From object oriented to functional domain modeling

by Mario Fusco
mario.fusco@gmail.com
@mariofusco

What is a functional program?

A program created using only ***pure functions***

No (observable) ***side effects*** allowed like:

- Reassigning a variable
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error

avoidable

- Printing to the console
- Reading user input
- Reading from or writing to a file
- Drawing on the screen

deferrable

Functional programming is a restriction on ***how*** we write programs, but not on ***what*** they can do

OOP vs FP

OOP makes code understandable
by **encapsulating** moving parts

FP makes code understandable
by **minimizing** moving parts

- Michael Feathers

Why Immutability?

- Immutable objects are often **easier to use**. Compare `java.util.Calendar` (mutable) with `java.time.LocalDate` (immutable)
- **Implementing** an immutable object is often easier, as there is less that can go wrong
- Immutable objects **reduce the number of possible interactions** between different parts of the program
- Immutable objects can be **safely shared** between multiple **threads**

A quick premise
It is not only black or white ...

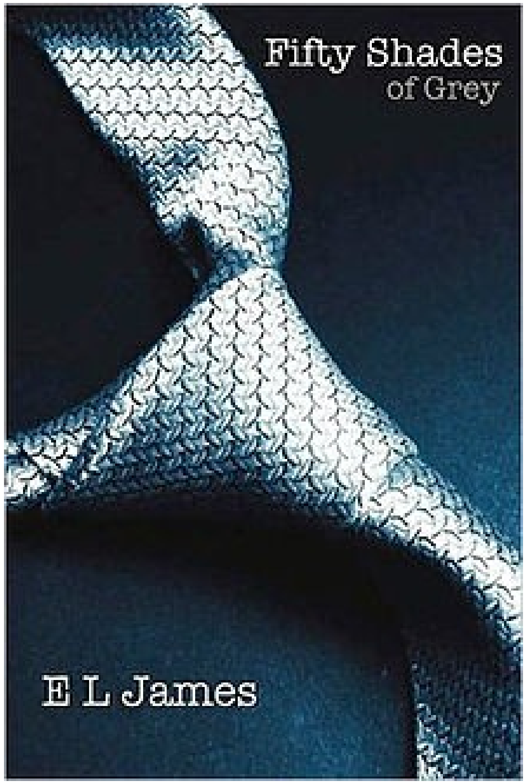
Object
Oriented
Programming

Functional
Programming

A quick premise

It is not only black or white ...

Object Oriented Programming



Functional Programming

… there are (at least) 50 shades of gray in the middle

The OOP/FP dualism - OOP

```
public class Bird { }

public class Cat {
    private Bird catch;
    private boolean full;

    public void capture(Bird bird) {
        catch = bird;
    }

    public void eat() {
        full = true;
        catch = null;
    }
}

Cat cat = new Cat();
Bird bird = new Bird();

cat.capture(bird);
cat.eat();
```

The story

The OOP/FP dualism - FP

```
public class Bird { }

public class Cat {
    public CatWithCatch capture(Bird bird) { return new CatWithCatch(bird); }
}

public class CatWithCatch {
    private final Bird catch;
    public CatWithCatch(Bird bird) { catch = bird; }
    public FullCat eat() { return new FullCat(); }
}
```

Immutability

```
public class FullCat { }
```

More expressive use of type system

```
BiFunction<Cat, Bird, FullCat> story =
        ((BiFunction<Cat, Bird, CatWithCatch>)Cat::capture)
                                                .andThen(CatWithCatch::eat);
```

Emphasis on verbs instead of names

```
FullCat fullCat = story.apply( new Cat(), new Bird() );
```

No need to test internal state: correctness enforced by the compiler

From Object to Function centric

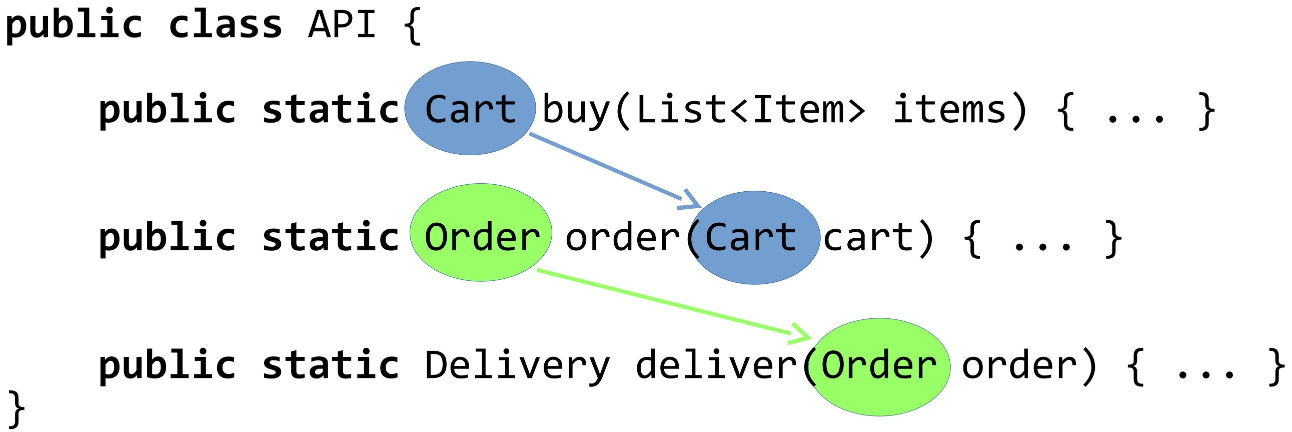
```
BiFunction<Cat, Bird, CatWithCatch> capture =
    (cat, bird) -> cat.capture(bird);
```

```
Function<CatWithCatch, FullCat> eat =
    CatWithCatch::eat;
```

```
BiFunction<Cat, Bird, FullCat> story = capture.andThen(eat);
```

Functions compose better than objects

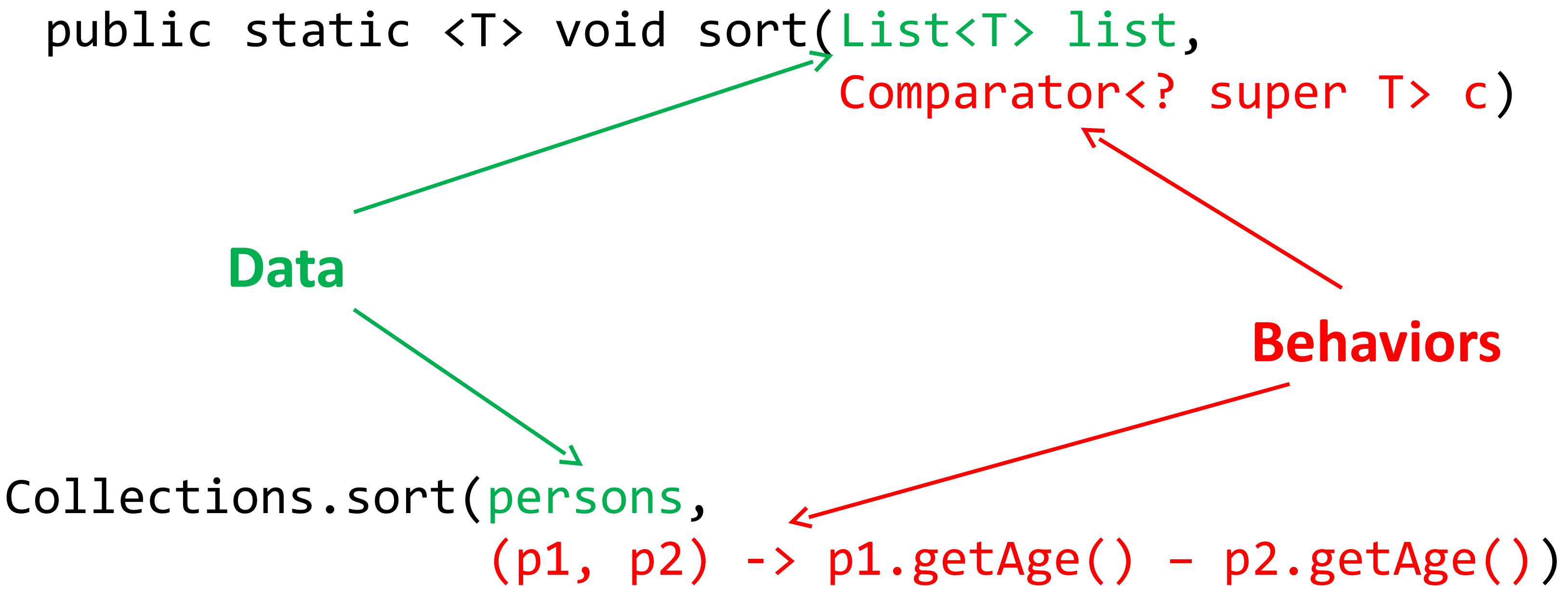
A composable functional API



```
public class API {

    public static Cart buy(List<Item> items) { ... }

    public static Order order(Cart cart) { ... }

    public static Delivery deliver(Order order) { ... }
}
```

```
Function<Delivery, List<Item>> oneClickBuy =
        ((Function<Cart, List<Item>>) API::buy)
            .andThen(API::order)
            .andThen(API::deliver);

Delivery d = oneClickBuy.apply(asList(book, watch, phone));
```

Essence of Functional Programming



```
public static <T> void sort(List<T> list,
                            Comparator<? super T> c)
```

Data

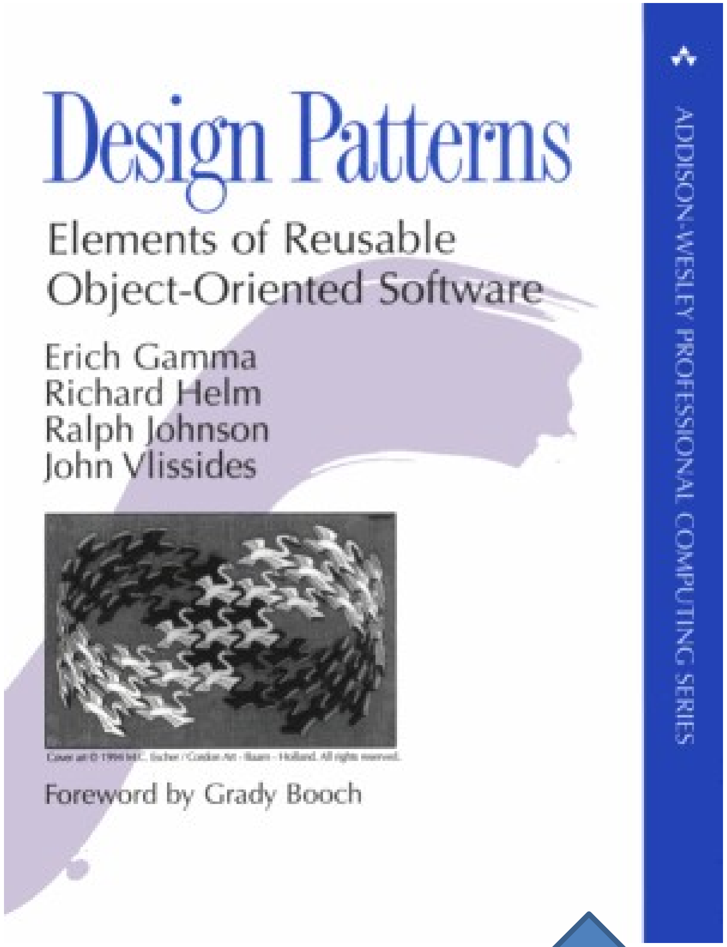
Behaviors

```
Collections.sort(persons,
                 (p1, p2) -> p1.getAge() – p2.getAge())
```

Data and behaviors are the same thing!

Higher-order functions

Are they so mind-blowing?


Design Patterns

Elements of Reusable
Object-Oriented Software

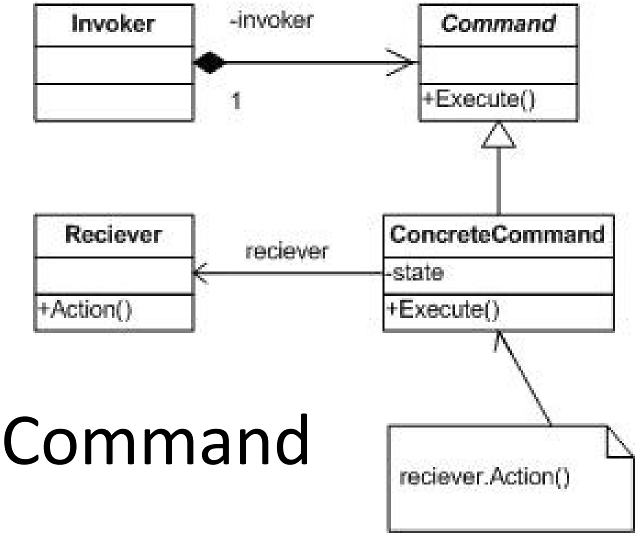
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

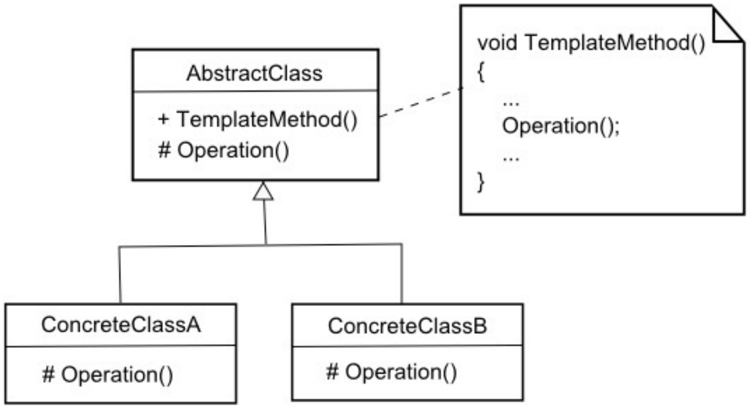
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

… but one of the most influent sw engineering book is almost completely dedicated to them

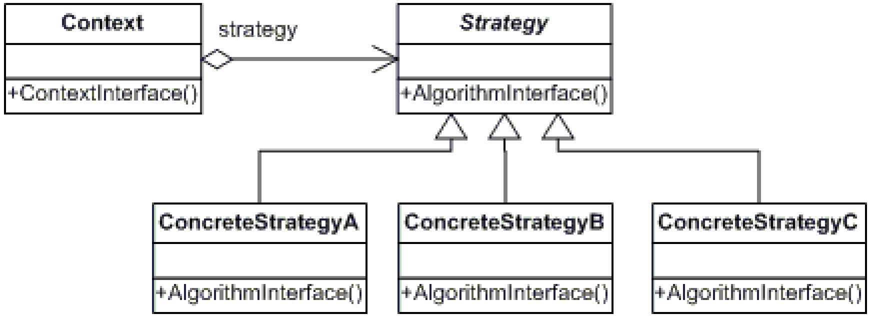


Command

Template Method

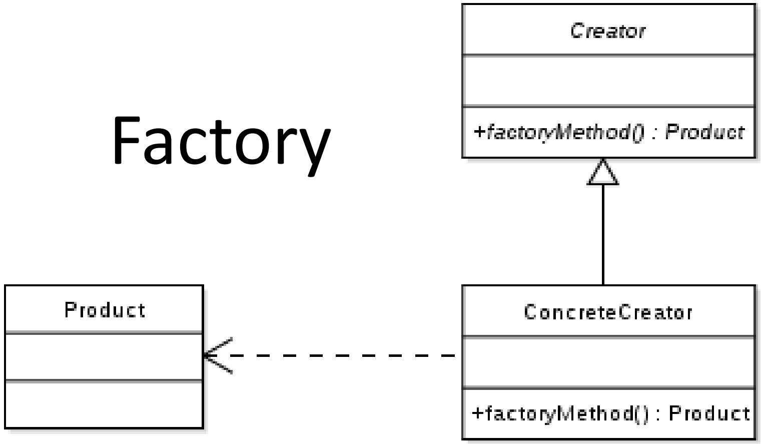


Functions are more general and higher level abstractions



Strategy

Factory

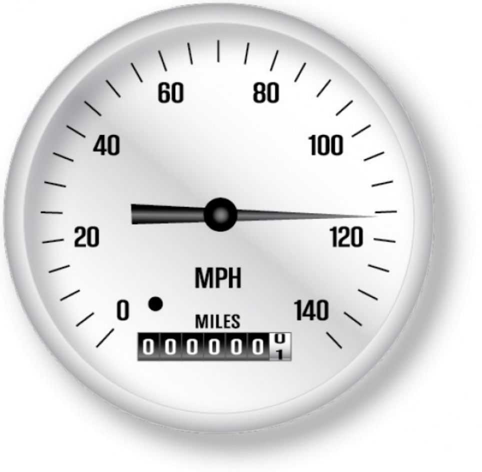


A strategy pattern Converter

```
public interface Converter {
    double convert(double value);
}

public abstract class AbstractConverter implements Converter {
    public double convert(double value) {
        return value * getConversionRate();
    }
    public abstract double getConversionRate();
}

public class Mi2KmConverter extends AbstractConverter {
    public double getConversionRate() { return 1.609; }
}

public class Ou2GrConverter extends AbstractConverter {
    public double getConversionRate() { return 28.345; }
}
```

Using the Converter

```
public List<Double> convertValues(List<Double> values,
                                  Converter converter) {
    List<Double> convertedValues = new ArrayList<Double>();
    for (double value : values) {
        convertedValues.add(converter.convert(value));
    }
    return convertedValues;
}
```



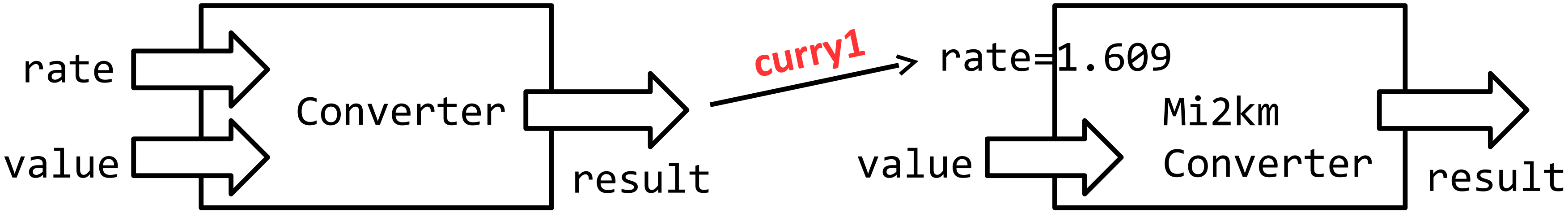
```
List<Double> values = Arrays.asList(10, 20, 50);

List<Double> convertedDistances =
                   convertValues(values, new Mi2KmConverter());
List<Double> convertedWeights =
                   convertValues(values, new Ou2GrConverter());
```

A functional Converter

```
public class Converter implements
                       ExtendedBiFunction<Double, Double, Double> {
    @Override
    public Double apply(Double conversionRate, Double value) {
        return conversionRate * value;
    }
}

@FunctionalInterface
public interface ExtendedBiFunction<T, U, R> extends
                                             BiFunction<T, U, R> {
    default Function<U, R> curry1(T t) {
        return u -> apply(t, u);
    }

    default Function<T, R> curry2(U u) {
        return t -> apply(t, u);
    }
}
```

Currying

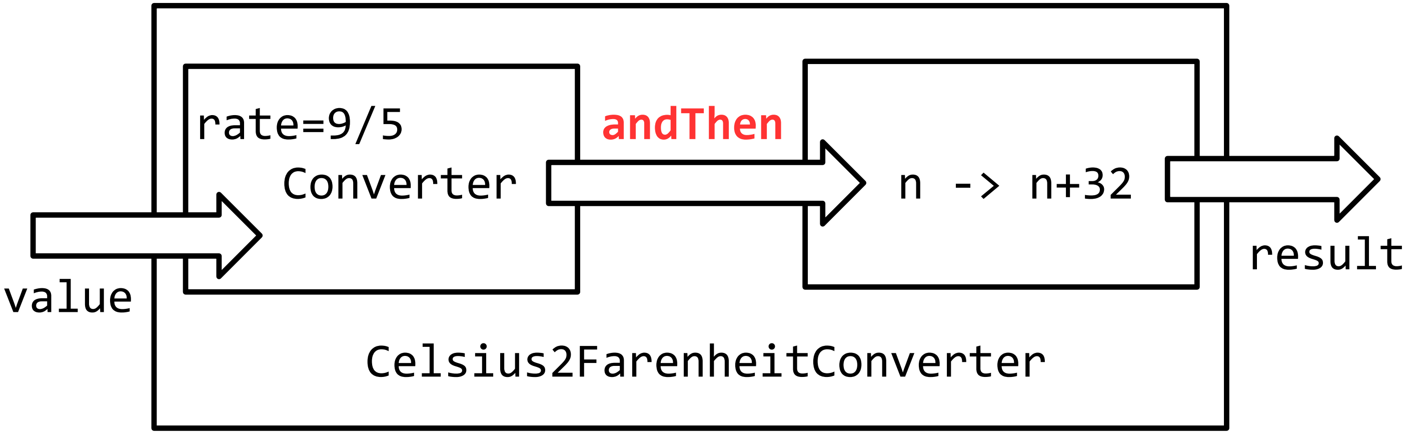
```
Converter converter = new Converter();
double tenMilesInKm = converter.apply(1.609, 10.0);

Function<Double, Double> mi2kmConverter = converter.curry1(1.609);
double tenMilesInKm = mi2kmConverter.apply(10.0);
```



```
List<Double> values = Stream.of(10, 20, 50)
                            .map(mi2kmConverter)
                            .collect(toList())
```

Function Composition

Celsius → Fahrenheit : `F = C * 9/5 + 32`



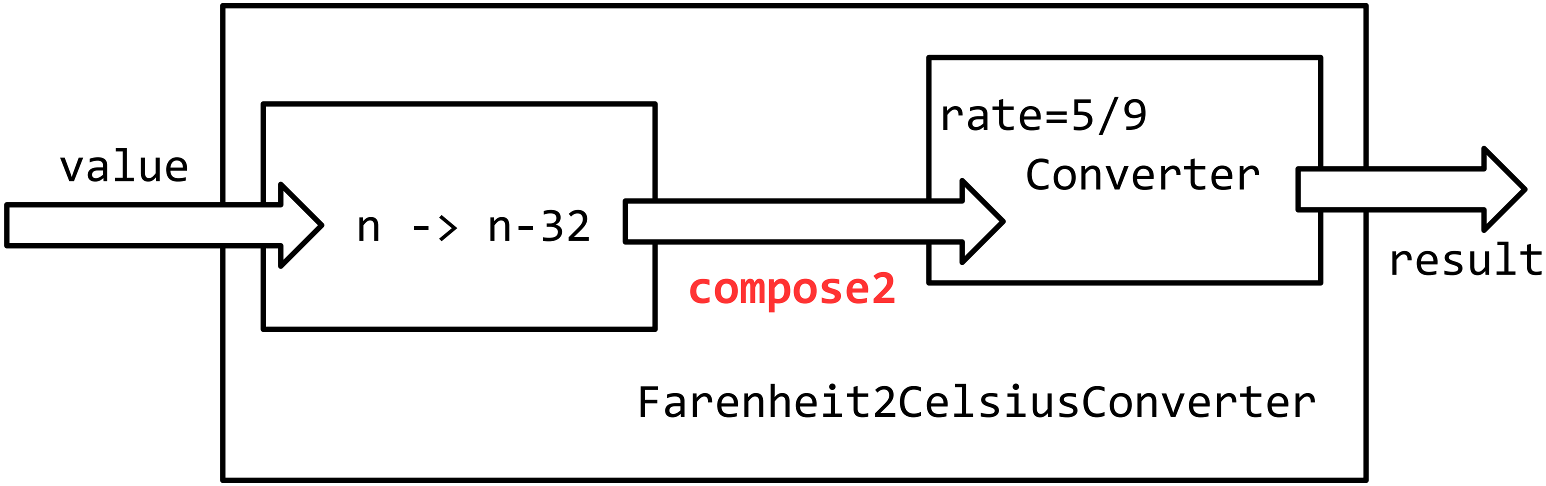
```
Function<Double, Double> c2fConverter =
                new Converter().curry1(9.0/5)
                               .andThen(n -> n + 32);
```

More Function Composition

```
default <V> Function<V, R>
                    compose(Function<? super V, ? extends T> before) {
    return (V v) -> apply(before.apply(v));
}


@FunctionalInterface
public interface ExtendedBiFunction<T, U, R> extends
                                             BiFunction<T, U, R> {
    default <V> ExtendedBiFunction<V, U, R>
                  compose1(Function<? super V, ? extends T> before) {
        return (v, u) -> apply(before.apply(v), u);
    }

    default <V> ExtendedBiFunction<T, V, R>
                  compose2(Function<? super V, ? extends U> before) {
        return (t, v) -> apply(t, before.apply(v));
    }
}
```

More Function Composition

Fahrenheit → Celsius :   `C = (F - 32) * 5/9`



```
Function<Double, Double> f2cConverter =
               new Converter().compose2((Double n) -> n - 32)
                                 .curry1(5.0/9);
```

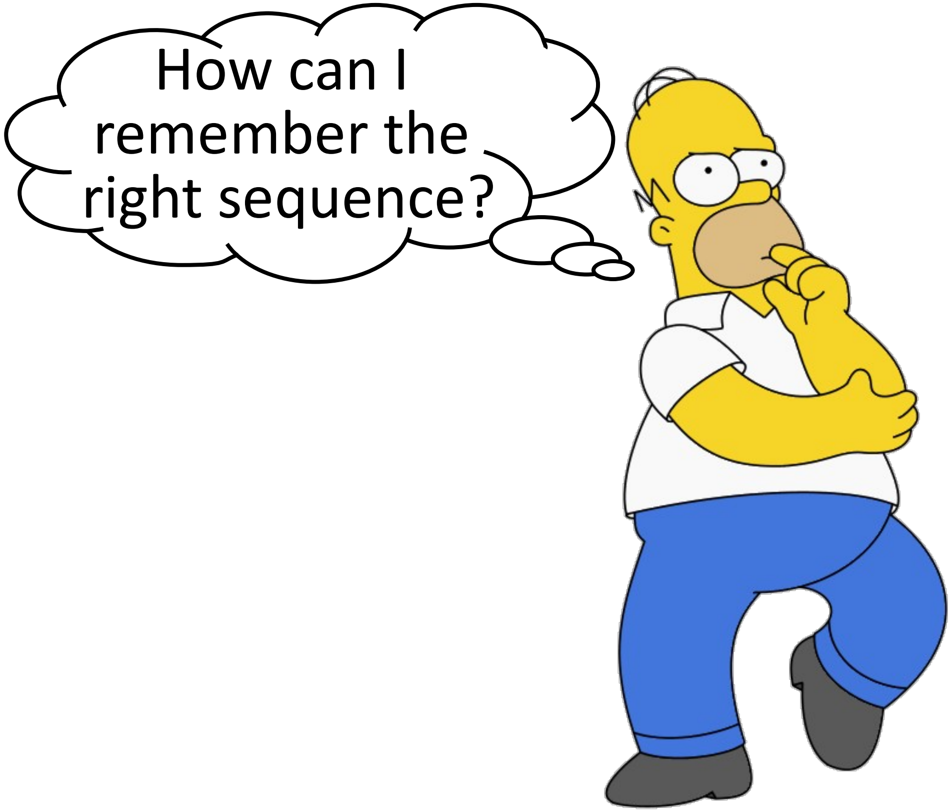
Functions are **building blocks** to create other functions

A Salary Calculator

```
public class SalaryCalculator {

    public double plusAllowance(double d) { return d * 1.2; }

    public double plusBonus(double d) { return d * 1.1; }

    public double plusTax(double d) { return d * 0.7; }

    public double plusSurcharge(double d) { return d * 0.9; }

    public double calculate(double basic, boolean... bs) {
        double salary = basic;
        if (bs[0]) salary = plusAllowance(salary);
        if (bs[1]) salary = plusBonus(salary);
        if (bs[2]) salary = plusTax(salary);
        if (bs[3]) salary = plusSurcharge(salary);
        return salary;
    }
}
```

Using the Salary Calculator

```
double basicBobSalary = ...;

double netBobSalary =
    new SalaryCalculator().calculate( basicBobSalary,
                                      false, // allowance
                                      true,  // bonus
                                      true,  // tax
                                      false  // surcharge
                                    );
```



A Salary Calculator Builder

```
public class SalaryCalculatorBuilder extends SalaryCalculator {

    private boolean hasAllowance;
    private boolean hasBonus;
    private boolean hasTax;
    private boolean hasSurcharge;

    public SalaryCalculatorFactory withAllowance() {
        hasAllowance = true;
        return this;
    }

    // ... more withX() methods

    public double calculate(double basic) {
        return calculate( basic, hasAllowance, hasBonus,
                                 hasTax, hasSurcharge );
    }
}
```

Using the Salary Calculator Factory

```
double basicBobSalary = ...;

double netBobSalary = new SalaryCalculatorBuilder()
        .withBonus()
        .withTax()
        .calculate( basicBobSalary );
```

Better,
but what if I have to
add another function?

Isolating Salary Rules

```
public final class SalaryRules {

    private SalaryRules() { }

    public static double allowance(double d) { return d * 1.2; }

    public static double bonus(double d) { return d * 1.1; }

    public static double tax(double d) { return d * 0.7; }

    public static double surcharge(double d) { return d * 0.9; }
}
```

A Functional Salary Calculator

```
public class SalaryCalculator {

    private final List<Function<Double, Double>> fs =
            new ArrayList<>();

    public SalaryCalculator with(Function<Double, Double> f) {
        fs.add(f);
        return this;
    }

    public double calculate(double basic) {
        return fs.stream()
                 .reduce( Function.identity(), Function::andThen )
                 .apply( basic );
    }
}
```

Using the Functional Salary Calculator

```
double basicBobSalary = ...;

double netBobSalary = new SalaryCalculator()
        .with( SalaryRules::bonus )
        .with( SalaryRules::tax )
        .calculate( basicBobSalary );
```

- No need of any special builder to improve readability

Using the Functional Salary Calculator

```
double basicBobSalary = ...;

double netBobSalary = new SalaryCalculator()
        .with( SalaryRules::bonus )
        .with( SalaryRules::tax )
        .with( s -> s * 0.95 ) // regional tax
        .calculate( basicBobSalary );
```

- No need of any special builder to improve readability

- Extensibility comes for free

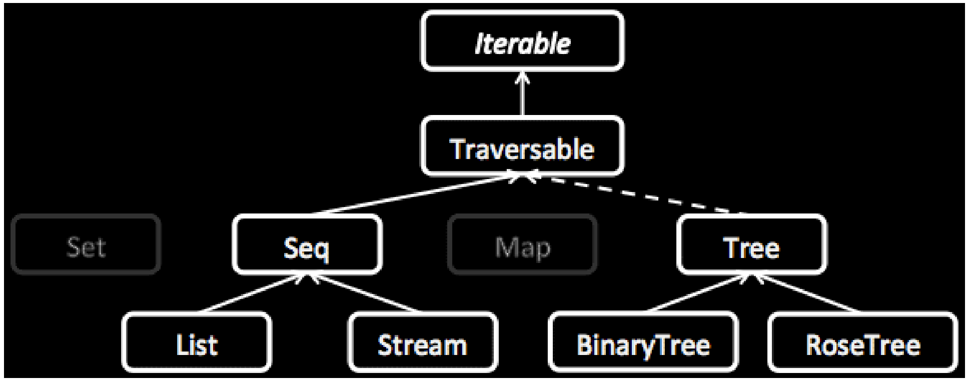
A (better) Functional Salary Calculator

```
public class SalaryCalculator {

    private final Function<Double, Double> calc;

    public SalaryCalculator() { this( Function::identity() ); }

    private SalaryCalculator(Function<Double, Double> calc) {
        this.calc = calc;
    }

    public SalaryCalculator with(Function<Double, Double> f) {
        return new SalaryCalculator( calc.andThen(f) );
    }

    public double calculate(double basic) {
        return calc.apply( basic );
    }
}
```

JAVASLANG
A functional Library for Java 8

```
Tuple3<Person, Account, Building>
```

Immutable Collections



Failure Handling

```
final A result = Try.of(() -> bunchOfWork())
    .recover(x -> Match
        .caze((Exception_1 e) -> ...)
        .caze((Exception_2 e) -> ...)
        .caze((Exception_n e) -> ...)
        .apply(x))
    .orElse(other);
```

Pattern Matching

Let's have a coffee break ...

```
public class Cafe {

    public Coffee buyCoffee(CreditCard cc) {
        Coffee cup = new Coffee();
        cc.charge( cup.getPrice() );
        return cup;
    }

    public List<Coffee> buyCoffees(CreditCard cc, int n) {
        return Stream.generate( () -> buyCoffee( cc ) )
                     .limit( n )
                     .collect( toList() );

    }
}
```

Side-effect

How can we test this without contacting the bank or using a mock?

How can reuse that method to buy more coffees without charging the card multiple times?

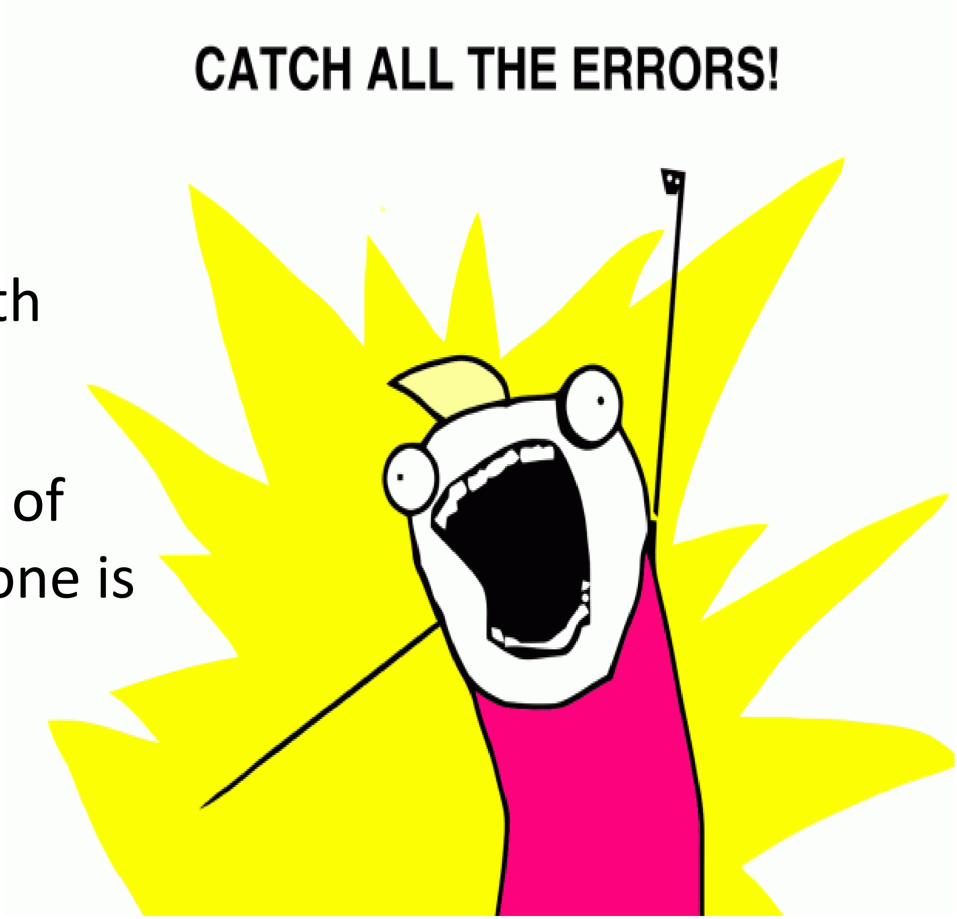
… but please a side-effect free one

```
import javaslang.Tuple2;
import javaslang.collection.Stream;

public class Cafe {
    public Tuple2<Coffee, Charge> buyCoffee(CreditCard cc) {
        Coffee cup = new Coffee();
        return new Tuple2<>(cup, new Charge(cc, cup.getPrice()));
    }

    public Tuple2<List<Coffee>, Charge> buyCoffees(CreditCard cc, int n) {
        Tuple2<Stream<Coffee>, Stream<Charge>> purchases =
                                       Stream.gen( () -> buyCoffee( cc ) )
                                             .subsequence( 0, n )
                                             .unzip( identity() );
        return new Tuple2<>( purchases._1.toJavaList(),
                             purchases._2.foldLeft( new Charge( cc, 0 ),
                                                    Charge::add) );
    }
}
```

```
public Charge add(Charge other) {
    if (cc == other.cc)
        return new Charge(cc, amount + other.amount);
    else
        throw new RuntimeException(
         "Can't combine charges to different cards");
}
```

Error handling with Exceptions?

- Often abused, especially for flow control
- Checked Exceptions harm API extensibility/modificability
- They also plays very badly with lambdas syntax
- Not composable: in presence of multiple errors only the first one is reported
- In the end just a **GLORIFIED MULTILEVEL GOTO**



Error handling
The functional alternatives

`Try<Value>`

- Signal that the required computation may eventually fail

`Either<Exception, Value>`

- The functional way of returning a value which can actually be one of two values: the error/exception (Left) or the correct value (Right)

`Validation<List<Exception>, Value>`

- Composable: can accumulate multiple errors

A OOP BankAccount …

```
public class Balance {
    final BigDecimal amount;
    public Balance( BigDecimal amount ) { this.amount = amount; }
}

public class Account {
    private final String owner;
    private final String number;
    private Balance balance = new Balance(BigDecimal.ZERO);

    public Account( String owner, String number ) {
        this.owner = owner;
        this.number = number;
    }

    public void credit(BigDecimal value) {
        balance = new Balance( balance.amount.add( value ) );
    }

    public void debit(BigDecimal value) throws InsufficientBalanceException {
        if (balance.amount.compareTo( value ) < 0)
            throw new InsufficientBalanceException();
        balance = new Balance( balance.amount.subtract( value ) );
    }
}
```

Mutability

Error handling using Exception

… and how we can use it

```
Account a = new Account("Alice", "123");
Account b = new Account("Bob", "456");
Account c = new Account("Charlie", "789");
```

```
List<Account> unpaid = new ArrayList<>();
for (Account account : Arrays.asList(a, b, c)) {
    try {
        account.debit( new BigDecimal( 100.00 ) );
    } catch (InsufficientBalanceException e) {
        unpaid.add(account);
    }
}
```

Ugly syntax

```
List<Account> unpaid = new ArrayList<>();
Stream.of(a, b, c).forEach( account -> {
    try {
        account.debit( new BigDecimal( 100.00 ) );
    } catch (InsufficientBalanceException e) {
        unpaid.add(account);
    }
} );
```

Mutation of enclosing scope

Cannot use a parallel Stream

Error handling with Try monad

```
public interface Try<A> {
    <B> Try<B> map(Function<A, B> f);
    <B> Try<B> flatMap(Function<A, Try<B>> f);
    boolean isFailure();
}
```

map defines monad's policy for **function application**

flatMap defines monad's policy for **monads composition**

```
public Success<A> implements Try<A> {
    private final A value;
    public Success(A value) { this.value = value; }
    public boolean isFailure() { return false; }
    public <B> Try<B> map(Function<A, B> f) {
        return new Success<>(f.apply(value));
    }
    public <B> Try<B> flatMap(Function<A, Try<B>> f) {
        return f.apply(value);
    }
}

public Failure<A> implements Try<A> {
    private final Object error;
    public Failure(Object error) { this.error = error; }
    public boolean isFailure() { return true; }
    public <B> Try<B> map(Function<A, B> f) { return (Failure<B>)this; }
    public <B> Try<B> flatMap(Function<A, Try<B>> f) { return (Failure<B>)this; }
}
```

A functional BankAccount …

```
public class Account {
    private final String owner;
    private final String number;
    private final Balance balance;

    public Account( String owner, String number, Balance balance ) {
        this.owner = owner;
        this.number = number;
        this.balance = balance;
    }

    public Account credit(BigDecimal value) {
        return new Account( owner, number,
                            new Balance( balance.amount.add( value ) ) );
    }


    public Try<Account> debit(BigDecimal value) {
        if (balance.amount.compareTo( value ) < 0)
            return new Failure<>(  new InsufficientBalanceError() );
        return new Success<>(
            new Account( owner, number,
                         new Balance( balance.amount.subtract( value ) ) ) );
    }
}
```

Immutable

Error handling without Exceptions

… and how we can use it

```
Account a = new Account("Alice", "123");
Account b = new Account("Bob", "456");
Account c = new Account("Charlie", "789");
```

```
List<Account> unpaid =
    Stream.of( a, b, c )
          .map( account ->
                new Tuple2<>( account,
                              account.debit( new BigDecimal( 100.00 ) ) ) )
          .filter( t -> t._2.isFailure() )
          .map( t -> t._1 )
          .collect( toList() );
```

```
List<Account> unpaid =
    Stream.of( a, b, c )
          .filter( account ->
                   account.debit( new BigDecimal( 100.00 ) )
                          .isFailure() )
          .collect( toList() );
```

From Methods to Functions

```
public class BankService {

    public static Try<Account> open(String owner, String number,
                                                  BigDecimal balance) {
        if (initialBalance.compareTo( BigDecimal.ZERO ) < 0)
            return new Failure<>(  new InsufficientBalanceError() );
        return new Success<>( new Account( owner, number,
                                           new Balance( balance ) ) );
    }

    public static Account credit(Account account, BigDecimal value) {
        return new Account( account.owner, account.number,
                         new Balance( account.balance.amount.add( value ) ) );
    }

    public static Try<Account> debit(Account account, BigDecimal value) {
        if (account.balance.amount.compareTo( value ) < 0)
            return new Failure<>(  new InsufficientBalanceError() );
        return new Success<>(
            new Account( account.owner, account.number,
                  new Balance( account.balance.amount.subtract( value ) ) ) );
    }
}
```

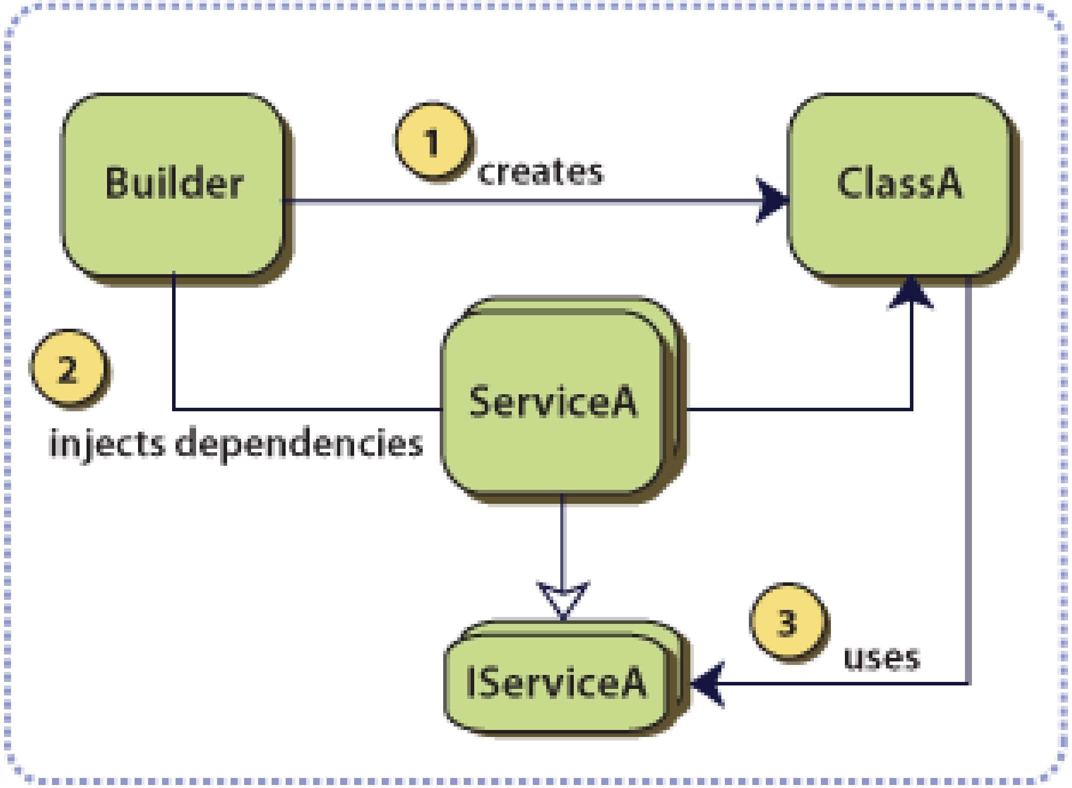
Decoupling state and behavior

```
import static BankService.*

Try<Account> account =
        open( "Alice", "123", new BigDecimal( 100.00 ) )
            .map( acc -> credit( acc, new BigDecimal( 200.00 ) ) )
            .map( acc -> credit( acc, new BigDecimal( 300.00 ) ) )
            .flatMap( acc -> debit( acc, new BigDecimal( 400.00 ) ) );
```

The object-oriented paradigm couples state and behavior

Functional programming decouples them

… but I need a BankConnection!



What about dependency injection?

A naïve solution

```
public class BankService {
    public static Try<Account> open(String owner, String number,
                                    BigDecimal balance, BankConnection bankConnection) {
        ...
    }

    public static Account credit(Account account, BigDecimal value,
                                           BankConnection bankConnection) {
        ...
    }

    public static Try<Account> debit(Account account, BigDecimal value,
                                               BankConnection bankConnection) {
        ...
    }
}
```

Necessary to create the BankConnection in advance ...

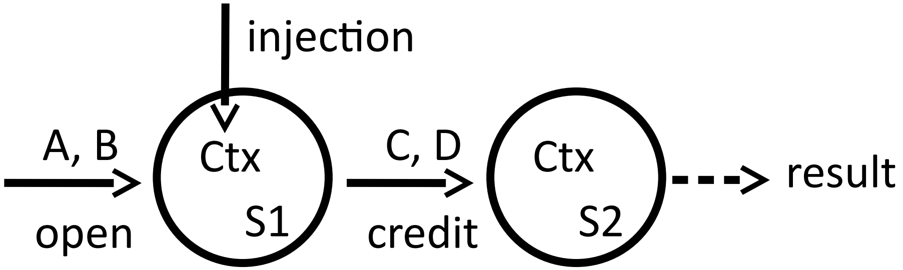
... and pass it to all methods

```
BankConnection bconn = new BankConnection();
Try<Account> account =
        open( "Alice", "123", new BigDecimal( 100.00 ), bconn )
            .map( acc -> credit( acc, new BigDecimal( 200.00 ), bconn ) )
            .map( acc -> credit( acc, new BigDecimal( 300.00 ), bconn ) )
            .flatMap( acc -> debit( acc, new BigDecimal( 400.00 ), bconn ) );
```

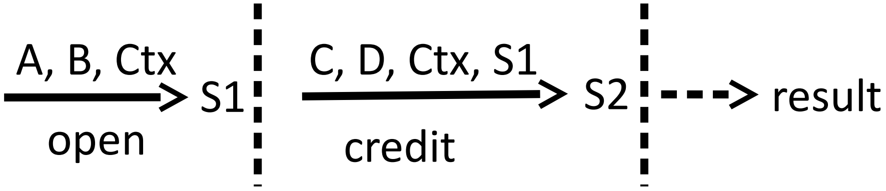
Making it lazy

```
public class BankService {
    public static Function<BankConnection, Try<Account>>
                      open(String owner, String number, BigDecimal balance) {
        return (BankConnection bankConnection) -> ...
    }

    public static Function<BankConnection, Account>
                      credit(Account account, BigDecimal value) {
        return (BankConnection bankConnection) -> ...
    }

    public static Function<BankConnection, Try<Account>>
                      debit(Account account, BigDecimal value) {
        return (BankConnection bankConnection) -> ...
    }
}
```

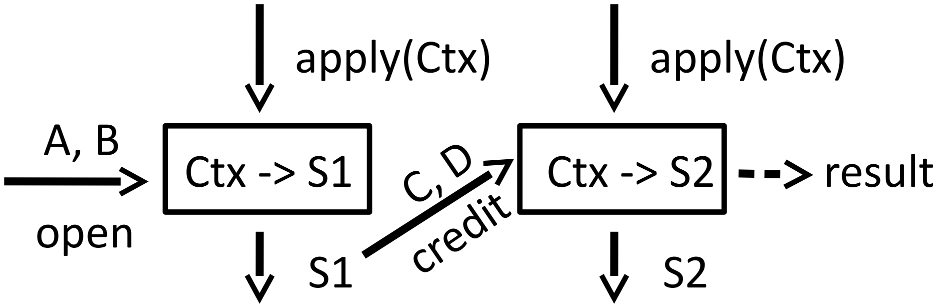
```
Function<BankConnection, Try<Account>> f =
    (BankConnection conn) ->
        open( "Alice", "123", new BigDecimal( 100.00 ) )
        .apply( conn )
        .map( acc -> credit( acc, new BigDecimal( 200.00 ) ).apply( conn ) )
        .map( acc -> credit( acc, new BigDecimal( 300.00 ) ).apply( conn ) )
        .flatMap( acc -> debit( acc, new BigDecimal( 400.00 ) ).apply( conn ) );

Try<Account> account = f.apply( new BankConnection() );
```



Pure OOP implementation



Static Methods



Lazy evaluation

Introducing the Reader monad ...

```
public class Reader<R, A> {
    private final Function<R, A> run;

    public Reader( Function<R, A> run ) {
        this.run = run;
    }

    public <B> Reader<R, B> map(Function<A, B> f) {
        ...
    }

    public <B> Reader<R, B> flatMap(Function<A, Reader<R, B>> f) {
        ...
    }

    public A apply(R r) {
        return run.apply( r );
    }
}
```

The reader monad provides an environment to wrap an abstract computation without evaluating it

Introducing the Reader monad ...

```
public class Reader<R, A> {
    private final Function<R, A> run;

    public Reader( Function<R, A> run ) {
        this.run = run;
    }

    public <B> Reader<R, B> map(Function<A, B> f) {
        return new Reader<>((R r) -> f.apply( apply( r ) ));
    }

    public <B> Reader<R, B> flatMap(Function<A, Reader<R, B>> f) {
        return new Reader<>((R r) -> f.apply( apply( r ) ).apply( r ));
    }

    public A apply(R r) {
        return run.apply( r );
    }
}
```

The reader monad provides an environment to wrap an abstract computation without evaluating it

… and combining it with Try

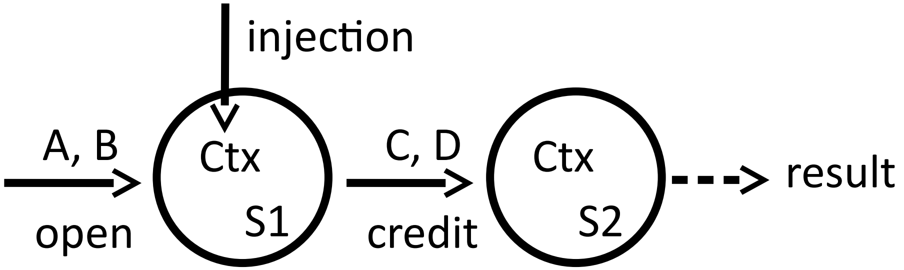
```
public class TryReader<R, A> {
    private final Function<R, Try<A>> run;

    public TryReader( Function<R, Try<A>> run ) {
        this.run = run;
    }

    public <B> TryReader<R, B> map(Function<A, B> f) {
        ...

    }

    public <B> TryReader<R, B> mapReader(Function<A, Reader<R, B>> f) {
        ...

    }

    public <B> TryReader<R, B> flatMap(Function<A, TryReader<R, B>> f) {
        ...

    }

    public Try<A> apply(R r) {
        return run.apply( r );
    }
}
```

… and combining it with Try

```
public class TryReader<R, A> {
    private final Function<R, Try<A>> run;

    public TryReader( Function<R, Try<A>> run ) {
        this.run = run;
    }

    public <B> TryReader<R, B> map(Function<A, B> f) {
        return new TryReader<R, B>((R r) -> apply( r )
                                      .map( a -> f.apply( a ) ));
    }

    public <B> TryReader<R, B> mapReader(Function<A, Reader<R, B>> f) {
        return new TryReader<R, B>((R r) -> apply( r )
                                      .map( a -> f.apply( a ).apply( r ) ));
    }

    public <B> TryReader<R, B> flatMap(Function<A, TryReader<R, B>> f) {
        return new TryReader<R, B>((R r) -> apply( r )
                                      .flatMap( a -> f.apply( a ).apply( r ) ));
    }

    public Try<A> apply(R r) {
        return run.apply( r );
    }
}
```

A more user-friendly API

```
public class BankService {
    public static TryReader<BankConnection, Account>
                      open(String owner, String number, BigDecimal balance) {
        return new TryReader<>( (BankConnection bankConnection) -> ... )
    }

    public static Reader<BankConnection, Account>
                      credit(Account account, BigDecimal value) {
        return new Reader<>( (BankConnection bankConnection) -> ... )
    }

    public static TryReader<BankConnection, Account>
                      debit(Account account, BigDecimal value) {
        return new TryReader<>( (BankConnection bankConnection) -> ... )
    }
}


TryReader<BankConnection, Account> reader =
    open( "Alice", "123", new BigDecimal( 100.00 ) )
        .mapReader( acc -> credit( acc, new BigDecimal( 200.00 ) ) )
        .mapReader( acc -> credit( acc, new BigDecimal( 300.00 ) ) )
        .flatMap( acc -> debit( acc, new BigDecimal( 400.00 ) ) );

Try<Account> account = reader.apply( new BankConnection() );
```



injection

A, B

open

Ctx

S1

C, D

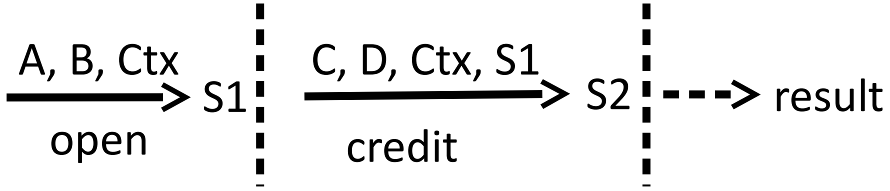
credit

Ctx

S2

result

Pure OOP implementation



A, B, Ctx

open

S1

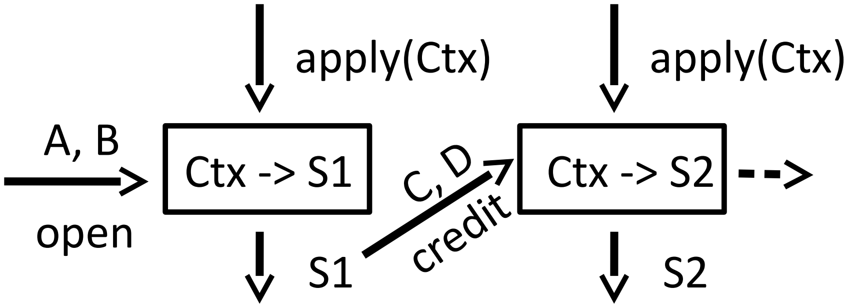
C, D, Ctx, S1

credit

S2

result

Static Methods



apply(Ctx)

apply(Ctx)

A, B

open

Ctx -> S1

C, D

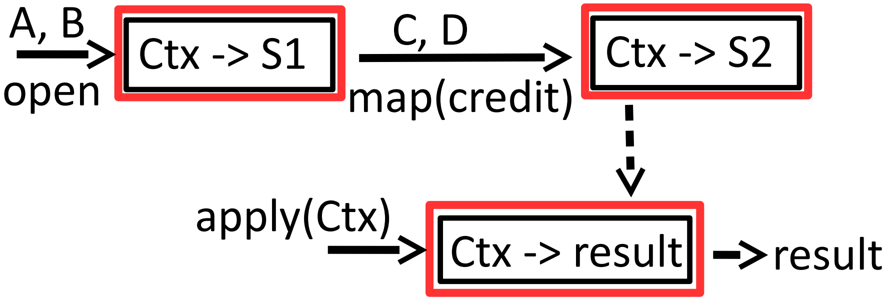
credit

Ctx -> S2

S1

S2

Lazy evaluation



A, B

open

Ctx -> S1

C, D

map(credit)

Ctx -> S2

apply(Ctx)

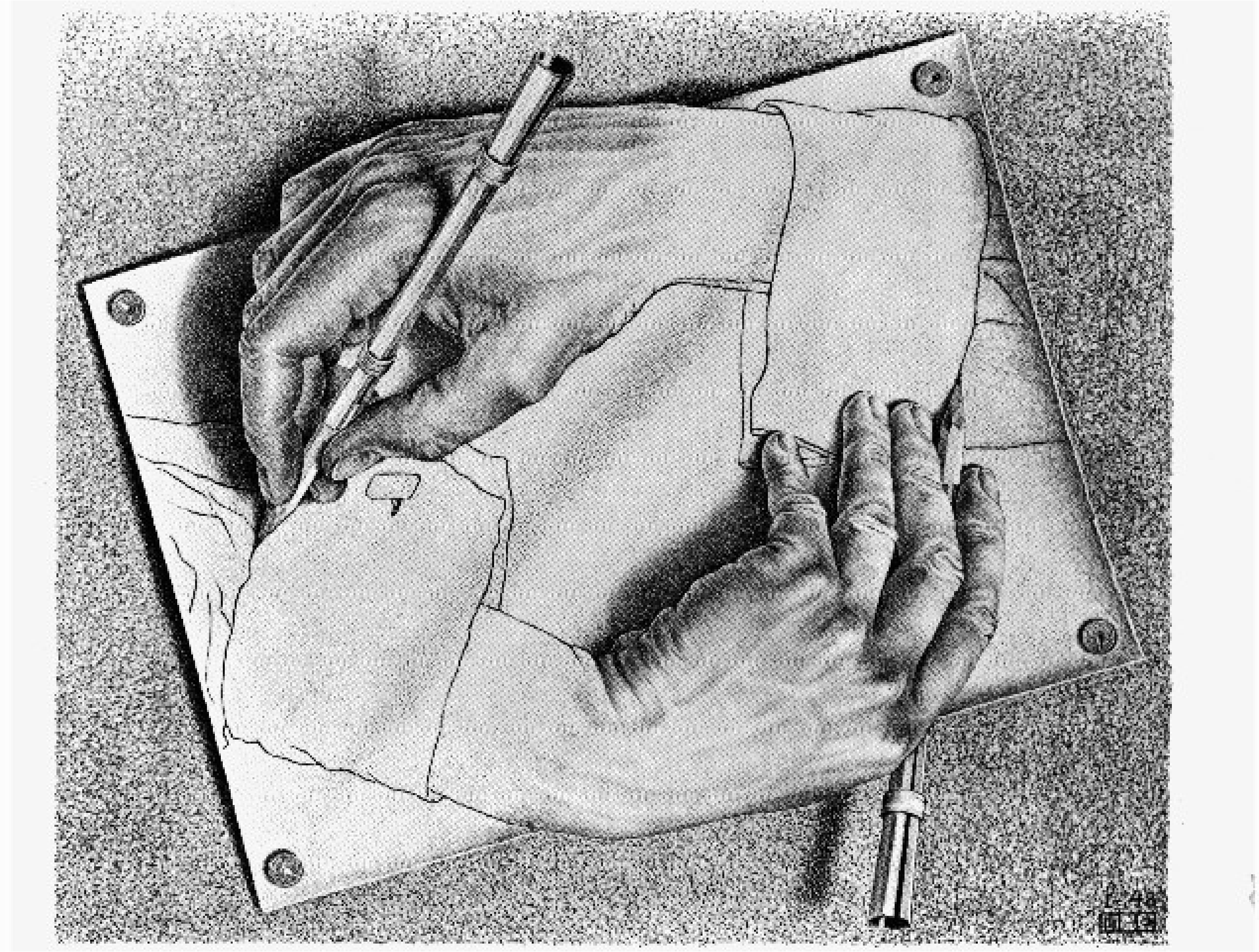
Ctx -> result

result

Reader monad

Wrap up
Toward a functional domain model

API design is an iterative process



Strive for immutability

```
private final Object obj;
```

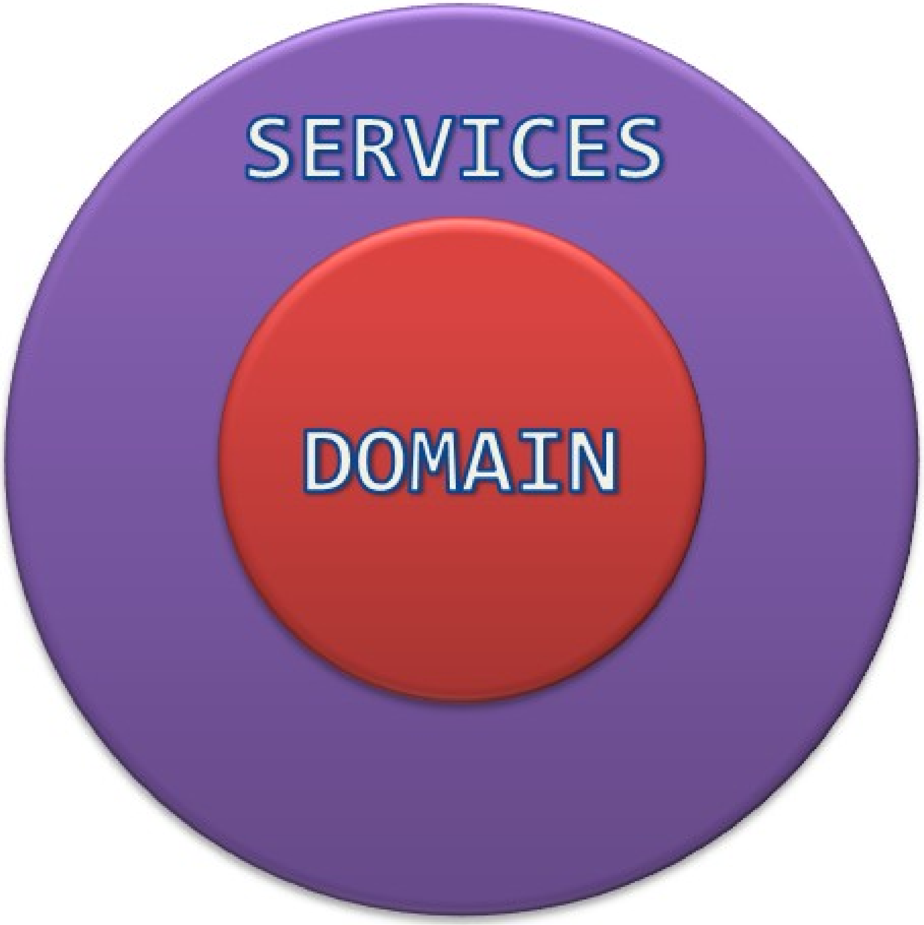
Confine side-effects

Avoid using exceptions for error handling

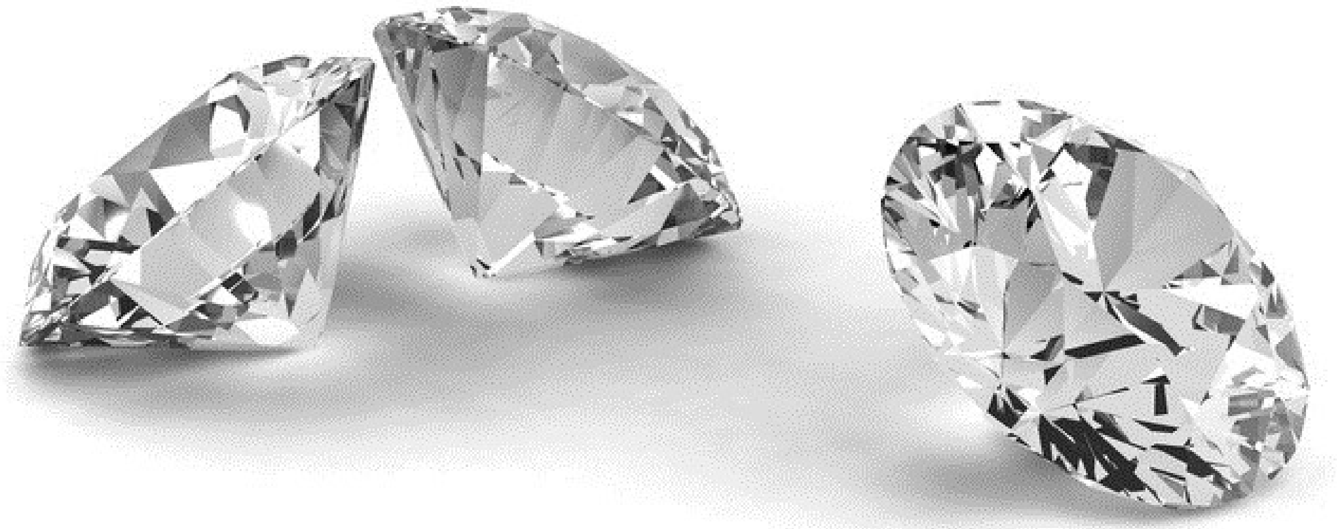
Say it with types

```
Tuple3<
    Function<
       BankConnection,
       Try<Account>
    >,
    Optional<Address>,
    Future<
        List<Withdrawal>
    >
>
```

Use anemic object



Put domain logic in pure functions



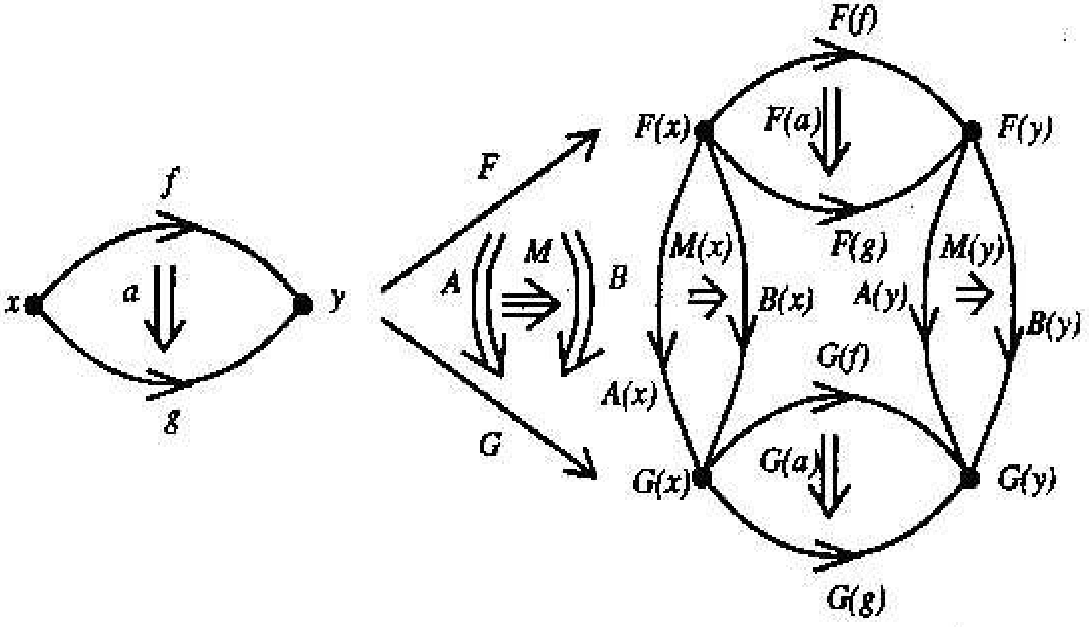
FP allows better Reusability & Composability

$$\frac{\text{OOP}}{\text{FP}} =$$

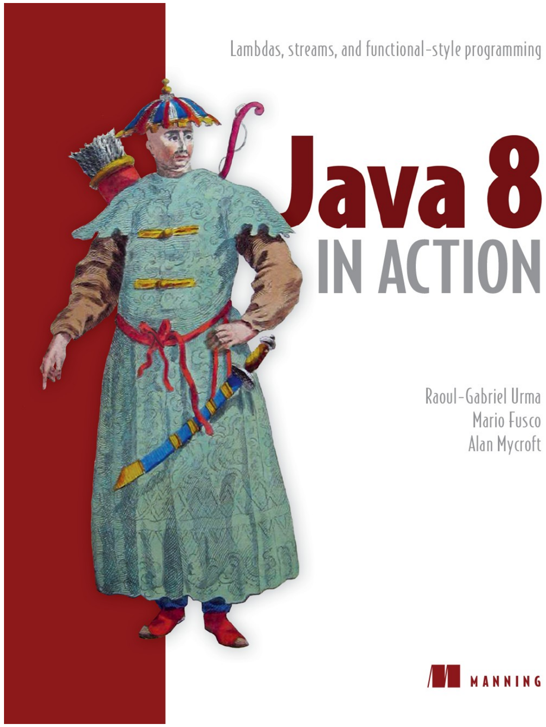
Throw away your GoF copy ...

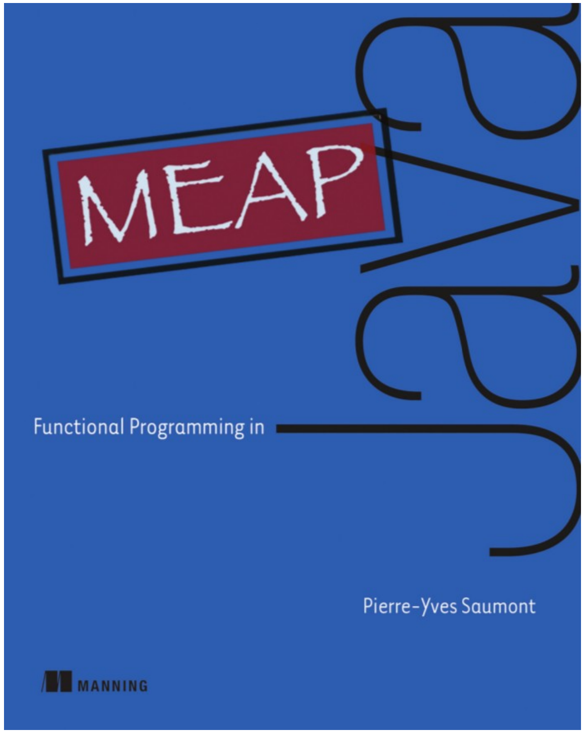
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch

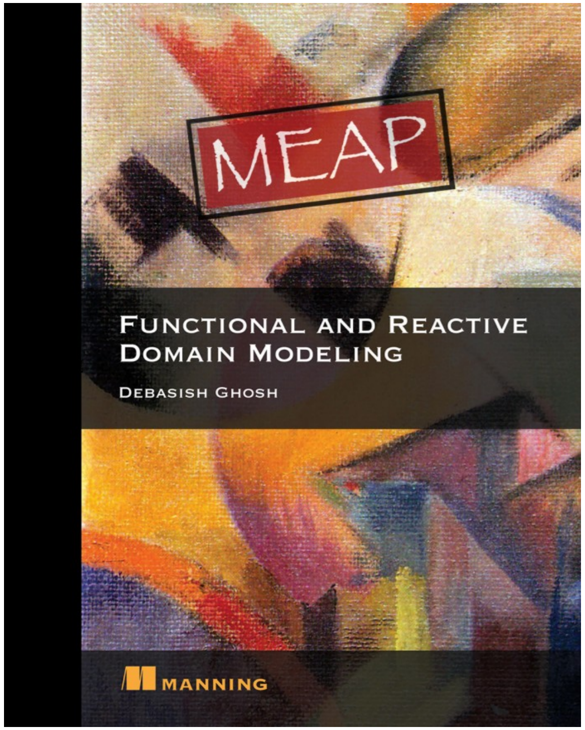
… and learn some functional patterns



Suggested readings

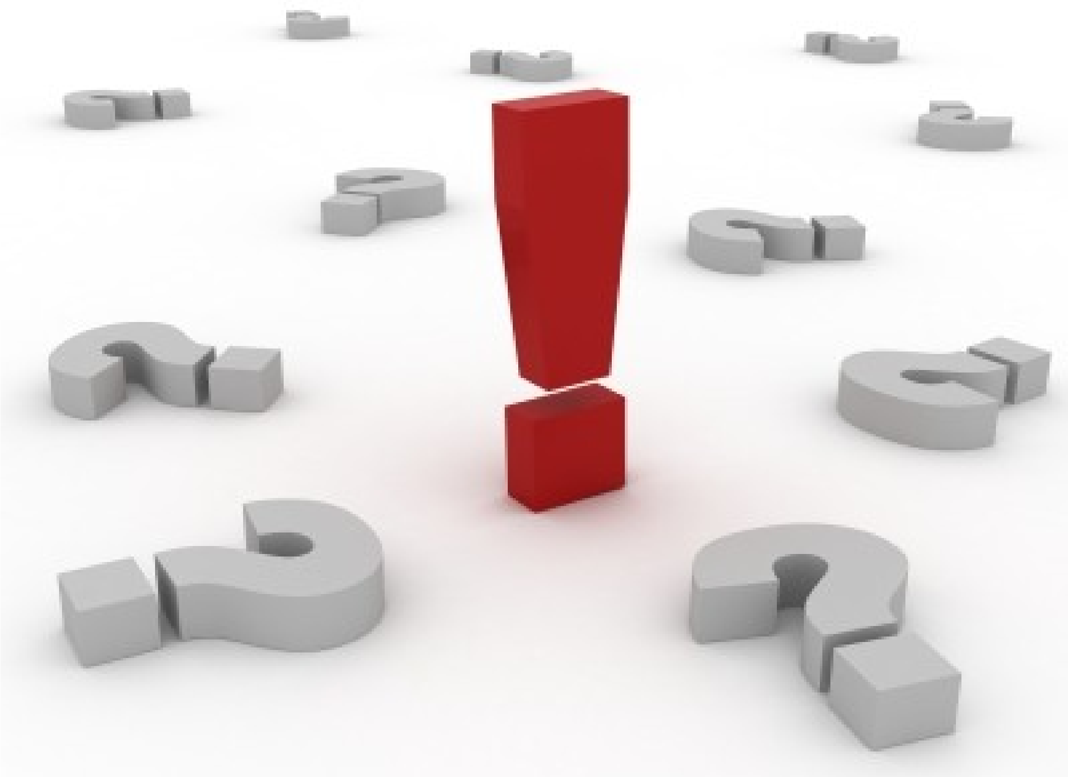






Thanks … Questions?

Q



A

Mario Fusco
Red Hat – Senior Software Engineer

mario.fusco@gmail.com
twitter: @mariofusco