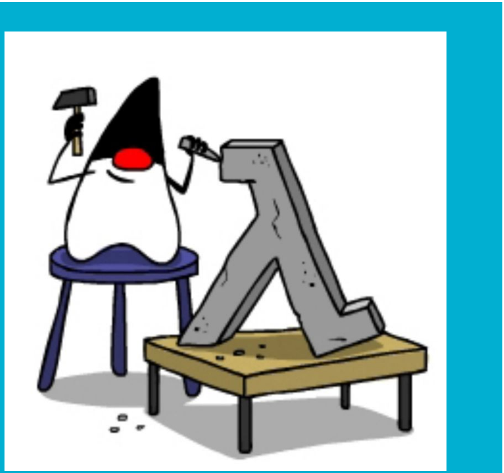
Java 8 Monads

1 | What a monad is and why should you care.

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

2 | Java 8 monad implementations



3 | Javaslang monad implementations

JAVASLANG

What a monad is and why you should care

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

What do these have in common?

Optional<T> Stream<T> Future<T>

What do these have in common?

Optional<T> Stream<T> Future<T>

You can ‘nest’ them,
and you can ‘flatten’ them.

Nesting Optional<T>

“Maybe I have a T”	Optional<T>
“Maybe I have an Optional<T>”	Optional<Optional<T>>
“Maybe I have an Optional<Optional<T>>”	Optional<Optional<Optional<T>>>

But it’s still one of two things.

It’s either a T or it’s not.

Optional<Optional<Optional<T>>>

But it's still one of two things.

It's either a T or it's not.

Optional<Optional<Optional<T>>>

So it's still this:

Optional<T>

Stream<Stream<Stream<T>>>

So it's really just a Stream<T>

I can get retrieve one *T* at a time.

Future<Future<Future<T>>>

So it's really just a Future<T>

I can get a *T* at some time in the future.

The mathematicians got there first…

…hence the funny name…

…and the Greek letters.

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

η - is what I’ve been calling “nesting”

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

μ - is what I’ve been calling “flattening”

Stream<T>

wrap on the
“outside”

Stream<Stream<T>>

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

wrap on the
“inside”

flatten

Stream<Stream<T>>

flatten

Stream<T>

[1,2,3,4,5]

wrap on the
“outside”

[[1,2,3,4,5]]

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

wrap on the
“inside”

flatten

[[1],[2],[3],[4],[5]]

flatten

[1,2,3,4,5]

Java 8 Monads

“nesting” in Java 8

This is the *.of(x)* method.

For instance:

```
	Stream.of(Stream.of(Stream.of(5)));

	Optional.of(Optional.of(3));
```

“flattening” in Java 8

What is your kid’s dog’s name?

“flattening” in Java 8

What is your kid’s dog’s name?

- Maybe you have a kid
- Maybe your kid has a dog
- Maybe the dog hasn’t been named yet

- Maybe you have a kid
- Maybe your kid has a dog
- Maybe the dog hasn’t been named yet

```
class You {
    Optional<Kid> getKid();
}

class Kid {
    Optional<Dog> getDog();
}

class Dog {
    Optional<Name> getName();
}
```

“flattening” in Java 8

What is your kid’s dog’s name?

```
you.getKid()
    .flatMap ( x -> x.getDog() )
    .flatMap ( x -> x.getName() )
```

“flattening” in Java 8

flatMap is the way to flatten in Java 8

It doesn’t correspond 1-1 to **μ**.

flatMap corresponds to ***“wrapping on the inside”*** and then applying ***μ***.

Harder to learn, but easier to code with

```
Optional.of( 3 )
   .map( doSomething )
   .μ()
   .map( doSomethingElse )
   .μ()
   .map( doAnotherThing )
   .μ();
```

```
Optional.of( 3 )
   .flatMap( doSomething )
   .flatMap( doSomethingElse )
   .flatMap( doAnotherThing );
```

Stream<X>

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "T(\eta_X)"'] \arrow[dr, equal] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"'] & T(X)
\end{tikzcd}$$

.map(f)

.flatMap(f)

.μ()

Stream<Stream<X>>

Stream<X>

Why all the maths?

- **Reasoning**

```
something
    .flatMap ( x -> of (3*x) )

==?

something
    .map ( x -> x*3 )
```

- Reasoning
- **Guiding development**

$$\begin{tikzcd}
T(X) \arrow[r, "\eta_{T(X)}"] \arrow[d, "\text{.map(f)}\quad T(\eta_X)"'] \arrow[dr, equal, "\text{.flatMap(f)}"] & T(T(X)) \arrow[d, "\mu_X"] \\
T(T(X)) \arrow[r, "\mu_X"', "{\text{.}\mu\text{()}}"' {yshift=-12pt}] & T(X)
\end{tikzcd}$$

```
class M<A> {

    M<B> flatMap(Function<A,M<B>> f) {
        return map(f).µ();
    }

}
```

- Reasoning
- Guiding development
- **Simplicity**

Which of these would you rather receive as a parameter?

```
Integer:
      [...,-2,-1,0,1,2,3,...]

Integer:
      [...,-2,-1,-0,null,0,1,2,3,...]
```

- Reasoning
- Guiding development
- Simplicity
- **Maths *just works***

```
x() == x()
```

(is ***true*** *in maths)*

Let’s look at some Java 8 “unmaths”

How does it help with coding?

- It can help with structure

Structure:
Three “programs” which fit together like Lego.

getKid

getDog

getName

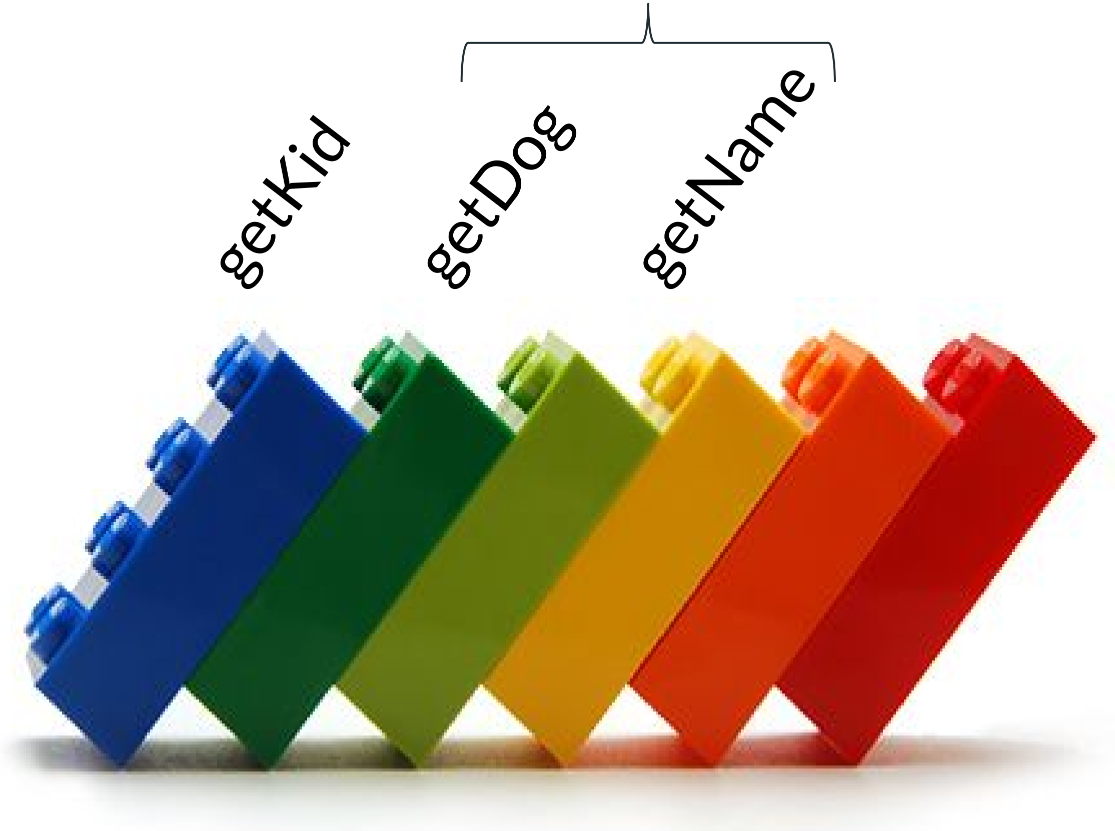
getKidsDog

getKid

getDog

getName

getDogsName

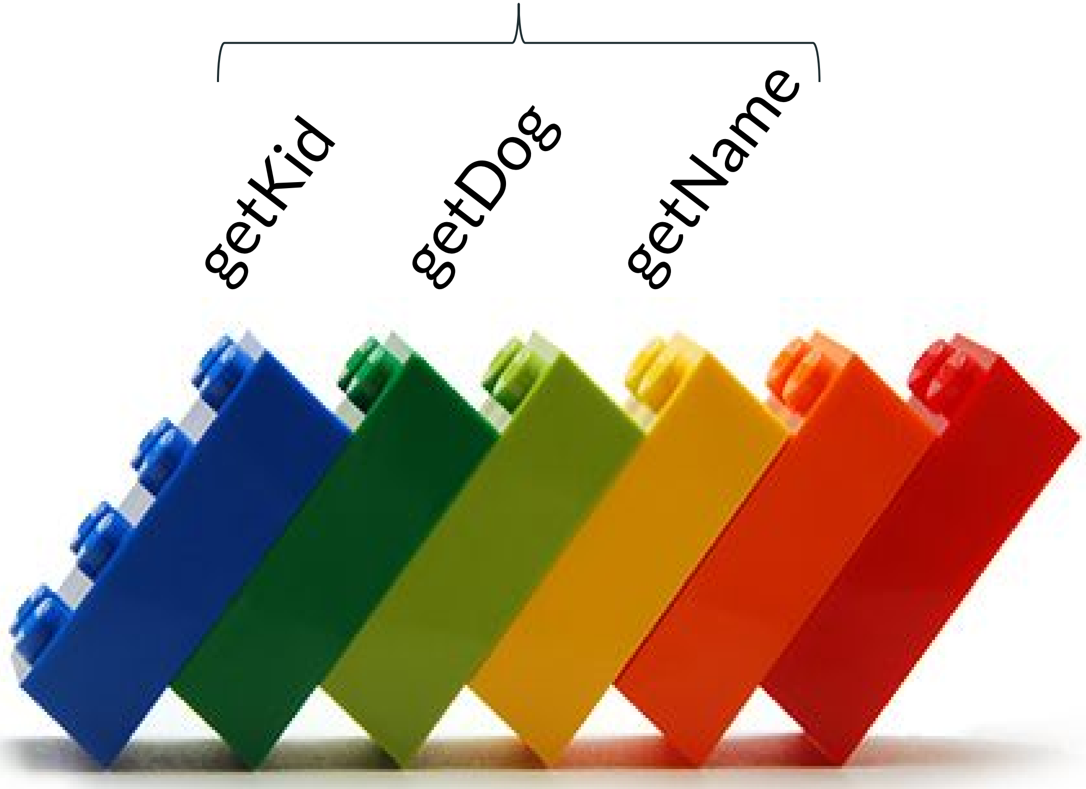


getKid

getDog

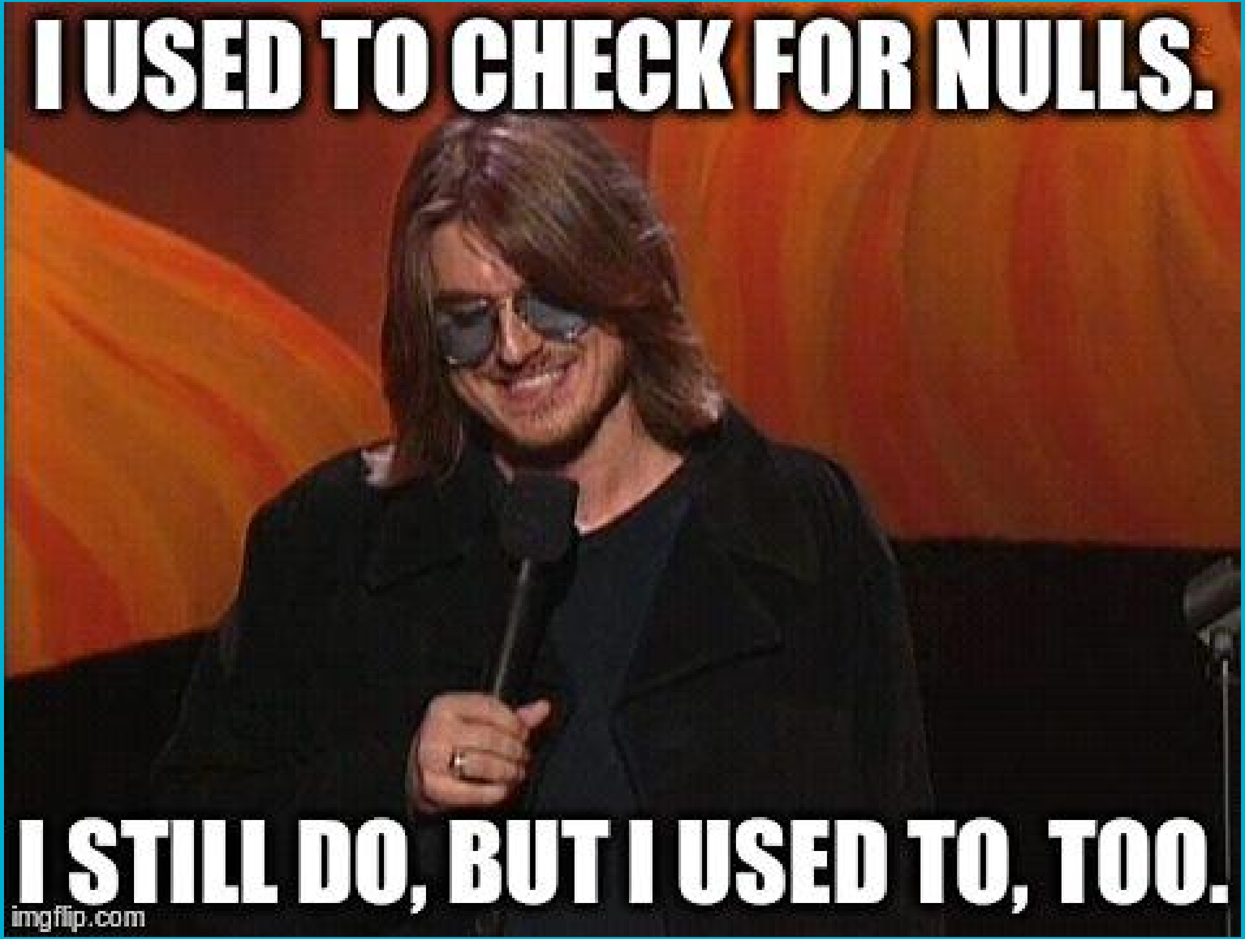
getName

getKidsDogsName



Optional<T> does null-checking for you, but…

- Only If you **use it like a monad**. Otherwise you're just doing manual checking in a different way.


I USED TO CHECK FOR NULLS.
I STILL DO, BUT I USED TO, TOO.
imgflip.com

Optional<T> does null-checking for you, but...

- Only If you **use it like a monad**. Otherwise you're just doing manual checking in a different way.
- Null-checking is a one-liner! This is not exactly solving a hard problem for you.

What about harder problems?

Streaming files

```
static Stream<File> walkDirectory(File directory) {

     Stream<File> filesInThisDirectory = Stream.of(directory.listFiles());

     return filesInThisDirectory
               .flatMap( x -> x.isDirectory()
                                   ? walkDirectory(x)
                                   : Stream.of(x) );
}
```

Stream.flatMap(...)

```
default <U> Stream<U> flatMap(final Function<? super T, ? extends Iterable<? extends U>> mapper) {
    Objects.requireNonNull(mapper, "mapper is null");
    return (Stream)(this.isEmpty()?Stream.Empty.INSTANCE:ofAll((Iterable)(new Iterator() {
    final Iterator<? extends T> inputs = Stream.this.iterator();
    java.util.Iterator<? extends U> current = java.util.Collections.emptyIterator();

    public boolean hasNext() {
        boolean currentHasNext;
        while(!(currentHasNext = this.current.hasNext()) && this.inputs.hasNext()) {
        this.current = ((Iterable)mapper.apply(this.inputs.next())).iterator();
        }

        return currentHasNext;
    }

    public U next() {
        return this.current.next();
    }
    })));
}
```

Future.flatMap(...)

```
default <U> Future<U> flatMapTry(CheckedFunction<? super T, ? extends Future<? extends U>> mapper) {
        Objects.requireNonNull(mapper, "mapper is null");
        Promise promise = Promise.make(this.executorService());
        this.onComplete((result) -> {
                result.mapTry(mapper).onSuccess(promise::completeWith).onFailure(promise::failure);
                });
        return promise.future();
}
```

What else is monadic?

LINQ (C#)

```
// DataContext takes a connection string
DataContext db = new   DataContext("c:\\northwind\\northwnd.mdf");

// Get a typed table to run queries
Table<Customer> Customers = db.GetTable<Customer>();

// Query for customers from London
var q =
   from c in Customers
   where c.City == "London"
   select c;

foreach (var cust in q)
   Console.WriteLine("id = {0}, City = {1}", cust.CustomerID, cust.City);
```

What else is monadic?

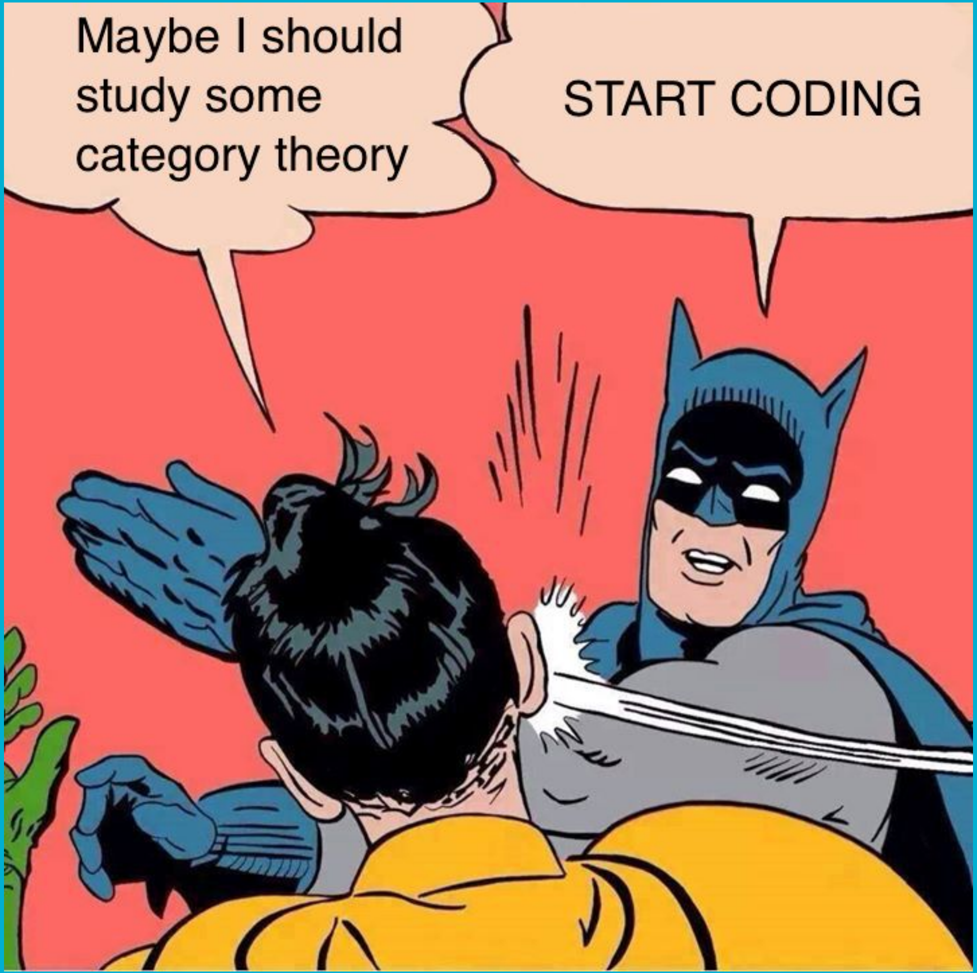
Parser Combinators (Haskell, Java, ...)

```
warcEntry :: Parser WarcEntry
warcEntry = do

    header <- warcHeader

    crlf

    body <- do
        contentLength <- getContentLength header
        compressionMode <- getCompressionMode header
        warcbody contentLength compressionMode

    crlf
    crlf

return (WarcEntry header body)
```

What else is monadic?

RxJava / Observables / FRP

What now?


Maybe I should study some category theory
START CODING