

by Mario Fusco

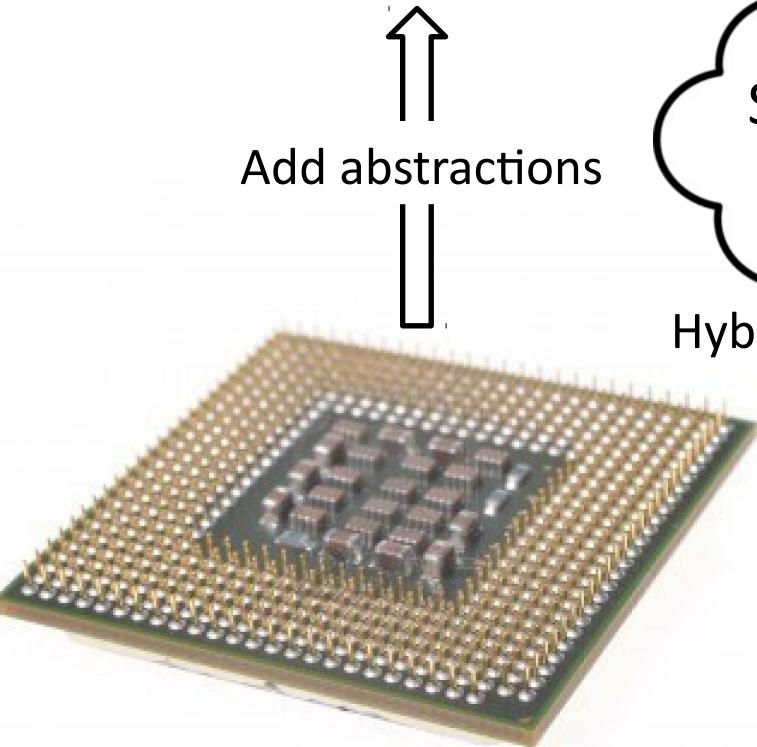
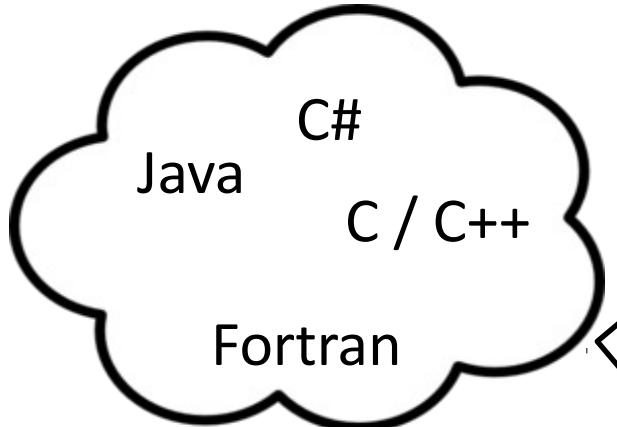
mario.fusco@gmail.com

twitter: @mariofusco

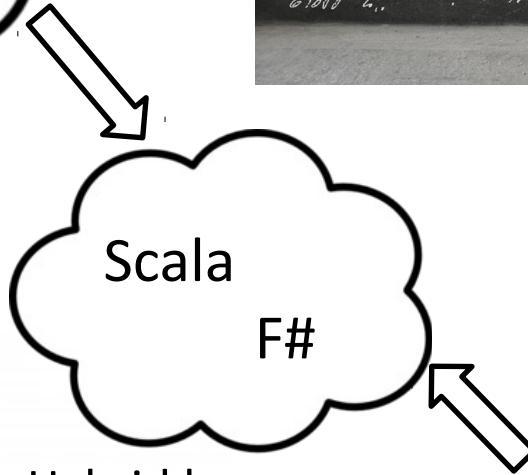


Monadics  
Java™

## Imperative languages

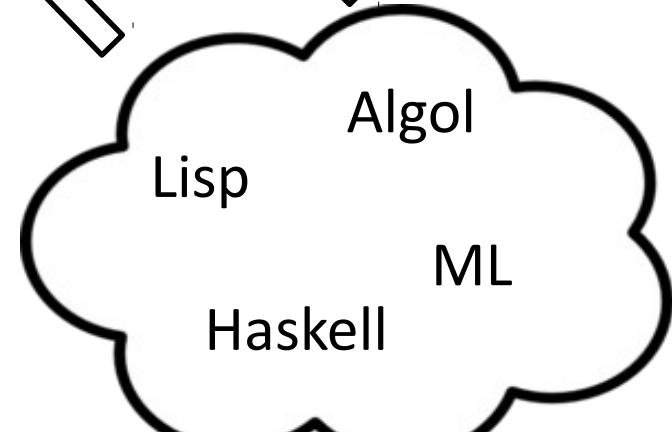


Add abstractions



Hybrid languages

Subtract abstractions



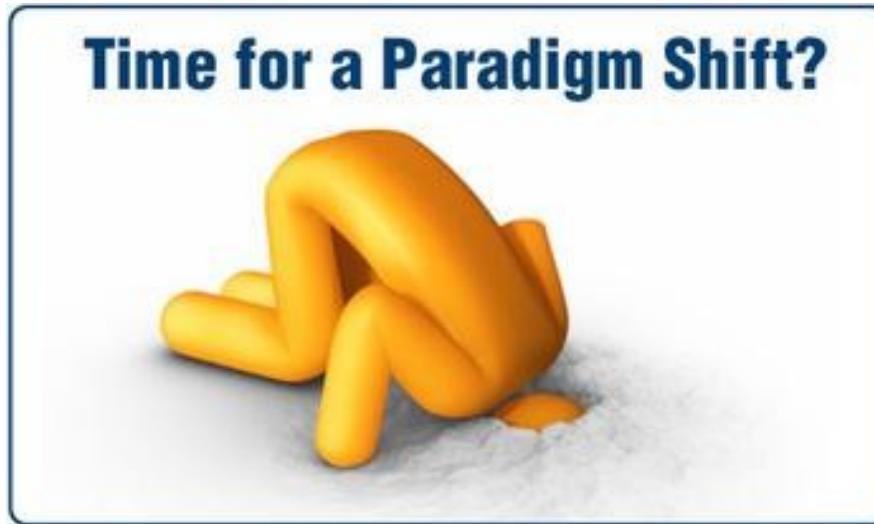
Functional languages

A blackboard filled with mathematical equations and calculations, including:  
70000 + 60000 + 70000 = 200000  
Parallel sections N, 76 = 800000 + 75.0 \* 20000 + 50000 + 75000 + 45000 + 55000 + 65000 + 85000 =  
Ex4 action : 86.000 + 56000 + 15.000 + 65.000 ÷ (4500.000 + 75.000 + 85.000) + 5000 + 4000 =  
10.000 + 11.500 + 12.500 + 65.000 + 75.000 + 150.000 + 165.000 + 840.000 + 650.000 =  
450000 + 56000 + 500000 + 85000 (3%) <= H - (Ez + VEz) < 45.000 Dz ~ 10  
Ez Ie Cz (x+dz) 1 + N (log dz) 5000.000 + ... PN [1 + 1/dz + 1/(dz)^2] = Pdz (+)  
AP + P 500.000 + Xz Xz + dxz = LM 65.000 + 75.000 + 65000 ~ 900 x (1/dz - 1/z) 2, LM 6000  
XP Wz Up 1 65.000 xz + Pz xPz (V1 V< (x-z)^2 65.000 + 35.000 + 20000  
f(x) = 1/M (x+k)^2 = xz 1/M x^2 + Zx + 1/2dz - Y 5.1 5 = 110/1100  
65.000 + 75.000 + 65000 + 75000 + 5000 + 5000 + 5000 + 5000 f(4)z  
450000 → √ 56.000^2 Mdz/f(x) N 1/2 x 1/2 x [area + 20000 40000 50000 60000 70000 80000 90000 100000 110000 120000 130000 140000 150000 160000 170000 180000 190000 200000 210000 220000 230000 240000 250000 260000 270000 280000 290000 300000 310000 320000 330000 340000 350000 360000 370000 380000 390000 400000 410000 420000 430000 440000 450000 460000 470000 480000 490000 500000 510000 520000 530000 540000 550000 560000 570000 580000 590000 600000 610000 620000 630000 640000 650000 660000 670000 680000 690000 700000 710000 720000 730000 740000 750000 760000 770000 780000 790000 800000 810000 820000 830000 840000 850000 860000 870000 880000 890000 900000 910000 920000 930000 940000 950000 960000 970000 980000 990000 1000000 1010000 1020000 1030000 1040000 1050000 1060000 1070000 1080000 1090000 1100000 1110000 1120000 1130000 1140000 1150000 1160000 1170000 1180000 1190000 1200000 1210000 1220000 1230000 1240000 1250000 1260000 1270000 1280000 1290000 1300000 1310000 1320000 1330000 1340000 1350000 1360000 1370000 1380000 1390000 1400000 1410000 1420000 1430000 1440000 1450000 1460000 1470000 1480000 1490000 1500000 1510000 1520000 1530000 1540000 1550000 1560000 1570000 1580000 1590000 1600000 1610000 1620000 1630000 1640000 1650000 1660000 1670000 1680000 1690000 1700000 1710000 1720000 1730000 1740000 1750000 1760000 1770000 1780000 1790000 1800000 1810000 1820000 1830000 1840000 1850000 1860000 1870000 1880000 1890000 1900000 1910000 1920000 1930000 1940000 1950000 1960000 1970000 1980000 1990000 2000000 2010000 2020000 2030000 2040000 2050000 2060000 2070000 2080000 2090000 2100000 2110000 2120000 2130000 2140000 2150000 2160000 2170000 2180000 2190000 2200000 2210000 2220000 2230000 2240000 2250000 2260000 2270000 2280000 2290000 2300000 2310000 2320000 2330000 2340000 2350000 2360000 2370000 2380000 2390000 2400000 2410000 2420000 2430000 2440000 2450000 2460000 2470000 2480000 2490000 2500000 2510000 2520000 2530000 2540000 2550000 2560000 2570000 2580000 2590000 2600000 2610000 2620000 2630000 2640000 2650000 2660000 2670000 2680000 2690000 2700000 2710000 2720000 2730000 2740000 2750000 2760000 2770000 2780000 2790000 2800000 2810000 2820000 2830000 2840000 2850000 2860000 2870000 2880000 2890000 2900000 2910000 2920000 2930000 2940000 2950000 2960000 2970000 2980000 2990000 3000000 3010000 3020000 3030000 3040000 3050000 3060000 3070000 3080000 3090000 3100000 3110000 3120000 3130000 3140000 3150000 3160000 3170000 3180000 3190000 3200000 3210000 3220000 3230000 3240000 3250000 3260000 3270000 3280000 3290000 3300000 3310000 3320000 3330000 3340000 3350000 3360000 3370000 3380000 3390000 3400000 3410000 3420000 3430000 3440000 3450000 3460000 3470000 3480000 3490000 3500000 3510000 3520000 3530000 3540000 3550000 3560000 3570000 3580000 3590000 3600000 3610000 3620000 3630000 3640000 3650000 3660000 3670000 3680000 3690000 3700000 3710000 3720000 3730000 3740000 3750000 3760000 3770000 3780000 3790000 3800000 3810000 3820000 3830000 3840000 3850000 3860000 3870000 3880000 3890000 3900000 3910000 3920000 3930000 3940000 3950000 3960000 3970000 3980000 3990000 4000000 4010000 4020000 4030000 4040000 4050000 4060000 4070000 4080000 4090000 4100000 4110000 4120000 4130000 4140000 4150000 4160000 4170000 4180000 4190000 4200000 4210000 4220000 4230000 4240000 4250000 4260000 4270000 4280000 4290000 4300000 4310000 4320000 4330000 4340000 4350000 4360000 4370000 4380000 4390000 4400000 4410000 4420000 4430000 4440000 4450000 4460000 4470000 4480000 4490000 4500000 4510000 4520000 4530000 4540000 4550000 4560000 4570000 4580000 4590000 4600000 4610000 4620000 4630000 4640000 4650000 4660000 4670000 4680000 4690000 4700000 4710000 4720000 4730000 4740000 4750000 4760000 4770000 4780000 4790000 4800000 4810000 4820000 4830000 4840000 4850000 4860000 4870000 4880000 4890000 4900000 4910000 4920000 4930000 4940000 4950000 4960000 4970000 4980000 4990000 5000000 5010000 5020000 5030000 5040000 5050000 5060000 5070000 5080000 5090000 5100000 5110000 5120000 5130000 5140000 5150000 5160000 5170000 5180000 5190000 5200000 5210000 5220000 5230000 5240000 5250000 5260000 5270000 5280000 5290000 5300000 5310000 5320000 5330000 5340000 5350000 5360000 5370000 5380000 5390000 5400000 5410000 5420000 5430000 5440000 5450000 5460000 5470000 5480000 5490000 5500000 5510000 5520000 5530000 5540000 5550000 5560000 5570000 5580000 5590000 5600000 5610000 5620000 5630000 5640000 5650000 5660000 5670000 5680000 5690000 5700000 5710000 5720000 5730000 5740000 5750000 5760000 5770000 5780000 5790000 5800000 5810000 5820000 5830000 5840000 5850000 5860000 5870000 5880000 5890000 5900000 5910000 5920000 5930000 5940000 5950000 5960000 5970000 5980000 5990000 6000000 6010000 6020000 6030000 6040000 6050000 6060000 6070000 6080000 6090000 6100000 6110000 6120000 6130000 6140000 6150000 6160000 6170000 6180000 6190000 6200000 6210000 6220000 6230000 6240000 6250000 6260000 6270000 6280000 6290000 6300000 6310000 6320000 6330000 6340000 6350000 6360000 6370000 6380000 6390000 6400000 6410000 6420000 6430000 6440000 6450000 6460000 6470000 6480000 6490000 6500000 6510000 6520000 6530000 6540000 6550000 6560000 6570000 6580000 6590000 6600000 6610000 6620000 6630000 6640000 6650000 6660000 6670000 6680000 6690000 6700000 6710000 6720000 6730000 6740000 6750000 6760000 6770000 6780000 6790000 6800000 6810000 6820000 6830000 6840000 6850000 6860000 6870000 6880000 6890000 6900000 6910000 6920000 6930000 6940000 6950000 6960000 6970000 6980000 6990000 7000000 7010000 7020000 7030000 7040000 7050000 7060000 7070000 7080000 7090000 7100000 7110000 7120000 7130000 7140000 7150000 7160000 7170000 7180000 7190000 7200000 7210000 7220000 7230000 7240000 7250000 7260000 7270000 7280000 7290000 7300000 7310000 7320000 7330000 7340000 7350000 7360000 7370000 7380000 7390000 7400000 7410000 7420000 7430000 7440000 7450000 7460000 7470000 7480000 7490000 7500000 7510000 7520000 7530000 7540000 7550000 7560000 7570000 7580000 7590000 7600000 7610000 7620000 7630000 7640000 7650000 7660000 7670000 7680000 7690000 7700000 7710000 7720000 7730000 7740000 7750000 7760000 7770000 7780000 7790000 7800000 7810000 7820000 7830000 7840000 7850000 7860000 7870000 7880000 7890000 7900000 7910000 7920000 7930000 7940000 7950000 7960000 7970000 7980000 7990000 8000000 8010000 8020000 8030000 8040000 8050000 8060000 8070000 8080000 8090000 8100000 8110000 8120000 8130000 8140000 8150000 8160000 8170000 8180000 8190000 8200000 8210000 8220000 8230000 8240000 8250000 8260000 8270000 8280000 8290000 8300000 8310000 8320000 8330000 8340000 8350000 8360000 8370000 8380000 8390000 8400000 8410000 8420000 8430000 8440000 8450000 8460000 8470000 8480000 8490000 8500000 8510000 8520000 8530000 8540000 8550000 8560000 8570000 8580000 8590000 8600000 8610000 8620000 8630000 8640000 8650000 8660000 8670000 8680000 8690000 8700000 8710000 8720000 8730000 8740000 8750000 8760000 8770000 8780000 8790000 8800000 8810000 8820000 8830000 8840000 8850000 8860000 8870000 8880000 8890000 8900000 8910000 8920000 8930000 8940000 8950000 8960000 8970000 8980000 8990000 9000000 9010000 9020000 9030000 9040000 9050000 9060000 9070000 9080000 9090000 9100000 9110000 9120000 9130000 9140000 9150000 9160000 9170000 9180000 9190000 9200000 9210000 9220000 9230000 9240000 9250000 9260000 9270000 9280000 9290000 9300000 9310000 9320000 9330000 9340000 9350000 9360000 9370000 9380000 9390000 9400000 9410000 9420000 9430000 9440000 9450000 9460000 9470000 9480000 9490000 9500000 9510000 9520000 9530000 9540000 9550000 9560000 9570000 9580000 9590000 9600000 9610000 9620000 9630000 9640000 9650000 9660000 9670000 9680000 9690000 9700000 9710000 9720000 9730000 9740000 9750000 9760000 9770000 9780000 9790000 9800000 9810000 9820000 9830000 9840000 9850000 9860000 9870000 9880000 9890000 9900000 9910000 9920000 9930000 9940000 9950000 9960000 9970000 9980000 9990000 10000000

A person is standing at the chalkboard, writing more equations.

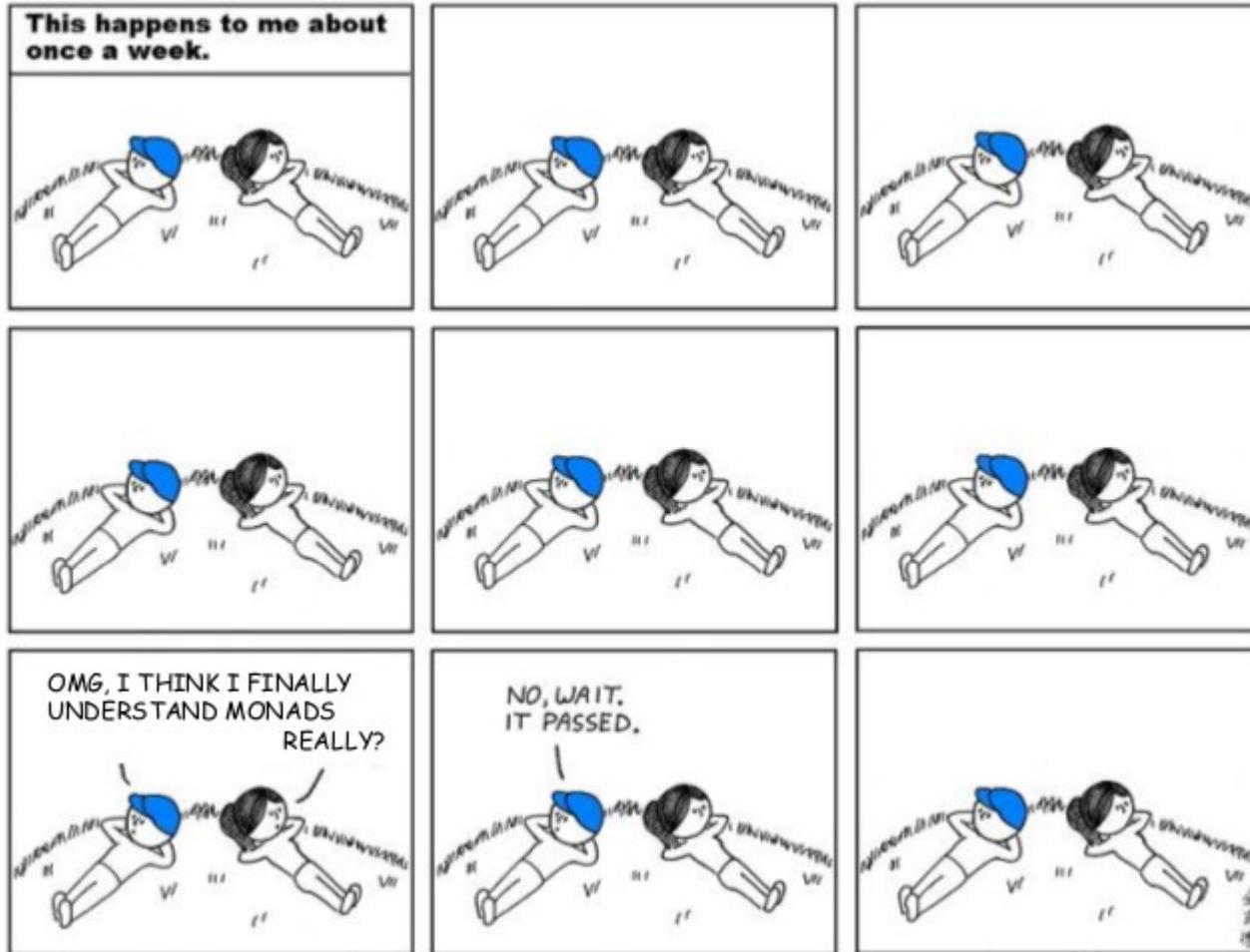
# **new language < new paradigm**

Learning a new language is relatively easy compared with learning a new paradigm.



Functional Programming is more a new way of thinking than a new tool set

# What is a monad?



# What is a monad?

A monad is a triple  $(T, \eta, \mu)$  where  $T$  is an endofunctor  $T: X \rightarrow X$  and  $\eta: I \rightarrow T$  and  $\mu: T \times T \rightarrow T$  are 2 natural transformations satisfying these laws:

Identity law:  $\mu(\eta(T)) = T = \mu(T(\eta))$

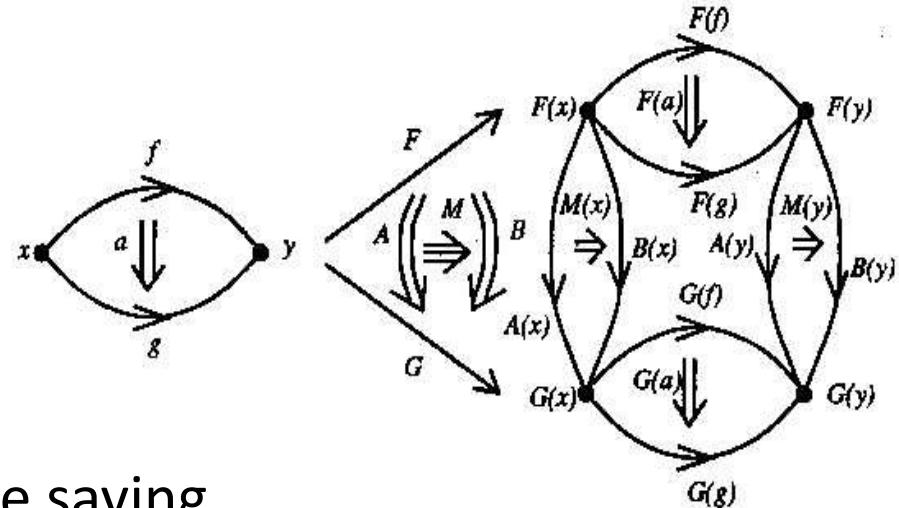
Associative law:  $\mu(\mu(T \times T) \times T)) = \mu(T \times \mu(T \times T))$

In other words: "*a monad in X is just a monoid in the category of endofunctors of X, with product  $\times$  replaced by composition of endofunctors and unit set by the identity endofunctor*"

## What's the problem?

# ... really? do I need to know this?

In order to understand monads you need to first learn Category Theory



... it's like saying ...



In order to understand pizza you need to first learn Italian

# ... ok, so let's try to ask Google ...



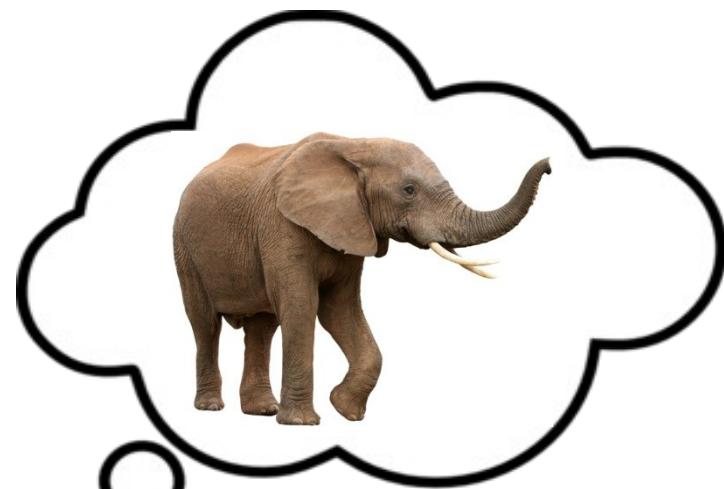
monad is a monoid in the category of endofunctors

monad is **a monoid in the category of endofunctors**

monad is **a burrito**

monad is **a functor**

monad is **an elephant**



**... no seriously, what is a monad?**

A  
**monad**  
is a  
**structure**  
that puts a  
**value**  
in a  
**computational context**

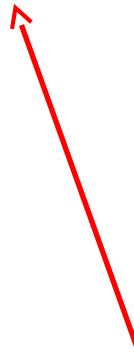
# **... and why should we care about?**

- Reduce code duplication
- Improve maintainability
- Increase readability
- Remove side effects
- Hide complexity
- Encapsulate implementation details
- Allow composability

# Monadic Methods

```
M<A> unit(A a);  
M<B> bind(M<A> ma, Function<A, M<B>> f);
```

```
interface M {  
    M<B> map(Function<A, B> f){  
        return flatMap( x -> unit( f.apply(x) ) );  
    }  
  
    M<B> flatMap(Function<A, M<B>> f);  
}
```



map can be defined for every monad as  
a combination of flatMap and unit

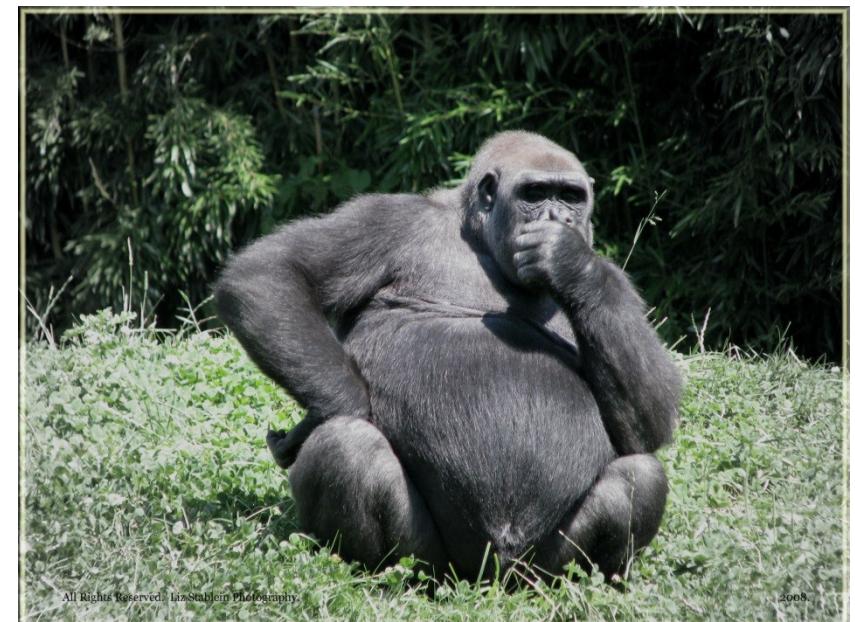
# Finding Car's Insurance Name

```
public class Person {  
    private Car car;  
    public Car getCar() { return car; }  
}  
  
public class Car {  
    private Insurance insurance;  
    public Insurance getInsurance() { return insurance; }  
}  
  
public class Insurance {  
    private String name;  
    public String getName() { return name; }  
}
```



# Attempt 1: deep doubts

```
String getCarInsuranceName(Person person) {  
    if (person != null) {  
        Car car = person.getCar();  
        if (car != null) {  
            Insurance insurance = car.getInsurance();  
            if (insurance != null) {  
                return insurance.getName()  
            }  
        }  
    }  
    return "Unknown";  
}
```



# Attempt 2: too many choices



```
String getCarInsuranceName(Person person) {  
    if (person == null) {  
        return "Unknown";  
    }  
    Car car = person.getCar();  
    if (car == null) {  
        return "Unknown";  
    }  
    Insurance insurance = car.getInsurance();  
    if (insurance == null) {  
        return "Unknown";  
    }  
    return insurance.getName()  
}
```

# What wrong with nulls?

- ✗ **Errors source** → NPE is by far the most common exception in Java
- ✗ **Bloatware source** → Worsen readability by making necessary to fill our code with null checks
- ✗ **Breaks Java philosophy** → Java always hides pointers to developers, except in one case: the null pointer
- ✗ **A hole in the type system** → Null has the bottom type, meaning that it can be assigned to any reference type: this is a problem because, when propagated to another part of the system, you have no idea what that null was initially supposed to be
- ✗ **Meaningless** → Don't have any semantic meaning and in particular are the wrong way to model the absence of a value in a statically typed language

“Absence of a signal should never be used as a signal” - J. Bigelow, 1947

Tony Hoare, who invented the null reference in 1965 while working on an object oriented language called ALGOL W, called its invention his

**“billion dollar mistake”**

# Optional Monad to the rescue

```
public class Optional<T> {  
    private static final Optional<?> EMPTY = new Optional<>(null);  
    private final T value;  
  
    private Optional(T value) {  
        this.value = value;  
    }  
  
    public<U> Optional<U> map(Function<? super T, ? extends U> f) {  
        return value == null ? EMPTY : new Optional(f.apply(value));  
    }  
  
    public<U> Optional<U> flatMap(Function<? super T, Optional<U>> f) {  
        return value == null ? EMPTY : f.apply(value);  
    }  
}
```

# Rethinking our model

```
public class Person {  
    private Optional<Car> car;  
    public Optional<Car> getCar() { return car; }  
}
```

```
public class Car {  
    private Optional<Insurance> insurance;  
    public Optional<Insurance> getInsurance() { return insurance; }  
}
```

```
public class Insurance {  
    private String name;  
    public String getName() { return name; }  
}
```

Using the type system  
to model nullable value

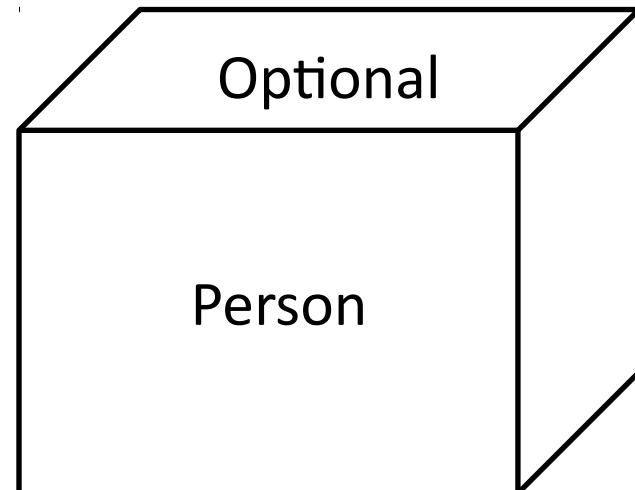
# Restoring the sanity

```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```



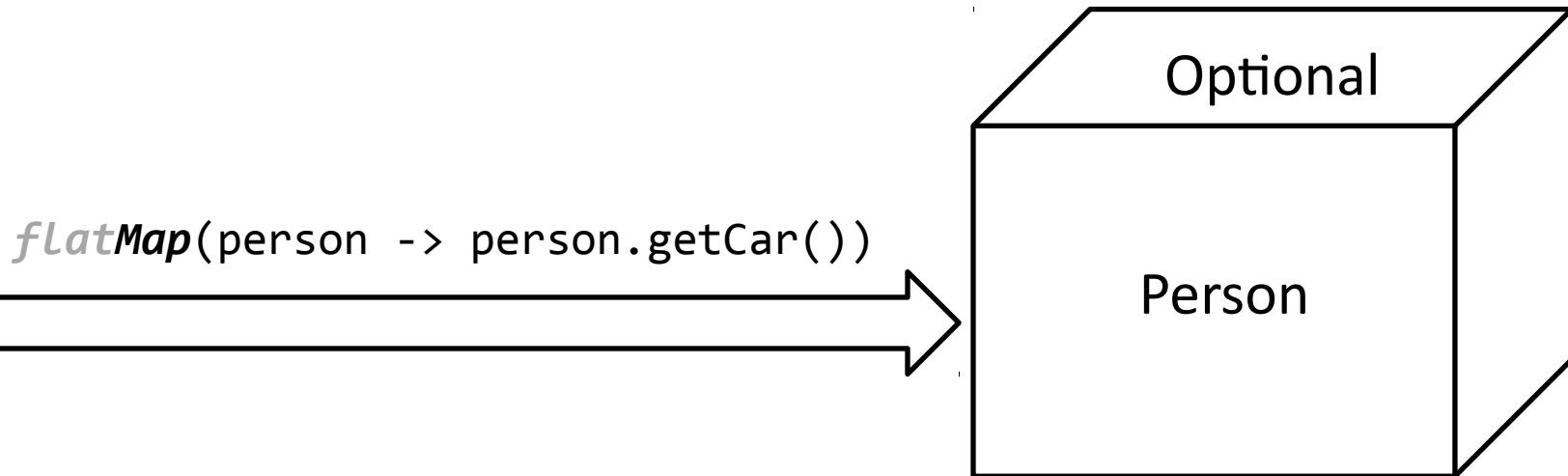
# Restoring the sanity

```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```



# Restoring the sanity

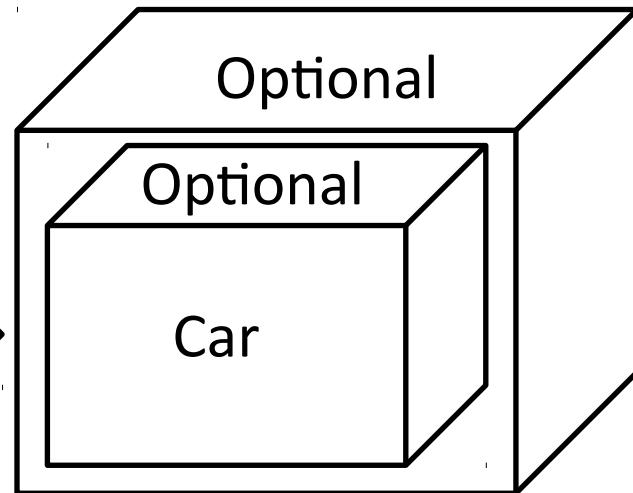
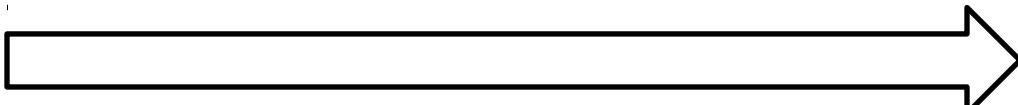
```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```



# Restoring the sanity

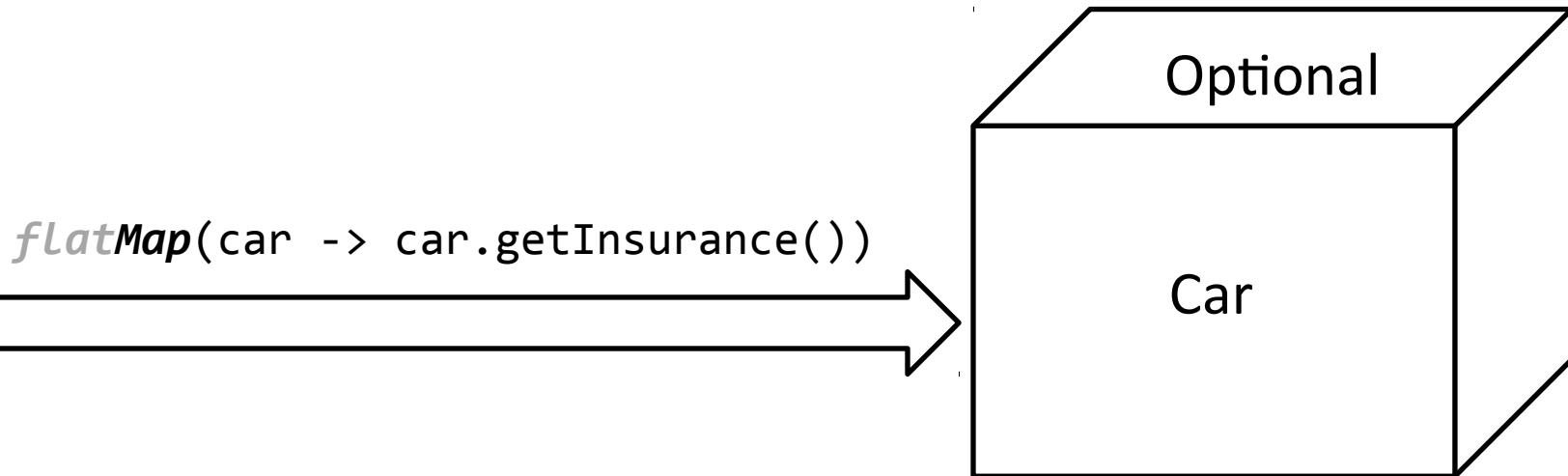
```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```

**flatMap(person -> person.getCar())**



# Restoring the sanity

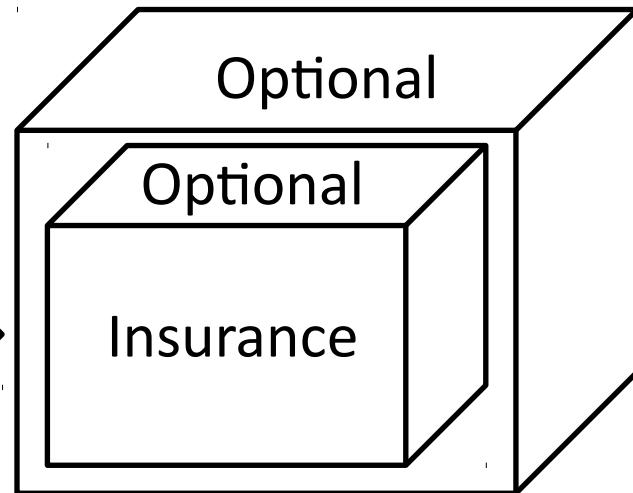
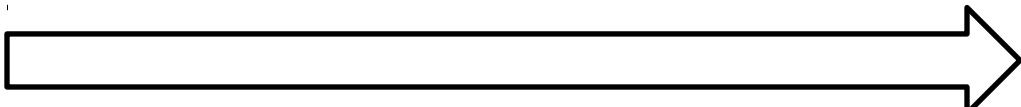
```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```



# Restoring the sanity

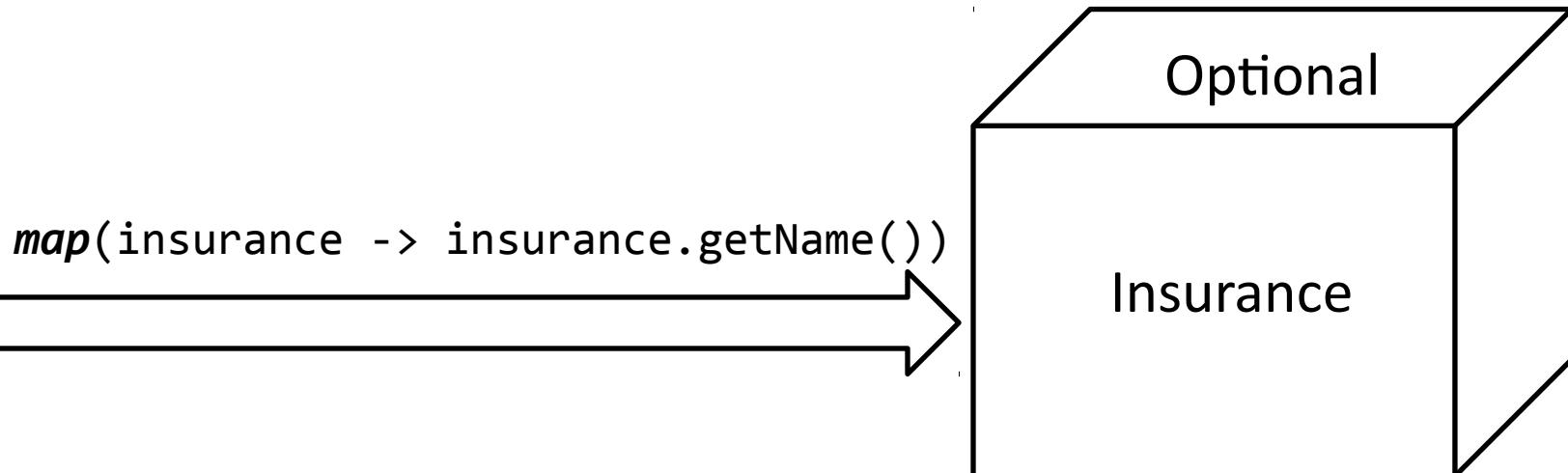
```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```

*flatMap(car -> car.getInsurance())*



# Restoring the sanity

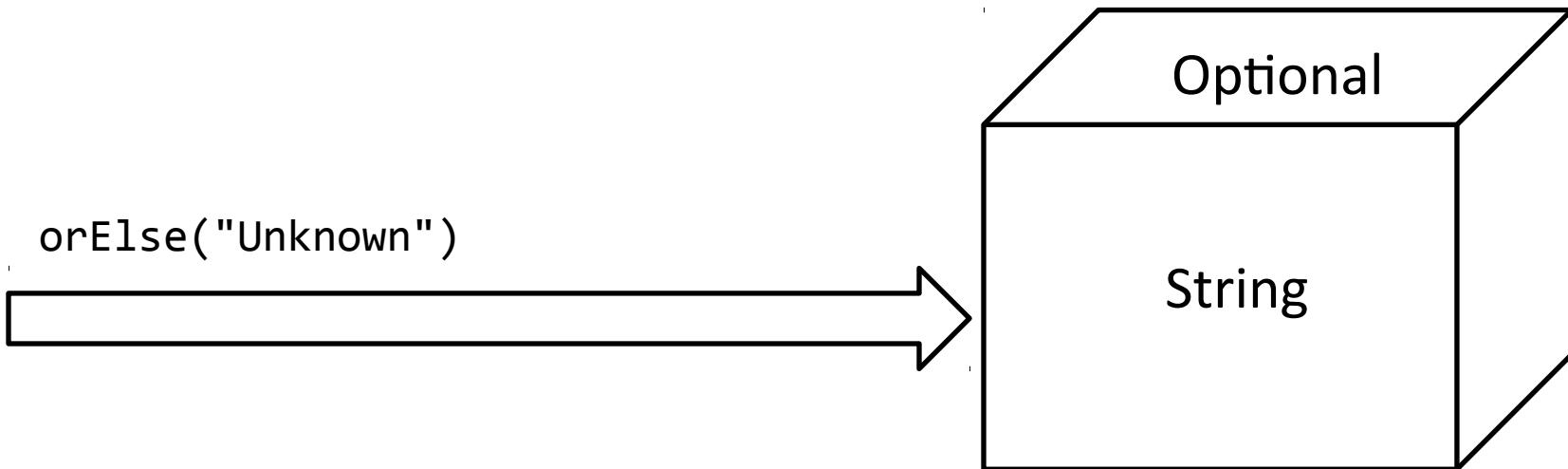
```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```



# Restoring the sanity

```
String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(person -> person.getCar())  
        .flatMap(car -> car.getInsurance())  
        .map(insurance -> insurance.getName())  
        .orElse("Unknown");  
}
```

orElse("Unknown")



# Why map and flatMap ?

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

person

```
.flatMap(Person::getCar)  
.flatMap(Car::getInsurance)  
.map(Insurance::getName)  
.orElse("Unknown");
```

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



Emergency Kittens

@EmergencyKittens



Follow

cat in a box, in a box.

[pic.twitter.com/5cHJN3NHvz](https://pic.twitter.com/5cHJN3NHvz)

Reply Retweet Favorite More



RETWEETS

246

FAVORITES

487



5:45 PM - 23 Mar 2014

Flag media

This is what happens when you don't use flatMap

# The Optional Monad

The Optional monad makes  
the possibility of missing data

*explicit*

in the type system, while

*hiding*

the boilerplate of "if non-null" logic



# Stream: another Java8 monad

## map

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an intermediate operation.

## flatMap

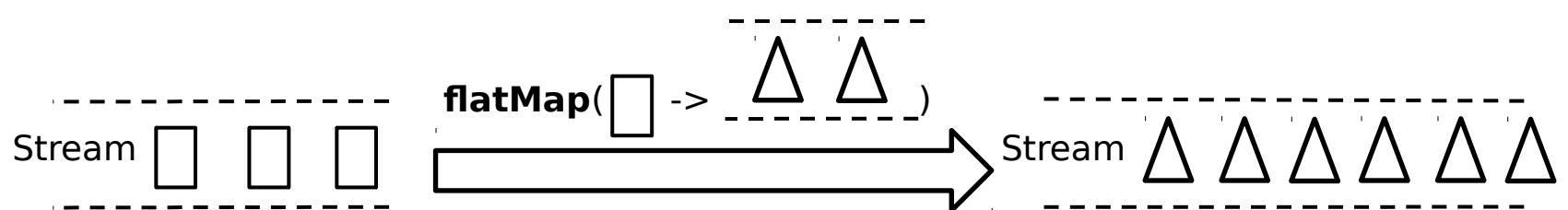
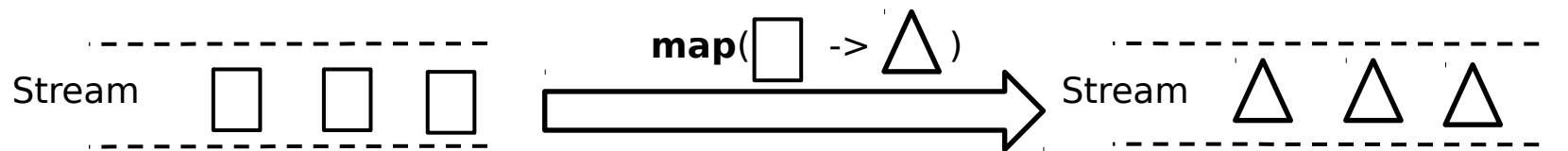
```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

Returns a stream consisting of the results of replacing each element of this stream with the contents of the stream produced by applying the provided mapping function to each element. (If the result of the mapping function is `null`, this is treated as if the result was an empty stream.)

This is an intermediate operation.

# Using map & flatMap with Streams

```
building.getApartments().stream().  
    .flatMap(apartment -> apartment.getPersons().stream())  
    .map(Person::getName);
```



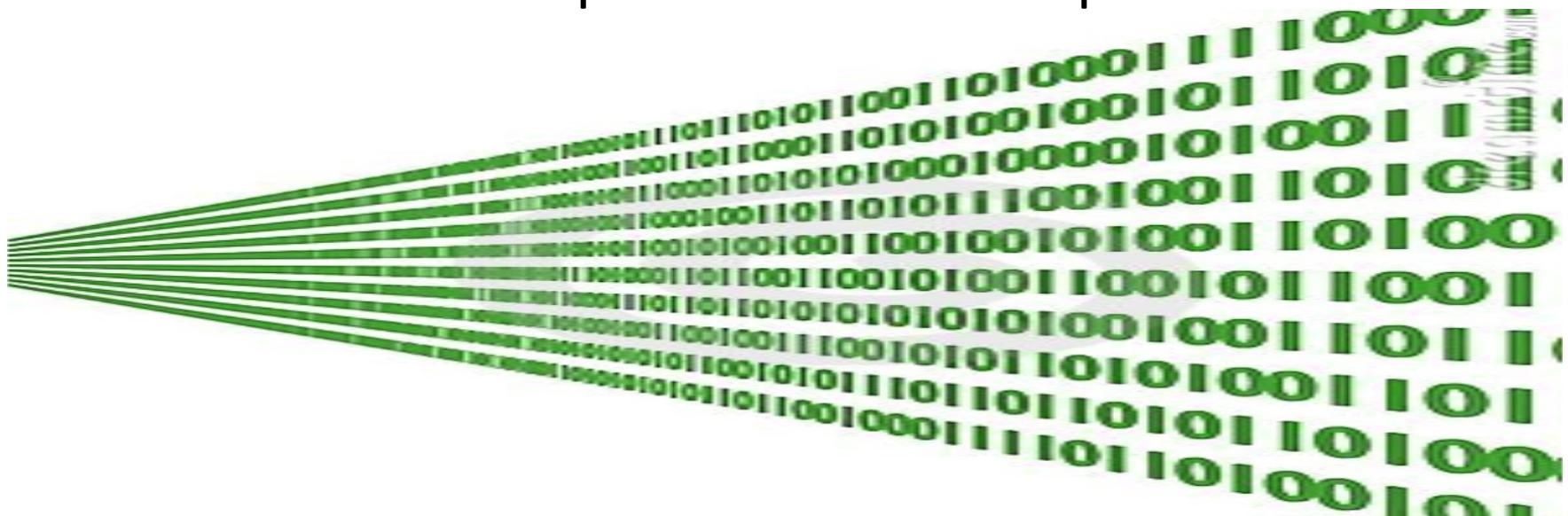
**Given  $n > 0$  find all pairs  $i$  and  $j$   
where  $1 \leq j \leq i \leq n$  and  $i+j$  is prime**

```
Stream.iterate(1, i -> i+1).limit(n)
    .flatMap(i -> Stream.iterate(1, j -> j+1).limit(n)
                .map(j -> new int[]{i, j}))
    .filter(pair -> isPrime(pair[0] + pair[1]))
    .collect(toList());
```

```
public boolean isPrime(int n) {
    return Stream.iterate(2, i -> i+1)
        .limit((long) Math.sqrt(n))
        .noneMatch(i -> n % i == 0);
}
```

# The Stream Monad

The Stream monad makes  
the possibility of multiple data  
*explicit*  
in the type system, while  
*hiding*  
the boilerplate of nested loops



# No Monads syntactic sugar in Java :(

```
for { i <- List.range(1, n)
      j <- List.range(1, i)
      if isPrime(i + j) } yield {i, j}
```



translated by the compiler in



```
List.range(1, n)
  .flatMap(i =>
    List.range(1, i)
      .filter(j => isPrime(i+j))
      .map(j => (i, j)))
```

Scala's for-comprehension  
is just syntactic sugar to  
manipulate monads



# Are there other monads in Java8 API?



# CompletableFuture

## thenApplyAsync

```
public <U> CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn)
```

Description copied from interface: [CompletionStage](#)

Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage's result as the argument to the supplied function. See the [CompletionStage](#) documentation for rules covering exceptional completion.

## thenComposeAsync

```
public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn)
```

Description copied from interface: [CompletionStage](#)

Returns a new CompletionStage that, when this stage completes normally, is executed using this stage's default asynchronous execution facility, with this stage as the argument to the supplied function. See the [CompletionStage](#) documentation for rules covering exceptional completion.

# Promise: a monadic CompletableFuture

```
public class Promise<A> implements Future<A> {  
    private final CompletableFuture<A> future;  
  
    private Promise(CompletableFuture<A> future) {  
        this.future = future;  
    }  
    public static final <A> Promise<A> promise(Supplier<A> supplier) {  
        return new  
            Promise<A>(CompletableFuture.supplyAsync(supplier));  
    }  
  
    public <B> Promise<B> map(Function<? super A, ? extends B> f) {  
        return new Promise<B>(future.thenApplyAsync(f));  
    }  
    public <B> Promise<B> flatMap(Function<? super A, Promise<B>> f) {  
        return new Promise<B>(  
            future.thenComposeAsync(a -> f.apply(a).future));  
    }  
    // ... omitting methods delegating the wrapped future  
}
```

# Composing long computations

```
public int slowLength(String s) {  
    someLongComputation();  
    return s.length();  
}
```

```
public int slowDouble(int i) {  
    someLongComputation();  
    return i*2;  
}
```

```
String s = "Hello";  
Promise<Integer> p = promise(() -> slowLength(s))  
    .flatMap(i -> promise(() -> slowDouble(i)));
```

# The Promise Monad

The Promise monad makes  
asynchronous computation  
*explicit*  
in the type system, while  
*hiding*  
the boilerplate thread logic



# Creating our own Monad



# Lost in Exceptions

```
public Person validateAge(Person p) throws ValidationException {  
    if (p.getAge() > 0 && p.getAge() < 130) return p;  
    throw new ValidationException("Age must be between 0 and 130");  
}  
  
public Person validateName(Person p) throws ValidationException {  
    if (Character.isUpperCase(p.getName().charAt(0))) return p;  
    throw new ValidationException("Name must start with uppercase");  
}  
  
List<String> errors = new ArrayList<String>();  
try {  
    validateAge(person);  
} catch (ValidationException ex) {  
    errors.add(ex.getMessage());  
}  
try {  
    validateName(person);  
} catch (ValidationException ex) {  
    errors.add(ex.getMessage());  
}
```

# Defining a Validation Monad

```
public abstract class Validation<L, A> {

    protected final A value;

    private Validation(A value) {
        this.value = value;
    }

    public abstract <B> Validation<L, B> map(
            Function<? super A, ? extends B> mapper);

    public abstract <B> Validation<L, B> flatMap(
            Function<? super A, Validation<?, ? extends B>> mapper);

    public abstract boolean isSuccess();
}
```

# Success !!!

```
public class Success<L, A> extends Validation<L, A> {  
    private Success(A value) { super(value); }  
  
    public <B> Validation<L, B> map(  
        Function<? super A, ? extends B> mapper) {  
        return success(mapper.apply(value));  
    }  
  
    public <B> Validation<L, B> flatMap(  
        Function<? super A, Validation<?, ? extends B>> mapper) {  
        return (Validation<L, B>) mapper.apply(value);  
    }  
  
    public boolean isSuccess() { return true; }  
  
    public static <L, A> Success<L, A> success(A value) {  
        return new Success<L, A>(value);  
    }  
}
```

# Failure :(((

```
public class Failure<L, A> extends Validation<L, A> {
    protected final L left;
    public Failure(A value, L left) {super(value); this.left = left;}

    public <B> Validation<L, B> map(
        Function<? super A, ? extends B> mapper) {
        return failure(left, mapper.apply(value));
    }

    public <B> Validation<L, B> flatMap(
        Function<? super A, Validation<?, ? extends B>> mapper) {
        Validation<?, ? extends B> result = mapper.apply(value);
        return result.isSuccess() ?
            failure(left, result.value) :
            failure(((Failure<L, B>)result).left, result.value);
    }

    public boolean isSuccess() { return false; }
}
```

# The Validation Monad

The Validation monad makes  
the possibility of errors  
*explicit*

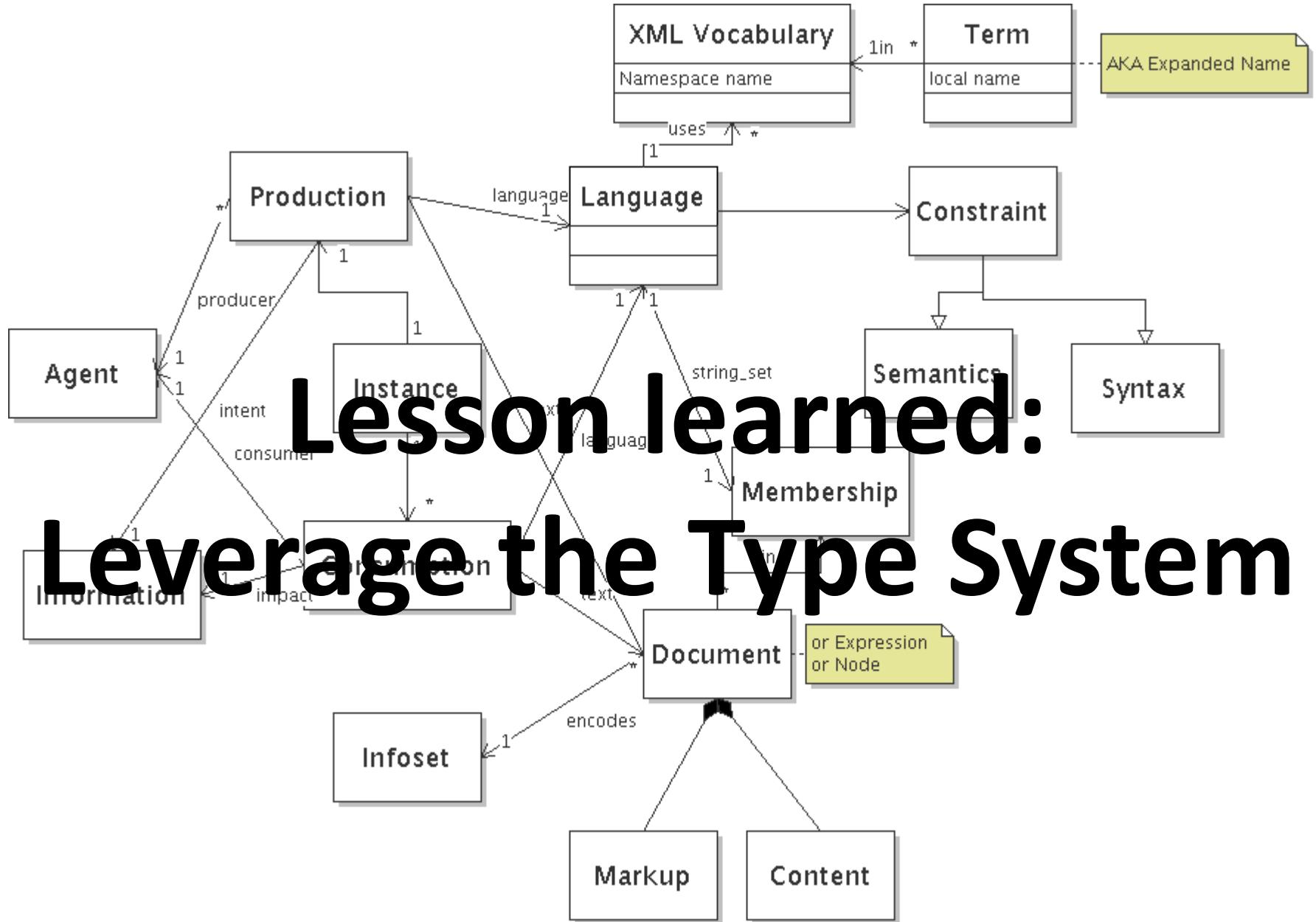
in the type system, while  
*hiding*  
the boilerplate of "try/catch" logic



# Rewriting validating methods

```
public Validation<String, Person> validateAge(Person p) {  
    return (p.getAge() > 0 && p.getAge() < 130) ?  
        success(p) :  
        failure("Age must be between 0 and 130", p);  
}
```

```
public Validation<String, Person> validateName(Person p) {  
    return Character.isUpperCase(p.getName().charAt(0)) ?  
        success(p) :  
        failure("Name must start with uppercase", p);  
}
```



# Gathering multiple errors - Success

```
public class SuccessList<L, A> extends Success<List<L>, A> {  
  
    public SuccessList(A value) { super(value); }  
  
    public <B> Validation<List<L>, B> map(  
        Function<? super A, ? extends B> mapper) {  
        return new SuccessList(mapper.apply(value));  
    }  
  
    public <B> Validation<List<L>, B> flatMap(  
        Function<? super A, Validation<?, ? extends B>> mapper) {  
        Validation<?, ? extends B> result = mapper.apply(value);  
        return (Validation<List<L>, B>)(result.isSuccess() ?  
            new SuccessList(result.value) :  
            new FailureList<L, B>(((Failure<L, B>)result).left,  
                result.value));  
    }  
}
```

# Gathering multiple errors - Failure

```
public class FailureList<L, A> extends Failure<List<L>, A> {  
  
    private FailureList(List<L> left, A value) { super(left, value); }  
  
    public <B> Validation<List<L>, B> map(  
        Function<? super A, ? extends B> mapper) {  
        return new FailureList(left, mapper.apply(value));  
    }  
  
    public <B> Validation<List<L>, B> flatMap(  
        Function<? super A, Validation<?, ? extends B>> mapper) {  
        Validation<?, ? extends B> result = mapper.apply(value);  
        return (Validation<List<L>, B>)(result.isSuccess() ?  
            new FailureList(left, result.value) :  
            new FailureList<L, B>(new ArrayList<L>(left) {{  
                add(((Failure<L, B>)result).left);  
            }}, result.value));  
    }  
}
```

# Monadic Validation

```
Validation<List<String>, Person>
    validatedPerson = success(person).failList()
        .flatMap(Validator::validAge)
        .flatMap(Validator::validName);
```



# Homework: develop your own Transaction Monad



The Transaction monad makes transactionally  
*explicit*  
in the type system, while  
*hiding*  
the boilerplate propagation of invoking rollbacks

# Alternative Monads Definitions

Monads are parametric types with two operations flatMap and unit that obey some algebraic laws

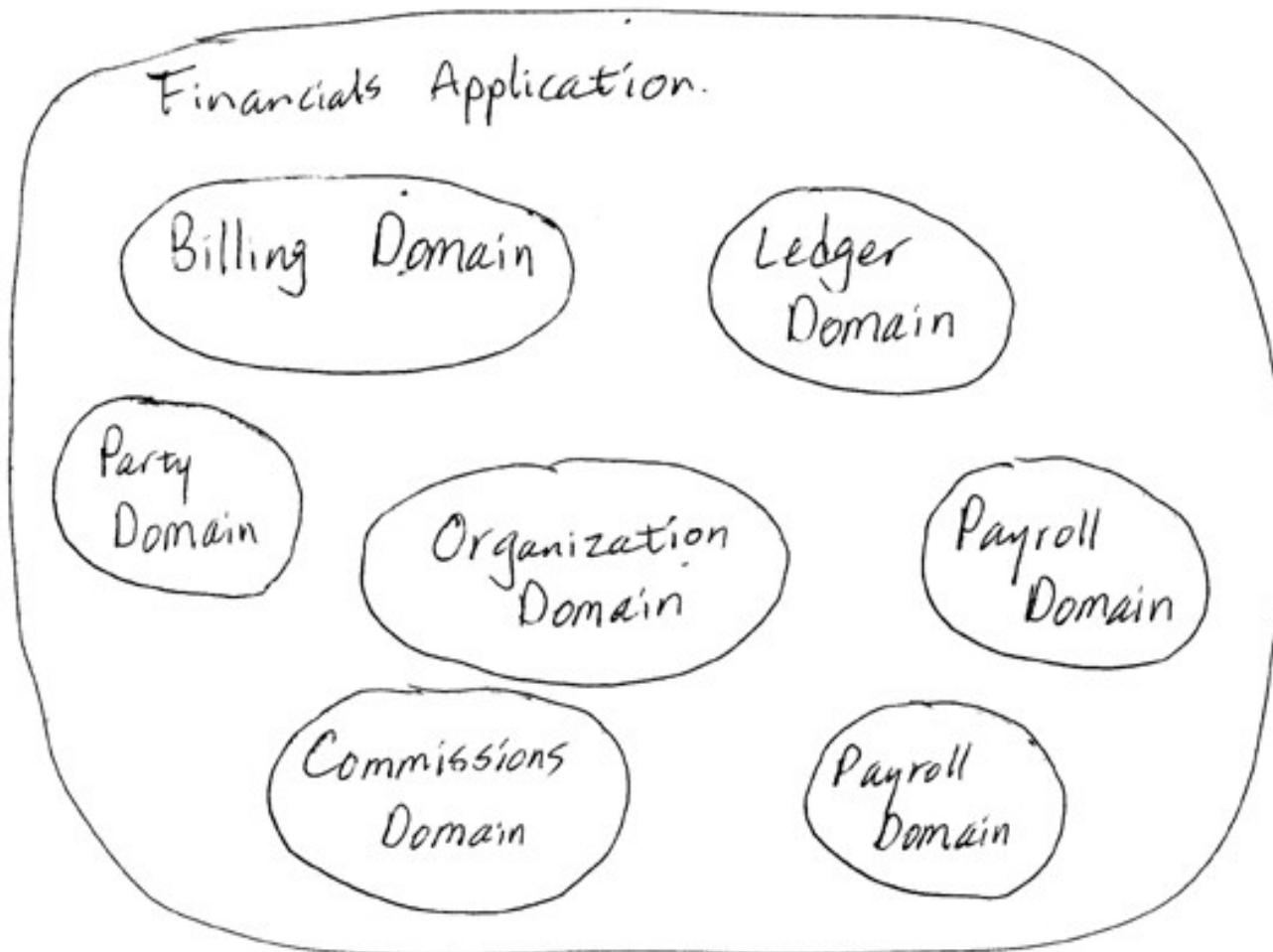
Monads are structures that represent computations defined as sequences of steps

Monads are chainable containers types that confine values defining how to transform and combine them

Monads are return types that guide you through the happy path

# Functional Domain Design

## A practical example



# 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 ) );  
    }  
}
```

The diagram illustrates three annotations pointing to specific parts of the `Account` class code:

- A line labeled "Mutability" points to the declaration of the `final` variable `balance`.
- A line labeled "Error handling using Exception" points to the `debit` method, specifically to the `throws` clause and the `throw` statement.
- A line labeled "Mutability" also points to the assignment in the `debit` method where the `balance` variable is updated.

# ... 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();
}

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<B>(f.apply(value));
    }
    public <B> Try<B> flatMap(Function<A, Try<B>> f) {
        return f.apply(value);
    }
}

public Failure<A> implements Try<T> {
    private final Object error;
    public Failure(Object error) { this.error = error; }
    public boolean isFailure() { return false; }
    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;<-- Immutable  
  
    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) ← without Exceptions  
            return new Failure<>( new InsufficientBalanceError() );  
        return new Success<>(  
            new Account( owner, number,  
                        new Balance( balance.amount.subtract( value ) ) ) );  
    }  
}
```

# ... 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 ) ) ) );  
    }  
}
```

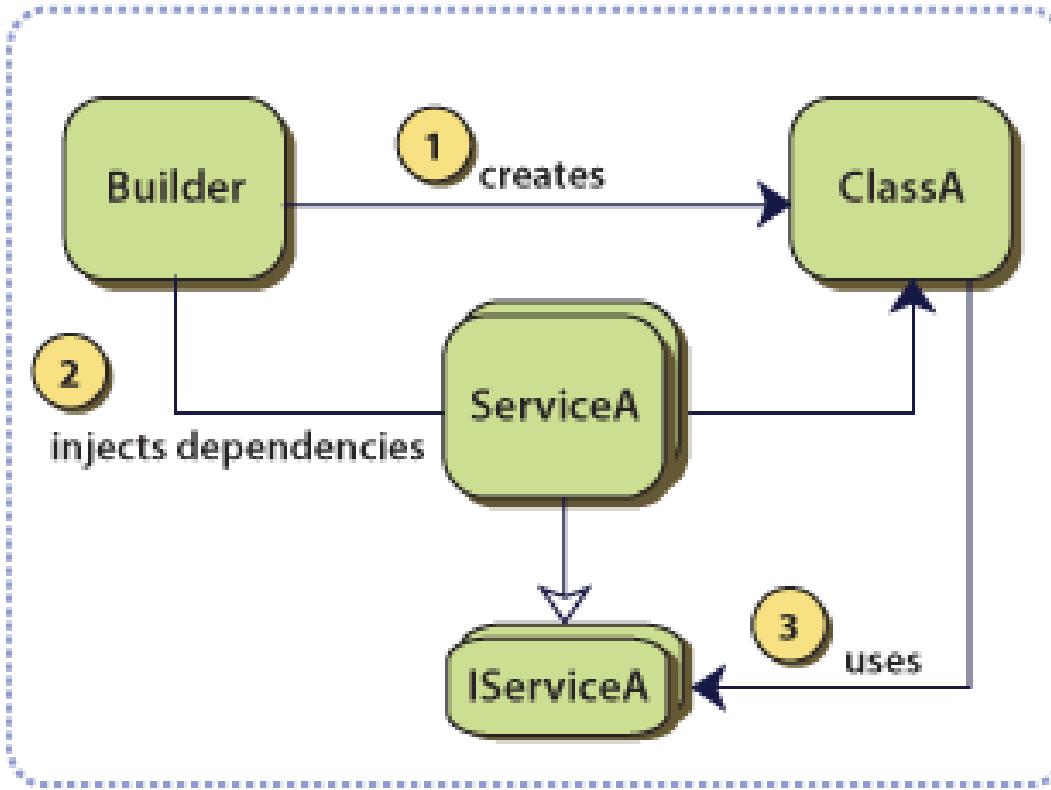
# Decoupling state and behavior

```
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 ...

```
BankConnection bconn = new BankConnection();
```

```
Try<Account> account =
```

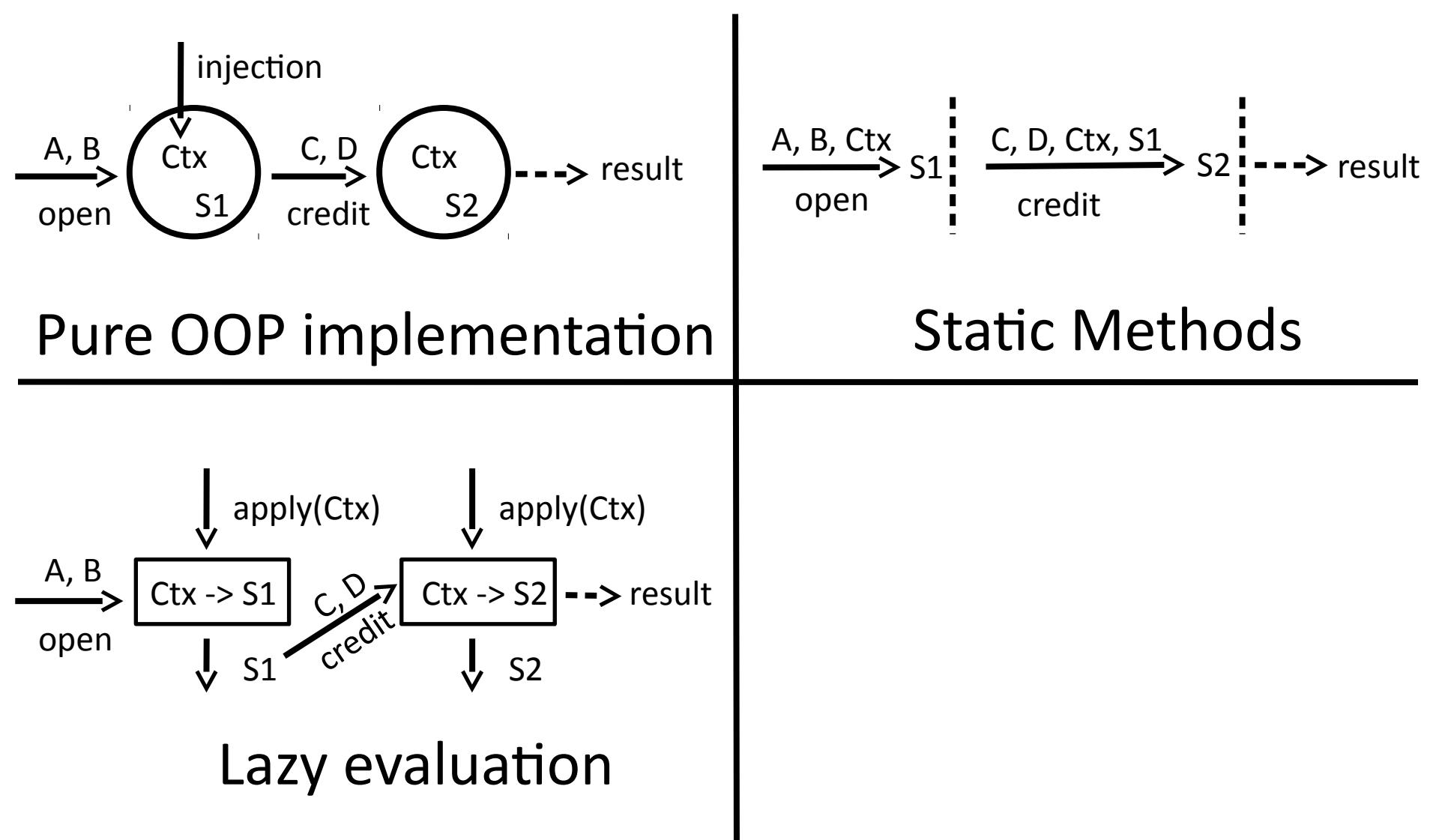
```
    open( "Alice", "123", new BigDecimal( 100.00 ), bconn )  
    .map( acc -> credit( acc, new BigDecimal( 200.00 ), bconn ) )  
    .map( acc -> credit( acc, new BigDecimal( 300.00 ), bconn ) )  
    .flatMap( acc -> debit( acc, new BigDecimal( 400.00 ), bconn ) );
```

... and pass it to all methods



# 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() );
```



# 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

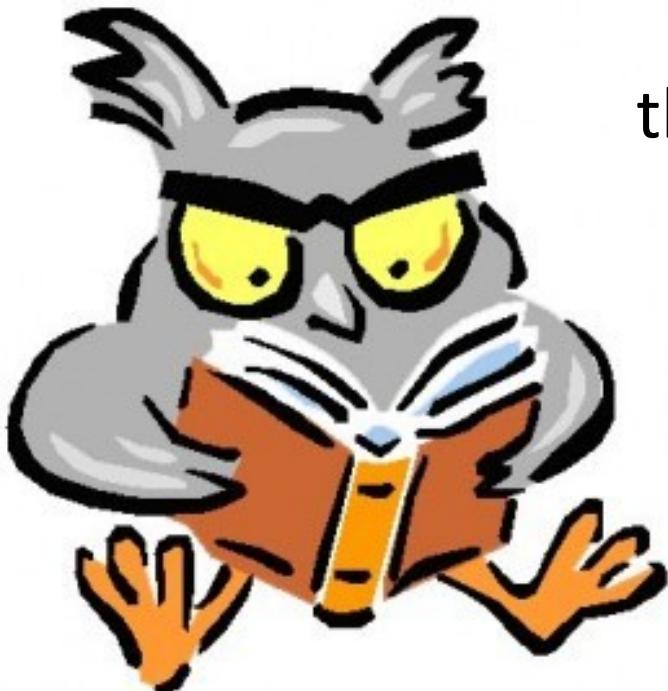
# Introducing the Reader monad ...

```
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

# The Reader Monad

The Reader monad makes  
a lazy computation  
*explicit*  
in the type system, while  
*hiding*  
the logic to apply it



In other words the reader monad  
allows us to treat **functions as  
values with a context**

We can act as if we already know  
what the functions will return.

# ... 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

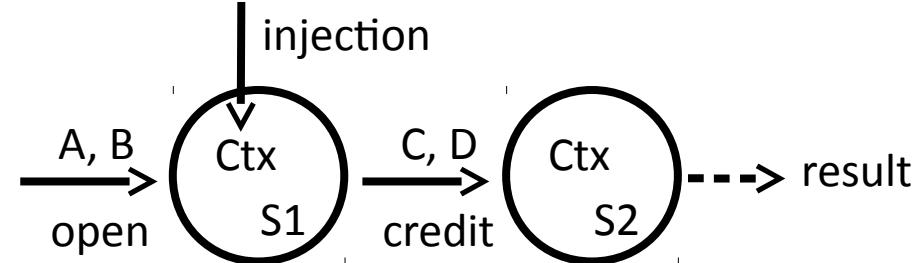
```
public class TryReader<R, A> {  
    private final Function<R, Try<A>> run;  
  
    public TryReader( Function<R, Try<A>> run ) {  
        this.run = run;  
    }  
  
    public <B> TryReader<R, B> map(Function<A, B> f) {  
        return new TryReader<R, B>((R r) -> apply( r )  
            .map( 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

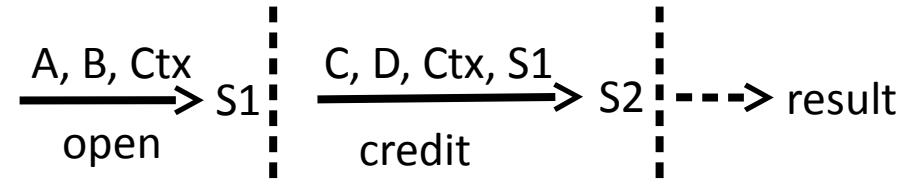
```
public class BankService {  
    public static TryReader<BankConnection, Account>  
        open(String owner, String number, BigDecimal balance) {  
            return new TryReader<>((BankConnection bankConnection) -> ...)  
    }  
  
    public static Reader<BankConnection, Account>  
      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

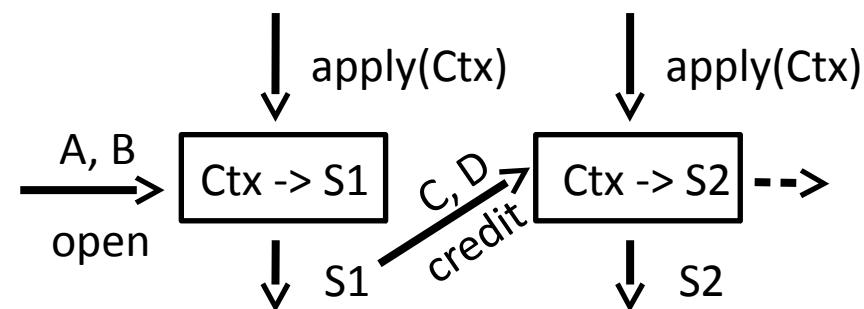
```
Try<Account> account = reader.apply(new BankConnection());
```



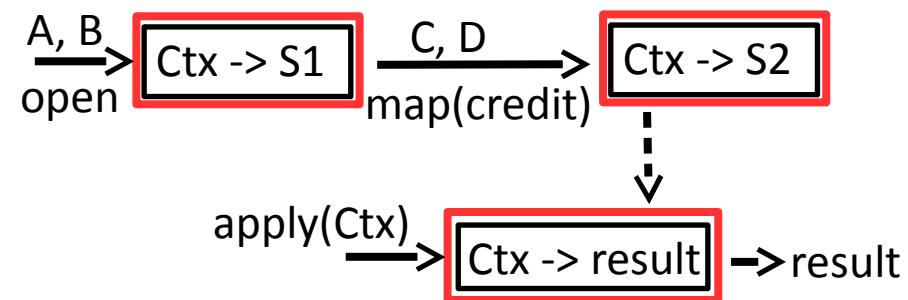
## Pure OOP implementation



## Static Methods



## Lazy evaluation



## Reader monad

# To recap: a Monad is a design pattern

## *Alias*

- flatMap that shit

## *Intent*

- Put a value in a computational context defining the policy on how to operate on it

## *Motivations*

- Reduce code duplication
- Improve maintainability
- Increase readability
- Remove side effects
- Hide complexity
- Encapsulate implementation details
- Allow composability

## *Known Uses*

- Optional, Stream, Promise, Validation, Transaction, State, ...

# TL;DR

**Use Monads whenever possible  
to keep your code clean and  
encapsulate repetitive logic**

Lambdas, streams, and functional-style programming



# Java 8 IN ACTION

Raoul-Gabriel Urma  
Mario Fusco  
Alan Mycroft

MANNING

# Thanks ... Questions?

Q



A

Mario Fusco  
Red Hat – Senior Software Engineer

[mario.fusco@gmail.com](mailto:mario.fusco@gmail.com)  
twitter: @mariofusco