From Object-Disoriented programming to Dysfunctional programming

or: How I Learned to Stop Worrying and Love the Lambda!

Warmup Lap: Java 8 Functions

New representation of methods, contained in java.util.function;

old way

new way

```
static Integer add3(Integer x) {
    return x + 3;
}
```

```
Function<Integer, Integer> f_add3 =
    x -> x + 3;
```

```
static Integer mul(Integer x, Integer y) {
    return x * y;
}
```



```
BiFunction<Integer, Integer, Integer> f_mul =
    (x,y) -> x * y;
```

Warmup Lap: Java 8 Functions

What's the difference?

old way

```
static Integer add3(Integer x) {
    return x + 3;
}
```

new way

```
Function<Integer, Integer> f_add3 =
    x -> x + 3;
```

```
static Integer mul(Integer x, Integer y) {
    return x * y;
}
```

```
BiFunction<Integer, Integer, Integer> f_mul =
    (x,y) -> x * y;
```

executes when “touched”

can be passed around
and executed later

Warmup Lap: Java 8 Functions

Why does this matter?
(the **short** version)

old way

```
static Integer mul(Integer x, Integer y) {
    return x * y;
}
```

new way

```
BiFunction<Integer, Integer, Integer> f_mul =
    (x,y) -> x * y;
```

I want to time how long it takes!

time (mul (4, 7));

* Evaluate 4 * 7
* Start the clock
* Stop the clock

Useless - **we can't do it this way**!

time (f_mul, 4, 7);

* Start the clock
* Evaluate 4 * 7
* Stop the clock

* Profit!

Warmup Lap: Java 8 Functions

Why *else* does this matter? (the **long** version)

It was a simpler time...

- Remember textbook examples?
- Small elegant snippets of code from a blog?

A modest example:
Updating counts in a HashMap

http://stackoverflow.com/questions/4157972/how-to-update-a-value-given-a-key-in-a-java-hashmap

It was a simpler time...

Then the real world happened:

- Exceptions + Try / Catch / Finally
- Logging
- Caching
- Concurrency (control-flow related)
- Parallelism (performance related)

Code gets ugly, quickly

The question is:

Can we bring our code back to looking clean and simple,
without compromising, in any way, all those real-world issues?

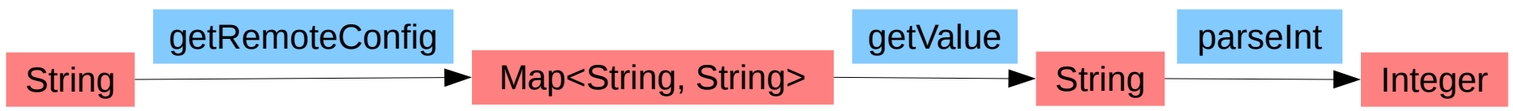
Attempt #1

Chaining our functions together:

```
Integer result =
      getRemoteConfig(“127.0.0.1:4000”)
            .getValue(“meaningOfLife”)
            .parseInt();
```

This is what we'd be able to do if “everything just worked”.

It's composable



Attempt #1

Chaining our functions together:

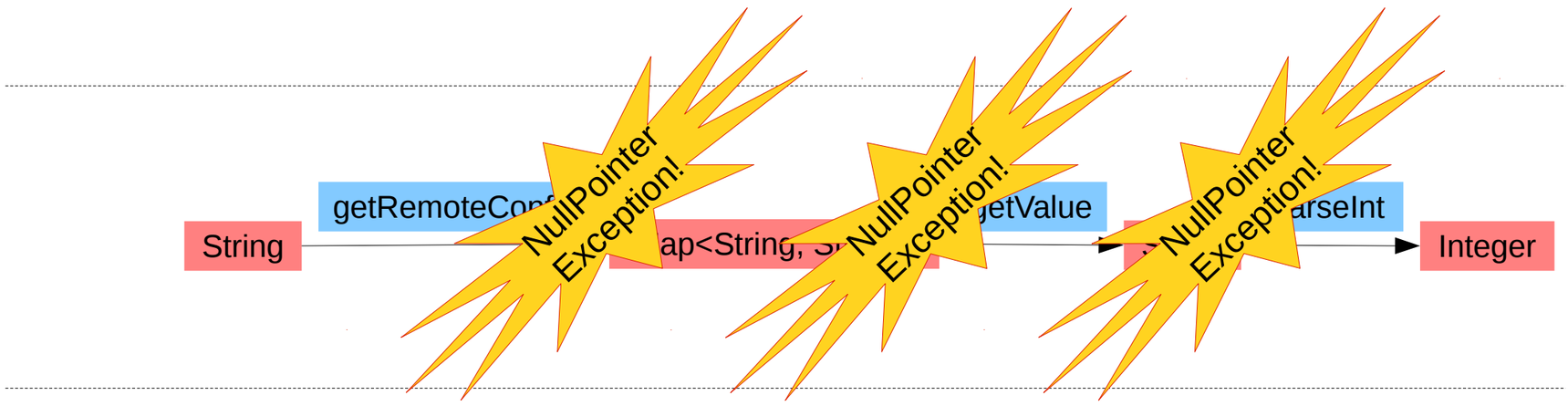
```
Integer result =
      getRemoteConfig(“127.0.0.1:4000”)
            .getValue(“meaningOfLife”)
            .parseInt();
```

This is what we'd be able to do if “everything just worked”.

But this is the real world, and it doesn't just work!

It's composable

It explodes !



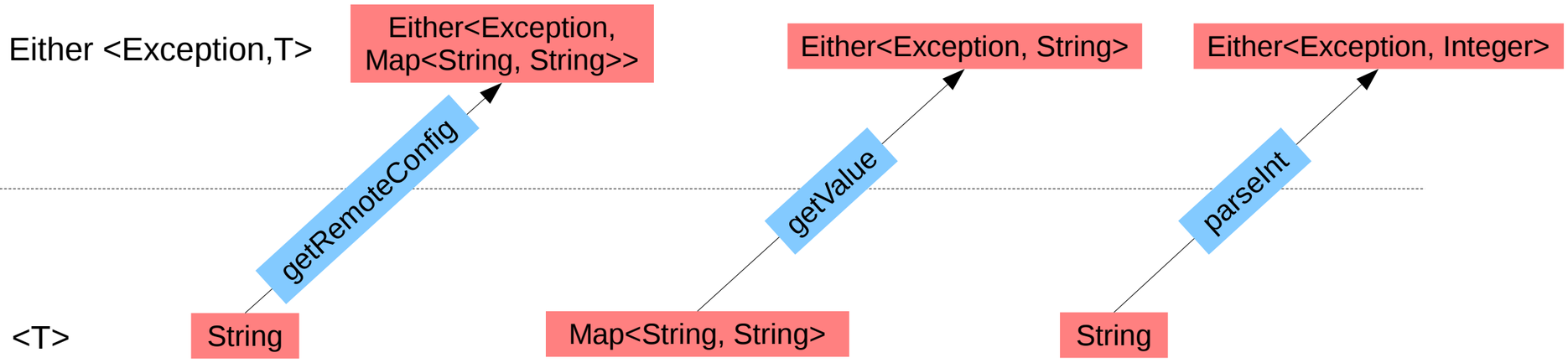
Attempt #2

Let's be **explicit about failure** so that the caller can't accidentally detonate the null-bomb!

I'll use a wrapper type called Either<Exception, T> to do so.

All functions that might not work “should return Either”

It doesn't explode!



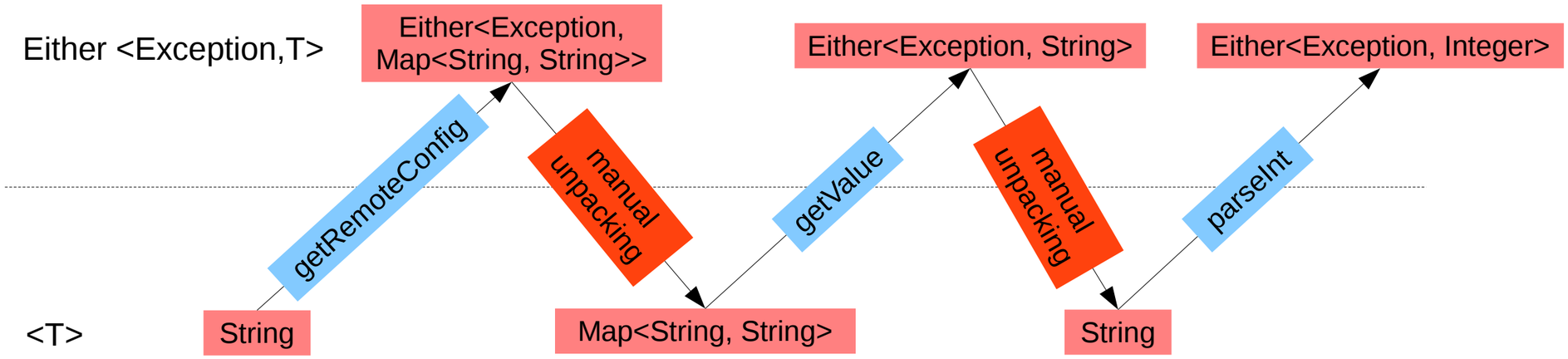
Attempt #2

Let's be **explicit about failure** so that the caller can't accidentally detonate the null-bomb!

I'll use the wrapper type Either<Exception, T> to do so.

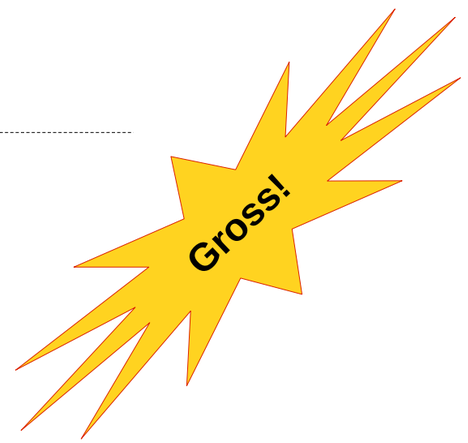
It doesn't explode!

It doesn't compose!




```
if (eitherRemoteConfig.isValid()) {
      remoteConfig = eitherRemoteConfig.get();
} else {
      //What now?
}
```

```
if (eitherValue.isValid()) {
      value = eitherValue.get();
} else {
      //What now?
}
```



Attempt #2 Either<Exception, T>

What have we actually gained? Nothing!

It's actually GROSSER than the two existing ways of doing this (null check or try/catch)

It doesn't explode!

It doesn't compose!

```
if (eitherRemoteConfig.isValid()) {

      remoteConfig =
            eitherRemoteConfig.get();

      remoteConfig.doSomething();

} else {
      // What now?
}
```

GROSS! (and uncomposable)

```
if (remoteConfig != null) {



      remoteConfig.doSomething();

} else {
      // What now?
}
```

Uncomposable

```
try {



      remoteConfig.doSomething();

} catch (Exception e) {
      // What now?
}
```

Uncomposable

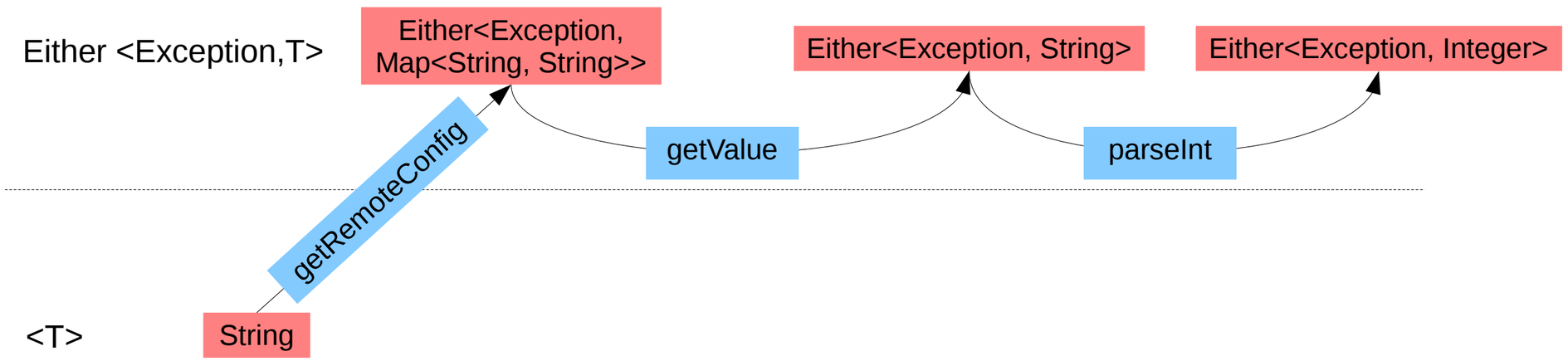
Attempt #3

Let's pursue the Either approach, but try to restore composability!

Can we make those blue functions at the bottom work accept Eithers instead?

It doesn't explode!

It composes! Sort of...



But now getValue(...) and parseInt(...) have bad types!

getValue can only pull from an Either Map, and parseInt can only parse an Either Int.

This is not good. What if we switch the order of the calls? Which functions act on raw values, and ones act on Eithers?

Completely unreusable!

Attempt #4

Let's get back to using functions from A to Either <B>, because Either <A> to Either <B> wasn't working out so well. How do we join these things together without having to hard-code stuff?

Let's reach into the functional toolbag... and pull out ***map !***

map lets us write functions, at the <T> level, but then to "***lift them up***" to whatever level we need!

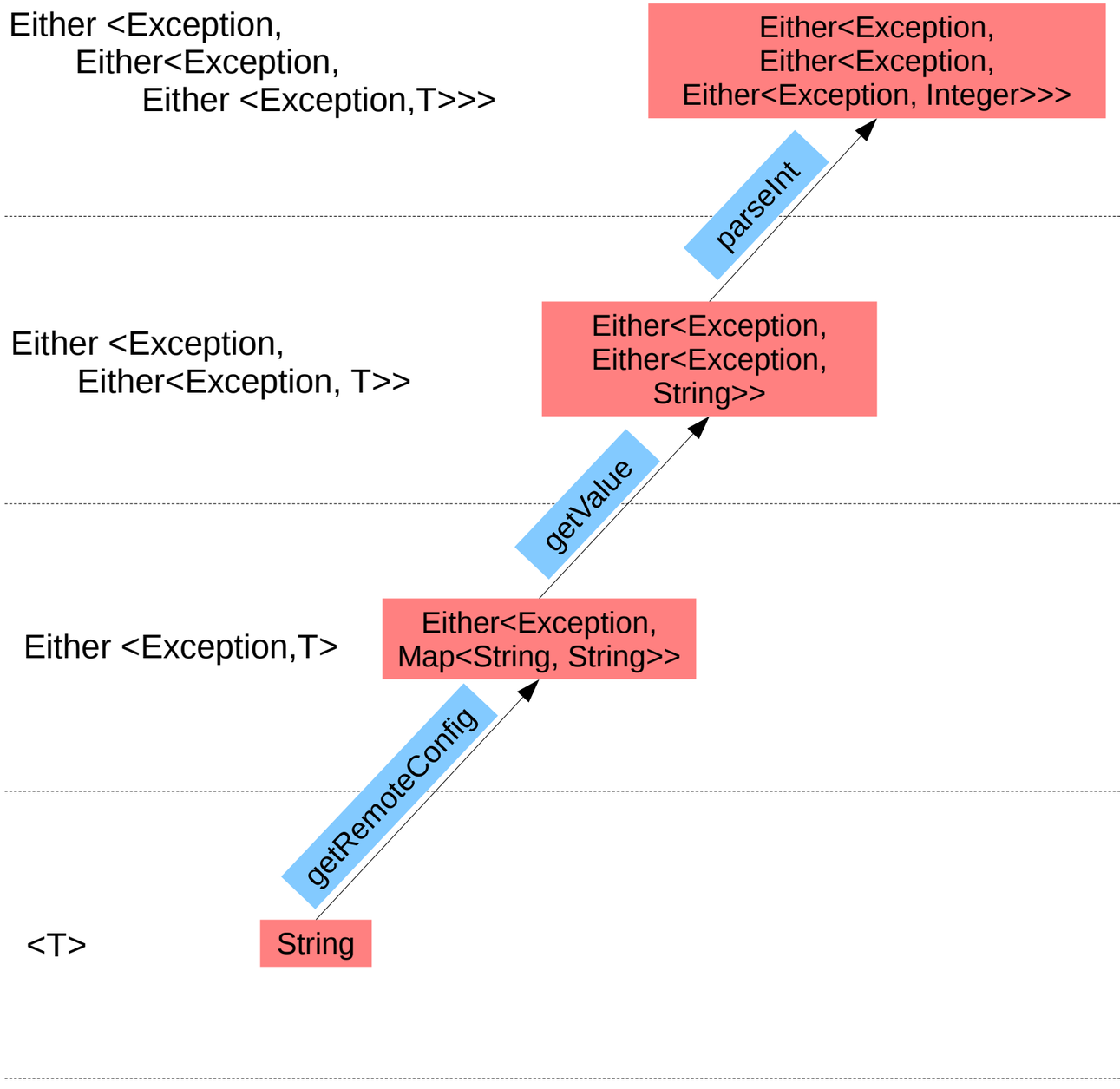
What this looks like in code:

```
getRemoteConfig("http://127.0.0.1:4000")
      .map (conf -> getValue (conf, “meaningOfLife”))
      .map (str -> parseInt (str));
```

... Doesn't look too bad, until...

It doesn't explode!

Composes... badly



Attempt #4

It doesn't explode!

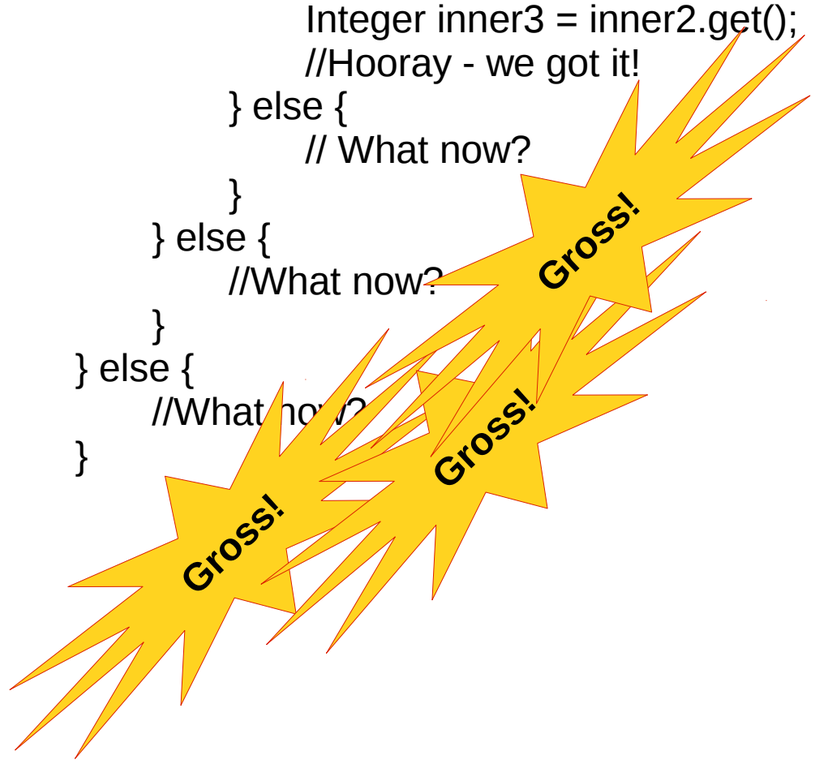
Composes... badly

What this looks like in code:

```
getRemoteConfig("http://127.0.0.1:4000")
        .map (conf -> getValue (conf, “meaningOfLife”))
        .map (str -> parseInt (str));
```

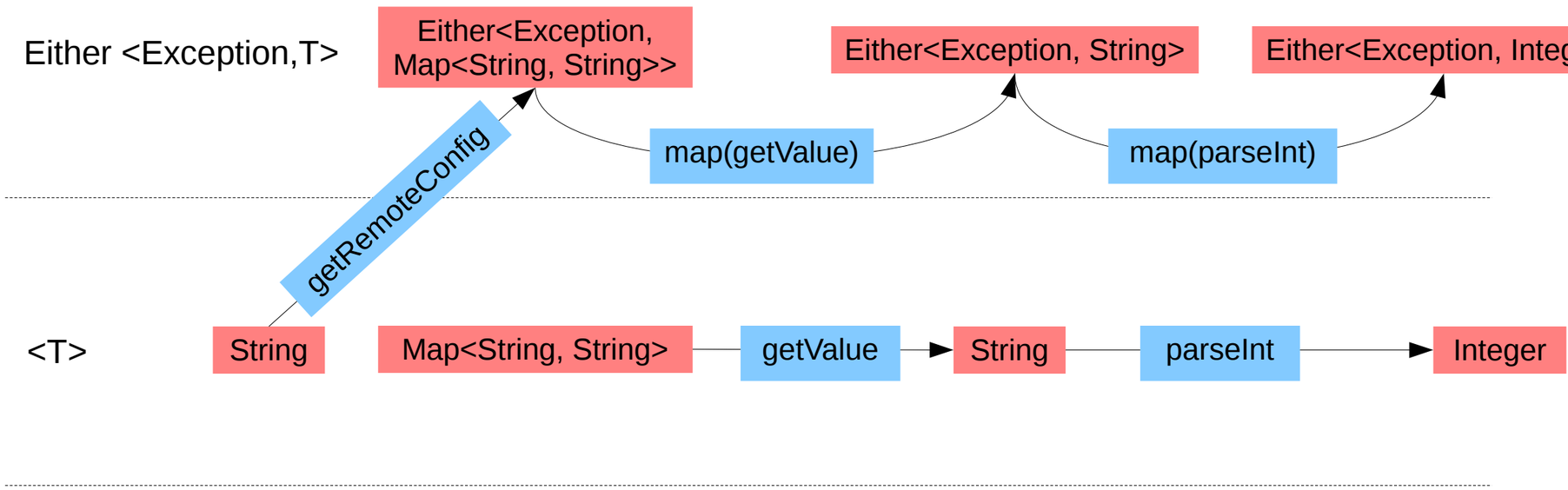
... Doesn't look too bad, until...

```
// Try to inspect our result

Either <Exception, Either<Exception, Either <Exception,Integer>>> result;

if (result.isValid()) {

        Either<Exception, Either <Exception,Integer>> inner1 = result.get();

        if (inner1.isValid()) {

                Either <Exception,Integer>> inner2 = inner1.get();

                if(inner2.isValid()) {
                        Integer inner3 = inner2.get();
                        //Hooray - we got it!
                } else {
                        // What now?
                }
        } else {
                //What now?
        }
} else {
        //What now?
}
```



Attempt #5

Let's keep trying with ***map.*** So that we don't "climb too high", let's rewrite our *getValue(...)* and *parseInt(...)* as values over simple functions (they don't return Either anymore)
Now we can lift them up with map .



We write getValue as *Map->String* and map lifts it up to
Either<Exception, Map> -> Either <Exception, String>

Likewise, we write parseInt as *String->Integer* and map lifts it up to
Either<Exception, String> -> Either <Exception, Integer>

We're getting closer!

It doesn't explode!

It composes! Sort of...

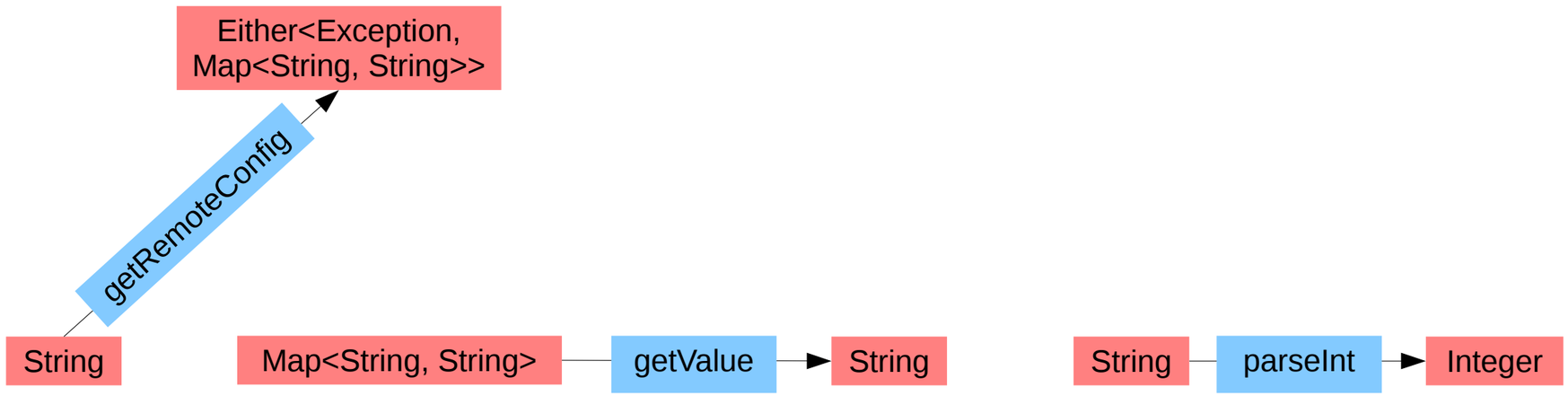
More reusable but not perfect

Attempt #5

Let's look at what we ended up with...

So we got by with writing the three functions, and didn't need any manual unpacking/repacking.

However, there is something visibly wrong with this!



It doesn't explode!

It composes! Sort of...

More reusable but not perfect

* It's no longer **consistent** - getRemoteConfig is somehow different to the others!

* *getValue(...)* and *parseInt(...)* are now **dishonest** - they advertise that they always work, but they must be able to fail just like *getRemoteConfig(...)*.

* This only works if we can assume getRemoteConfig goes first, it won't work for other cases.

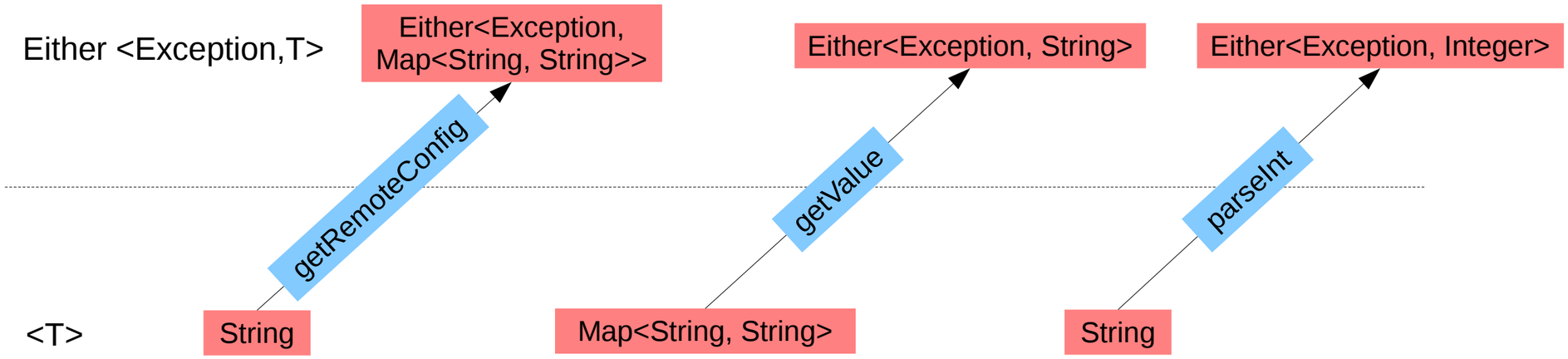
Attempt #6

Going back to the consistent version. We have three functions that all honestly advertise possible failure - so they're honest.

But we still to improve composition.

It doesn't explode!

Uncomposable

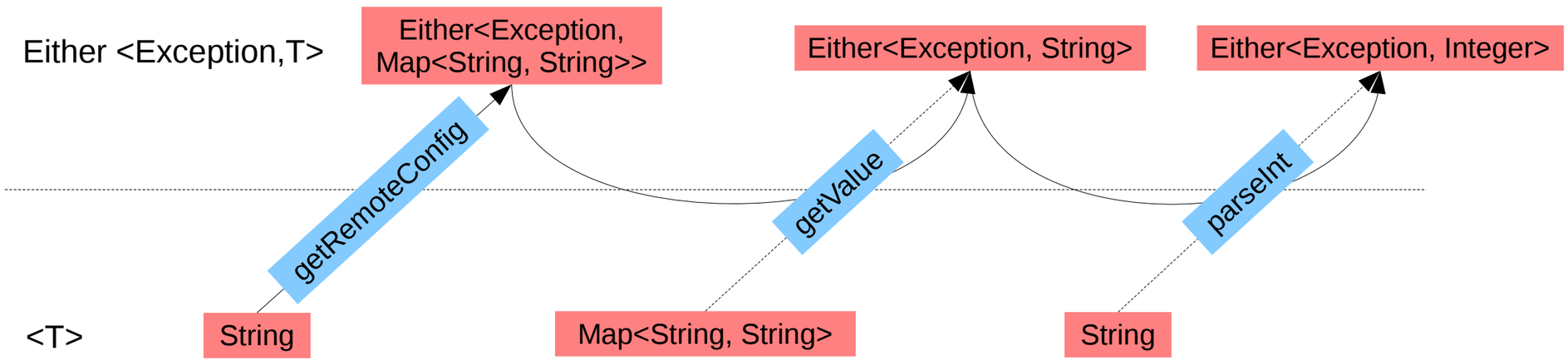


Attempt #6 - flatMap

Where *map* lifts a function from the bottom row (<T>) up to the next row, *flatMap* instead just lifts the “left side” of the function up - or viewed another way, it lowers the previous value down, then applies the next function to lift it back up.
So we can write functions that declare that they fail in a consistent way, and *flatMap* pieces them together.

It doesn't explode!

Composable



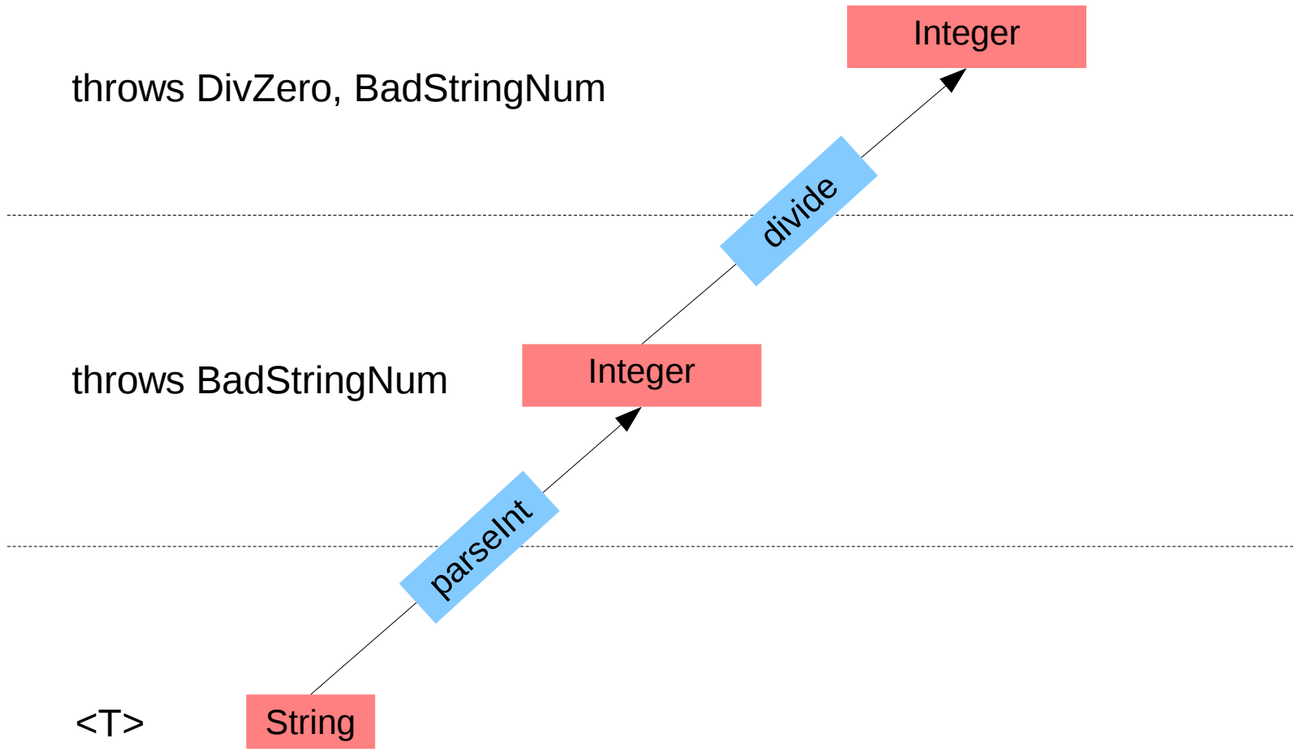
What we've written here is:

```
Either<Exception, Integer> result =
      getRemoteConfig(“127.0.0.1:4000”)
            .flatMap(conf -> getValue(conf, “meaningOfLife”))
            .flatMap(str -> parseInt(str));
```

Which is as similar as you can get to our naive code at the beginning, but safe!

```
Integer result =
      getRemoteConfig(“127.0.0.1:4000”)
            .getValue(“meaningOfLife”)
            .parseInt();
```

Why not regular exceptions?




```
Integer value = null;
try {
        value = map.get(“key”);
catch (NullPointerException npe) {
       ??
}
```

Fighting the language

How do you clean up after your objects:
Finalisers?

How do you serialize your objects?
Using in-built Java serialization?

Inheritance?
Try not to use this! Favour has-relationships over is-relationships

Mutability
Look at String. The Java way is clearly to favour immutability!
Look at byte[]. The Java way is clearly to favour mutability!

Construct an Object?
Don't use constructors, use factories instead!

Objects?
Nah, Beans are better!

Don't use other people's code because it's non-standard
Don't roll your own code because other people have already written it

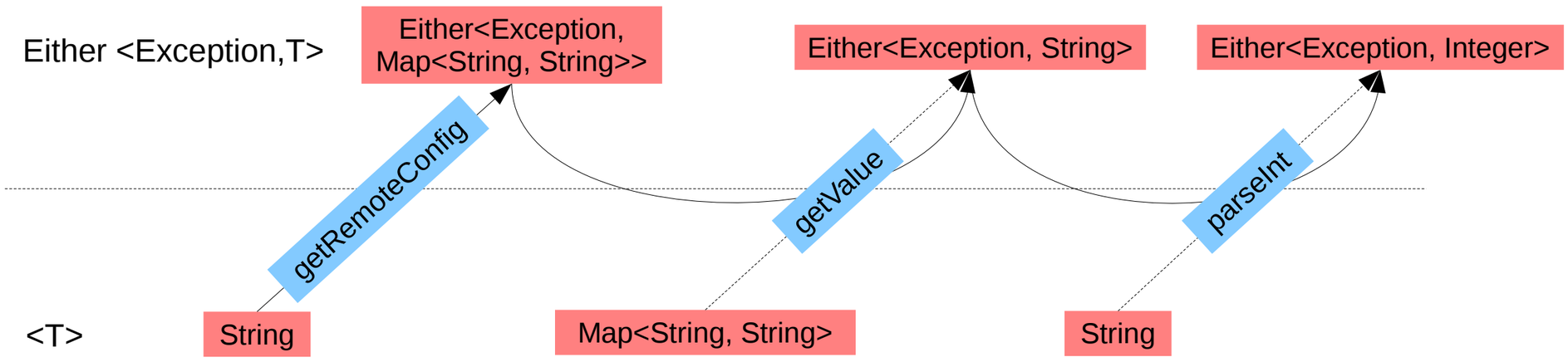
Fail fast!
Don't keep trying to do something when you know something's wrong, just bail out!
Also, make your code fault-tolerant, so your system doesn't crash when something's wrong!

But what does “composable” actually mean?

It means bigger pieces are the same kind of thing as smaller pieces!

When you add two numbers, you get a third number.
Num 4 * Num 3 = *Num* 12
as opposed to
Num 4 * Num 3 = *Product* 12

Num 4 + Num 7 = *Num* 11
as opposed to
Num 4 + Num 7 = *Sum* 11



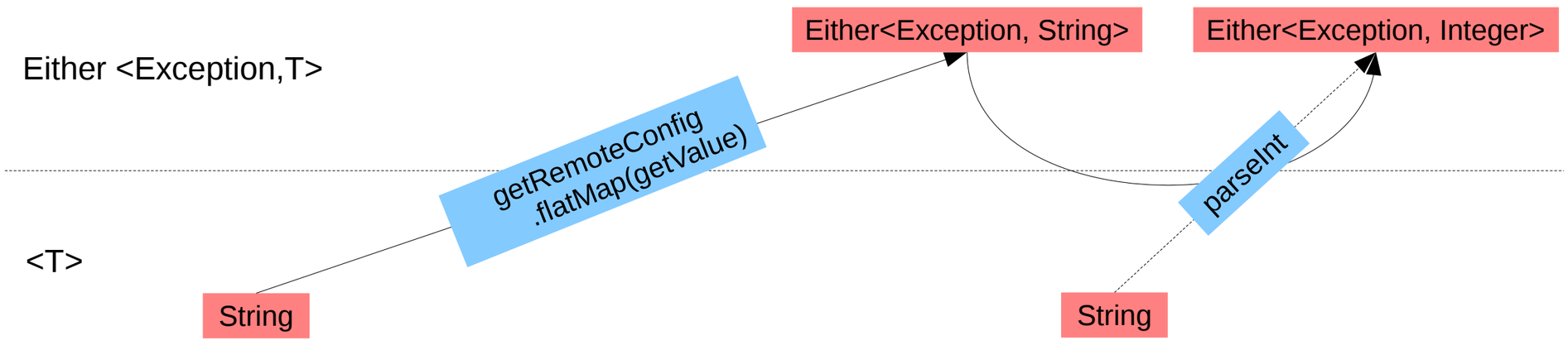
We've composed here! We've turned
three operations of type A -> Either<B>
into
one operation of type A -> Either <B>

getRemoteConfig goes from *<A>* to *Either<Exception, B>*
getValue goes from *<A>* to *Either<Exception, B>*
parseInt goes from *<A>* to *Either<Exception, B>*

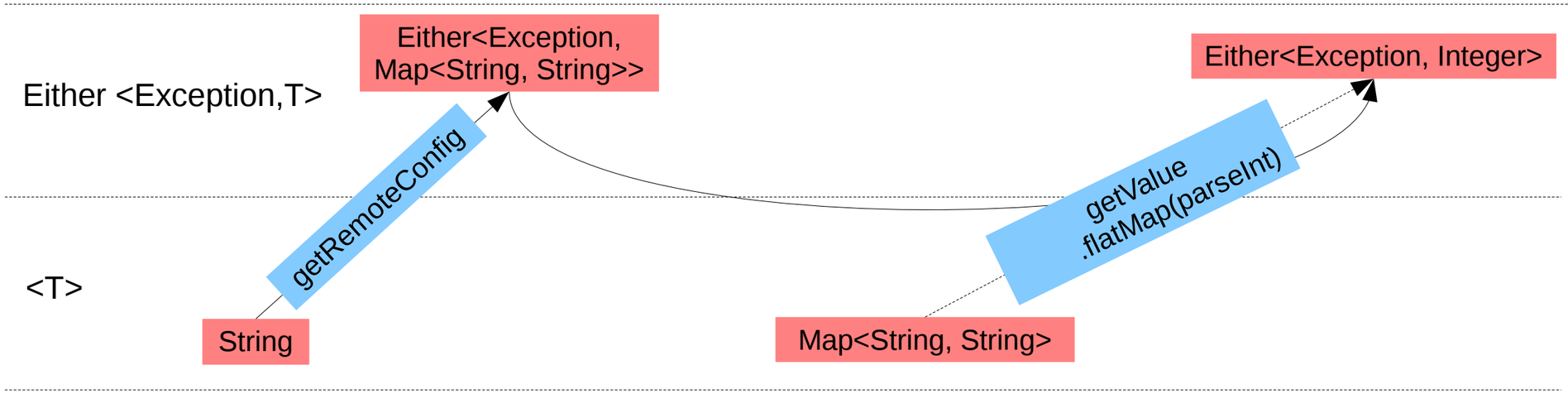
we can use them without needing to introduce a new thing called *RemoteConfigValueGetter* or *ValueGetterParser*.

Let's try joining different sub-pieces to one another!

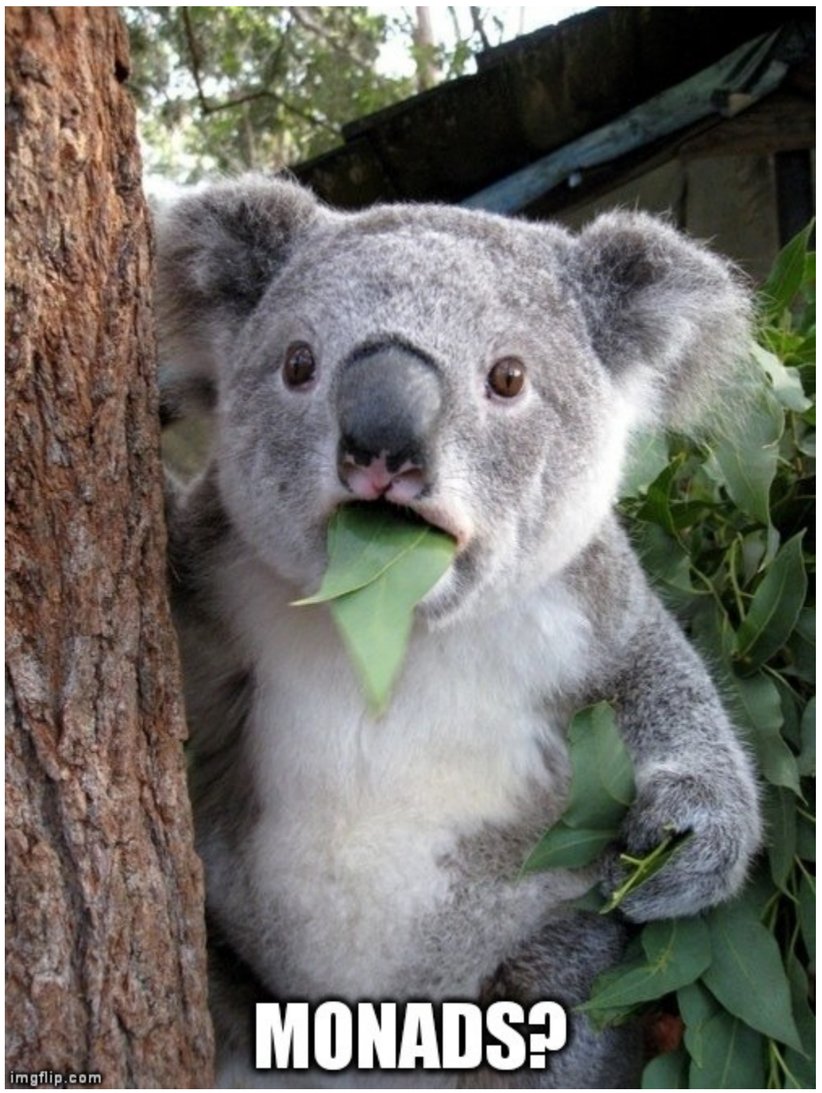
Left two functions squished into one function



Right two functions squished into one function

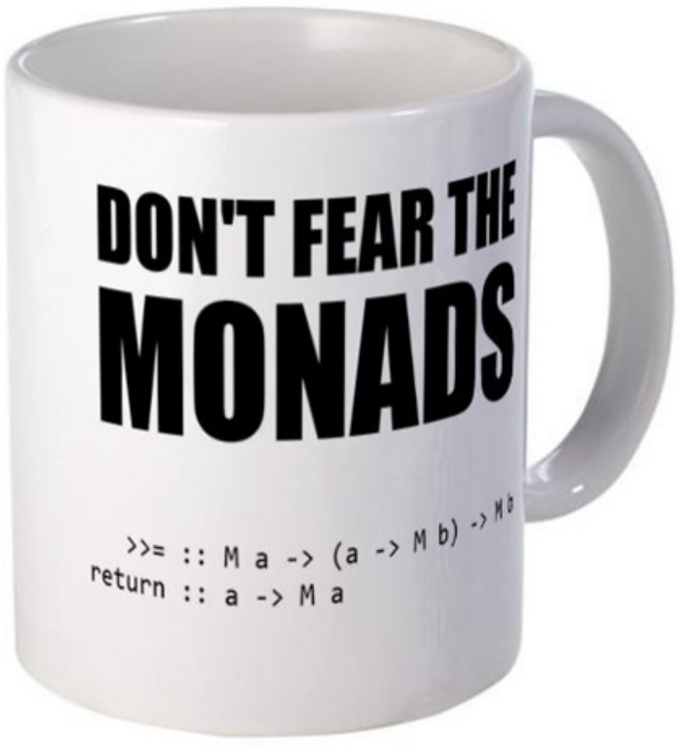


We just discovered monads!



Can we formalise this notion?

Step 1: Take definition of monad (from lucky mug)



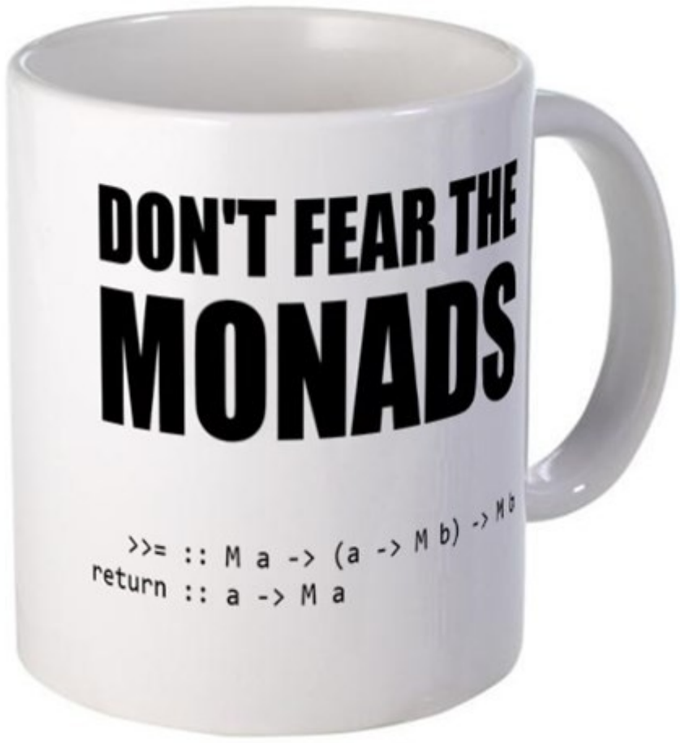
Something that implements the following two functions:

>>= :: M a -> (a -> M b) -> M b
return :: a -> M a

Step 2: Give the functions equally-meaningless names

flatMap :: M a -> (a -> M b) -> M b
of :: a -> M a

FORMALISE IT!



>>= :: M a -> (a -> M b) -> M b
return :: a -> M a

flatMap :: M a -> (a -> M b) -> M b
of :: a -> M a

From now on I'll just focus on “flatMap” - the interesting one!

Step 3: Rewrite in an Object-Oriented Manner

those 'a's and 'b's there are generic types, so let's capitalise them and surround them with echelons:

flatMap :: M <A> -> (<A> -> M <B>) -> M <B>

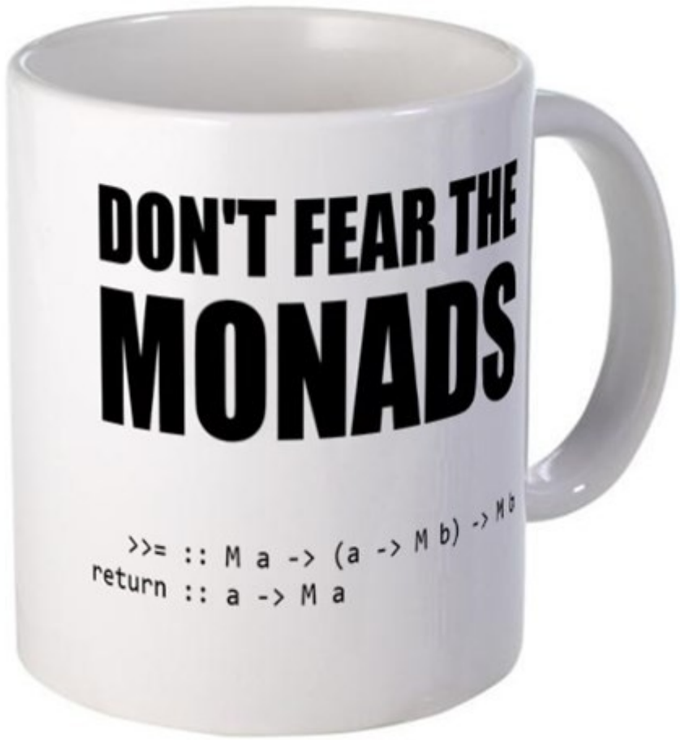
flatMap's first parameter can be “itself”, because it's now an object's method. So it should be left off the signature.

(M <A>).flatMap :: (<A> -> M <B>) -> M <B>

The last item is the return type, so let's put that on the left.

(M <B>) (M <A>).flatMap (<A> -> M <B>)

FORMALISE IT!



>>= :: M a -> (a -> M b) -> M b

flatMap :: M a -> (a -> M b) -> M b

flatMap :: M <A> -> (<A> -> M <B>) -> M <B>

(M <A>).flatMap :: (<A> -> M <B>) -> M <B>

(M <B>) (M <A>).flatMap (<A> -> M <B>)

Step 4: Deal with that functional parameter

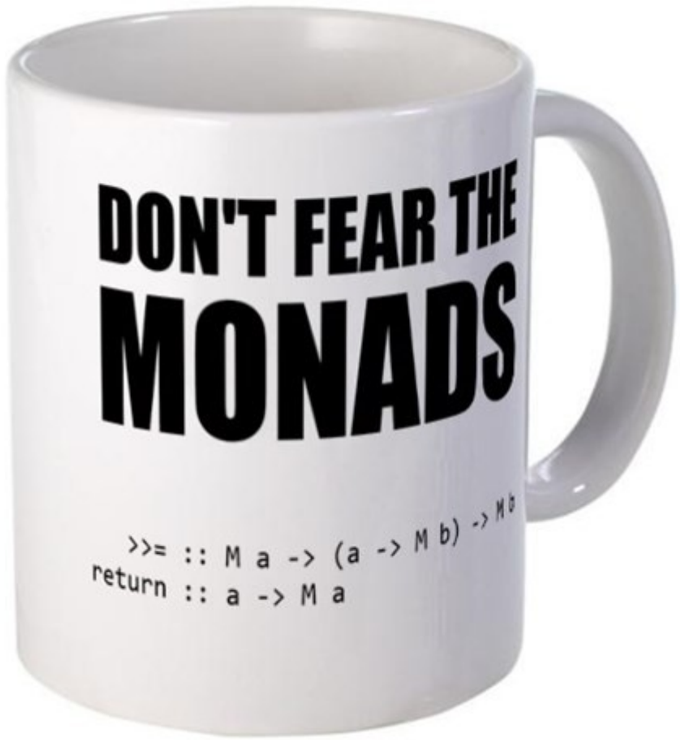
<A> -> M<B> is a function, so let's represent it with a Java 8 Lambda!

M <B> (M <A>).flatMap (Function<A, M<B>> mapper);

Step 5: Make it look more like a method

public <B> Monad<B> flatMap (Function<A, Monad<B>> mapper);

FORMALISE IT!



>>= :: M a -> (a -> M b) -> M b

flatMap :: M a -> (a -> M b) -> M b

flatMap :: M <A> -> (<A> -> M <B>) -> M <B>

(M <A>).flatMap :: (<A> -> M <B>) -> M <B>

(M <B>) (M <A>).flatMap (<A> -> M <B>)

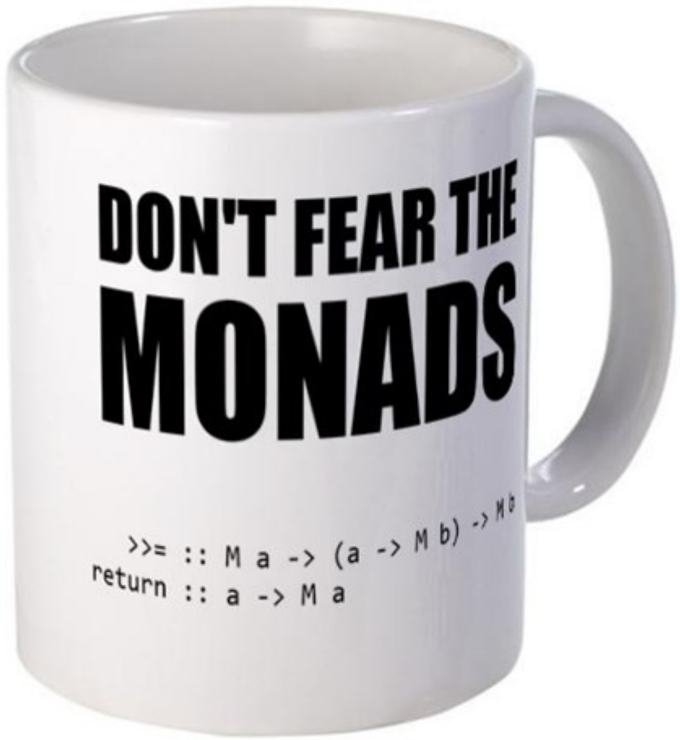
M <B> (M <A>).flatMap (Function<A, M<B>> mapper);

public <B> Monad<B> flatMap (Function<A, Monad<B>> mapper);

Step 6: Replace the A and B with more Java-looking letters

public <U> Monad<U> flatMap (Function<T, Monad<U>> mapper);

FORMALISE IT!



>>= :: M a -> (a -> M b) -> M b

flatMap :: M a -> (a -> M b) -> M b

flatMap :: M <A> -> (<A> -> M <B>) -> M <B>

(M <A>).flatMap :: (<A> -> M <B>) -> M <B>

(M <B>) (M <A>).flatMap (<A> -> M <B>)

M <B> (M <A>).flatMap (Function<A, M<B>> mapper);

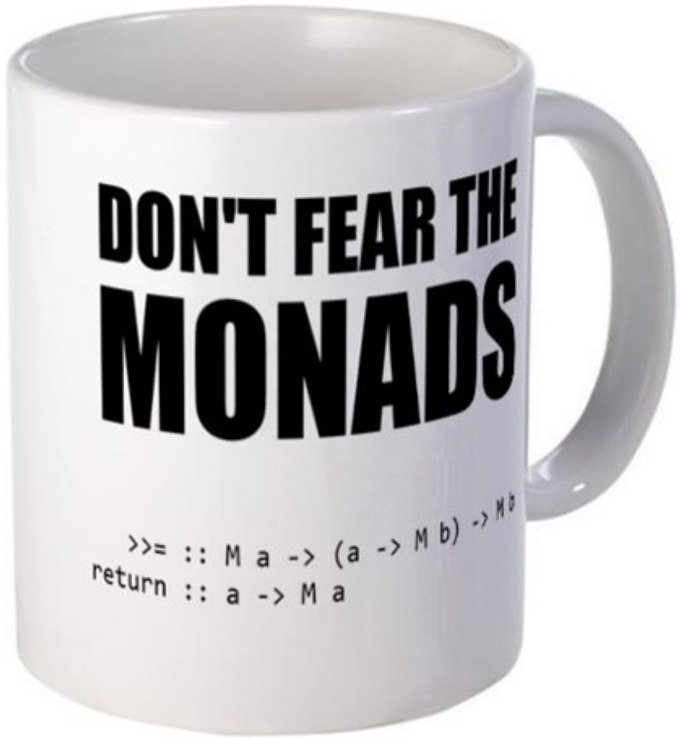
public <B> Monad<B> flatMap (Function<A, Monad<B>> mapper);

public <U> Monad<U> flatMap (Function<T, Monad<U>> mapper);

Step 7: Make the method *slightly* more generic

public <U> Monad<U> flatMap (Function<? super T> Monad<U>> mapper);

FORMALISE IT!



Step 8: Profit!

So we turned this:
>>= :: M a -> (a -> M b) -> M b

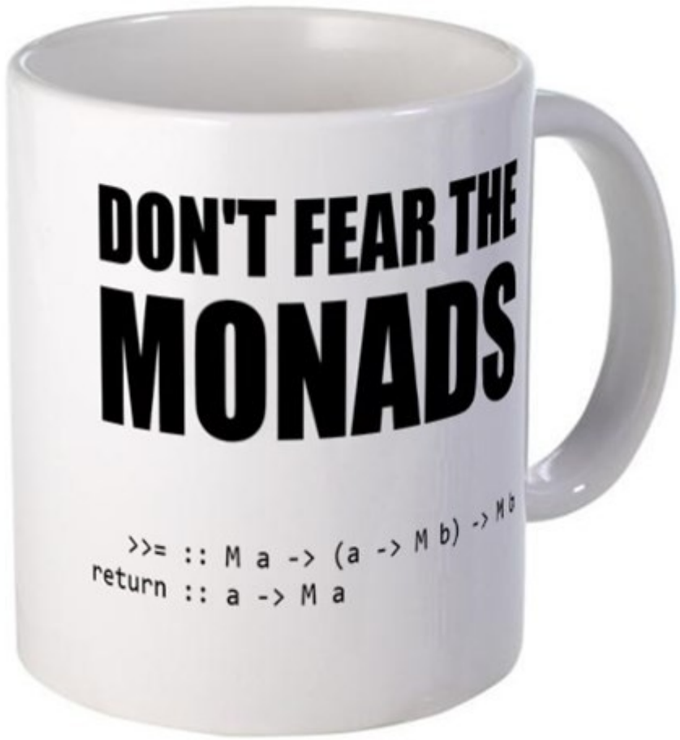
into this:
public<U> **Monad**<U> flatMap (Function<? super T> **Monad**<U>> mapper);

Where have we seen this before? The standard Java library!

java.util.Optional:
public<U> **Optional**<U> flatMap(Function<? super T, **Optional**<U>> mapper);

java.util.Stream:
*<R> **Stream**<R> flatMap(Function<? super T, ? extends **Stream**<? extends R>> mapper);*

FORMALISE IT!



And before I forget!

My lucky mug has two method signatures on it.

We looked at the interesting one, but just for completeness,
let's look at the other one

We already dealt with:
>>= :: M a -> (a -> M b) -> M b
public<U> **Monad**<U> flatMap (Function<? super T> **Monad**<U>> mapper);

Here's the other one:
return :: a -> M a

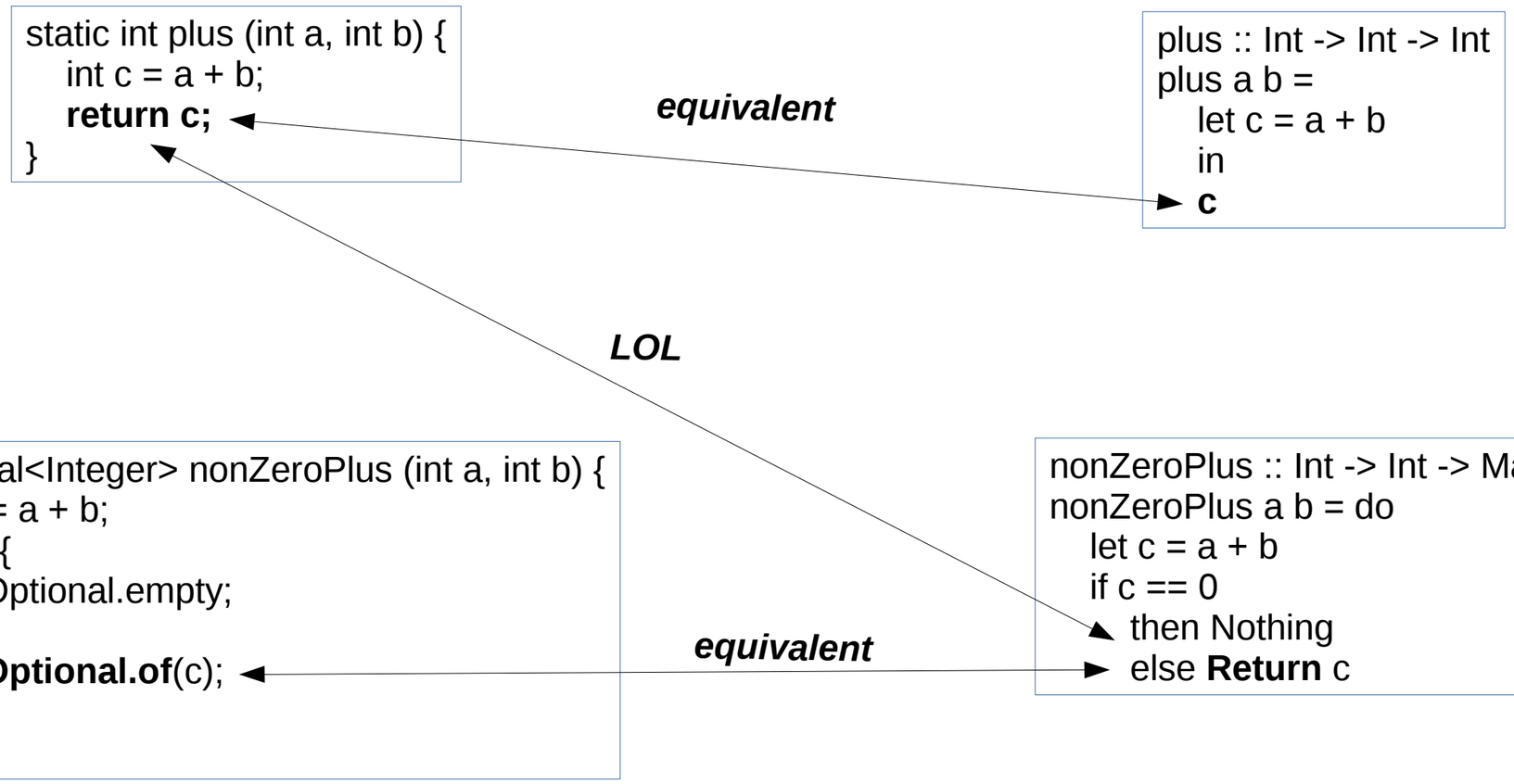
- Give it an equally silly name (“of” instead of “return”)
- Turn the 'a's into '<T>'s
- Move its return type from the right to the left
- The Java version is already static, so we don't have to OO-ify this one.

public static <T> Optional<T> of(T value);
public static <T> Stream<T> of(T t);

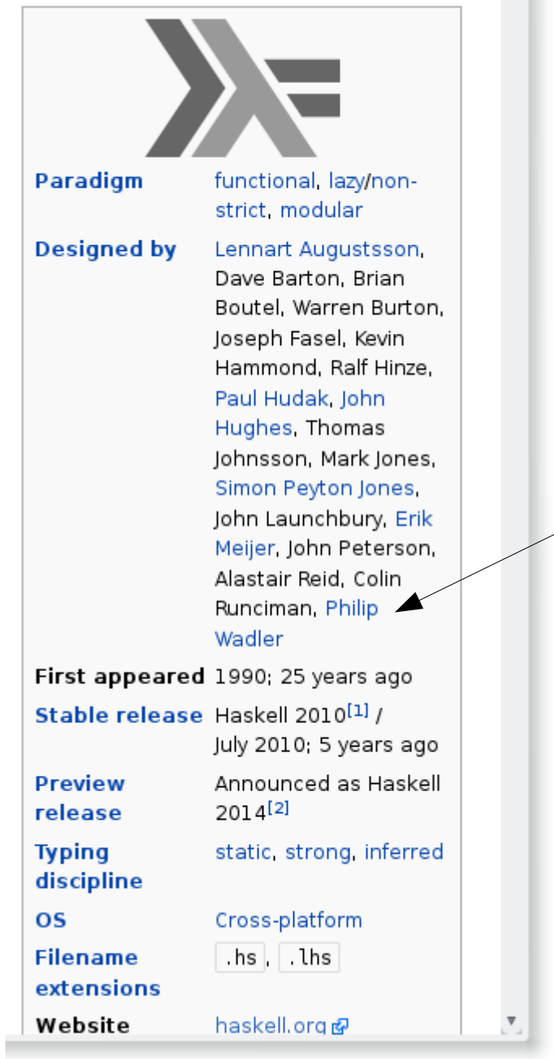
Side note:

Why on Earth did they call it “return” ?

It's the terrible computer-science equivalent of a **pun**!



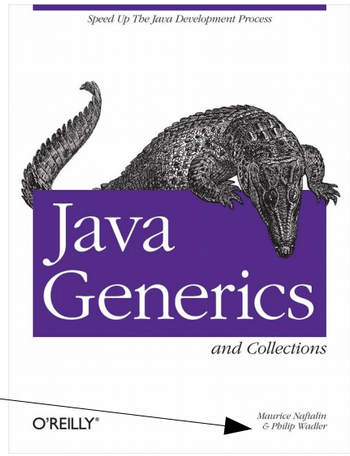
Who would do such a thing?



- ***Put monads into Haskell***

This guy!

Philip Wadler
http://homepages.inf.ed.ac.uk/wadler



- ***Designed Java Generics (along with Bracha, Stoutamire and Odersky)***

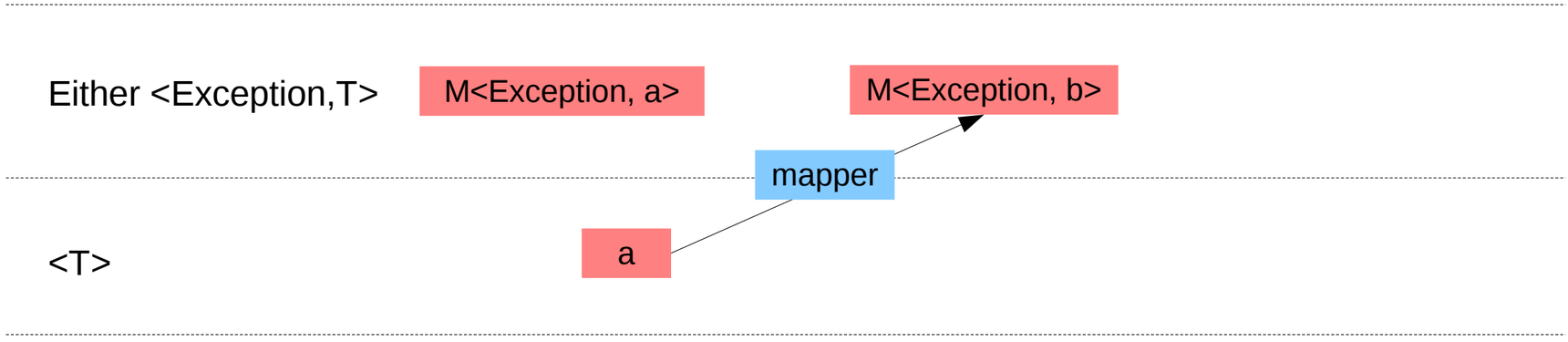
Here's some Java 8 standard library code.
Try to spot the bits he didn't influence!

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
       if (!isPresent())
        return empty();
    else {
        return Objects.requireNonNull(mapper.apply(value));
    }
}
```

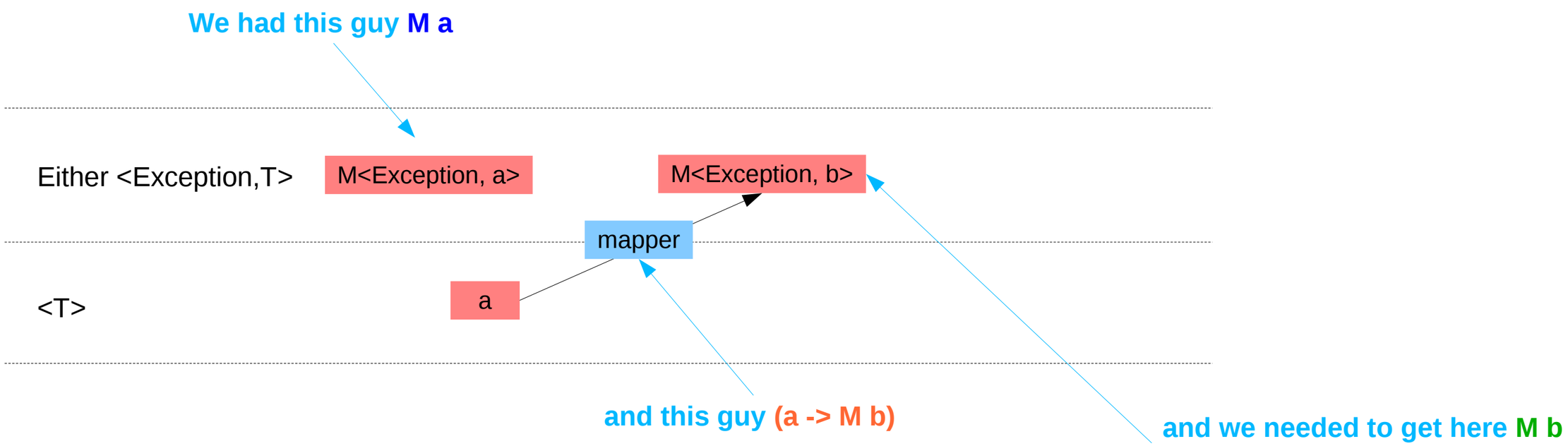
His thesis was entitled "**Listlessness is Better than Laziness**"

Recap

Remember what we were trying to do here?



Recap



This is what flatMap does!

***flatMap* is precisely the *most general* and *flexible* thing that lets you put these things together!**

M a -> (a -> M b) -> M b

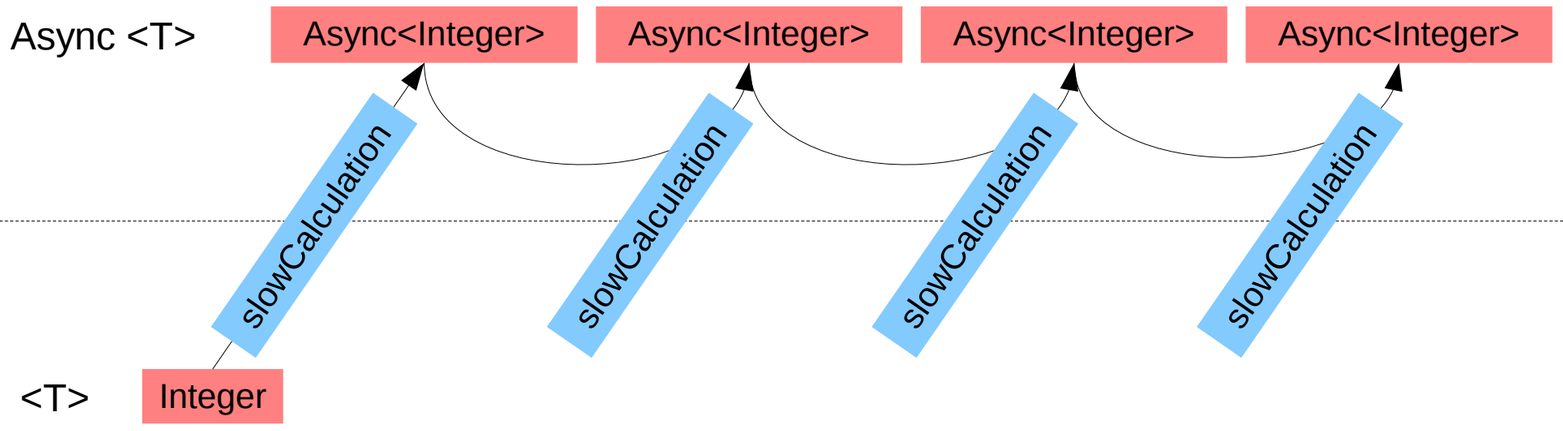
Here's the real kicker:

This type signature wasn't even a “clever answer” to our question, it was our question itself, just *disguised.*

“How do I get from M a and (a -> M b) to M b ?”

It's not just Either

Asynchronous computation



Four async computations flatMapped into one async computation

Putting them all together