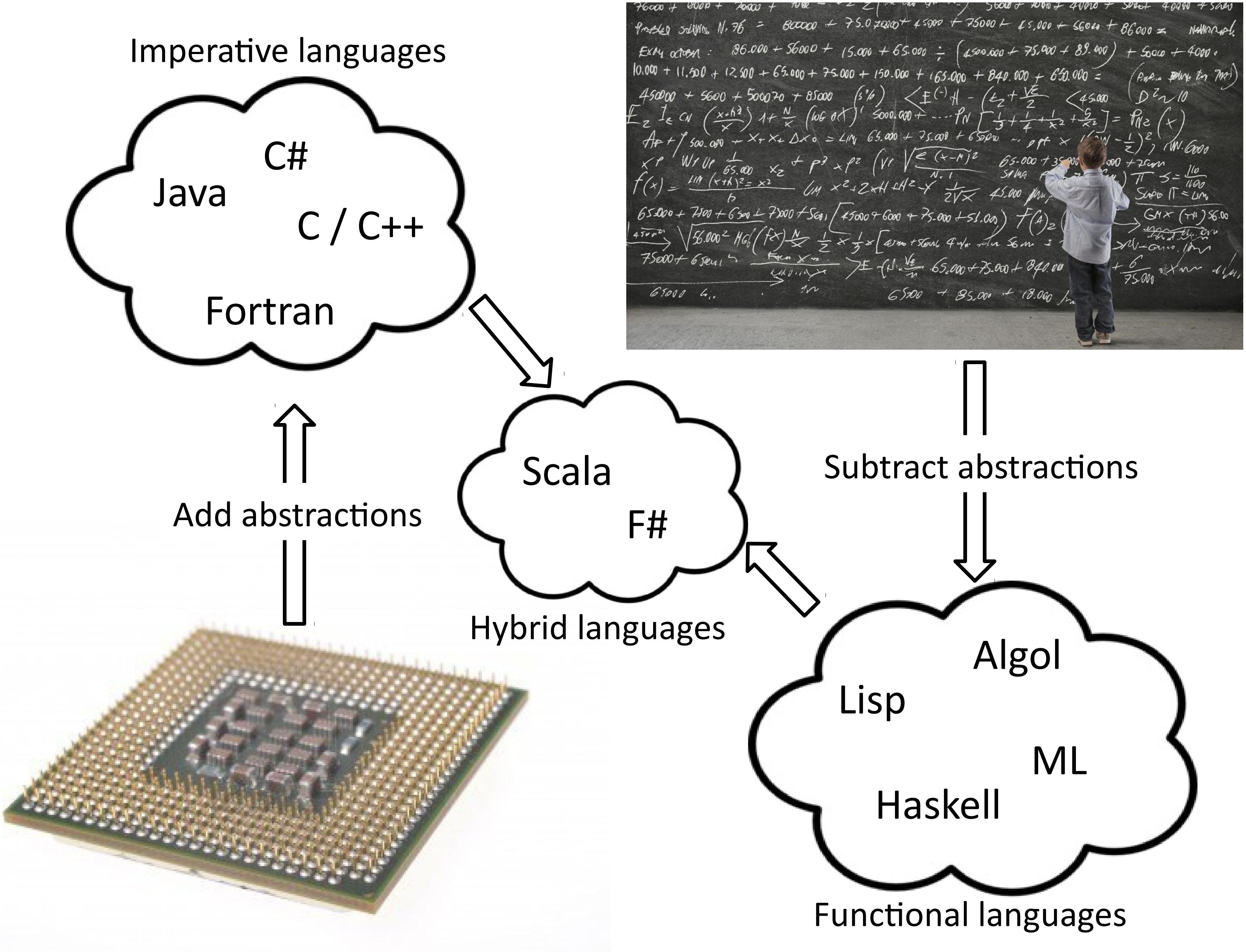
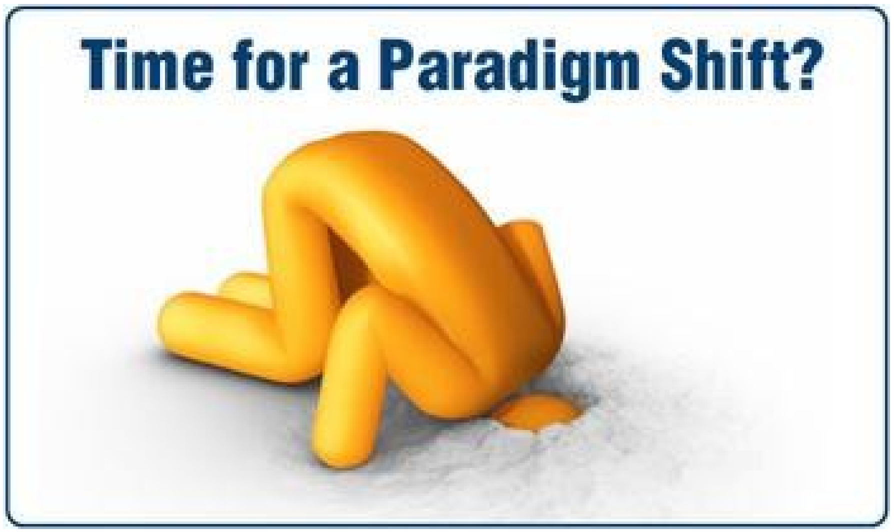
by Mario Fusco
mario.fusco@gmail.com
twitter: @mariofusco

Monadic Java


Imperative languages
C#
Java
C / C++
Fortran
Add abstractions
Scala
F#
Hybrid languages
Subtract abstractions
Algol
Lisp
ML
Haskell
Functional languages

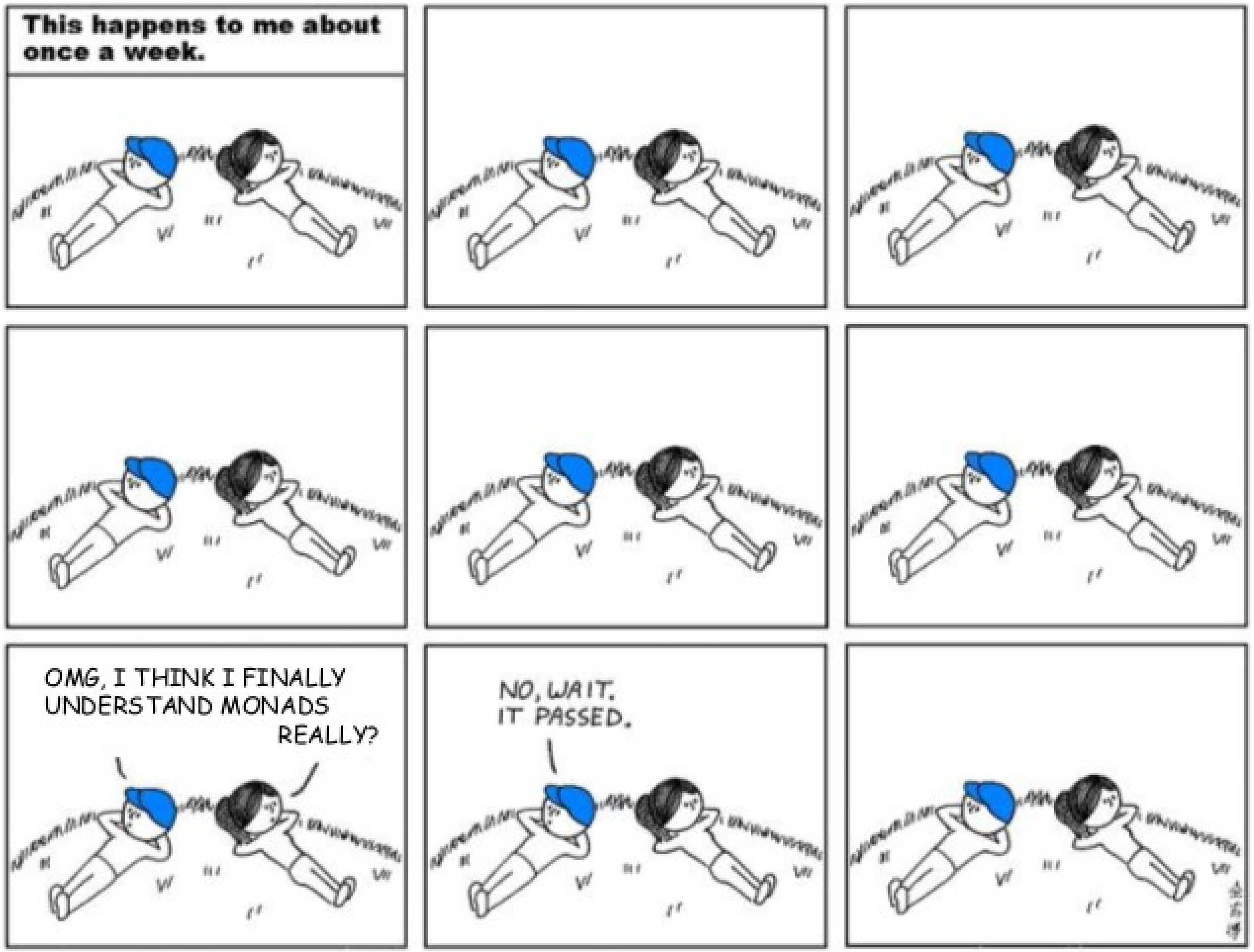
new language < new paradigm

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, η, μ)** where T is an endofunctor **T: X → X** and **η: I → T** and **μ: T x T → T** are 2 natural transformations satisfying these laws:

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

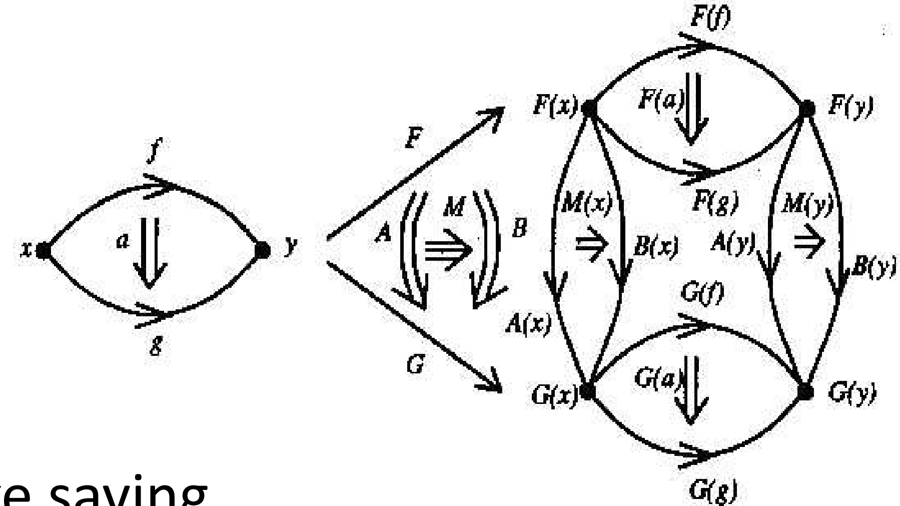
Associative law: $\mu(\mu(T \times T) \times T)) = \mu(T \times \mu(T \times T))$

In other words: "*a monad in X is just a monoid in the category of endofunctors of X, with product × 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 Cathegory Theory



… it's like saying …

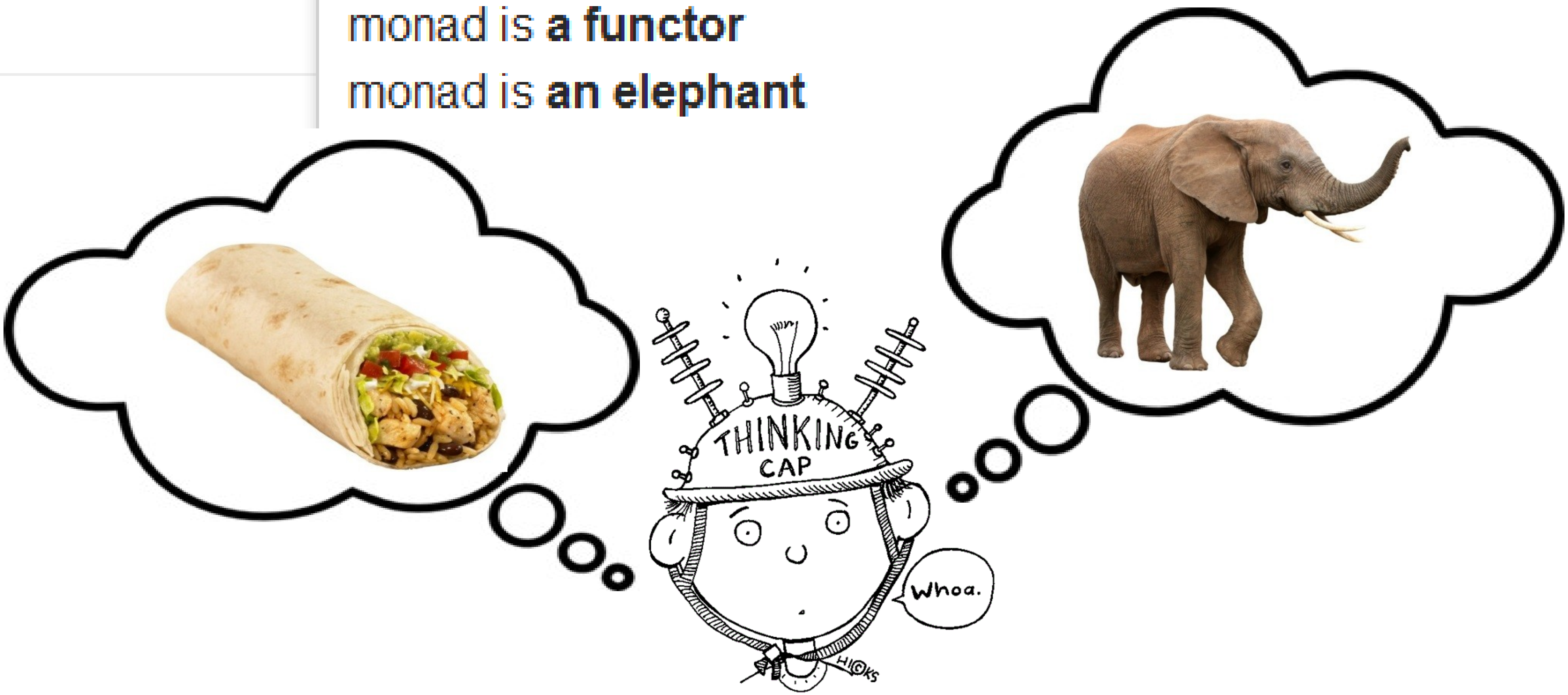
In order to understand pizza you need to first learn Italian

… ok, so let's try to ask Google …

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 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. Bigalow, 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; }
}
```

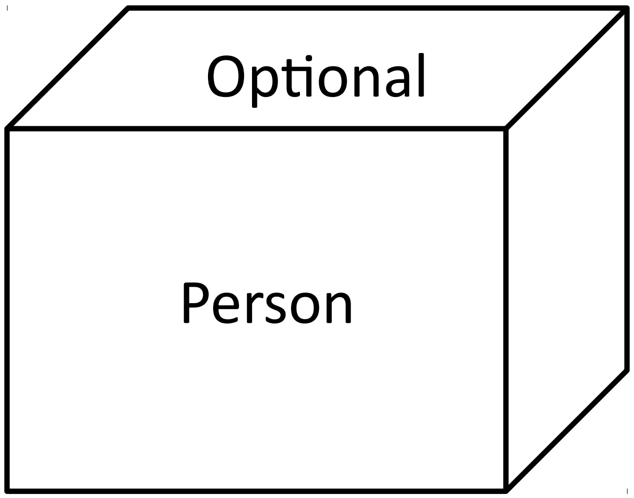
Using the type system
to model nullable value

Restoring the sanity

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

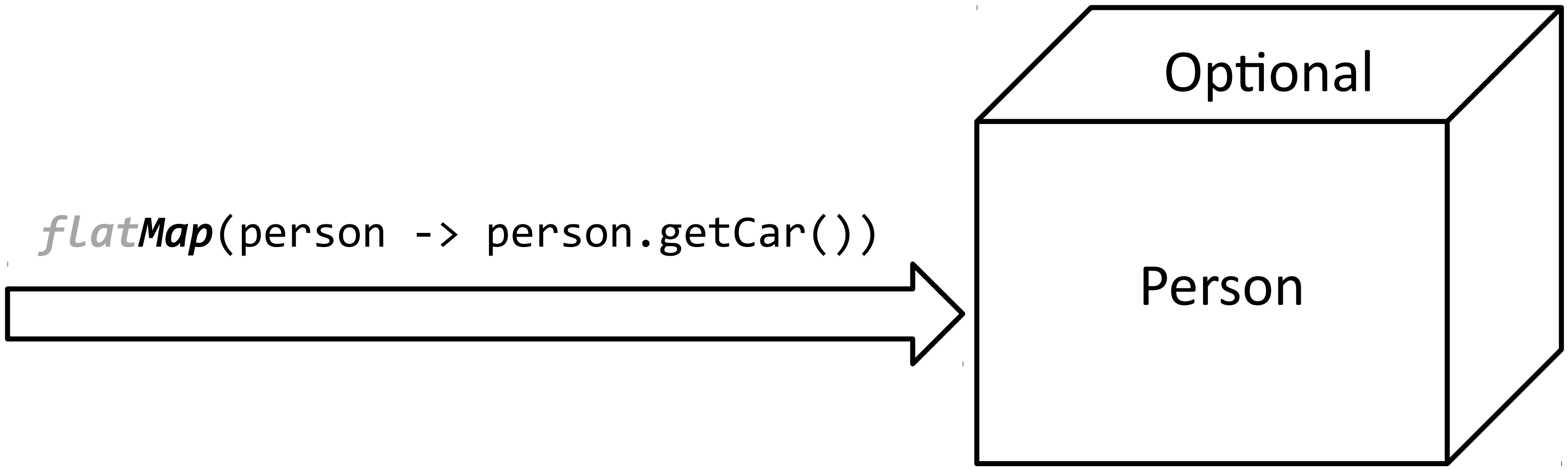
Restoring the sanity

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



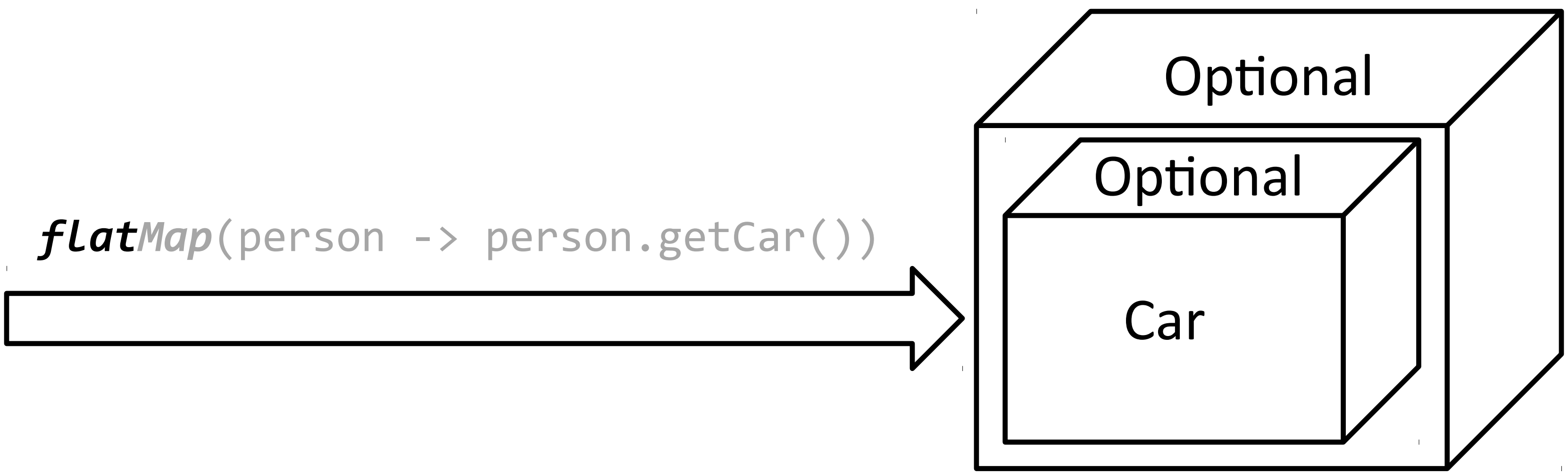
Restoring the sanity

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



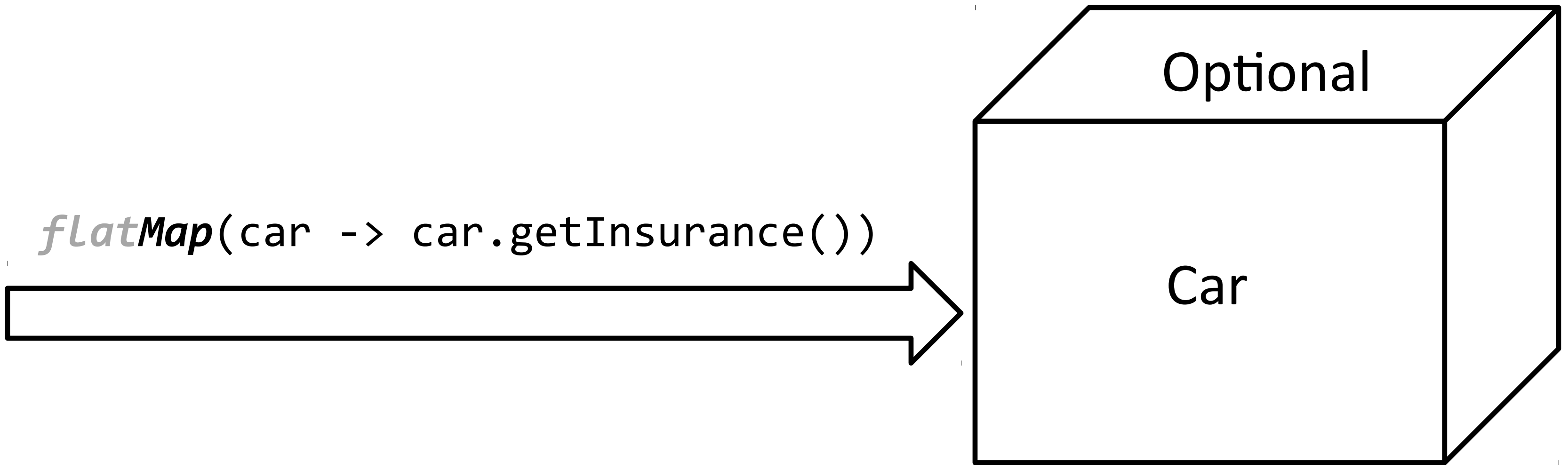
Restoring the sanity

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



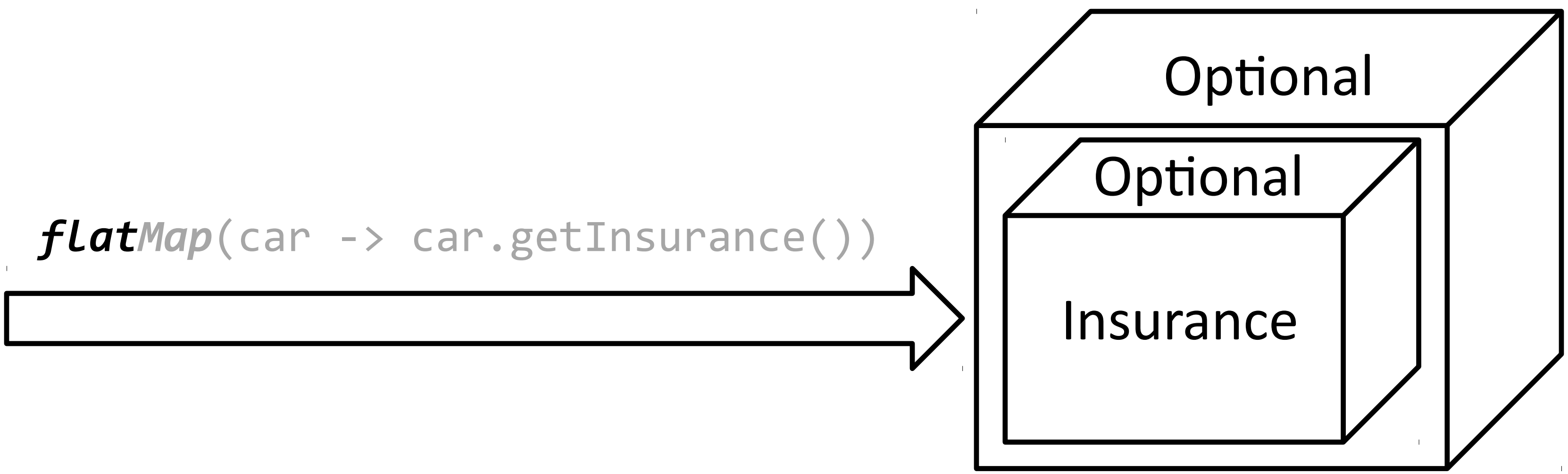
Restoring the sanity

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



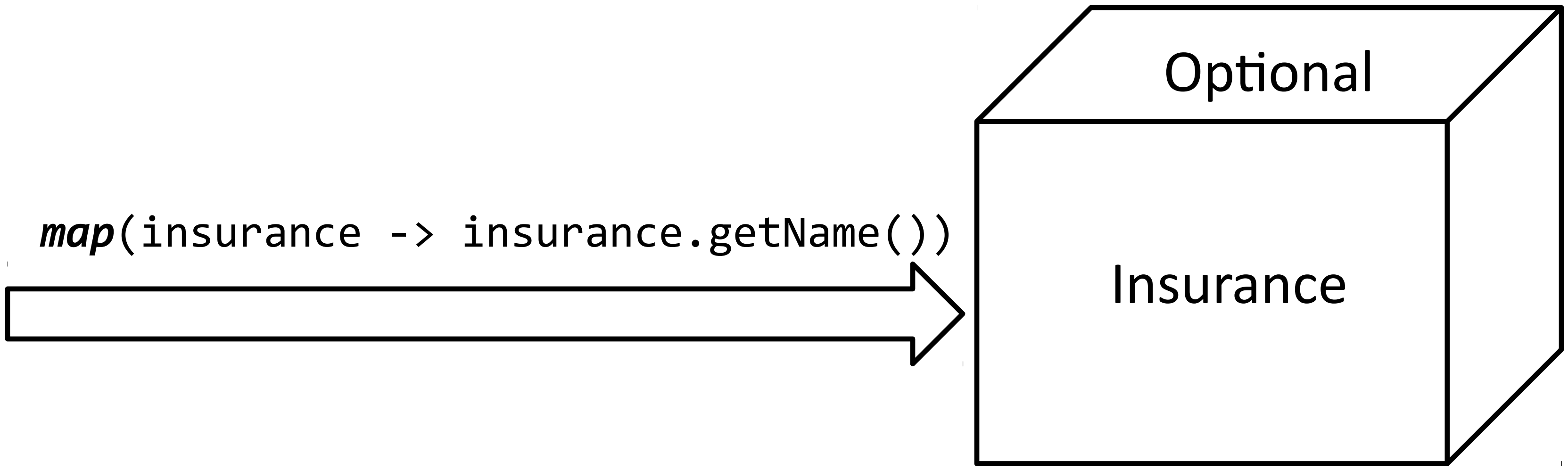
Restoring the sanity

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



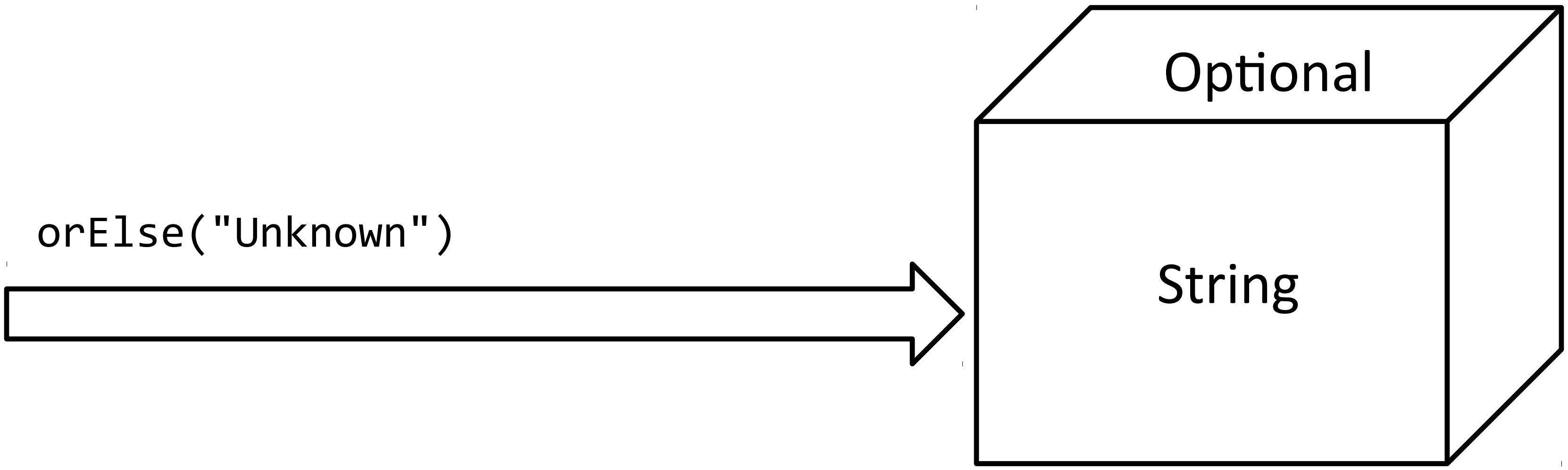
Restoring the sanity

```
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");
}
```



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
@EmrgencyKittens

Follow

cat in a box, in a box.
pic.twitter.com/5cHJN3NHvz

Reply Retweet Favorite More

RETWEETS 246 FAVORITES 487

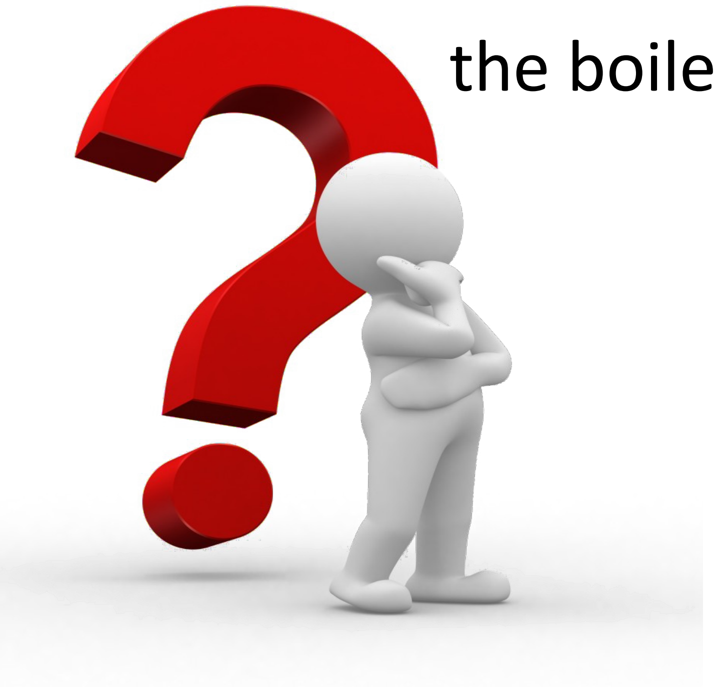
5:45 PM - 23 Mar 2014

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

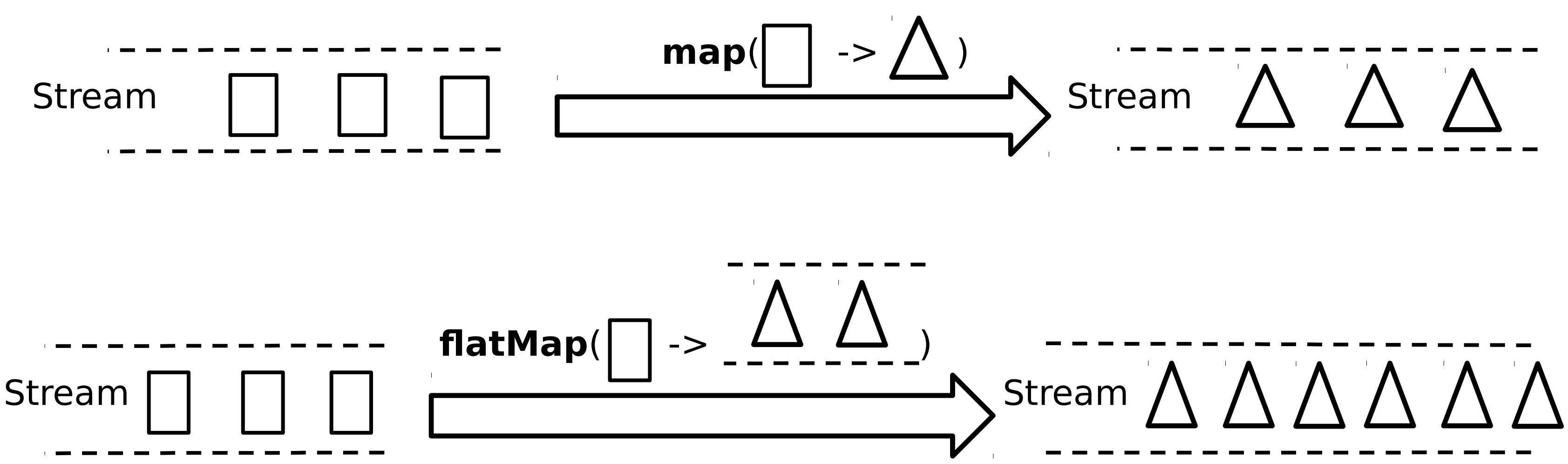
```
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

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);
```



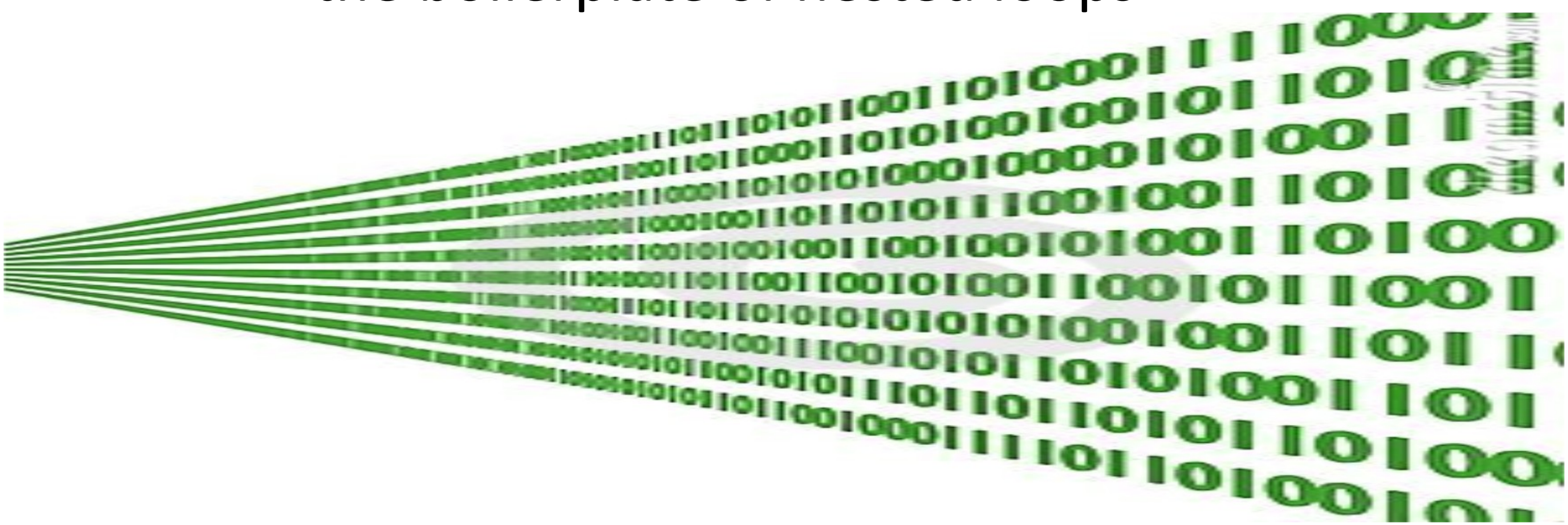
Given n>0 find all pairs i and j where 1 ≤ j ≤ i ≤ n and i+j is prime

```
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)
```

map

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)
```

flatMap

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

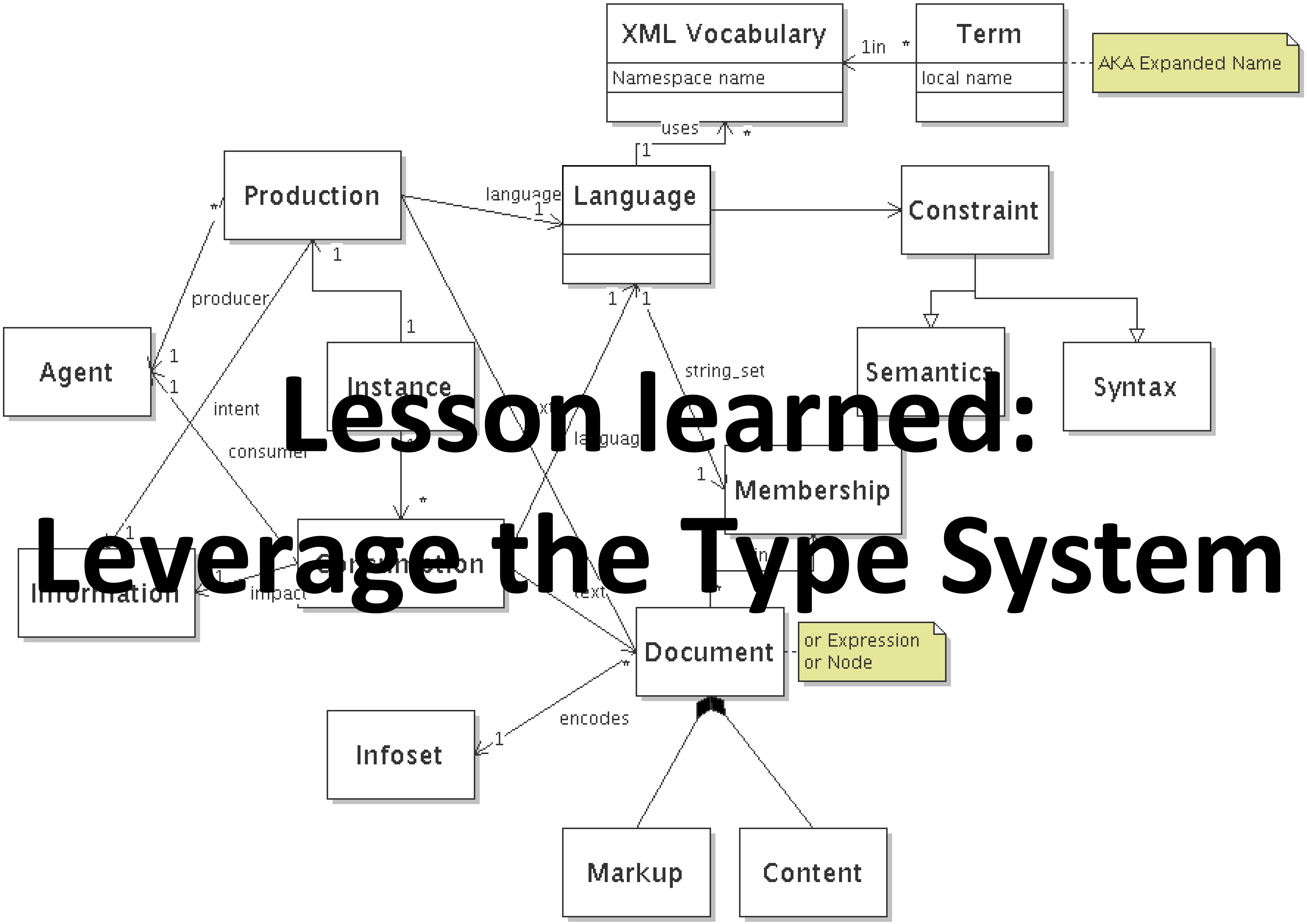
The Validation monad makes
the possibility of errors
explicit
in the type system, while
hiding
the boilerplate of "try/catch" logic

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);
}
```

Lesson learned: Leverage the Type System



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));
    }
}
```

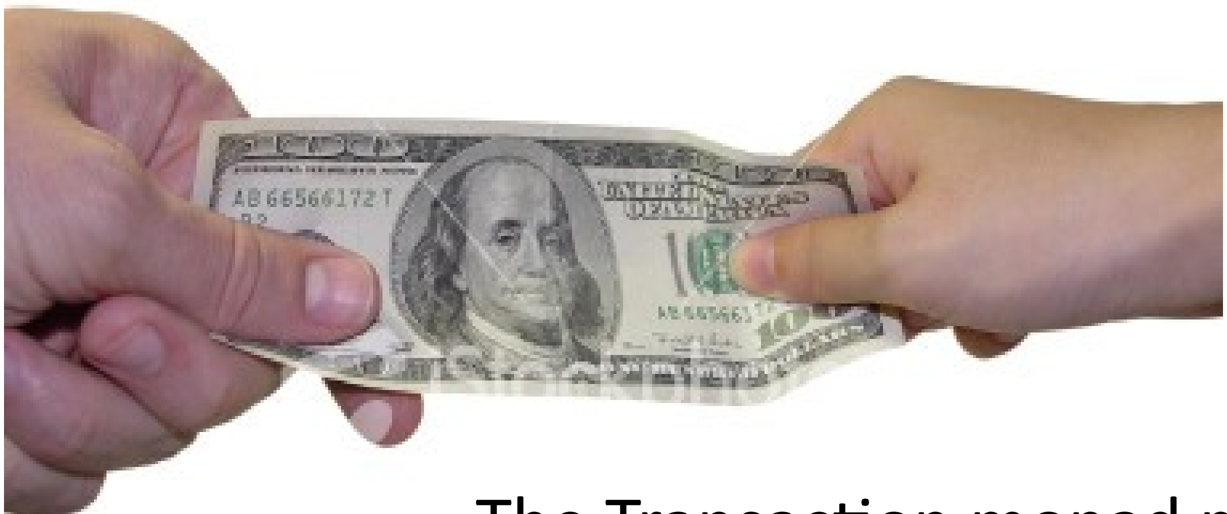
Gathering multiple errors - Failure

```
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

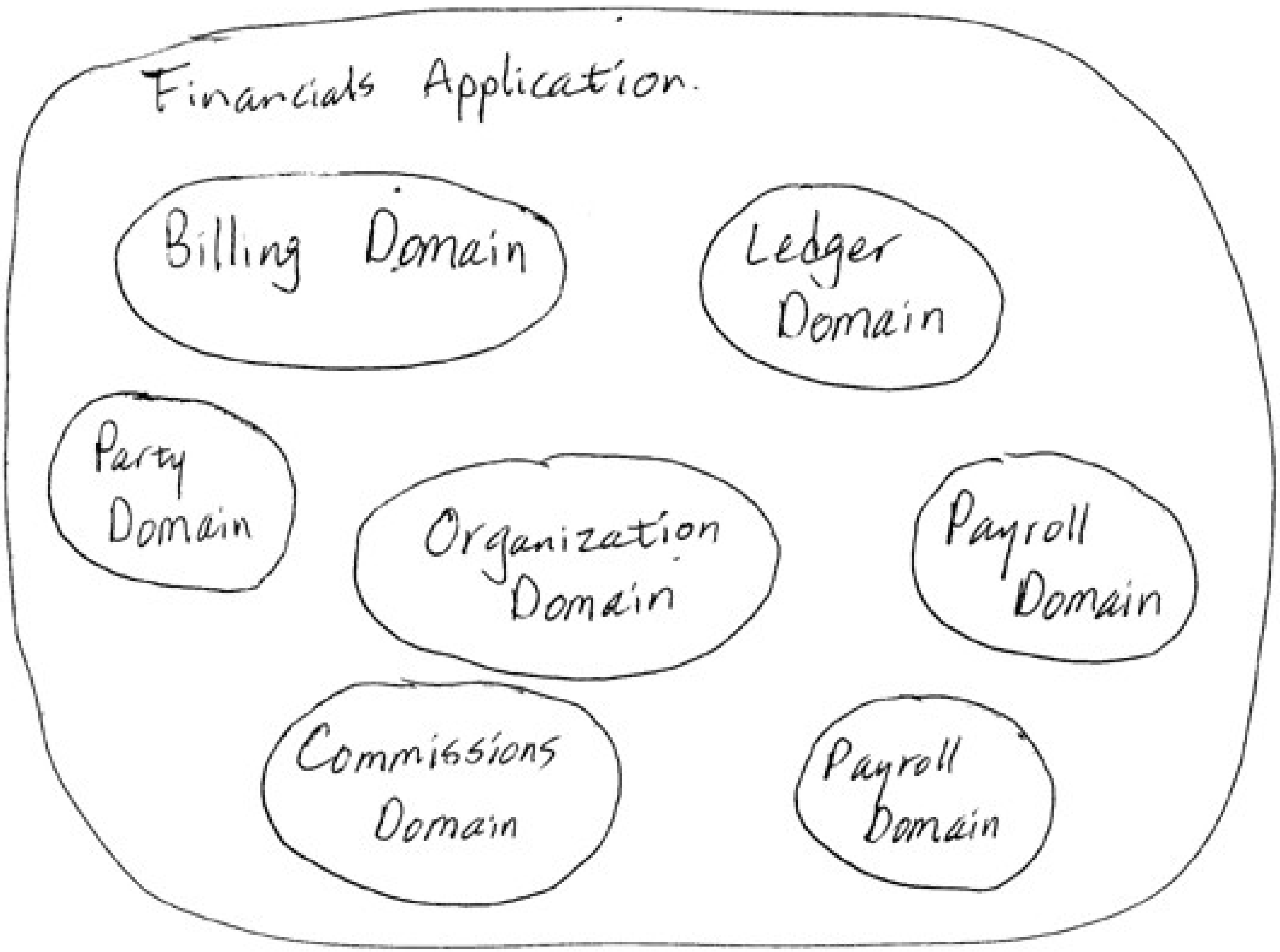
Monads are parametric types with two operations flatMap and unit that obey some algebraic laws

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 ...

```java
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

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

```java
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

```java
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<>(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;

    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 ) ) ) );
    }
}
```

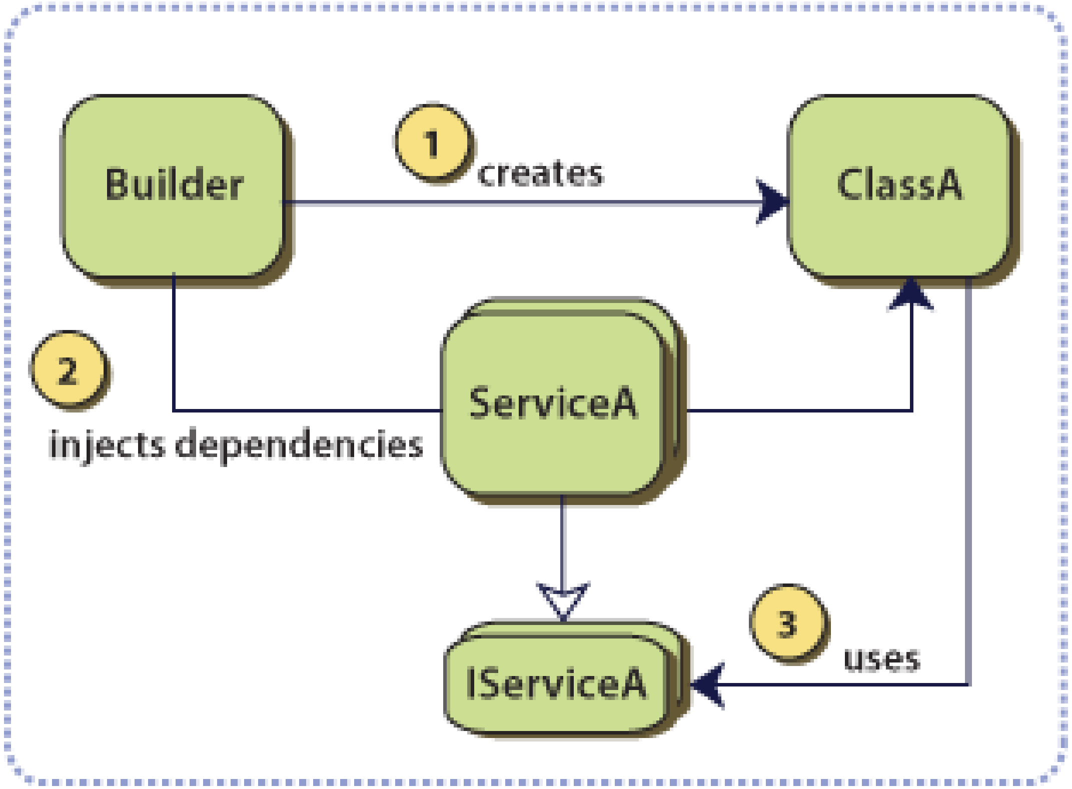
Decoupling state and behavior

```
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) {
        ...
    }
}
```

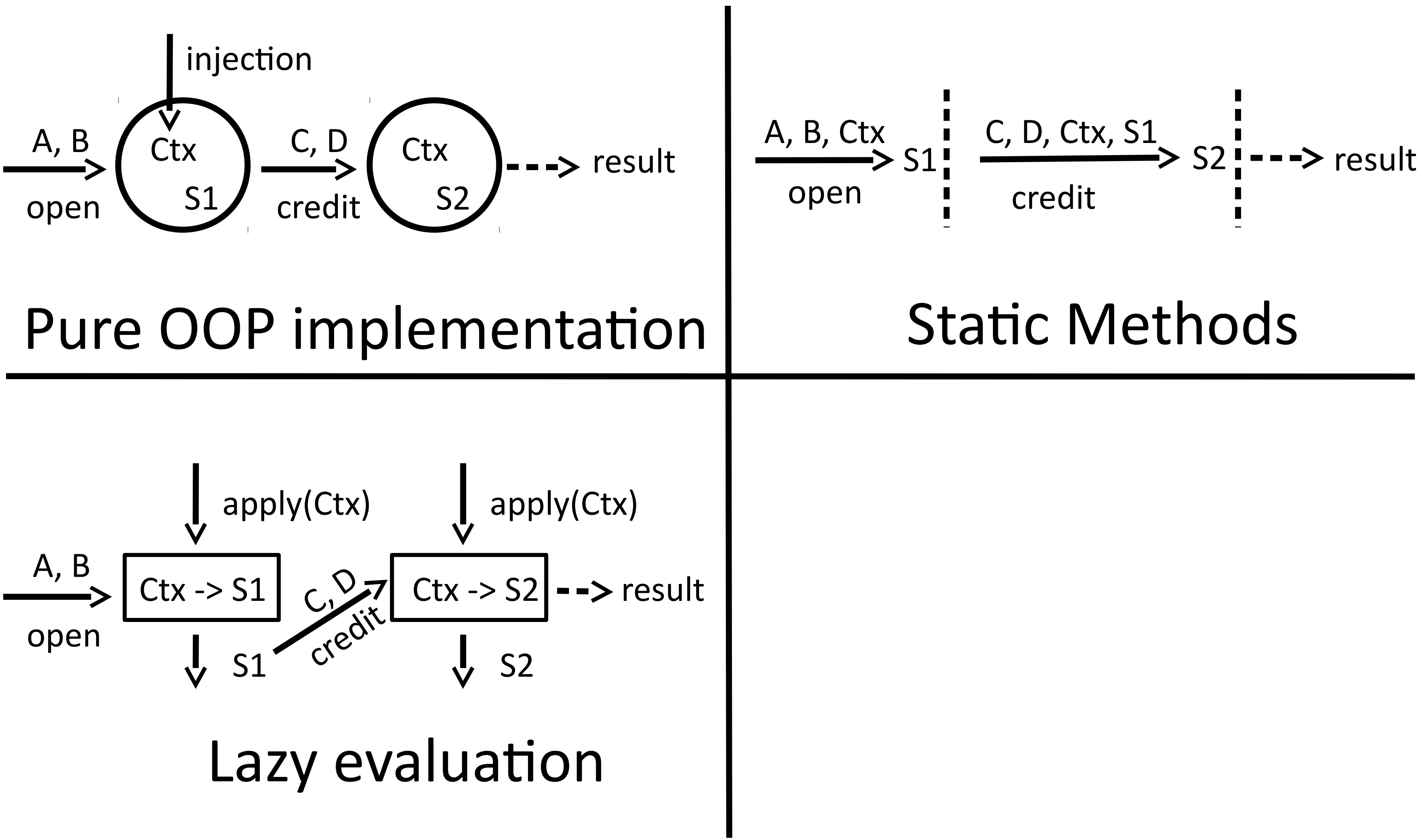
Necessary to create the BankConnection in advance ...

... 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() );
```



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
result
S1
S2
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

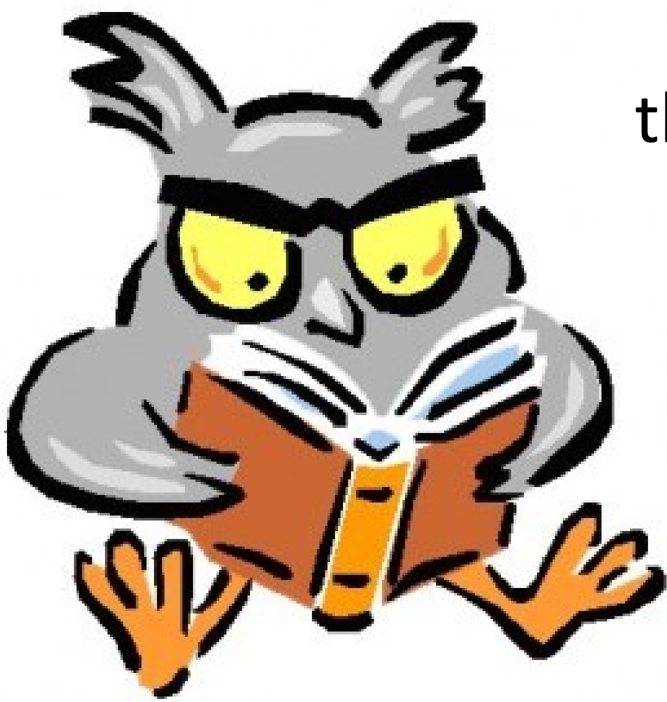
Introducing the Reader monad ...

```
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

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

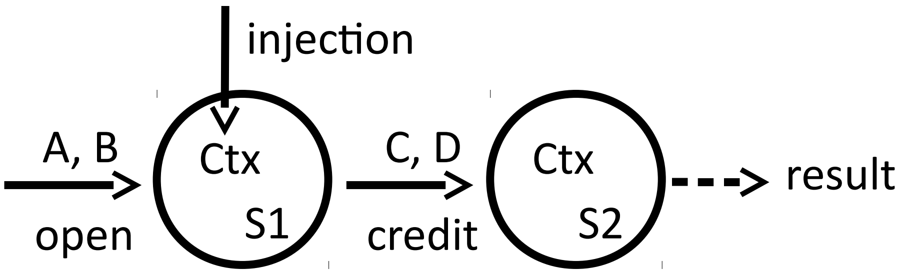
```
public class TryReader<R, A> {
    private final Function<R, Try<A>> run;

    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

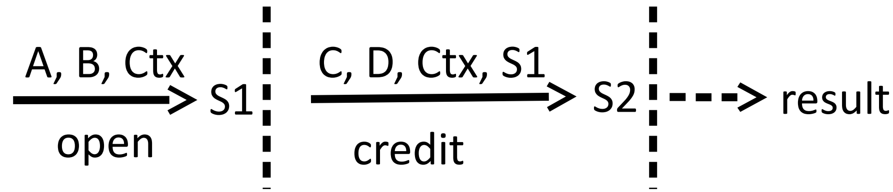
```
public class TryReader<R, A> {
    private final Function<R, Try<A>> run;

    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

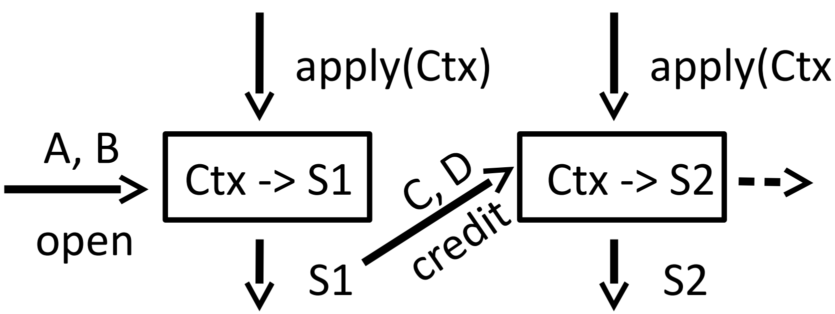
```
public class BankService {
    public static TryReader<BankConnection, Account>
                    open(String owner, String number, BigDecimal balance) {
        return new TryReader<>( (BankConnection bankConnection) -> ... )
    }

    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() );
```



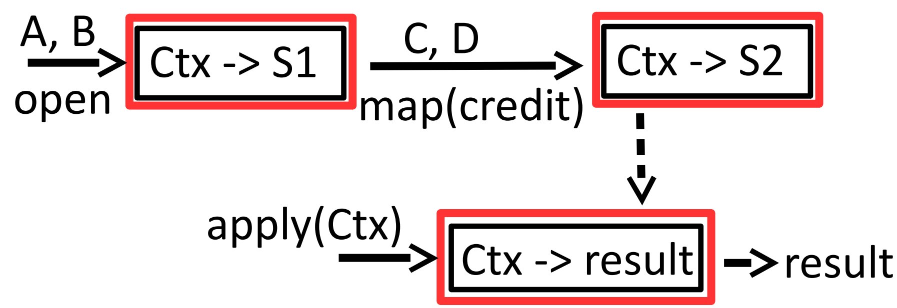
Pure OOP implementation



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
- Encapusalate 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
twitter: @mariofusco