Seminário de Linguagem de Programação

Kotlin

Integrantes: Gabriel Valdino, Luiz Felipe Boina, Matheus Rizzi e Renan Bottacine

Introdução

Introdução

- Teve início em 2010, lançamento em 2016 e última release há 3 meses (1.3);
- Iniciativa da JetBrains em trazer novos conceitos a linguagem Java;
- Foco em ser concisa, expressiva, ferramental e na interoperabilidade;
- Multiparadigma (Funcional e Orientada à objetos);
- Inspirada em Java, Scala, C# e Groovy;
- Plataformas alvo:
  - JVM;
  - Js;
  - Android;
- Feita para uso interno mas diversas empresas utilizam, tais como American Express GitHub, Netflix, NBC News Digital, Uber

Introdução

- Kotlin REPL;
- Híbrida (na JVM);
- HelloWorld.kt;
- Compilando: kotlinc HelloWorld.kt -include-runtime -d HelloWorld.jar ;
- Executando: java -jar HelloWorld.jar;

```
fun main(args : Array<String>) {
 println("Hello, World!")
}
```

Convenções

- Segue as convenções de Java;
- camelCase para métodos e atributos;
- UpperCase para tipos, nomes de classes e objetos;
- Ponto e vírgula é opcional (requerido somente para separar atributos de funções em enum class);
- caminho reverso dos pacotes;
- múltiplas classes por arquivo;
- pacotes não precisam ser idêntico ao caminho;

Amarrações

Sintaxe

- Pacotes e importações

```
// everything in 'org.example' becomes accessible
import org.example.*
import org.example.Message // Message is accessible
// testMessage stands for 'org.test.Message'
import org.test.Message as testMessage
```

Sintaxe

- Funções

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
fun sum(a: Int, b: Int) = a + b
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is
${a + b}")
}
```

Sintaxe

- Variáveis

```
val a: Int = 1  // Inicializada na
definiçõa
val b = 2       // Tipo `Int` é inferido
// O tipo é necessário quando não
inicializada
val c: Int
c = 3           // Inicialização tardia
var x = 5       // Tipo `Int` é inferido
x += 1
```

Sintaxe

- Comentários
  - Possui KDoc similar ao JavaDoc

```
// Comentário de uma linha

/* Comentário
de múltiplas linhas */

/* Comentário começa aqui
/* Comentário misturado */
Termina aqui. */
```

Sintaxe

- Condicionais

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun maxOf(a: Int, b: Int) = if (a > b) a
else b
```

```
fun describe(obj: Any): String =
    when (obj) {
        1           -> "One"
        "Hello"     -> "Greeting"
        is Long     -> "Long"
        !is String -> "Not a string"
        else        -> "Unknown"
    }
```

Sintaxe

- Valores nulos

```
fun parseInt(str: String): Int? {
    // ...
}
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    if (x != null && y != null) {
        /* x and y are automatically cast
        to non-nullable after null check */
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(b!!.length)
println(a?.length) // Unnecessary
safe call
```

Sintaxe

- Checagem de tipo

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```

Sintaxe

- Loops

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
for (item in items) {
   println(item)
}
while (index < items.size) {
   println("item at $index is ${items[index]}")
   index++
}
```

Identificadores

- Hard keywords que sempre são interpretados como keywords;
- Soft e Modifier keywords podem ser usadas como identificadores dependendo do contexto

Kotlin keywords List

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		

Amarração

- Estaticamente tipada para JVM;

```
val dyn: dynamic = … ( para Web )
```

- Escopo aninhado e estático;
- Inferência de tipos durante a compilação;

Variáveis e Constantes

Escopo e Nomeação

- Variáveis só existem dentro do escopo que foram declaradas. Portanto, uma variável declarada dentro de um loop só existe nesse loop
- Pela convenção, nomes de variáveis devem seguir **lowerCamelCase**
- Em geral, os identificadores de variáveis podem consistir em letras, dígitos e underscores e não podem começar com um dígito.
- Nomes entre crases (`), são identificadores válidos

```
val `Eu não acredito que isso não dá erro!` =  "Pode acreditar"
```

Declaração

- Em Kotlin declaramos as variáveis informando as palavras reservadas **val** ou **var**, o nome da variável, o tipo e o valor

```
val nome: String = "Renan"
var idade: Int = 30
```

- Pode-se omitir o tipo da variável, assim ele é inferido pelo valor atribuído

```
val nome = "Renan"
var idade = 30
```

Declaração

- Variáveis declaradas com **val** são read-only (apenas leitura), depois que é atribuído um valor inicial ela não pode mudar.

```
val nome: String = "Renan"
nome = "Luiz"
// Error: Val cannot be reassigned
```

- Variáveis declaradas com **var** são referências mutáveis

```
var idade: Int = 30
idade = 40
println(idade)        // Saída: 40
```

Variáveis de Nível Elevado

- Kotlin suporta a declaração de variáveis de nível elevado (top-level), ou seja, fora de uma função ou classe:

```
val preco = 2.99f
fun main() {
    val desconto = 0.5f
        println(preco - preco * desconto)    //Saída: 1.495
}
```

- A variável de nível elevado “preco” pode ser utilizada em qualquer lugar no projeto, inclusive em outros arquivos, enquanto a variável
local “desconto ”pode ser utilizada somente dentro da
função onde foi declarada

Variáveis de Nível Elevado

- Caso uma variável local, declarada dentro de uma função, tenha o mesmo nome de uma variável de nível elevado, a variável de nível elevado será sombreada dentro da função.

```
val preco = 2.99f
fun main() {
    val preco = 4.99f
      val desconto = 0.5f
      println(preco - preco * desconto)     //Saída: 2.495
}
```

Variáveis estáticas

- não possui “static” como em Java mas permite o uso de “companion object” para produzir o efeito semelhante;

```
class Car(val horsepowers: Int) {
    companion object Factory {
        val cars = mutableListOf<Car>()

        fun makeCar(horsepowers: Int): Car {
            val car = Car(horsepowers)
            cars.add(car)
            return car
        }
    }
}
val car = Car.makeCar(150)
println(Car.Factory.cars.size)
```

Endereçamento de memória

- Kotlin rodando na JVM não permite endereçamento de memória, entretanto rodando nativo no Android é possível;
- Kotlin não possui suporte a variáveis anónimas.

```
// Kotlin Native v0.5
import kotlinx.cinterop.*

fun main(args: Array<String>) {
    val intVar = nativeHeap.alloc<IntVar>()
    intVar.value = 42
    with(intVar) { println("Value is $value, address is $rawPtr") }
    nativeHeap.free(intVar) //Value is 42, address is 0xc149f0


}
```

Coletor de lixo

- Utiliza o que está implementado na JVM (similar a Java com sistema de gerações), geralmente descarta os objetos sem referência.

Memória

- Memória primária segue o padrão de pilha e monte como em Java;
- Memória secundária para ações de I/O e serialização basta importar as classes de Java para tal.

```
import java.io.Serializable

class Foo(val someField:Int): Serializable {
    companion object {
        private const val serialVersionUID = 20180617104400L
    }
}
```

Valores e Tipos de Dados

Tipagem Estática vs. Dinâmica

- Por ser uma linguagem com tipagem estática, Kotlin ainda precisa interoperar com ambientes sem tipo, como o ecossistema JavaScript.
- Para facilitar esses casos de uso, o tipo **dynamic** está disponível na linguagem

```
val dyn: dynamic = ...
```

- O tipo **dynamic** não é suportado no código destinado à JVM
- O tipo **dynamic** desliga o checador de tipos do Kotlin

Tipos Básicos - Números

Tipo	Tamanho (bits)	Menor Valor	Maior Valor
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
Long	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

```
val int = 1
val long = 1L
val byte: Byte = 1
```

Tipos Básicos - Números

Tipo	Tamanho (bits)	Bits Significativos	Bits do Expoente	Dígitos Decimais
Float	32	24	8	6-7
Double	64	53	11	15-16

```
val float_pi = 3.14f                  //Float
val double_pi = 3.14                  //Double
val notaçãoCientífica = 1.235e10      //Double
```

Tipos Básicos - Números

- Representações:

  Decimal: 25
  Hexadecimal: **0x**19
  Binário: **0b**11001
  Octal não é suportado

- Pode-se usar underscores para maior legibilidade:

```
val bytes = 0b11010010_01101001_10010100_10010010
val long = 1234_5678_9012_3456L
val hex = 0xFF_EC_DE_5E
```

Tipos Básicos - Números

- **NaN** é considerado igual a ele mesmo
- **NaN** é considerado maior que qualquer outro elemento, incluindo POSITIVE_INFINITY
- **-0.0** é considerado menor que 0.0

```
import
kotlin.Double.Companion.POSIT
IVE_INFINITY
import
kotlin.Double.Companion.NaN

val nan = NaN
val inf = POSITIVE_INFINITY

println(nan.compareTo(nan))
println(nan.compareTo(inf))
println(0.0.compareTo(-0.0))

//Saída:  0
          1
          1
```

Tipos Básicos - Characters

- São representados com aspas simples

```
var c = 'a'
```

- Caracteres especiais são representados com barra invertida

\t	Inserts tab	\’	Inserts single quote character
\b	Inserts backspace	\”	Inserts double quote character
\n	Inserts newline	\\	Inserts backslash
\r	Inserts carriage return	\$	Inserts dollar character

- Para outros caracteres utiliza-se a sintaxe Unicode

```
var c = '\uFF00'
```

Tipos Básicos - Characters

- Diferente de linguagens como C, não podem ser tratados diretamente como números

```
fun check(c: Char) {
    if (c == 1) {//...} //ERROR: Operator '==' cannot
                        be applied to 'Char' and 'Int'
}
```

- Podem ser convertidos explicitamente para inteiros

```
fun check(c: Char) {
    if (c.toInt == 1) {//...}
}
```

Tipos Básicos - Booleans

- Só assumem os valores **true** ou **false**
- Operadores em Booleanos:
  - || - disjunção
  - && - conjunção
  - ! - negação
- || e && são operadores com curto-circuito

Tipos Básicos - Array

- Arrays em Kotlin são **Invariantes**
- ```
  var vetor = Array(5) { i -> (i * i).toString() }
  //[“0”, ”1”, ”4”, ”9”, ”16”]
  ```
- **arrayOf** permite criar array com tipos diferentes

  ```
  var outroVetor= arrayOf ("1", 2, '3', 4.0, 5.0f, 6L)
  ```
- Pode-se fazer operações **get** e **set** com o operador **[ ]**

  ```
  outroVetor[0] = 7.0
  println(outroVetor[0]) //printa: 7.0
  ```
  ```

Tipos Básicos - Array

- Kotlin possui classes especializadas para representar arrays de tipos primitivos: intArrayOf, doubleArrayOf, charArrayOf, ...

```
// Array de int com tamanho 5 e valores [0, 0, 0, 0, 0]
val arr = IntArray(5)
// Array de int com tamanho 5 e valores [42, 42, 42, 42, 42]
val arr2 = IntArray(5) { 42 }
// Array de int com tamanho 5 e valores [0, 2, 4, 6, 8]
var arr3 = IntArray(5) { it * 2 }
```

Tipos Básicos - Unsigned integers

Tipo	Tamanho (bits)	Intervalo
UByte	8	0 - 255
UShort	16	0 - 65535
UInt	32	0 - 2^{32} - 1
ULong	64	0 - 2^{64} - 1

```
val s: UShort = 1u   // UShort
val l: ULong = 1u    // ULong
val a = 1UL          // ULong
val a1 = 42u         // UInt é o padrão
```

Tipos Básicos - Strings

- São imutáveis
- Os elementos da string são **char** e podem ser acessados pelo operador **[ ]**

```
var str = "Olá"
println(str[0])      //Saída: O
```

- Strings podem ser iteradas com **for**

```
for (c in str) { println(c) }
```

- Podem ser concatenadas com o operador **+**

```
str = str + ", Mundo!"
println(str)
//Saída: Olá, Mundo!
```

Tipos Básicos - Strings

- Escaped Strings: pode-se utilizar caracteres especiais **\n, \t, ** ...

```
val s = "Olá, Mundo!\n"
```

- Raw String: é delimitada por três aspas duplas (“”””) e não contém caracteres especiais

```
val text = """
    for (c in "foo")
        print(c)
    """
```

Tipos Básicos - Strings

- Pode-se remover os espaços em branco com a função **trimMargin()** e marcando o início das linhas com uma barra vertical (|)

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

Tipos Básicos - Strings

- Strings podem conter pedaços de código que são avaliados e cujos resultados são concatenados na string. É utilizado o caracter **$** no início das expressões

```
val i = 10
println("i = $i")
val s = "abc"
println("$s.length is ${s.length}")

//Saída:  i = 10
          abc.length is 3
```

Classes Enum

- Possuem características de classe
- Cada constante enum é um objeto
- Como cada enum é uma instância da classe enum, eles podem ser inicializados

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
fun main() {
    var c = Color.RED
    println(c.name)
    println(c.ordinal)
    println(c.rgb)
}
//Saída:    RED
            0
            16711680
```

Intervalos

- Pode-se criar intervalos de valores utilizando a função **rangeTo()** e o operador **…**

```
var intervalo = 0..10  //equivale à 0<= intervalo <=10
```

- Pode ser utilizados em expressões condicionais junto com **in** ou **!in**

```
var num = 5
if(num in 0..10)          //se num >= 0 && num <= 10
    println(num)
```

- Intervalos de tipos Integral (IntRange, LongRange, CharRange) podem ser iterados

```
for (i in 'a'..'e') print(i)   //Saída: abcde
for (i in 4 downTo 1) print(i) //Saída: 4321
```

Intervalos

- Também é possível iterar sobre um intervalo com uma passo arbitrário

```
for (i in 1..8 step 2) print(i)
//Saída: 1357
```

- Utiliza-se **until** para iterar sobre um intervalo que não inclui o último elemento

```
for (i in 1 until 10) print(i)
//Saída: 123456789
```

Checagem de Tipos

- A checagem de tipo são feitas com os operadores **is** e **!is**
- Operador **is** compara o tipo da variável e retorna um booleano **true** se os tipos combinam. O operador **!is** retorna o inverso do operador **is**

```
if (forma is Circulo) {
    print("é um Circulo")
} else if (forma is Quadrado) {
    print("é um Quadrado")
} else if (forma is Retangulo) {
    print("é um Retângulo")
}
```

Expressões e Comandos

Operadores Unários

Expressão	Traduzido para
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

```
var x = 2
println(+x)          //Saída: 2

var y = 2
println(-y)
//Saída: -2
println(y.unaryMinus())
//Saída: -2

var z = true
println(z.not())     //Saída: false
println(!z)          //Saída: false
println(!!z)         //Saída: true
```

Operadores Unários

Expressão	Traduzido para
++a / a++	a.inc()
--a / a--	a.dec()

```
var x = 2
println(++x)        //Saída: 3
println(x.inc())    //Saída: 3

var y = 2
println(y++)        //Saída: 2
println(y)          //Saída: 3

var x = 2
println(--x)        //Saída: 1
println(x.dec())    //Saída: 1

var y = 2
println(y--)    //Saída: 2
println(y)      //Saída: 1
```

Operadores Binários

Expressão	Traduzido para
a + b	a.plus(b)
a - b	a.minus(b)

```
println(2.5 + 7.8)      //Saída: 10.3
println(2.5.plus(7.8)) //Saída: 10.3
println("Olá, " + "Mundo!")
//Saída: Olá, Mundo!
println("Olá, ".plus("Mundo!"))
//Saída: Olá, Mundo!
println('a' + 1)        //Saída: b
println('a'.plus(1))    //Saída: b

println(2.5 - 7.8) //Saída:-5.3
println(2.5.minus(7.8)) //Saída:-5.3
println('b' - 1)         //Saída: a
println('b'.minus(1))    //Saída: a
```

Operadores Binários

Expressão	Traduzido para
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b)

```
println(2.2 * 9)
//Saída: 19.8
println(2.2.times(9))
//Saída: 19.8

println(19.8 / 2.2)
//Saída: 9.0
println(19.8.div(2.2))
//Saída: 9.0

println(3 % 2)     //Saída: 1
println(3.rem(2)) //Saída: 1
```

Operadores Binários

Expressão	Traduzido para
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)

```
var x = 10
x += 5
println(x)          //Saída: 15

var x = 10
x -= 5
println(x)          //Saída: 5

var x = 10
x *= 5
println(x)          //Saída: 50
```

Operadores Binários

Expressão	Traduzido para
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)

```
var x = 10
x /= 5
println(x)          //Saída: 2

var x = 10
x %= 5
println(x)          //Saída: 0
```

Operadores Binários

Expressão	Traduzido para
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

```
println(10 == 50)
//Saída: false
println(10 != 50)
//Saída: true
println(10 > 50)
//Saída: false
println(10 < 50)
//Saída: true
println(10 >= 50)
//Saída: false
println(10 <= 10)
//Saída: true
```

Operadores Binários

Expressão	Traduzido para
a in b	b.contains(a)
a !in b	!b.contains(a)
a[i]	a.get(i)
a[i] = b	a.set(i, b)

```
var x = 10
var y =arrayOf(0,1,5,8,13,20)

println(x in y)  //Saída: false

println(x !in y) //Saída: true

println( y[2] )  //Saída: 5

y[2] = x
println( y[2] )  //Saída: 10
```

Operadores Lógicos

Expressão	Traduzido para
a || b	(a)or(b)
a && b	(a)and(b)

```
var x = true
var y = false

println(x || y) //Saída: true
println(x || x) //Saída: true
println(y || y) //Saída: false

println(x && y) //Saída: false
println(x && x) //Saída: true
println(y && y) //Saída: false
```

Operadores Bit a Bit

Expressão
a.shl(b)
a.shr(b)
a.ushr(b)
a.and(b)
a.or(b)
a.xor(b)
a.inv()

```
var x = 0b1010
var y = 0b1000
println(Integer.toBinaryString(x.shl(1)))
//Saída: 10100
println(Integer.toBinaryString(x.shr(1)))
//Saída: 101
println(Integer.toBinaryString(x.ushr(1)))
//Saída: 101
println(Integer.toBinaryString(x.and(y)))
//Saída: 1000
println(Integer.toBinaryString(x.or(y)))
//Saída: 1010
println(Integer.toBinaryString(x.xor(y)))
//Saída: 10
println(Integer.toBinaryString(x.inv()))
//Saída: 11111111111111111111111111110101
```

Controle de Fluxo - Expressão If

- Em kotlin if é uma expressão, ou seja, retorna um valor
- Não há necessidade de existir operador ternário (condition ? then : else)

```
// Uso Tradicional
    var max = a
    if (a < b) max = b
```

```
// Com else
    var max: Int
    if (a > b) {
        max = a
    } else {
        max = b
    }
```

Controle de Fluxo - Expressão If

- Em kotlin if é uma expressão, ou seja, retorna um valor
- Não há necessidade de existir operador ternário (condition ? then : else)

```
// Como  Expressão
val max = if (a > b) a else b
```

```
// Como  Expressão
    val max = if (a > b) {
        print("a")
        a
    } else {
        print("b")
        b
    }
```

Controle de Fluxo - Expressão When

- when substitui o operador switch de linguagens C-like

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x não é 1 nem 2")
    }
}
```

Controle de Fluxo - Expressão When

- Também pode ser usado como expressão

```
var b = when (x) {
    1 -> {print("x == 1")
        10}
    2 -> {print("x == 2")
        20}
    else -> {print("x não é 1 nem 2")
        30}
}
```

Controle de Fluxo - Expressão When

- Se muitos casos devem ser tratados da mesma maneira, as condições podem ser combinadas com vírgulas

```
when (x) {
    0, 1 -> print("x == 0 ou x == 1")
    else -> print("x != 0 e x != 1")
}
```

Controle de Fluxo - Expressão When

- Pode-se utilizar expressões arbitrárias (não só constantes) como condições

```
var x = 1
var s = "1"
when (x) {
    s.toInt() -> print("s.toInt() == x")
    else -> print("s.toInt() != x")
}

//Saída: s.toInt() == x
```

Controle de Fluxo - Expressão When

- Pode-se verificar se o valor está ou não em um intervalo

```
    when (x) {
        in 0..10 -> print("x está no intervalo [0,10]")
        !in 20..40 -> print("x não está no intervalo [20,40]")
        else -> print("nenhuma das anteriores")
}
```

Controle de Fluxo - Expressão When

- Pode ser utilizado com cadeia de if else se nenhum argumento for passado.
- É executada a instrução na qual sua condição for verdadeira

```
when {
    x.isOdd() -> print("x is odd")
    x.isEven() -> print("x is even")
    else -> print("x is funny")
}
```

Controle de Fluxo - For Loops

- O loop **for** itera através de qualquer coisa que forneça um iterador

```
var vet = arrayOf("Abacaxi ", "Banana ", "Goiaba ")
for(fruta in vet) print(fruta)
//Saída: Abacaxi Banana Goiaba

for (i in 1..3) print(i)
//Saída: 123

for (i in 6 downTo 0 step 2) print(i)
//Saída: 6420
```

Controle de Fluxo - For Loops

- O loop **for** itera através de qualquer coisa que forneça um iterador

```
var vet = arrayOf("Abacaxi ", "Banana ", "Goiaba ")
for(i in vet.indices) print(vet[i])
//Saída: Abacaxi Banana Goiaba

for ((indice, valor) in vet.withIndex()) {
    println("o elemento em $indice é $valor")
}
//Saída:  o elemento em 0 é Abacaxi
          o elemento em 1 é Banana
          o elemento em 2 é Goiaba
```

Controle de Fluxo - While Loops

- while e do..while funcionam como em outras linguagens

```
var x = 5
while (x > 0) {
    print(x--)
}
//Saída: 54321

do {
    var y = retiraDado()
} while (y != null)
```

Controle de Fluxo - Break e Continue

- Kotlin suporta os operadores tradicionais break e continue em loops
- Também pode-se utilizar break e continue com labels

```
for (i in 1..100) {
    for (j in 1..100) {
        if (...) break
    }
}
```

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

Operadores

Precedence	Title	Symbols
Highest	Postfix	++, --, ., ?., ?
	Prefix	-, +, ++, --, !, labelDefinition
	Type RHS	:, as, as?
	Multiplicative	*, /, %
	Additive	+, -
	Range	..
	Infix function	SimpleName
	Elvis	?:
	Named checks	in, !in, is, !is
	Comparison	<, >, <=, >=
	Equality	==, \!==
	Conjunction	&&
	Disjunction	||
Lowest	Assignment	=, +=, -=, *=, /=, %=

Modularização

Funções - Declaração e Chamada

- Funções em kotlin são declaradas com a palavra-chave **fun**

```
fun dobro(x: Int): Int {
    return 2 * x
}
```

- Utiliza-se a abordagem tradicional para chamar as funções

```
val resultado = dobro(4)
println(resultado)
//Saída: 8
```

Funções - Parâmetros

- Os parâmetros de função são definidos usando a notação Pascal, ou seja, **nome: tipo**. Parâmetros são separados usando vírgulas. Cada parâmetro deve ser explicitamente tipado

```
fun powerOf(number: Int, exponent: Int) { /*...*/ }
```

- Os parâmetros de função podem ter valores padrão, que são usados quando um argumento correspondente é omitido

```
fun read(b: Array<Byte>, inicio: Int = 0, fim: Int = b.size) {}
```

Funções - Parâmetros

- Os parâmetros das funções podem ser passados por posição e por nome

```
fun read(b: Array<Byte>, inicio: Int =  0, fim: Int = b.size) {
/*...*/ }
read(vetor)
read(vetor, 1)
read(vetor, fim = 10)
```

- Quando uma função é chamada com argumentos posicionais e nomeados, todos os argumentos posicionais devem ser colocados antes do primeiro nomeado.

```
read(vetor, fim = 10)    //Permitido
read(b = vetor, 1, 10)   //Não Permitido
```

Funções - Retorno

- Se uma função não retornar nenhum valor útil, seu tipo de retorno será Unit. Unit é um tipo com apenas um valor - Unit. Este valor não precisa ser retornado explicitamente

```
fun printHello(name: String?): Unit {
    if (name != null)
    println("Hello ${name}")
    else
    println("Hi there!")
    // `return Unit` ou `return` é opcional
}
```

Funções - Single-expression

- Quando uma função retorna uma única expressão, as chaves podem ser omitidas e o corpo é especificado após o símbolo **=**

```
fun dobro(x: Int): Int = x * 2
```

Funções - Varargs

- Um parâmetro de uma função pode ser marcado com o modificador **vararg**, permitindo que um número variável de argumentos seja passados

```
fun multiprint(vararg strings: String): Unit {
    for (string in strings)
    println(string)
}
```

- Podemos passar argumentos um por um ou, se já temos um vetor e queremos passar seu conteúdo para a função, usamos o operador spread

```
multiprint("a", "b", "c")
multiprint(“a”, “b”, *a, “f”)
```

```
val a = arrayOf(“c”, “d”, “e”)
```

Funções - Notação Infixa

- As funções marcadas com a palavra-chave **infix** também podem ser chamadas usando a notação infixa

```
infix fun Int.menos(n: Int):Int {
        return this-n
}
println(5.menos(3))
println(5 menos 3)
//Saída:    2
 2
```

Funções - Escopo

- Além das funções de nível superior, as funções do Kotlin também podem ser locais e funções de membro
- Funções Locais: uma função dentro de outra função

```
fun printArea(width: Int, height: Int): Unit {
    fun calculateArea(width: Int, height: Int): Int =
width * height
    val area = calculateArea(width, height)
    println("The area is $area")
}
```

Funções - Escopo

- Além das funções de nível superior, as funções do Kotlin também podem ser locais e funções de membro
- Funções Membro: é uma função definida dentro de uma classe ou objeto

```
class MinhaClasse() {
    fun ola() {
    print("Olá, Mundo!")
    }
}
    MinhaClasse.ola()
    //Saída: Olá, Mundo!
```

Funções - Genéricas

- As funções podem ter parâmetros genéricos que são especificados usando **<>** antes do nome da função

```
fun <T> choose(t1: T, t2: T, t3: T): T {
    return when (Random().nextInt(3)) {
    0 -> t1
    1 -> t2
    else -> t3
    }
}
    println(choose(5, "7.5", '9'))
```

Expressões lambda

```
fun main(args: Array<String>){
    val product = { a: Int, b: Int -> a * b }

    println(product(2,3))
}
```

Funções de Ordem Superior

- Uma função de ordem superior é uma função que aceita outra função como parâmetro, retorna uma função como seu valor de retorno ou ambas

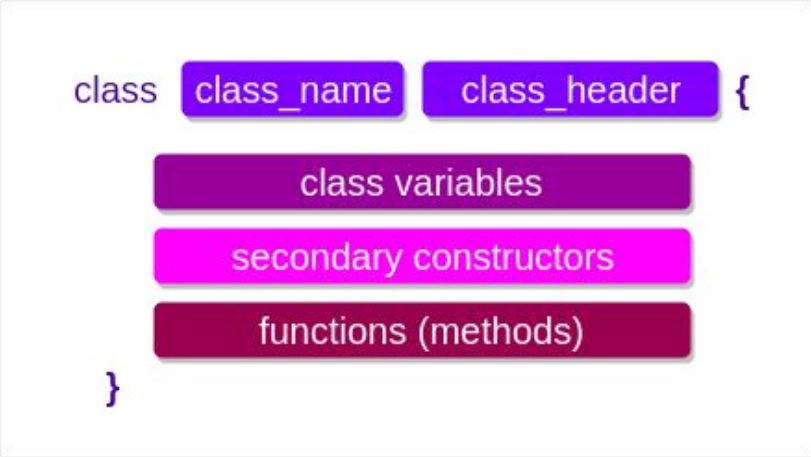
```
fun funcao(str: String, fn: (String) -> String): Unit {
        val aplicado = fn(str)
        println(aplicado)
 }
```

- Para chamar essa função, podemos passar uma função literal

```
funcao("ola", { it.reversed() })
```

Classes

- Palavra reservada class



Construtores

- Construtor primário- parte do cabeçalho da classe, não pode conter nenhum código,logica de inicialização feita no bloco init
- Construtor secundário - uso palavra chave constructor

Classes - construtor primário

```
class Person(firstName: String, lastName: String, yearOfBirth: Int) {
    val fullName = "$firstName $lastName"
    var age: Int

    init {
        age = 2018 - yearOfBirth
    }
}
```

Classes - construtor secundário

```
class Person (var name: String, var age: Int){
    var profession: String = "Not Mentioned"

    constructor (name: String, age: Int, profession: String):
this(name,age){
        this.profession = profession
    }

    fun printPersonDetails(){
        println("$name whose profession is $profession, is $age years old." )
    }
}
```

Data Class

- Lidar com dados e não referências
- toString(), equals(), hashCode(),copy() construídos implicitamente.

```
data class Movie(var name: String, var studio: String, var rating:
Float)
```

Data Class

```
data class User(var name:String, var id:Int)
fun main(args: Array<String>){
    var user1=User("Joao",10)
    var user2=User("Joao",10)

    println(user1.toString())

    if(user1==user2){
        println("Iguais")
    }else{
        println("Diferentes")
    }
    var user3=user1.copy()
    println(user3)
}
```

Classes abstratas

- Métodos abstratos devem ser marcados com abstract e deverão ser substituídos em suas subclasses
- Podem ter métodos não abstratos

Classes abstratas

```
abstract class Employee(val name: String,val experience: Int) {
    abstract var salary: Double
    abstract fun dateOfBirth(date:String)

    fun employeeDetails() {
        println("Name of the employee: $name")
        println("Experience in years: $experience")
        println("Annual Salary: $salary")
    }
}

class Engineer(name: String,experience: Int) :
Employee(name,experience) {
    override var salary = 500000.00
    override fun dateOfBirth(date:String){
        println("Date of Birth is: $date")
    }
}
```

Sealed Classes

- Usadas para representar hierarquias de classes restritas, quando um valor pode ter um dos tipos de um conjunto limitado
- Todas subclasses declaradas no mesmo arquivo
- Sem classes seladas, não há exaustão (cobertura completa) de possibilidade.

Sealed Classes

```
sealed class Shape{
    class Circle(var radius: Float): Shape()
    class Square(var length: Int): Shape()
    class Rectangle(var length: Int, var breadth: Int): Shape()
}

fun eval(e: Shape) =
    when (e) {
        is Shape.Circle -> println("Circle area is ${3.14*e.radius*e.radius}")
        is Shape.Square -> println("Square area is ${e.length*e.length}")
        is Shape.Rectangle -> println("Rectagle area is ${e.length*e.breadth}")
    }
```

Generics

```
class Company<T> (text : T){
    var x = text
    init{
        println(x)
    }
}

fun main(args: Array<String>){
    var name: Company<String> = Company<String>("Doze")
    var rank: Company<Int> = Company<Int>(12)
}
```

Polimorfismo

Ad Hoc

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

```
fun main(args: Array<String>){
    val inteiro : Int = 2
    val long : Long
    long = inteiro
}
error:type mismatch: inferred type is Int but Long was
expected
```

Ad Hoc

- Sobrecarga- possível fazer sobrecarga de operadores

```
data class Point(val x: Int, val y: Int)
operator fun Point.plus(other: Point) = Point(x + other.x, y + other.y)

fun main() {
 val p1 = Point(0, 1)
 val p2 = Point(1, 2)
 println(p1 + p2)
}
//Point(x=1, y=3)
```

Ad Hoc

- Sobrecarga-permite sobrecarga de funções

```
fun main() {
    println(square(34))
    println(square(62.34))
}

fun square(num: Int) = num * num
fun square(num: Double) = num * num
```

Universal - paramétrico

```
class Company<T> (text : T){
    var x = text
    init{
        println(x)
    }
}
fun main(args: Array<String>){
    var name: Company<String> = Company<String>("teste")
    var rank: Company<Int> = Company<Int>(12)
}
```

Universal - inclusão

- Suportado pela linguagem devido a ser orientada a objeto

Universal - inclusão

- Herança- toda classe em Kotlin possui um supertipo chamado Any
- Any tem métodos equals(), hashCode() e toString() definido para toda classe
- Uma classe herda somente de uma classe base

```
open class Food(val price: Double)
class Hamburger(price: Double) : Food(price)
```

Universal - inclusão

```
open class Animal() {
    open var colour: String = "White"
}
class Dog: Animal() {
    override var colour: String = "Black"
    fun sound() {
        println("Dog makes a sound of woof woof")
    }
}
```

Exceções

Exceções

- Unchecked (checada em tempo de execução), capturada em tempo de execução
- Derivadas da classe Throwable
- Não precisam ser declaradas explicitamente nas assinaturas de funções, como em Java

Exceções

```
fun main (args: Array<String>){
    try {
        var int = 10 / 0
        println(int)
    } catch (e: ArithmeticException) {
        println(e)
    } finally {
        println("This block always executes")
    }
}
```

```
fun test(password: String) {

    if (password.length < 6)
        throw ArithmeticException("Senha Fraca")
    else
        println("Senha Forte")
}

fun main(args: Array<String>) {
    test("abcd")
}
```

Classe nothing

- Utilizada para representar "um valor que nunca existe"
- Se uma função tem o tipo de retorno Nothing, significa que ela nunca retorna
- Loop infinito, lançar erro

Classe nothing

```
fun main(args: Array<String>) {
    forever()
}
fun forever(): Nothing{
    while(true){
        Thread.sleep(1)
        println("oi")
    }
}
```

Classe nothing

Ex.:

```
fun throwException(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

Concorrência

Coroutines

- Meio de escrever códigos de forma assíncrona
- Similar aos conceitos de *async/await*
- “Threads leves”
- Representam operações que esperam algo (IO bound)
- *suspending function*

```
import kotlinx.coroutines.*


fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    runBlocking{
        delay(2000L)
    }
}
```

Coroutines

- Meio de escrever códigos de forma assíncrona
- Similar aos conceitos de *async/await*
- “Threads leves”
- Representam operações que esperam algo (IO bound)
- *suspending function*

```
import kotlinx.coroutines.*


fun main() {
    GlobalScope.launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
    runBlocking{
        delay(2000L)
    }
}
```

Saída:
Hello,
World!

Coroutines

- Meio de escrever códigos de forma assíncrona
- Similar aos conceitos de *async/await*
- “Threads leves”
- Representam operações que esperam algo (IO bound)
- *suspending function*

```
import kotlinx.coroutines.*


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

Saída:
Hello,
World!

Coroutines

- Meio de escrever códigos de forma assíncrona
- Similar aos conceitos de *async/await*
- “Threads leves”
- Representam operações que esperam algo (IO bound)
- *suspending function*

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Saída:
Hello,
World!

Coroutines

- Meio de escrever códigos de forma assíncrona
- Similar aos conceitos de *async/await*
- “Threads leves”
- Representam operações que esperam algo (IO bound)
- *suspending function*

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(100_000) {
        launch {
            delay(1000L)
            print(".")
        }
    }
}
```

Saída:

.....
.....
.....
.....

Threads

- Recomendadas quando se há necessidade de maior poder de processamento (CPU bound)

```
import kotlinx.coroutines.*

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

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

Saída:
Thread 2
Thread 1

Coroutines vs Threads

```
import kotlinx.coroutines.*
fun threadName(): String = Thread.currentThread().name
fun main() {
   runBlocking {
         launch {
            test1WithCoroutines()
         }
         launch {
            test2WithCoroutines()
         }
      }
}
suspend fun test1WithCoroutines() { … }
suspend fun test2WithCoroutines() { … }
```

```
fun threadName(): String = Thread.currentThread().name
fun main() {
   test1WithThread( )
   test2WithThread( )
}
fun test1WithThread() { … }
fun test2WithThread() { … }
```

Coroutines vs Threads

```
suspend fun test1WithCoroutines() {
    println("Start coroutines test 1: ${threadName()}")
    delay(500)
    println("End coroutines test 1: ${threadName()}")
}
suspend fun test2WithCoroutines() {
    println("Start coroutines test 2: ${threadName()}")
    delay(1000)
    println("End coroutines test 2: ${threadName()}")
}
```

```
fun test1WithThread() {
    println("Start thread test 1: ${threadName()}")
    Thread.sleep(500)
    println("End thread test 1: ${threadName()}")
}
fun test2WithThread() {
    println("Start thread test 2: ${threadName()}")
    Thread.sleep(1000)
    println("End thread test 2: ${threadName()}")
}
```

Coroutines vs Threads

```
suspend fun test1WithCoroutines() {
    println("Start coroutines test 1: ${threadName()}")
    delay(500)
    println("End coroutines test 1: ${threadName()}")
}
suspend fun test2WithCoroutines() {
    println("Start coroutines test 2: ${threadName()}")
    delay(1000)
```

Saída:

```
Start coroutines test 1: main @coroutine#2
Start coroutines test 2: main @coroutine#3
End coroutines test 1: main @coroutine#2
End coroutines test 2: main @coroutine#3
```

```
fun test1WithThread() {
    println("Start thread test 1: ${threadName()}")
    Thread.sleep(500)
    println("End thread test 1: ${threadName()}")
}
fun test2WithThread() {
    println("Start thread test 2: ${threadName()}")
```

Saída:

```
Start thread test 1: main
End thread test 1: main
Start thread test 2: main
End thread test 2: main
```

Semaphores

- Problema dos Leitores - Escritores

```
import java.util.concurrent.Semaphore
internal object ReadersWritersProblem {

    var readLock = Sempahore(1)
    var writeLock = Semaphore(1)
    var readCount = 0


    internal class Read : Runnable{...}

    internal class Write : Runnable{...}


    @Throws(Exception::class)

    @JvmStatic

    fun main(args : Array<String>){...}

}
```

Semaphores

- Problema dos Leitores - Escritores

```
@Throws(Exception::class)
@JvmStatic
fun main(args : Array<String>){
    val read = Read()
    val write = Write()
    val t1 = Thread(read)
    val t2 = Thread(read)
    val t3 = Thread(write)
    val t4 = Thread(read)
    t1.start()
    t1.start()
    t1.start()
    t1.start()
}
```

Semaphores

- Problema dos Leitores - Escritores
- Leitor

```
internal class Read : Runnable {
        override fun run() {
            try {
                readLock.acquire()
                readCount++
                if (readCount == 1) {
                    writeLock.acquire()
                }
                readLock.release()
                //Leitura
                readLock.acquire()
                readCount--
                if (readCount == 0) {
                    writeLock.release()
                }
                readLock.release()
            } catch (e: InterruptedException) {
                println(e.message)
            }
        }
    }
```

Semaphores

- Problema dos Leitores - Escritores
- Escritor

```
internal class Write : Runnable {
        override fun run() {
            try {
                writeLock.acquire()
                //Escrita
                writeLock.release()
            }catch(e: InterruptedException){
                println(e.message)
            }
        }
    }
```

Avaliação da linguagem

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	
Aprendizado	Não	Não		
Eficiência	Sim	Sim		
Portabilidade	Não	Não		

- Assim como Java, Kotlin não oferece recursos para controlar diretamente o hardware.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	
Eficiência	Sim	Sim		
Portabilidade	Não	Não		

- Coletor de lixo;
- Null Safety.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Não	Não

- Comunidade menor;
- Muitos conceitos que podem ser complexos e não ortogonais.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	

- Null Safety, Coletor de lixo;
- Compilável e Interpretável.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim

- JVM;
- Aplicações Android.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende	Depende	Depende	Depende

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial		
Reusabilidade	Parcial	Sim		
Integração	Sim	Sim		
Custo	Depende	Depende	Depende	Depende

- Classes;
- Expressões Lambda.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim		
Integração	Sim	Sim		
Custo	Depende	Depende	D	

- OO;
- Concisa.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim		
Custo	Depende	Depende		

- Open Source;
- Polimorfismo;
- Diversas bibliotecas para criação de APIs.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende	Depende	Depende	[illegible]

- 100% interoperável com Java.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende	Depende	Depende	Depende

- Gratuita;
- Poucos desenvolvedores.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Programador	Sistema	Sistema

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim			
Gerenciamento de Memória	Programador	Programador	Sistema	Sistema

- Kotlin também requer a definição explícita de entidades, associando-as a um escopo, que pode ser estático ou aninhado.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	
Gerenciamento de Memória	Programador	Programador		

- Ampla variedade de expressões e comandos.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Programador	Sistema	[illegible]

- Tipos Básicos;
- Ampla variedade;
- *assign-once* & *read-only*.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Programador	Sistema	Sistema

- Coletor de lixo.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Persistência dos Dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	JSON, biblioteca de classes, serialização
Passagem de Parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por cópia.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Persistência dos Dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	JSON, biblioteca de classes, serialização
Passagem de Parâmetros	Lista variável e por valor	Lista variável, defau valor refe	Lista variável,	ariável,

- Semelhante à Java na serialização e em suas bibliotecas;
- JSON se apresenta como opção voltada ao desenvolvimento Android.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Persistência dos Dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	JSON, biblioteca de classes, serialização
Passagem de Parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por cópia.

- *var* (mutável) & *val* (imutável).

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos, exceto Coerção

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	
Verificação de Tipos	Estática	Estática / Dinâmica		
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos, exceto Coerção

- Apresenta mecanismos de classes e pacotes.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâm	Estática /	ica /
Polimorfismo	Coerção e Sobrecarga	Todo		

- Fortemente tipada, bastante rigoroso semelhantemente a Java.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	

- *Smart cast*;
- Inferência de tipos;
- Pode haver amarração tardia quando é utilizado o paradigma funcional.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos, exceto Coerção

- Não há a conversão implícita de tipos.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca de funções)	Sim	Sim

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca	Sim	Sim

- Capturadas e checadas em tempo de execução;
- Não precisa ser declarada explicitamente na assinatura da função.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca de funções)	Sim	Sim

- Coroutines;
- Threads;
- Semáforos;
- Monitores.