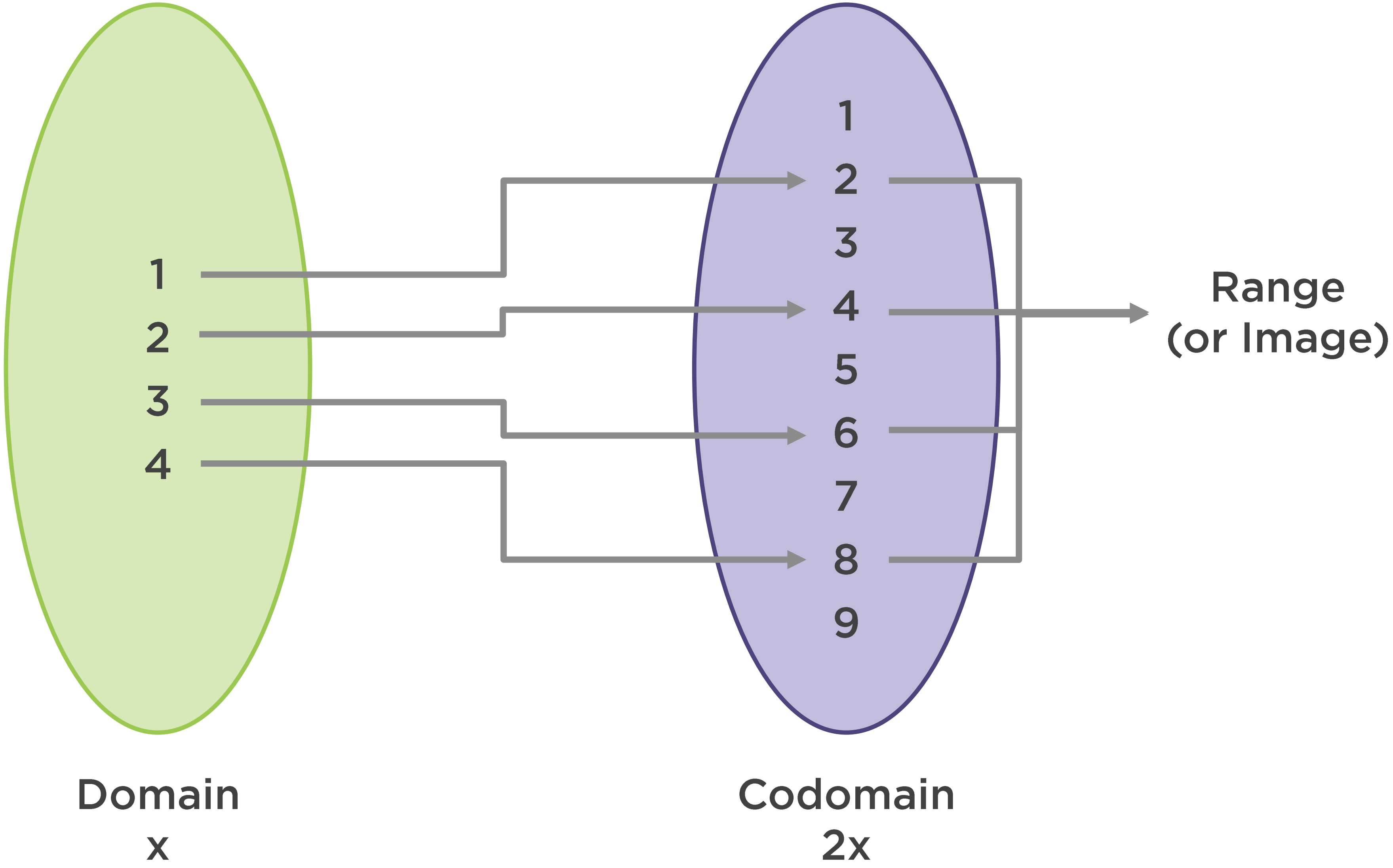
Building Complex Functionality by Composing Functions

Esteban Herrera
JAVA ARCHITECT

@eh3rrera   www.eherrera.net

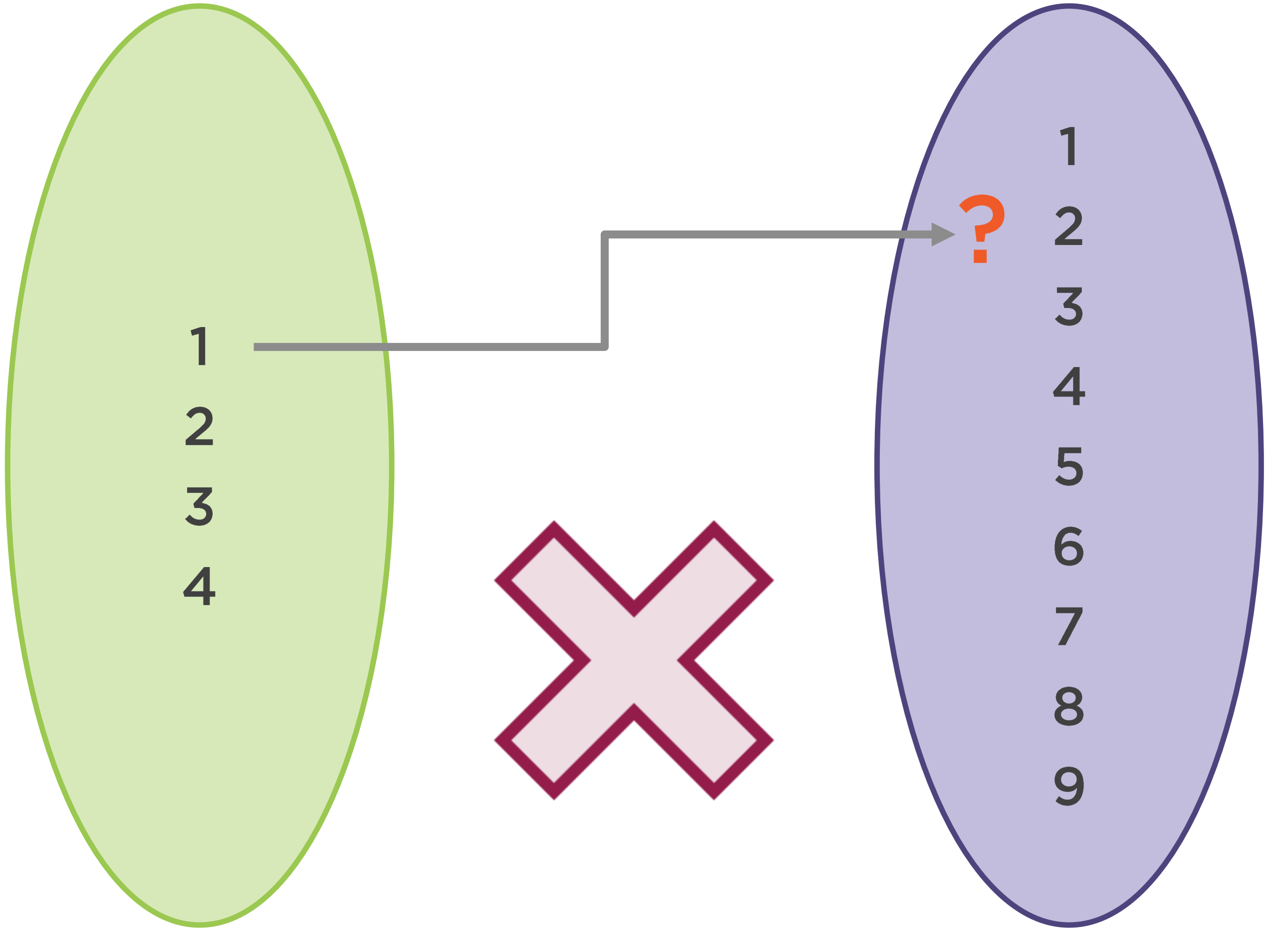
Mathematical Functions



Type of a Function

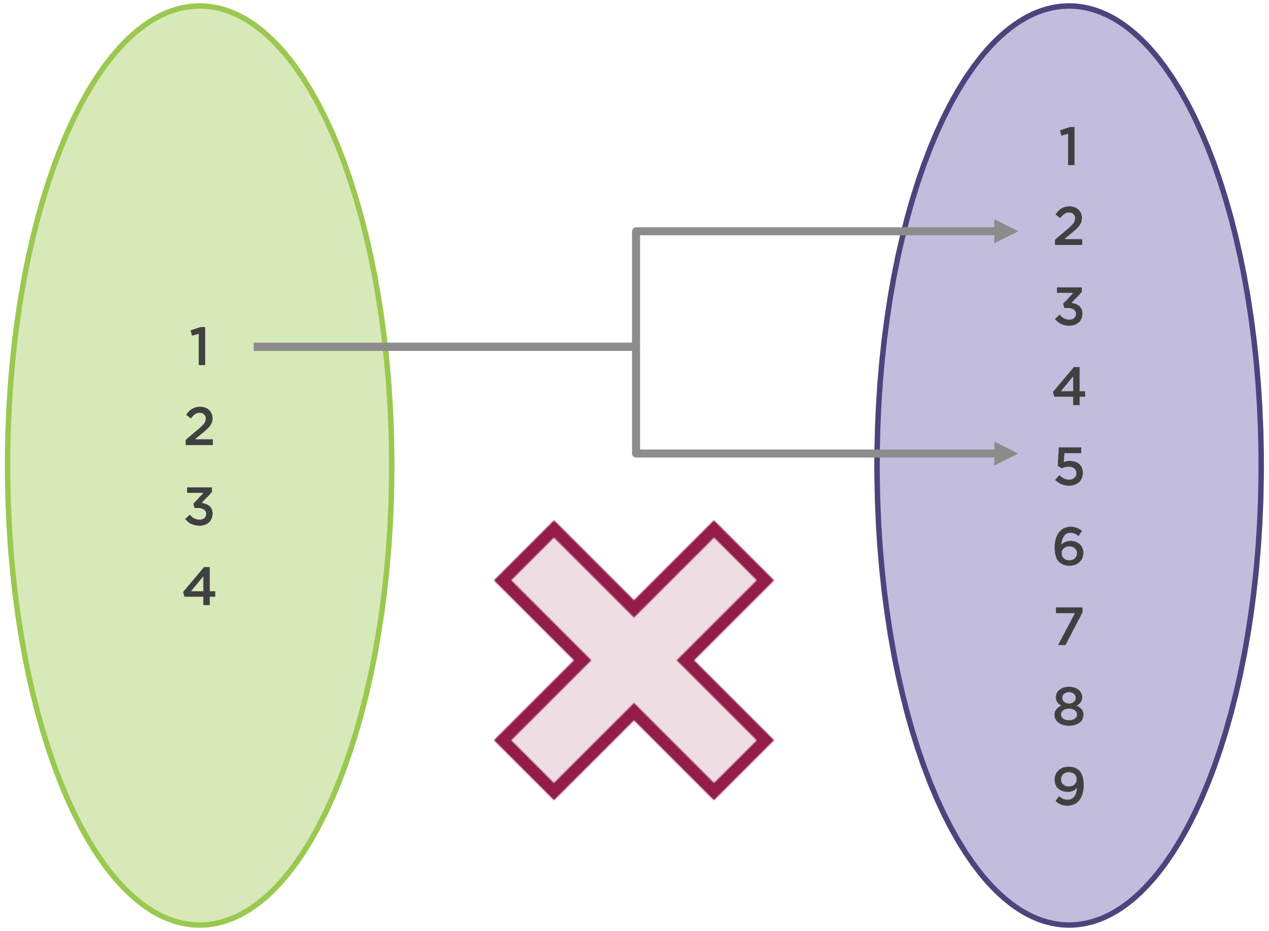
$$f : A \rightarrow B$$

Domain-Codomain Rules



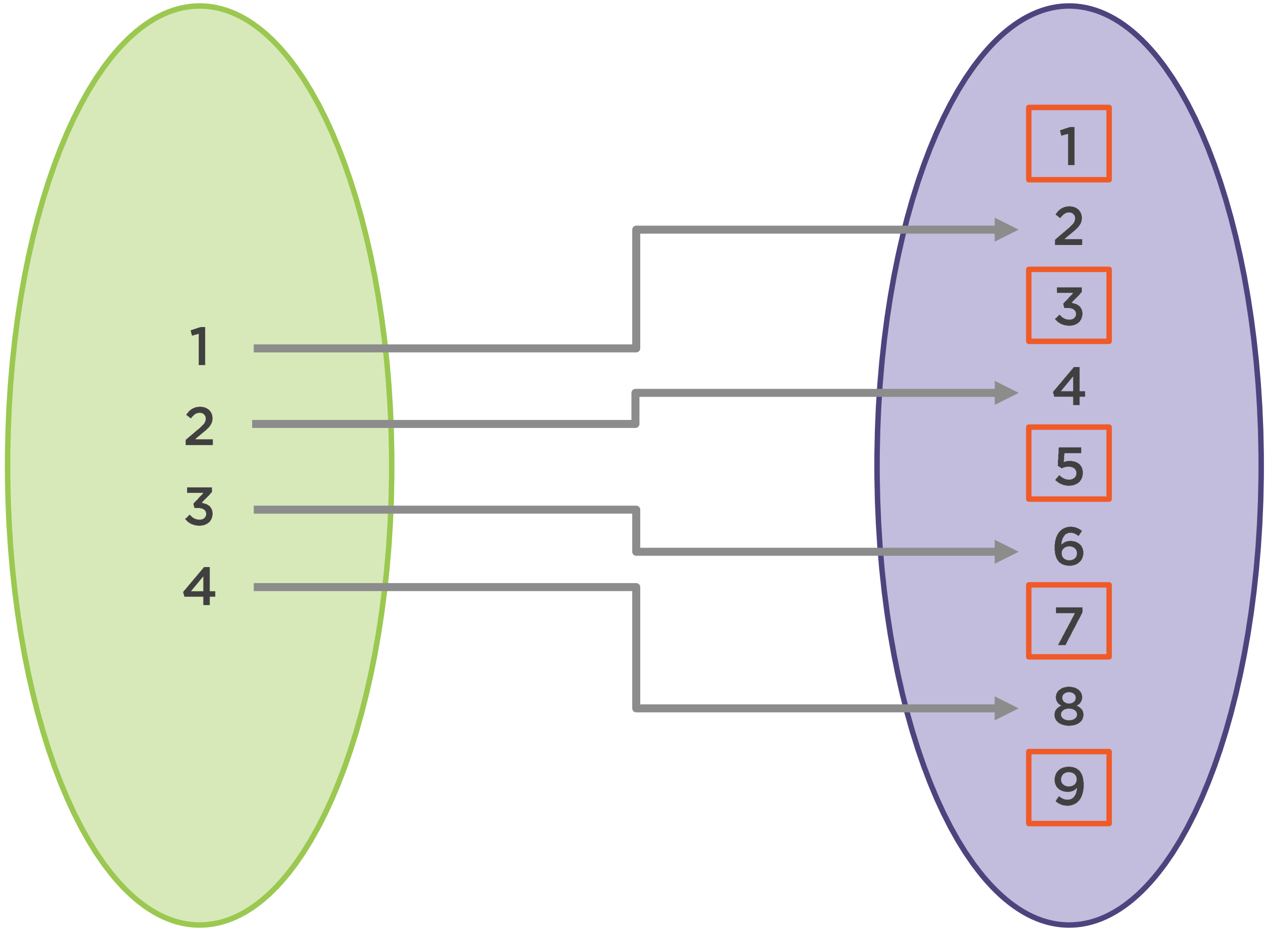
There cannot exist elements in the domain
with no corresponding value in the codomain

Domain-Codomain Rules



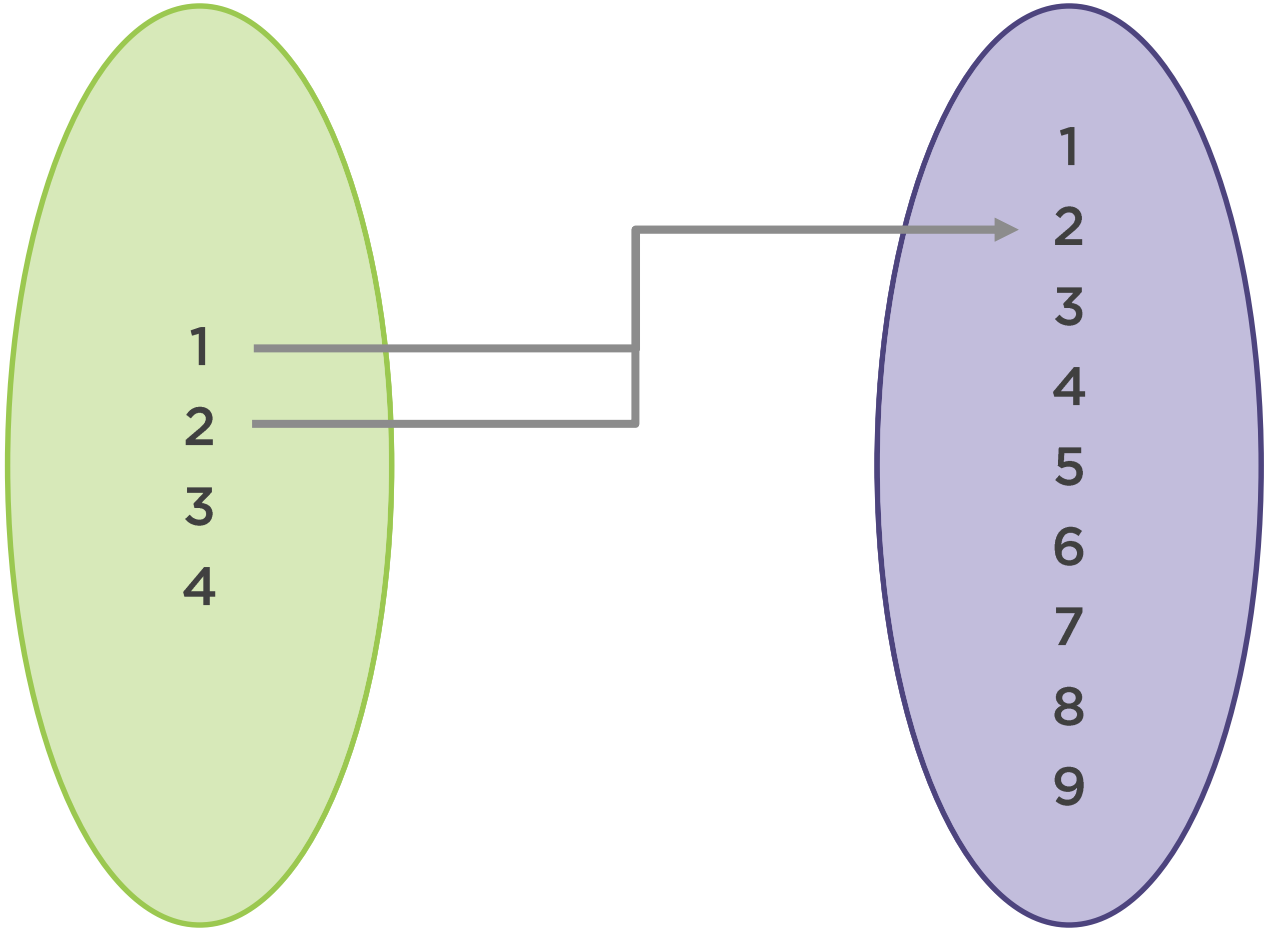
There cannot exist two elements in the codomain corresponding to the same element of the domain

Domain-Codomain Rules



There may be elements in the codomain
with no corresponding element in the domain

Domain-Codomain Rules



There may be elements in the codomain with more than one corresponding element in the domain

Functions in Programming

Block of instructions executed sequentially

Reconcile Programming and Math Functions

- ✓ They must not mutate their argument or anything outside the function

- ✓ They must always return a value

- ✓ When called with the same argument, they must always return the same result

Representing Functions in Java

Methods Belong to Classes

```
class MyClass {
    public Object myMethod(Param param) {
        // ...
    }
}
```

Anonymous Classes

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // ...
    }
});
```

Lambda Expressions

```
button.addActionListener( e -> {
        // ...
});
```

Example of a Functional Interface

```java
public interface Function<T, R> {
  R apply(T arg);
}
```

Example of a Functional Interface

```
public interface Function<T, R> {
  R apply(T arg);






}
```

Example of a Functional Interface

```
public interface Function<T, R> {
  R apply(T arg);
  default <V> Function<V, R> compose(Function<V, T> f) {
     // ...
  }



}
```

Example of a Functional Interface

```
public interface Function<T, R> {
  R apply(T arg);
  default <V> Function<V, R> compose(Function<V, T> f) {
     // ...
  }
  static <T> Function<T, T> identity() {
     // ...
  }
}
```

A Function

```
Function<Integer, Integer> addOne = new Function() {
    @Override
    public int apply(int arg) {
        return arg + 1;
    }
};
```

A Function

as a lambda expression

```
Function<Integer, Integer> addOne = arg -> arg + 1;
```

A Function

```
Function<Integer, Integer> addOne = arg -> arg + 1;

System.out.println( addOne(2) );
```

This would be great

A Function

```
Function<Integer, Integer> addOne = arg -> arg + 1;

System.out.println( addOne.apply(2) );
```

We need to
apply the function

A Function

```
Function<Integer, Integer> addOne = (Integer arg) -> arg + 1;
```

A Function

```
Function<Integer>.addOne(arg -> arg + 1);
```

A Function

```
Function<Integer, Integer> addOne = Math::incrementExact;
```

A Function

```
Function<Integer, Integer> addOne = i -> Math.incrementExact(i);
```

java.util.function

package

Main Functional Interfaces

Predicate

Function

Consumer

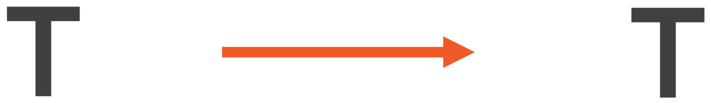
Supplier

UnaryOperator

Main Functional Interfaces

Function

UnaryOperator



T → T

Predicate

T → Boolean

Supplier

( ) → T

Consumer

```
@FunctionalInterface
public interface Consumer {
    void accept(T t);
}
```

Not for functions

High-order Functions

A High-order Function

```
hFunc( l -> l + 1 )
```

Takes a function as its input

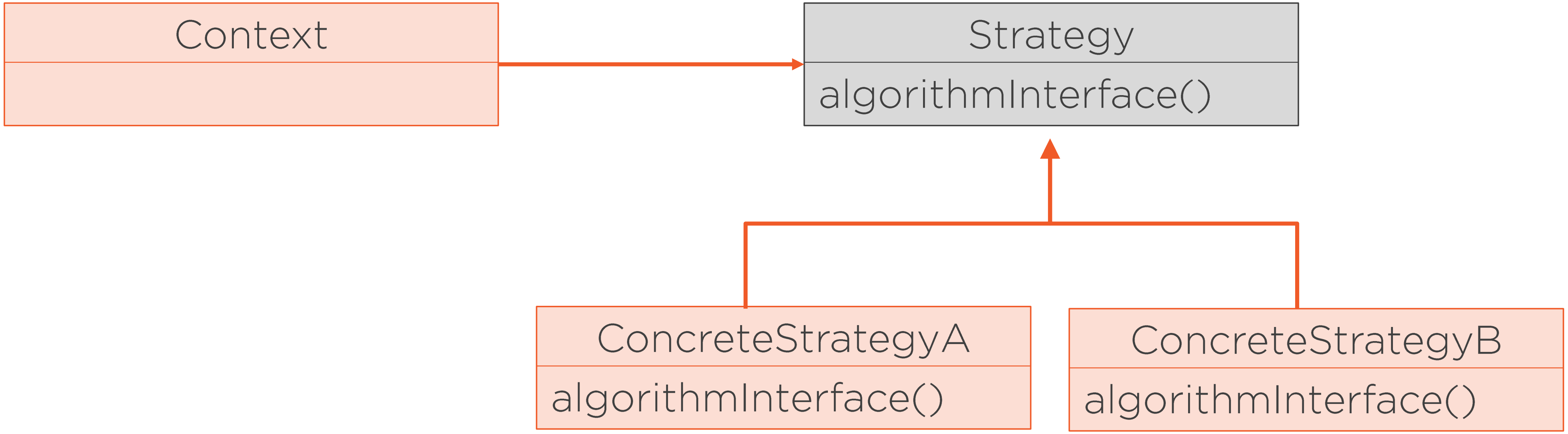
```
Function<Long, Long>
f = hFunc()
```

Returns a function as its output

```
Function<Long, Long>
f = hFunc( l -> l + 1 )
```

Or both

Strategy Pattern



```
interface RewardPointsGenerator {
   RewardPoints calculate(Order order);
}
```

```
Order processOrder(
    Order order, RewardPointsGenerator rewardPointGenerator) {

     // ...
     RewardPoints rp = rewardPointGenerator.calculate(order);

     // ...
}
```

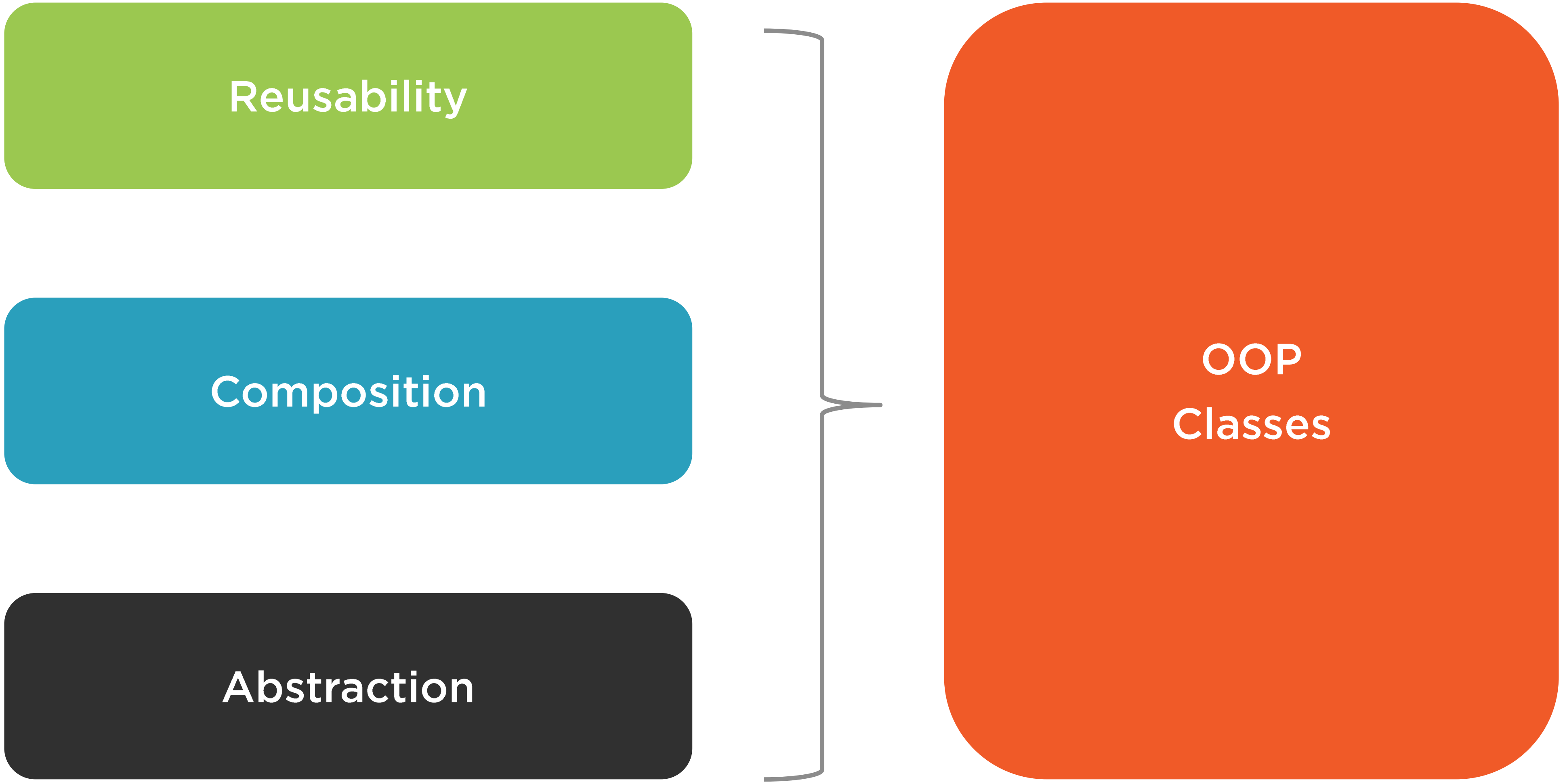
```
RewardPointsGenerator totalBasedRP = order -> { /*...*/ };
RewardPointsGenerator numProductsBasedRP = order -> { /*...*/ };

Order processedOrder1 = processOrder(order, totalBasedRP);
Order processedOrder2 = processOrder(order, numProductsBasedRP);
```

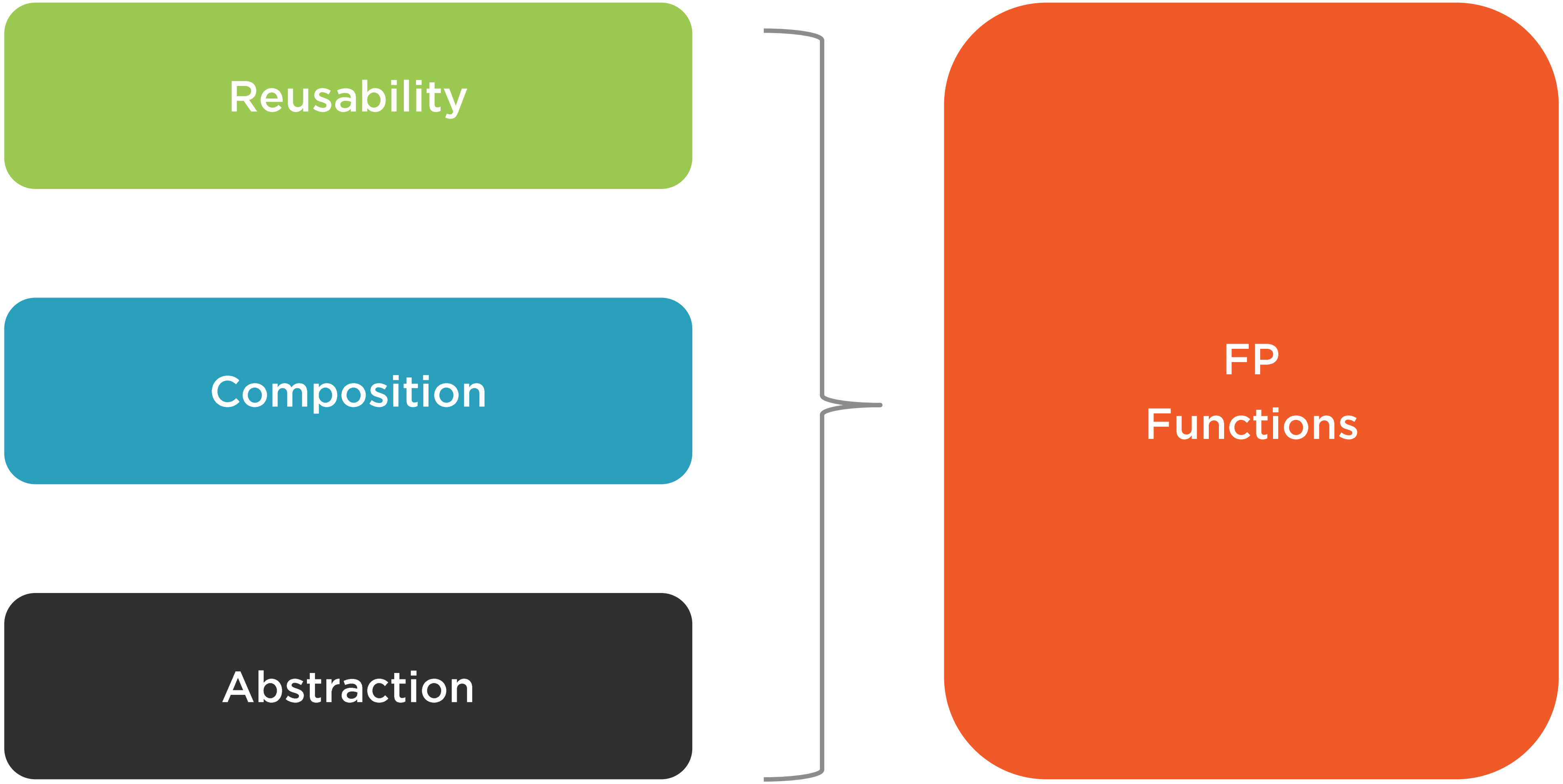
```
Function<Order, RewardPoints> totalBasedRP =
                                           order -> { /*...*/ };
Function<Order, RewardPoints> numProductsBasedRP =
                                           order -> { /*...*/ };


Order processOrder(
       Order order, Function<Order, RewardPoints> rPGenerator) {
     // ...
     Integer rewardPoints = rPGenerator.apply(order);
     // ...
}
```

```
Order processedOrder1 = processOrder(
        order, this::totalBasedRewardPoints
);

Order processedOrder2 = processOrder(
        order, this::numProductsBasedRewardPoints
);
```

High-order Functions

Small, concise units of code


Reusability
Composition
Abstraction
OOP
Classes



Reusability

Composition

Abstraction

FP
Functions

```
List<Integer> filteredList = new ArrayList<Integer>();
for (int n : listOfNumbers) {
    if (n % 3 == 0) {
        filteredList.add(n);
    }
}
```

```
List<Integer> filteredList = listOfNumbers.stream()
                               .filter(n -> n % 3 == 0)
                               .collect(Collectors.toList());
```

```
Predicate<Integer> divisableBy3 = n -> n % 3 == 0;
List<Integer> filteredList = listOfNumbers.stream()
                               .filter(divisableBy3)
                               .collect(Collectors.toList());
```

```java
Predicate<Integer> divisableBy3 = n -> n % 3 == 0;
List<Integer> filteredList = listOfNumbers.stream()
                               .filter(divisableBy3)
                               .map(IntegerUtils::intToString)
                               .collect(Collectors.toList());
```

Composing Functions

Composition

Nesting functions, passing the result of one function as the input of the next.

$$f(x) = x + 10$$

$$g(x) = x * 10$$

$$f \circ g(x) = f(g(x))$$

$$f(x * 10)$$

$$(x * 10) + 10$$

$$f \circ g(1) = f(g(1))$$

$$f(1 * 10)$$

$$10 + 10$$

$$20$$

$$g \circ f(1) = g(f(1))$$

$$g(1 + 10)$$

$$11 * 10$$

$$110$$

$f(x) = x + 10$

$g(x) = x + 5$

$f \circ g(1) = f(g(1)) = f(1 + 5) = 6 + 10 = 16$

$g \circ f(1) = g(f(1)) = g(1 + 10) = 11 + 5 = 16$

$$f(x) = x * 10$$

$$g(x) = x * 5$$

$$f \circ g(1) = f(g(1)) = f(1 * 5) = 5 * 10 = 50$$

$$g \circ f(1) = g(f(1)) = g(1 * 10) = 10 * 5 = 50$$

Associative property

Mathematical principle that proves that the grouping of values does not affect the result.

$$1 + 2 + 3$$

$$=$$

$$(1 + 2) + 3$$

$$=$$

$$1 + (2 + 3)$$

Associative Property in Functional Composition

$$f = a \circ b \circ c$$

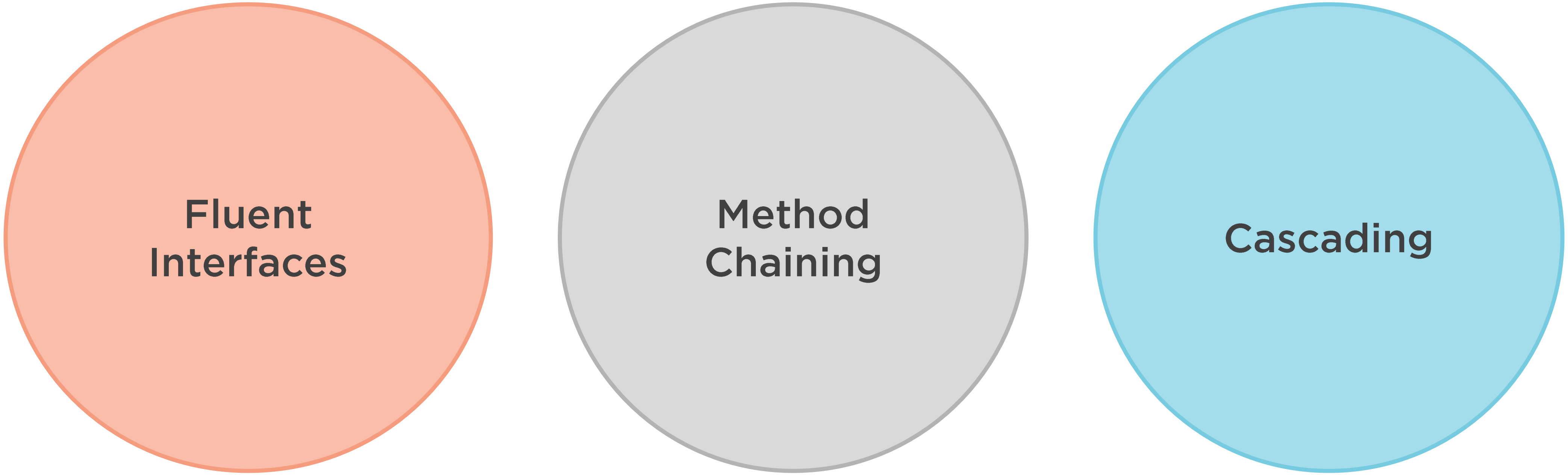
$$g = a \circ b$$

$$f = g \circ c$$

$$g = b \circ c$$

$$f = a \circ g$$

Confusing Concepts



Fluent Interfaces

Method Chaining

Cascading

Example of a Fluent Interface ...

```
BigDecimal amount = BigDecimal.TEN
                      .add(BigDecimal.ONE)
                      .multiply(BigDecimal.TEN)
                      .subtract(BigDecimal.ONE);
```

… And Method Chaining

```java
BigDecimal amount = BigDecimal.TEN
                      .add(BigDecimal.ONE)
                      .multiply(BigDecimal.TEN)
                      .subtract(BigDecimal.ONE);
```

Cascading

```
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

Function Composition = Fluent Interface + Method Chaining

andThen

```
default <V> Function<T, V> andThen(Function<R, V> after) {
  return t -> after.apply( apply(t) );
}
```

```
f.compose(g)

==

g.andThen(f)
```

Function Identity

```
static <T> Function<T, T> identity() {
  return t -> t;
}
```

Composing Predicates

Predicate

T → Boolean

Predicate Methods

```
Predicate<T> and(Predicate<? super T> other)
Predicate<T> or(Predicate<? super T> other)
```

Identity of AND

true && true = true

false && true = false

Identity of OR

true || false = true

false || false = false

Things to Remember

A function is a mapping from:

- A domain (the values that go into a function)
- To a codomain (all the possible values that can come out of the function)

The actual values that come out of the function are the range or image

Things to Remember

Functional programming requirements

- Functions must not mutate anything outside the function
- Functions must not mutate their argument
- Functions must always return the same result when called with the same argument

Things to Remember

In Java, functions can be represented by:

- Static methods
- Or Lambda expressions or method references backed by a functional interface

Things to Remember

Most useful functional interfaces

- Predicate
- Function
- Supplier
- Consumer

Things to Remember

A function becomes a high-order function when:

- It takes a function as its input or argument
- It returns a function as its output
- Or do both

High-order functions promote:

- Abstraction
- Composition
- Reusing of behavior

Things to Remember

Composition

- Nesting functions, passing the result of one as the input of the next
- Composed functions are applied in inverse order.

Associative property

- A mathematical principle that proves that the grouping of values does not affect the result
- Also applies to functional composition

Things to Remember

java.util.function.Function interface

- compose

```
Function compose(Function before) {
  return t -> apply(before.apply(t));
}
```

Things to Remember

java.util.function.Function interface

- andThen

```
Function andThen(Function after) {
  return t -> after.apply(apply(t));
}
```

Things to Remember

f.compose(g) is the same as g.andThen(f)

Things to Remember

java.util.function.Function interface

- Identity

```
Function identity() {
    return t -> t;
}
```

Things to Remember

Predicate

- T -> Boolean

Compose predicates

- and()
- or()

In the Next Module

Currying and partial application