

THE DEVELOPER'S CONFERENCE




Trilha - Kotlin

Estendendo os poderes de Kotlin: Usando FP com Arrow

Eduardo Castro

Engenheiro de Software

Sobre mim

- Eduardo Castro
- Engenheiro de Software na Dafiti Group
- Áreas de interesse
 - Programação funcional
 - Programação reativa
- @jeduardocosta   



Agenda



- Falando um pouco sobre programação funcional
- Apresentando Arrow
- Manipulação de erros com Option, Try e Either
- Validação de campos com Validated
- Atualizando estruturas imutáveis com Optics
- Mapeando efeitos com IO e Arrow Fx
- O que mais?
- Conclusões



THE
DEVELOPER'S
CONFERENCE

Falando um pouco sobre programação funcional

theory naturality traversable effects
monads currying adjunctions calculus
products semigroup evaluation
data associative monad algebraic
trampoline pure prism
monoid try classes bifunctor free
foldable limits state IO reader traverse flatmap
type composition epimorphisms
kleisli lazy natural function optics option transformations
purity partial lenses coproducts representable types
functors bicategories arity either abstraction
applicatives high-order lambda functional
identity map closure

Uma definição



THE
DEVELOPER'S
CONFERENCE



John A De Goes
@jdegoes

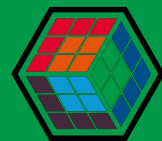


FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

4:32 PM · Nov 30, 2017 · [Twitter Web Client](#)



THE
DEVELOPER'S
CONFERENCE

Apresentando Arrow

<> Code

! Issues 165

🔗 Pull requests 30

🛡 Security

📊 Insights

Functional companion to Kotlin's Standard Library <http://arrow-kt.io>

maven central 0.9.0

build passing

kotlin 1.3

License Apache 2.0

📊 codecov 43%

📦 arrow-kt

Arrow is a library for Typed Functional Programming in Kotlin.

Arrow aims to provide a *lingua franca* of interfaces and abstractions across Kotlin libraries. For this, it includes the most popular data types, type classes and abstractions such as `Option`, `Try`, `Either`, `IO`, `Functor`, `Applicative`, `Monad` to empower users to write pure FP apps and libraries built atop higher order abstractions.

Engineering the Future of Enterprise

47 Degrees helps you take the next step in **modernizing** your legacy applications to be **real-time, secure,** and **responsive,** by using battle-tested **functional programming** and cutting-edge technologies.

[EXPERTISE](#)[SERVICES](#)

Proud member of the Scala Center's advisory board

47 Degrees, along with members from IBM, Verizon, Goldman-Sachs, Nitro, and Lightbend, happily contributes to advancements in the community with a seat on the Scala Center's advisory board.

[Read more >](#)

Arrow.kt



TYPECLASSES

Functor, Applicative, Monad and all their friends so you can take advantage of ad-hoc polymorphism in Kotlin



DATA TYPES

Option, Try, Either, Eval, NonEmptyList and many other data types based on algebraic properties



INTEGRATIONS

Integrations with Rx2 and other popular frameworks and data types in the Kotlin ecosystem



THE
DEVELOPER'S
CONFERENCE

Manipulação de erros com Option, Try e Either

Option



THE
DEVELOPER'S
CONFERENCE

```
val someValue: Option<String> = Some("I am wrapped in something")  
someValue // Some(I am wrapped in something)
```

```
val emptyValue: Option<String> = None  
emptyValue // None
```

```
val value1 = maybeItWillReturnSomething(true)  
val value2 = maybeItWillReturnSomething(false)
```

```
value1.getOrElse { "No value" } // Found value  
value2.getOrElse { "No value" } // No value
```

Option



THE
DEVELOPER'S
CONFERENCE

```
val myString: String? = "Nullable string"  
val option: Option<String> = Option.fromNullable(myString)
```

```
val nullableValue: String? = "Hello"  
nullableValue.toOption() // Some(Hello)
```

```
val someValue: Option<Double> = Some(20.0)  
val value = when(someValue) {  
    is Some -> someValue.t  
    is None -> 0.0  
}  
value // 20.0
```

Option



THE
DEVELOPER'S
CONFERENCE

```
val number: Option<Int> = Some(3)
```

```
val noNumber: Option<Int> = None
```

```
val mappedResult1 = number.map { it * 1.5 } // Some(4.5)
```

```
val mappedResult2 = noNumber.map { it * 1.5 } // None
```

```
number.fold({ 1 }, { it * 3 }) // 9
```

```
noNumber.fold({ 1 }, { it * 3 }) // 1
```

Option



THE
DEVELOPER'S
CONFERENCE

```
import arrow.core.extensions.option.monad.binding
```

```
binding {  
    val (a) = Some(1)  
    val (b) = Some(1 + a)  
    val (c) = Some(1 + b)  
    a + b + c  
}  
// Some(6)
```

Option



```
import arrow.core.extensions.option.monad.binding
```

```
binding {  
    val (x) = Some(1)  
    val (y) = none<Int>()  
    val (z) = Some(1 + y)  
    x + y + z  
}  
// None
```


Try



THE
DEVELOPER'S
CONFERENCE

```
open class GeneralException: Exception()  
class NoConnectionException: GeneralException()  
class AuthorizationException: GeneralException()
```

```
fun checkPermissions() {  
    throw AuthorizationException()  
}
```

```
fun getLotteryNumbersFromCloud(): List<String> {  
    throw NoConnectionException()  
}
```

```
fun getLotteryNumbers(): List<String> {  
    checkPermissions()  
    return getLotteryNumbersFromCloud()  
}
```

Try



THE
DEVELOPER'S
CONFERENCE

```
try {  
    getLotteryNumbers()  
} catch (e: NoConnectionException) {  
    //...  
} catch (e: AuthorizationException) {  
    //...  
}
```

Try



THE
DEVELOPER'S
CONFERENCE

```
try {  
  getLotteryNumbers()  
} catch (e: NoConnectionException) {  
  //...  
} catch (e: AuthorizationException) {  
  //...  
}
```

```
val lotteryTry = Try { getLotteryNumbers() }  
lotteryTry // Success(value=10)  
lotteryTry // Failure(exception=Line_1$AuthorizationException)
```

Try



THE
DEVELOPER'S
CONFERENCE

```
lotteryTry.recover { exception ->
    emptyList()
}
// Success(value=[])
```

```
Try { getLotteryNumbers(Source.NETWORK) }.recoverWith {
    Try { getLotteryNumbers(Source.CACHE) }
}
```

```
lotteryTry.fold(
    { emptyList<String>() },
    { it.filter { it.toIntOrNull() != null } })
// []
```

Try



THE
DEVELOPER'S
CONFERENCE

```
import arrow.core.extensions.`try`.monad.binding
```

```
binding {  
  val (a) = Try { "3".toInt() }  
  val (b) = Try { "4".toInt() }  
  val (c) = Try { "5".toInt() }  
  a + b + c  
}  
  
// Success(value=12)
```

Try



```
import arrow.core.extensions.`try`.monad.binding
```

```
binding {  
    val (a) = Try { "10".toInt() }  
    val (b) = Try { "none".toInt() }  
    val (c) = Try { "5".toInt() }  
    a + b + c  
}
```

```
// Failure(exception=java.lang.NumberFormatException: For input string: "none")
```

Either



```
val right: Either<String, Int> = Either.Right(5)
right // Right(b=5)
```

```
val left: Either<String, Int> = Either.Left("Something went wrong")
left // Left(a=Something went wrong)
```

```
val right: Either<String, Int> = Either.Right(5)
right.flatMap{Either.Right(it + 1)} // Right(b=6)
```

```
val left: Either<String, Int> = Either.Left("Something went wrong")
left.flatMap{Either.Right(it + 1)} // Left(a=Something went wrong)
```

Either



THE
DEVELOPER'S
CONFERENCE

```
Either.cond(true, { 42 }, { "Error" }) // Right(b=42)
```

```
Either.cond(false, { 42 }, { "Error" }) // Left(a=Error)
```

```
val x = "hello".left()
```

```
x.getOrElse { 7 } // 7
```

```
val x = "hello".left()
```

```
x.getOrElse { "$it world!" } // hello world!
```


Either



```
val x = magic("2")
val value = when(x) {
  is Either.Left -> when (x.a){
    is NumberFormatException -> "Not a number!"
    is IllegalArgumentException -> "Can't take reciprocal of 0!"
    else -> "Unknown error"
  }
  is Either.Right -> "Got reciprocal: ${x.b}"
}
value // Got reciprocal: 0.5
```

Either



THE
DEVELOPER'S
CONFERENCE

```
// Exception Style
```

```
fun parse(s: String): Int =  
    if (s.matches(Regex("-?[0-9]+"))) s.toInt()  
    else throw NumberFormatException("$s is not a valid integer.")  
  
fun reciprocal(i: Int): Double =  
    if (i == 0) throw IllegalArgumentException("Cannot take reciprocal of 0.")  
    else 1.0 / i  
  
fun stringify(d: Double): String = d.toString()
```

Either



THE
DEVELOPER'S
CONFERENCE

```
// Either Style
```

```
fun parse(s: String): Either<NumberFormatException, Int> =  
    if (s.matches(Regex("-?[0-9]+"))) Either.Right(s.toInt())  
    else Either.Left(NumberFormatException("$s is not a valid integer."))  
  
fun reciprocal(i: Int): Either<IllegalArgumentException, Double> =  
    if (i == 0) Either.Left(IllegalArgumentException("Cannot take reciprocal of  
0."))  
    else Either.Right(1.0 / i)  
  
fun stringify(d: Double): String = d.toString()
```

Either



// Either Style

```
fun magic(s: String): Either<Exception, String> =  
    parse(s).flatMap{reciprocal(it)}.map{stringify(it)}
```

Either



THE
DEVELOPER'S
CONFERENCE

```
import arrow.core.extensions.either.monad.*
```

```
binding {  
    val (a) = Either.Right(1)  
    val (b) = Either.Right(1 + a)  
    val (c) = Either.Right(1 + b)  
    a + b + c  
}  
// Right(3)
```

Either



THE
DEVELOPER'S
CONFERENCE

```
import arrow.core.extensions.either.monad.*
```

```
binding {  
    val (a) = Either.Right(1)  
    val (b) = Either.Left("invalid number")  
    val (c) = Either.Right(1 + b)  
    a + b + c  
}  
// Left("invalid number")
```

Option + Try + Either



THE
DEVELOPER'S
CONFERENCE

```
binding {  
  val (a) = Option(1)  
  val (b) = Try { "2".toInt() }  
  val (c) = Either.Right(3)  
  a + b + c  
}  
  
// Success(value=6)
```

```
val foo = Try { 2 / 0 }  
val bar = foo.toEither()  
val baz = bar.toOption()
```



THE
DEVELOPER'S
CONFERENCE

Validação de campos com Validated

Validated



THE
DEVELOPER'S
CONFERENCE

```
@higherkind sealed class Validated<out E, out A> : ValidatedOf<E, A> {  
    data class Valid<out A>(val a: A) : Validated<Nothing, A>()  
    data class Invalid<out E>(val e: E) : Validated<E, Nothing>()  
}
```

Validated



`Validated<Nel<ValidationError>, String>` for email

`Validated<Nel<ValidationError>, String>` for phone numbers

`Validated<Nel<ValidationError>, Data>` for the whole request

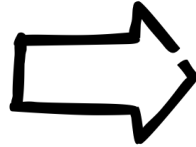
```
sealed class ValidationError {  
    object InvalidMail : ValidationError()  
    object InvalidPhoneNumber : ValidationError()  
}
```

Validated



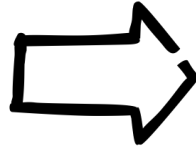
THE
DEVELOPER'S
CONFERENCE

`valid.mail@provider.com`



`Valid(valid.mail@provider.com)`

`invalid.mail`



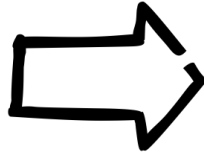
`Invalid([InvalidMail])`

Validated



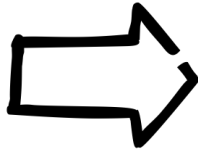
THE
DEVELOPER'S
CONFERENCE

1234567890



Valid(1234567890)

NotAValidNumber



Invalid([InvalidPhoneNumber])

Validated



```
fun String.validatedMail(): Validated<Nel<ValidationError>, String> =  
    when {  
        validMail(this) -> this.valid()  
        else -> ValidationError.InvalidMail.nel().invalid()  
    }
```

```
fun String.validatedPhoneNumber(): Validated<Nel<ValidationError>, String> =  
    when {  
        validNumber(this) -> this.valid()  
        else -> ValidationError.InvalidPhoneNumber.nel().invalid()  
    }
```

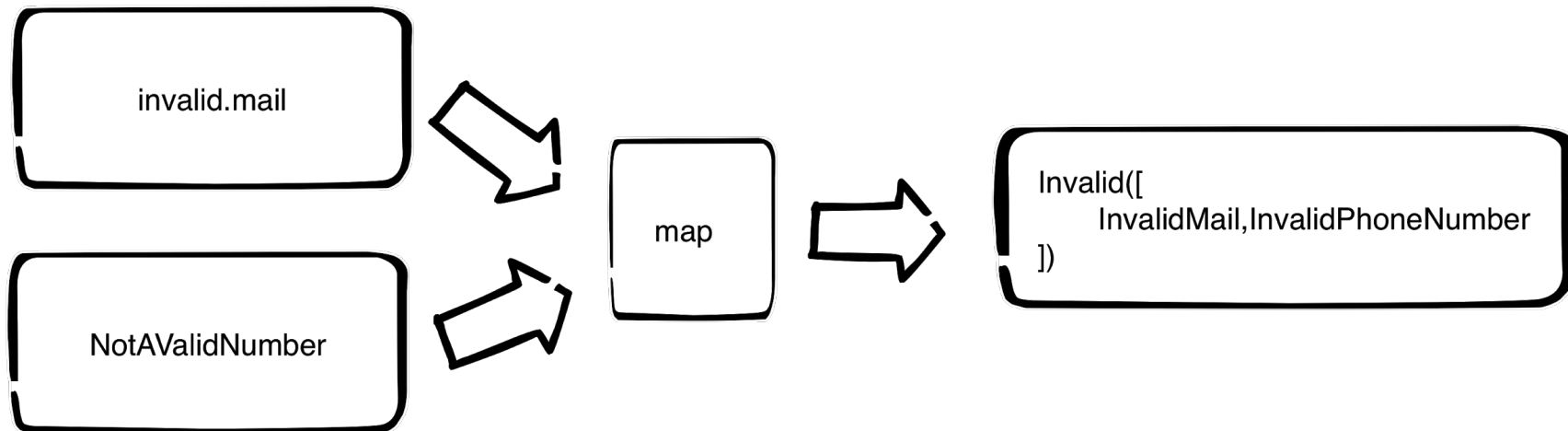
Validated



THE
DEVELOPER'S
CONFERENCE

```
fun validateData(mail: String,  
                phoneNumber: String): Validated<Nel<String>, Data> {  
  
    return Validated.applicative<Nel<ValidationError>>(Nel.semigroup())  
        .map(mail.validatedMail(), phoneNumber.validatedPhoneNumber()) {  
            Data(it.a, it.b)  
        }.fix()  
  
}
```

Validated





THE
DEVELOPER'S
CONFERENCE

Atualizando estruturas imutáveis com Optics

Optics



THE
DEVELOPER'S
CONFERENCE

```
data class Street(val number: Int, val name: String)
data class Address(val city: String, val street: Street)
data class Company(val name: String, val address: Address)
data class Employee(val name: String, val company: Company)

val john = Employee("John Doe",
    Company("Kategory",
        Address("Functional city", Street(42, "lambda street")))))
```

Optics



THE
DEVELOPER'S
CONFERENCE

```
employee.copy(  
  company = employee.company.copy(  
    address = employee.company.address.copy(  
      street = employee.company.address.street.copy(  
        name = employee.company.address.street.name.capitalize()  
      )  
    )  
  )  
)  
)
```

```
// Employee(name=John Doe, company=Company(name=Arrow, address=Address(city=Functional city,  
street=Street(number=23, name=LAMBDA STREET))))
```

Optics

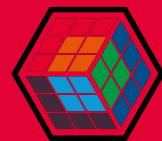


THE
DEVELOPER'S
CONFERENCE

```
@optics data class Street(val number: Int, val name: String)
@optics data class Address(val city: String, val street: Street)
@optics data class Company(val name: String, val address: Address)
@optics data class Employee(val name: String, val company: Company)
```

```
val optional: Optional<Employee, String> =
Employee.company.address.street.name
```

```
optional.modify(john, String::toUpperCase)
// Employee(name=John Doe, company=Company(name=Kategory,
address=Address(city=Functional city, street=Street(number=42, name=LAMBDA
STREET))))
```



THE
DEVELOPER'S
CONFERENCE

Mapeando efeitos com IO e Arrow Fx

Uma definição



THE
DEVELOPER'S
CONFERENCE



John A De Goes
@jdegoes



FP is just programming with functions. Functions are:

1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

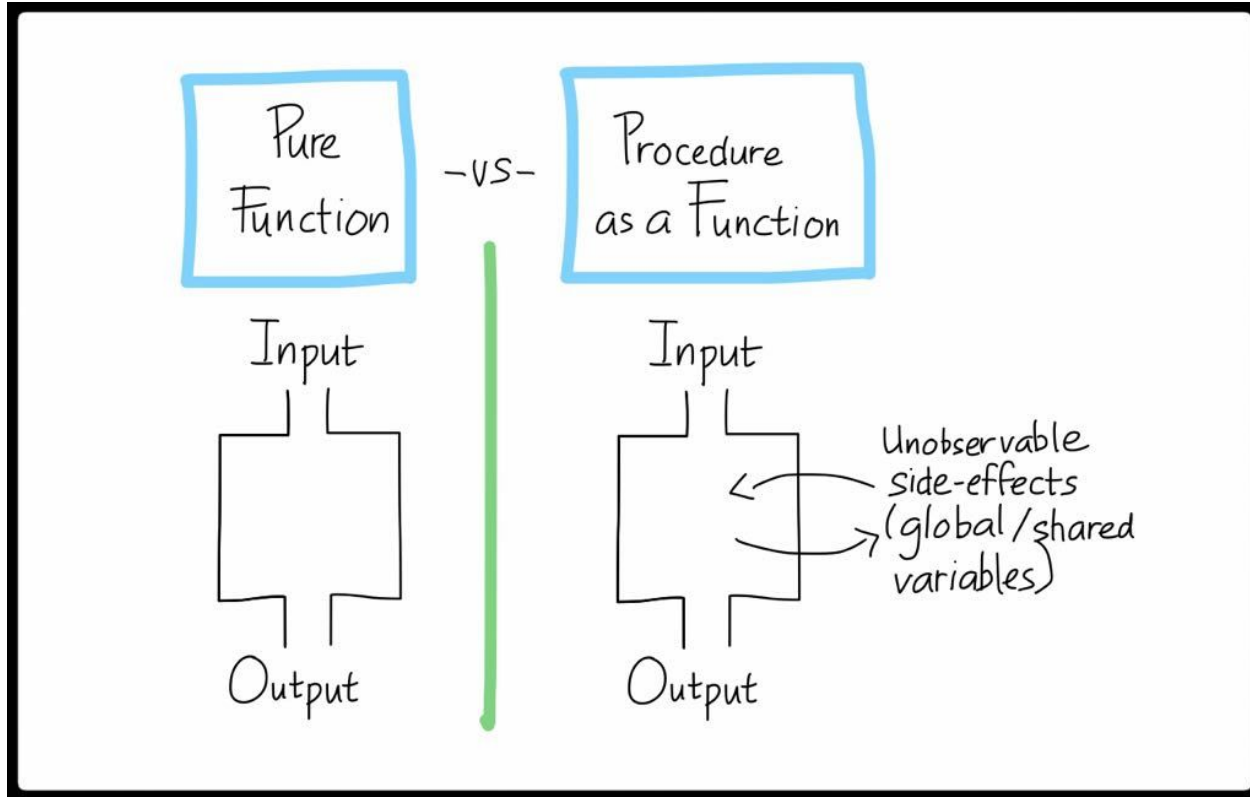
The rest is just composition you can learn over time.

4:32 PM · Nov 30, 2017 · [Twitter Web Client](#)

Side effects



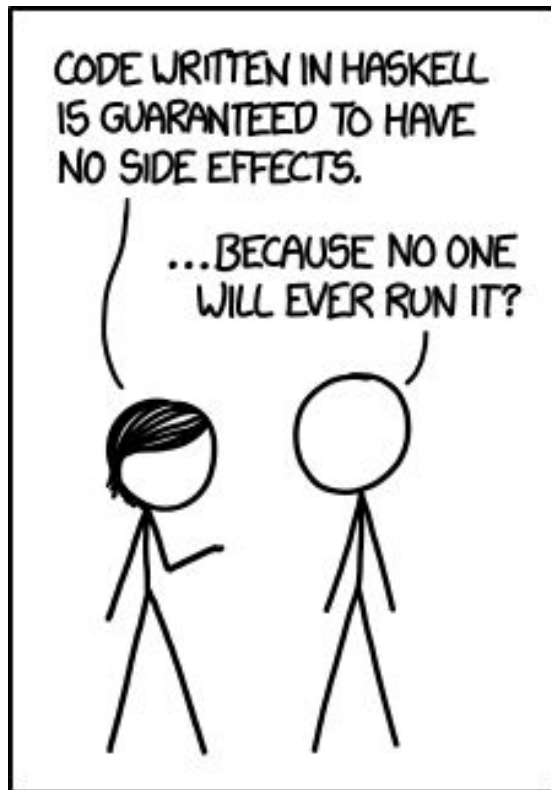
THE
DEVELOPER'S
CONFERENCE



Side effects



THE
DEVELOPER'S
CONFERENCE



IO



THE
DEVELOPER'S
CONFERENCE

```
fun printHelloWorld() = {  
    println("Hello World!")  
}  
  
val computation = printHelloWorld()
```


IO



THE
DEVELOPER'S
CONFERENCE

```
fun printHelloWorld() = {  
    println("Hello World!")  
}  
  
val computation = printHelloWorld()
```

```
fun printHelloWorld() IO<Unit> =  
    IO { println("Hello World!") }
```

```
val result = printHelloWorld()
```

Arrow Fx



THE
DEVELOPER'S
CONFERENCE

```
suspend fun printHello() Unit = println("Hello world")
fun program() = fx {
    effect { printHello() }
}
```

Arrow Fx



THE
DEVELOPER'S
CONFERENCE

```
suspend fun printHello() Unit = println("Hello world")  
fun program() = fx {  
    effect { printHello() }  
}
```

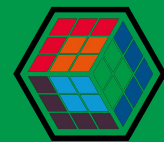
```
fun program() IO<Unit> = fx {  
    !effect { println("Hello World") }  
}  
fun main() { // edge of the world  
    unsafe { runBlocking { program() } }  
}
```

Arrow Fx



THE
DEVELOPER'S
CONFERENCE

```
fx {  
    val res = !NonBlocking.parMapN(  
        effect { Thread.currentThread().name },  
        effect { throw RuntimeException("BOOM!") },  
        effect { Thread.currentThread().name },  
        Tuple3  
    ).handleErrorWith { error: Throwable  
        effect { println("One of the ops failed!") }  
    }  
    !effect { println(res) }  
}
```



THE
DEVELOPER'S
CONFERENCE

O que mais, hein??

O que mais?



- Integrações com Rx2, Reactor, kotlin.coroutines, Retrofit, Kindej
- Muitas type classes (Functor, Applicative, Semigroup, Traverse)
- Free monads
- Recursion schemas
- arrow-mtl (tagless final architecture)
- Free algebras



THE
DEVELOPER'S
CONFERENCE

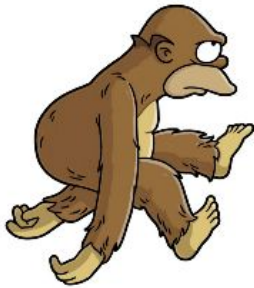
Conclusões



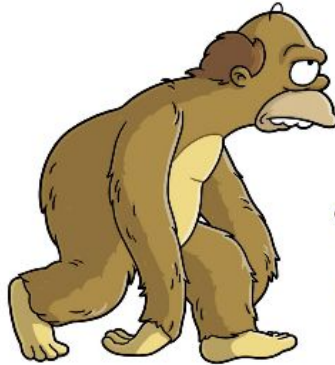
THE
DEVELOPER'S
CONFERENCE



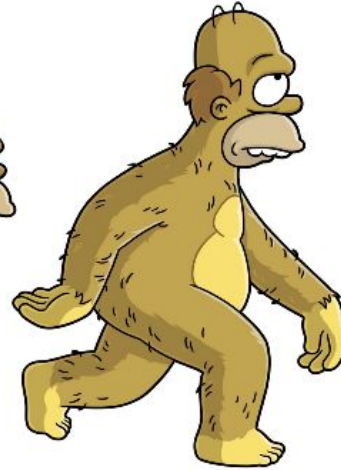
MACHINE



ASSEMBLY



PROCEDURAL



OBJECT ORIENTED



FUNCTIONAL

© 1999 SEGA

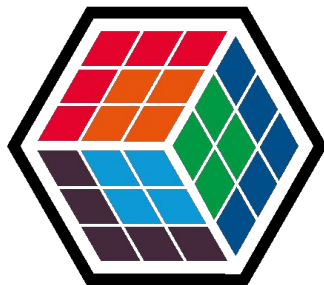


FUNCTIONAL COMPANION TO
KOTLIN'S STANDARD LIBRARY

Referências



- <https://arrow-kt.io/docs>
- <https://twitter.com/jdegoes>
- <https://caster.io/courses/functional-programming-in-kotlin-with-arrow>
- <http://danielecampogiani.com/blog/2018/02/android-functional-validation-4-validated/>
- <https://www.47deg.com/presentations/2019/06/07/arrowfx-fp-for-the-masses-jc/>



THE DEVELOPER'S CONFERENCE