Functions

Functions are nothing but a mathematical concept. That is they are a mapping from a source set (called the domain) to a resulting set(called the range or co-domain). A function has one necessary condition, that each element in the domain should map onto one and only one element in the range.

$f(x) = x + 1$ for $x > 0$

Functions dont perform any computation, that is f(x) above does not add 1 to x. It is only equivalent to (x+1). That is you can replace x+1 whenever you come across f(x).

Inverse Functions

A function having a domain A and a range B, may or may not have an inverse function. An inverse function is a function which maps from B to A.
$f(x) = x + 1 \rightarrow$ Successor
$fx() = x - 1 \rightarrow$ Predecessor is not a function in N where N is the set of positive numbers including 0.

$f(x) = 2 * x$

Partial Functions

Partial functions are functions which don’t define a relationship for all elements in the domain.
$f(x) = 1/x$

By adding an error to the co-domain partial functions can be converted to total functions.

Functions With Multiple Args

There's no such thing as a function of several arguments.

A function is a relation between a source set and a target set. It isn't a relation between two or more source sets and a target set. A function can't have several arguments.
But the product of two sets is itself a set, so a function from such a product of sets into a set may appear to be a function of several arguments.
f(x,y) = x + y is a function from (N X N) to N. It has only one argument which is a set of elements from (N X N).
(N X N) - is the set of all pairs of numbers. Generally modeled in FP as a *Pair* which is a special case of a *Tuple*.

Function Currying

The function f(3, 5) might be considered as a function from N to a set of functions of N.
So f(x,y) = x + y can be represented as
f(x)(y) = g(y)
Where g(y) = x + y
f(x) = g → which means that the result of applying the function f to the argument *x* is a new function g.
Applying this g function to *y* gives:
g(y) = x + y
When applying g, x is no longer a variable, it is a constant.
f(3)(5) = g(5) = 3 + 5 = 8

Function Currying

The only new thing here is that the codomain of f is a set of functions instead of a set of numbers. The result of applying f to an integer is a function. The result of applying this function to an integer is an integer

Functional Methods

A method can be functional if it respects the requirements of a pure function:

- It must not mutate anything outside the function. No internal mutation may be visible from the outside.
- It must not mutate its argument.
- It must not throw errors or exceptions.
- It must always return a value.
- When called with the same argument, it must always return the same result.

Functional Java - apply

```
public interface Function {
  int apply(int arg);
}

Function triple = new Function() {
 @Override
   public int apply(int arg) {
       return arg * 3;
    }
};
```

Functional Java - apply

```
Function square = new Function() {
 @Override
  public int apply(int arg) {
     return arg * arg;
   }
};
square.apply(triple.apply(2)); // composing functional applications
```

Functional Java - compose

Function composition is a binary operation on functions, just as addition is a binary operation on numbers.

```
Function compose(final Function f1, final Function f2) {
  return new Function() {
    @Override
    public int apply(int arg) {
      return f1.apply(f2.apply(arg));
    }
  };
}
compose(triple, square).apply(3)
```

Polymorphic Functions

```
public interface Function<T, U> {
  U apply(T arg);
}
Function<Integer, Integer> triple = new Function<Integer, Integer>() {
  @Override
  public Integer apply(Integer arg) {
    return arg * 3;
  }
};
```

Polymorphic Functions

```
Function<Integer, Integer> square = new Function<Integer, Integer>() {
 @Override
 public Integer apply(Integer arg) {
  return arg * arg;
 }
};
```

Write the compose method by using these two new functions.

Polymorphic Functions

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                          Function<Integer, Integer> f2) {
  return new Function<Integer, Integer>() {

    @Override
    public Integer apply(Integer arg) {
      return f1.apply(f2.apply(arg));
    }
  };
}
```

Lambdas

```
Function<Integer, Integer> triple = x -> x * 3;
Function<Integer, Integer> square = x -> x * x;
```

Write a new version of the compose method by using lambdas

```
static Function<Integer, Integer> compose(Function<Integer, Integer> f1,
                                          Function<Integer, Integer> f2) {
  return arg -> f1.apply(f2.apply(arg));
}
```

Multi-Arity Functions

There are no functions with multiple arguments.
f(x,y) -> x + y
(N X N) -> N
Arguments can be applied one by one, each application of one argument returning a new function, except for the last one → Currying

Function for adding two numbers signature
Function<Integer, Function<Integer, Integer>> → You apply a function to the first argument which returns a function.
Equivalent to
Integer -> (Integer -> Integer) → <Integer, <Integer, Integer>>(Java way for this)

Multi-Arity Functions

Write a function to add two numbers ?

Write a function to multiply two numbers ?

Define a generic interface so that you can extend addition and multiplication or any function which takes 2 integers and returns an integer.

Multi-Arity Functions

```
Function<Integer, Function<Integer, Integer>> add = x -> y -> x + y;

public interface BinaryOperator extends  Function<Integer, Function<Integer,
Integer>> {}

BinaryOperator add = x -> y -> x + y;

BinaryOperator mult = x -> y -> x * y;
```

Applying Curryed Functions

You apply the function to the first argument, and then apply the result to the next argument, and so on until the last one.
Apply the add function → add.apply(3).apply(5) → 8

Higher Order Functions

Function<Integer, Integer> triple = x -> x * 3;
Function<Integer, Integer> square = x -> x * x;

Write a function to compose triple and square ?

Type - Function<Integer, Integer> -->T
First Arg - T
Second Arg - T
Return Type - T
Function<T, Function<T, T>>

Higher Order Functions

Function<Function<Integer, Integer>,Function<Function<Integer, Integer>,
Function<Integer, Integer>>>

Implementation → x -> y -> z -> x.apply(y.apply(z));

Function<Function<Integer, Integer>,
Function<Function<Integer, Integer>, Function<Integer, Integer>>> compose =
x -> y -> z -> x.apply(y.apply(z));
Function<Integer, Integer> f = compose.apply(square).apply(triple);
f.apply(2) → output ??

Higher Order Functions

****Assignment - Write a polymorphic version of compose.

Java Issues - Remember Java doesn't allow standalone generic properties. To be generic, a property must be created in a scope defining the type parameters. Only classes, interfaces, and methods can define type parameters, so you have to define your property inside one of these elements.

Variance

Variance describes how parameterized types behave in relation to subtyping.
Covariance means that Matcher<Red> is considered a subtype of Matcher<Color> if Red is a subtype of Color.

An Integer is a subtype of Object,
Is List<Integer> a subtype of List<Object> .
A List<Integer> is an Object, but it is not a List<Object>.
Function<Integer, Integer> is not a Function<Object, Object>

Method References

Method references is a syntax that can be used to replace a lambda when the lambda implementation consists of a method call with a single argument

Function<Double, Double> sin = x -> Math.sin(x);
Can be replaced with
Function<Double, Double> sin = Math::sin;
using method references.

Used to make a function out of a method using method references.

Closures

```
public void aMethod() {
  double taxRate = 0.09;
  Function<Double, Double> addTax  = price -> price + price * taxRate;
}
```

addTax closes over the *local variable* taxRate.

```
public void aMethod() {
  double taxRate = 0.09;
  Function<Double, Double> addTax  = price -> price + price * taxRate;
  taxRate = 0.13;
}
```

Method to Currying

Convert the following method into a curried function:

```
<A, B, C, D> String func(A a, B b, C c, D d) {
  return String.format("%s, %s, %s, %s", a, b, c, d);
}
```