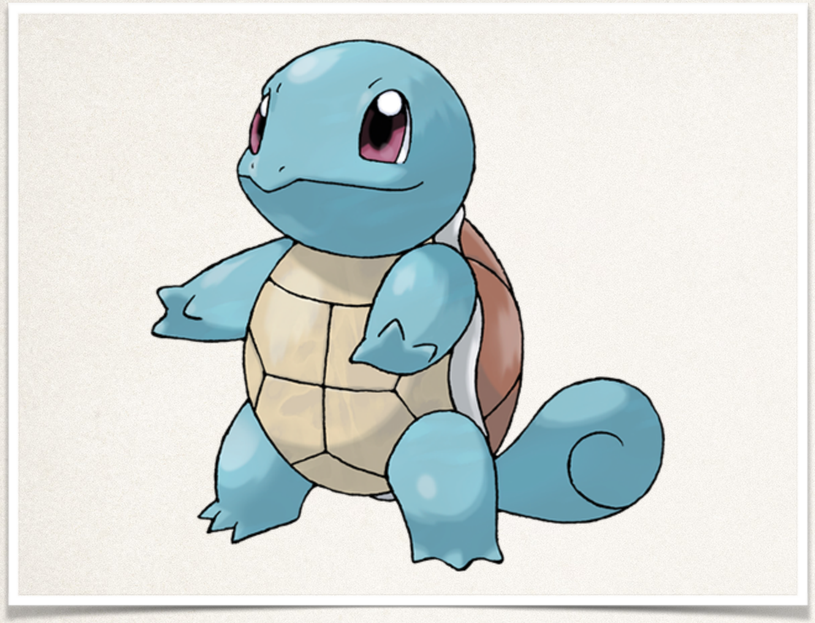
Kotlin na ponta da Flecha

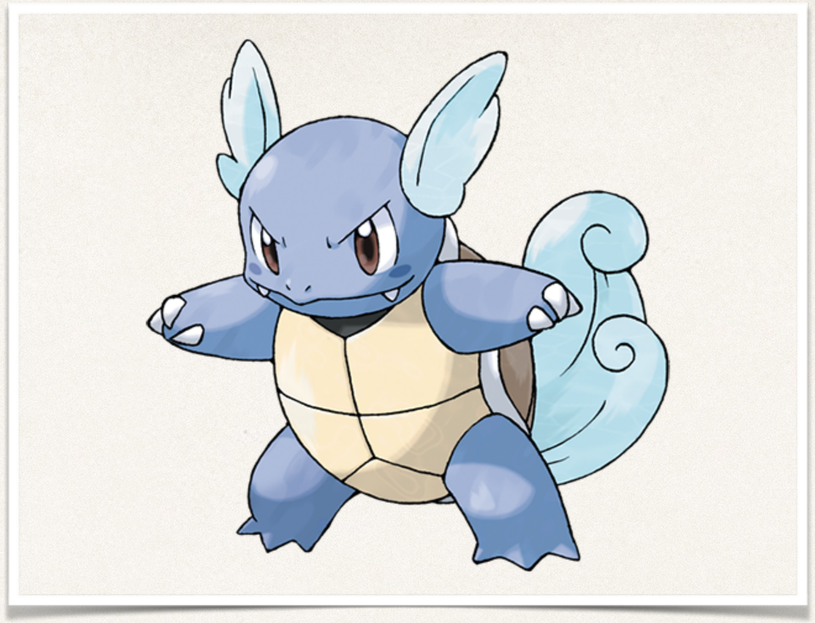
Uma introdução a utilização do paradigma funcional junto ao KOTLIN, através da biblioteca ARROW

Jonathas Hernandes

- Desenvolvedor iOS
  - Apple Developer's Academy
- Desenvolvedor Android
  - 4all
  - IWS - ISOBAR
  - GETNET
- Desenvolvedor Mobile / FLUTTER
  - KOBE



- Calculo Lambda
  - O valor de seu output depende apenas do parâmetro de determinada função.
  - 1930~



- Procedural
  - Instrui diretamente o computador como completar a determinada tarefa em passos lógicos.
  - 1957–1964

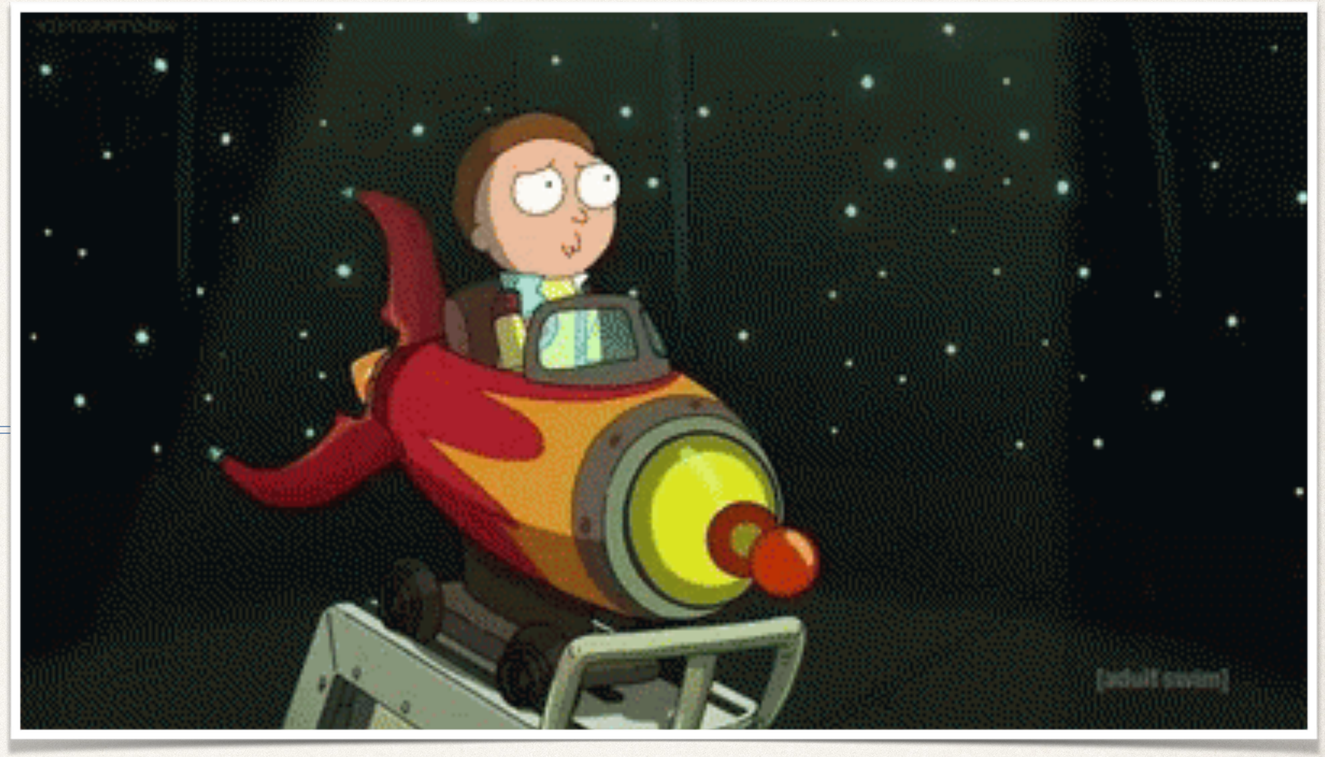
- Orientação a Objeto
  - O programa é desenvolvido a partir da criação de objetos que interagem entre si.
  - 1960

KOTLIN

Paradigma Funcional

É como dirigir um carro! Só que ele voa, navega em 3 dimensões e não tem ré.

Basicamente, o que muda muda é a forma de navegar. Segue a mesma física - ou seja, acontece o mesmo se você colidir com um muro em alta velocidade!




```
fun sum(left : Int, right: Int): Int {
    return left + right
}
```

Pureza

Funções são como filhotes: não podem fazer mal a ninguém.

```
fun simpleLoop(end: Int) : Int {
    var acc = 0
    for (i in 0..end) {
        acc += i
    }
    return acc
}
print(simpleLoop(10))
```

```
fun sumRange(start : Int, end : Int, acc : Int) : Int{
    if (start > end) {
        return acc
    }
    return sumRange(start+1, end, acc + start)
}

print(sumRange(1, 10, 0))
```

Imutabilidade

Não existem variáveis na programação funcional.

```
val sum : (Int, Int) -> Int = {
    left: Int, right: Int ->
    left + right
}
```

```
fun receivingFunctionSum(value : Int,
                         func : (Int, Int) -> Int) : Int {
    return value + func(10, 10)
}

receivingFunctionSum(10, sum)
```

Funções = Cidadãos de primeira ordem

Funções podem ser definitas tanto como variáveis como parâmetros de outras funções

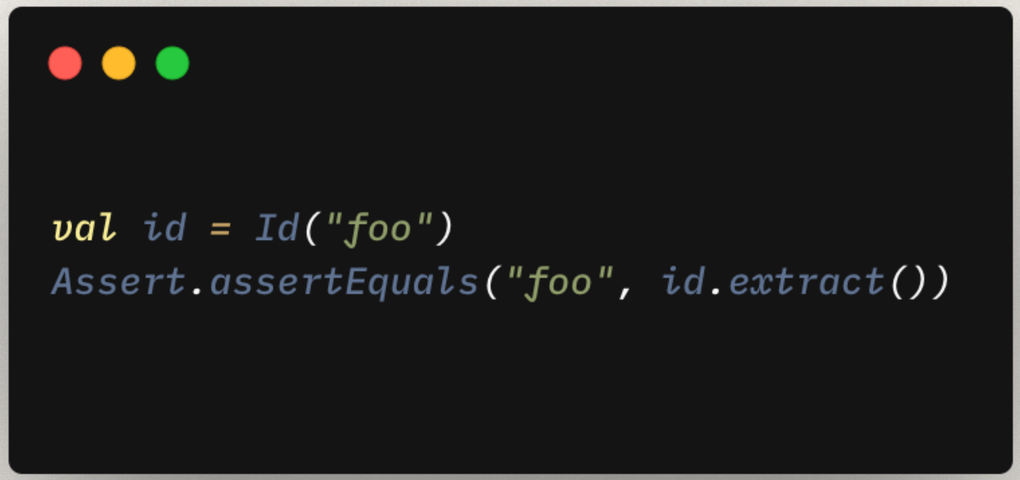
```
apply plugin: 'kotlin-kapt'

def arrow_version = "0.10.3"
dependencies {
    implementation "io.arrow-kt:arrow-core:$arrow_version"
    implementation "io.arrow-kt:arrow-syntax:$arrow_version"
    kapt    "io.arrow-kt:arrow-meta:$arrow_version"
}
```

“All told, 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"

– Saunders Mac Lane : Categories for the Working Mathematician

MONADAS

Basicamente o seu gato dentro de uma caixa.



```
val id = Id("foo")
Assert.assertEquals("foo", id.extract())
```

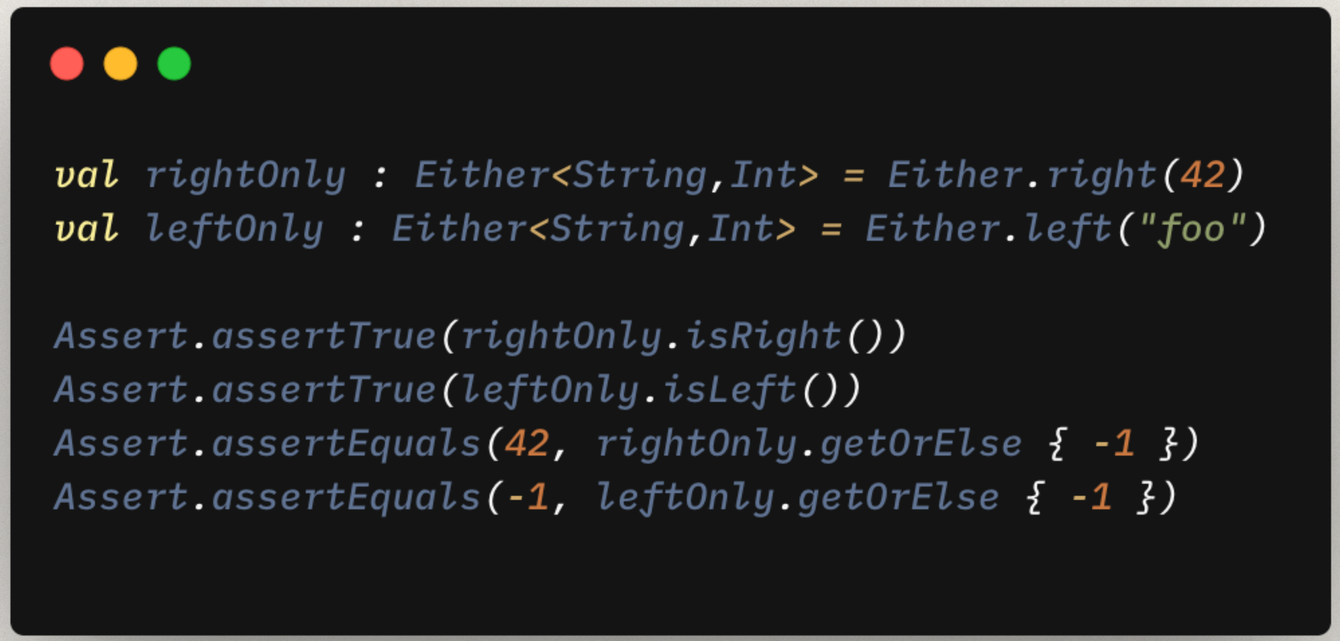
ID

Encapsulamento simples de um dado

```
val factory = Option.just(42)
val constructor = Option(42)
val emptyOptional = Option.empty<Integer>()
val fromNullable = Option.fromNullable(null)

Assert.assertEquals(42, factory.getOrElse { -1 })
Assert.assertEquals(factory, constructor)
Assert.assertEquals(emptyOptional, fromNullable)
```

OPTION

Similar ao existente em JAVA, porém sem ter que programar em JAVA



```
val rightOnly : Either<String,Int> = Either.right(42)
val leftOnly : Either<String,Int> = Either.left("foo")

Assert.assertTrue(rightOnly.isRight())
Assert.assertTrue(leftOnly.isLeft())
Assert.assertEquals(42, rightOnly.getOrElse { -1 })
Assert.assertEquals(-1, leftOnly.getOrElse { -1 })
```

EITHER

É capaz de encapsular dois valores distintos.

Error Handling

Option para casos de valor inválido:

```
fun parseInput(s : String) : Option<Int> = Option.fromNullable(s.toIntOrNull())
```

```
fun computeWithOption(input : String)
: Option<Boolean> {
    return parseInput(input)
      .filter(::isEven)
      .map(::biggestDivisor)
      .map(::isSquareNumber)
}
```

```
fun computeWithOptionClient(input : String) : String {
    val computeOption = computeWithOption(input)
    return when(computeOption) {
        is None -> "Not an even number!"
        is Some -> "The greatest divisor
                    is square number:${computeOption.t}"
    }
}
```

Either para controle de exceções:

```
sealed class ComputeProblem {
    object OddNumber : ComputeProblem()
    object NotANumber : ComputeProblem()
}
```

```
fun parseInput(s : String) : Either<ComputeProblem, Int> =
  Either.cond(s.toIntOrNull() != null,
              { -> s.toInt() },
              { -> ComputeProblem.NotANumber } )
```

```kotlin
fun computeWithEither(input : String) : Either<ComputeProblem, Boolean> {
    return parseInput(input)
      .filterOrElse(::isEven) { -> ComputeProblem.OddNumber }
      .map (::biggestDivisor)
      .map (::isSquareNumber)
}
```

```kotlin
fun computeWithEitherClient(input : String) {
    val computeWithEither = computeWithEither(input)
    when(computeWithEither) {
        is Either.Right -> "The greatest divisor is square number:
                                                ${computeWithEither.b}"
        is Either.Left -> when(computeWithEither.a) {
            is ComputeProblem.NotANumber -> "Wrong input! Not a number!"
            is ComputeProblem.OddNumber -> "It is an odd number!"
        }
    }
}
```

Dependency Injection

Injetando dependencias com Reader:

- myReader.run(context)

```
class MyView : View { // an activity OR fragment
    // ...
    override fun onResume() {
      super.onResume()
      viewModel.getSuperHeroes().run(
            GetHeroesContext(
                view = this,
                getSuperHeroesUseCase = GetSuperHeroesUseCase(),
                heroesRepository = HeroesRepository(),
                dataSources = listOf(MemoryHeroesDataSource())
            )
        )
    }
}
```

Where to go?

- Scala with cats
  - https://books.underscore.io/scala-with-cats/scala-with-cats.pdf
- LIB DOCS
  - https://arrow-kt.io/
- Arrow Meta - Enabling Functional Programming in the Kotlin Compiler
  - https://www.youtube.com/watch?v=WKR384ZeBgk

Obrigado