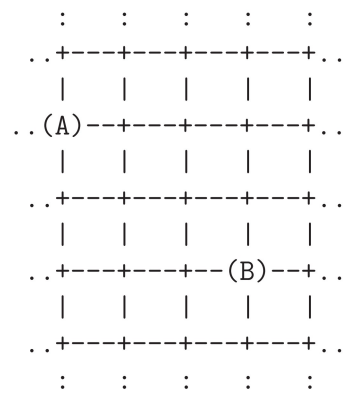
A city is built up of equally spaced horizontal streets and vertical avenues that run perpendicular to each other. Users of a city navigation application comprises of drivers and pedestrains. Each user can find the distance between itself and another user.

Suppose that a user `(A)` is situated at street x_A and avenue y_A, and another user `(B)` is situated at street x_B and avenue y_B.



- If a **driver** is situated at `(A)`, then the distance between `(A)` and `(B)` is computed using the city-block distance

$$|x_A - x_B| + |y_A - y_B|$$

  In the above example, the distance is 5.

- If a **pedestrian** is situated at `(A)`, then the distance between `(A)` and `(B)` is computed using the Euclidean distance

$$\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

  In the above example, the distance is $\sqrt{13}$.

Define appropriate interfaces/classes following the sample run below:

```
jshell> Stream.<User>of(new Driver(1, 1), new Pedestrian(4, 5)).
   ...> map(user -> new Driver(3, 2).distanceTo(user)).
   ...> toList()
$.. ==> [3.0, 4.0]

jshell> Stream.<User>of(new Driver(1, 1), new Pedestrian(4, 5)).
   ...> map(user -> new Pedestrian(3, 2).distanceTo(user)).
   ...> toList()
$.. ==> [2.23606797749979, 3.1622776601683795]
```

Note the following:

- your solution should demonstrate the "Tell-Don't-Ask" principle;
- your solution should be extendable to other types of users with different distance computations based on offsets $x_A - x_B$ and $y_A - y_B$

1. [**4 marks**] Write the interface/class `User`, as well as any other dependencies.
2. [**2 marks**] Write the interfaces/classes for `Driver` followed by `Pedestrian`, as well as any other dependencies.

   You need not include `import` statements in your answers above.

You are given a `Count` context that keeps count of the number of function mappings where the input is different from the output.

```
import java.util.function.Function;

class Count<T> {
    private final T value;
    private final int count;

    private Count(T value, int count) {
        this.value = value;
        this.count = count;
    }

    static <T> Count<T> of(T value) {
        return new Count<T>(value, 0);
    }

    <R> Count<R> map(Function<? super T, ? extends R> mapper) {
        R r = mapper.apply(value);
        if (r.equals(value)) {
            return new Count<R>(r, count);
        }
        return new Count<R>(r, count + 1);
    }

    public String toString() {
        return value + ":" + count;
    }
}

jshell> Count.of(5)
$.. ==> 5:0

jshell> Stream.<Function<Integer,Integer>>of(x -> x * 1, x -> x + 1).
   ...> map(f -> Count.of(5).map(f)).toList()
$.. ==> [5:0, 6:1]
```

3. [**3 marks**] Write the `flatMap` method to aggregate the counts following the sample run below:

```
jshell> Count<Integer> count = Count.of(5).map(x -> x + 1)
count ==> 6:1

jshell> count.flatMap(x -> Count.of(x))
$.. ==> 6:1

jshell> count.flatMap(x -> Count.of(x).map(y -> y * 1))
$.. ==> 6:1

jshell> count.flatMap(x -> Count.of(x).map(y -> y + 1))
$.. ==> 7:2
```

4. [**2 marks**] Write the `equals` method to facilitate the testing of functor and monad laws.

5. [**3 marks**] Write `jshell` tests to demonstrate whether the functor and monad laws are obeyed. You should either

   - write **all** relevant tests to show that functor and monad laws are obeyed; or
   - write **one** test to show that not all functor/monad laws obeyed.

6. [**3 marks**] Given a quadratic equation (i.e. polynomial equation of degree 2) of the form:

$$ax^2 + b^x + c = 0$$

The solution of the equation can either be

- two real roots given by
$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
if the discriminant $b^2 - 4ac > 0$;
- one real root $\frac{-b}{2a}$ if the discriminant $b^2 - 4ac = 0$; or
- no real root if the discriminant $b^2 - 4ac < 0$.

Complete the following method `findRealRoots(double a, double b, double c)` that takes in the coefficients a, b and c of the quadratic equation $a^2x + bx + c == 0$, and returns a `List<Double>` comprising of the roots of the equations as shown below:

```
jshell> findRealRoots(4,19,-5) // two real roots
$.. ==> [-5.0, 0.25]

jshell> findRealRoots(-3,6,-3) // one real root
$.. ==> [1.0]

jshell> findRealRoots(4,12,10) // no real root
$.. ==> []
```

ANSWER:

```
List<Double> findRealRoots(double a, double b, double c) {
    return Stream.<Double>of(Math.sqrt(b*b - 4 * a * c))
        ...
}
```

Include the complete method definition of `findRealRoots` in your answer. The following sample run should be helpful.

```
jshell> Double d = Math.sqrt(-1)
d ==> NaN

jshell> d.isNaN()
$.. ==> true
```

This question follows from question 6.

Suppose the coefficients of the quadratic equation a, b and c takes some time to compute, but nonetheless computed independently from each other. Write an overloaded `findRealRoots` method that takes no arguments so that it takes minimal time to obtain the real roots of the equation.

Note that the methods `a()`, `b()` and `c()` are available that each returns the corresponding `double` coefficient value after some time.

```
List<Double> findRealRoots() {
   // Make use of methods a(), b(), c() that returns the coefficients.
}
```

You may make use of the overloaded method in question 6.

7. [**3 marks**] Define `findRealRoot()` so that it makes use of the `CompletableFuture` context.

8. [**3 marks**] Define `findRealRoot()` so that it makes use of the `Stream` context.

In this course, you are well aware that Java does not provide a proper immutable list; the only list that seems immutable is the one created using `List.of` where the list update methods (e.g. `add`) can be invoked, but throws an exception as a side effect. As such, much of our list processing is done via Java streams, but these do not come with the usual `List` methods like `add`, `get`, `remove`, `set`, etc.

In this question, we shall create our own `ImList` which encapsulates a `Stream` (`elems`) and the number of elements (`size`). You are given the `ImList` class below:

```
import java.util.List;
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;

record Entry<E>(int index, E elem) {}

class ImList<E> {
    private final Stream<Entry<E>> elems;
    private final int size;

    private ImList() {
        this.elems = Stream.<Entry<E>>of();
        this.size = 0;
    }

    static <E> ImList<E> of() {
        return new ImList<E>();
    }

    int size() {
        return this.size;
    }

    boolean isEmpty() {
        return this.size() == 0;
    }

    void forEach(Consumer<? super E> consumer) {
        this.elems.forEach(x -> consumer.accept(x.elem()));
    }
}
```

The `Entry` record (or class) allows us to track the correspondence between an element with its index. Any violation of the following will result in no marks for the question.

- only the following generic `Stream` methods are allowed:
  - factory methods `of` and two-argument `iterate`
  - transformation methods `map` and `flatMap`
  - `limit`, `filter`, `findFirst`, `sorted`
- do not include any other `import` statements.

9. [**3 marks**] Write a private constructor that takes in an appropriate `Stream` and a `size`.

   Next, write the factory method `of` that takes in the list of elements as a `List` and creates the corresponding `ImList`.

```
jshell> ImList.<Integer>of(List.of(1,2,3))
   ...> forEach(x -> System.out.print(x))
1
2
3

jshell> ImList.<Number>of(List.<Integer>of(1,2,3))
   ...> forEach(x -> System.out.print(x))
1
2
3
```

In this course, you are well aware that Java does not provide a proper immutable list; the only list that seems immutable is the one created using `List.of` where the list update methods (e.g. `add`) can be invoked, but throws an exception as a side effect. As such, much of our list processing is done via Java streams, but these do not come with the usual `List` methods like `add`, `get`, `remove`, `set`, etc.

In this question, we shall create our own `ImList` which encapsulates a `Stream` (`elems`) and the number of elements (`size`). You are given the `ImList` class below:

```
import java.util.List;
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;

record Entry<E>(int index, E elem) {}

class ImList<E> {
    private final Stream<Entry<E>> elems;
    private final int size;

    private ImList() {
        this.elems = Stream.<Entry<E>>of();
        this.size = 0;
    }

    static <E> ImList<E> of() {
        return new ImList<E>();
    }

    int size() {
        return this.size;
    }

    boolean isEmpty() {
        return this.size() == 0;
    }

    void forEach(Consumer<? super E> consumer) {
        this.elems.forEach(x -> consumer.accept(x.elem()));
    }
}
```

The `Entry` record (or class) allows us to track the correspondence between an element with its index. Any violation of the following will result in no marks for the question.

- only the following generic `Stream` methods are allowed:
  - factory methods `of` and two-argument `iterate`
  - transformation methods `map` and `flatMap`
  - `limit`, `filter`, `findFirst`, `sorted`
- do not include any other `import` statements.

10. [**4 marks**] Write the `addAll` method that takes in an `ImList` and returns a new `ImList` that concatenates this new immutable list to the end of the existing one.

    Next, write the `add` method that takes in an element and calls the `addAll` method.

```
jshell> ImList<Number> nums = ImList.<Number>of(List.<Number>of(1, 2))
nums ==> ImList@..

jshell> nums.addAll(ImList.<Integer>of(List.<Integer>of(3, 4))).
   ...> forEach(x -> System.out.print(x))
1
2
3
4
```

In this course, you are well aware that Java does not provide a proper immutable list; the only list that seems immutable is the one created using `List.of` where the list update methods (e.g. `add`) can be invoked, but throws an exception as a side effect. As such, much of our list processing is done via Java streams, but these do not come with the usual `List` methods like `add`, `get`, `remove`, `set`, etc.

In this question, we shall create our own `ImList` which encapsulates a `Stream` (`elems`) and the number of elements (`size`). You are given the `ImList` class below:

```
import java.util.List;
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;

record Entry<E>(int index, E elem) {}

class ImList<E> {
    private final Stream<Entry<E>> elems;
    private final int size;

    private ImList() {
        this.elems = Stream.<Entry<E>>of();
        this.size = 0;
    }

    static <E> ImList<E> of() {
        return new ImList<E>();
    }

    int size() {
        return this.size;
    }

    boolean isEmpty() {
        return this.size() == 0;
    }

    void forEach(Consumer<? super E> consumer) {
        this.elems.forEach(x -> consumer.accept(x.elem()));
    }
}
```

The `Entry` record (or class) allows us to track the correspondence between an element with its index. Any violation of the following will result in no marks for the question.

- only the following generic `Stream` methods are allowed:
  - factory methods `of` and two-argument `iterate`
  - transformation methods `map` and `flatMap`
  - `limit`, `filter`, `findFirst`, `sorted`
- do not include any other `import` statements.

11. [**4 marks**] Write the `get` method, followed by the `indexOf` method:
    - `Optional<E> get(int index)`
    - `int indexOf(Object obj)`

```
jshell> ImList.<Number>of(List.of(1, 2)).get(0)
$.. ==> Optional[1]

jshell> ImList.<Number>of(List.of(1, 2)).get(2)
$.. ==> Optional.empty

jshell> ImList.<Number>of(List.of(1, 2)).indexOf(2)
$.. ==> 1

jshell> ImList.<Number>of(List.of(1, 2)).indexOf(3)
$.. ==> -1
```

Since both `get` and `indexOf` involves searching the list, first define a private `find` method that takes in an appropriate `Predicate` and returns an `Optional` value. Next, define the `get` and `indexOf` methods by making use of the `find` method.

It is also interesting to note that since `Stream` can only be operated once, encapsulating a `Stream` does not allow us to repeatedly invoke `get` and `indexOf` on the same `ImList`. This is fine as we can always replace `Stream` with our own version of the infinite list `IFL`.

In this course, you are well aware that Java does not provide a proper immutable list; the only list that seems immutable is the one created using `List.of` where the list update methods (e.g. `add`) can be invoked, but throws an exception as a side effect. As such, much of our list processing is done via Java streams, but these do not come with the usual `List` methods like `add`, `get`, `remove`, `set`, etc.

In this question, we shall create our own `ImList` which encapsulates a `Stream` (`elems`) and the number of elements (`size`). You are given the `ImList` class below:

```
import java.util.List;
import java.util.Comparator;
import java.util.Optional;
import java.util.stream.Stream;
import java.util.function.Predicate;
import java.util.function.Function;
import java.util.function.Consumer;

record Entry<E>(int index, E elem) {}

class ImList<E> {
    private final Stream<Entry<E>> elems;
    private final int size;

    private ImList() {
        this.elems = Stream.<Entry<E>>of();
        this.size = 0;
    }

    static <E> ImList<E> of() {
        return new ImList<E>();
    }

    int size() {
        return this.size;
    }

    boolean isEmpty() {
        return this.size() == 0;
    }

    void forEach(Consumer<? super E> consumer) {
        this.elems.forEach(x -> consumer.accept(x.elem()));
    }
}
```

The `Entry` record (or class) allows us to track the correspondence between an element with its index. Any violation of the following will result in no marks for the question.

- only the following generic `Stream` methods are allowed:
  - factory methods `of` and two-argument `iterate`
  - transformation methods `map` and `flatMap`
  - `limit`, `filter`, `findFirst`, `sorted`
- do not include any other `import` statements.

12. [**6 marks**] Write the following methods in order:
    - `ImList<E> remove(int index)`
    - `ImList<E> set(index index, E elem)`
    - `ImList<E> sort(Comparator<? super E> cmp)`

```
jshell> ImList.<Integer>of(List.of(1, 2)).remove(0).
   ...> forEach(x -> System.out.println(x))
2

jshell> ImList.<Integer>of(List.of(1, 2)).remove(0).remove(1).
   ...> forEach(x -> System.out.println(x))
2

jshell> ImList.<Integer>of(List.of(1, 2)).remove(0).remove(0).
   ...> forEach(x -> System.out.println(x))

jshell> ImList.<Integer>of(List.of(1, 2)).
   ...> set(0,10).set(1,20).set(2,30).
   ...> forEach(x -> System.out.println(x))
10
20

jshell> ImList.<Integer>of(List.of(1, 2)).sort((x,y) -> y - x).
   ...> forEach(x -> System.out.println(x))
2
1
```

— END OF PAPER —