QCon SÃO PAULO

Zup

Kotlin

Augusto Branquinho
Mike Shigueru Matsumoto

2. Kotlin

- Designed by Jetbrains
- Statically typed
- Modern
- Drastically reduce the boilerplate
- Safe null pointer
- Java, Android, and Javascript
- Native
- Kotlin in production - Google, Netflix, Uber, Coursera, Basecamp, Nubank, OLX, Stone, Colab

Topic1.kt

```kotlin
package br.com.zup.kotlin

fun main(args: Array<String>) {
    println( "hello world")
}

fun sum(x: Int, y: Int): Int {
    return x + y
}
```

?

Topic1.kt

```
package br.com.zup.kotlin


class Car(val name: String, val year: Int) {
    fun something() {}
}

class Plane(var name: String, var year: Int)
```

```kotlin
fun returnNothing(a: Int): Unit {
}

fun returnNothing(a: Int) {
}
```

```kotlin
fun sum(x: Int, y: Int): Int {
    return x + y
}

fun sum(x: Int, y: Int) = x + y
```

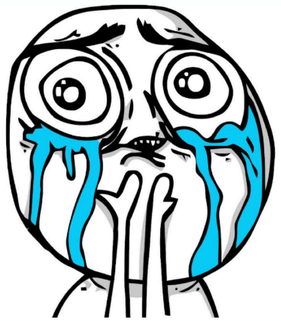
```
val s1: String = "str1"

val s2 = "str2"

    s2 = "str2"
```

Val cannot be reassigned

```
var i = 1

i = 1 + 2
```

```
val s3: String? = null

s3.toUpperCase()
```

Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?



```
val s3: String? = null

s3.toUpperCase()
```

wed on a nullable receiver of type String?

```
if (s3 ≠ null) {
    s3.toUpperCase()
}
```

```
s3?.toUpperCase()
```

```
val s3: String? = null

s3?.toUpperCase() ?: "elvisoperator"
```

😡

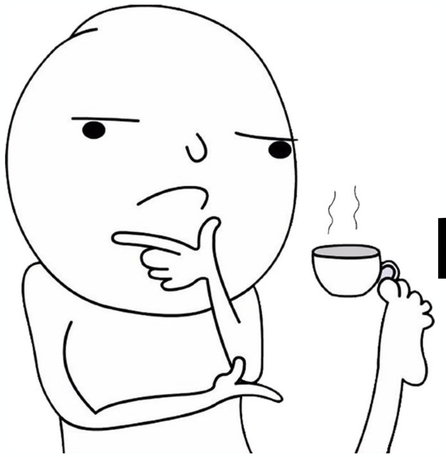
```
s3!!.toUpperCase()
```

```
fun foobar1(a: Int,
            date: LocalDateTime = LocalDateTime.now(),
            s: String? = null) { ... }
```

```
fun foobar1(a: Int,
            date: LocalDateTime = LocalDateTime.now(),
            s: String? = null) { ... }
```

```
foobar1( a: 1)
foobar1( a: 1, now().plusDays( days: 1))
foobar1( a: 1, now().plusDays( days: 1), s: "myString")

foobar1(a = 1, s = "myString")
```

```kotlin
class Foobar(a: Int = 100)
fun foobar1(a: Int,
            date: LocalDateTime = LocalDateTime.now(),
            s: String? = null) { ... }


foobar1( a: 1)
foobar1( a: 1, now().plusDays( days: 1))
foobar1( a: 1, now().plusDays( days: 1), s: "myString")

foobar1(a = 1, s = "myString")
```



Puzzle: How many overloads methods in Java?

```
fun foobar1(a: Int,
            date: LocalDateTime = LocalDateTime.now(),
            s: String? = null) { ... }

foobar1(a = 1, s = "myString")
```

```kotlin
val numbers: IntRange = (1..10)

for (i: Int in numbers) {
    print("i=$i")
}

for (i: Int in 1..100) {
    print("i=$i")
}
```

```
val s1 = "foobar"

println ("s1 = $s1")
println ("s1 = ${s1.toUpperCase()}")

val s2: String = """
                 |asdf
                 |asdf
                 |asdf
""".trimMargin()
```

```
data class UserCreatedEvent(val id: UserId,
                            val name: UserName)
```

```
data class UserCreatedEvent(val id: UserId,
                            val name: UserName)

val event1 = UserCreatedEvent(
    UserId( id: 1), UserName( name: "Alan Turing"))
event1.hashCode()
event1.toString()
event1.equals(null)
event1 == event2      event1 === event2
event1.name
```

```kotlin
data class UserCreatedEvent(val id: UserId,
                            val name: UserName)

val event1 = UserCreatedEvent(
    UserId( id: 1), UserName( name: "Alan Turing"))
event1.hashCode()
event1.toString()
event1.equals(null)
event1 == event2
event1.name
```

event1 == event2

```kotlin
data class UserCreatedEvent(val id: UserId,
                            val name: UserName)
val event1 = UserCreatedEvent(
    UserId( id: 1), UserName( name: "Alan Turing"))
event1.hashCode()
event1.toString()
event1.equals(null)
event1 == event2
event1.name
val event3 : UserCreatedEvent = event1.copy(
        name = UserName( name: "Turing Alan"))
```

```kotlin
val user = User(
        name: "user1", City( name: "cityName1"))

val (userName : String , city : City ) = user
val (userName2 : String , _ : City ) = user


for ((name : String , _ : City ) in users) {
    println("name ${name.toUpperCase()}")
}
```

```kotlin
data class Pair<out A, out B>
```

```kotlin
data class Triple<out A, out B, out C>
```

```kotlin
fun foobar(a: Any) {

    if (a is String) {
        // ((String)a).toUpperCase()
        a.toUpperCase()
    }

    if (a is Int) {
        a.toLong()
    }
}
```

```kotlin
val b: String = a as String
b.toUpperCase()
```

```
fun foobar(a: Int): Int {
    val number:Int =  if (a == 1) {
        a * 2
    } else {
        -1
    }
    return number
}
```

```
fun foobar(a: Int): Int {
    val number:Int =  if (a == 1) {
        a * 2
    } else {
    }
    re
}
```

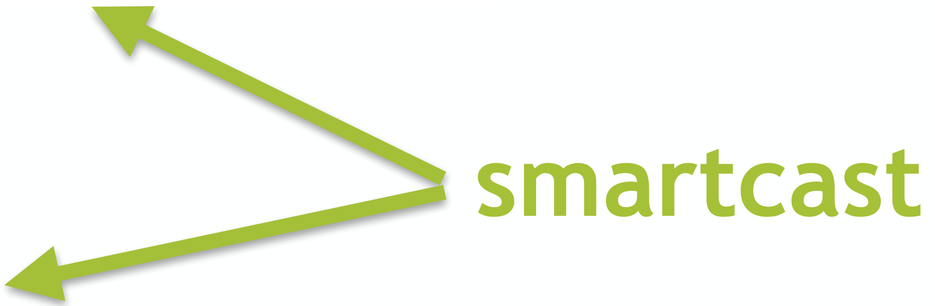
```
fun foobarOne(a: Int) = if (a == 1) {
        a * 2
} else {
        -1
}
```

```
fun foobarTwo(a: Int) = try {
    a * 2
} catch (e: Exception) {
    -1
}
```

```
fun fo                    {
    a
} else {
    -1
}
```

```kotlin
when (any) {
    is String → any.toUpperCase()
    is Int → {
        println("type Int")
        any.toBigDecimal()
    }
    1..10 → println("range 1..10")
    "aaa" → println("aaa")
    else → println("I quit!")
}
```

```
when (any) {
    is String → any.toUpperCase()
    is Int → {
        println("type Int")
        any.toBigDecimal()
    }
    1..10 → println("range 1..10")
    "aaa" → println("aaa")
    else → println("I quit!")
}
```



```
when (maybeNull) {
    null → println("null value")
    else → println("not null value")
}
```

```
when (maybeNull) {
    null → println("null value")
    else → println("not null value")
}
```

```
when {
    maybeNull ≠ null → {
        println("not null")
    }
    else → println("not null value")
}
```

```
when (maybeNull) {
    null → println("null value")
    else → println("not null value")
}
```

```
when {
    maybeNull ≠ null → {
        println("not null")
    }
    else → println("not null value")
}
```

```
when (maybeNull) {
    null → println("null value")
    else → println("not null value")
}
```

```
when {
    maybeNull ≠ null → {
        println("not null")
    }
    else → println("not null value")
}
```

Also can be returned or be assigned to a variable

```
fun failsALot(a: Int): Int = try {
    a * 2
} catch (e: Exception) {
    fail(e)
}

private fun fail(e: Exception) {
    throw RuntimeException(e)
}
```

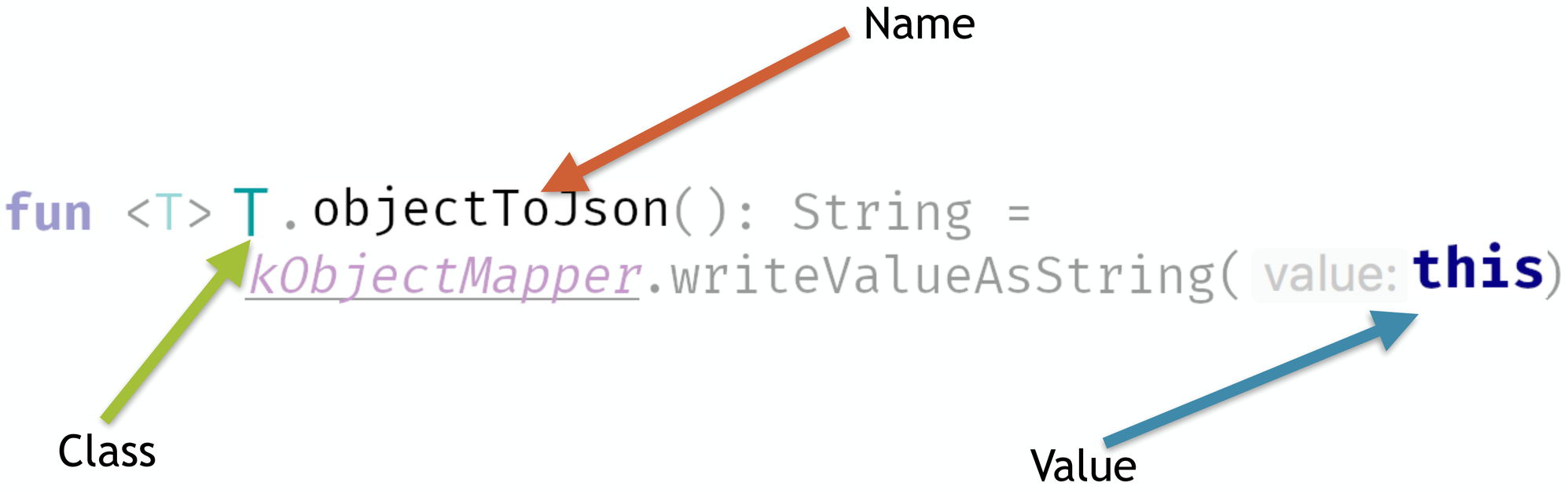
Type mismatch.
Required: Int
Found: Unit

```
fun failsALot(a: Int): Int = try {
    a * 2
} catch (e: Exception) {
    fail(e)
}

private fun fail(e: Exception): Nothing {
    throw RuntimeException(e)
}
```

2. Kotlin

```
fun <T> T.objectToJson(): String =
        kObjectMapper.writeValueAsString( value: this)
```

2. Kotlin




```
data class User(val name: String, val age: Int)

val json: String = user.objectToJson()
```

```kotlin
class Amount(val value: Long) {
    operator fun plus(other: Amount): Amount { ... }
    operator fun inc(): Amount { ... }
    operator fun not() { ... }
    operator fun div(other: Amount) { ... }
}
```

2. Kotlin

```
class Amount(val value: Long) {
    operator fun plus(other: Amount): Amount { ... }
    operator fun inc(): Amount { ... }
    operator fun not() { ... }
    operator fun div(other: Amount) { ... }
}
```

```
var amount100 = Amount( value: 100)
var amount200 = Amount( value: 200)

val amount300: Amount = amount100 + amount200
amount100++
!amount100
amount200 / amount100
```

```kotlin
val some: (x: Int, y: Int) → Int =
        { x:Int, y:Int → x * y }

val doubleIt: (x: Int) → Int =
        { it * it }

fun currying(): (Int) → (Int) → Int =
        {x:Int → { y:Int → x + y } }

fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)
```

```kotlin
val some: (x: Int, y: Int) → Int =
        { x:Int, y:Int → x * y }

val doubleIt: (x: Int) → Int =
        { it * it }

fun currying(): (Int) → (Int) → Int =
        {x:Int → { y:Int → x + y } }

fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)
```

```kotlin
val some: (x: Int, y: Int) → Int =
        { x:Int, y:Int → x * y }

val doubleIt: (x: Int) → Int =
        { it * it }

fun currying(): (Int) → (Int) → Int =
        {x:Int → { y:Int → x + y } }

fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)
```

```
val some: (x: Int, y: Int) → Int =
        { x: Int, y: Int → x * y }

val doubleIt: (x: Int) → Int =
        { it * it }

fun currying(): (Int) → (Int) → Int =
        {x: Int → { y: Int → x + y } }

fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)
```

```kotlin
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)
```

```
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)

highOrder( age: 10, { age : Int → age * 2 })
```

2. Kotlin

```
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)

highOrder( age: 10, { age: Int → age * 2 })
highOrder( age: 10, { it * 2 })
```

2. Kotlin

```
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)

highOrder( age: 10, { age: Int → age * 2 })
highOrder( age: 10, { it * 2 })

highOrder( age: 10) { it: Int
    it * 2
}
```

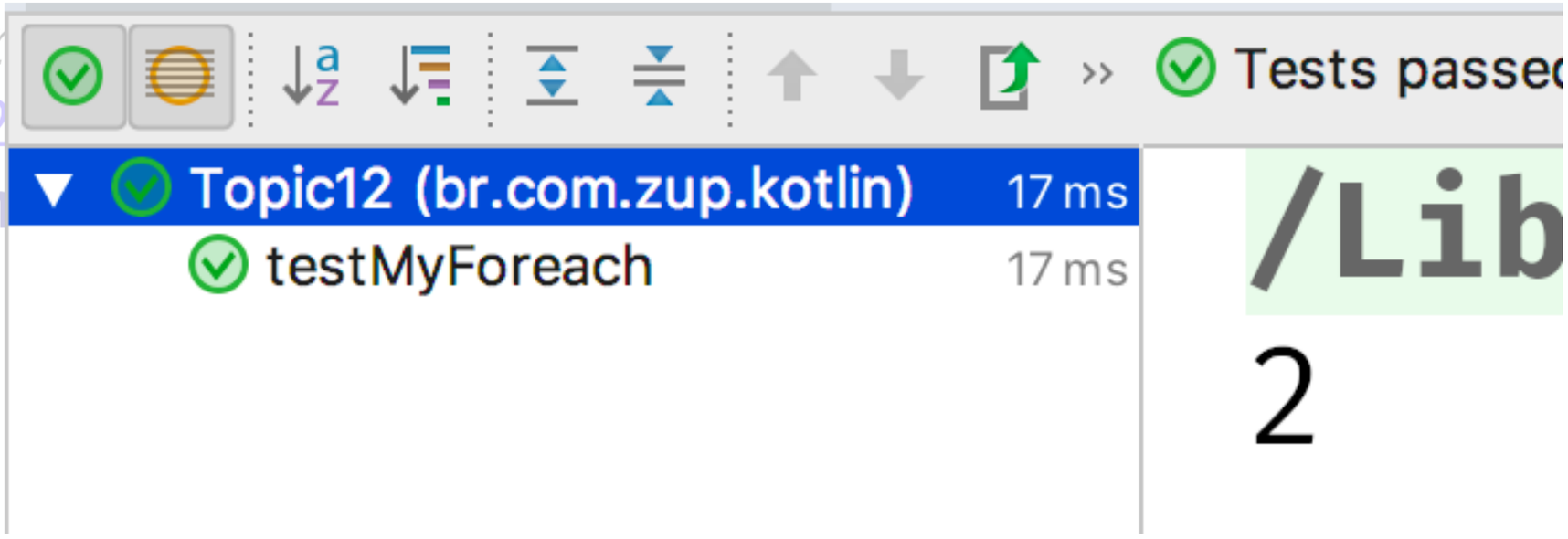
```kotlin
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)

highOrder( age: 10, { age : Int → age * 2 })
highOrder( age: 10, { it * 2 })

highOrder( age: 10) { it: Int
    it * 2
}

highOrder { it * 2 }
```

```kotlin
fun highOrder(age: Int = 0,
              op: (age: Int) → Int): Int = op(age)

highOrder( age: 10, { age : Int → age * 2 })
highOrder( age: 10, { it * 2 })

highOrder( age: 10) { it: Int
    it * 2
}

highOrder { it * 2 }
highOrder { _ : Int → 3 }
```

```
inline fun <T> Iterable<T>.myForeach(f: (T) → Unit) {
    for (e:T in this) f(e)
}

fun call(): Int {
    (1..100).myForeach {it: Int
        if (it % 2 == 0)
            return it
    }
    return -1
}
```

```kotlin
inline fun <T> Iterable<T>.myForeach(f: (T) → Unit) {
    for (e:T in this) f(e)
}

fun call(): Int {
    (1..100).myFor
        if (it % 2
            return
    }
    return -1
}
```

??



```kotlin
val a: List<Int> = listOf(1, 2, 3) // emptyList<Int>()
                                   // setOf<Int>()
                                   // emptySet<Int>()
                                   // mapOf<Int, String>()
                                   // emptyMap<Int, String>()
val c: List<Int> = a + listOf(4, 5)
val d: MutableList<Int> = a.toMutableList() // d.toList()
d.add(1111)

mapOf(1 to "aaa",
      2 to "bbb",
      3 to "ccc")
```

```
val a:List<Int> = listOf(1, 2, 3) // emptyList<Int>()
                                  // setOf<Int>()
                                  // emptySet<Int>()
                                  // mapOf<Int, String>()
```

```
public infix fun <A, B> A.to(that: B): Pair<A, B>
        = Pair(this, that)
```

```
val d:MutableList<Int> = a.toMutableList() // d.toList()
d.add(1111)
```

```
mapOf(1 to "aaa",
      2 to "bbb",
      3 to "ccc")
```

```
IntStream.rangeClosed(1, 100)
        .filter((i) → i % 2 == 0)
        .map((even) → even * 2)
        .boxed()
        .collect(Collectors.toList());
```

2. Kotlin

```
IntStream.rangeClosed(1, 100)
         .filter((i) → i % 2 == 0)
         .map((even) → even * 2)
         .boxed()
         .collect(Collectors.toList());
```



```
(1..100).filter { it % 2 == 0 }
        .map { it * 2 }
```

2. Kotlin

```
numbers.filterIndexed { ... } //.mapIndexed { }
numbers.first { ... }
numbers.find { ... } ?: println("null")
numbers.none { ... } //any { }
numbers.sum() // sumBy { }
numbers.min() // numbers.maxBy {}
numbers.reduce { ... }
numbers.fold( initial: 0, { ... }) // foldRight
numbers.take( n: 1)
numbers.drop( n: 10)
numbers.toList().slice( indices: 0..10)
numbers.chunked( size: 10) { ... }
numbers.zip(listOf(4, 5, 6))
numbers.partition { ... }
numbers.groupBy { ... } // List to Map
mapOf(1 to "a").toList() // Map To List<Pair>
```

2. Kotlin

```
(1..10).map { it: Int
    println("eager evaluation $it")
    ^map it * 2
}
//.forEach { println("result: $it") }
```

Tests passed: 1 of 1 test – 7 ms

```
eager evaluation 7
eager evaluation 8
eager evaluation 9
eager evaluation 10
```

2. Kotlin

```
(1..10).map { it: Int
    println("eager evaluation $it")
    ^map it * 2
}
//.forEach { println("result: $it") }
```

```
Tests passed: 1 of 1 test – 7 ms
eager evaluation 7
eager evaluation 8
eager evaluation 9
eager evaluation 10
```

```
(1..10).asSequence()
        .filter { it: Int
            println("lazy evaluation $it")
            ^filter it % 2 == 0
        }
       //.forEach { println("only even: $it") }
```

```
Tests passed: 1 of 1 test – 35 ms
/Library/Java/JavaVirtualMachir

Process finished with exit code
```

2. Kotlin

```
class MyService(val myRepository: MyRepository):
        MyRepository by myRepository {

    val b: String by lazy { "b lazy init" }

    fun doSomething() {
        save() // myRepository.save()
    }

    companion object Singleton {
        const val PI: Double = 3.14
        fun foobar() {}
    }
}
```

2. Kotlin

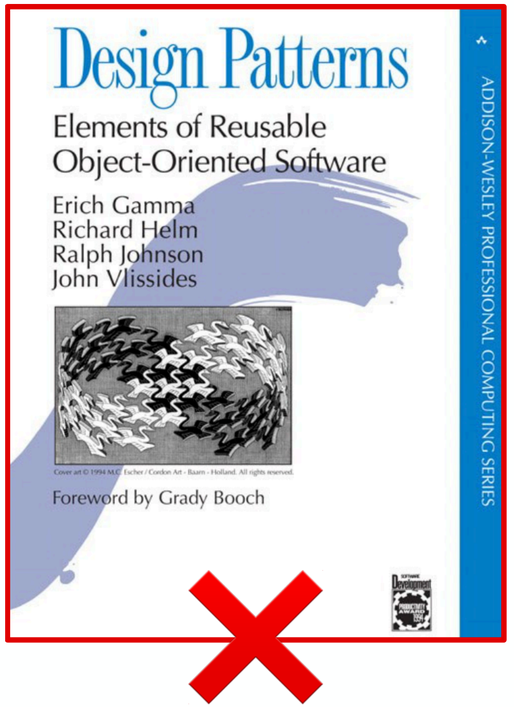
```
class MyService(val myRepository: MyRepository):
        MyRepository by myRepository {

    val b: String by lazy { "b lazy init" }

    fun doSomething() {
        save() // myRepository.save()
    }

    companion object Singleton {
        const val PI: Double = 3.14
        fun foobar() {}
    }
}
```

2. Kotlin

```
class MyService(val myRepository: MyRepository):
        MyRepository by myRepository {

    val b: String by lazy { "b lazy init" }

    fun doSomething() {
        save() // myRepository.save()
    }

    companion object Singleton {
        const val PI: Double = 3.14
        fun foobar() {}
    }
}
```

2. Kotlin

```
class MyService(val myRepository: MyRepository):
        MyRepository by myRepository {

    val b: String by lazy { "b lazy init" }

    fun doSomething() {
        save() // myRepository.save()
    }

    companion object Singleton {
        const val PI: Double = 3.14
        fun foobar() {}
    }
}
```



Helpers

```
@Test fun `I have a cool name`() { ... }

assertTrue { ... } //assertFails { }

assertFailsWith <ArithmeticException> { ... }

val time: Long = measureTimeMillis { ... }
```

Helpers

```
fun doSomething1(x: Int): Int {
    TODO( reason: "dont call this method yet")
}

@Deprecated( message: "nooooo",
             level = WARNING,
             replaceWith = ReplaceWith( expression: "good()"))


@JvmName( name: "sumInt")
fun sum(l: List<Int>) { ... }

@JvmName( name: "sumLong")
fun sum(l: List<Long>) { ... }
```

```kotlin
fun <T> readerValue(value: Any, clazz: Class<T>): T { ... }

fun <T> String.toJson(): T {
    return readerValue( value: this, T::class.java )
}

inline fun <reified T> String.json(): T {
    return readerValue( value: this, T::class.java)
}
```

- Kotlin - Tacit programming, also called point-free style

```
val game : Game = getGameInfo()
val players : List<Player> = getPlayers(game)
val records : List<PlayerHistory> = updateResults(game, players)
displayResults(records)
```

- let, also, apply, run and with

```
getGameInfo()
        .let { Pair(it, getPlayers(it)) }
        .let { updateResults(it.first, it.second) }
        .also { displayResults(it) }
```

2. Kotlin

- Kotlin - Tacit programming, also called point-free style

```
val game : Game = getGameInfo()
val players : List<Player> = getPlayers(game)
val records : List<PlayerHistory> = updateResults(game, players)
displayResults(records)
```

```
if (game ≠ null) {
    game.let { ... }
}
```

- let

```
g                                    Players(it)) }
        .let { updateResults(it.first, it.second) }
        .also { displayResults(it) }
```

2. Kotlin

- Kotlin - Tacit programming, also called point-free style

```
val game: Game = getGameInfo()
val players: List<Player> = getPlayers(game)
val records: List<PlayerHistory> = updateResults(game, players)
displayResults(records)
```

```
if (game ≠ null) {
    game.let { ... }
}
```

```
game?.let { ... }
```

- let

```
g                                    Players(it)) }
        .let { updateResults(it.first, it.second) }
        .also { displayResults(it) }
```

- let: conversion of value (*aka: map()*)
  - *it -> different*

```
public inline fun <T, R> T.let(block: (T) → R): R

val gameResult: GameResult = game.let { it: Game
    println("it $it")
    ^let GameResult(it) // return
}
```

- let: conversion of value (*aka: map())*
  - *it -> different*

```
public inline fun <T, R> T.let(block: (T) → R): R

val gameResult: GameResult = game.let { it: Game
    println("it: $it")
    ^let Game
}
```



```
game.let { mapGameToResult(it) }

game.let (::mapGameToResult)
```

- also: additional processing on an object in a call chain
  - *it -> same*

```
public inline fun <T> T.also(block: (T) → Unit): T
```

```
val game : Game = totalWar
        .also { println(it.name) }
        .also { it.startGame() }
```

- apply: post-construction configuration
  - *this -> same*

```
public inline fun <T> T.apply(block: T.() → Unit): T
```

```
val warThunder = Game( name: "War Thunder")
val game : Game = warThunder.apply { this: Game
    type = FS
    vendor = "Gaijin"
    initSomething()
}
```

- run: used with lambdas with some side-effects
  - *this -> different*

```
public inline fun <T, R> T.run(block: T.() → R): R
```

```
val gameContext: GameContext? = warThunder?.run { this: Game
    startGame()
}
```

- with: should be used to call multiple methods on an object created somewhere else
  - *this -> different*

```
public inline fun <T, R> with(receiver: T, block: T.() → R): R
```

```
val result: String = with(totalWar) { this: Game
    clear()
    loadPlayers()
    startGame()
    ^with "foobar"
}
```

Imperative Style

```
var page : PageScroll<Person>  = personRepository
        .startScroll(matchAllQuery(), size: 20)
do {
    page = personRepository.continueScroll(page.scrollId)
    // do something
} while (page.hasContent())
```

2. Kotlin

```
ScrollStream
        .startScroll { ... }
        .continueScroll { ... }
        .clearScroll { ... }
        .filter { ... }
        .takeWhile { ... }
        .forEach { ... }
```

Funcional Compose Style

2. Kotlin

```
ScrollStream
        .startScroll { ... }
        .continueScroll { ... }
        .clearScroll { ... }
        .filter { ... }
        .takeWhile { ... }
        .forEach { ... }
```

Funcional Compose Style

```
tailrec fun forEach(scrollId: String? = null,
                    takeAccumulator: Int,
                    action: (element: T) → Unit) { ... }
```

- Anemic Domain Model - *kotlin extensions*

```kotlin
fun Order.isPending(): Boolean =
        this.status == PENDING

fun List<Order>.hasPending(): Boolean =
        this.any(Order::isPending)

fun Customer.hasPendingOrders(): Boolean =
        this.orders.hasPending()
```

```kotlin
if (customer.hasPendingOrders()) {
    // do something
}
```

- Anemic Domain Model - *kotlin extensions*

```
fun Order.isPending(): Boolean =
        this.status == PENDING

fun List<Order>.hasPending(): Boolean =
        this.any(Order::isPending)

fun Customer.hasPendingOrders(): Boolean =
        this.orders.hasPending()
```

```
if (customer.hasPendingOrders()) {
    // do something
}
```

- Anemic Domain Model - *kotlin extensions*

```
fun Order.isPending(): Boolean =
        this.status == PENDING

fun List<Order>.hasPending(): Boolean =
        this.any(Order::isPending)

fun Customer.hasPendingOrders(): Boolean =
        this.orders.hasPending()
```

```
if (customer.hasPendingOrders()) {
    // do something
}
```

- Anemic Domain Model - *kotlin extensions*

```
fun Order.isPending(): Boolean =
        this.status == PENDING

fun List<Order>.hasPending(): Boolean =
        this.any(Order::isPending)

fun Customer.hasPendingOrders(): Boolean =
        this.orders.hasPending()
```

```
if (customer.hasPendingOrders()) {
    // do something
}
```

Events

```
sealed class OrderEvent : Event() {

    fun apply(aggregateId: AggregateId, handler: OrderEventHandler) =
            when (this) {
                is OrderCreated → handler.on(aggregateId, this)
                is ProductAdded → handler.on(aggregateId, this)
                is ProductDeleted → handler.on(aggregateId, this)
            }

}

data class OrderCreated(val orderId: String, val status: OrderStatus) : OrderEvent()
data class ProductAdded(val productId: String) : OrderEvent()
data class ProductDeleted(val productId: String) : OrderEvent()
```

4. Kotlin, DDD, ES & CQRS

> Events

Sealed Class

Pattern Matching

Smart Casts

```
sealed class OrderEvent : Event() {

    fun apply(aggregateId: AggregateId, handler: OrderEventHandler) =
            when (this) {
                is OrderCreated -> handler.on(aggregateId, this)
                is ProductAdded -> handler.on(aggregateId, this)
                is ProductDeleted -> handler.on(aggregateId, this)
            }

}
```

Final

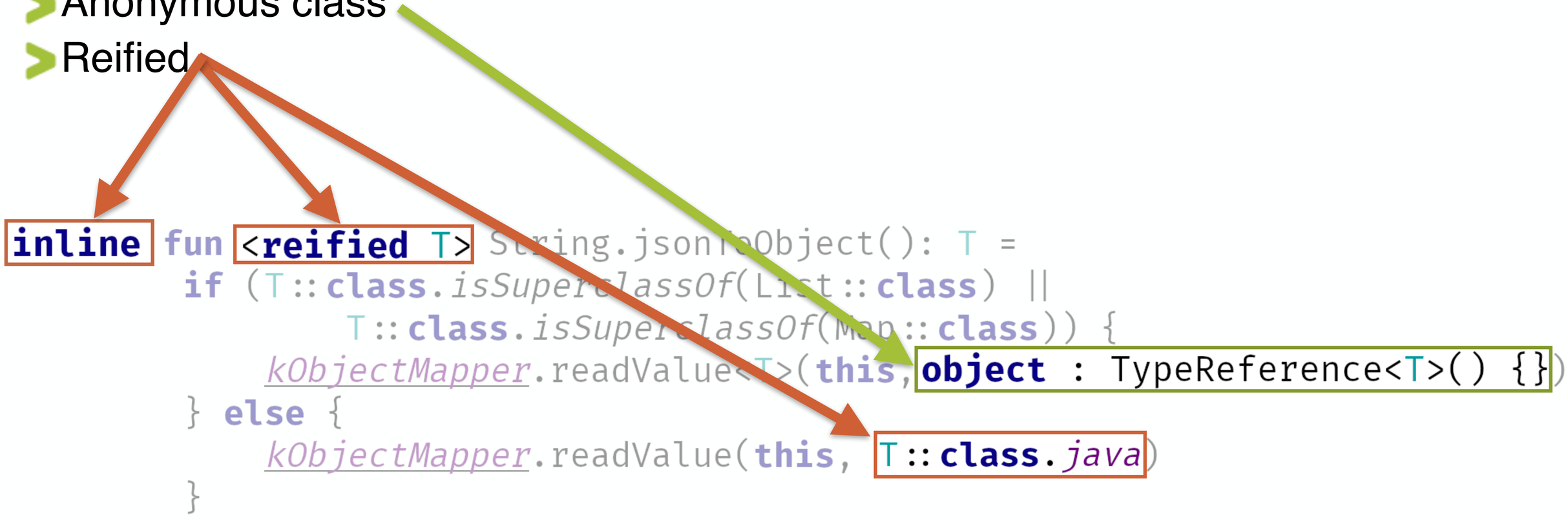
```
data class OrderCreated(val orderId: String, val status: OrderStatus) : OrderEvent()
data class ProductAdded(val productId: String) : OrderEvent()
data class ProductDeleted(val productId: String) : OrderEvent()
```

5. Kotlin 102

- Anonymous class
- Reified

```
inline fun <reified T> String.jsonToObject(): T =
        if (T::class.isSuperclassOf(List::class) ||
                T::class.isSuperclassOf(Map::class)) {
            kObjectMapper.readValue<T>(this, object : TypeReference<T>() {})
        } else {
            kObjectMapper.readValue(this, T::class.java)
        }
```

- Anonymous class
- Reified



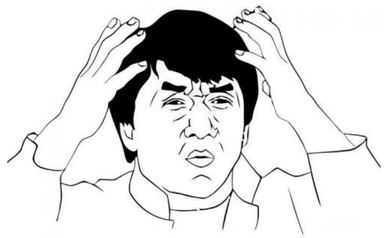
```
inline fun <reified T> String.jsonToObject(): T =
        if (T::class.isSuperclassOf(List::class) ||
                T::class.isSuperclassOf(Map::class)) {
            kObjectMapper.readValue<T>(this, object : TypeReference<T>() {})
        } else {
            kObjectMapper.readValue(this, T::class.java)
        }

data class User(val name: String, val age: Int)

val users: List<User> = """[{"name":"User 1","age": 21}]""".jsonToObject()
```

5. Kotlin 102

- Kotlin is the path to Functional Programming. Functional Programming leads to Pure Functions, Pure Functions leads to Monad and *Asynchronous Programming*. Asynchronous Programming leads to Reactive Programming.

Master Yoda - Return of the Jedi

- Functional Programming
  - Purity (no-side effect)
  - Referencial Transparency
  - Purity vs I/O
    - Exceptions propagation
    - Thread limits



```
fun get(page: PageNumber, callback: Callback<List<Game>>) {
    try {
        val games: List<Game> = gameRestV1.getGames(page)
        callback.onSuccess(games)
    } catch (e: Exception) {
        logger.error(e)
        callback.onError(e.message)
    }
}
```

- Monad - Either

```
sealed class Either<L, R>
```

```
class Left<L, R>(val left: L)
```

```
class Right<L, R>(val right: R)
```

```
fun get(page: PageNumber): List<Game> {

    val result: Either<Failure, List<Game>> = gameRest.getGames(page)

    return result.fold({ it: Failure
                emptyList()
            },
            { it: List<Game>
                it
            })
}
```

- Monad - Either

```
result.isLeft()
result.isRight()
result.get()
result.failure()
result.success { it: List<Game>
}
result.failure { it: Failure
}
```

Monad - Either -> Either -> Either

```
fun updateWithCallBackHell(gameId: GameId) {
    gameRest.getGame(gameId).fold(
            { logger.error(it) },
            { game : Success<Game> →
                gameRest.getPlayers(game.result).fold(
                        { logger.error(it) },
                        { it: Success<GamePlayer>
                            gameRest.updateResults(it.result).fold(
                                    { logger.error(it) },
                                    { result : Success<GameResult> →
                                        result.also { it: Success<GameResult>
                                            logger.info( msg: "finally!")
                                        }
                                    })
                        })
            })
}
```

FOLD HELL

- Either foldCompose

```
gameRest.getGame(gameId)
        .foldCompose({ ... }, {gameRest.getPlayers(it.result) })
        .foldCompose({ ... }, {gameRest.updateResults(it.result)})
        .fold({ /* i quit!!! */ }, { /* success */ })
```

- Either foldCompose

```
gameRest.getGame(gameId)
        .foldCompose({ ... }, {gameRest.getPlayers(it.result) })
        .foldCompose({ ... }, {gameRest.updateResults(it.result)})
        .fold({ /* i quit!!! */ }, { /* success */ })
```

- Either foldCompose

```
gameRest.getGame(gameId)
        .foldCompose({ ... }, {gameRest.getPlayers(it.result) })
        .foldCompose({ ... }, {gameRest.updateResults(it.result)})
        .fold({ /* i quit!!! */ }, { /* success */ })
```

- Either foldCompose

```
gameRest.getGame(gameId)
        .foldCompose({ ... }, {gameRest.getPlayers(it.result) })
        .foldCompose({ ... }, {gameRest.updateResults(it.result)})
        .fold({ /* i quit!!! */ }, { /* success */ })
```

- Either foldCompose
  - ping-pong effect

```
gameRest.getGame(gameId)
    .foldCompose(
        { backupRest.tryGetGameAndPlayers(gameId, it) }, { ... })
    .foldCompose({ ... }, { gameRest.updateResults(it.result) })
    .fold({ ... }, {  /* success */ })
```

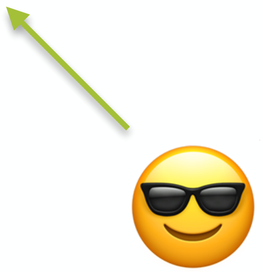
- Either foldCompose
  - ping-pong effect

```
gameRest.getGame(gameId)
    .foldCompose(
        { backupRest.tryGetGameAndPlayers(gameId, it) }, { ... })
    .foldCompose({ ... }, { gameRest.updateResults(it.result) })
    .fold({ ... }, {  /* success */ })
```

5. Kotlin 102

- Either foldCompose
  - ping-pong effect

```
gameRest.getGame(gameId)
    .foldCompose(
        { backupRest.tryGetGameAndPlayers(gameId, it) }, { ... })
    .foldCompose({ ... }, { gameRest.updateResults(it.result) })
    .fold({ ... }, {  /* success */ })
```



- Either getOrElse, andThen

```
gameRest.getGame(gameId)
        .getOrElse { backupRest.getGame(gameId) }
        .getOrElse { backupRest.createNewGame() }
        .andThen { gameRest.getPlayers(it.result) }
        .andThen { gameRest.updateResults(it.result) }
        .fold({ ... }, { it /* success */ })
```

- Either getOrElse, andThen

```
gameRest.getGame(gameId)
        .getOrElse { backupRest.getGame(gameId) }
        .getOrElse { backupRest.createNewGame() }
        .andThen { gameRest.getPlayers(it.result) }
        .andThen { gameRest.updateResults(it.result) }
        .fold({ ... }, { it /* success */ })
```

5. Kotlin 102

zkotlin

https://github.com/ZupIT/zkotlin

maven central 1.0.5

Extensions for Kotlin Language

- Either<L, R> - Examples
  - Either.Left<L,R> Either.Right<L,R>
    - Either.right<Int, String>("right value")
    - Either.left<Int, String>(-1)
    - Either.rightOf("right value")
    - Either.leftOf(-1)
  - fun isRight(): Boolean
  - fun isLeft(): Boolean
  - fun component1(): L
  - fun component2(): R
  - fun get(): R
  - fun failure(): L
  - fun success(onSuccess: (R) -> T): T
  - fun failure(onFail: (L) -> T): T
  - fun fold(left: (L) -> T, right: (R) -> T): T
  - fun <TL, TR> foldCompose(left: (L) -> Either<TL, TR>, right: (R) -> Either<TL, TR>): Either<TL, TR>

Coroutines

- simplify asynchronous programming by putting the complications into libraries
- we put the complications into libraries
- experimental != unsecurity

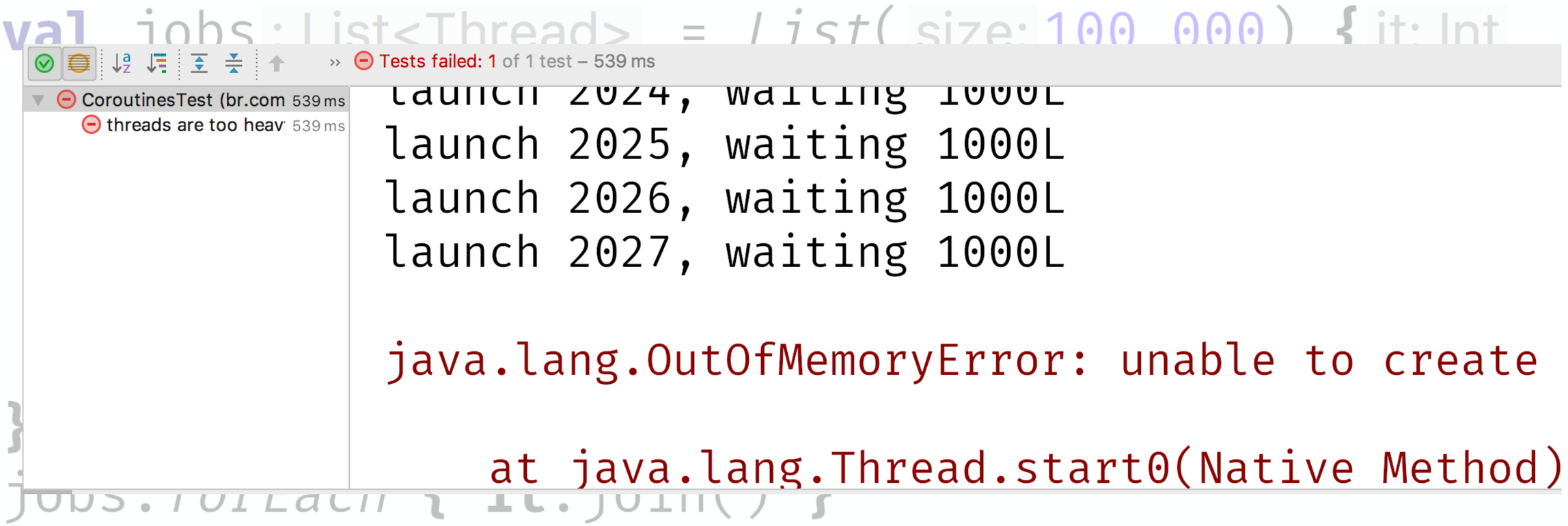
Coroutines

What's the result?

```
val jobs: List<Thread> = List( size: 100_000) { it: Int
    thread {
        println("launch $it, waiting 1000L")
        Thread.sleep( millis: 1000L)
        print("$it done ")
    }
}
jobs.forEach { it.join() }
```

Coroutines

What's the result?

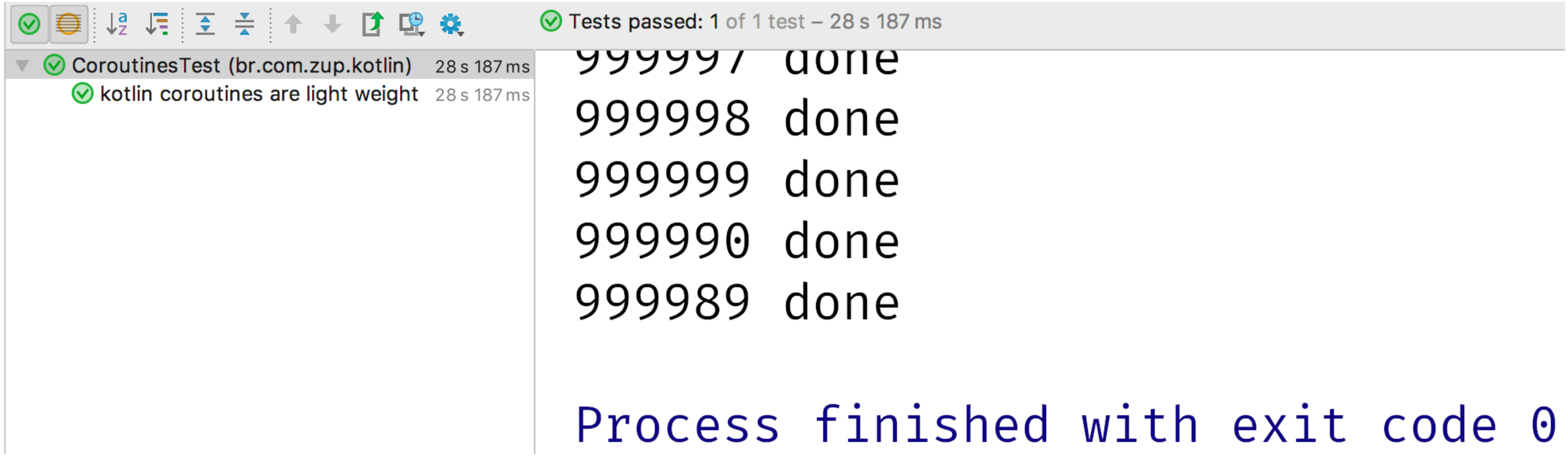


- Coroutines are light weight!

```
@Test
fun `kotlin coroutines are light weight`(): Unit = runBlocking {

    val jobs: List<Job> = List( size: 1_000_000) { it: Int
        launch { this: CoroutineScope
            println("launch $it, waiting 200L")
            delay( time: 200L)
            println("$it done ")
        }
    }
    jobs.forEach { it.join() }
}
```

- Coroutines are light weight!

```
@Test
fun `kotlin coroutines are light weight`(): Unit = runBlocking {

    val jobs: List<Job> = List( size: 1_000_000) { it: Int
```




```
}
```

- Coroutines - may suspend/resume

```
suspend fun findCatalogId(id: CatalogId): Catalog { ... }
suspend fun findOffers(id: CatalogId): List<Offer> { ... }
```

another supending fun

```
@Test
fun `no callback hell, sequential call`(): Unit = runBlocking { this:
  val catalog: Catalog = service.findCatalogId(CatalogId( value: 1))

  val offers: List<Offer> = service.findOffers(catalog.id)

  assertEquals(listOf("Offer1", "Offer2"),
               offers.map { it.name })
}
```

Zup

- Coroutines - async/await

```
val findOne1: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 1)) }

val findOne2: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 2)) }

runBlocking { this: CoroutineScope
  val catalog1: Catalog = findOne1.await()
  val catalog2: Catalog = findOne2.await()

  print("($catalog1) ($catalog2)")
}
```

- Coroutines - async/await

```
val findOne1: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 1)) }

val findOne2: Deferred<Catalog> =
    async { service.findCatalogId(CatalogId( value: 2)) }

runBlocking { this: CoroutineScope
  val catalog1: Catalog = findOne1.await()
  val catalog2: Catalog = findOne2.await()

  print("($catalog1) ($catalog2)")
}
```

- Coroutines - async/await

```
val findOne1: Deferred<Catalog> =
```

Tests passed: 1 of 1 test – 9 s 62 ms

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152
starting findCatalogId(2) timeout: 5
starting findCatalogId(1) timeout: 9
returning from findCatalogId(2)
returning from findCatalogId(1)
(Catalog(id=1, name=catalog 1)) (Catalog(id=2,
Process finished with exit code 0
```

```
val
run
print("($catalog1) ($catalog2)")
}
```

Either, Coroutines, Extensions, zkotlin all together

- *Wraps I/O problems with Either<L,R>*

```
sealed class ServiceError(val error: String?,
                          val exception: Exception? = null) {
    class NotFound(error: String?): ServiceError(error)
    class UnknownError(error: String, exception: Exception):
            ServiceError(error, exception)
}
```

> *Wraps I/O problems with Either<L,R>*

```
fun findBundleById(bundleId: BundleId):
                    Either<ServiceError, BundleRepresentation> =
    try {
        findOne(bundleId)
                ?.let( ::eitherRight)

                ?: eitherLeftNotFound(bundleId)

    } catch (e: Exception) {
        eitherLeftUnexpectedError(bundleId, e)
    }
```

- *Wraps I/O problems with Either<L,R>*

```
fun findBundleById(bundleId: BundleId):
                Either<ServiceError, BundleRepresentation> =
    try {
        findOne(bundleId)
                ?.let(::eitherRight)

                ?: eitherLeftNotFound(bundleId)

    } catch (e: Exception) {
        eitherLeftUnexpectedError(bundleId, e)
    }
```

```
Either.right<ServiceError, BundleRepresentation>(bundle)
```

- *Wraps I/O problems with Either<L,R>*

```
fun findBundleById(bundleId: BundleId):
                Either<ServiceError, BundleRepresentation> =
    try {
        findOne
                ?.let(::eitherRight)

                ?: eitherLeftNotFound(bundleId)

    } catch (e: Exception) {
        eitherLeftUnexpectedError(bundleId, e)
    }
```

```
Either.left<ServiceError, BundleRepresentation>(
        NotFound( error: "BundleId $bundleId not found"))
```

```
Either.left(UnknownError(
        error: "Unexpected error BundleId $bundleId.", e))
```

- *Wraps I/O problems with Either<L,R>*

```
fun findBundleById(bundleId: BundleId):
                Either<ServiceError, BundleRepresentation> =
    try {
        findOne(bundleId)
```

```
typealias EitherBundle = Either<ServiceError, Bundle>

typealias DeferredEitherBundle = Deferred<EitherBundle>

typealias EitherBundleRepresentation =
        Either<ServiceError, BundleRepresentation>
```

Zup

- sync {}

```
private fun findAtPcm(tenant: Tenant, bundleId: BundleId):
        EitherBundleRepresentation = tenant { this: Tenant
    bundleSearch.findBundleById(bundleId)
}
```

- async {}

```
private fun asyncFindAtPcm(tenant: Tenant,
                           bundleId: BundleId):
        DeferredEitherBundleRepresentation = async {
    findAtPcm(tenant, bundleId)
}
```

```
> sync {}

  private fun findAtPcm(tenant: Tenant, bundleId: BundleId):
          EitherBundleRepresentation = tenant { this: Tenant
      bundleSearch.findBundleById(bundleId)
  }
```

```
inline operator fun <T> invoke(block: Tenant.() → T): T {
    ThreadLocalTenantIdentifierContext.set(organization, application)
    return block()
}
```

```
          DeferredEitherBundleRepresentation = async {
      findAtPcm(tenant, bundleId)
  }
```

runBlocking

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
        = runBlocking { this: CoroutineScope
    bundleIds
            .map(asyncFindPcmByTenant(tenantContext))
            .await()
            .foldError(::handlerFindPcmError)
            .foldSuccess(::mapToModel)

            .map(asyncSaveByTenant(tenantContext))
            .await()
            .foldError(::handlerSaveError)
            .foldSuccess()
}
```

5. Kotlin 102

> runBlocking

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
        = runBlocking { this: CoroutineScope
    bundleIds
            .map(asyncFindPcmByTenant(tenantContext))
            .await()
            .foldError(::handlerFindPcmError)
            .foldSuccess(::mapToModel)

            .map(asyncSaveByTenant(tenantContext))
            .await()
            .foldError(::handlerSaveError)
            .foldSuccess()
}
```

zkotlin

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
        = runBlocking { this: CoroutineScope
    bundleIds
            .map(asyncFindPcmByTenant(tenantContext))
            .await()
            .foldError(::handlerFindPcmError)
            .foldSuccess(::mapToModel)
```

```
suspend fun <T> List<Deferred<T>>.await(): List<T> =
        this.map { it.await() }
```

```
            .foldSuccess()
}
```

zkotlin

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
        = runBlocking { this: CoroutineScope
    bundleIds
            .map(asyncFindPcmByTenant(tenantContext))
            .await()
            .foldError( ::handlerFindPcmError)
            .foldSuccess( ::mapToModel)
}
```

```
inline fun <L, R> List<Either<L, R>>.foldError(
        onError: (L) → Unit): List<Either<L, R>> { ... }

inline fun <L, R, T> List<Either<L, R>>.foldSuccess(
        onSuccess: (R) → T): List<T> { ... }
```

zkotlin

```
private fun saveBundleIds(tenantContext: TenantContext,
                          bundleIds: List<BundleId>): List<Bundle>
        = runBlocking { this: CoroutineScope
    bundleIds
            .map(asyncFindPcmByTenant(tenantContext))
            .await()
            .foldError( ::handlerFindPcmError)
}
```

```
private val asyncFindPcmByTenant = ::asyncFindAtPcm.curried()

fun <A, B, R> ((A, B) → R).curried(): (A) → (B) → R =
        { p1: A →
            { p2: B → this(p1, p2) }
        }
```

- Tests
  - Mockito Kotlin

```
private val productEventRepository = mock<EventStoreRepository<Product>> {
    on { get(AggregateId( value: "Product ID 1")) } doReturn product
    on { get(AggregateId( value: "Product ID 2")) } doReturn product
    on { get(AggregateId( value: "Product ID 3")) } doAnswer {
        throw Repository.NotFoundException()
    }
}
```

```
verify(productEventRepository, Times(1)).get(AggregateId("Product ID 1"))
```

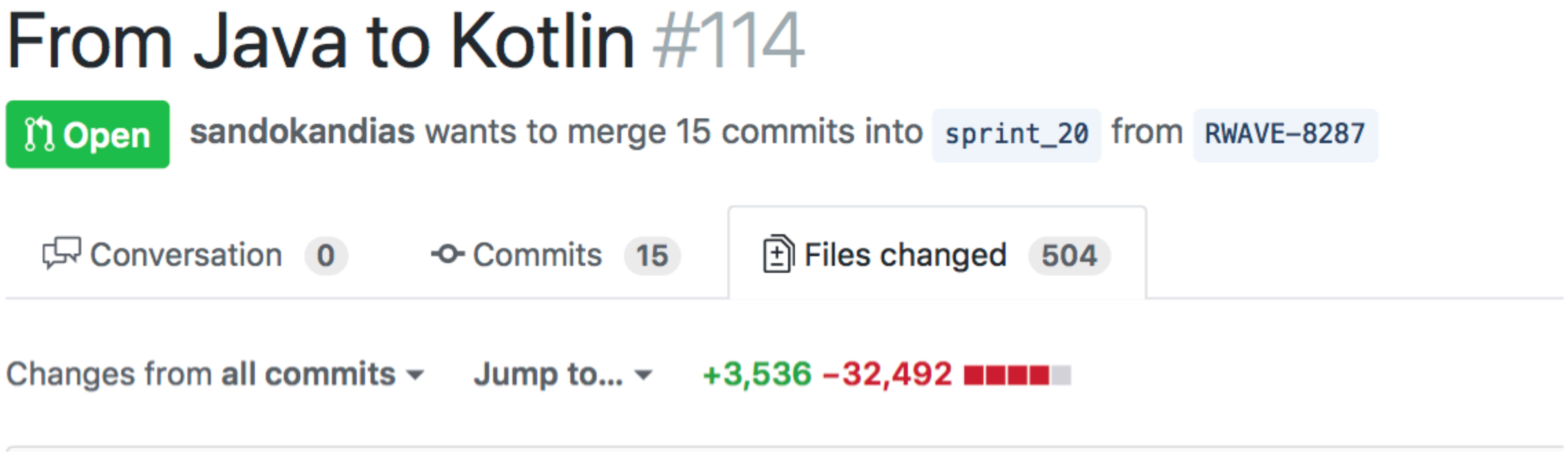
Tests

```
fun <E : Throwable> assertException(
        expectedException: Class<E>,
        expectedMessage: String,
        block: () → Any
) = try {
    block()
} catch (e: Throwable) {
    assertTrue(
            "The expected exception is [${expectedException.canonicalName}]," +
                    " but throws [${e.javaClass.canonicalName}].",
            { expectedException == e.javaClass }
    )
    assertTrue(
            "The expected message is [$expectedMessage]," +
                    " but throws the message [${e.message}].",
            { expectedMessage == e.message }
    )
}
```

- Tests

```
assertException(
        expectedException = AttemptChangeOrderStatusException::class.java,
        expectedMessage = "It is not allowed change order status."
) {
    order.addProduct( productId: "Product ID 2", productEventRepository)
}

assertException(
        expectedException = AttemptChangeOrderStatusException::class.java,
        expectedMessage = "It is not allowed change order status.",
        assertThat = { e: AttemptChangeOrderStatusException →
            e.status == OrderStatus.FINISHED
        },
        block = {
            order.addProduct( productId: "Product ID 2", productEventRepository)
        }
)
```

ZupIT


From Java to Kotlin #114
Open
sandokandias wants to merge 15 commits into sprint_20 from RWAVE-8287
Conversation 0
Commits 15
Files changed 504
Changes from all commits
Jump to...
+3,536 −32,492

6. Kotlin, Java & Legacy

- Lambdas returning Unit

```
tailrec fun forEach(scrollId: String? = null,
                    action: (content: List<T>) → Unit) { ... }
```

6. Kotlin, Java & Legacy

- Lambdas returning Unit

```
tailrec fun forEach(scrollId: String? = null,
                    action: (content: List<T>) → Unit) { ... }
```

- Kotlin

```
.forEach { content: List<Person> →
    //do something

    // no return
}
```

6. Kotlin, Java & Legacy

- Lambdas returning Unit

```
tailrec fun forEach(scrollId: String? = null,
                    action: (content: List<T>) → Unit) { ... }
```

- Kotlin

```
.forEach { content: List<Person> →
    //do something

    // no return
}
```

- Java

```
.forEach( scrollId: null, content → {
    //do something

    return null;
});
```

6. Kotlin, Java & Legacy

- Lambdas returning Unit

```
tailrec fun forEach(scrollId: String? = null,
                    action: (content: List<T>) → Unit) { ... }
```

- *void` type in Java.*

```
public object Unit {
    override fun toString(): String = "kotlin.Unit"
}
```

6. Kotlin, Java & Legacy

- Just for Java

```java
@FunctionalInterface
public interface Consumer<T> {

    /** ... */
    void accept(T t);
```

```kotlin
fun forEach(action: Consumer<List<T>>) {
    forEach( scrollId: null) { e: List<T> -> action.accept(e) }
}
```

SpringBoot - ALL OPEN

final

```
5  @Configuration
6  class MyConfig {
8  }
```

Classes annotated with '@Configuration' could be implicitly subclassed and must not be final more... (⌘F1)

```
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-allopen</artifactId>
    <version>${kotlin.version}</version>
</dependency>
```

- SpringBoot - JPA

```
@Entity
data class Customer(@field:Id val id: CustomerId,
                    val name: CustomerName,
                    @field:Enumerated(EnumType.STRING)
                    val gender: Gender)
```

```
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-maven-noarg</artifactId>
    <version>${kotlin.version}</version>
</dependency>
```

Spring Data QueryDSL - Default Methods Java 8

```
@Repository
interface CustomerRepository : JpaRepository<Customer, CustomerId>,
                               QueryDslPredicateExecutor<Customer>,
                               QuerydslBinderCustomizer<QCustomer> {

    @JvmDefault
    override fun customize(bindings: QuerydslBindings,
                           qCustomer: QCustomer) {
     bindings.bind(String::class.java).all {
         path: StringPath, values →
             BooleanBuilder().apply {
               values.forEach { this.or(path.containsIgnoreCase(it)) }
         }
     }
    }
```

6. Kotlin, Java & Legacy

Spring Data QueryDSL - Default Methods Java 8

```
@Repository
interface CustomerRepository : JpaRepository<Customer, CustomerId>,
                               QueryDslPredicateExecutor<Customer>,
                               QuerydslBinderCustomizer<QCustomer> {

    @JvmDefault
    override fun customize(bindings: QuerydslBindings,
                           qCustomer: QCustomer) {
     bindings.bind(String::class.java).all {
         path: StringPath, values →
             BooleanBuilder().apply {
               values.forEach { this.or(path.containsIgnoreCase(it)) }
         }
     }
    }
```

Kotlin 1.2.40

6. Kotlin, Java & Legacy

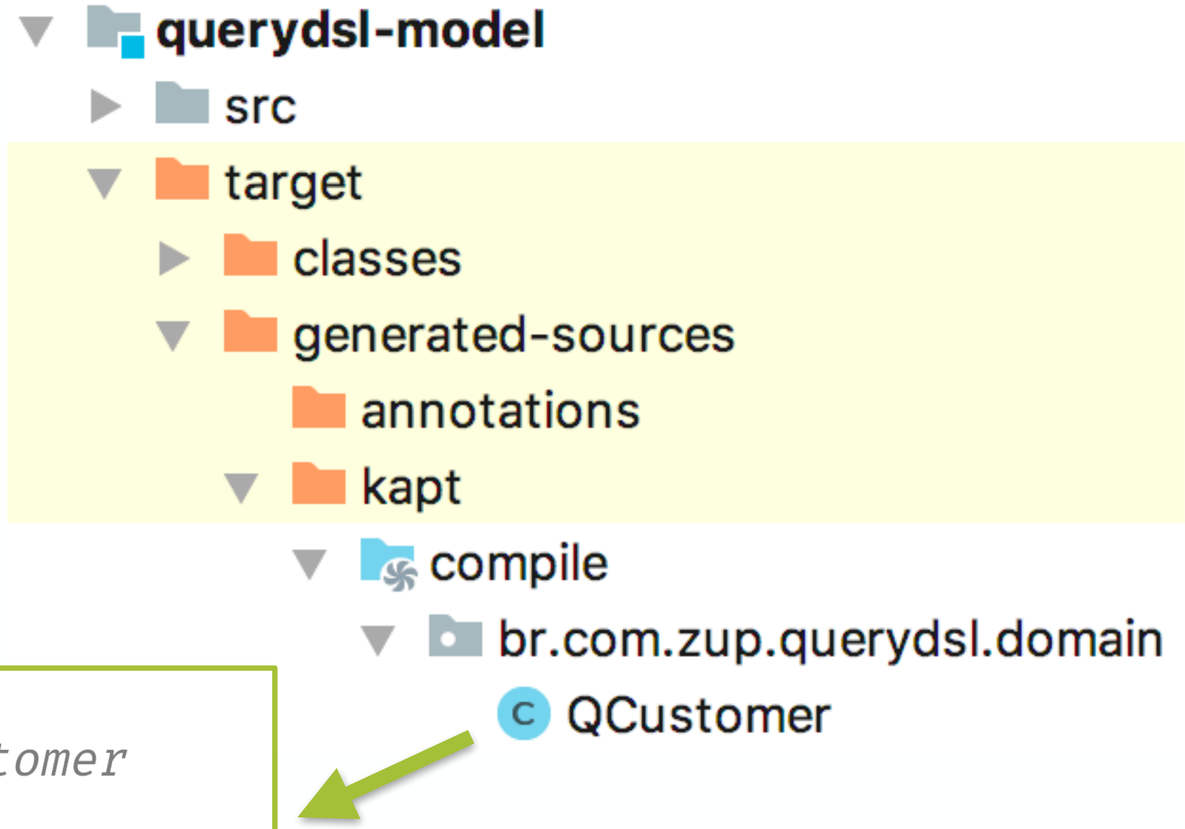
Spring Data QueryDSL - Plugins - Annotation Processing with Kotlin

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>apt-maven-plugin</artifactId>
</plugin>
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs ... >
        <annotationProcessorPaths>
            <annotationProcessorPath>
                <groupId>com.querydsl</groupId>
                <artifactId>querydsl-apt</artifactId>
                <version>${querydsl.version}</version>
                <classifier>jpa</classifier>
            </annotationProcessorPath>
        </annotationProcessorPaths>
    </configuration>
</execution>
```

6. Kotlin, Java & Legacy

QCustomer - isolate module

```
<plugin>
    <groupId>com.mysema.maven</groupId>
    <artifactId>apt-maven-plugin</artifactId>
</plugin>
<execution>
    <id>kapt</id>
    <goals>
        <goal>kapt</goal>
    </goals>
    <configuration>
        <sourceDirs ... >
        <annotationProcessorPaths>
            <annotationProcessorPath>
```




```
/**
 * QCustomer is a Querydsl query type for Customer
 */
@Generated("com.querydsl.codegen.EntitySerializer")
public class QCustomer extends EntityPathBase<Customer>
```

```
    </configuration>
</execution>
```

6. Kotlin, Java & Legacy

Final classes & methods

Spock

```
<dependency>
    <groupId>de.jodamob.kotlin</groupId>
    <artifactId>kotlin-runner-spock</artifactId>
    <version>${kotlin.runner.spock.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib-nodep</artifactId>
    <version>${cglib.nodep.version}</version>
    <scope>test</scope>
</dependency>
```

Mockito

```
<dependency>
    <groupId>com.nhaarman</groupId>
    <artifactId>mockito-kotlin</artifactId>
    <version>${mockito.kotlin.version}</version>
    <scope>test</scope>
</dependency>
```