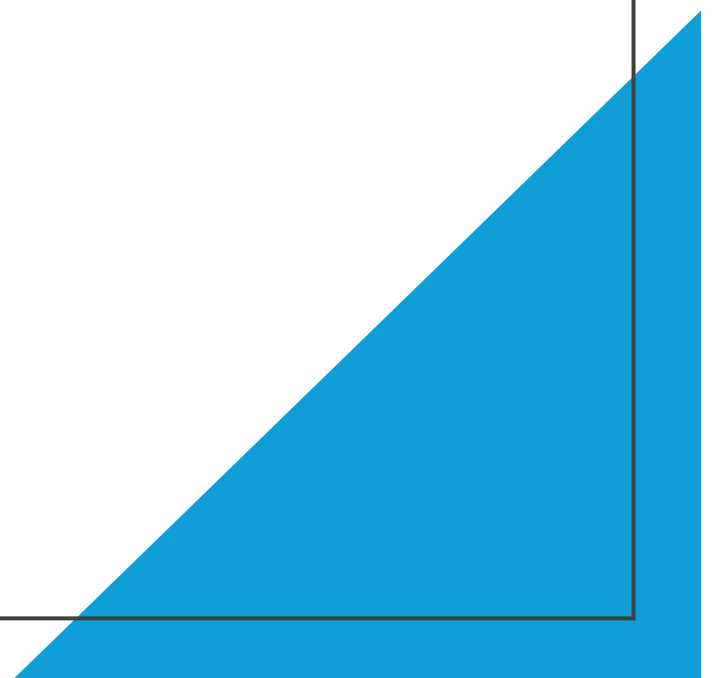


# FP with Java

An Introduction...



# Content

1

## Part I

- FP is unfamiliar
- 3 Core Principles of FP design
- Pattern 1: Functions as parameters

2

## Part II

- Map – Filter – Collect

3

## Part III

- FP vs. OOP

# Sources / Credits

[1] [Functional Design Patterns / Scott Wlaschin](#)

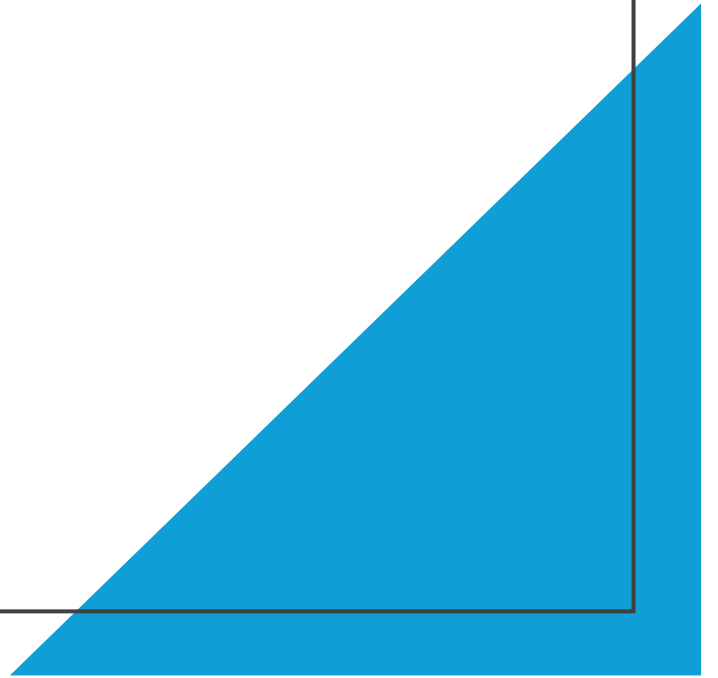
[2] [The Functional Toolkit / Scott Wlaschin](#)

[3] [OOP vs. FP with C# and F# / UrsENZler](#)



# Part I

FP in General



Functional programming is <sup>unfamiliar</sup> ~~scary~~

Functor  
Currying  
Catamorphism  
Applicative  
Monad  
Monoid



- 3 Core Principles of FP design
- Pattern 1: Functions as parameters
- Pattern 2: Composing multi-parameter functions
- Pattern 3: "bind"
- Pattern 4: "map"
- Pattern 5: Monoids

These "patterns" can be built-in or not, depending on your programming language

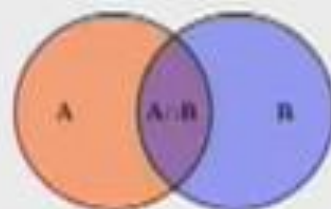
Functions are things



Composition everywhere



Types are not classes



*I'm skipping referential transparency, purity, etc*



## Parameterize all the things

```
public static int Product(int n)
{
    int product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }
    return product;
}

public static int Sum(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += i;
    }
    return sum;
}
```

Diagram annotations:

- Common Code:** Points to the `for` loop in both functions.
- Initial Value:** Points to `product = 1` and `sum = 0`.
- Action:** Points to `product *= i` and `sum += i`.

## Parameterize all the things

```
let product n =
  let initialValue = 1
  let action productSoFar x = productSoFar * x
  [1..n] |> List.fold action initialValue

let sum n =
  let initialValue = 0
  let action sumSoFar x = sumSoFar + x
  [1..n] |> List.fold action initialValue
```

Diagram annotations:

- Initial Value:** Points to `initialValue` in both functions.
- Parameterized action:** Points to the `action` lambda functions.
- Common code extracted:** Points to the `List.fold` call in both functions.

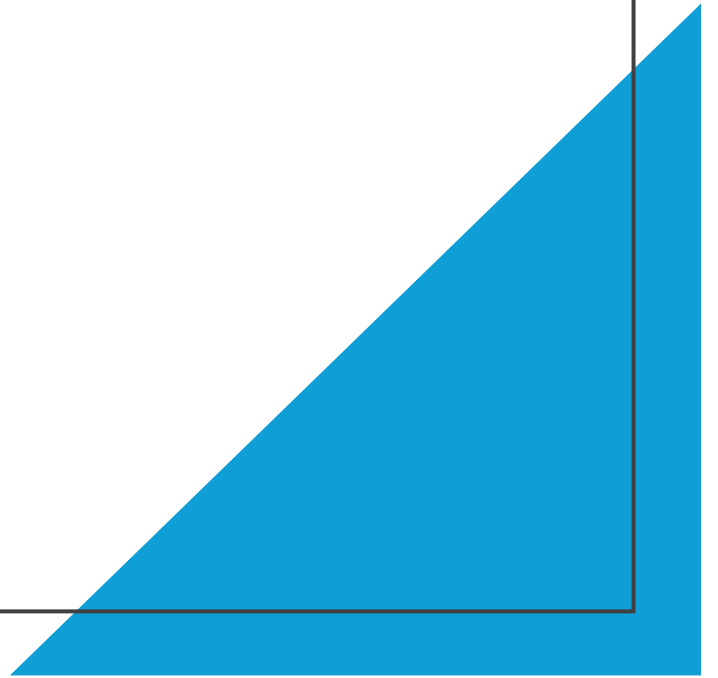
Lots of collection functions like this:  
"fold", "map", "reduce", "collect", etc.

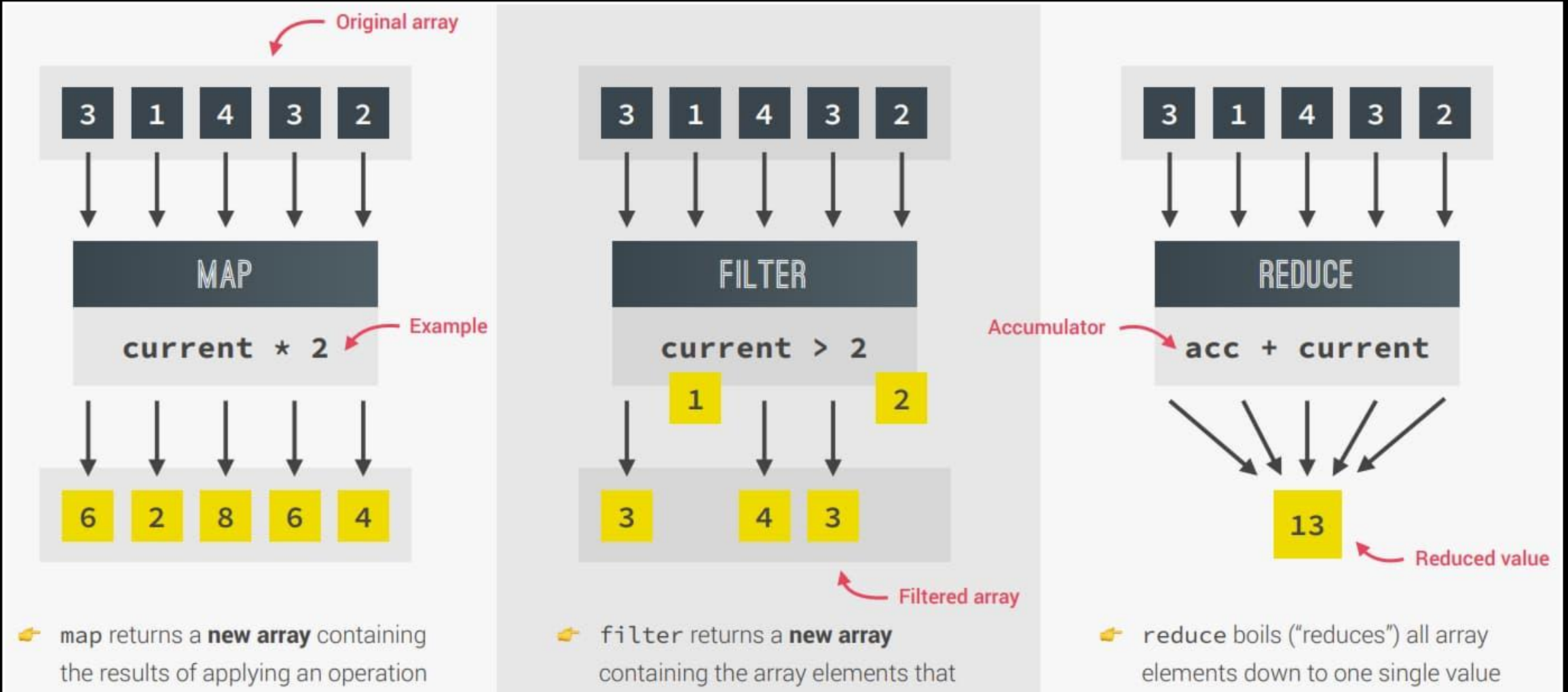
## Pattern 1 - Functions as Parameters



# Part II

FP with Java





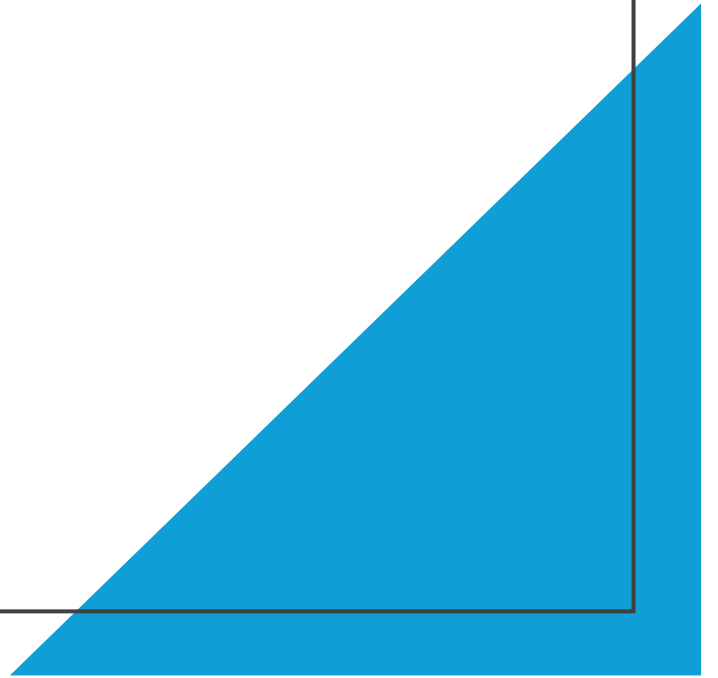
Map – Filter – ~~Reduce~~ Collect

Map – Filter –  
Collect



# Part III

FP vs. OOP



## **OO pattern/principle**

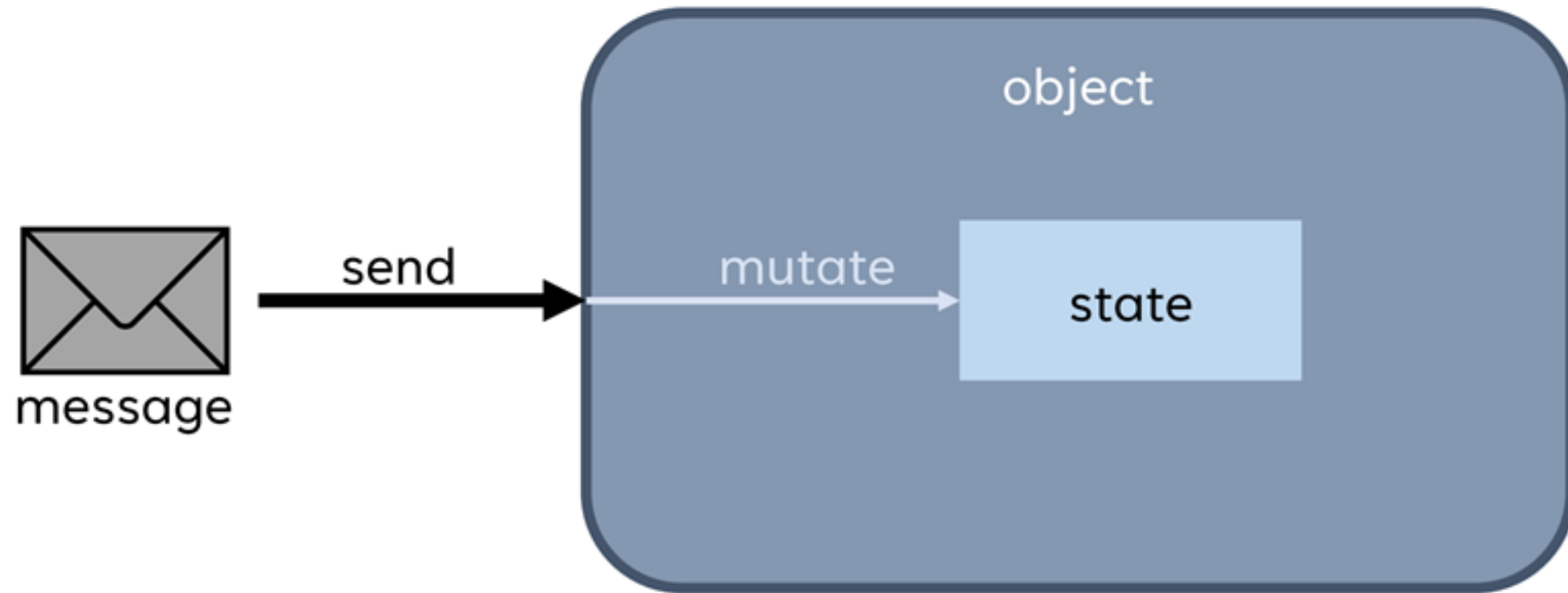
- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

## **FP equivalent**

- Functions
- Functions
- Functions, also
- Functions
- You will be assimilated!
- Functions again
- Functions
- Resistance is futile!

*Seriously, FP patterns are different*

# OOP (simplified)





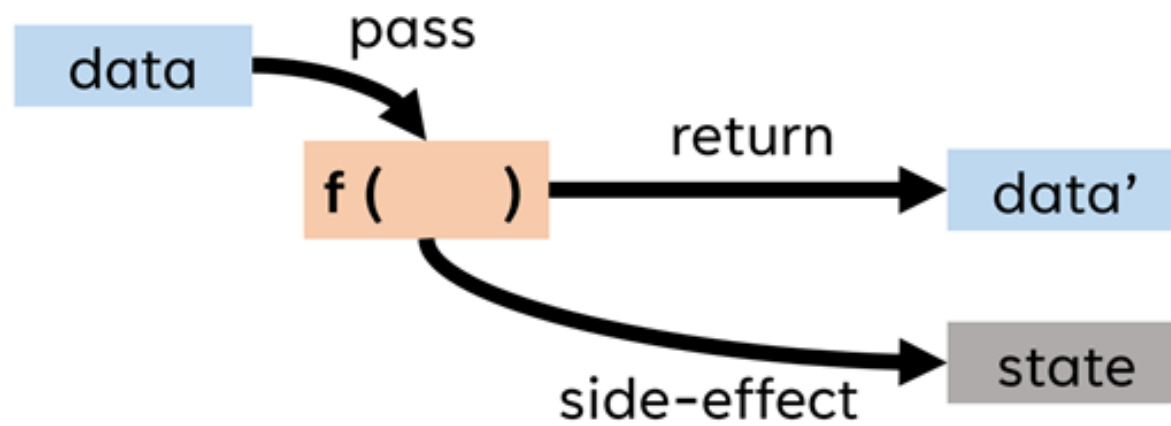
# FP (simplified)

data

calculations



actions



service call, database,  
filesystem, ...



# FP vs. OOP

when to use

**OOP**

composition root  
(when there is variability)

sub-system facades  
("caches")

plug-ins  
(external extensibility)

**FP**

business logic

domain modelling

algorithms

**Mix concepts from both paradigms!**