What are monads?

@jlgarhdez

slides

But first...

What is functional programming?

Functional programming!

A programming paradigm that:

- treats functions as first class citicens
- encourages immutability
- avoids global state
- makes your code honest
- Avoids side effects (encourages purity)

What are side effects?

Everything that your code does, apart of operating with values.

How can we implement this?

```
public int sum(Int a, Int b) {
  // ???
}
```

```java
public int sum(int a, int b) {
  launchMissiles(); // Side effect!

  return a + b;
}
```

```java
public int sum(int a, int b) {
  this.pass = 4; // Side effect!

  return a + b;
}
```

```java
public int sum(int a, int b) {
  throw new Exception("message"); // Side effect!

  return a + b;
}
```

More bits of FP

Imagine this

```
class IO {
  def read(): String = readLine()// reads from standard in
  def write(text: String): Unit = println(text)// writes t
}
```

you could use it in the following way:

```
def hello(): String = {
  write("what is your name?")
  val name = read()
  val hello = "hello" + name
  write(hello)
  return hello
}
```

The good part:

That that code is idiomatic and concise!

The bad part:

That code is not pure! (has side effects)

How can we fix this?

Transform this

```
class IO {
  def read(): String = readLine()
  def write(text: String): Unit = println(text)
}
```

Into this!

This is our small `IO` language

```
trait IO[A]
case class Read() extends IO[String]
case class Write(str: String) extends IO[Unit]
```

This kind of construction is called Algebraic Data Type.

Interpreter

Since our previous ADT is not *effectful*, meaning that can not execute side effects, we need its companion, the interpreter!

```
def interpret(op: IO[A]): A = op match {
  case Read() => readLine()
  case Write(text) => printLn(text)
}
```

Now, let's write our `hello` function!

```
def pureHello(): IO[String] = {
  Write("what is your name?")
  val name: IO[String] = Read()
  Write("hello" + name) // ERROR!, you can not concat
  // String and IO[String]
}
```

How can we fix it?

Lets add a way to sequence IO operations to our ADT!

```
trait IO[A]
case class Read() extends IO[String]
case class Write(str: String) extends IO[Unit]
case class Sequence[A, B](
  fa: IO[A],
  fn: A => IO[B])
  extends IO[B]
```

Can we write `hello` now?

```
def pureHello: IO[String] = {
  Sequence(
    Write("What is your name?"),
    (_) => {
      Sequence(
        Read(),
        (name) => {
          Write("hello, " + name)
        }
      )
    })
}
```

We can't! we are returning an `IO[Unit]`, because of the last `Write()`... Lets fix that!

Add another case to our ADT

Let's add a way to put values inside `IO`

```
trait IO[A]
case class Read() extends IO[String]
case class Write(str: String) extends IO[Unit]
case class Sequence[A, B](
  fa: IO[A],
  fn: A => IO[B]) extends IO[B]
case class Point[A](a: A) extends IO[A]
```

Finally, we can write it now!

```
def pureHello: IO[String] = {
  Sequence(
    Write("What is your name?"),
    (_) => {
      Sequence(
        Read(),
        (name) => {
          Sequence(
            Write("hello, " + name),
            (_) => {
              Point("hello, " + name)
            }
          )
        }
      )
    })
}
```

The good parts

That code is pure!

The bad parts

THAT CODE IS UGLY

Let's fix that!

Introducing TypeClasses

Typeclasses are a way to give more behaviour to an class. You can think of them like interfaces in OOP.

But better

Much better

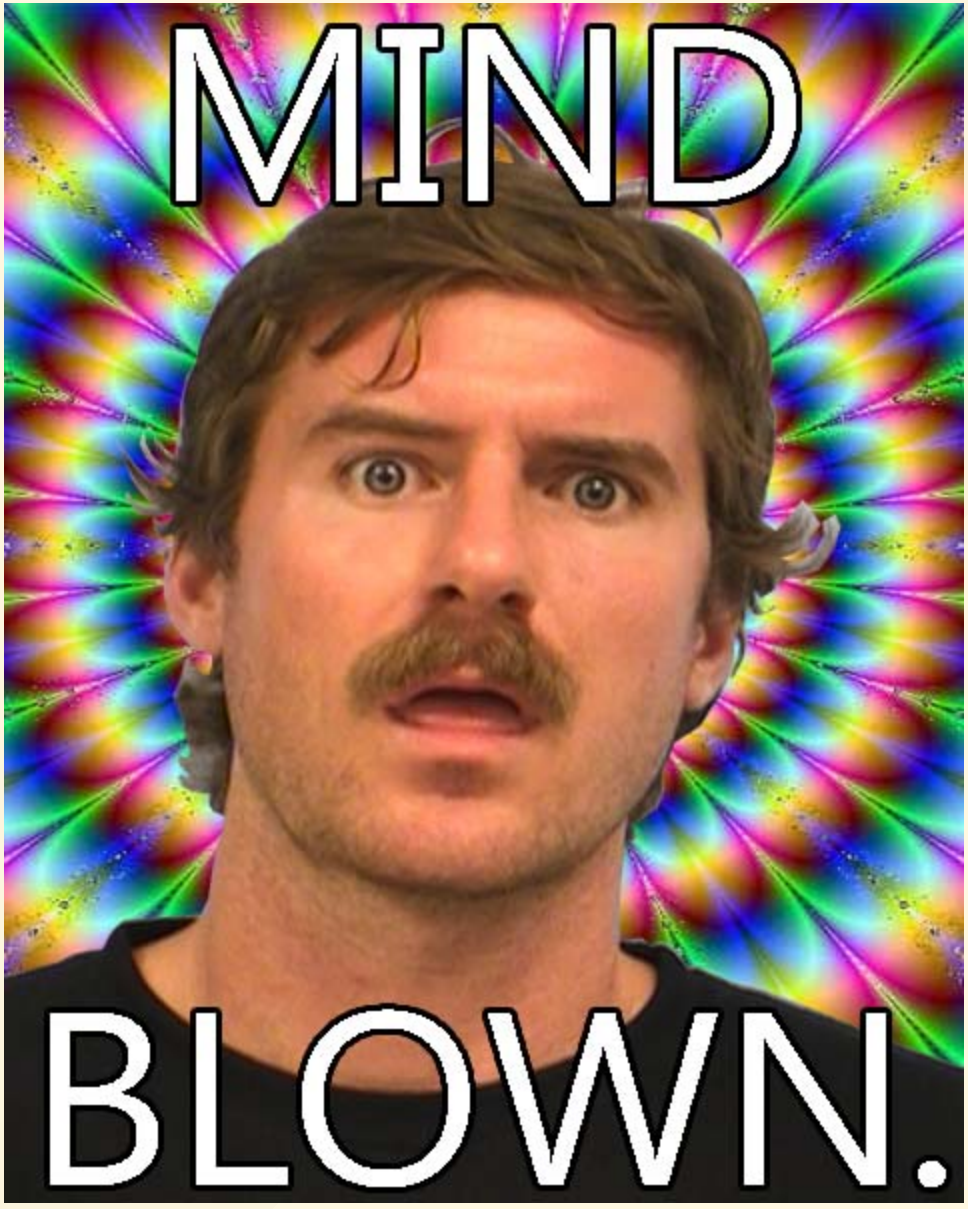
Some typeclasses

```
trait Eq[A] {
  def equals(a: A, b: A): Bool
}

val intEquality = new Eq[Int] {
  def equals(a: Int, b: Int) = a == b
}
```

```
trait Show[A] {
  def show(a: A): String
}

val showInt = new ToString[Int] {
  def show(a: Int) = a.toString
}
```

Guess what?

Monad is a typeclass!


MIND
BLOWN.


COMEDY
CENTRAL

Monad

```
trait Monad[M[_]] {

  def point[A](a: A): M[A]

  def flatMap[A, B](ma: M[A])(fn: A => M[B]): M[B]

}
```

What are monads?

Monad is a typeclass for defining imperative languages.

Back to our IO example

Remember all the weirdness we needed to do in order to do our `pureHello` function?

We are really close to simplifying it a lot!

Let's create a `Monad` instance for our `IO` language first!

```
val ioMonad = new Monad[IO] {
  def point[A](a: A): IO[A] = Point(a)
  def flatMap[A, B](ma: IO[A])(fn: A => IO[B]): IO[B] =
    Sequence(ma, fn)
}
```

Now, let's rewrite our `pureHello`

```
def pureHello: IO[String] = for {
  _ <- Write("What's your name?")
  name <- Read()
  _ <- Write("Hello, " + name)
} yield "Hello, " + name
```

We only need to review our interpreter...

New interpreter

```
def ioInterp[A](op: IO[A]): A = op match {
  case Write(text) => println(text)
  case Read() => readLine()
  case Point(x) => x
  case Sequence(x, fn) => ioInterp(fn(ioInterp(x)))
}
```

And, combine!

```
ioInterp(pureHello()) // this actually does IO
                      // thanks to the interpreter
```

Small recap

- Functional programming is cool
- Typeclasses are here to help
- Monad is for imperative programming
- Monads + Interpreters = WIN!

We are done,

thanks!

@jlgarhdez

meetup.com/FP-Madrid/

meetup.com/Haskell-MAD/