# Introdução à linguagem Kotlin

MOV786205 – CST em Análise e Desenvolvimento de Sistemas

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

# Licenciamento

# O que você vai aprender?

## Conceitos básicos de programação com Kotlin

- Como criar um projeto gradle e Kotlin no IntelliJ IDEA
- Paralelo com a linguagem Java
- Sintaxe básica
- Variáveis e tipos de dados
- Operadores
- Estruturas de decisão e repetição
- Leitura de dados pelo teclado

# Linguagem Kotlin

**https://kotlinlang.org/**

- Multiplataforma (JVM, nativa, JavaScript, WebAssembly)
- Concisa e segura contra nulidade, que executa na JVM e é interoperável com Java
- Suporte a **programação funcional e orientada a objetos**, com recursos como funções de ordem superior, lambdas e extensões
- É a linguagem oficial para desenvolvimento Android, com suporte completo no Android Studio

# Sintaxe básica

- Não é necessário o ponto e vírgula (`;`) no final de cada linha
- O compilador infere o tipo de dado das variáveis, mas é possível declarar explicitamente
- A convenção[1] é usar `camelCase` para nomes de variáveis e funções, e `PascalCase` para nomes de classes
- Arquivos Kotlin devem ter a extensão `.kt` e podem conter múltiplas declarações de funções e classes
- Os arquivos devem ser organizados em pacotes e é recomendável que a estrutura de diretórios reflita a hierarquia dos pacotes

[1] https://kotlinlang.org/docs/coding-conventions.html

# Estrutura de diretórios de projeto com gradle

- Java e Kotlin usam a mesma estrutura de diretórios, com os arquivos de código-fonte em `src/main/java` ou `src/main/kotlin`, respectivamente
- Os arquivos de teste devem estar em `src/test/java` ou `src/test/kotlin`

```
IntroducaoKotlin
`-- src
    |-- main
    |   |-- kotlin
    |   |   |-- ads
    |   |   |   `-- mov
    |   |   |       `-- Main.kt
    |   `-- resources
    `-- test
        |-- kotlin
        `-- resources
```

# Estrutura de um programa

- Em Java, é composta por uma classe com um método `main` (estático) com um parâmetro. O nome do arquivo deve ser o mesmo nome da classe

```java
package ads.mov;

public class App {
    public static void main(String[] args) {
        System.out.println("Olá mundo!");
    }
}
```

- Em Kotlin, não é necessário declarar uma classe para o programa, e os parâmetros são opcionais da função `main`

```kotlin
package ads.mov

fun main() {
    println("Olá mundo!")
}
```

# Olá mundo! em Kotlin

```kotlin
package ads.mov

fun olaMundo() {
    println("Olá mundo!")
}

fun main() {
    olaMundo() // Chama a função olaMundo
}
```

- O arquivo `.kt` do programa pode ter qualquer nome, sendo comum usar `Main.kt`
- Não existe o conceito de `static` em Kotlin
  - Depois vemos como isso é tratado em Kotlin

# Declaração de variáveis em Kotlin

- A declaração de variáveis é feita com as palavras-chave:
  - `val`, para variáveis imutáveis
  - `var`, para variáveis mutáveis

```kotlin
// Declaração de variáveis com tipo explícito
var nome: String // tipo String, mutável
nome = "João" // atribuição de valor

// Declaração de variáveis com tipo explícito e inicialização
var idade: Int = 30 // tipo Int, mutável e inicializado com 30

// Declaração de variáveis com tipo inferido
val valor = 25.0 // tipo inferido como Int, imutável
var ativo = true // tipo inferido como Boolean, mutável

idade = 20 // atribuição de novo valor
// valor = 3.0 // erro: não é possível atribuir valor a uma variável imutável
```

# Por que variáveis imutáveis?

## Conceito de imutabilidade

- Previne modificações não intencionais, tornando o código mais seguro e previsível
- Facilita a concorrência, pois elimina a necessidade de sincronização para acesso a dados
- Melhora a performance, permitindo otimizações pelo compilador
- Conceito central do paradigma funcional, permitindo funções puras
  - Funções puras são aquelas que não têm efeitos colaterais e sempre retornam o mesmo resultado para os mesmos parâmetros
  - Operam sobre dados sem modificá-los e criam novos dados a partir deles

# Tipos primitivos em Java

| Tipo | Tamanho | Descrição | Exemplo |
|---|---|---|---|
| `int` | 32 bits | Inteiro | `int i = 10;` |
| `long` | 64 bits | Inteiro longo | `long l = 100000L;` |
| `short` | 16 bits | Inteiro curto | `short s = 1000;` |
| `float` | 32 bits | Ponto flutuante | `float f = 3.14f;` |
| `double` | 64 bits | Ponto flutuante | `double d = 3.14159;` |
| `boolean` | 1 bit | Lógico | `boolean b = true;` |
| `char` | 16 bits (Unicode) | Caractere | `char c = 'A';` |
| `byte` | 8 bits | Byte | `byte y = 127;` |

# Não existem tipos primitivos em Kotlin, todos são objetos

| Tipo | Tamanho | Descrição | Exemplo |
|---|---|---|---|
| `Int` | 32 bits | Inteiro | `var i: Int = 10` |
| `Long` | 64 bits | Inteiro longo | `var l: Long = 100000L` |
| `Short` | 16 bits | Inteiro curto | `var s: Short = 1000` |
| `Float` | 32 bits | Ponto flutuante | `var f: Float = 3.14f` |
| `Double` | 64 bits | Ponto flutuante | `var d: Double = 3.14159` |
| `Boolean` | 1 bit | Lógico | `var b: Boolean = true` |
| `Char` | 16 bits | Caractere | `var c: Char = 'A'` |
| `Byte` | 8 bits | Byte | `var y: Byte = 127` |

# Operadores (alguns exemplos)

| Tipo | Operadores | Descrição |
|---|---|---|
| Aritméticos | `+`, `-`, `*`, `/`, `%` | Soma, subtração, multiplicação, divisão, módulo |
| Comparação | `==`, `!=`, `<`, `>`, `<=`, `>=` | Igualdade, diferença, maior/menor |
| Lógicos | `&&`, `\|\|`, `!` | E, OU, NÃO lógico |
| Atribuição | `=`, `+=`, `-=`, `*=`, `/=`, `%=` | Atribuição e operações compostas |
| Inc/Dec | `++`, `--` | Incremento/decremento |
| Intervalo | `in`, `!in` | Pertence ou não a um intervalo |
| Tipo | `is`, `!is` | Verifica tipo |
| Elvis | `?:` | Valor padrão se nulo |
| Chamada segura | `?.` | Acesso seguro a membros de objeto nulo |

# Segura contra nulidade

- Kotlin é projetada para evitar erros de nulidade, que são comuns em Java
  - Java dispara uma exceção de ponteiro nulo (`NullPointerException`) se tentar acessar um membro de um objeto nulo
- Variáveis que podem ser nulas devem ser declaradas com o sufixo `?`
  - Isso obriga o programador a lidar explicitamente com valores nulos

```
// Declaração de variável que permite nulo
var nome: String?

// Declaração de variável que não permite nulo
var cidade: String

nome = null  // Válido, pois permite nulo
cidade = null // ERRO, pois não permite guardar nulo
```

# Segurança contra nulidade em Kotlin

- O operador de chamada segura `?.` permite acessar membros de um objeto apenas se ele não for nulo, retornando `null` caso contrário
- O operador Elvis (`?:`) fornece um valor padrão quando uma expressão resulta em `null`, tornando o código mais robusto

```kotlin
var nome: String? = null

// Operador Elvis: se 'nome' for nulo, retorna o valor à direita
println(nome ?: "Nome não informado") // Saída: Nome não informado

nome = "Maria"
// Chamada segura: só acessa 'length' se 'nome' não for nulo
println(nome?.length) // Saída: 5

// Encadeando operadores: retorna o tamanho ou zero se 'nome' for nulo
val tamanho = nome?.length ?: 0
println("Tamanho do nome: $tamanho")
```

# Leitura de dados pelo teclado

## Funções `readln` e `readlnOrNull`

- `readln` – Lê uma linha do teclado e retorna uma String não nula
- `readlnOrNull` – Lê uma linha do teclado e retorna uma String ou nulo

```
fun main() {
    print("Digite seu nome: ")
    // Operador Elvis (?:) fornece um valor padrão caso a entrada seja nula
    val nome = readlnOrNull() ?: ""

    print("Digite sua idade: ")
    // toIntOrNull() converte a String para Int, retornando nulo se a conversão falhar
    val idade = readlnOrNull()?.toIntOrNull() ?: 0

    println("Olá, $nome! Você tem $idade anos.")
}
```

# Funções de operações numéricas

## Números são objetos em Kotlin, e assim podemos invocar suas funções

```kotlin
package ads.mov

fun main() {
    val numero1 = 10
    val numero2 = 3

    // Operações aritméticas
    println("Soma: ${numero1.plus(numero2)}") // Soma
    println("Multiplicação: ${numero1.times(numero2)}") // Multiplicação
    println("Divisão: ${numero1.div(numero2)}") // Divisão
    println("Módulo: ${numero1.rem(numero2)}") // Módulo

    // Funções de comparação
    println("É maior que? ${numero1 > numero2}") // Maior que
    println("É menor que? ${numero1 < numero2}") // Menor que

    // Converter um Int para String
    val numeroComoString = numero1.toString()
}
```

# Strings I

- Podem ser declaradas com três aspas (`"`) para strings multilinha
  - Permitem quebras de linha e indentação sem a necessidade de caracteres especiais
  - Podem ser usadas para criar templates de texto
  - No início de uma linha, o caractere `|` pode ser usado para indicar a margem
- Strings podem conter expressões interpoladas usando o símbolo `$` ou `${expressão}` para avaliar expressões dentro da string

# Strings II

```
val nome = "João"
val idade = 30
val mensagem = "Olá, meu nome é $nome e tenho $idade anos."
val variasLinhas = """Olá, meu nome é $nome
|E tenho $idade anos.
|Essa é uma string multilinha.""".trimMargin()

println(mensagem)
println(variasLinhas)
```

```
val str1 = "Olá"
val str2 = "Mundo"
val str3 = "$str1 $str2" // Concatenação

println("Tamanho da string: ${str3.length}")
println("Primeiro caractere: ${str3[0]}")
println("Substituição: ${str3.replace("Mundo", "Kotlin")}")
```

# Argumentos de linha de comando

```
fun main(args: Array<String>) {
    if (!args.isEmpty()) {

        // Percorrendo os argumentos passados na linha de comando
        args.forEachIndexed { indice, argumento ->
            println("argumento[$indice]: $argumento")
        }

        // Abaixo um exemplo com for
        // apesar de que o uso do `forEachIndexed` é mais idiomático em Kotlin
        for (indice in args.indices) {
            println("argumento[$indice]: ${args[indice]}")
        }
    }
}
```

# Vetores (arranjos)

- A classe `Array` é usada para criar vetores em Kotlin, com tamanho fixo
- Em Kotlin é preferível usar coleções como `List`, `Map` ou `Set` para manipulação de dados

```
val numeros = arrayOf(1, 2, 3, 4, 5) // Cria um vetor de inteiros
println("Tamanho do vetor: ${numeros.size}")
println("Primeiro elemento: ${numeros[0]}")
println("Último elemento: ${numeros[numeros.size - 1]}")
println("Todos os elementos: ${numeros.joinToString(", ")}")
println("Elementos pares: ${numeros.filter { it % 2 == 0 }}")

// Cria um vetor de inteiros com tamanho 5, sem inicialização
var idade = IntArray(5)
// Cria um vetor de inteiros com tamanho 10, inicializado com zeros
var outro = Array(10) { 0 }
// Cria um vetor de strings com tamanho 3, inicializado com strings vazias
var nomes = Array(3) { "" }
```

# Listas, Mapas e Conjuntos

## Coleções imutáveis e mutáveis, que são mais flexíveis que os vetores

- **Listas** – coleções ordenadas de elementos, que podem conter duplicatas
  - `listOf()` – cria uma lista imutável
  - `mutableListOf()` – cria uma lista mutável
- **Mapas** – coleções de pares chave-valor, onde cada chave é única
  - `mapOf()` – cria um mapa imutável
  - `mutableMapOf()` – cria um mapa mutável
- **Conjuntos** – coleções de elementos únicos, sem ordem específica
  - `setOf()` – cria um conjunto imutável
  - `mutableSetOf()` – cria um conjunto mutável

# Listas, Mapas e Conjuntos: exemplos

```
// Lista imutável de inteiros com inferência de tipo
val numeros = listOf(1, 2, 3, 4, 5)

// Lista mutável de inteiros com tipo explícito
val numerosMutavel: MutableList<Int> = mutableListOf(1, 2, 3, 4, 5)

println("Lista de números: $numeros")

val nomes = mutableListOf("João", "Maria", "Pedro") // Lista mutável de strings
nomes.add("Ana") // Adiciona um elemento
println("Lista de nomes: $nomes")

val mapa = mapOf("a" to 1, "b" to 2, "c" to 3) // Mapa imutável de chaves e valores
println("Mapa: $mapa")

val mapaMutavel = mutableMapOf("x" to 10, "y" to 20) // Mapa mutável de chaves e valores
mapaMutavel["z"] = 30 // Adiciona um novo par chave-valor
println("Mapa mutável: $mapaMutavel")
```

# Tipos misturados em coleções e arranjos

- Kotlin permite criar coleções com tipos misturados, mas é recomendável evitar isso para manter a clareza do código
- Para coleções com tipos misturados, é comum usar o tipo `Any`, que é a superclasse de todos os tipos em Kotlin

```
val misturado = mutableListOf(1, "dois", 3.0, true) // Lista com tipos misturados
println("Lista misturada: $misturado")

// Com o tipo Any explícito
val misturadoComAny: MutableList<Any> = mutableListOf(1, "dois", 3.0, true)
// Ou ainda
val misturado2 = mutableListOf<Any>(1, "dois", 3.0, true)

// Array com tipos misturados
val arranjo = arrayOf<Any>("Juca", 10)
```

# Estruturas de decisão: if, else

https://kotlinlang.org/docs/control-flow.html

```kotlin
val idade = 17

if (idade < 16) {
    println("Não pode votar no Brasil")
} else if (idade in 16..17) { // o operador 'in' é usado para verificar intervalos
    println("Pode votar no Brasil")
} else {
    println("Pode votar e dirigir no Brasil")
}

// if else como expressão, equivalente ao operador ternário em Java
val numero = if (10 % 2 == 0) "par" else "ímpar"
```

- O trecho `12..18` gera um intervalo de números de 12 a 18, inclusive

# Estrutura de decisão: when

## Equivalente ao switch-case do Java, mas mais poderoso e flexível

```
val numero = 2
when (numero) {
    1 -> println("Número é 1")
    2 -> println("Número é 2")
    else -> println("Número desconhecido")
}

when (numero) {
    1, 2 -> println("Número é 1 ou 2")
    in 3..5 -> println("Número está entre 3 e 5")
    else -> println("Número desconhecido")
}

// Usado como expressão
val resultado = when (numero) {
    1 -> "Um"
    2 -> "Dois"
    else -> "Outro"
}
```

# Estruturas de decisão: when e coleções

```
fun main() {
    // Criando uma lista mista de diferentes tipos
    val listaMista = listOf("Kotlin", 42, 3.14, true)

    // Iterando sobre a lista mista
    for (item in listaMista) {
        // Verificando o tipo de cada item e imprimindo uma mensagem
        when (item) {
            is String -> println("String: $item")
            is Int -> println("Inteiro: $item")
            is Double -> println("Double: $item")
            is Boolean -> println("Booleano: $item")
            else -> println("Tipo desconhecido")
        }
    }
}
```

# Estruturas de repetição: for

```
for (i in 1..5) { // Iterando sobre um intervalo de 1 a 5
    println("Número: $i")
}

for (i in 1..10 step 2) { // Iterando sobre um intervalo com passo de 2
    println("Número: $i")
}

// Iterando sobre uma lista
val numeros = listOf(1, 2, 3, 4, 5) // ou arrayOf(1, 2, 3, 4, 5)
for (numero in numeros) {
    println("Número: $numero")
}
// Iterando sobre um lista com índice
for ((indice, numero) in numeros.withIndex()) {
    println("Número na posição $indice: $numero")
}
// Iterando sobre um mapa
val mapa = mapOf("a" to 1, "b" to 2, "c" to 3)
for ((chave, valor) in mapa) {
    println("Chave: $chave, Valor: $valor")
}
```

# Estruturas de repetição: while e do-while

```
// Estrutura de repetição while
var j = 0
while (j < 5) {
    println("Número: $j")
    j++
}

// Estrutura de repetição do-while
var k = 0
do {
    println("Número: $k")
    k++
} while (k < 5)
```

# Aula baseada em


JETBRAINS. **Kotlin Programming Language**. 2025. Disponível em: https://kotlinlang.org/docs/. Acesso em: 9 jul. 2025.