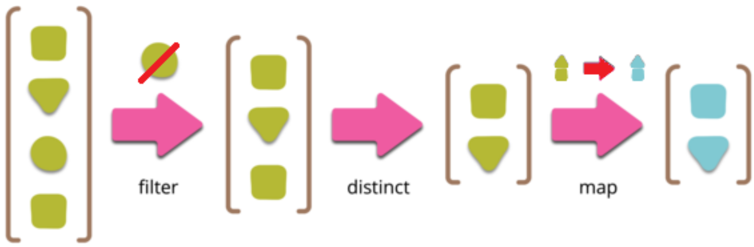
CS2030 Programming Methodology
Semester 1 2023/2024

11 & 12 October 2023
Problem Set #6 Suggested Guidance
Functional Interfaces

This problem set is meant as a follow up to lecture #8.
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 explore the additional pipeline operations in `ImList` that take in different functional interfaces, and write various tests to test each of the method.

1. Let us start by exploring the `map` operation. Given an immutable list `ImList<E>`, the `map` method takes in a `Function<? super E, ? exends R>` and maps each element of type `E` to `R`.

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

   ```
   interface Function<T, R> {
      R apply(T t);
   }
   ```

   (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.

   ```
   jshell> Function<String,Integer> f = x -> x.length()
   f ==> $Lambda$15/0x00000008000a9440@735b5592

   jshell> f.apply("one")
   $.. ==> 3
   ```

   (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) {
        List<R> newList = new ArrayList<R>(); // create empty ArrayList
        for (E elem : this) {
            newList.add(mapper.apply(elem)); // add to mutable ArrayList
        }
        return new ImList<R>(newList); // wrap ImList around ArrayList
    }
```

(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

```
jshell> ImList<String> list = new ImList<String>(List.of("one","two","three"))
list ==> [one, two, three]

jshell> Function<String,Integer> f = x -> x.length()
f ==> $Lambda$20/0x0000000800c0a208@5b6f7412

jshell> list.map(f) // E bound to String; R bound to Integer
$.. ==> [3, 3, 5]

jshell> Function<Object,Integer> g = x -> x.hashCode()
g ==> $Lambda$21/0x0000000800c0a858@27bc2616

jshell> list.map(g) // E bound to String; R bound to Integer
$.. ==> [110182, 115276, 110339486]

jshell> ImList<Number> newList = list.map(g) // E bound to String; R to Number
newList ==> [110182, 115276, 110339486]

jshell> newList.add(0.1)
$.. ==> [110182, 115276, 110339486, 0.1]
```

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) {
        List<E> newList = new ArrayList<E>();
        for (E elem : this) {
            if (pred.test(elem)) {
                newList.add(elem);
            }
        }
        return new ImList<E>(newList);
    }
```

(a)
```
interface Predicate<T> {
   boolean test(T t);
}
```
(b)
```
jshell> Predicate<String> pred = x -> x.length() == 3
x ==> $Lambda$20/0x0000000800c09a08@12edcd21

jshell> pred.test("one")
$.. ==> true
```
(c) *As given.*

(d)
```
jshell> ImList<String> list = new ImList<String>(List.of("one","two","three"))
list ==> [one, two, three]

jshell> list.filter(x -> x.length() == 3)
$.. ==> [one, two]

jshell> Predicate<Object> pred = x -> x.hashCode() < 1_000_000
pred ==> $Lambda$24/0x0000000800c0b550@1ddc4ec2

jshell> list.filter(pred)
$.. ==> [one, two]
```

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 elem : this) {
            consumer.accept(elem);
        }
    }
```

(a)
```
interface Consumer<T> {
   void accept(T t);
}
```
(b)
```
jshell> Consumer<String> consumer = x -> System.out.println("[" + x + "]")
consumer ==> $Lambda$25/0x0000000800c0bbb0@30dae81

jshell> consumer.accept("one")
[one]
```
(c) *As given. Note that since* `ImList` *implements the* `Iterable` *interface,* `forEach` *is already defined in* `Iterable` *as a* `default` *method.*

(d)
```
jshell> ImList<String> list = new ImList<String>(List.of("one","two","three"))
list ==> [one, two, three]

jshell> list.forEach(x -> System.out.print(x + " "))
one two three
```

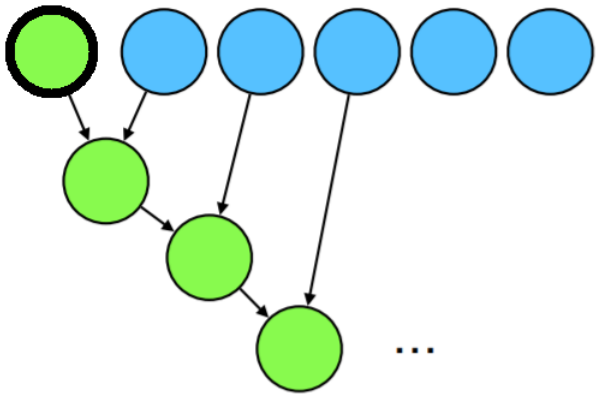
```
jshell> Consumer<Object> consumer = x ->
   ...> System.out.print(x.hashCode() + " ")
consumer ==> $Lambda$28/0x0000000800c0f000@378fd1ac

jshell> list.forEach(consumer)
110182 115276 110339486
```

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 elem : this) {
        identity = acc.apply(identity, elem);
    }
    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.



(a)
```
interface BiFunction<T,U,R> {
   R apply(T t, U u);
}
```

(b)
```
jshell> BiFunction<String,Integer,Integer> bif = (x,y) -> x.length() + y
bif ==> $Lambda$29/0x0000000800c0f438@7e6cbb7a

jshell> bif.apply("one", 2)
$.. ==> 5
```

(c) *As given.*

(d)
```
jshell> ImList<String> list = new ImList<String>(
   ...> List.of("one","two","three"))
list ==> [one, two, three]
```

```
jshell> list.reduce(1, (x,y) -> x * y.length())
$.. ==> 45

jshell> BiFunction<Object,Object,Integer> bif =
   ...> (x,y) -> x.hashCode() + y.hashCode()
bif ==> $Lambda$32/0x0000000800c10000@4d405ef7

jshell> Number n = 1
n ==> 1

jshell> list.reduce(n, bif) // E bound to String, U bound to Number
$.. ==> 110564945
```

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)
```

```
jshell> f.apply("A")
$.. ==> [A+, A-, AX]

jshell> new ImList<String>(List.<String>of("A", "P")).
   ...> flatMap(f)
$.. ==> [A+, A-, AX, P+, P-, PX]
```

The following illustrates the use of bounded wildcards in `flatMap`

```
jshell> ImList<String> strings = new ImList<String>(
   ...> List.of("one", "two", "three"))
strings ==> [one, two, three]

jshell> Function<Object, ImList<Integer>> f1 =
   ...>    x -> new ImList<Integer>(List.of(1,2,3))
f1 ==> $Lambda$20/0x0000000800c0a208@5b6f7412

jshell> ImList<Number> list1 = strings.flatMap(f1)
list1 ==> [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

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

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

Using `map` *instead of* `flatMap` *results in every element of the list mapped to another* `ImList`*.*

```
jshell> new ImList<String>(List.<String>of("A", "P")).map(f)
$.. ==> [[A+, A-, AX], [P+, P-, PX]]
```