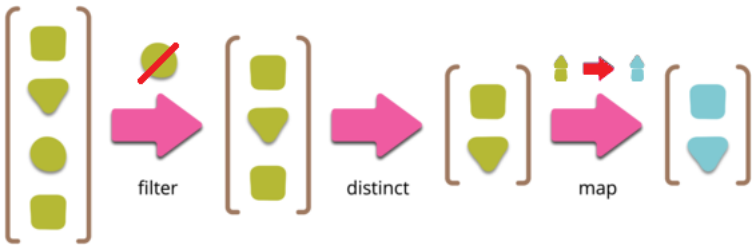
CS2030 Programming Methodology
Semester 2 2022/2023

8 & 9 March 2022
Problem Set #6
Functional Interfaces

You should now be very familiar with our `ImList` used as an immutable version of a list. The `ImList` can be extended further as a *collection pipeline*.

> *"Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next.*
>
> — *Martin Fowler*



In this problem set, we shall explore the additional pipeline operations in `ImList` that take in different functional interfaces. We shall also be writing various tests to test each of the method.

1. Let us start by exploring the `map` operation. Given an immutable list `ImList<T>` that is type-parameterized to `T`, the `map` method takes in a `Function<T,R>` and maps each element of type `T` to `R`.

   (a) By referring to the the Java API, find out the single abstract method (SAM) of the `Function` functional interface.

   (b) Using JShell, show how a lambda can be expressed and assigned to a variable of an appropriately type-parameterized `Function`. Also, show how the SAM can be invoked via the lambda.

   (c) Include the following `map` method in class `ImList<E>` that maps each element of the current list and returns a new `ImList` of mapped elements.

```
import java.util.function.Function;
...
    <R> ImList<R> map(Function<? super E, ? extends R> mapper) {
        ImList<R> newList = new ImList<R>();

        for (E t : this) {
            newList = newList.add(mapper.apply(t));
        }
        return newList;
    }
```

(d) Use JShell to test the `map` operation. Test the generality of the operation by exploiting the bounded wildcards in the definition of the `map` method

2. Now repeat the steps involved in question 1 for each of the following methods:

   i. `filter` which takes in a `Predicate<? super E>` and filters (let through) elements that satisfies the predicate;

```
import java.util.function.Predicate;
...
    ImList<E> filter(Predicate<? super E> pred) {
        ImList<E> newList = new ImList<E>();

        for (E t : this.elems) {
            if (pred.test(t)) {
                newList = newList.add(t);
            }
        }
        return newList;
    }
```

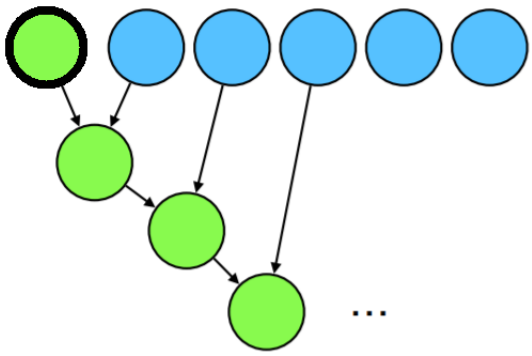
   ii. `forEach` which takes in a `Consumer<? super E>` and terminates the pipeline by performing an action on each element;

```
import java.util.function.Consumer;
...
    public void forEach(Consumer<? super E> consumer) {
        for (E t : this.elems) {
            consumer.accept(t);
        }
    }
```

   iii. `reduce` which takes in a seed value of type `U` and a two-argument (bi-function) of the form `BiFunction<? super U,? super E, ? extends U>`

```
import java.util.function.BiFunction;
...
<U> U reduce(U identity,
    BiFunction<? super U, ? super E, ? extends U> acc) {
    for (E t : this) {
        identity = acc.apply(identity, t);
    }
    return identity;
}
```

   Reduction starts with the seed value and iterates through the elements while performing the reduction. The reduction ends with a value of type `U` that is returned from the method.



3. Lastly, study the `flatMap` operation which takes in a `Function` whose resultant is an `ImList`.

```
<R> ImList<R> flatMap(
    Function<? super E, ? extends ImList<? extends R>> mapper) {
    ImList<R> newList = new ImList<R>();
    for (E t : this) {
        newList = newList.addAll(mapper.apply(t));
    }
    return newList;
}
```

Given the following implementation of a `Function`

```
jshell> Function<String, ImList<String>> f = x ->
   ...> new ImList<String>(List.<String>of("+","-","X")).
   ...>     map(y -> x + y)
f ==> $Lambda$15/0x00000001000a9440@51565ec2
```

(a) What is the outcome of `f.apply("A")`?

(b) What is the outcome of the following?

```
new ImList<String>(List.<String>of("A", "P")).flatMap(f)
```

(c) What happens if instead of `flatMap`, we use `map`?

```
new ImList<String>(List.<String>of("A", "P")).map(f)
```