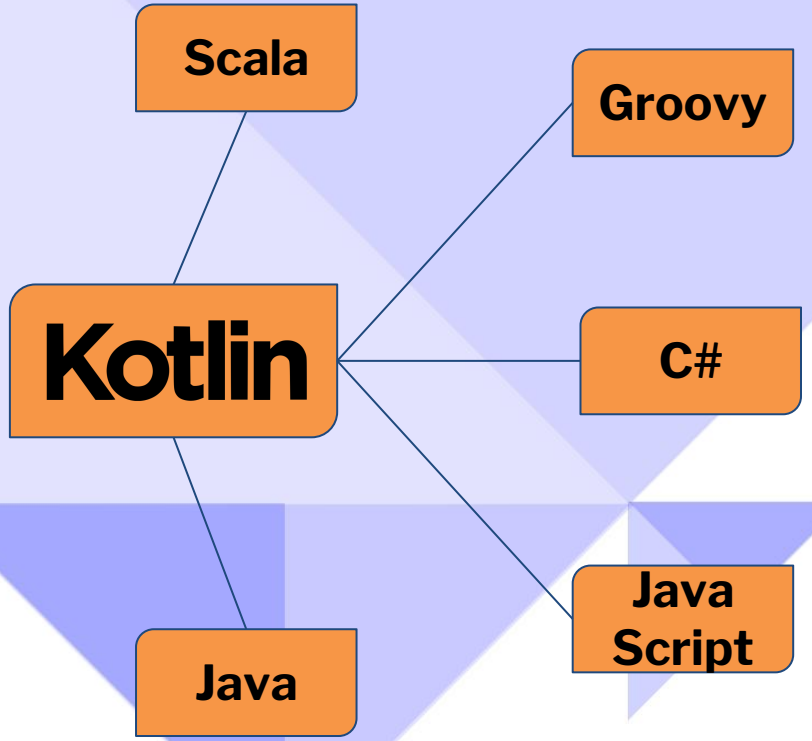
Carolina Manso
Julio Bissoli
Lorena Bassani
Murilo Scalser
Ricardo Braun

História

História

- Kotlin é uma linguagem de programação desenvolvida pela JetBrains.
- Anunciada em 2011 porém primeira versão estável foi em 2016.
- Motivação: melhorar a produtividade; Obter uma linguagem com uma sintaxe mais simples, de fácil aprendizado e ao mesmo tempo com maior desempenho.
- Em 2017 se tornou a nova linguagem oficial da plataforma Android.

Influências



THE APACHE SOFTWARE FOUNDATION

Permissões

- Uso comercial
- Modificação
- Distribuição
- Uso de patentes
- Uso privado

Limitações

- Uso de marca registrada
- Responsabilidade
- garantia

Condições

- Licença e aviso de direitos autorais
- Mudanças de estado

Kotlin

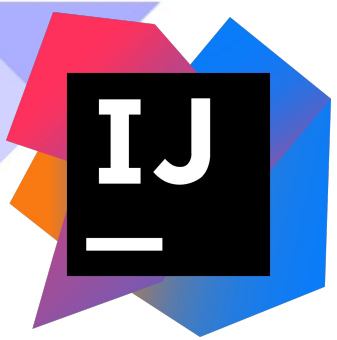
Mini tutorial

Começando

Pré-Requisito: todo ambiente da JVM e Java configurado e instalado na sua máquina.

Testando seu código

- Linha de comando
- IDE



Kotlin Standard Library

- *kotlin.annotation*
- *kotlin.browser*
- *kotlin.collections*
- *kotlin.comparisons*
- *kotlin.concurrent*
- *kotlin.contracts*
- *kotlin.coroutines*
- *kotlin.coroutines.experimental*
- *kotlin.coroutines.experimental.intrinsics*
- *kotlin.coroutines.intrinsics*
- *kotlin.dom*
- *kotlin.experimental*
- *kotlin.io*
- *kotlin.js*
- *kotlin.jvm*
- *kotlin.jvm.functions*
- *kotlin.math*
- *kotlin.native*
- *kotlin.native.concurrent*
- *kotlin.native.ref*
- *kotlin.properties*
- *kotlin.random*
- *kotlin.ranges*
- *kotlin.reflect*
- *kotlin.reflect.full*
- *kotlin.reflect.jvm*
- *kotlin.sequences*
- *kotlin.streams*
- *kotlin.system*
- *kotlin.text*
- *kotlinx.cinterop*
- *kotlinx.wasm.jsinterop*

Hello World

```
package org.kotlinlang.play          // 1

fun main() {                         // 2
        println("Hello World!") // 3
}
```

Comentários - Igual a JAVA

Pacotes e Importações - 1 - Sintaxe igual a JAVA.

main - 2 - A função main é o ponto de entrada da aplicação Kotlin.

println - 3 - A função foi implicitamente importada da biblioteca de kotlin.

Fim de linha - 3 - O uso do ponto e vírgula no final de uma expressão é opcional.

Sintaxe Básica

```
fun soma(a: Int, b: Int): Int {
        return a + b
}

/*Funções podem ter o corpo de expressão e tipo de retorno implícitos */
fun subtrai(a: Int, b: Int) = a - b

fun printSoma(a: Int, b: Int): Unit{
        println("soma de $a e $b é ${a+b}")
}

/*type Unit é quando o retorno não tem valor significativo e este também pode ser omitido */
fun printSub(a: Int, b: Int){
                println("subtração de $a e $b é ${a-b}")
}
```

Funções e Retornos – A sintaxe de uma função padrão é:

```
fun foo(<param : Tipo>) : <Tipo de retorno>{
        //implementação
}
```

Mas funções também podem ter o corpo de expressão e tipo de retorno implícitos

```
fun foo(<param : Tipo>) = <expressão>
```

Tipo Unit – O tipo Unit é semelhante ao void e pode ser omitido quando é retorno em função.

Sintaxe Básica

```
//Variaveis imutaveis
val cidade: String= “Vitória”
val anoNascimento = 1996

//Variaveis mutaveis
var nome: String  = “Lucas”
var idade = 22

//Expressões Condicionais
if (a > b) { return a
} else { return b }

//Uso para retorno
val max = if (a > b) a else b
```

Variáveis Mutáveis - variáveis usuais

Variáveis Imutáveis - semelhante às constantes

Expressões Condicionais – A expressão *if-else* pode ser usado para obter um valor.

Nesse caso o *else* é obrigatório.

Classes

```
class Cliente

class Contato(val id: Int, var email: String)

class ExemploMutavel<E>(vararg items: E) {

  private val elements = items.toMutableList()
  fun push(element: E) = elements.add(element)
  fun peek(): E = elements.last()
  fun pop(): E = elements.removeAt(elements.size - 1)
  fun isEmpty() = elements.isEmpty()
  fun size() = elements.size
  override fun toString() =
                "MutableStack(${elements.joinToString()})"
}
```

Classes - Sem parâmetro.

Data Class - Atributos e métodos construídos implicitamente

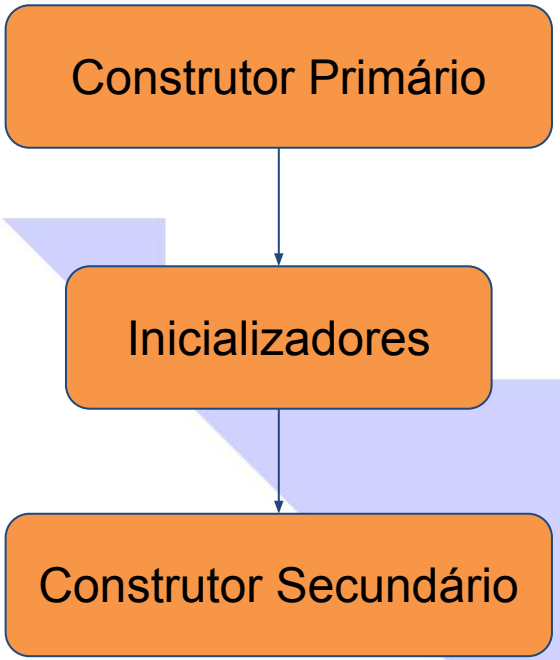
Classes Genéricas – Usadas para declarar classes que atuam sobre qualquer classe.

Construtores

```
class Cliente(name: String) {
    val ClienteKey = name.toUpperCase()
}
```

```
class Pessoa {
    constructor(parent: Pessoa) {
        parent.children.add(this)
    }
}
```

```
class Construtor {
    init {
        println("Init block")
    }
    constructor(i: Int) {
        println("Constructor")
    }
}
```



Late Init

```
class ExemploLateInit{
    private lateinit var inicializaDepois : IntRange

    fun InicializaVal(inicializador: Int){
        inicializaDepois = IntRange(0, inicializador)
    }

    fun getInicializaDepois() = inicializaDepois
}

fun main(args: Array<String>){
    val exemplo = ExemploLateInit();

    exemplo.InicializaVal(9)

    exemplo.getInicializaDepois().forEach { i -> print("$i ") }
}
```

Late Init – Inicialização de propriedade posteriormente, não no construtor.

A lateinit inicializa com null.

Funciona apenas com variáveis mutáveis e em tipos não primitivos.

Late Init

```
class ExemploLateInit{
    private lateinit var inicializaDepois : IntRange

    fun InicializaVal(inicializador: Int){
        inicializaDepois = IntRange(0, inicializador)
    }

    fun getInicializaDepois() = inicializaDepois
}

fun main(args: Array<String>){
    val exemplo = ExemploLateInit();

    exemplo.InicializaVal(9)

    exemplo.getInicializaDepois().forEach { i -> print("$i ") }
}
```

Late Init – Inicialização de propriedade posteriormente, não no construtor.

A lateinit inicializa com null.

Funciona apenas com variáveis mutáveis e em tipos não primitivos.

Saída:
0 1 2 3 4 5 6 7 8 9

Loops e Controle de Fluxo

```
// Usando o 'for' em loops
        val itens = listOf("Ola", "Mundo")
        for (item in itens){
                println(item)
        }

/* Usando 'when' e tipo 'Any' */
fun descrição(obj: Any): String =
   when(obj) {
        1 -> "um"
        "Hello" -> "Oi em Ingles!"
        true -> "Tipo Verdadeiro"
        else -> "Desconhecido"
        /*sempre necessário ter um else */
}
```

Loops – *for* e *while.*

Controle de Fluxo – *when:* Kotlin conta com um controle de fluxo *when*, parecido com um *switch-case* (Não presente na linguagem).

Quebra de fluxo em laço

```
fun main(args: Array<String>) {
    here@ for (i in 1..5) {
        for (j in 1..4) {
            if (i == 3 || j == 2)
                continue@here
            println("i = $i; j = $j")
        }
    }
}
```

Labels – Labels podem ser usados junto a return, break ou continue para especificar para onde o fluxo é quebrado.

Quebra de fluxo em laço

```
fun main(args: Array<String>) {
    here@ for (i in 1..5) {
        for (j in 1..4) {
            if (i == 3 || j == 2)
                continue
            println("i = $i; j =
        }
    }
}
```

Saída:

```
i = 1; j = 1
i = 1; j = 3
i = 1; j = 4
i = 2; j = 1
i = 2; j = 3
i = 2; j = 4
i = 4; j = 1
i = 4; j = 3
i = 4; j = 4
i = 5; j = 1
i = 5; j = 3
i = 5; j = 4
```

```
fun main(args: Array<String>) {
    here@ for (i in 1..5) {
        for (j in 1..4) {
            if (i == 3 || j == 2)
                continue@here
            println("i = $i; j = $j"
        }
    }
}
```

Saída:

```
i = 1; j = 1
i = 2; j = 1
i = 4; j = 1
i = 5; j = 1
```

Kotlin

Explicação Teórica

Características

Características

- 100% compatível com Java.
- 100% compatível com JavaScript
- HTML Builder
- A linguagem é padronizada pela Kotlin foundation:
  - As principais funções da Kotlin foundation são:
    - Conservação da marca registrada associada ao projeto Kotlin
    - Escolher o líder de projeto da linguagem
    - Controle de mudanças incompatíveis com a linguagem

Execução

Interpretada

Compilada

Híbrida

Execução

Interpretada **ıda** **a**

```
alert(myModule.foo());
```

Código Kotlin em JavaScript

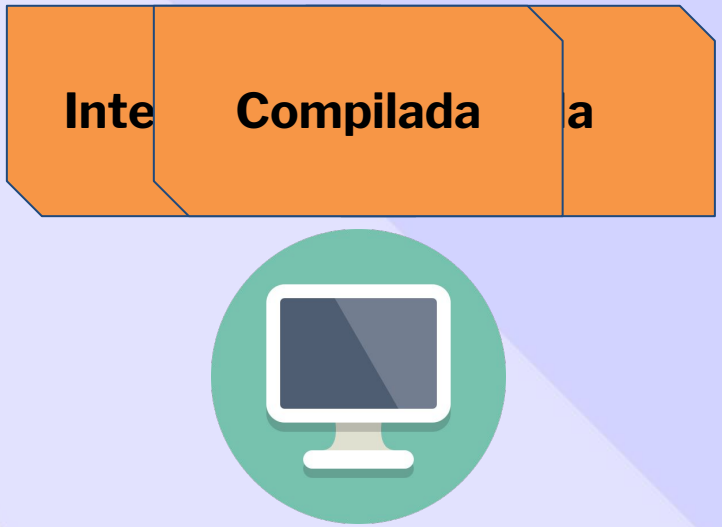
```
fun jsTypeOf(o: Any): String {
    return js("typeof o")
}
fun foo() = "Hello"
```

Código JavaScript em Kotlin

Pode ser Interpretada:

- O compilador Kotlin para JavaScript procura:
  - Prover saída de tamanho otimizada e escrita de forma legível
  - Prover interoperabilidade com os módulos já existentes do sistema
  - Prover as mesmas funcionalidades que possui na JVM.

Execução



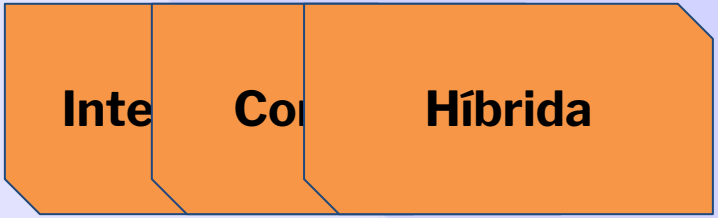
Pode ser compilada:

- plataformas listadas no manual da linguagem:
  - OS (arm32, arm64, emulator x86_64)
  - MacOS (x86_64)
  - Android (arm32, arm64)
  - Windows (mingw x86_64)
  - Linux (x86_64, arm32, MIPS, MIPS little endian)
  - WebAssembly (wasm32)

Permitindo que um programa Kotlin rode sem uma máquina virtual Java. É utilizado principalmente para incluir em projetos com outra linguagens tais quais C, C++, Swift, Objective-C, entre outras.

Possui coletor de lixo mesmo não estando vinculado a JVM.

Execução




```
fun main(args: Array<String>){

  var javaThread = Thread(
        Runnable{
            System.out.println("Sou uma Thread Java!")
        }
   )

  javaThread.start()
  javaThread.join()

}
```

Pode ser Híbrida:

- Compilada para bytecode da JVM
- 100% compatível com Java
  - Pode ser chamada em código Java
  - Pode chamar código Java

O maior foco de Kotlin é ser Híbrida e rodar na JVM.

Paradigmas

```
class imp : Runnable{
  override fun run(){
                println(
        "Estou implementando uma Interface!!")
  }
}
```

```
fun main(args: Array<String>){
        val array = intArrayOf(1,2,3)

        array.forEach { i -> print("$i ") }

}
```

- Orientado a Objetos
  O principal paradigma de Kotlin é orientado a objetos, com classes e herança.

- Funcional
  Kotlin também possui suporte ao paradigma funcional, com expressões lambda (anônimas).

Amarrações

Amarrações

- A Linguagem possui amarração estática e escopo estático e aninhado.

```
fun main(args: Array<String>){
        val a: Int
        var b = 0
        a = 3
        println(a)
        println(b)
}
```

Amarrações

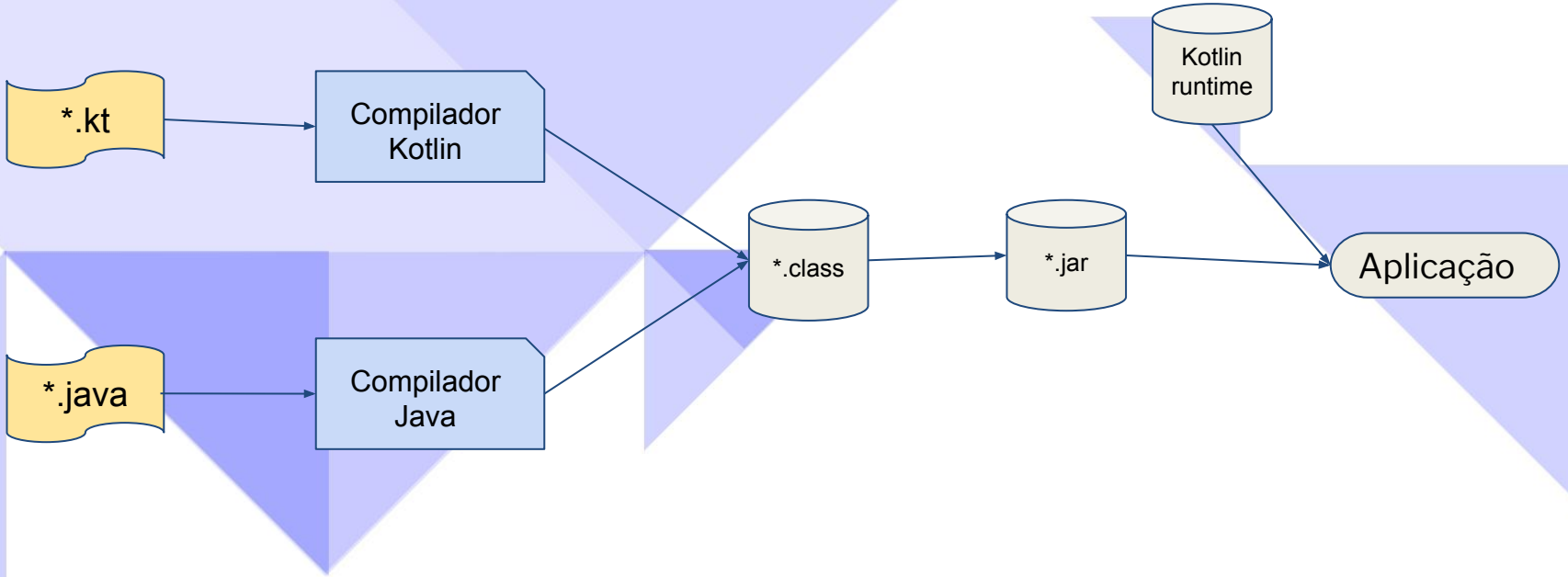
- A Linguagem possui amarração estática e escopo estático e aninhado.

```
fun main(args: Array<String>){
        val a: Int
        var b = 0
        a = 3
        println(a)
        println(b)
}
```

saída:
3
0

Amarrações

- A inferência de tipos ocorre em tempo de compilação



Identificadores

- Palavras Chave (Hard Keywords)

as	*break*	*class*	*continue*	*do*	*else*
false	*for*	*fun*	*if*	*in*	*interface*
is	*null*	*object*	*package*	*return*	*super*
this	*throw*	*true*	*try*	*typealias*	*typeof*
val	*var*	*when*	*while*		

Identificadores

- Os tokens a seguir agem como palavras-chave no contexto quando são aplicáveis e podem ser usados como identificadores em outros contextos (Soft Keywords)

by	catch	constructor	delegate
dynamic	field	file	finally
get	import	init	param
property	receiveris	set	setparam
where			

Identificadores

- (Modifier Keywords)

```
fun main(args: Array<String>){
        val public = "Publica"
        var a = Teste()
        println(public)
        println(a.nome)
}

class Teste {
    public val nome = "Nome"
}
```

- Modifier Keywords são tokens que podem ser observados como palavras chaves ou identificadores.
- Sendo definidas de acordo com contexto

Identificadores

- (Modifier Keywords)

```
fun main(args: Array<String>){
        val public = "Publica"
        var a = Teste()
        println(public)
        println(a.nome)
}

class Teste {
    public val nome = "Nome"
}
```

saída:
Publica
Nome

- Modifier Keywords são tokens que podem ser observados como palavras chaves ou identificadores.
- Sendo definidas de acordo com contexto

Identificadores

Não é permitido declarar identificadores:

@minhaVar

Começados com caracteres especiais

1minhaVar

Começados com números

class

Mesmo nome de palavras chave

minha Var

Separadas por espaço

Operadores

- Alguns exemplos de operadores da linguagem

Comparação

< > <= >=

Operadores Matemáticos

+ - / % *

Igualdade

== !=

Igualdade Referencial

=== !==

Incremento e Decremento

++ --

Operadores Lógicos

&& || !

Verificador de Nulidade

? ?. ?:

Operadores

Operador	Método
a++	a.inc(b)
a--	a.dec(b)
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

Operador	Método
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, … , i_n]	a.get(i_1, …, i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, … , i_n] = b	a.set(i_1, …, i_n, b)
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

Operadores

Operador	Método
a++	a.inc(b)
a--	a.dec(b)
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

```
fun main() {
 var a = 2
 var b = 4
 println(a+b)
 println(a.plus(b))
}
```

Operadores podem ser usados na sua forma mais conhecida ou como o método que representa

Operadores

Operador	Método
a++	a.inc(b)
a--	a.dec(b)
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)
a..b	a.rangeTo(b)
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b)

```
fun main() {
 var a = 2
 var b = 4
 println(a+b)
 println(a.plus(b))
}
```

saída:
6
6

Operadores podem ser usados na sua forma mais conhecida ou como o método que representa

Operadores

- Operador de chamada segura “?”
  - Em Kotlin existe um operador que trata “nulables”. Evitando os terríveis NullPointerException que acontecem normalmente em Java.

```
fun main(args: Array<String>){
        fun strLen(value: String?) = value?.length
        println(strLen(null))
}
```

Operadores

- Operador de chamada segura “?”
  - Em Kotlin existe um operador que trata “nulables”. Evitando os terríveis NullPointerException que acontecem normalmente em Java.

```
fun main(args: Array<String>){
        fun strLen(value: String?) = value?.length
        println(strLen(null))
}
```

saída:
null

Sobrecarga

- Sobrecarga de função:

```
class Teste{
    fun a(i: Int) = i
    fun a(b: String) = b
}
fun main() {
        var t = Teste()
        println(t.a(2))
        println(t.a("teste de sobrecarga"))
}
```

- Sobrecarga de operadores

```
data class Point(val x: Int, val y: Int)
operator fun Point.unaryMinus() =
Point(-x, -y)

fun main() {
      val point = Point(10, 20)
       println(-point)
}
```

Sobrecarga

- Sobrecarga de função:

```
class Teste{
    fun a(i: Int) = i
    fun a(b: String) = b
}
fun main() {
        var t = Teste()
        println(t.a(2))
        println(t.a("teste de sobrecarga"))
}
```

saída:
2
teste de sobrecarga

- Sobrecarga de operadores

```
data class Point(val x: Int, val y: Int)
operator fun Point.unaryMinus() =
Point(-x, -y)

fun main() {
      val point = Point(10, 20)
       println(-point)
}
```

saída:
Point(x=-10, y=-20)

Tipos da linguagem

Tipos ~~Primitivos~~ Básicos

Tipo	Tamanho em Bits	Exemplo
Double	64	123.5, 123.5e10
Float	32	123.5f , 123.5F
Long	64	123L
Int	32	123 , 0x0F , 0b00001011
Short	16	123
Byte	8	123
ULong	64	255u
UInt	32	255u
UShort	16	255u
UByte	8	255u
Char	16 (UTF-16)	‘a’ , '\uFF00'
Boolean	8	true

Tipos ~~Primitivos~~ Básicos

```
fun main(args: Array<String>){
   val inteira : Int = 4
   val long : Long

   long = inteira
}
```

Type mismatch.
Required: Long
Found: Int

Kotlin não permite polimorfismo de coerção para atribuir valores a variáveis e constantes.

```
fun main(args: Array<String>){
   val inteira : Int = 4
   val long : Long

   long = inteira.toLong()
}
```

Caso o programador deseje transferir o valor de uma variável para uma de um outro tipo, é preciso fazer casting explícito.

Coleções de tipos kotlin.collections

Interfaces:

AbstractCollection	*AbstractIterator*	*AbstractList*	*AbstractMap*
AbstractMutableCollection	*AbstractMutableList*	*AbstractMutableMap*	*AbstractMutableSet*
AbstractSet	*Collection*	*Iterable*	*List*
Map	*Set*		

Coleções de tipos kotlin.collections

ArrayList	*BooleanIterator*	*ByteIterator*	*CharIterator*	*DoubleIterator*
FloatIterator	*Grouping*	*HashMap*	*HashSet*	*IndexedValue*
LinkedHashSet	*IntIterator*	*Iterator*	*LinkedHashMap*	*ListIterator*
LongIterator	*MutableCollection*	*MutableIterable*	*MutableIterator*	*MutableList*
MutableListIterator	*MutableMap*	*MutableSet*	*RandomAccess*	*ShortIterator*
UByteIterator	*UIntIterator*	*ULongIterator*	*UShortIterator*	

Tipos Compostos

Array
Vetores em Kotlin são dados pela classe Array.
Para criar um Array, usamos a função arrayOf( )

arrayOf(1, 2, 3) → [1, 2, 3]

Os tipos básicos possuem Arrays sem overhead de boxing e unboxing:

Tipo de Array	Construtor do Array
ByteArray	byteArrayOf ( )
ShortArray	shortArrayOf( )
IntArray	intArrayOf( )
DoubleArray	doubleArrayOf( )
FloatArray	floatArrayOf( )

Tipos Compostos

String

Strings são imutáveis.

```
fun main(args: Array<String>){
    val text = "abc"
    text[0] = 'a'
}
```

Unresolved reference.

Pode usar o símbolo $ para imprimir expressões e variáveis numa String.

```
fun main(args: Array<String>){
    val text = """
        for (c in "str")
        |print(c)
    """.trimMargin()
    println(text)
}
```

saída:

```
    for (c in "str")
print(c)
```

```
fun main(args: Array<String>){
    val s = "abc" + 1
    val price = """${'$'}9.99
"""
    print(price)
    println("$s.length is ${s.length}")
}
```

saída:

$9.99
abc1.length is 4

Passagem de Parâmetros

```
fun imprimeSaudacao(saudacao: String = "Olá!",
                    mensagem: String = "Sejam bem vindos",
                    nome : String = "Visitantes")
        = println("$saudacao $mensagem, $nome!")



fun main(args: Array<String>){
   imprimeSaudacao()

   imprimeSaudacao("Oi oi!", “Desejamos um bom dia a vocês”)

   imprimeSaudacao(nome = "Turma de LP!")
}
```

Em Kotlin, parâmetros podem ter valores padrão.

A correspondência entre parâmetros formais e reais pode ser feita tanto posicional quanto por nome.

Passagem de Parâmetros

```
fun imprimeSaudacao(saudacao: String = "Olá!",
                    mensagem: String = "Sejam bem vindos",
                    nome : String = "Visitantes")
        = println("$saudacao $mensagem, $nome!")



fun main(args: Array<String>){
   imprimeSaudacao()

   imprimeSaudacao("Oi oi!", “Desejamos um bom

   imprimeSaudacao(nome = "Turma de LP!")
}
```

Em Kotlin, parâmetros podem ter valores padrão.

A correspondência entre parâmetros formais e reais pode ser feita tanto posicional quanto por nome.

Saída:
Olá! Sejam bem vindos, Visitantes!
Oi oi! Desejamos um bom dia a vocês, Visitantes!
Olá! Sejam bem vindos, Turma de LP!!

Expressões e Comandos

Função

As funções do Kotlin são cidadãos de *primeira classe* , o que significa que elas podem ser armazenadas em variáveis e estruturas de dados.

```
fun main( ){
   val imprime : (String) -> Unit = { s -> println(s) }

   imprime("Sou uma função")
}
```

saída:
Sou uma função

Funções de Extensão

As funções do Kotlin são cidadãos de *primeira classe* , o que significa que elas podem ser armazenadas em variáveis e estruturas de dados.

```
fun main() {
        fun String.isValidaParaSenha() = this.count() >= 6
        var a = "12345".isValidaParaSenha()
        var b = "asdf12".isValidaParaSenha()
    println("$a $b")
}
```

saída:
false
true

Funções lambda

Funções lambda são funções que não são declaradas, mas são passadas imediatamente como uma expressão.

```
fun main(args: Array<String>) {
  val sum: (Int, Int) -> Int = { x, y -> x + y }

  val imprime: (Int) -> Unit = { p-> println(p)}

  imprime(sum(4, 3))

}
```

saída:
7

Funções

Você poderia imaginar o que esse código faria.

```
fun main() {
[redacted]

soma(1)(1)
}
```

Funções

- Over-Kotlin(uso exagerado da linguagem) pode atrapalhar no seu código

```
fun main() {
val espia: (x:Int) -> Unit = { println("sou uma função escondida") }
fun soma(x:Int = 1) = espia

soma(1)(1)
}
```

saída:
sou uma função escondida

Funções Infixadas

- Regras:

  Eles devem ser funções-membro ou funções de extensão

  Eles devem ter um único parâmetro

  O parâmetro não deve aceitar um número variável de argumentos e não deve ter nenhum valor padrão

```
fun main() {
    infix fun Int.murilo(x:Int) = (this * 2)+x
    var x = 1 murilo 2 // 1.murilo(2)
    println(x)
}
```

saída:
4

Precedência de Operadores

Precedência	Operadores	Símbolos
Alta	Pós fixado	++, --, ., ?., ?
	Pré fixado	-, +, ++, --, !, labelDefinition
	Tipo RHS	:, as, as?
	Multiplicativo	*, /, %
	Aditivo	+, -
	Intervalo	..
	Função Infixada	SimpleName
	Elvis	?:
	Checagem de Nomes	in, !in, is, !is
	Comparação	<, >, <=, >=
	Igualdade	==, \!==
	Conjunção	&&
	Disjunção	||
Baixa	Atribuição	=, +=, -=, *=, /=, %=

Curto circuito

```
fun main(args:Array<String>) {
    var i=2
    var j=9
    if(i > 1 || j++ >= 9){
        println(j)
    }
        if(i < 1 || j++ >= 9){
        println(j)
   }
}
```

- Em kotlin alguns operadores como || e && apresentam curto circuito

saída:
9
10

Retorno

```
fun main() {
        var a : MutableList<Int> = mutableListOf()
        var n = a.add(1)
        a.addAll(listOf(1,2,3,4,5))
        println("- Sou uma lista: $a . \n- Também sou uma lista: $n.")
}
```

saída:
- Sou uma lista: [1, 1, 2, 3, 4, 5]
- Também sou uma lista: true

Avaliação de expressões

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}
fun main() {
    println("Ainda não computado")
    println(lazyValue)
    println(lazyValue)
}
```

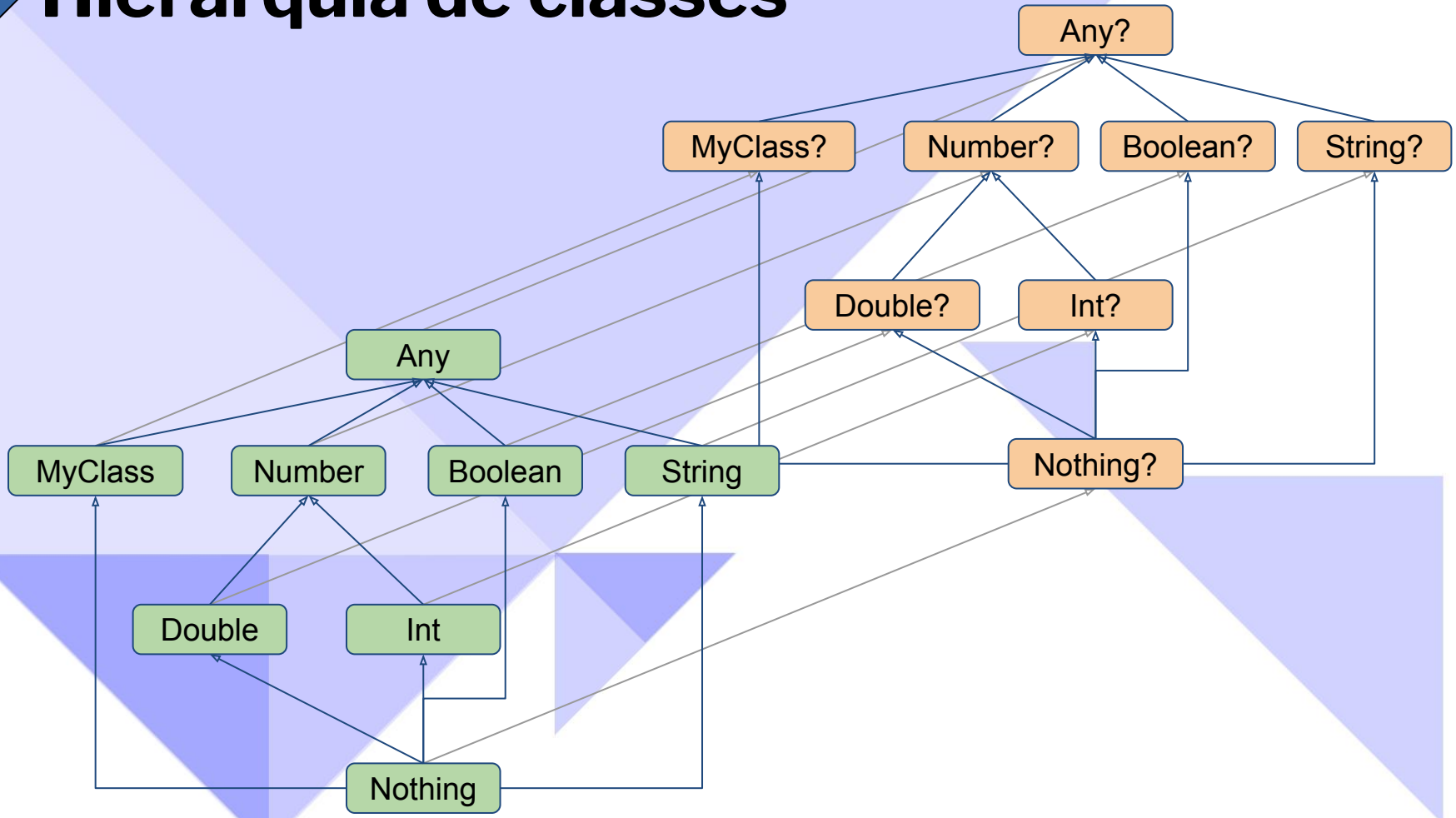
saída:

```
Ainda não computado
computed!
Hello
Hello
```

- Kotlin pode ter avaliação tardia (lazy evaluation), quando utilizado o paradigma funcional
  - Mas esse tipo de avaliação deve ser explicitada.

Modularização

Hierarquia de classes



Tipagem

```
fun main(args: Array<String>){
    val tipada = 4

    println(tipada is Int)
}
```

Saída:
true

Inferência de Tipo - Kotlin faz inferência de tipo.

```
fun main(args: Array<String>){
    var inteira = 4

    inteira = 8L
}
```

The integer literal does not conform to the expected type Int

Fortemente Tipada - Não é possível alterar o tipo da variável.

Herança

```
open class Base(p: Int) {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived(p: Int) : Base(p) {
    override fun v() {
        super.v( )
        //mais código
    }
}
```

```
open class Foo {
    open val x: Int get() { ... }
}

class Bar1 : Foo() {
    override var x: Int = ...
}
```

Herança - Uma classe herda somente de uma classe base.

Amarração tardia - Executa sempre o métodos da classe mais especializada.

Sobrescrita de propriedades - Propriedades redeclaradas devem ser do mesmo tipo.
Pode redeclarar uma val como var.

Classes Abstratas

```
abstract class Person(val age: Int = 40)  {

   fun displayAge() {
      println("My Age is $age")
   }

   abstract fun displayJob()
}
```

Classes Abstratas - Não podem ser instanciadas.

Podem ter métodos não abstratos.

Métodos abstratos devem ser marcados com abstract e deverão ser substituídos em suas subclasses.

```
fun main(args:Array<String>){
   var p1 = object : Person(15) {
         override fun displayJob() {
                  println("I do not have a job!")
         }
    }

   p1.displayAge()
   p1.displayJob()
}
```

Saída:
My Age is 15
I do not have a job!

Enum e Sealed Classes

```
sealed class Operation {
   class Add(val value: Int) : Operation()
   class Substract(val value: Int) : Operation()
   class Multiply(val value: Int) : Operation()
   class Divide(val value: Int) : Operation()
}
fun execute(x: Int, op: Operation) = when (op) {
   is Operation.Add -> x + op.value
   is Operation.Substract -> x - op.value
   is Operation.Multiply -> x * op.value
   is Operation.Divide -> x / op.value
}

fun main(args: Array<String>){
   val op = Operation.Add(8)

   println(execute(3, op))
}
```

Saída:
11

```
enum class Direction{
        NORTH, SOUTH, WEST, EAST
}
```

Sealed Classes - São como enums com Super Poderes!

Declaram quais classes podem ser subclasses dela e apenas elas podem.

Geralmente é usada como um enum de classes por uma função de execução.

Interfaces

```
interface MinhaInterface {
  val test: String

  fun foo() :String
  fun hello() = "Hello there"
}

class InterfaceImp : MinhaInterface, Runnable{
      override fun run() {
            println(this.foo() +  " from ${this.test}")
      }

  override var test = "Kotlin"
  override fun foo()
      = "Lol " + super.hello()
}

fun main(args: Array<String>){
  val imp = InterfaceImp()
  Thread(imp).run()
}
```

Saída:
Lol Hello there from Kotlin

Interfaces - Podem ter métodos implementados e declarar atributos.

Uma classe pode implementar mais de uma interface.

Generics

```
class Box<T>(t: T) {
    var value = t

   fun unbox() : T = value
}

fun main( ){
    val box : Box<Int>  = Box(42)
    println(box.unbox())
}
```

Saída:
42

Polimorfismo paramétrico - Permite criar classes que podem receber qualquer tipo de parâmetro.

Extensões

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1]
    this[index1] = this[index2]
    this[index2] = tmp
}

fun main( ){
    var lista = mutableListOf(1,2,3,4)
    lista.swap(1,2)
    lista.forEach({p -> print("$p ")})
}
```

Saída:
1 3 2 4

Extensões - Permitem criar novos métodos para classes já existentes,

Exceções

Exceções

```
@Throws(IOException::class)
fun readFile(name: String): String {...}
```

Exceções – Toda exceção em Kotlin é não checada.

```
//Será traduzido para:

String readFile(String name) throws IOException {...}
```

Só é necessário declarar que um método lança uma exceção caso o módulo Kotlin venha a ser chamado por uma linguagem que exija, como Java.

Exceções

```
/*Usando variáveis nulas '?'*/

var b: String? = "abc"
b = null // ok
print(b)

// Só é feito se b não for null
println(b?.length)

//Elvis
val l = b?.length ?: -1

//Assumindo riscos
val l = b!!.length

//Safe casting
val aInt: Int? = a as? Int
```

Null-safety – Kotlin é considerada null-safe.

Elvis – Se a variável a esquerda não for null, retorna ela, caso contrário retorna a expressão a direita

Try-Catch e Finally

```
/*Usando try catch '?'*/
try {
   // código
}
catch (e: SomeException) {
   // tratamento
}
finally {
   // bloco opcional e sempre executado
}

//Try é uma expressão!!
val a: Int? = try { parseInt(input) }
    catch (e: NumberFormatException) { null }

//Throw é do tipo Nothing
val s = person.name ?:
     throw IllegalArgumentException("Name required")
println(s)
```

Throw como Nothing – Se a variável receber throw, o código depois dele não executa.

Concorrência

Threads

```
fun main(args: Array<String>){

  val t1 = thread(start = true) {
     Thread.sleep(1000)
     println("Thread 1")
  }

  val t2  = thread(start = true) {
     Thread.sleep(500)
     println("Thread 2")
  }

}
```

Threads - Apesar do incentivo ao uso de Co-rotinas, Threads ainda são uma boa pedida para tarefas que pedem poder de processamento(CPU bound).

Threads

```
fun main(args: Array<String>){

  val t1 = thread(start = true) {
     Thread.sleep(1000)
     println("Thread 1")
  }

  val t2  = thread(start = true) {
     Thread.sleep(500)
     println("Thread 2")
  }

}
```

Saída:
Thread 2
Thread 1

Threads - Apesar do incentivo ao uso de Co-rotinas, Threads ainda são uma boa pedida para tarefas que pedem poder de processamento(CPU bound).

Co-rotinas

```
import kotlinx.coroutines.*

fun main() {
        val job = GlobalScope.launch {
                delay(1000L)
                println("World!")
         }
        println("Hello,")
        runBlocking {
                job.join()
        }
}
```

Coroutines - Threads Leves

Quando suspendem a operação não bloqueia a thread.

Pode voltar a executar em outra thread.

Co-rotinas

```
import kotlinx.coroutines.*

fun main() {
        val job = GlobalScope.launch {
                delay(1000L)
                println("World!")
         }
        println("Hello,")
        runBlocking {
                job.join()
        }
}
```

Saída:
Hello, World!

Coroutines - Threads Leves

Quando suspendem a operação não bloqueia a thread.

Pode voltar a executar em outra thread.

Co-rotinas

```
import kotlin.system.measureTimeMillis

fun meaninglessCounter(): Long {
  var counter = 0L
  for (i in 1..10_000_000_000) {
    counter += 1L
  }
  return counter
}

fun main(args: Array<String>) {

  var time = measureTimeMillis {
    val one = meaninglessCounter()
    val two = meaninglessCounter()
    println("The answer is ${one + two}")
  }
  println("Sequential completed in $time ms")
}
```

```
import kotlinx.coroutines.*
import kotlin.system.measureTimeMillis

fun meaninglessCounter(): Long { … }

fun main(args: Array<String>) {
   var time2 = measureTimeMillis {

        runBlocking {
                val one = async { counter() }
                val two = async { counter() }

               println(
          "The answer is ${one.await() + two.await()}")
        }
   }
   println("Concurrent completed in $time2 ms")
}
```

Co-rotinas

```
import kotlin.system.measureTimeMillis

fun meaninglessCounter(): Long {
  var counter = 0L
  for (i in 1..10_000_000_000) {
    counter += 1L
  }
  return counter
}

fun main(args: Array<String>) {

  var time = measureTimeMillis {
    val one = meanin
    val two = meani
    println("The ans
  }
  println("Sequential completed in $time ms")
}
```

Saída:
The answer is 20000000000
Sequential completed in 8285 ms

```
import kotlinx.coroutines.*
import kotlin.system.measureTimeMillis

fun meaninglessCounter(): Long { … }

fun main(args: Array<String>) {
   var time2 = measureTimeMillis {

        runBlocking {
                val one = async { counter() }
                val two = async { counter() }

                println(
          "The answ
        }
   }
   println("Concurrent completed in $time2 ms")
}
```

Saída:
The answer is 20000000000
Concurrent completed in 7665 ms

Igual a JAVA

Igual a Java

- Gerenciamento de Memória
- Não permite acesso à endereços reais
- Acesso a Memória Secundária
- Comentários no código
- Mecanismo Contra Colisão de Nomes
- Inclusão de bibliotecas
- JVM

Avaliação da Linguagem

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Depende
Portabilidade	Não	Sim	Sim
Método de Projeto	Estruturado e OO	OO	Estruturado, OO e Funcional
Evolutibilidade	Parcial	Sim	Parcial

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	
Aprendizado	Não	Não	
Eficiência	Sim	Parcial	
Portabilidade	Não	Sim	
Método de Projeto	Estruturado e OO	OO	Estruturado, OO e Funcional
Evolutibilidade	Parcial	Sim	Parcial

- Não lida diretamente com o hardware

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	
Eficiência	Sim	Parcial	
Portabilidade	Não	Sim	
Método de Projeto	Estruturado e OO	OO	Funcional
Evolutibilidade	Parcial	Sim	Parcial

- Null-Safety
- Coletor de Lixo
- Fortemente Tipada

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	
Portabilidade	Não	Sim	
Método de Projeto	Estruturado e OO	OO	
Evolutibilidade	Parcial	Sim	Parcial

- Multiparadigma
- Várias sintaxes
- Interoperabilidade com mais de uma linguagem

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Depende
Portabilidade	Não	Sim	
Método de Projeto	Estruturado e OO	OO	
Evolutibilidade	Parcial	Sim	

- Pode ser compilada, interpretada e híbrida
- Contém seu próprio coletor
- Null-Safety

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Depende
Portabilidade	Não	Sim	Sim
Método de Projeto	Estruturado e OO	OO	
Evolutibilidade	Parcial	Sim	

- Android
- Pode ser compilada, interpretada(navegador) e híbrida(JVM)

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Depende
Portabilidade	Não	Sim	Sim
Método de Projeto	Estruturado e OO	OO	Estruturado, OO e Funcional
Evolutibilidade	Parcial	Sim	

- Pode ou Não ter funções em classes
- Lambda

Avaliação

Critérios Gerais	C++	Java	Kotlin
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Sim
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	
Portabilidade	Não	Sim	
Método de Projeto	Estruturado e OO	OO	
Evolutibilidade	Parcial	Sim	Parcial

- Interoperabilidade
- Promove encapsulamento
- Linguagem Concisa

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de classes, serialização, JSON
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por valor e por cópia	Lista variável, por valor e por cópia. Posicional e por nome.

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	
Escopo	Sim	Sim	
Expressões e Comandos	Sim	Sim	
Tipos Primitivos e Compostos	Sim	Sim	Sim
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de classes, serialização, JSON
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por valor e por cópia	Lista variável, por valor e por cópia. Posicional e por nome.

- Muitas bibliotecas disponíveis
- Open source

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	
Expressões e Comandos	Sim	Sim	
Tipos Primitivos e Compostos	Sim	Sim	
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de classes, serialização, JSON
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por valor e por cópia	Lista variável, por valor e por cópia. Posicional e por nome.

- Interoperabilidade

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	
Tipos Primitivos e Compostos	Sim	Sim	
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca d classes, serialização	serialização, JSON
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por valor e por cópia	Lista variável, por valor e por cópia. Posicional e por nome.

- Escopo estático e aninhado

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca d classes, serializaçã	
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por v e por cópia	e por cópia. Posicional e por nome.

- Ampla variedades de expressões e comandos

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serializaçã	
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por v e por cópia	

- Não possui tipos primitivos, mas possui os tipos basicos.

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	Biblioteca de classes, serialização, JSON
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por val e por cópia	

- Grande Variedades de persistência de Dados

Avaliação

Critérios Gerais	C++	Java	Kotlin
Reusabilidade	Sim	Sim	Sim
Integração	Sim	Parcial	Sim
Escopo	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	
Persistência dos Dados	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	
Passagem de Parâmetros	Lista variável, default por valor	Lista variável, por valor e por cópia	Lista variável, por valor e por cópia. Posicional e por nome.

- Varargs
- Entrada constante

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	Estática/Dinâmica
Polimorfismo	Todos	Todos	Parcial
Exceções	Parcial	Sim	Parcial
Concorrência	Não (biblioteca de funções)	Sim	Sim

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	
Sistema de tipos	Parcial	Sim	
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	
Polimorfismo	Todos	Todos	Parcial
Exceções	Parcial	Sim	Parcial
Concorrência	Não (biblioteca de funções)	Sim	Sim

- Coletor de lixo

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	
Polimorfismo	Todos	Todos	
Exceções	Parcial	Sim	Parcial
Concorrência	Não (biblioteca de funções)	Sim	Sim

- mecanismos de classes e pacotes

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	
Polimorfismo	Todos	Todos	
Exceções	Parcial	Sim	
Concorrência	Não (biblioteca de funções)	Sim	Sim

- Fortemente Tipada

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	Estática/Dinâmica
Polimorfismo	Todos	Todos	
Exceções	Parcial	Sim	
Concorrência	Não (biblioteca de funções)	Sim	

- Possui inferência de Tipos
- Amarração Tardia

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	Estática/Dinâmica
Polimorfismo	Todos	Todos	Parcial
Exceções	Parcial	Sim	
Concorrência	Não (biblioteca de funções)	Sim	

- Exceção a Coerção

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	Estática/Dinâmica
Polimorfismo	Todos	Todos	Parcial
Exceções	Parcial	Sim	Parcial
Concorrência	Não (biblioteca de funções)	Sim	

- Não possui exceções checadas

Avaliação

Critérios Gerais	C++	Java	Kotlin
Gerenciamento de memória	Programador	Sistema	Sistema
Encapsulamento e proteção	Sim	Sim	Sim
Sistema de tipos	Parcial	Sim	Sim
Verificação de tipos	Estática/Dinâmica	Estática/Dinâmica	
Polimorfismo	Todos	Todos	
Exceções	Parcial	Sim	
Concorrência	Não (biblioteca de funções)	Sim	Sim

- Coroutines
- Thread
- Semáforos
- Monitores

Trabalho

Referência

Referências

- https://kotlinlang.org/docs/reference/
- https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html
- https://movile.blog/programacao-assincrona-em-kotlin-dissecando-coroutines/
- https://antonioleiva.com/sealed-classes-kotlin/
- https://www.programiz.com/kotlin-programming/
- https://kotlinlang.org/foundation/kotlin-foundation.html