

Creating Reusable Functions with Partial Application and Currying



Esteban Herrera

JAVA ARCHITECT

@eh3rrera www.eherrera.net



Closure

The ability of a method to reference a variable or another method in its enclosing context.



Closure Example

```
Integer a = 2;
```

```
Function<Integer, Integer> f = new Function<Integer, Integer>() {  
    @Override  
    public Integer apply(Integer x) {  
        return x * a;  
    }  
};
```



Effectively final

A variable is not explicitly marked as final, but its initial value is never changed.



Closure Example

```
Integer a = 2;
```

```
Function<Integer, Integer> f = x -> x * a;
```

```
// Equivalent to  $f(x) = x * 2$ ?
```



Closure Example

```
Integer a = Integer.parseInt(args[0]);
```

```
Function<Integer, Integer> f = x -> x * a;
```

```
// Equivalent to  $f(x) = x * 2$ ?
```



Correct Way to Express the Function

$$f(x, a) = x * a$$



Reusing Is Harder

```
Integer a = Integer.parseInt(args[0]);
```

```
Function<Integer, Integer> x = x -> x * a;
```



Should We Implement This?

$$f(x, a) = x * a$$



Function Interface

```
@FunctionalInterface
```

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



BiFunction Interface

@FunctionalInterface

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



BinaryOperator Interface

@FunctionalInterface

```
public interface BinaryOperator<T> extends BiFunction<T,T,T> {  
}
```



Implement a QuaterFunction?

@FunctionalInterface

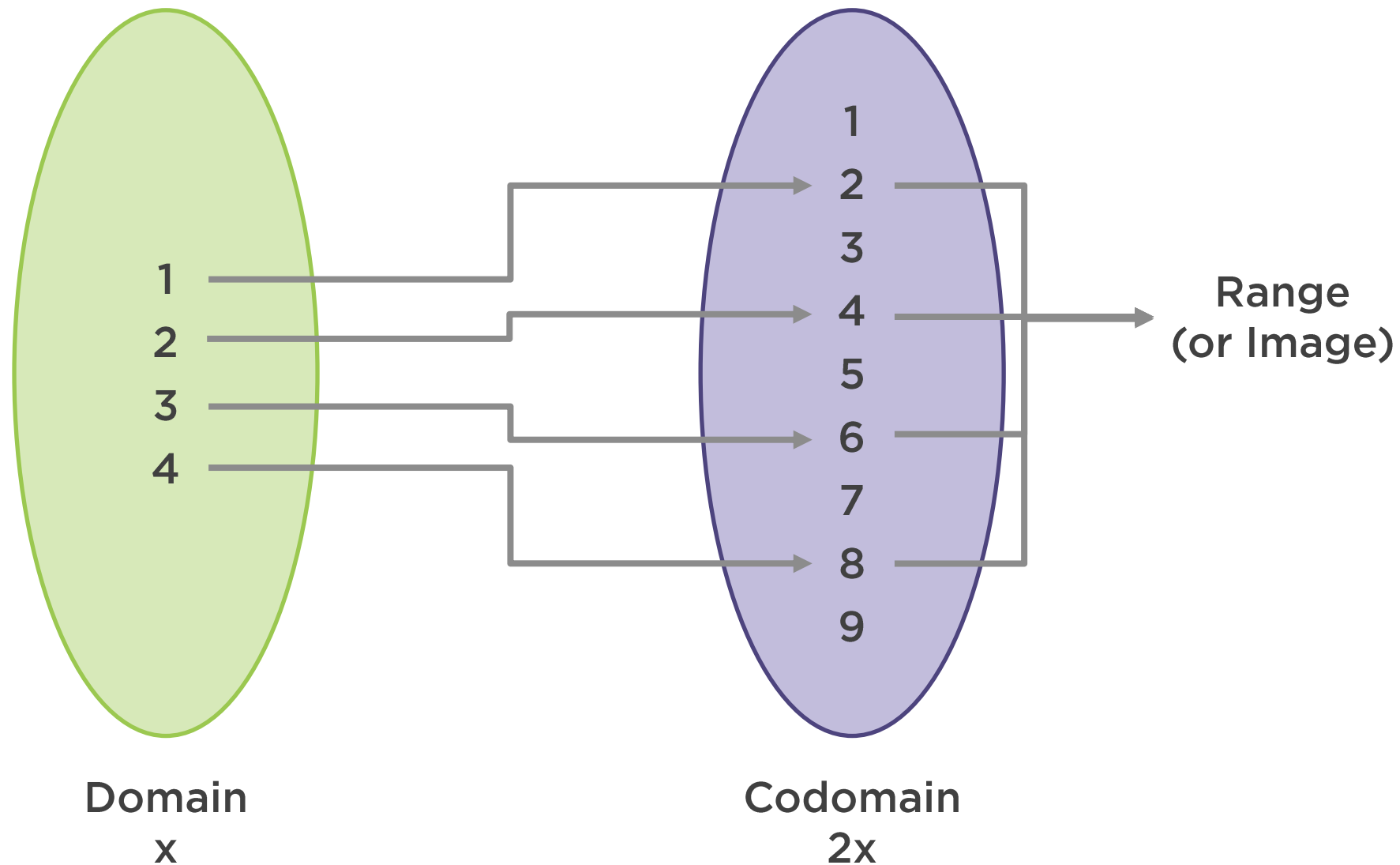
```
public interface QuaterFunction<T, U, V, W, R> {  
    R apply(T t, U u, V v, W w);  
}
```



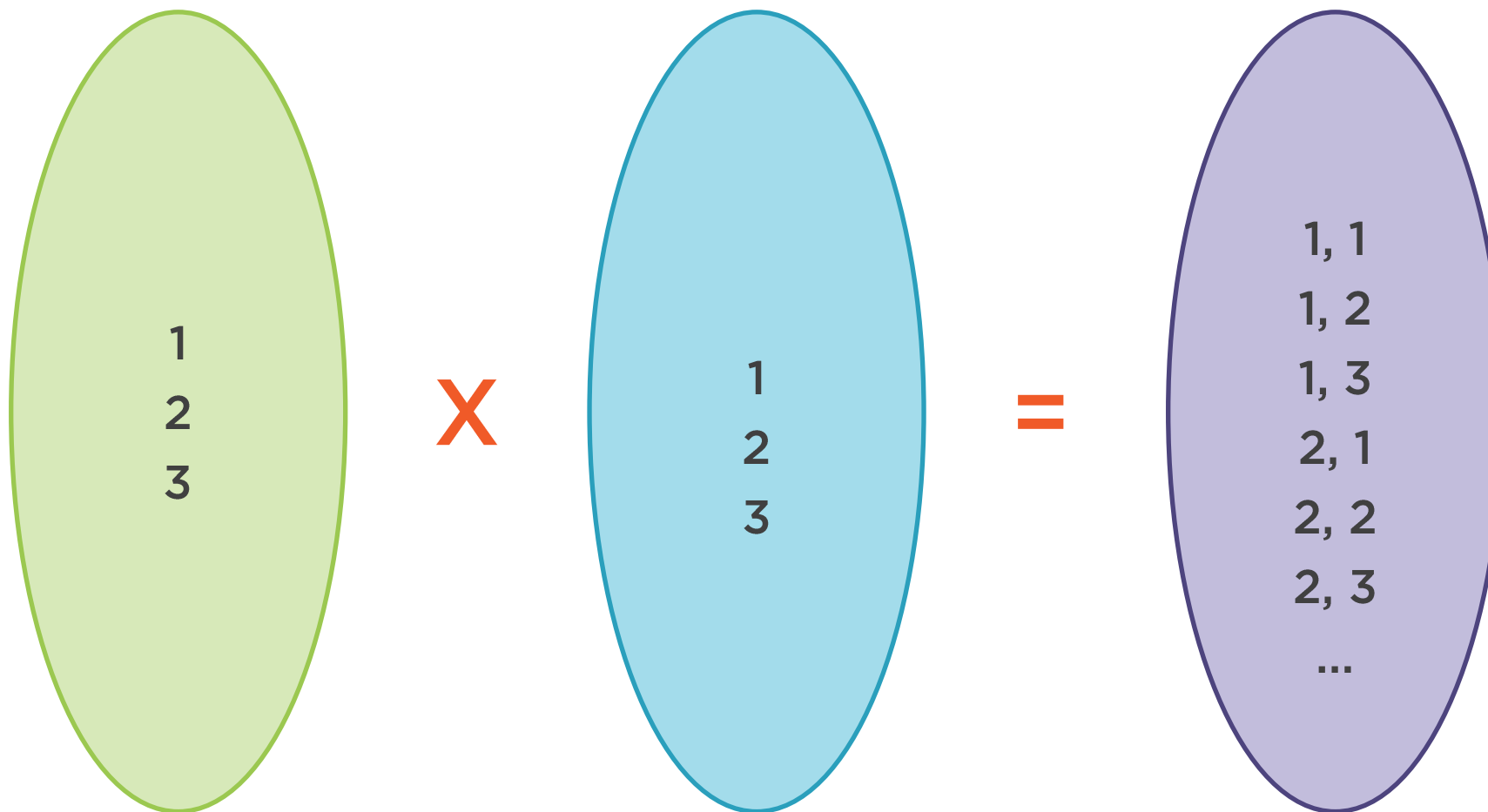
Thinking About Functions of Multiple Arguments



Mathematical Functions



Product of Two Sets



$$f(a, b) = a + b$$

Domain

$A \times B$



Tuples

$(1, 2, 3, 4, \dots)$



$$f(a, b) = a + b$$



$$f((1, 2)) = a + b$$



$$f((1, 2)) = 1 + 2$$



$$f(1, 2) = 1 + 2$$



Implementation of a Tuple

```
class Tuple<T, U> {  
    public final T _1;  
    public final U _2;  
  
    public Tuple(T _1, U _2) {  
        this._1 = _1;  
        this._2 = _2;  
    }  
}
```



```
Function< Tuple< Integer, Integer >, Integer > f =  
    tuple -> tuple._1 + tuple._2;
```

```
Integer i = f.apply( new Tuple<>(2, 1) ); // 3
```



Currying

Converting a function of multiple arguments to a function of one argument each step at a time.



$$f(a, b) = a + b$$



$$f(a) = g(b) = a + b$$

The range of f consists of functions



$$f(1) = g(b) = 1 + b$$

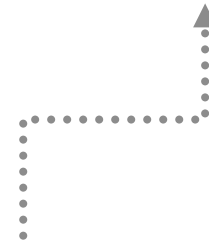


$$f(1) = g(2) = 1 + 2$$



In Java

```
int i = add( 1, 2 )
```



Evaluated at the same time



With currying,
the arguments of
a function are evaluated
at different times.



Partial application

Supplying fewer arguments than the ones required by the function.



add(1, 2, 3, 4, 5)



add3(3, 4, 5)



add6(4, 5)

add(1, 2, 3, 4, 5)



add1(2, 3, 4, 5)



add3(3, 4, 5)

All currying is partial application, but not all partial application is currying.



Using Currying in Java



In Mathematics

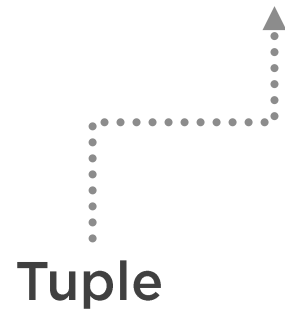
$$f(a, b) = a + b$$



In Mathematics

$$f((a, b)) = a + b$$

Tuple



Curried Version

$$f(a)(b) = g(b)$$

$$g(b) = a + b$$



Integer -> (Integer -> Integer)



<Integer, <Integer, Integer>>



Curried Function in Java

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



Curried Function in Java

```
Function<Integer, Function<Integer, Integer>> add =  
    a -> b -> a + b;
```



Curried Function in Java

```
Function<Integer, Function<Integer, Integer>> add =  
    new Function<Integer, Function<Integer, Integer>>() {  
        @Override  
        public Function<Integer, Integer> apply(Integer a) {  
            return new Function<Integer, Integer>() {  
                @Override  
                public Integer apply(Integer b) {  
                    return a + b;  
                }  
            };  
        }  
    };  
};
```



Curried Function in Java

```
Function<Integer, Function<Integer, Integer>> add =  
    a -> b -> a + b;
```



Apply a Curried Function in Java

```
Integer i = add.apply(1).apply(2); // 3
```



Curried Function Syntax in Java

```
Function<Integer, Function<Integer, Integer>> add =  
    a -> b -> a + b;
```



Curried Function Syntax in Java

Function<



Curried Function Syntax in Java

`Function<Integer,`



Curried Function Syntax in Java

```
Function<Integer, Function<Integer, Integer>> add =  
    a -> b -> a + b;
```



The Importance of the Order of Arguments



Order Doesn't Matter Here

$$f(x, y) = x + y$$

$$f(y, x) = x + y$$



But What About Curried Functions?

$$f(x)(y) = x + y$$

$$f(y)(x) = x + y$$



In curried functions,
the order does matter.



Things to Remember



Closures

- Using them as if they were implicit parameters of a function is not recommended

There are no functions with several arguments

Functions that appear to have several arguments are either:

- Functions of tuples
- Functions returning functions

Things to Remember



Currying

- Converting a function with many arguments into a series of one-argument functions

Partial application

- Supplying fewer arguments than the ones required by the function

Things to Remember



Function: (A, B, C) -> D

Curried function: A -> B -> C -> D

Function<A,

Function<B,

Function<C,

Integer

>

>

>



Things to Remember



Argument order

- From the most specific to the least specific

In the Next Module

Abstract control structures

