Desenvolvimento para Dispositivos Móveis

Introdução à Linguagem Kotlin

Professor José Pacheco de Almeida Prado
Jose.prado@faculdadeimpacta.com.br

Introdução

- Desde Google I/O 2017 o desenvolvimento de aplicativos para Android suportam a linguagem Kotlin
- Deixa o desenvolvimento mais produtivo
- Desenvolvida pela JetBrains, mesma do Android Studio
- Sintaxe moderna, expressiva, simples e agradável
- Compilada para executar na JVM
  - Interoperabilidade total com Java

Introdução

- Exemplo, comparando Java com Kotlin para clicar em um botão no app:
- Em Java

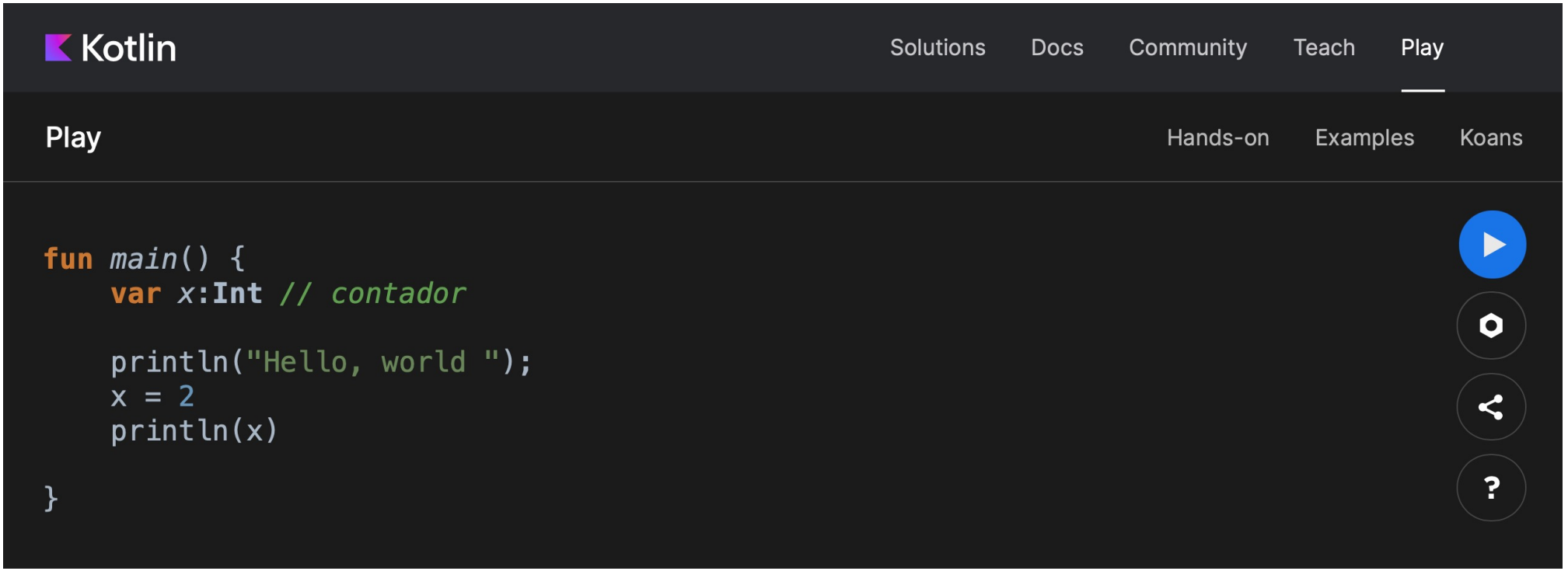
```
View btClicar = findViewById(R.id.btClicar);
btClicar.setOnClickListener(new View.OnClickListener() {
    @Override public void onClick(View v) {
        clicarBotao()
} });
```

- Em Kotlin

```
findViewById<Button>(R.id.btClicar).setOnClickListener(clicarBotao());
```

Treinar Kotlin

- Acesse https://play.kotlinlang.org/para testar alguns exemplos e se familiarizar com a linguagem Kotlin:



Treinar Kotlin

- Um programa Kotlin tem a extensão .kt
- HelloWorld – hello_world.kt

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

- Sintaxe básica:
  - fun: define uma função
  - Tipo do parâmetro especificado depois do nome , separado por : (args: Array<String>)
  - Não precisa de ;
  - Bloco definido por chaves {}

Funções print e println

- Imprimir no console
  - print(String): sem quebra de linha
  - println(String): com quebra de linha

```
fun main(args: Array<String>) {
    print("Hello, world")
    println("Hello, world")
    println("Kotlin")
}
```

String Templates

- Utilizar o valor de uma variável para imprimir ou com outra string sem necessidade de concatenar
- `$variavel` ou `${objeto.propriedade}`

```
fun main(args: Array<String>) {
    var nome = "Fernando"
    println("Olá $nome")
    println("$nome possui ${nome.length}")
    var nome_completo = "$nome Sousa"
    println(nome_completo)
}
```

- Para saber mais: https://kotlinlang.org/docs/reference/basic-types.html#string-templates

var e val

- var: criar variável
- val: criar uma constante (valor atribuído não pode ser alterado)
- Sintaxe básica:

```
var variavel:tipo = valor
val variavel:tipo = valor
```

- O tipo vem depois do nome da variável, separado por :, e pode ser omitido caso seja feita uma atribuição na declaração

```
var variavel = valor
val variavel = valor
```

- Para saber mais: https://kotlinlang.org/docs/reference/basic-syntax.html

Tipos numéricos

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Tipos numéricos

```
val numeroMilhao = 1_000_000
val numeroCartao = 1234_5678_9012_3456L
val cpf = 999_999_999_99L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001
```

var e val

- Exemplos

```
fun main(args: Array<String>) {
    var nome:String = "Fernando"
    println("Olá $nome")
    nome = "Fernando Sousa"
    println("Olá $nome")
}
```

```
fun main(args: Array<String>) {
    var nome:String = ”Ana"
    println("Olá $nome")
    var sobrenome = "Silva"
    println("Olá $nome $sobrenome")
}
```

```
fun main(args: Array<String>) {
    val nome:String = ”Maria"
    println("Olá $nome")
    nome = “Maria Silava“ // erro de compilação - Val cannot be reassigned
    println("Olá $nome")
}
```

Conversão de tipos: as e is

- as: converter tipos (cast)
  - as? : cast seguro. Retorna null caso a conversão não possa ser feita (exceção)
- is: verificar se uma variável é de um tipo
  - Se utilizado dentro de um if e for verdadeiro, a conversão é automática (Smart Cast)

```
fun main(args: Array<String>) {
    var s:Any = "Fernando"
    println(s as String) // transforma s em uma String
    println(s as? Int) // cast seguro: não é possível converter String em Int
    if (s is String) { // verdadeiro: converte s em uma String
        println("$s é uma string")
    }
}
```

- Para saber mais: https://kotlinlang.org/docs/reference/typecasts.html

Objetos nulos (Null Safety)

- Em Kotlin não é possível armazenar valores nulos em variáveis e objetos, por padrão
- A forma de fazer isso é explicitar que uma variável pode receber o valor nulo
- O código a seguir não compila: nome não pode receber nulo

```
fun main(args: Array<String>) {
    var nome = "Fernando"
    println("Olá $nome")
    nome = null // Erro de compilação
    println("Olá $nome")
}
```

- Para receber nulo, a variável deve ter tipo e o operador ?

```
fun main(args: Array<String>) {
    var nome:String? = "Fernando"
    println("Olá $nome")
    nome = null // OK
    println("Olá $nome")
}
```

Objetos nulos (Null Safety)

- Se uma variável nula for chamada sem verificação, o código não vai nem compilar

```
fun main(args: Array<String>) {
    var nome:String? = "Fernando"
    println("Olá $nome")
    nome = null // OK
    println("Olá $nome")
    println("$nome possui ${nome.length} caracteres") // Erro de compilação
}
```

- É preciso verificar antes se a variável é nula

```
fun main(args: Array<String>) {
    var nome:String? = "Fernando"
    println("Olá $nome")
    nome = null // OK
    println("Olá $nome")
    if (nome != null) {
        println("$nome possui ${nome.length} caracteres")
    }
}
```

Objetos nulos (Null Safety)

- Ou então utilizar o operador de safe call (?)
  - Ignora a chamada se o objeto for nulo

```
fun main(args: Array<String>) {
    var nome:String? = "Fernando"
    println("Olá $nome")
    nome = null // OK
    println("Olá $nome")

    println("$nome possui ${nome?.length} caracteres") // Erro de compilação
}
```

- Para saber mais: https://kotlinlang.org/docs/reference/null-safety.htm

Operador ternário e Elvis

- Feito com if/else

```
fun parOuImpar(a: Int): String {
    return if (a % 2 == 0) "par" else "impar"
}

fun main(args: Array<String>) {
    println(parOuImpar(1))
    println(parOuImpar(2))
}
```

- Elvis (?:)
  - Se o valor da variável não for nulo, utilize seu valor; caso contrário, utilize outro

```
fun enviarEmail(usuario: String, titulo: String? = null): String {
    val s = titulo?: “Bem vindo”
    return “$s $usuario”
}

fun main(args: Array<String>) {
    println(enviarEmail(“Fernando”))
    println(enviarEmail(“Fernando”, “Olá”))
}
```

Operador ternário e Elvis

- Para saber mais:
  - https://kotlinlang.org/docs/reference/control-flow.html
  - https://kotlinlang.org/docs/reference/null-safety.html

Funções

- Sintaxe básica

```
fun nomeFuncao(param1: Tipo, param2: Tipo, …): TipoRetorno {
    // Corpo da função
}
```

- Exemplo:

```
fun main(args: Array<String>) {
    var nome = "Fernando"
    imprimir(nome)
    val soma = somar(2, 3)
    imprimir("Soma: $soma")
}
// recebe uma string e não retorna nada (Unit)
fun imprimir(s: String): Unit {
    println(s)
}

// Recebe 2  inteiros e retorna uma inteiro
fun somar(a: Int, b: Int): Int {
    return a + b
}
```

- Unit identifica que a função não retorna nada. Seu uso é opcional

Funções

- Se a função tiver apenas uma linha, pode-se utilizar a sintaxe resumida. Inclusive o tipo de retorno pode ser omitido
  - Basta tirar as chaves e usar o operador de igualdade, tudo em uma linha

```
fun main(args: Array<String>) {
    var nome = "Fernando"
    imprimir(nome)
    val soma = somar(2, 3)
    imprimir("Soma: $soma")
}
// recebe uma string e não retorna nada (Unit)
fun imprimir(s: String) = println(s)

// Recebe 2  inteiros e retorna uma inteiro
fun somar(a: Int, b: Int) = return a + b
```

- Para saber mais: https://kotlinlang.org/docs/reference/functions.html

Funções – argumentos padrão

- Parâmetros de funções podem ter valores padrão, evitando a sobrecarga de métodos

```
fun main(args: Array<String>) {
    var i = getInteiro("5")
    println(i)
    i = getInteiro(null)
    println(i)
    i = getInteiro(null, 2)
    println(i)
}

// Função que transforma uma string num inteiro; caso a string seja nula,
// retorna 0, o valor doo argumento padrão
fun getInteiro(s: String?, padrao: Int = 0): Int {
    if (s != null) {
        return s.toInt()
    }
    return padrao
}
```

- Mais: https://kotlinlang.org/docs/reference/functions.html#default-arguments

Funções – argumentos nomeados

- O nome do parâmetro pode ser utilizado na chamada da função
  - Possibilita passagem de parâmetros em qualquer ordem

```
fun teste(nome: String?, sobrenome: String? = null, faculdade: String? = null) {
    println("Nome: $nome, Sobrenome: $sobrenome, Faculdade: $faculdade")
}


fun main(args: Array<String>) {
    teste(”Jose", ”Silva", "Impacta")
    teste(”jose")
    teste(”Jose", faculdade = "Impacta")
}
```

- Mais: https://kotlinlang.org/docs/reference/functions.html#named-arguments

Tipos Genéricos

- Um tipo genérico serve, por exemplo, para criar funções que podem fazer a mesma coisa para tipos diferentes
  - Por exemplo, criar uma lista. Independente do tipo, o algoritmo de criar a lista é o mesmo

```
fun main(args: Array<String>) {
    val strings = toList<String>("ADS","BD","GTI")
    println(strings)
    val ints = toList<Int>(1,2,3,4,5)
    println(ints)
}

fun <T> toList(vararg args: T): List<T>{
    val list = ArrayList<T>()
    for (s in args)
        list.add(s)
    return list
}
```

- Mais: https://kotlinlang.org/docs/reference/generics.html

Funções – varargs

- Uma função que tem um parâmetro varargs (normalmente o último) pode receber um ou mais parâmetros, separados por vírgula
  - Utiliza-se a palavra reservada vararg antes do parâmetro

```
fun main(args: Array<String>) {
    var list = toList("ADS","BD","GTI")
    print(list)
}

fun toList(vararg args: String): List<String>{
    val list = ArrayList<String>()
    for (s in args)
        list.add(s)
    return list
}
```

- Mais: https://kotlinlang.org/docs/reference/functions.html#variable-number-of-arguments-varargs

Classes

- Para criar uma classe basta utilizar a palavra reservada class
- O construtor padrão é definido na mesma linha da classe

```
class Aluno(nome: String, idade: Int) {
    val nome: String
    val idade: Int
    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }
    override fun toString(): String {
        return "Aluno: $nome, idade: $idade"
    }
}

fun main(args: Array<String>) {
    val a1 = Aluno("Fernando", 20)
    println(a1)
    println("Aluno: ${a1.nome}, idade: ${a1.idade}")
}
```

Classes

- Não é preciso utilizar new para criar uma instância da classe
- O método toString foi sobrescrito para ser chamado sempre que a classe for convertida em String
- Mais: https://kotlinlang.org/docs/reference/classes.html

Herança

- Para utilizar herança basta utilizar : e o nome da classe mãe

```
open class Pessoa(nome: String, idade: Int) {
    val nome: String
    val idade: Int
    init {// bloco de inicialização
        this.nome = nome
        this.idade = idade
    }
    open fun adicionarDesconto(desconto: Int) {
              println("Desconto para a pessoa de $desconto")
    }
    override fun toString(): String {
        return "Pessoa: $nome, idade: $idade"
    }
}
class Aluno(nome: String, idade: Int): Pessoa(nome, idade) {
    override fun adicionarDesconto(desconto: Int) {
              println("Desconto para o aluno de $desconto")
    }
}
fun main(args: Array<String>) {
    val a1 = Aluno("Fernando", 20)
    println(a1)
    println("Aluno: ${a1.nome}, idade: ${a1.idade}")
    a1.adicionarDesconto(10)
}
```

Herança

- Para que uma classe possa ser herdada ela deve ser marcada como open
  - Todas as classes são final por padrão
  - A mesma regra vale para métodos

- Mais: https://kotlinlang.org/docs/reference/classes.html

Data Classes

- Data Classes são classes que contém somente informações
- Em Kotlin uma Data Class é criada apenas com uma linha

```
data class Aluno(val nome: String)
```

- Essa linha cria a classe Aluno com:
  - Atributos do construtor
  - Getters e setters
  - Métodos equals, toString e copy

- Mais: https://kotlinlang.org/docs/reference/data-classes.html

Listas

- Lista ordenada: objeto ArrayList
- Lista mutável: função mutableListOf<T>(elementos) → recomendado
- Lista imutável: função listOf<T>(elementos)

```
fun main(args: Array<String>) {
    // lista com ArrayList
    var arrayListInt = ArrayList<Int>()
    arrayListInt.add(1)
    arrayListInt.add(2)
    arrayListInt.add(3)
    println(arrayListInt)

    // mesma lista utilizando mutableListOf
    var mutableListInt = mutableListOf<Int>(1,2,3)
    println(mutableListInt)
    // adicionar
    mutableListInt.add(4)
    println(mutableListInt)

    // lista imutável
    var listString = listOf<String>("a","b","c")
    println(listString)
    // adicionar - erro
    listString.add("d")
    println(listString)
}
```

- Mais: https://kotlinlang.org/docs/reference/collections.html

Enum

- Objeto com constantes predefinidas

```
//enum simples
enum class Status {ATIVO, INATIVO}

// enum com construtor
enum class Conceitos(val nota: String) {
    APROVADO("A"),
    RECUPERACAO("B"),
    REPROVADO("C")
}

fun main(args: Array<String>) {
    val status = Status.ATIVO
    println(status)
    val conceito = Conceitos.APROVADO.nota
    println(conceito)
}
```

- Mais: https://kotlinlang.org/docs/reference/enum-classes.html

Higher-Order Functions e Lambdas

- Traz produtividade na escrita dos códigos
- Kotlin: paradigma funcional
  - Uma função pode receber como parâmetro e retorne outra função
  - Higher-order Functions

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)): List<Int> {
    val newList = arrayListOf<Int>()
    for (item in list) {
        if (filtro(item)) {
            newList.add(item)
        }
    }
    return newList
}
fun numerosPares(numero: Int) = numero % 2 == 0
fun numerosMaioresQue3(numero: Int) = numero > 3

fun main(args: Array<String>) {
    var ints = listOf(1,2,3,4,5)
    println(ints)
    var list = filtrar(ints, ::numerosPares)
    println(list)
    list = filtrar(ints, ::numerosMaioresQue3)
    println(list)
}
```

Higher-Order Functions e Lambdas

- filtro: (Int) -> (Boolean) Informa que a função enviada como parâmetro para filtro recebe um inteiro e retorna um booleano
- Para enviar a função utiliza-se a sintaxe ::funcao

Higher-Order Functions e Lambdas

- Lambda é uma sintaxe simplificada para enviar uma função como parâmetro
- A passagem e feita utilizando chaves

```
fun main(args: Array<String>) {
    var ints = listOf(1,2,3,4,5)
    println(ints)
    // forma 1
    var list = filtrar(ints, {numero:Int -> numerosPares(numero)})
    println(list)
    // forma 2
    list = filtrar(ints, {numerosMaioresQue3(it)})
    println(list)
    // forma 3
    list = filtrar(ints) {numerosMaioresQue3(it)}
    println(list)
}
```

Funções anônimas

- Com lambda é possível enviar uma função sem nem mesmo cria-la
  - Funções anônimas

```
fun filtrar(list: List<Int>, filtro: (Int) -> (Boolean)): List<Int> {
    val newList = arrayListOf<Int>()
    for (item in list) {
        if (filtro(item)) {
            newList.add(item)
        }
    }
    return newList
}

fun main(args: Array<String>) {
    var ints = listOf(1,2,3,4,5)
    println(ints)
    var list = filtrar(ints) { it % 2 == 0}
    println(list)
    list = filtrar(ints) {it > 3}
    println(list)
}
```

- Mais: https://kotlinlang.org/docs/reference/lambdas.html

Referências e créditos

- Aula baseada nos textos do livro:
  - LECHETA, R. R. Android Essencial com Kotlin. Edição: 1ª ed. Novatec, 2017.

Obrigado!