Desenvolvimento Mobile

Introdução ao Kotlin

Prof. Dr. Marcelo Otone Aguiar

Universidade Federal do Espírito Santo - UFES

31 de Outubro de 2024

Conteúdo

Conceitos Gerais

Porque escolher o Kotlin?

- Interoperabilidade com Java
- É uma linguagem concisa
- Possui proteção contra nulos

Interoperabilidade

Classe com código em Java

```java
public class Calculadora {
    public int somar(int a, int b) {
        return a + b;
    }
}
```

Invocação da classe em Kotlin

```kotlin
val calc = Calculadora()
val resultado = calc.somar(2, 2)
println(resultado)
```

Concisa

Exemplo de definição do click de um botão em Java

```
1  Button buttonlogin = (Button) findViewById(R.id.button_login);
2
3  buttonlogin.setOnClickListener(new View.OnClickListener() {
4      @Override
5      public void onClick(View view) {
6          efetuarLogin();
7      }
8  });
```

Código equivalente em Kotlin

```
1  val buttonlogin = findViewById<Button>(R.id.button_login)
2
3  buttonlogin.setOnClickListener {
4       efetuarLogin()
5  }
```

Concisa

Exemplo de classe definida em Java

```java
public class Cliente {
    private String email;
    private String nomeUsuario;

    public Cliente(String email, String nomeUsuario) {
        this.email = email;
        this.nomeUsuario = nomeUsuario;
    }
    public String getNomeUsuario() {
        return nomeUsuario;
    }
    public void setNomeUsuario(String nomeUsuario) {
        this.nomeUsuario = nomeUsuario;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

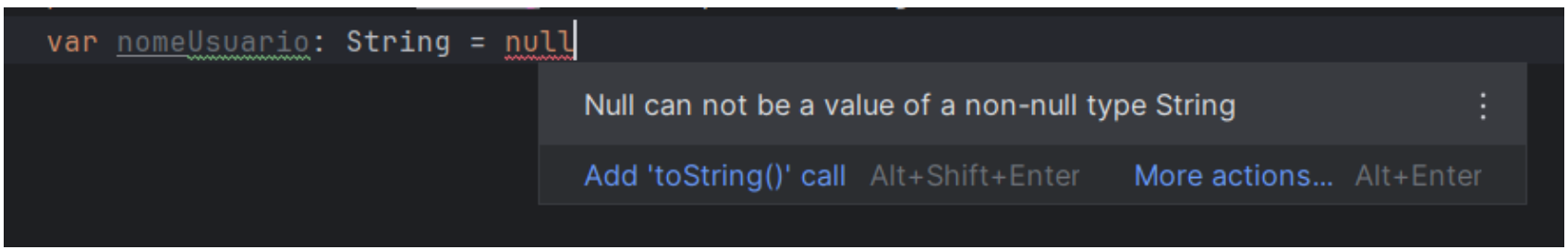
Concisa

Exemplo de classe definida em Kotlin

```
1 data class Cliente(var email: String, var nomeUsuario: String)
```

Proteção Contra Nulos

Em Kotlin nenhuma variável ou objeto pode ter um valor nulo



Mas se realmente for necessário, basta adicionar o operador **?**, contudo será necessário validar antes do uso.

```
1  var nomeUsuario: String? = null
2
3  if (nomeUsuario != null) {
4      println(nomeUsuario.length)
5  }
```

Definição de variáveis

Para criar variáveis em Kotlin, utilizamos a palavra reservada **var** seguida da definição de seu tipo, exemplo:

```
1 var idade: Int = 22
```

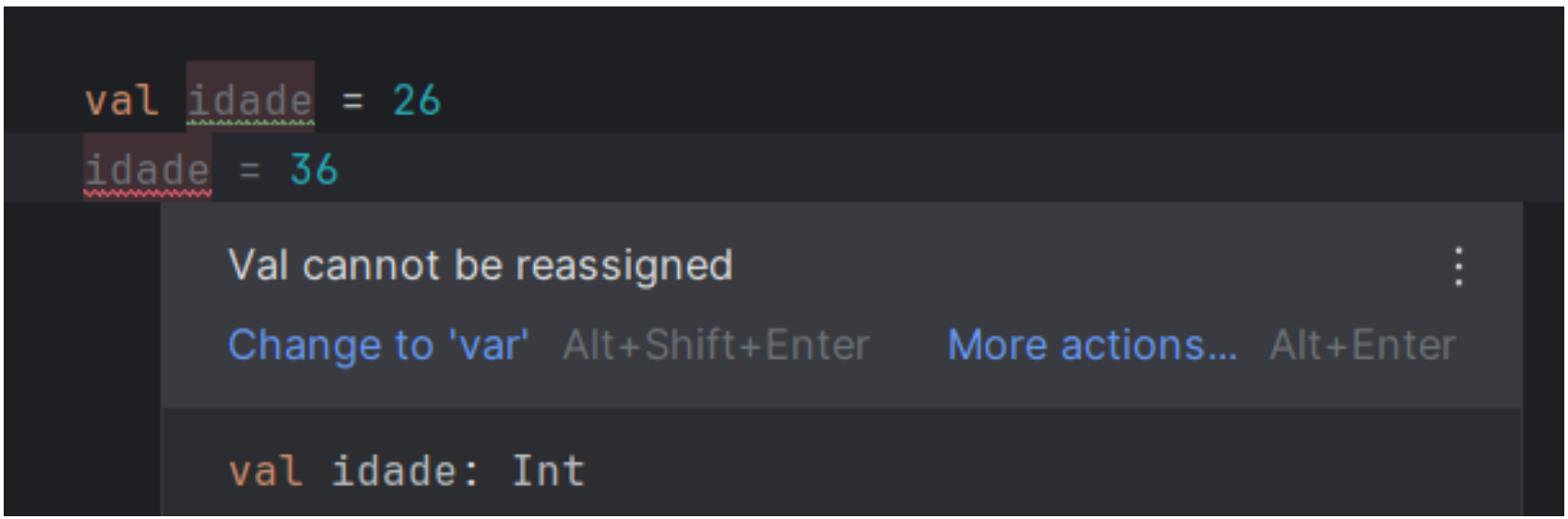
Em Kotlin, é obrigatório que a variável tenha um valor inicial, porém não é obrigatória a indicação de seu tipo. Assim, o código poderia ser escrito da seguinte forma:

```
1 var idade = 22
```

Definição de variáveis

Utilizamos a palavra reservada **val** quando queremos definir uma variável imutável (constante), exemplo:

```
1  val idade: Int = 26
```



Definição de variáveis

- Quando devo definir como *val* e quando devo definir como *var*?
- Por padrão, defina suas variáveis como val e, caso haja necessidade, mude para var.
- Isso melhora o resultado final do seu código em questão de segurança e clareza
- **Segurança:** você garante que variáveis imutáveis não terão seu valor trocado
- **Clareza:** ficam claras as intenções de *design* utilizadas na construção do código

Definição de variáveis

Tipo	Descrição	Memória
Double	Valores de ponto flutuante.	64 bits
Float	Valores de ponto flutuante.	32 bits
Int	Valores inteiros.	32 bits
Long	Valores inteiros.	64 bits
Short	Valores inteiros.	16 bits
Byte	Valores inteiros.	8 bits
String	É um tipo para guardar texto.	-
Char	Permite guardar um único caractere.	8 bits
Boolean	É usado para valores booleanos.	8 bits

Tipos de Dados

- Em **Kotlin** tudo é objeto.
- Na prática, isso quer dizer que todas as variáveis e tipos que usamos em **Kotlin** possuem propriedades e métodos.
- Como exemplo, se compararmos com o tipo *int* do **Java**, veremos que ambos servem para guardar números inteiros, porém com funcionamento diferente.
- O tipo *int* do **Java** é primitivo, já em **Kotlin** é um objeto, se assemelhando ao tipo *Integer* do **Java**.

Tipos de Dados

- Um exemplo são os métodos de conversão de tipos.
- Como o *int* em **Kotlin** é um objeto, ele possui métodos que permitem realizar as conversões a seguir:

```
1  val numero: Int = 35
2
3  var x: Double = numero.toDouble()
4  var y: Float = numero.toFloat()
5  var z: String = numero.toString()
```

Tipos de Dados: string

- Uma **string** é um tipo de dado utilizado para guardar conteúdos do tipo texto.
- Esse conteúdo deve ser definido com aspas duplas:
- Exemplo:

```
1 val mensagem = "Seja bem-vindo ao sistema"
```

Tipos de Dados: string

- Para facilitar o trabalho com **strings**, o **Kotlin** possui o recurso *templates*.
- *Templates de strings* são trechos de código inseridos diretamente em uma *string* para sua concatenação.
- Basta utilizar o operador **$**
- Exemplo:

```
1  val apelidoUsuario = "Pedro"
2
3  val mensagem = "Bem-vindo $apelidoUsuario"
4  println(mensagem)
```

Tipos de Dados: string

- Uma boa prática é, além de colocar o operador **$**, indicar o nome do objeto entre chaves.
- Exemplo:

```
1 val apelidoUsuario = "Pedro"
2
3 val mensagem = "Bem-vindo ${apelidoUsuario}"
4 println(mensagem)
```

Tipos de Dados: booleano

- O tipo booleano é definido como *Boolean*, e só pode receber valores *true* (verdadeiro) ou *false* (falso).
- As operações possíveis com o booleano são:

Operação	**Operador**
E	&&
OU	||
NÃO	!

Tipos de Dados: booleano

- Mas, como o *booleano* é um objeto, ele possui métodos para realizar as operações de **E**, **OU** e **NÃO**.
- Exemplo:

```
1  val teste1 = true
2  val teste2 = false
3
4  val res1 = teste1.and(teste2)
5  val res2 = teste1.or(teste2)
6  val res3 = teste1.not()
7
8  print("$res1 $res2 $res3")
```

Tipos de Dados: Arrays

- Em **Kotlin** os *arrays* também possuem tamanho fixo.
- Similar a outras linguagens, também é possível obter seus valores por índice.
- Exemplo:

```
1  val numerosInt: Array<Int> = arrayOf(30, 23, 13, 56, 21)
2
3  val numInt = numerosInt[2]
4
5  println(numInt)
```

Tipos de Dados: Listas

- Para as listas o tamanho não precisa ser previamente definido.
- Novos valores podem ser adicionados à lista conforme necessário.
- Contudo, em **Kotlin** é possível declarar listas **mutáveis** e **não mutáveis**.
- Exemplo:

```
1  val listaEstatica = listOf(55, 42, 21, 18)
2
3  val listaDinamica = mutableListOf(55, 42, 21, 18)
```

Tipos de Dados: Listas

- As listas **mutáveis** e **não mutáveis** disponibilizam vários métodos sobre elas.
- Dentre os métodos disponíveis para listas **mutáveis**, apenas o método *add()* não está disponível em uma lista **não mutável**.
- Exemplos:

```
1  val listaDinamica = mutableListOf(55, 42, 21, 18)
2
3  listaDinamica.add(5)
4  val primeiroValor = listaDinamica.first()
5  val ultimoValor = listaDinamica.last()
6  val valoresPares = listaDinamica.filter { it % 2 == 0 }
```

Estruturas de Decisão

- O *IF* funciona similar a outras linguagens.
- A instrução *ELSE* não é obrigatória.
- Condições compostas são admitidas.
- Condições aninhadas são admitidas.
- Exemplo:

```
1  val nota = 8
2
3  if (nota >= 7)
4      println("Aprovado")
5  else if (nota >= 5 && nota < 7)
6      println("Em recuperacao")
7  else
8      println("Reprovado")
```

Estruturas de Decisão

- O *IF* também pode ser implementado como operador ternário, sendo o seu resultado armazenado em uma variável.
- No exemplo a seguir, note que, se *valor1* é menor que *valor2*, então o conteúdo de *valor1* é armazenado na constante *menor*.
- Caso contrário, o conteúdo de *valor2* é armazenado na constante *menor*.
- Assim, a constante *menor* possui armazenado o resultado da condição analisada na instrução *if*.

```
1    val valor1 = 8
2    val valor2 = 4
3
4    val menor = if (valor1 < valor2) valor1 else valor2
5    println("Menor: ${menor}")
```

Estruturas de Decisão

- O **Kotlin** disponibiliza também a estrutura de decisão *WHEN* que possui um funcionamento similar ao *switch* em outras linguagens.
- Veja o exemplo de uso:

```
1  val opcao = 3
2  when (opcao) {
3      1 -> print("Escolheu a opcao 1")
4      2 -> print("Escolheu a opcao 2")
5      3 -> print("Escolheu a opcao 3")
6  else -> {
7      print("Escolheu uma opcao invalida!")
8      }
9  }
```

Estruturas de Decisão

Exemplo de validação de mais de um valor em uma condição.

```
1  val opcao = 3
2  when (opcao) {
3      1, 2 -> print("Escolheu a opcao 1 ou 2")
4      3, 4 -> print("Escolheu a opcao 3 ou 4")
5      5 -> print("Escolheu a opcao 5")
6  else -> {
7      print("Escolheu uma opcao invalida!")
8      }
9  }
```

Exemplo de validação de faixa de valores em uma condição.

```
1  val opcao = 3
2  when (opcao) {
3      in 1..5 -> print("Escolheu a opcao no intervalo de 1 a 5")
4      in 6..10 -> print("Escolheu a opcao no intervalo de 6 a 10")
5  else -> {
6      print("Escolheu uma opcao invalida!")
7      }
8  }
```

Estruturas de Decisão: Operadores Relacionais e de Identidade

Operador	**Descrição**
==	Igual a
!=	Diferente
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
===	Verifica se duas referências apontam para o mesmo objeto (identidade referencial).
!==	Verifica se duas referências **não** apontam para o mesmo objeto.

Estruturas de Repetição

- Em **Kotlin** o *for* deve ser implementado um pouco diferente das linguagens da qual ele descende.
- O Código a seguir mostra a forma como estamos acostumados a utilizar o for na linguagem **Java**, **C** e outras da família:

```
1 for (int i = 1; i < 10; i++) {
2     System.out.println("Numero: " + i);
3 }
```

No código a seguir, pode-se ver a implementação equivalente em **Kotlin**:

```
1 for (i in 1 until 10) {
2     println("Numero: ${i}")
3 }
```

Estruturas de Repetição

Exemplo do uso do *for* iterando em uma lista.

```
1  val lista = listOf("Pedro","Paulo","Sandra","Katia")
2
3  for (nome in lista) {
4      println("Nome: ${nome}")
5  }
```

É possível obter o índice também. Veja o exemplo:

```
1  val lista = listOf("Pedro","Paulo","Sandra","Katia")
2
3  for ((matricula, nome) in lista.withIndex()) {
4      println("Matricula: ${matricula+1} Nome: ${nome}")
5  }
```

Estruturas de Repetição

Exemplo de uso da estrutura *while*.

```
1  var temperatura = 0
2  while (temperatura < 50) {
3      println(temperatura.toString())
4      temperatura+=10
5  }
```

Exemplo de uso da estrutura *do while*.

```
1  var temperatura = 0
2  do {
3      println(temperatura.toString())
4      temperatura+=10
5  } while (temperatura < 50)
```

Funções

- Para definir funções em **Kotlin**, utilizamos a palavra reservada *fun*, seguida do nome da função, seus argumentos e seu retorno.
- No exemplo a seguir, foi definida a função *soma()* e, note que, a mesma é invocada na função *main* e o seu **retorno** é armazenado na constante *res*.

```
1  fun main() {
2      val res = soma(10, 20)
3      println("Resultado: ${res}")
4  }
5
6  fun soma(n1: Int , n2: Int): Int {
7      return n1 + n2
8  }
```

Funções

- Naturalmente, há casos em que a função não precisa de retorno.
- Nesses casos, ao invés de declarar o tipo de retorno, declare com a palavra reservada *Unit* ou simplesmente não defina o retorno, como no exemplo a seguir.

```
1  fun main() {
2      val res = soma(10, 20)
3      imprimir("Resultado: ${res}")
4  }
5
6  fun soma(n1: Int , n2: Int): Int {
7      return n1 + n2
8  }
9
10 fun imprimir(texto: String) {
11     println(texto)
12 }
```

Funções

- O Kotlin disponibiliza também um recurso chamado ***single expression functions*** para uso com funções de instrução única.
- Veja o exemplo anterior adaptado para este formato.

```
1  fun main() {
2      val res = soma(10, 20)
3      imprimir("Resultado: ${res}")
4  }
5
6  fun soma(n1: Int , n2: Int) = n1 + n2
7
8  fun imprimir(texto: String) = println(texto)
```

Definindo Classes

Para criar uma classe em **Kotlin**, utilizamos a palavra **class** seguida pelo nome da classe. E para definir suas propriedades, podemos criar variáveis. E os métodos são definidos por funções.

```
1  class Produto {
2      var codigo: Int = 0
3      var descricao: String = ""
4
5      fun cadastrar() {
6
7      }
8
9      fun consultar() {
10
11     }
12 }
```

Modificadores de Visibilidade

Modificador	**Descrição**
public (padrão)	A classe é acessível de qualquer lugar no projeto.
private	A classe é acessível apenas dentro do arquivo em que está definida.
protected	A classe é acessível apenas dentro da classe que a declara e suas subclasses. No entanto, *protected* não pode ser usado em nível de arquivo, apenas em classes.
internal	A classe é acessível apenas dentro do mesmo módulo. Um módulo é um conjunto de arquivos **Kotlin** que são compilados juntos.

Instanciar uma Classes (Criar um Objeto)

Exemplo de classe **com** atributos no construtor

```
1  fun main() {
2      val meuCarro = Carro("Fusca", 1975)
3
4      meuCarro.ligar()
5      meuCarro.detalhes()
6
7      meuCarro.ano = 1976
8      meuCarro.detalhes()
9  }
10
11 class Carro(val modelo: String, var ano: Int) {
12     fun ligar() {
13         println("$modelo esta ligado.")
14     }
15
16     fun detalhes() {
17         println("Modelo: $modelo, Ano: $ano")
18     }
19 }
```

Instanciar uma Classes (Criar um Objeto)

Exemplo de classe **sem** atributos no construtor

```kotlin
fun main() {
    val meuCarro = Carro()
    meuCarro.modelo = "Fusca"
    meuCarro.ano = 1975
    meuCarro.ligar()
    meuCarro.detalhes()
}

class Carro {
    var modelo: String = "Desconhecido"
    var ano: Int = 0

    fun ligar() {
        println("$modelo esta ligado.")
    }

    fun detalhes() {
        println("Modelo: $modelo, Ano: $ano")
    }
}
```

Herança

Para herdar uma classe em **Kotlin**, colocamos **:** na frente do nome da classe e, em seguida, instanciamos a classe pai.

```
1  import java.time.LocalDate
2
3  class produtoPerecivel: Produto() {
4      var dataValidade: LocalDate = LocalDate.now()
5
6      fun atualizarDataValidade(novaData: LocalDate) {
7          dataValidade = novaData
8      }
9  }
```

Herança

Mas, para isso funcionar, a classe Produto deve permitir explicitamente que se faça herança dela. Para isso, devemos utilizar a palavra reservada **open** na definição da classe.

```
1  import java.time.LocalDate
2
3  open class Produto {
4      var codigo: Int = 0
5      var descricao: String = ""
6      fun cadastrar() {
7
8      }
9      fun consultar() {
10
11     }
12 }
13
14 class produtoPerecivel: Produto() {
15     var dataValidade: LocalDate = LocalDate.now()
16     fun atualizarDataValidade(novaData: LocalDate) {
17         dataValidade = novaData
18     }
19 }
```

Classe de Dados

- **Classes de dados** geralmente não possuem nenhum método, somente propriedades, e permitem organizar os dados em uma aplicação.
- Em Kotlin as classes de dados possuem implementação simplificada, bastando indicar a palavra reservada ***data*** antes de ***class***.

```
1 data class Cliente {val nome: String, val email: String, val endereco: String}
```