Abstraction

Student name:	TA signature:

Concepts Check off understood concepts, connect related concepts, label connections

Add the following three concepts to the map: **Filter**, **Similarity**, and **Higher-Order Function**. Connect them and connect everything else as well.

Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

`map` • `filter` • `reduce`

Abstraction – Turn Differences into Parameters

Code duplication is bad. If we see two pieces of code that are the same, we should eliminate one. But what if the two pieces have a lot of **similarities**, but do have a few **differences**?

Abstraction over Expressions – Remember Workbook 3?

Here is some code that builds two houses side-by-side:

```
beside(redHouse(), blueHouse())
```

And here are two methods that construct a house:

```
public static Graphic redHouse() {
  final int width = 200;
  final int height = 200;
  return above(
    equilateralTriangle(width, RED),
    rectangle(width, height, RED)
  );
}
```

```
public static Graphic blueHouse() {
  final int width = 200;
  final int height = 100;
  return above(
    equilateralTriangle(width, BLUE),
    rectangle(width, height, BLUE)
  );
}
```

Do you **feel the pain**? You should! So many **similarities**, and so few **differences**! Highlight the differences in the above code.

Let's eliminate the code duplication. Replace the two methods above with a single method that can deal with both requests. To do this, simply turn the differences (between the two methods) into parameters:

```
public static Graphic
```

Now use your new method to build the two side-by-side houses:

```

```

Do you feel the joy? That's the joy of abstraction!

In this specific example, you **abstracted over different expressions**. You wrote a method that can construct a house of ***any*** color and ***any*** height. The specific color and height can be provided to the method via its parameters. This is much more generally usable. You (or others) can now use your method to easily build all kinds of different houses!

Abstraction over Types

Here is some code that computes the number of graphics-color combinations:

```
numberOfGraphics(graphics) * numberOfColors(colors)
```

And here are the two methods that do the counting:

```
public static int
numberOfGraphics(Sequence<Graphic> gs)
{
  return isEmpty(gs)
    ? 0
    : 1 + numberOfGraphics(rest(gs));
}
```

```
public static int
numberOfColors(Sequence<Color> cs)
{
  return isEmpty(cs)
    ? 0
    : 1 + numberOfColors(rest(cs));
}
```

Do you feel the pain? You should! So many similarities, and so few differences! Highlight the differences in the above code.

Let's eliminate the code duplication. Replace the two methods above with a single method that can deal with both requests. To do this, simply turn the differences (between the two methods) into **type** parameters:

```
public static <
```

Now use your new method to compute the number of graphics-color combinations:

```

```

Do you feel the joy? That's the joy of abstraction! In this specific example, you **abstracted over different types**. You wrote a method that can count the length of a sequence of ***any*** element **type**. The specific type can be provided to the method via its type parameter. This is much more generally usable. You (or others) can now use your method to easily count the length of any type of sequence.

Abstraction over Behaviors

Here is some code that reports (we don't really care what `report` does) the sum and product of a sequence of values:

```
report(sum(values), product(values))
```

And here are the two methods that compute the statistics:

`public static int`
`sum(Sequence<Integer> vs)`
`{`
`  return isEmpty(vs)`
`    ? 0`
`    : first(vs) + sum(rest(vs));`
`}`
	`public static int`
	`product(Sequence<Integer> vs)`
	`{`
	`  return isEmpty(vs)`
	`    ? 1`
	`    : first(vs) * product(rest(vs));`
	`}`

Do you **feel the pain**? You should! So many **similarities**, and so few **differences**! Highlight the differences in the above code.

Let's eliminate the code duplication. Replace the two methods above with a single method that can deal with both requests. To do this, simply—**SIMPLY?? YOU MUST BE KIDDING ME!!**—turn the differences (between the two methods) into parameters:

```
public static
```

Now use your new method to compute sum and product for your report:

```
report(
```

Ok, it's highly likely that you are unable to solve the above challenge initially.

That's perfectly fine! Continue with the next page (keeping in mind the goal of solving the above challenge). At the end of the worksheet, return here to succeed.

So, you're back, and you did it? Do you feel the joy? That's the joy of abstraction! In this specific example, you **abstracted over different behaviors**. You wrote a method that can do ***any*** kind of **aggregation computation** over a sequence of integers. The specific kind of aggregation, and the value to return if the sequence is empty, can be provided to the method via its parameters. This is much more generally usable. You (or others) can now use your method to easily do lots of different kinds of aggregations.

Functions as Values

We have seen different types of values, such as numbers (`int`, `double`), truth values (`boolean`), text (`String`), colors (`Color`), graphics (`Graphic`), and data structures (`Sequence`).

What can we **do** with a value (of any of those types)?

- ☐ pass a value via a parameter to a **method**
- ☐ return a value as a result from a **method**
- ☐ pass a value as an operand to an **operator**
- ☐ receive a value as a result from an **operator**

If a graphic or a color can be a value, couldn't a **function** also be a value? If a function could be a value, we could implement different functions (e.g., `add`, `multiply`) that have the same type (type signature), but have different behaviors. Then we could pass the corresponding function via a parameter to our method!

Of course, in Java that can be done! But we need to solve three challenges:

1. How to specify the **type** of a function
2. How to **create** a function object
3. How to **call** the function behind a function object

Assume we want to write a method named `compute`, which takes two parameters: a function `f` to execute, and a double `x`. When we call `compute`, we pass a **function object** as the first argument, and a double as the second argument. `compute` calls the given function (whatever it is) and passes the given `x` to that function as an argument. It then takes the result of the function and returns it:

```
public static double compute(TYPE f, double x) {
  return CALL f(x);
}
```

```
compute(CREATE FUNCTION OBJECT, 3.14)
```

Let's tackle the first challenge: What is the type of `f`? Looking at the expression inside `compute`, we see that `f` seems to be a method that takes a `double` and returns a `double`. In a clean programming language, we could write the type of `f` as `double` ⟶ `double`. Unfortunately, Java does not have such an arrow notation for function types.

Instead, we have to use types provided by some library. JTamaro provides `Function1` (function with one parameter) and others. Complete the table:

Parameters		Return	Type
Color	⟶	Graphic	Function1<Color,Graphic>
Integer	⟶	Double	Function1<Integer,Double>
Double, Double	⟶	Boolean	Function2<Double,Double,Boolean>
	⟶	String	Function0<String>
int	⟶	int	Function1<Integer,Integer>
Boolean, int	⟶	Double	
Sequence<Graphic>	⟶	Graphic	
Double, Double	⟶	Double	

Commented [MH1]: Start with creating a class with an instance method (based on the previous lab, with expressions, and then naturally end up with functional interface and calling apply)

Commented [J2]: Argument or parameter?

So, we solved the first challenge: how to specify the type:

```
public static double compute(Function1<Double,Double> f, double x) {
  return CALL f(x);
}
```

```
compute(CREATE FUNCTION OBJECT, 3.14)
```

The second challenge is how to **create** a function object of that type, so we can pass it to compute. One way to do this in Java is with **method references**.

Assume we already have the following classes and methods:

```
public class Num {
  public static double abs(double v) {
    return v < 0 ? -v : v;
  }
  public static double inc(double v) {
    return v + 1;
  }
  public static int zero() {
    return 0;
  }
}
public class Compare {
  public static boolean isZero(int a) {
    return a == 0;
  }
  public static boolean gt(double a, double b) {
    return a > b;
  }
}
public class Visual {
  public static Graphic rotate180(Graphic g) {
    return rotate(180, g);
  }
  public static Graphic underlay(Graphic a, Graphic b) {
    return overlay(b, a);
  }
}
```

We can create a function object by using a method reference, which consists of the class name, `::`, and the method name. Given the above code, `Num::abs` is an expression that produces a function object for the `abs` method in class `Num`.

Commented [J3]: s/of/for/

Complete the following table with matching function types:

Method Reference	Function Type
`Num::abs`	`Function1<Double,Double>`
`Num::inc`	
`Num::zero`	
`Compare::isZero`	
`Compare::gt`	
`Visual::rotate180`	
`Visual::underlay`	

So, we solved the first two challenges: how to specify the **type**, and how to **create** a function object:

```
public static double compute(Function1<Double,Double> f, double x) {
  return CALL F(X);
}

compute(Num::abs, 3.14)
```

The third and last challenge is how to **call** the function that's represented by a function object.

To call a function represented by a function object of type `Function0`, `Function1`, `Function2`, `Function3`, ..., you invoke its `apply` method as follows:

Type of f	Call
`Function0<Double>`	`f.apply()`
`Function1<Double,Double>`	`f.apply(3.14)`
`Function2<Double,Double,Double>`	`f.apply(3.14, 1.1)`
`Function3<Double,Double,Double,Double>`	`f.apply(3.13, 1.1, 0.4)`

Note that the last of the type parameters corresponds to the return type; thus, we have one more type parameter than function parameters.

If `g` has type `Function2<Integer,Double,Boolean>`, which of the following calls are ok? If a call is ok, what is the return type?

Call	Ok?	Return type
`g.apply()`	☐	
`g.apply("3")`	☐	
`g.apply(3, 1.1)`	☐	
`g.apply(3, 1.1, true)`	☐	

Note: The return type is given by the type of `g`!

So, we solved all three challenges: how to specify the **type**, how to **create** a function object, and how to **call** the function behind the function object:

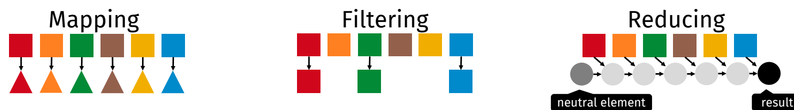
```
public static double compute(Function1<Double,Double> f, double x) {
  return f.apply(x);
}

compute(Num::abs, 3.14)
```

Note that we skipped quite a few details. You will learn about them later. For now, you know enough to implement your own functions like `compute`, which take another function as a parameter and call it. These kinds of functions, which "play" with other functions, are called **higher-order functions**.

Mapping, Filtering, and Reducing Once And For All

We previously learned about three patterns of repetitive computations:



We used those to solve many different problems, e.g., to turn angles or colors into graphics, or to compose multiple graphics into one.

This was a bit painful. We had to write a similarly-looking recursive method every time we wanted to do some repetitive computation. If we see similar code, we should feel the urge to **refactor** it, to **eliminate code duplication**.

A General Mapping Function

Here are two special-purpose **mapping** methods:

```
public static Sequence<Color> anglesToColors(Sequence<Integer> angles) {
  return isEmpty(angles)
    ? empty()
    : cons(
        angleToColor(first(angles)),
        anglesToColors(rest(angles))
      );
}
```

```
public static Sequence<Graphic> stringsToGs(Sequence<String> strings) {
  return isEmpty(strings)
    ? empty()
    : cons(
        stringToG(first(strings)),
        stringsToGs(rest(strings))
      );
}
```

What are the differences between the two mapping methods?

- ☐ The name of the methods – m
- ☐ The name of the parameters – p
- ☐ The type of the element of the sequence passed in – I
- ☐ The type of the element of the sequence returned – O
- ☐ The behavior applied to individual elements – f

If we keep what's similar and introduce a black label for each difference, we get:

```
public static Sequence<O> m(Sequence<I> p) {
  return isEmpty(p)
    ? empty()
    : cons(
        f(first(p)),
        m(rest(p))
      );
}
```

We want to abstract; to keep the similar code, and to turn differences into parameters (**type parameters** and **value parameters**). We want to develop a single method that can map from any sequence to any other sequence using any possible mapping function.

Changing the **name** of our method and the name of its parameters does not affect any values, types, or behaviors. Thus, we do not need to introduce parameters for the names. We can pick a good name that makes sense in general:

```
public static Sequence<O> map(Sequence<I> sequence) { … }
```

Our general-purpose `map` method should be able to map from any type of elements (indicated with the black label I) to any (possibly different) type of elements (labeled O). E.g., from `Integer` to `Color`, or from `String` to `Graphic`, like the two special-purpose methods `anglesToColors` and `stringsToGs`. Thus, we need to introduce **type parameters** for I and O:

```
public static <A,B> Sequence<B> map(Sequence<A> sequence) { … }
```

Our general-purpose `map` method now looks like this:

```
public static <A,B> Sequence<B> map(Sequence<A> sequence) {
  return isEmpty(sequence)
    ? empty()
    : cons(
        f(first(sequence)),
        map(rest(sequence))
      );
}
```

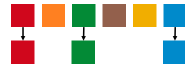
The only remaining difference is f. f stands for a function (`angleToColor` and `stringToG` in our two specific examples). Thus, we can introduce a **value parameter** (to allow passing a function object) to parameterize this difference:

```
public static <A,B> Sequence<B> map(
  Function1<A,B> mapper, Sequence<A> sequence) {
  return isEmpty(sequence)
    ? empty()
    : cons(
        mapper.apply(first(sequence)),
        map(mapper, rest(sequence))
      );
}
```

This is it! We have a general-purpose `map` method. It can map from whatever sequence to whatever other sequence using whatever mapping function!

Assume `angleToColor` and `stringToG` are methods in class `Demo`. Refactor the calls using the general purpose `map` function:

Using special-purpose function	Using general-purpose map
`anglesToColors(range(0, 360))`	
`stringsToGs(of("Hi", "Ciao!"))`	



A General Filter Function

Here are two special-purpose **filtering** methods:

```
public static Sequence<Double> positives(Sequence<Double> reals) {
  return isEmpty(reals)                         // termination condition
    ? empty()                                   // base case
    : (                                         // recursive case
      first(reals) >= 0                                    // predicate
        ? cons(first(reals), positives(rest(reals)))       // keep
        : positives(rest(reals))                           // drop
      );
}

public static Sequence<Integer> evens(Sequence<Integer> numbers) {
  return isEmpty(numbers)                       // termination condition
    ? empty()                                   // base case
    : (                                         // recursive case
      first(numbers) % 2 == 0                              // predicate
        ? cons(first(numbers), evens(rest(numbers)))       // keep
        : evens(rest(numbers))                             // drop
      );
}
```

What are the differences between the two filtering methods?

- ☐ The name of the methods – m
- ☐ The name of the parameters – a
- ☐ The type of the element of the sequence – T
- ☐ The predicate applied to individual elements – p

If we keep what's similar and introduce a black label for each difference, we get:

```
public static Sequence<T> m(Sequence<T> a) {
  return isEmpty(a)                             // termination condition
    ? empty()                                   // base case
    : (                                         // recursive case
      p                                                    // predicate
        ? cons(first(a), m(rest(a)))                       // keep
        : m(rest(a))                                       // drop
      );
}
```

Let's first pick general names for the method and the parameter:

```
public static Sequence<T> filter(Sequence<T> sequence) { … }
```

Now let's introduce a **type parameter**, to parameterize the type of elements the filter can process. Note that for filtering, the element type of the parameter and the return type is the same, so we only need one type parameter.

```
public static <E> Sequence<E> filter(Sequence<E> sequence) { … }
```

Our general-purpose `filter` method now looks like this:

```
public static <E> Sequence<E> filter(Sequence<E> sequence) {
  return isEmpty(sequence)
    ? empty()
    : (
      p
        ? cons(first(sequence), filter(rest(sequence)))
        : filter(rest(sequence))
      );
}
```

The predicate `p` stands for an entire expression. Here are the two predicates from the two special-purpose filtering methods:

```
first(sequence) >= 0      // first element of the sequence is positive
first(sequence) % 2 == 0  // first element of the sequence is even
```

We create a function for each of these predicate expressions. Both expressions need access to the sequence (so they can look at its first element):

```
public static boolean positive(Sequence<Double> sequence) {
  return first(sequence) >= 0;
}
public static boolean even(Sequence<Integer> sequence) {
  return first(sequence) % 2 == 0;
}
```

Now we introduce a parameter to our `filter` method, so we can pass a filter predicate:

```
public static <E> Sequence<E> filter(
  Function1<Sequence<E>,Boolean> predicate, Sequence<E> sequence) {
  return isEmpty(sequence)
    ? empty()
    : (
      predicate.apply(sequence)
        ? cons(first(sequence), filter(predicate, rest(sequence)))
        : filter(predicate, rest(sequence))
      );
}
```

The predicate must be an arity-1 (1 parameter) function, taking a parameter of type `Sequence<E>` (the same type of sequence the `filter` method gets and returns). The predicate has a return type of `Boolean` because it has to return whether to keep or to drop the element.

We are pretty much done. We could use our code as is. But our design is not ideal.

Do `filter` predicates REALLY need access to the sequence? Isn't a predicate supposed to only see an individual element (and not look at the rest)?

Let's refactor our solution, so the predicates look as follows:

```
public static boolean positive(double element) {
  return element >= 0;
}

public static boolean even(int element) {
  return element % 2 == 0;
}
```

That's cleaner, and it makes it **easier** to create your own predicates in the future!

Now refactor the filter method so it works with these new, more appropriate predicates:

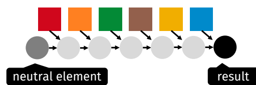
```
public static <E> Sequence<E> filter(
                                , Sequence<E> sequence)
  return isEmpty(sequence)
    ? empty()
    : (

        ? cons(first(sequence), filter(predicate, rest(sequence)))
        : filter(predicate, rest(sequence))
      );
}
```

This is it! We have a general-purpose `filter` method. It can filter whatever sequence we have using whatever filtering predicate!

Assume `positive` and `even` are methods in class `Demo`. Refactor the calls using the general-purpose `filter` function:

Using special-purpose function	Using general-purpose filter
`positives(of(-1.0, 2.0))`	
`evens(range(10))`	



A General Reduce Function

Here are two special-purpose **reducing** methods:

```
public static Integer product(Sequence<Integer> numbers) {
  return isEmpty(numbers)
    ? 1
    : first(numbers) * product(rest(numbers));
}
```

```
public static String join(Sequence<String> strings) {
  return isEmpty(strings)
    ? ""
    : first(strings) + join(rest(strings));
}
```

What are the differences between the two reducing methods?

- ☐ The name of the methods – m
- ☐ The name of the parameters – a
- ☐ The type of the element of the sequence – T
- ☐ The return type of the method - R
- ☐ The combining operation – c
- ☐ The neutral element – e

If we keep what's similar and introduce a black label for each difference, we get:

```
public static R m(Sequence<T> a) {
  return isEmpty(a)
    ? e
    : first(a)  c  m(rest(a));
}
```

Let's first pick general names for the method and the parameter:

```
public static R reduce(Sequence<T> sequence) { … }
```

Now let's introduce **type parameters**, to parameterize the type of elements the method can process, and the type of value it produces. Note that for reduction, the element type of the sequence and the type of value it produces do not need to be the same (although they often are the same).

```
public static <A,B> B reduce(Sequence<A> sequence) { … }
```

Our general-purpose `reduce` method now looks like this:

```
public static <A,B> B reduce(Sequence<A> sequence) {
  return isEmpty(sequence)
    ? e
    : first(sequence)  c  reduce(rest(sequence));
}
```

The neutral elements e from the two special-purpose reduction methods were:

```
1    // neutral element of product
""   // neutral element of join
```

We can add a parameter so callers can provide their desired neutral element:

```
public static <A,B> B reduce(B neutralElement, Sequence<A> sequence) {
  return isEmpty(sequence)
    ? neutralElement
    : first(sequence)  c  reduce(neutralElement, rest(sequence));
}
```

Here are the expressions involving the combining operations c from the two special-purpose reduction methods:

```
first(numbers) * product(rest(numbers))
first(strings) + join(rest(strings))
```

The combining operation really is just the `*` and the `+` operator. While we can create a function object using a method reference, Java does not have a similar concept for operators (there is no such thing as an "operator reference"). However, we can wrap an operator in a method, like this:

```
public static int mul(int a, int b) {
  return a * b;
}
public static String concat(String a, String b) {
  return a + b;
}
```

Then we can use method references (e.g., `Demo::mul` and `Demo::concat`) to refer to those operations. Now we can add another parameter to `reduce`, so callers can provide their desired combining operation in the form of a function object:

```
public static <A,B> B reduce(
  B neutralElement, Function2<A,B,B> combiner, Sequence<A> sequence) {
  return isEmpty(sequence)
    ? neutralElement
    : combiner.apply(
        first(sequence),
        reduce(neutralElement, combiner, rest(sequence)));
}
```

This is it! We have a general-purpose `reduce` method. It can reduce whatever sequence we have using whatever combining function and neutral element!

Assume `mul` and `concat` are methods in class `Demo`. Refactor the calls using the general purpose `reduce` function:

Using special-purpose function	Using general-purpose reduce
`product(of(10, 20, 30))`	
`join(of("Ciao", "Hi", "Ho"))`	

Now you're ready to **go back to page 4**, and solve the "Abstraction over Behaviors" challenge involving a `sum` and a `product` method.