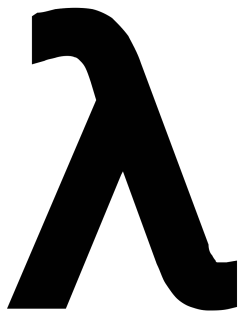

FUN with MONADS

Marcelo Piva

Software Engineer @ Nubank

@mpivaa
github.com/mpivaa
instagram.com/m.pivaa

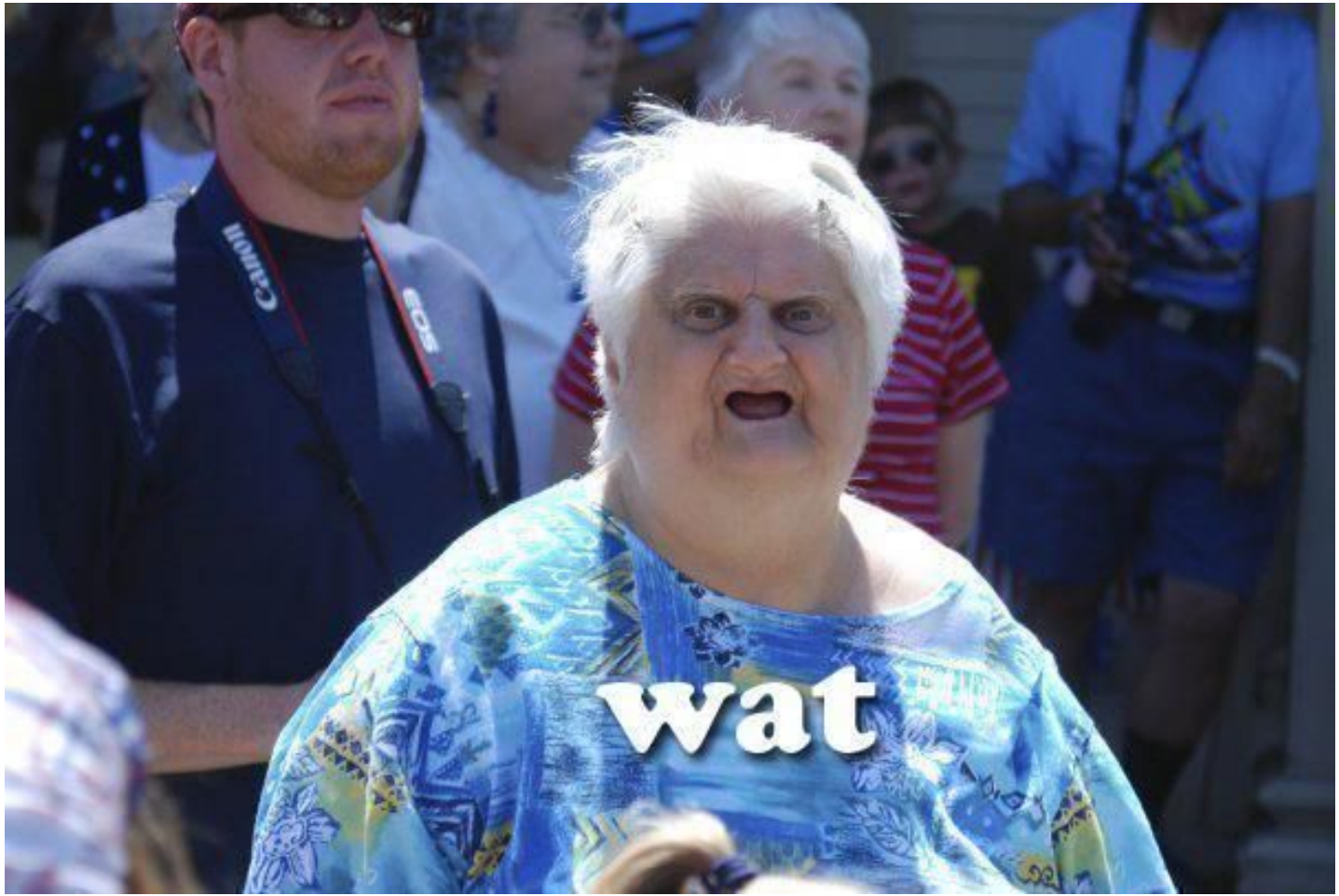
nubank



Monads


Google
Google Search
I'm Feeling Lucky

“Monads are just monoids in the category of endofunctors”


wat

Monad →

Why do we need **Monads**?

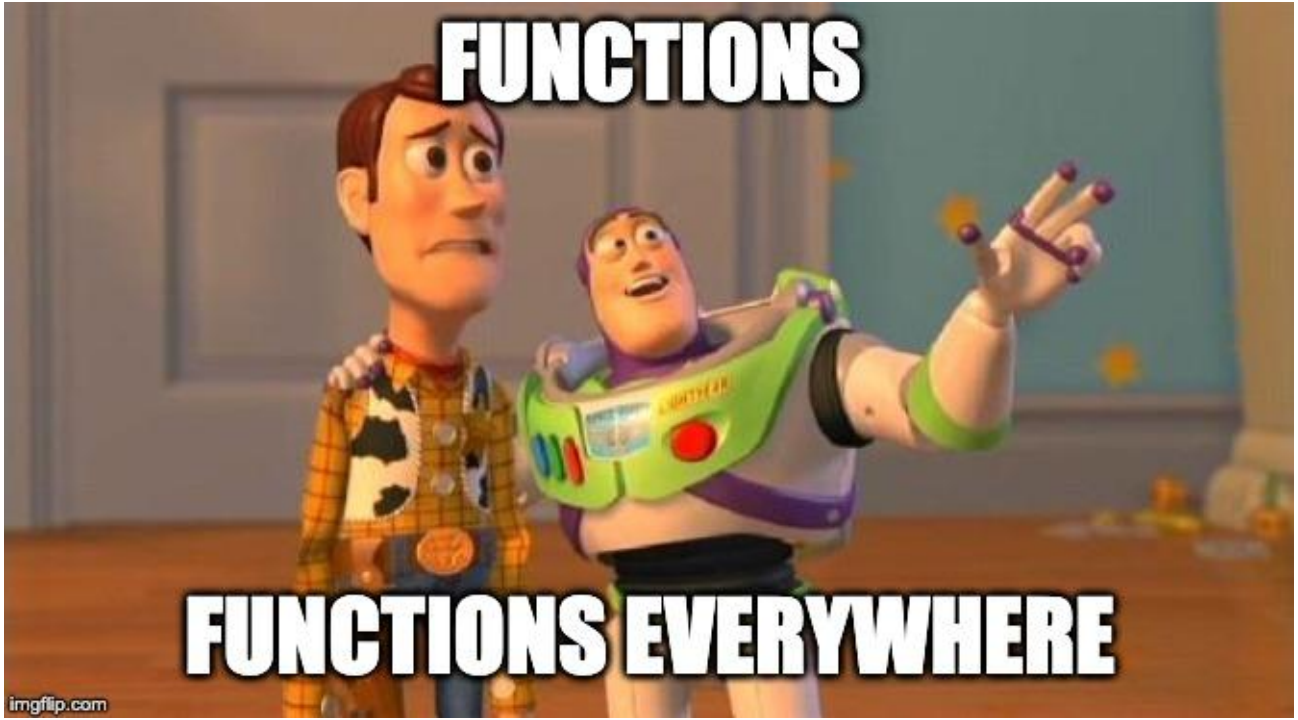
Why do we need **Monads**?

We want to use **only functions**.

We want to use **pure** functions.

We want to **compose** functions.

Class
Object
Instance
Inheritance
Polymorphism
Abstract Factory
Builder
Factory Method
Object Pool
Prototype
Singleton
Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Private Class Data
Proxy
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Null Object
Observer
State
Strategy


FUNCTIONS
FUNCTIONS EVERYWHERE
imgflip.com

Why do we need **Monads**?

We want to use **only functions**.

We want to use **pure** functions.

We want to **compose** functions.

Why do we need **Monads**?

We want to use **only functions**.

We want to use **pure** functions.

We want to **compose** functions.

Imperative mindset

```
1. a = do_a(x)
2. b = do_b(a)
3. if b
4.   do_c(b)
```

Functional mindset

```
do!(c(b(a(x))))
```

Functional mindset

```
do!(c(b(a(x))))
```

Functional mindset

```
do!(c(b(a(x))))
```

Functional mindset

```
do!(c(b(a(x))))
```

Functional mindset

```
do!(c(b(a(x))))
```

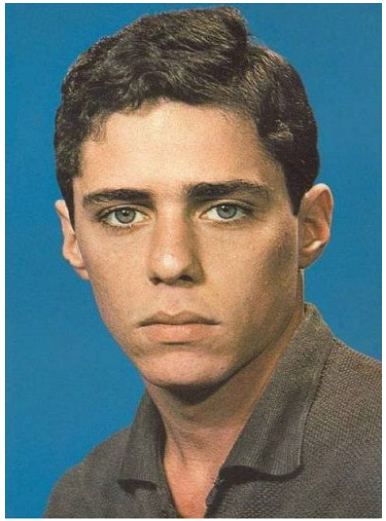
Composing **functions**

Imperative

```
public class TransactionRequest {
    public static String transact(Request request) {
        Transaction transaction = coerce(request);
        transaction = persist(transaction);
        notify(transaction);
    }
}
```

Imperative

```
public class TransactionRequest {
    public static String transact(Request request) {
        Transaction transaction = coerce(request);
        transaction = persist(transaction);
        notify(transaction);
    }
}
```

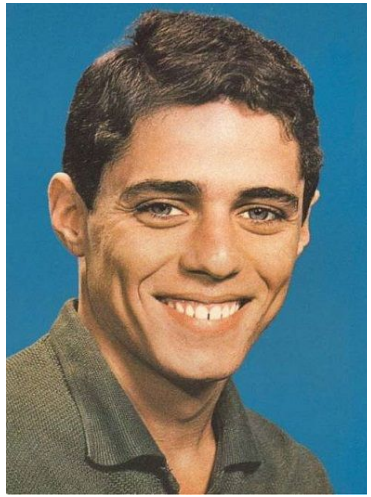


Functional

```
(defn transact! [transaction]
  (notify! (persist! (coerce transaction))))
```

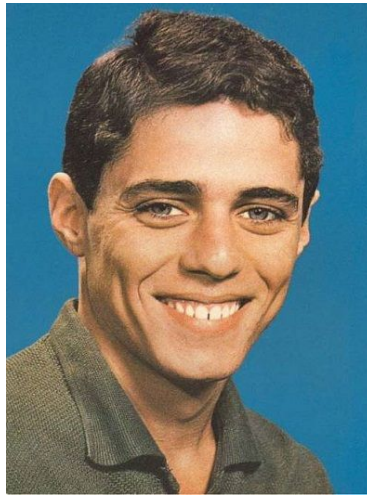
Functional

```
(defn transact! [transaction]
  (-> transaction
      coerce
      persist!
      notify!))
```



Functional

```
(defn transact! [transaction]
  (-> transaction
      coerce
      persist!
      notify!))
```



Imperative

```
public class TransactionRequest {
   public static String transact(Request request) {
       Transaction transaction = coerce(request);
       transaction = persist(transaction);
       notify(transaction);
   }
}
```

Functional

```
(defn transaction! [transaction]
 (-> transaction
     coerce
     persist!
     notify!))
```


I. AM. HAPPY.
LIFE IS GOOD.
quickmeme.com

Imperative

```
public class TransactionRequest {
    public static String transact(Request request) {
        Transaction transaction = coerce(request)
        transaction = persist(t
        notif
    }
}
```

Functional

```
(defn transacti
      persist!
      notify!))
```

NullPointerException

Imperative

```
public class TransactionRequest {
    public static String transact(Rec
        Transaction transaction = coe
        transaction = persist(t
        notif
    }
}
```

Functional

```
tion
t!
))
```

What happens when the request is invalid?

```
public class TransactionRequest {
    public static String transact(Request request) {
        Transaction transaction = coerce(request);
        transaction = persist(transaction);
        notify(transaction);
    }
}
```

What happens when the request is invalid?

```
public class TransactionRequest {
    public static String transact(Request request) {
        Transaction transaction = coerce(request);
        transaction = persist(transaction);
        notify(transaction);
    }
}
```

Or the transaction is not persisted?

What happens when the request is invalid?

```
public class TransactionRequest {
   public static String transact(Request request) {
       Transaction transaction = coerce(request);
       transaction = persist(transaction);
       notify(transaction);
   }
}
```

Or the transaction is not persisted?

Or the notification is not sent?

We need to handle **errors**

We need to handle **errors**

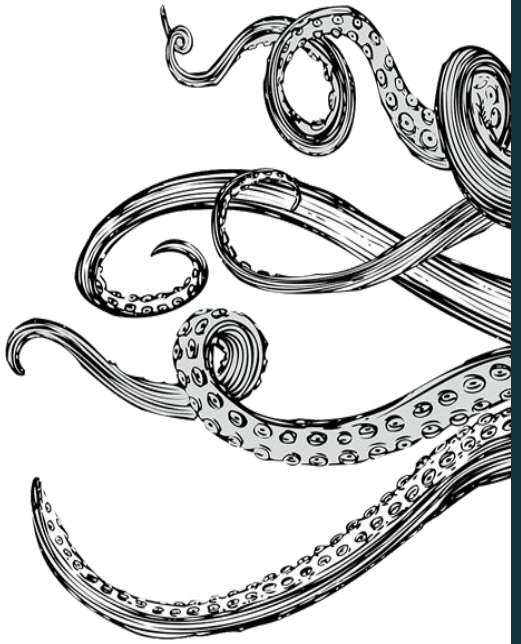
```
public static String transact(Request request) {
    try {
        Transaction transaction = coerce(request);
    } catch (Exception e) {
        return "Error adapting request";
    }

    try {
        transaction = persist(transaction);
        if(!transaction) {
            return "Error persisting transaction";
        }
    } catch(DBError e) {
        return "DB Error";
    }

    try {
        if(!notify(transaction)) {
            return "Error notifying transaction";
        }
    } catch(SMTPError e) {
        return "SMTP Error";
    }

    return "OK";
}
```

We need to handle **errors**



```
public static String transact(Request request) {
    try {
        Transaction transaction = coerce(request);
    } catch (Exception e) {
        return "Error adapting request";
    }

    try {
        transaction = persist(transaction);
        if(!transaction) {
            return "Error persisting transaction";
        }
    } catch(DBError e) {
        return "DB Error";
    }

    try {
        if(!notify(transaction)) {
            return "Error notifying transaction";
        }
    } catch(SMTPError e) {
        return "SMTP Error";
    }

    return "OK";
}
```

We need to handle **errors**

```
public static String transact(Request request) {
    try {
        Transaction transaction = coerce(request);
    } catch (Exception e) {
        return "Error adapting request";
    }

    try {
        transaction = persist(transaction);
        if(!transaction) {
            return "Error persisting transaction";
        }
    } catch(DBError e) {
        return "DB Error";
    }

    try {
        if(!notify(transaction)) {
            return "Error notifying transaction";
        }
    } catch(SMTPError e) {
        return "SMTP Error";
    }

    return "OK";
}
```

We need to handle **errors**

We need to handle **errors**

Exceptions = GOTO

When dealing with flow control

We need to handle **errors**

Before

```
(defn transact! [transaction]
  (-> transaction
      coerce
      persist!
      notify!))
```

We need to handle **errors**

Before

```
(defn transact! [transaction]
  (-> transaction
      coerce
      persist!
      notify!))
```

After

```
(defn transact! [transaction]
  (->= transaction
       coerce
       persist!
       notify!))
```

We need to handle **errors**

Befo

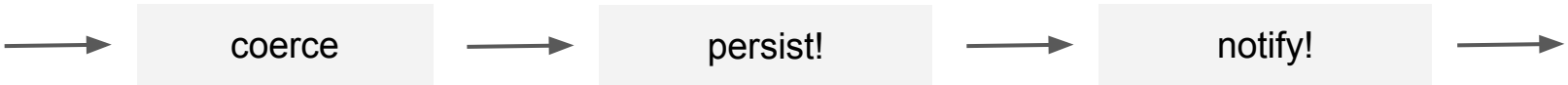
fter

```
(defn transact! [tr
  (-> transaction
      coerce
      persist!
      notify!))
```

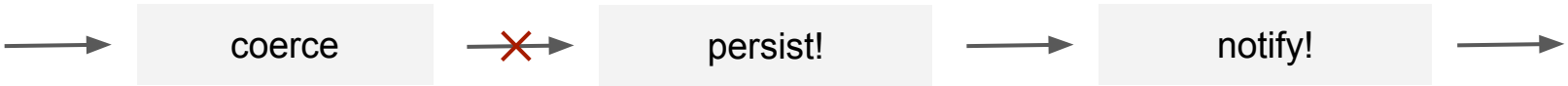
```
[transaction]
on
```

Monads

Composing functions



Composing functions



Return the **context** with the **result**

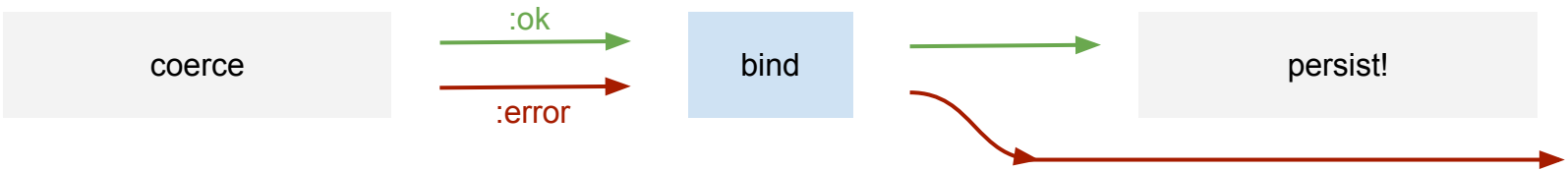
```
type Return = {:ok value} | {:error error}
```

How do we **compose** functions?

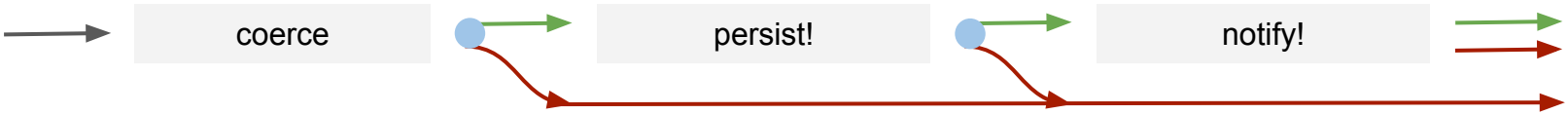
coerce

persist!

Enters the **bind** function



Composing with **bind**



How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

How **bind** works?

```
(defn bind [m f]
 (match m
   {:error e} e
   {:ok    v} (f v)))
```

Now we can **compose** again

```
(defn transact! [transaction]
 (let [transaction? (bind {:ok transaction} coerce)
       transaction? (bind transaction? persist!))
       transaction? (bind transaction? notify!)]
   transaction?))
```

Now we can **compose** again

```
(defn transact! [transaction]
 (let [transaction? (bind {:ok transaction} coerce)
       transaction? (bind transaction? persist!)
       transaction? (bind transaction? notify!)]
   transaction?))
```

We've just created a **Monad**,
and it's called **Either**

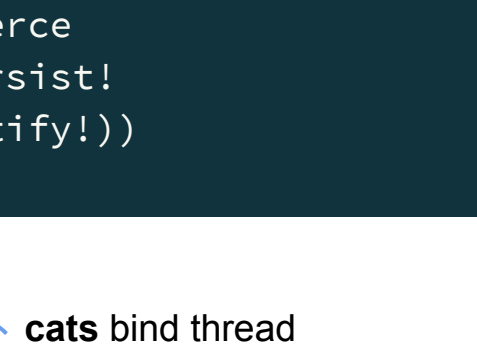
There are others, like: Maybe, IO, State...

We can do **better**

```
(defn transact! [transaction]
 (-> {:ok transaction}
     (bind coerce)
     (bind persist!)
     (bind notify!))
```

We can do **better**

```
(defn transact! [transaction]
  (->= transaction
       coerce
       persist!
       notify!))
```



cats bind thread

Libs like **cats** do all the hard work
https://github.com/funcool/cats
And can make our code cleaner

What is a **Monad**?

Just a **type** and **protocol**

Type

```
type Either = {:ok value} | {:error error}
```

```
type Maybe = {:ok value} | :nothing
```

Protocol

```
bind :: Either a -> (a -> Either b) -> Either b
return :: a -> Either a
```

Monads

everywhere?


THE PERFECT WORLD IS ALL IN
YOUR IMAGINATION
imgflip.com

Thanks!

- @mpivaa
- github.com/mpivaa
- instagram.com/m.pivaa

- https://t.me/clojurebrasil