Either<>()
|  Error:
|  cs2030s.fp.Either is abstract; cannot be instantiated
|  new Either<>()
|  ^------------^

jshell> Either.right("two").isLeft()
$.. ==> false
jshell> Either.right("two").isRight()
$.. ==> true
jshell> Either.right("two").getRight()
$.. ==> "two"
jshell> Either.left(2).isLeft()
$.. ==> true
jshell> Either.left(2).isRight()
$.. ==> false
jshell> Either.left(2).getLeft()
$.. ==> 2

jshell> // Expect NoSuchElementException
jshell> Either.left(2).getRight()
|  Exception java.util.NoSuchElementException
```

```
|        at Either$Left.getRight (Either.java:94)
|        at (#8:1)
jshell> Either.right("two").getLeft()
|  Exception java.util.NoSuchElementException
|        at Either$Right.getLeft (Either.java:167)
|        at (#9:1)

jshell> // Compilation error due to type mismatch
jshell> Either<String, Integer> e = Either.left(2)
|  Error:
|  incompatible types: inference variable L has incompatible bounds
|      equality constraints: java.lang.String
|      lower bounds: java.lang.Integer
|  Either<String, Integer> e = Either.left(2);
|                              ^------------^
jshell> Either<Double, Long> e = Either.right(true)
|  Error:
|  incompatible types: inference variable R has incompatible bounds
|      equality constraints: java.lang.Long
|      lower bounds: java.lang.Boolean
|  Either<Double, Long> e = Either.right(true);
|                           ^----------------^

jshell> String two = new String("two")
jshell> Either.right(two).equals(Either.right("two"))
$.. ==> true
jshell> Either.right(two).equals(Either.left("two"))
$.. ==> false
jshell> Either.left(two).equals(Either.right("two"))
$.. ==> false
jshell> Either.left(two).equals(Either.left("two"))
$.. ==> true
jshell> Either.right(two).equals(Either.right(2))
$.. ==> false
jshell> Either.left(two).equals(Either.left(2))
$.. ==> false
jshell> Either.left(null).equals(Either.left(null))
$.. ==> true
jshell> Either.right(null).equals(Either.right(null))
$.. ==> true

jshell> Either.right(20).toString()
$.. ==> "Right[20]"
jshell> Either.left("thirty").toString()
$.. ==> "Left[thirty]"
```

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

```
$ javac cs2030s/fp/Either.java
$ javac Test1.java
$ java Test1
$ java -jar checkstyle.jar -c cs2030_checks.xml cs2030s/fp/Either.java
$ javadoc -quiet -private -d docs cs2030s/fp/Either.java
```

map

Implement the `map` method which takes in two `Transformer`s so that they can be applied computation on the content of `Either`. If `map` is called on an `Either` instance that is a `left` it will apply the left `Transformer`, and if it is a `right` it will apply the right `Transformer`.

flatMap

Implement the `flatMap` method that takes in two `Transformer`s so that we can compose multiple methods that produce a `Either` together. If `flatMap` is called on a left, it will apply the left transformer. Otherwise it will apply the right transformer.

fold

Implement the `fold` method which takes in two `Transformer`s and folds the two possible types into a common type. That is, if this is an `Either<Integer, Double>` and your want to fold it into a `String` your left and right `Transformer`s need to map to `String`. If `fold` is called on a left, it will apply the left transformer. Otherwise it will apply the right transformer.

filterOrElse

The method `filterOrElse` takes in two arguments, a `BooleanCondition` and a `Transformer`. This method will behave differently dependent on whether the value is a right or a left. Is a value if a right, it will check if the the given `BooleanCondition` holds for the right value, if it is, it will return the right unchanged. It the predicate does not hold it will return a left with the value from applying the given tranformer to the right value. If it is a left it will only return the left unchanged.

Study carefully how `map`, `flatMap`, `fold`, and `filterOrElse` can be used in the examples below:

```
jshell> import cs2030s.fp.Either;
jshell> import cs2030s.fp.BooleanCondition;
jshell> import cs2030s.fp.Transformer;
jshell> Either.<Integer, String>left(2).map(i -> i + 2, s -> s + " + 2")
$.. ==> Left[4]
jshell> Either.<Integer, String>right("2").map(i -> i + 2, s -> s + " + 2")
$.. ==> Right[2 + 2]

jshell> Transformer<Object, Integer> hash = o -> o.hashCode();
jshell> Either<Number, Number> enn = Either.left(2).map(hash, hash);
jshell> Either<Number, Number> enn = Either.right(2).map(hash, hash);

jshell> Either.<Integer, String>left(2).flatMap(i -> Either.left(i + 2), s ->
Either.right(s + " + 2"));
$.. ==> Left[4]
jshell> Either.<Integer, String>right("2").flatMap(i -> Either.left(i + 2), s ->
Either.right(s + " + 2"));
$.. ==> Right[2 + 2]

jshell> Transformer<Object, Either<String, Integer>> strOrHash;
jshell> strOrHash = o -> (o.equals(8) ?
   ...>     Either.<String, Integer>left(o.toString()) :
   ...>     Either.<String, Integer>right(o.hashCode()));
jshell> Either<Object, Number> enn = Either.left(2).flatMap(strOrHash, strOrHash);
jshell> Either<Object, Number> enn = Either.left(8).flatMap(strOrHash, strOrHash);
jshell> Either<Object, Number> enn = Either.right(2).flatMap(strOrHash, strOrHash);
jshell> Either<Object, Number> enn = Either.right(8).flatMap(strOrHash, strOrHash);

jshell> Either.<List<Integer>, String>left(List.of(1,2,3)).fold(l -> l.size(), s ->
s.length());
```

```
$.. ==> 3
jshell> Either.<List<Integer>, String>right("hello there").fold(l -> l.size(), s ->
s.length());
$.. ==> 11
jshell> Either.<List<Integer>, String>left(List.of(1,2,3)).<Number>fold(hash,
hash);
$.. ==> 30817
jshell> Either.<List<Integer>, String>right("hello there").<Number>fold(hash,
hash);
$.. ==> 1791114646

jshell> Either.<String, Boolean>left("no change").filterOrElse(x -> x == true, x ->
"");
$.. ==> Left[no change]
jshell> Either.<String, Boolean>right(true).filterOrElse(x -> x == true, x -> "is
false");
$.. ==> Right[true]
jshell> Either.<String, Boolean>right(false).filterOrElse(x -> x == true, x -> "is
false");
$.. ==> Left[is false]

jshell> Transformer<Object, Exception> toException = o -> new
IllegalStateException(o + " is illegal");
jshell> BooleanCondition<Number> isPositive = n -> n.intValue() > 0;
jshell> Either.<Throwable, Integer>left(new
IllegalStateException()).filterOrElse(isPositive, toException);
$.. ==> Left[java.lang.IllegalStateException]
jshell> Either.<Throwable, Integer>right(0).filterOrElse(isPositive, toException);
$.. ==> Left[java.lang.IllegalStateException: 0 is illegal]
jshell> Either.<Throwable, Integer>right(8).filterOrElse(isPositive, toException);
$.. ==> Right[8]
```

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

```
$ javac cs2030s/fp/Either.java
$ javac Test2.java
$ java Test2
$ java -jar checkstyle.jar -c cs2030_checks.xml cs2030s/fp/Either.java
$ javadoc -quiet -private -d docs cs2030s/fp/Either.java
```

Write the javadoc documentation for `flatMap`, `fold`, and `filterOrElse` for `Either`.

Since we do not require you to write javadoc for every class and methods, `checkstyle` does not warn about missing javadoc for your class and methods. If you wrote some javadoc and did not format it properly, however, `checkstyle` will still warn you to help you write proper `javadoc` comments.

QUESTION 2: Streams (15 marks)

Marking Criteria

- correctness (13 marks)
- style (2 marks)

Background

In computing, we commonly organize data into tables for processing. In this question, we would like to explore how we can process and manipulate data stored in tables using Streams.

Consider the following table of customer records from a store. Each row of the table contains the name of a customer, and a list of purchases (identified by purchase ids, which are integers).

We will call this table the "Customer Table".

Names	Purchase Ids
Michelle	12, 56
Enzio	34, 90
Michael	78

Each purchase has a cost. The cost of each purchase is stored in another table called the "Sales Table". Each row in this table contains a purchase id and the corresponding cost of the purchase.

Purchase Ids	Cost
12	12.0
34	6.0
56	7.5
78	9.0
90	17.0

In this question, we will implement these tables using `Map`. Recall that a `Map` is an abstraction over a set of (key, value) pairs. Each pair (key, value) is a `Map.Entry` stored in the `Map`. Given a `Map.Entry`, we can retrieve the key with the `getKey()` method and retrieve the value with the `getValue()` method.

Treating each name as a key and the list of purchases as the value, the Customer Table can be represented as a `Map` from a `String` (Names) to a `List<Integer>` (Purchase Ids).

```
Map<String, List<Integer>> customerTable;
customerTable = Map.of(
    "Michelle", List.of(12, 56),
    "Enzio",    List.of(34, 90),
    "Michael",  List.of(78));
```

We can get the value of a `key` by using the `get` method.

```
customerTable.get("Michelle") // returns a List.of(12, 56)
```

Java `Map` provides methods to create a stream out of a `Map` entries, keys, and values.

```
customerTable.entrySet().stream() // returns a stream of `Map.Entry`
customerTable.keySet().stream()   // returns a stream of the Map keys
customerTable.values().stream()   // returns a stream of the Map values
```

Given the customers, list of their purchases, and the costs, we want to be able to build a table that maps between the name of the customer and the cost, as you can see below.

Customer name	Cost
Michelle	12.0
Michelle	7.5
Enzio	6.0
Enzio	17.0
Michael	9.0

Our final goal is to sum up the total cost of all the purchases made by each customer, as demonstrated below.

Customer name	Cost
Michelle	19.5
Enzio	23.0
Michael	9.0

Your Task

In this question, you are to write five `Stream` methods to operate on the Customer and Sales tables. Each method should only contain a single Stream pipeline. Nothing more. No local variables or classes can be defined.

You may call the methods you create when solving other parts of this questions.

getFilteredByKey

To get started, implement the `getFilteredByKey` methods. We have provided the skeleton for this first method in the `Query.java` file.

The `getFilteredByKey` takes in a table with type `Map<T, S>` and a predicate of type `Predicate<T>`. It returns a stream of entries (or rows) with the type `Stream<Map.Entry<T, S>>`, containing only rows in the original table for which the key passes the predicate.

Note that you do not have to worry about PECS for this question.

Study carefully how this method can be used in the examples below:

```
jshell> /open Query.java
jshell> Map<String, List<Integer>> customerTable = Map.of(
   ...>     "Michelle", List.of(12, 56),
   ...>     "Enzio",    List.of(34, 90),
   ...>     "Michael",  List.of(78));
jshell> Query.getFilteredByKey(customerTable, x ->
x.equals("Enzio")).forEach(System.out::println)
Enzio=[34, 90]
jshell> Query.getFilteredByKey(customerTable, x ->
x.startsWith("Mic")).forEach(System.out::println)
Michelle=[12, 56]
Michael=[78]
jshell> Query.getFilteredByKey(customerTable, x ->
x.startsWith("A")).forEach(System.out::println)
```

getIdsFromName

We now write a method to get all of the purchase ids for a given customer name.

Write the method `getIdsFromName` which takes in the Customer Table of type `Map<String, List<Integer>>` and a customer name (a `String`). It returns a `Stream<Integer>` containng all purchase ids for the given customer name. We can assume that there is at most one customer with the given name.

```
jshell> /open Query.java
jshell> Map<String, List<Integer>> customerTable = Map.of(
   ...>     "Michelle", List.of(12, 56),
   ...>     "Enzio",    List.of(34, 90),
   ...>     "Michael",  List.of(78));
jshell> Stream<Integer> purchaseIDs = Query.getIdsFromName(customerTable,
"Michelle")
jshell> purchaseIDs.collect(Collectors.toList());
$.. ==> [12, 56]
jshell> Stream<Integer> purchaseIDs = Query.getIdsFromName(customerTable, "Sam")
jshell> purchaseIDs.collect(Collectors.toList());
$.. ==> []
```

getCostsFromIDs

With the list of purchase IDs, we will now get the cost of each of these purchases. Write the method `getCostsFromIDs` that takes a Sales Table (of type `Map<Integer, Double>`) and a list of purchase ids (of type `Stream<Integer>`), and returns the cost of each purchase as a `Stream<Double>`. The costs returned must be in the same order as the corresponding purchase IDs.

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

```
jshell> /open Query.java
jshell> Map<Integer, Double> salesTable = Map.of(
   ...>     12, 12.0,
   ...>     34, 6.0,
   ...>     56, 7.5,
   ...>     78, 9.0,
   ...>     90, 17.0)
jshell> Stream<Double> costs = Query.getCostsFromIDs(salesTable, Stream.of(12))
jshell> costs.collect(Collectors.toList());
$.. ==> [12.0]
jshell> Stream<Double> costs = Query.getCostsFromIDs(salesTable, Stream.of(12, 90))
jshell> costs.collect(Collectors.toList());
$.. ==> [12.0, 17.0]
jshell> Stream<Double> costs = Query.getCostsFromIDs(salesTable, Stream.of(7))
jshell> costs.collect(Collectors.toList());
$.. ==> []
```

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

```
$ javac Query.java
$ javac Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml Query.java
```

allCustomerCosts

We will now put the information retrieved from the two tables together, and create a new table showing, on each row, the name of each customer and the cost of each purchase by the customer.

We will represent the output table as a `Stream<String>`, where every string in the stream is a row of the new table.

Write a method `allCustomerCosts` to do this. The method takes in a "Customer Table" and a "Sales Table" and returns a `Stream<String>` representing the new table. The order of the rows in the output does not matter.

Study carefully how this method can be used in the examples below:

```
jshell> /open Query.java
jshell> Map<String, List<Integer>> customerTable = Map.of(
   ...>     "Michelle", List.of(12, 56),
   ...>     "Enzio",    List.of(34, 90),
   ...>     "Michael",  List.of(78));
jshell> Map<Integer, Double> salesTable = Map.of(
   ...>     12, 12.0,
   ...>     34, 6.0,
   ...>     56, 7.5,
```

```
   ...>     78, 9.0,
   ...>     90, 17.0)
jshell> Map<String, List<Integer>> badCustomerTable = Map.of(
   ...>     "Bill", List.of(17),
   ...>     "Sam", List.of(19));
jshell> Map<Integer, Double> badSalesTable = Map.of(
   ...>     99, 3.0,
   ...>     98, 2.0);
jshell> Query.allCustomerCosts(customerTable,
salesTable).forEach(System.out::println);
Michelle: 12.0
Michelle: 7.5
Michael: 9.0
Enzio: 6.0
Enzio: 17.0
jshell> Query.allCustomerCosts(customerTable,
badSalesTable).forEach(System.out::println);
jshell> Query.allCustomerCosts(badCustomerTable,
salesTable).forEach(System.out::println);
```

totaledCustomerCosts

Finally, we will now create a new table to show the name of each customer and the total cost of the purchases by the customer.
We will again represent the output table as a `Stream<String>`, where every string in the stream is a row of the table.

Write a method `totaledCustomerCosts` to do this. The method takes in a "Customer Table" and a "Sales Table" and returns a `Stream<String>` representing the new table. The order of the rows in the output does not matter.

Study carefully how this method can be used in the examples below:

```
jshell> /open Query.java
jshell> Map<String, List<Integer>> customerTable = Map.of(
   ...>     "Michelle", List.of(12, 56),
   ...>     "Enzio",    List.of(34, 90),
   ...>     "Michael",  List.of(78));
jshell> Map<Integer, Double> salesTable = Map.of(
   ...>     12, 12.0,
   ...>     34, 6.0,
   ...>     56, 7.5,
   ...>     78, 9.0,
   ...>     90, 17.0)
jshell> Map<String, List<Integer>> badCustomerTable = Map.of(
   ...>     "Bill", List.of(17),
   ...>     "Sam", List.of(19));
jshell> Map<Integer, Double> badSalesTable = Map.of(
   ...>     99, 3.0,
   ...>     98, 2.0);
jshell> Query.totaledCustomerCosts(customerTable,
salesTable).forEach(System.out::println);
Michelle: 19.5
Michael: 9.0
Enzio: 23.0
jshell> Query.totaledCustomerCosts(customerTable,
badSalesTable).forEach(System.out::println);
Michelle: 0.0
```

```
Michael: 0.0
Enzio: 0.0
jshell> Query.totaledCustomerCosts(badCustomerTable,
badSalesTable).forEach(System.out::println);
Bill: 0.0
Sam: 0.0
```

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

```
$ javac Query.java
$ javac Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml Query.java
```

```
                              END OF PAPER
```