
MATTER
SUPPLY
CO

Programación funcional para No-Batos

MATTER
SUPPLY
CO

Dos pilares de la FP

MATTER
SUPPLY
CO

Tipos y Funciones

MATTER
SUPPLY
CO

Kotlin es multiparadigma

MATTER
SUPPLY
CO

MATTER SUPPLY CO

FP vs POO

MATTER
SUPPLY
CO

Clases vs Tipos

MATTER
SUPPLY
CO

```
val cristian = Persona(nombre="Cristian", apellido="Gomez")
val carlos = Persona(nombre="Carlos", apellido="Gomez")
val carmen = Persona(nombre="Carmen", apellido="Gomez")
val spock = Perro(nombre="Spock")
val familia:Familia = listOf(cristian, carlos, carmen, spock)
```

```
sealed class Integrante
data class Persona(
    val nombre: String, val apellido: String
) : Integrante()
data class Perro(val nombre: String) : Integrante()
typealias Familia = List<Integrante>
```

Categorías

MATTER
SUPPLY
CO

MATTER
SUPPLY
CO

Int → String

String → Int

MATTER
SUPPLY
CO

$$\text{Int} \longrightarrow \text{Int}$$

$$\text{Int} \to \text{Unit}$$

```
fun sum5(a:Int):Int = a + 5
```

```
val sum5: (a: Int) -> Int = { a -> a + 5 }
```

```
val sum5: (a: Int) -> Int = { a -> a + 5 }
```

El lenguaje es importante

MATTER
SUPPLY
CO

Tipos de datos

MATTER
SUPPLY
CO

Algebraicos

MATTER
SUPPLY
CO

```
val a = 2 + 3
val b = 2 * 3
val c = 2 - 3
val d = 2 / 3
val e = 2 % 3
```

Lógicos

MATTER
SUPPLY
CO

```
val a = true
val b = false
val c = a and b
val d = c or b
```

Comunicación

MATTER
SUPPLY
CO

```
val a = "hola"
val b = "mundo"
val c = "$a $b"
```

Compuestos

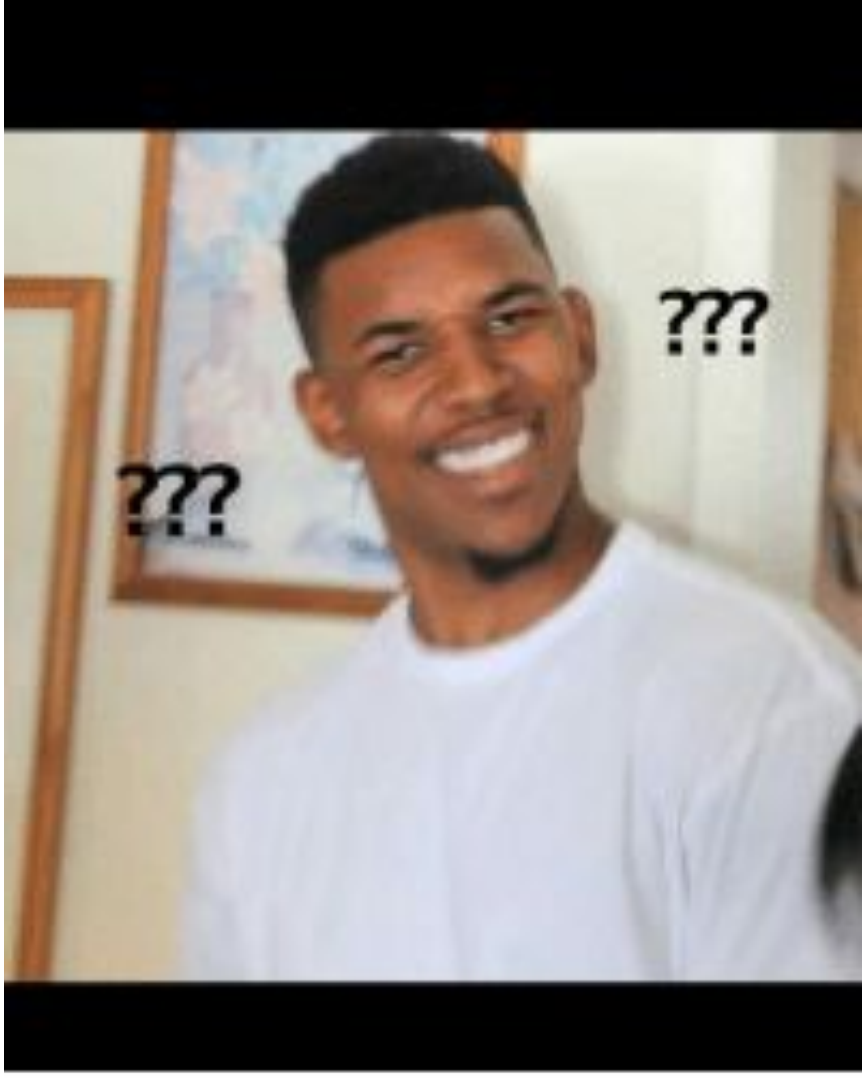
MATTER
SUPPLY
CO

```
sealed class Integrante
data class Persona(
    val nombre: String, val apellido: String
) : Integrante()
```

Opcionales

MATTER
SUPPLY
CO

```
data class Persona(
    val nombre: String?, val apellido: String
)
```



MATTER
SUPPLY
CO

```
data class Persona(
    val nombre: String?, val apellido: String
)
```

Mutabilidad

MATTER
SUPPLY
CO

var/val

MATTER
SUPPLY
CO

Colecciones

MATTER
SUPPLY
CO

```kotlin
val a = arrayOf("hola", "mundo")
val b = intArrayOf(1, 2, 3, 4, 5)
val c = mapOf(
    Pair("Carlos", 33),
    Pair("Cristian", 31),
    Pair("Carmen", 64)
)
val d = setOf(1,1,3,4,5,6,6)
val e = listOf<Integrante>(carlos, cristian, carmen, spock)
```

```
// val a = arrayOf("hola", "mundo") // No hay versión mutable
// val b = intArrayOf(1, 2, 3, 4, 5) // No hay versión mutable
val c = mutableMapOf(
    Pair("Carlos", 33),
    Pair("Cristian", 31),
    Pair("Carmen", 64)
)
val d = mutableSetOf(1,1,3,4,5,6,6)
val e = mutableListOf<Integrante>(carlos, cristian, carmen,
spock)
```

Sum Types/ADT

MATTER
SUPPLY
CO

```
sealed class Tree
object Leaf : Tree()
data class Node(val value: Int, val left: Tree, val right: Tree):
Tree()
val tree = Node(
    5,
    Node(1, Leaf, Leaf),
    Node(3, Leaf,
        Node(4, Leaf, Leaf)
    )
)
```

```
sealed class Tree
object Leaf : Tree()
data class Node(val value: Int, val left: Tree, val right: Tree):
Tree()
val tree = Node(
    5,
    Node(1, Leaf, Leaf),
    Node(3, Leaf,
        Node(4, Leaf, Leaf)
    )
)
```

Funciones

MATTER
SUPPLY
CO

```
val sum: (a: Int, b: Int) -> Int = { a, b -> a + b }
```

Funciones que producen funciones

MATTER
SUPPLY
CO

```
val mayorQueCinco:(Int) -> Boolean = validarMayor(5)
val SeisMayorQue5:Boolean = mayorQueCinco(6)
print(SeisMayorQue5) // true
```

```kotlin
fun validarMayor(base: Int): (Int) -> Boolean = { valor -> valor > base }
```

```
fun validarMayor(base: Int): (Int) -> Boolean = { valor -> valor > base }
```

```
fun validarMayor(base: Int): (Int) -> Boolean = { valor -> valor > base }
```

```
fun validarMayor(base: Int): (Int) -> Boolean = { valor -> valor > base }
```

MATTER
SUPPLY
CO

Resultado:

Int → (Int → Boolean)

MATTER
SUPPLY
CO

Currificación (?)

MATTER
SUPPLY
CO

Extensiones

MATTER
SUPPLY
CO

```
val DiezMayorQueCinco = 10.mayorQue(5)
fun Int.mayorQue(valor: Int):Boolean = this > valor
```

```
val DiezMayorQueCinco = 10 mayorQue 5
infix fun Int.mayorQue(valor: Int):Boolean = this > valor
```

Composición

MATTER
SUPPLY
CO

Encadenamiento (chaining)

MATTER
SUPPLY
CO

f(x) compuesto g(x)

MATTER
SUPPLY
CO

$$g(f(x))$$

MATTER
SUPPLY
CO

f >== g

f `then` g

teniendo X,
aplico f y despues g

MATTER
SUPPLY
CO

Functors

MATTER
SUPPLY
CO

map

MATTER
SUPPLY
CO

```
listOf(1).map { it + 5 } // => [6]
```

```
listOf(1)
    .map { it + 5 }
    .map { it / 5 }
    .map { it.toDouble() }
    .map { floor(it) }
    .map { "$it" }
```

MATTER SUPPLY CO

morphims

MATTER
SUPPLY
CO

MATTER
SUPPLY
CO

A → D

MATTER
SUPPLY
CO

Run/Let

MATTER
SUPPLY
CO

this/it

MATTER
SUPPLY
CO

```
1    .let { it + 5 }
     .let { it / 5 }
     .let { it.toDouble() }
     .let { floor(it) }
     .let { "$it" }
```

Todos los tipos
Son “functors”

MATTER
SUPPLY
CO

Ejemplo

MATTER
SUPPLY
CO

```
val carrito = Carrito(
    listOf(
        Producto.PorCantidad("Leche litro", 1, 4000.0),
        Producto.PorCantidad("Huevos", 1, 4000.0),
        Producto.PorPeso("Tomates por kilo", 2.0, 2000.0)
    )
)
val resultado = carrito
    .run { sumarTodos() }
    .run { agregarImpuestos() } // => 14160.0
```

```
val carrito = Carrito(
    listOf(
        Producto.PorCantidad("Leche litro", 1, 4000.0),
        Producto.PorCantidad("Huevos", 1, 4000.0),
        Producto.PorPeso("Tomates por kilo", 2.0, 2000.0)
    )
)
val resultado = carrito
    .run { sumarTodos() }
    .run { agregarImpuestos() } // => 14160.0
```

Resultado:
Carrito → Double

MATTER
SUPPLY
CO

Efectos secundarios

MATTER
SUPPLY
CO

Funciones puras e impuras

MATTER SUPPLY CO

Cajas

MATTER
SUPPLY
CO

```
val some = Box.of(3)
    .flatMap { Box.of(it + 1)
        .flatMap { Box.of(it + 5) }
    }
    .flatMap { Box.of(it + 2) }
println(some) // Box(11)
```

```kotlin
class Box<out A>(val value:A) {

    companion object {
        fun <A> of(value: A): Box<A> = Box(value)
    }

    //Wrapped f(value)
    fun <B> map(f: (A) -> B): Box<B> = Box(f(value))

    //'Raw' (value)
    fun <B> flatMap(f: (A) -> Box<B>): Box<B> = f(value)
}
```

```
class Box<out A>(val value:A) {

    companion object {
        fun <A> of(value: A): Box<A> = Box(value)
    }

    //Wrapped f(value)
    fun <B> map(f: (A) -> B): Box<B> = Box(f(value))

    //'Raw' (value)
    fun <B> flatMap(f: (A) -> Box<B>): Box<B> = f(value)
}
```

MATTER SUPPLY CO

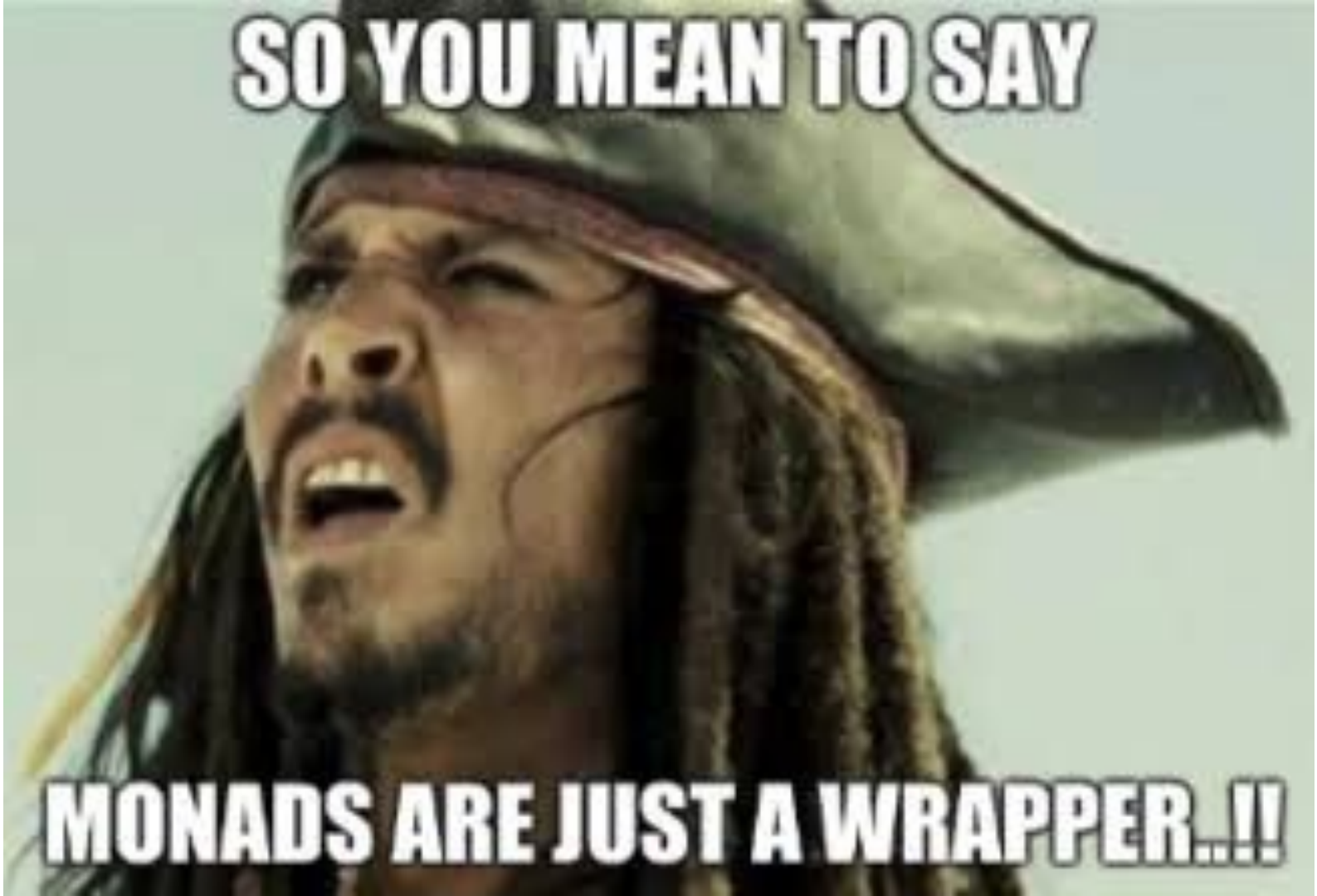
```
class Box<out A>(val value:A) {

    companion object {
        fun <A> of(value: A): Box<A> = Box(value)
    }

    //Wrapped f(value)
    fun <B> map(f: (A) -> B): Box<B> = Box(f(value))

    //'Raw' (value)
    fun <B> flatMap(f: (A) -> Box<B>): Box<B> = f(value)
}
```

```
fun <B> flatMap(f: (A) -> Box<B>): Box<B>
```

```
fun <B> flatMap(f: (A) -> Monad<B>): Monad<B>
```

Ahora tenemos Monadas

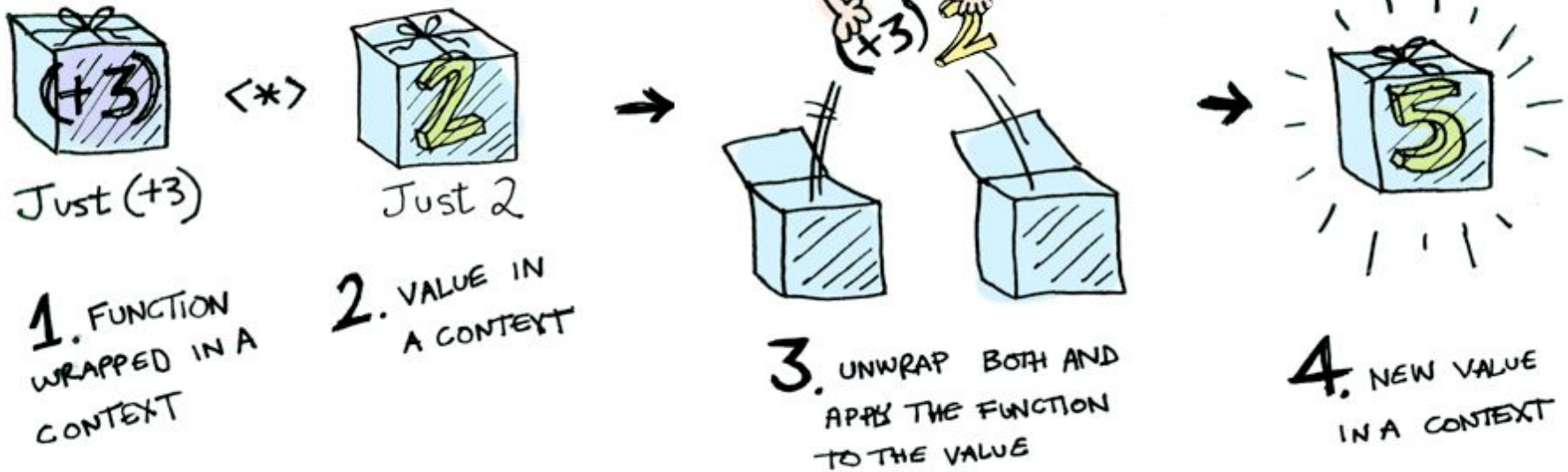
MATTER
SUPPLY
CO


SO YOU MEAN TO SAY
MONADS ARE JUST A WRAPPER..!!

Promises y Observables son Monadas

MATTER
SUPPLY
CO

```
fun <A, B> Box<(A) -> B>.apply(f: Box<A>): Box<B> = f.map(this.value)
```


(+3)
<*>
2
Just (+3)
Just 2
1. FUNCTION WRAPPED IN A CONTEXT
2. VALUE IN A CONTEXT
(+3)
2
3. UNWRAP BOTH AND APPLY THE FUNCTION TO THE VALUE
5
4. NEW VALUE IN A CONTEXT

```
val sum: (Int) -> (Int) -> Int = {
    first:Int -> {
        second: Int -> first + second
    }
}
val applicative = Box.of(sum).apply(Box.of(2)).apply(Box.of(3))
println(applicative) // Box(5)
```

```
val sum: (Int) -> (Int) -> Int = {
    first:Int -> {
        second: Int -> first + second
    }
}
val applicative = Box.of(sum).apply(Box.of(2)).apply(Box.of(3))
println(applicative) // Box(5)
```

cristianfgr@gmail.com
cristian.gomez@mattersupply.com
@iyubinest