

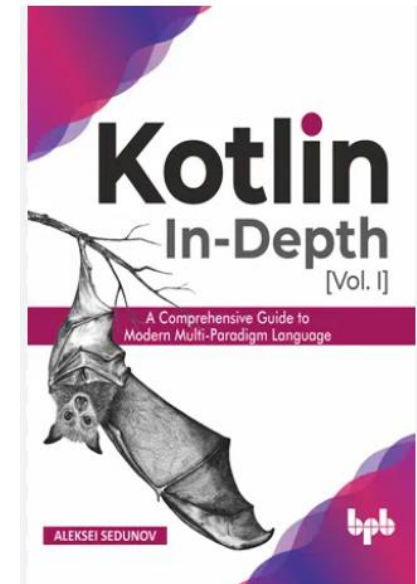
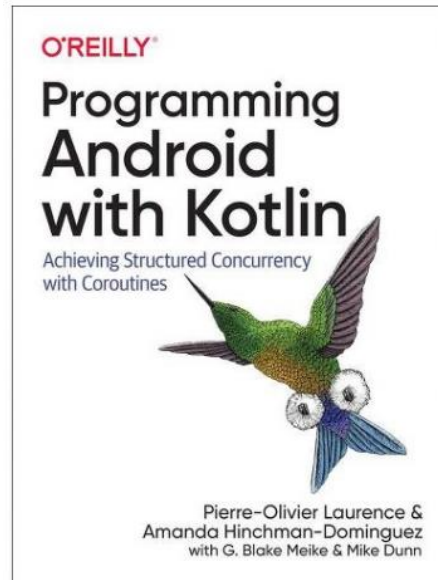
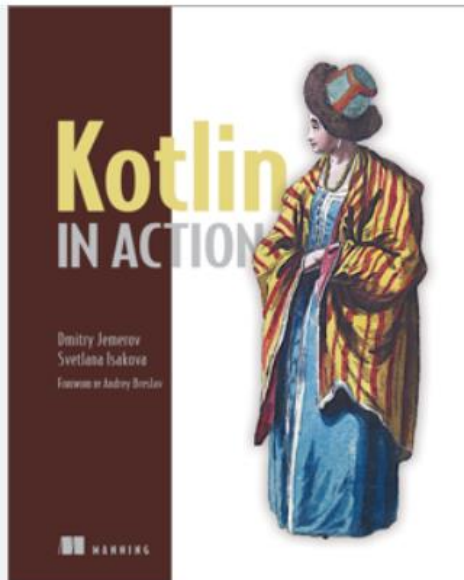
Desenvolvimento de Aplicativos

Unidade 2 – A linguagem Kotlin



Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@online.uscs.edu.br
aparecidovfreitas@gmail.com

Bibliografia



A linguagem Kotlin

- ⊕ **Kotlin** não é muito diferente de Java;
- ⊕ **Kotlin** introduziu algumas novas funcionalidades de linguagem, mas há mais semelhanças do que diferenças entre **Kotlin** e **Java**;
- ⊕ Para o programador **Java**, a curva de aprendizado para **Kotlin** não é íngreme.



Literals

- ⊕ São notações para se representar valores fixos no código;
- ⊕ No código abaixo, **5**, **.02**, **"Hello"**, **`1`** e **true** são literais.

Kotlin_02_1.kt

```
1 fun main() {  
2  
3     var a = 5  
4     var b = .02  
5     var c = "Hello"  
6     var d = '1'  
7     var e = true  
8  
9     println (a);  
10    println (b);  
11    println (c);  
12    println (d);  
13    println (e);  
14 }
```

Integer

Double

String

Character

Boolean

[Done] exited with code=0 in 3.517 seconds

[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_D
-include-runtime -d Kotlin_02_1.jar && java

5
0.02
Hello
1
true

Variáveis

- ⊕ São declaradas por meio de um identificador usando a palavra-chave **var** seguida pelo seu tipo.

◀ Kotlin_02_2.kt

```
1  fun main() {  
2      //-----  
3      var a: Int  
4      a = 99  
5      println (a)  
6      //-----  
7      var b: Int = 33;  
8      println(b)  
9      //-----  
10     var c = 55;  
11     println(c)  
12 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_U  
-include-runtime -d Kotlin_02_2.jar  
99  
33  
55
```

Variáveis

- ⊕ Podem também ser declaradas com a palavra-chave **val**;
- ⊕ Neste caso podem ser inicializadas apenas uma vez dentro do bloco de execução onde foram definidas;
- ⊕ Isso as torna efetivamente constantes e equivalem à palavra-chave **final** em Java.

```
Kotlin_02_3.kt
1  fun main() {
2      //-----
3      val a: Int
4      a = 99
5      println (a)
6      //-----
7      a = 55
8      println(a)
9  }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_Aplicac
-include-runtime -d Kotlin_02_3.jar && java -jar Koti
Kotlin_02_3.kt:7:2: error: val cannot be reassigned
    a = 55
    ^
```

Expressões em Kotlin

- ⊕ Uma expressão combina operadores, funções, valores literais, variáveis ou constantes e sempre é **avaliada para um valor**;
- ⊕ Ela também pode ser parte de uma expressão mais complexa;



1. Expressões Aritméticas:

- Soma: ``val soma = 5 + 3`` (resulta em ``8``)
- Multiplicação: ``val produto = 4 * 2`` (resulta em ``8``)

2. Expressões de Comparação:

- Igualdade: ``val isEqual = (5 == 5)`` (resulta em ``true``)
- Maior que: ``val isGreater = (5 > 3)`` (resulta em ``true``)

3. Expressões Lógicas:

- E lógico: ``val andResult = (true && false)`` (resulta em ``false``)
- Ou lógico: ``val orResult = (true || false)`` (resulta em ``true``)

Expressões em Kotlin

Kotlin_02_4.kt

```
1 fun main() {  
2     //-----  
3     val a: Int  
4     a = 5 + 3  
5     println (a)  
6     //-----  
7     val b: Boolean  
8     b = (9 == 9)  
9     println(b)  
10 }
```

```
[Running] cd "d:\USCS\  
-include-runtime -d Ko  
8  
true
```


Declarações em Kotlin

- ✦ Declarações podem conter expressões, mas por si só, não são avaliadas para um valor;
- ✦ Declarações são instruções que definem variáveis, funções, classes ou outros elementos do código;
- ✦ Diferentemente de expressões, que sempre retornam um valor, as declarações **servem para estruturar o programa** e **não** produzem um valor diretamente.


1. Declaração de Variáveis:

- Usando ``val`` (imutável): ``val` nome = "João"```
- Usando ``var`` (mutável): ``var` idade = 30``

2. Declaração de Funções:

- Uma função simples:

kotlin

 Copy code

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



Atribuições em Kotlin

- ⊕ Atribuições são operações onde um valor é atribuído a uma variável;
- ⊕ A atribuição é fundamental em qualquer linguagem de programação, pois permite que se armazene e se manipule dados em variáveis;
- ⊕ O operador de atribuição em **Kotlin** é o sinal de igual (=).



Atribuições em Kotlin

1. Atribuição Básica:

- ``var a = 5`` - A variável ``a`` é inicializada com o valor ``5``.
- ``val b = "Kotlin"`` - A variável ``b`` imutável é inicializada com a string ``"Kotlin"``.

2. Reatribuição:

- ``a = 10`` - O valor da variável ``a`` é atualizado para ``10`` (isso só é possível porque ``a`` foi declarado como ``var``, ou seja, é uma variável mutável).

3. Atribuição com Operações:

- ``a += 2`` - Equivalente a ``a = a + 2``, isso adiciona ``2`` ao valor atual de ``a``.
- ``a *= 3`` - Equivalente a ``a = a * 3``, isso multiplica o valor atual de ``a`` por ``3``.



Atribuições em Kotlin

4. Atribuição a Propriedades de Objetos:

- `usuario.nome = "João"` - Atualiza a propriedade `nome` do objeto `usuario` para `"João"`.

5. Atribuição a Elementos de Array ou Coleção:

- `array[0] = 1` - Define o primeiro elemento de um array para `1`.
- `lista[2] = "Kotlin"` - Define o terceiro elemento de uma lista para `"Kotlin"`.





Então, atribuições em Java e Kotlin
são feitas da mesma forma ?

Atribuições em Java

- ⊕ Em **Java**, as atribuições podem retornar um **valor**;
- ⊕ Quando se realiza uma atribuição, o valor da atribuição é o valor que foi atribuído à variável;
- ⊕ Isso permite que atribuições sejam usadas em expressões maiores ou mesmo em outras atribuições.



Esse código Java compila ?

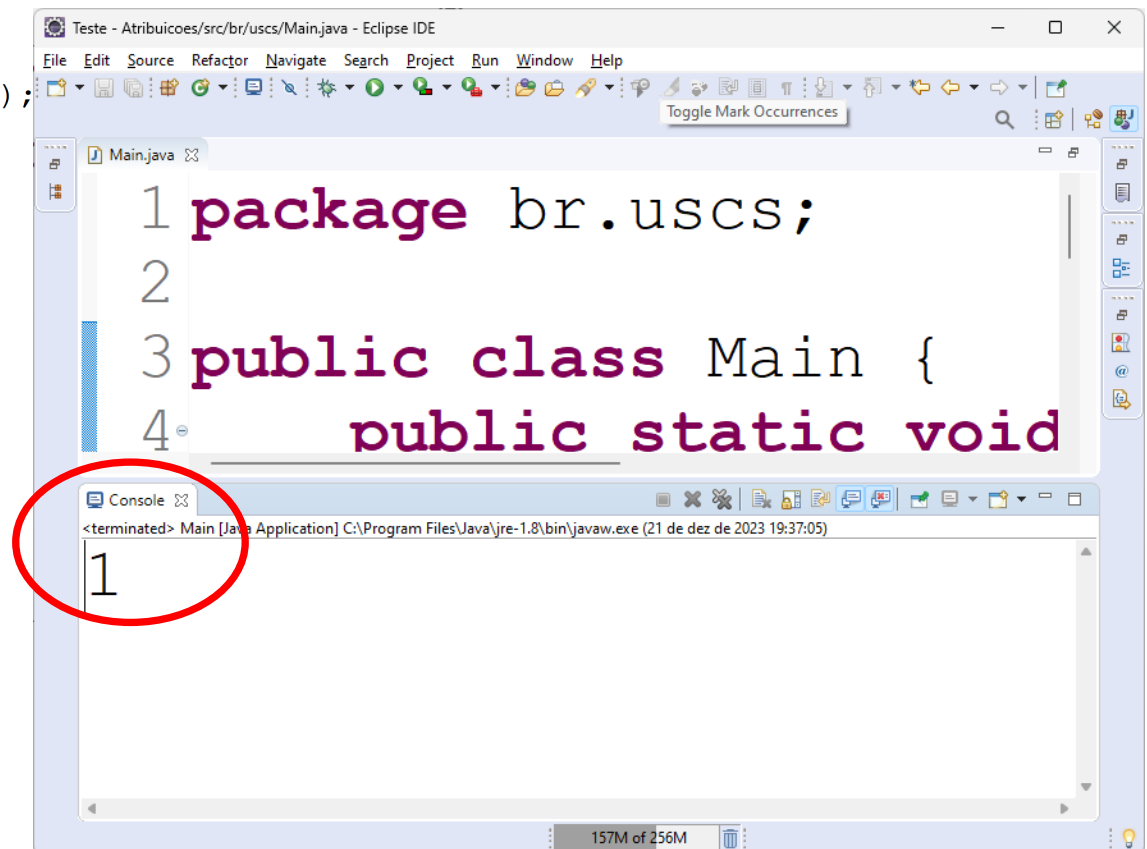
```
1 package br.uscs;
2
3 public class Main {
4     public static void main(String[] args) {
5         int x, a = 10, b = 3;
6
7         while ( (x = a % b) != 0) {
8             a = b;
9             b = x;
10        }
11        System.out.println(b);
12    }
13 }
```



Esse código Java compila ?



```
1 package br.uscs;
2
3 public class Main {
4     public static void main(String[] args) {
5         int x, a = 10, b = 3;
6
7         while ((x = a % b) != 0) {
8             a = b;
9             b = x;
10        }
11        System.out.println(b);
12    }
13 }
```



O mesmo código compila em Kotlin?

```
Kotlin_02_5.kt
1  fun main() {
2      //-----
3      var x: Int
4      var a = 10
5      var b = 3
6
7      while ((x = a % b) != 0) {
8          a = b
9          b = x
10     }
11     println(b)
12 }
```

O mesmo código compila em Kotlin?

Kotlin_02_5.kt

```
1 fun main() {  
2     //-----  
3     var x: Int  
4     var a = 10  
5     var b = 3  
6  
7     while ((x = a % b) != 0) {  
8         a = b  
9         b = x  
10    }  
11    println(b)  
12 }
```

PERIGO



```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_Aplicativos\For  
-include-runtime -d Kotlin_02_5.jar && java -jar Kotlin_02_5.  
Kotlin_02_5.kt:7:10: error: assignments are not expressions,  
this context
```

```
    while ((x = a % b) != 0) {  
            ^
```

Em Kotlin atribuições não retornam valor !!!!



Qual a razão ?

Atribuições em Kotlin

Em Kotlin, atribuições não retornam valor como parte da filosofia de design da linguagem que prioriza clareza e segurança. Aqui estão algumas razões específicas para essa escolha:

1. **Clareza e Legibilidade:** Kotlin enfatiza a clareza e a legibilidade do código. Ao evitar que as atribuições retornem um valor, a linguagem reduz a possibilidade de escrever código confuso ou difícil de entender. Em linguagens onde atribuições retornam valores, como C ou Java, é comum ver expressões complexas que combinam atribuições e outras operações, o que pode dificultar a compreensão do que o código está fazendo.
2. **Prevenção de Erros:** Evitar que atribuições retornem valores ajuda a prevenir certos tipos de erros comuns em programação. Por exemplo, em C, é fácil confundir o operador de atribuição (`=`) com o operador de igualdade (`==`) em uma instrução condicional. Em Kotlin, essa fonte de erros é eliminada, pois a atribuição em si não pode ser usada como uma expressão booleana.

Atribuições em Kotlin

3. **Imutabilidade e Segurança de Tipos:** Kotlin incentiva o uso de variáveis imutáveis (declaradas com ``val``) e possui um sistema de tipos forte. Ao limitar o comportamento das atribuições, Kotlin reforça essas práticas de codificação segura e confiável.
4. **Consistência com Outras Convenções da Linguagem:** Kotlin tem várias outras convenções e características que se alinham com a decisão de não permitir que as atribuições retornem valores, como inferência de tipo e a distinção clara entre expressões e declarações.

Em resumo, essa escolha de design em Kotlin visa tornar o código mais seguro, mais claro e mais fácil de manter, alinhando-se com os objetivos gerais da linguagem de ser concisa, expressiva e segura em termos de tipos.

Revisitando o código para Kotlin

```
1 package br.uscs;
2
3 public class Main {
4     public static void main(String[] args) {
5         int x, a = 10, b = 3;
6
7         while ((x = a % b) != 0) {
8             a = b;
9             b = x;
10        }
11        System.out.println(b);
12    }
13 }
```

Código Kotlin

Kotlin_02_6.kt

```
1  fun main() {  
2      var x: Int  
3      var a = 10  
4      var b = 3  
5      var verdade = true  
6  
7      while (verdade) {  
8          x = a % b  
9          if (x != 0) {  
10             a = b  
11             b = x  
12         }  
13         else  
14             verdade = false  
15     }  
16     println(b)  
17 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS" & java -jar  
-include-runtime -d Kotlin_02_6.jar  
1
```

Operadores Aritméticos

1. **Adição (`+`):** Soma dois valores.
 - Exemplo: `a + b`
2. **Subtração (`-`):** Subtrai um valor de outro.
 - Exemplo: `a - b`
3. **Multiplicação (`*`):** Multiplica dois valores.
 - Exemplo: `a * b`
4. **Divisão (`/`):** Divide um valor por outro. Em Kotlin, a divisão entre dois inteiros resulta em um inteiro. Para obter um resultado de ponto flutuante, pelo menos um dos números deve ser de ponto flutuante.
 - Exemplo: `a / b`
5. **Resto da Divisão (`%`):** Retorna o resto de uma divisão de um número por outro.
 - Exemplo: `a % b`

Operadores de Atribuição

1. **Adição atribuída (``+=``):** ``a += b`` é equivalente a ``a = a + b``
2. **Subtração atribuída (``-=``):** ``a -= b`` é equivalente a ``a = a - b``
3. **Multiplicação atribuída (``*=``):** ``a *= b`` é equivalente a ``a = a * b``
4. **Divisão atribuída (``/=``):** ``a /= b`` é equivalente a ``a = a / b``
5. **Resto da divisão atribuída (``%=``):** ``a %= b`` é equivalente a ``a = a % b``

Operadores Lógicos

Em Kotlin, os operadores lógicos são usados para formar expressões booleanas, que resultam em um valor `true` ou `false`. Os operadores lógicos básicos em Kotlin são:

1. **E Lógico (`&&`):** Retorna `true` se ambas as expressões booleanas forem verdadeiras.
 - Exemplo: `a && b`
2. **OU Lógico (`||`):** Retorna `true` se pelo menos uma das expressões booleanas for verdadeira.
 - Exemplo: `a || b`
3. **NÃO Lógico (`!`):** Inverte o valor de uma expressão booleana, ou seja, `!true` é `false` e `!false` é `true`.
 - Exemplo: `!a`

Esses operadores são frequentemente usados em estruturas de controle de fluxo, como instruções `if`, `while`, e `for`, para combinar, inverter ou comparar valores booleanos e controlar o fluxo de execução do programa.

Operadores de Igualdade

1. **Operador de Igualdade Estrutural (`==`):** Verifica se os valores de duas variáveis são iguais. Internamente, ele traduz para a chamada da função `equals()`. Por exemplo, `a == b` verifica se `a` é igual a `b` em termos de valor. É importante notar que esse operador não compara as referências dos objetos, mas sim seus conteúdos.
2. **Operador de Não Igualdade Estrutural (`!=`):** O oposto do operador de igualdade estrutural, verifica se dois valores não são iguais. Por exemplo, `a != b` verifica se `a` é diferente de `b` em termos de valor.

Operadores de Igualdade Referencial

Além disso, há operadores de igualdade referencial, usados para comparar as referências de objetos:

1. **Operador de Igualdade Referencial (``===``):** Verifica se duas referências de objeto apontam para o mesmo objeto. Por exemplo, ``a === b`` é verdadeiro se ``a`` e ``b`` referem-se exatamente ao mesmo objeto.
2. **Operador de Não Igualdade Referencial (``!==``):** Verifica se duas referências de objeto não apontam para o mesmo objeto. Por exemplo, ``a !== b`` é verdadeiro se ``a`` e ``b`` não referem-se ao mesmo objeto.

Esses operadores são fundamentais para comparações em Kotlin, permitindo verificar tanto a igualdade em termos de valor quanto em termos de referência de objeto.

Blocos

- ⊕ Frequentemente, pode ser preciso se escrever várias declarações e agrupá-las;
- ⊕ Pode-se fazer isso por meio de blocos.
- ⊕ Pode-se definir um bloco de código usando-se um par de chaves;
- ⊕ Blocos são encontrados em muitas construções Kotlin, como classes, funções, interfaces, loops, ramificações, etc.

```
fun greet(name:String) {  
    println("Hello $name")  
}
```

Comentários

- ⊕ Definidos da mesma forma como em Java;

```
// Isto é um comentário de linha única  
val numero = 5 // Isto também é um comentário
```

```
/* Isto é um comentário  
   de múltiplas linhas.  
   Pode abranger várias linhas. */  
val nome = "ChatGPT"
```

Tipos Básicos

- ⊕ **Kotlin** possui alguns tipos básicos, mas eles **não** são os mesmos que os tipos primitivos do Java, porque todos os tipos em **Kotlin** são objetos;
- ⊕ Eles são apenas chamados de tipos básicos porque são de uso muito comum;
- ⊕ Esses tipos são números, caracteres, booleanos, arrays e strings.



Tipos Numéricos

1. **Inteiro (``Int``)**: Representa um número inteiro de 32 bits. Exemplo de valor: ``123``.
2. **Long (``Long``)**: Representa um número inteiro de 64 bits, maior que ``Int``. Exemplo de valor: ``123L`` (o sufixo 'L' é usado para indicar um literal ``Long``).
3. **Curto (``Short``)**: Representa um número inteiro de 16 bits. Menos comum e usado principalmente para otimizar a memória em cenários específicos. Exemplo de valor: ``123.toShort()``.
4. **Byte (``Byte``)**: Representa um número inteiro de 8 bits. Semelhante ao ``Short``, é usado para economia de memória. Exemplo de valor: ``123.toByte()``.
5. **Ponto Flutuante (``Float``)**: Representa um número de ponto flutuante de 32 bits. Exemplo de valor: ``123.4f`` ou ``123.4F`` (o sufixo 'f' ou 'F' é usado para indicar um literal ``Float``).
6. **Duplo (``Double``)**: Representa um número de ponto flutuante de 64 bits, com maior precisão que ``Float``. Exemplo de valor: ``123.4`` (sem sufixo, é interpretado como ``Double``).

Conversão de Tipos numéricos

Em Kotlin, a conversão de um tipo numérico para outro é feita usando métodos específicos de conversão. Cada tipo numérico básico (como ``Int``, ``Long``, ``Float``, ``Double``, etc.) tem métodos para converter para outros tipos numéricos. Esses métodos são:

- ``toByte()``: converte o valor para ``Byte``
- ``toShort()``: converte o valor para ``Short``
- ``toInt()``: converte o valor para ``Int``
- ``toLong()``: converte o valor para ``Long``
- ``toFloat()``: converte o valor para ``Float``
- ``toDouble()``: converte o valor para ``Double``
- ``toChar()``: converte o valor para ``Char``

Conversão de Tipos numéricos

```
val intVal = 10
val doubleVal = intVal.toDouble() // Converte de Int para Double

val longVal = 100L
val floatVal = longVal.toFloat() // Converte de Long para Float

val doubleVal2 = 123.45
val intVal2 = doubleVal2.toInt() // Converte de Double para Int
```

Conversão de Tipos numéricos

É importante observar que ao converter de um tipo com maior capacidade de armazenamento ou precisão para um tipo com menor capacidade, como de `'Double'` para `'Int'`, pode haver perda de informação (por exemplo, a parte fracionária é perdida na conversão de `'Double'` para `'Int'`). Da mesma forma, converter um número grande para um tipo com menor capacidade de armazenamento pode resultar em overflow.

Tipos Caracteres

Em Kotlin, o tipo de dado para representar caracteres é ``Char``. Cada valor ``Char`` representa um caractere único e é definido usando aspas simples. Aqui estão alguns detalhes importantes sobre o tipo ``Char`` em Kotlin:

1. **Representação:** Um ``Char`` é representado por um único caractere dentro de aspas simples. Por exemplo: ``'a'``, ``'1'``, ``'%'``.
2. **Não é um Número:** Ao contrário de algumas outras linguagens, em Kotlin, ``Char`` não é tratado como um tipo numérico. Isso significa que você não pode diretamente tratar um ``Char`` como um número (diferente de linguagens como C, onde os caracteres são representados por valores numéricos ASCII).
3. **Operações com ``Char``:** Você pode realizar várias operações com ``Char``, como comparações (``==``, ``!=``, ``<``, ``>`` etc.) e algumas operações específicas de caracteres (como verificar se um caractere é uma letra ou um dígito). Exemplo: ``char.isDigit()``, ``char.isLetter()``.

Tipos Caracteres

4. **Código Unicode:** Cada `Char` em Kotlin representa um caractere UTF-16. Você pode obter o código numérico correspondente a um `Char` usando a função `toByte()`, `toShort()`, `toInt()`, ou `toLong()`.
5. **Escape Sequences:** Kotlin suporta sequências de escape em `Char`. Por exemplo, `'\n'` para quebra de linha, `'\t'` para tabulação, `'\b'` para backspace, `'\''` para aspas simples, `'\\'` para barra invertida, entre outros.
6. **Caracteres Unicode:** Você pode representar um caractere Unicode em Kotlin usando a notação de escape `\u` seguida por um código Unicode de quatro dígitos. Por exemplo, `'\u0041'` representa o caractere 'A'.

Em resumo, `Char` em Kotlin é usado para representar caracteres individuais de maneira simples e intuitiva, com suporte para operações básicas de caracteres e representação Unicode.

Tipos Booleanos

Em Kotlin, o tipo booleano é representado pela palavra-chave `Boolean`. Este tipo é usado para representar valores de verdade, que podem ser apenas dois: `true` (verdadeiro) ou `false` (falso). Os valores booleanos são frequentemente usados em controle de fluxo (como em instruções condicionais e loops) e na avaliação de expressões lógicas. Aqui estão alguns detalhes importantes sobre o tipo `Boolean` em Kotlin:

1. **Valores Literais:** Em Kotlin, os dois únicos valores literais do tipo `Boolean` são `true` e `false`.
2. **Operadores Booleanos:** Kotlin suporta os operadores lógicos padrão para trabalhar com valores booleanos:
 - `&&` (E lógico): Retorna `true` se ambos os operandos são verdadeiros.
 - `||` (OU lógico): Retorna `true` se pelo menos um dos operandos é verdadeiro.
 - `!` (NÃO lógico): Inverte o valor do operando (de `true` para `false` e vice-versa).

Arrays

- ⊕ Em Kotlin, um array é uma coleção de elementos de tamanho fixo com o mesmo tipo;
- ⊕ São representados pela classe `Array`, que tem métodos e propriedades úteis;
- ⊕ Podem ser criados pela função built-in `arrayOf()`.

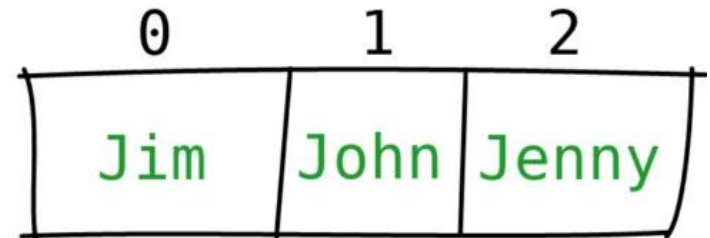
```
arrayOf("Jim", "John", "Jenny")
```

```
names[0]      Array<String>
```

```
names.size
```



names



Arrays

Kotlin_02_7.kt

```
1 fun main() {
2     // Criando um array de inteiros
3     val numeros = arrayOf(1, 2, 3, 4, 5)
4
5     // Acessando elementos do array
6     println("Primeiro número: ${numeros[0]}") // Acessa o primeiro elemento
7     println("Segundo número: ${numeros[1]}") // Acessa o segundo elemento
8
9     // Modificando um elemento
10    numeros[2] = 10 // Modifica o terceiro elemento para 10
11
12    // Imprimindo todos os elementos do array
13    println("Todos os números:")
14    for (numero in numeros) {
15        println(numero)
16    }
17
18    // Tamanho do array
19    println("Tamanho do array: ${numeros.size}")
20
21    // Realizando operações com os elementos do array
22    val soma = numeros.sum() // Soma todos os elementos
23    println("Soma dos números: $soma")
24 }
```


Strings

Em Kotlin, `String` é um tipo de dado que representa uma sequência de caracteres. As strings em Kotlin são imutáveis, o que significa que uma vez criada, a sequência de caracteres que ela representa não pode ser alterada.

Como Declarar Strings

Strings Literais:

- **Strings regulares:** São declaradas com aspas duplas. Exemplo: `val simples = "Isto é uma String"`

Operações com Strings

1. Concatenação:

- Strings podem ser concatenadas usando o operador `+`. Exemplo: `val fullName = firstName + " " + lastName`

2. Interpolação de Strings:

- Kotlin suporta interpolação de string com a sintaxe `$variavel` ou `{expressao}`. Exemplo: `val saudacao = "Olá, $nome!"`

3. Acesso a Caracteres:

- Acessar um caractere específico usando o índice: `val char = nome[0]`

4. Iterar sobre Caracteres:

- Você pode iterar sobre cada caractere de uma string com um loop `for`:

```
for (char in nome) {  
    println(char)  
}
```

Operações com Strings

5. Métodos Úteis:

- Kotlin fornece muitos métodos úteis para strings, como ``length`` (para o comprimento da string), ``toUpperCase()``, ``toLowerCase()``, ``substring()``, ``startsWith()``, ``endsWith()``, ``replace()``, etc.

6. Comparação:

- Strings podem ser comparadas usando ``==`` para comparação de conteúdo e ``===`` para comparação de referência. Exemplo: ``if (str1 == str2)``

Em Kotlin, as strings são tratadas como arrays de caracteres, portanto, você pode fazer muitas das operações que faria em um array, como acessar elementos individuais pelo índice e iterar sobre eles. A imutabilidade das strings em Kotlin ajuda a manter o código seguro e eficiente, pois evita modificações acidentais dos dados.

Operações com Strings

Em Kotlin, as strings são geralmente criadas usando literais de string, mas você também pode criar strings por meio de construtores. Este método é menos comum, mas pode ser útil em situações específicas, especialmente quando você está trabalhando com arrays de caracteres ou precisa inicializar uma string de uma maneira não padrão.

Criando Strings a partir de Arrays de Caracteres

Você pode criar uma string a partir de um array de caracteres usando o construtor `String`. Aqui está um exemplo:

```
fun main() {  
    val charArray = charArrayOf('K', 'o', 't', 'l', 'i', 'n')  
    val stringFromArray = String(charArray)  
    println(stringFromArray) // Saída: Kotlin  
}
```

Controle de Fluxo – if

Em Kotlin, o comando `if` é usado para controle de fluxo e pode ser utilizado tanto como uma instrução condicional quanto como uma expressão.

Como uma Instrução Condicional

Como em outras linguagens de programação, `if` em Kotlin testa uma condição e executa um bloco de código se essa condição for verdadeira.

kotlin

```
if (condicao) {  
    // Bloco de código que é executado se condicao é verdadeira  
}
```

Controle de Fluxo – if

Você também pode adicionar um bloco `else` para tratar o caso em que a condição é falsa.

kotlin

```
if (condicao) {  
    // Bloco de código que é executado se condicao é verdadeira  
} else {  
    // Bloco de código que é executado se condicao é falsa  
}
```

if como expressão

Como uma Expressão

Uma característica única de Kotlin é que ``if`` pode ser usado como uma expressão que retorna um valor. Isso significa que você pode usar ``if-else`` para atribuir diretamente o resultado a uma variável ou retorná-lo de uma função.

kotlin

```
val resultado = if (condicao) {  
    "A condição foi cumprida"  
} else {  
    "A condição não foi cumprida"  
}
```

if encadeado

Você pode encadear várias instruções `if` para verificar várias condições.

```
if (condicao1) {  
    // Código para condicao1 verdadeira  
} else if (condicao2) {  
    // Código para condicao2 verdadeira  
} else {  
    // Código se todas as condições anteriores forem falsas  
}
```


if – pontos importantes

- A condição dentro de um `if` deve ser uma expressão booleana (`true` ou `false`).
- Como expressão, `if` deve sempre ter um ramo `else` para garantir que um valor seja retornado em todos os casos.

Usar `if` como uma expressão é uma prática comum em Kotlin e ajuda a escrever um código mais conciso e expressivo.

Comando when

- ❖ Kotlin não tem o comando **switch**, mas tem o construto **when**.

Em Kotlin, o comando `when` é uma forma avançada e flexível de controle de fluxo, similar ao `switch` em outras linguagens como Java ou C#, mas com capacidades mais poderosas. `When` avalia uma expressão e executa o bloco de código correspondente ao primeiro caso que corresponde ao valor da expressão.

Estrutura Básica

A estrutura básica de um `when` é a seguinte:

kotlin

Copy code

```
when (expressao) {  
    valor1 -> {  
        // Bloco de código para valor1  
    }  
    valor2 -> {  
        // Bloco de código para valor2  
    }  
    else -> {  
        // Bloco de código se nenhum dos casos anteriores corresponder  
    }  
}
```

Comando when

1. Usando `when` com Valores Constantes:

kotlin

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

Comando when

2. Vários Valores em um Único Caso:

kotlin

```
val x = 2
when (x) {
    0, 1 -> println("x é 0 ou 1")
    else -> println("x é maior que 1")
}
```

Comando when

3. Usando Intervalos em `when`:

kotlin

```
val x = 5
when (x) {
    in 1..4 -> println("x está entre 1 e 4")
    in 5..10 -> println("x está entre 5 e 10")
    else -> println("x está fora do intervalo 1-10")
}
```

Comando when

4. `when` como Expressão:

kotlin

```
val x = 2
val str = when (x) {
    1 -> "x é 1"
    2 -> "x é 2"
    else -> "x é outro número"
}
println(str)
```

Comando when

5. `when` Sem Argumento:

kotlin

```
val x = 10
when {
    x % 2 == 0 -> println("x é par")
    x % 2 != 0 -> println("x é ímpar")
    else -> println("x é um número estranho")
}
```

Comando when

6. `when` com Tipos:


kotlin

```
val x: Any = "Kotlin"
when (x) {
    is Int -> println("x é um Inteiro")
    is String -> println("x é uma String")
    else -> println("x é de outro tipo")
}
```


Comando while

A estrutura `while` executa o bloco de código enquanto a condição especificada é verdadeira. A condição é verificada antes da execução do bloco de código em cada iteração.


kotlin

 Copy code

```
while (condicao) {  
    // Bloco de código a ser repetido  
}
```

Exemplo:

kotlin

 Copy code


```
var contador = 0  
while (contador < 5) {  
    println("Contador é $contador")  
    contador++  
}
```

Neste exemplo, o bloco de código dentro do `while` será executado enquanto `contador` for menor que 5.

Comando do while

O comando `do-while` é semelhante ao `while`, mas a diferença principal é que no `do-while` o bloco de código é executado pelo menos uma vez, pois a condição é verificada após a execução do bloco.


kotlin

 Copy code

```
do {  
    // Bloco de código a ser repetido  
} while (condicao)
```

Comando do while

kotlin

 Copy code

```
var contador = 0
do {
    println("Contador é $contador")
    contador++
} while (contador < 5)
```


Neste exemplo, o bloco de código dentro do `do` será executado primeiro e, em seguida, a condição `contador < 5` será verificada. O bloco será repetido enquanto a condição for verdadeira.

Comando for

Em Kotlin, o comando `for` é usado para iterar sobre uma coleção ou intervalo de elementos. Ele é frequentemente utilizado para percorrer arrays, listas, conjuntos, intervalos e outras estruturas que implementam a interface `Iterable`.

Estrutura Básica do Comando `for`

kotlin

 Copy code


```
for (item in colecao) {  
    // Bloco de código a ser executado para cada elemento  
}
```

Aqui, `item` é uma variável que representa o elemento atual na coleção ou intervalo `colecao` a cada iteração do loop.

Comando for

1. Iterando sobre um Array ou Lista:

kotlin


 Copy code

```
val numeros = arrayOf(1, 2, 3, 4, 5)
for (numero in numeros) {
    println(numero)
}
```

Este exemplo percorre um array de inteiros, imprimindo cada número.

2. Iterando sobre um Intervalo:

kotlin

 Copy code


```
for (i in 1..5) {
    println(i)
}
```

Este exemplo itera sobre um intervalo de 1 a 5.

Comando for

3. Iterando em Ordem Decrescente:

kotlin


 Copy code

```
for (i in 5 downTo 1) {  
    println(i)  
}
```

Itera de 5 a 1 em ordem decrescente.

4. Iterando com Passos:

kotlin

 Copy code

```
for (i in 1..10 step 2) {  
    println(i)  
}
```

Itera de 1 a 10 com um passo de 2 (ou seja, imprime números ímpares).

Comando for

- O comando `for` em Kotlin é baseado no uso do operador `iterator`, o que significa que ele pode ser usado com qualquer objeto que tenha um método `iterator()` retornando um `Iterator`.
- Dentro do loop `for`, você pode usar `break` para sair do loop e `continue` para pular para a próxima iteração do loop.

O `for` em Kotlin é uma ferramenta poderosa e flexível para iterar sobre coleções e intervalos, com uma sintaxe limpa e capacidades expressivas.

Exception Handling

Em Kotlin, o tratamento de exceções (Exception Handling) é um mecanismo para lidar com erros de tempo de execução de uma maneira controlada. Ele permite que você defina blocos de código que são executados quando ocorrem erros (exceções) e permite que você recupere o fluxo normal do programa. O tratamento de exceções em Kotlin é muito similar ao de Java, dado que ambos usam o modelo de exceções verificadas e não verificadas.

Estrutura Básica do Tratamento de Exceções


O tratamento de exceções em Kotlin é feito principalmente por meio dos comandos ``try``, ``catch`` e ``finally``.

Exception Handling

1. Bloco `try`:

- O bloco `try` contém o código que pode gerar uma exceção. Se uma exceção ocorrer dentro deste bloco, ela é jogada, e o fluxo de execução é transferido para o bloco `catch` correspondente.

kotlin

 Copy code


```
try {  
    // Código que pode gerar uma exceção  
}
```

Exception Handling

2. Bloco `catch`:

- O bloco `catch` é usado para capturar a exceção lançada pelo bloco `try`. É aqui que você pode lidar com a exceção, seja registrando um erro, tomando ações corretivas, etc.

kotlin

 Copy code


```
catch (e: SomeException) {  
    // Código para lidar com a exceção  
}
```

Exception Handling

3. Bloco `finally` (opcional):

- O bloco `finally` é opcional e é executado independentemente de uma exceção ter sido lançada ou capturada. É geralmente usado para limpar recursos, como fechar arquivos ou conexões de banco de dados.

kotlin

 Copy code

```
finally {  
    // Código que sempre é executado após os blocos try/catch  
}
```

Exception Handling

Exemplo de Tratamento de Exceções

kotlin

```
try {  
    val resultado = someFunctionThatMightThrowException()  
    println(resultado)  
} catch (e: SomeSpecificException) {  
    println("Ocorreu uma exceção específica: ${e.message}")  
} catch (e: Exception) {  
    println("Ocorreu uma exceção genérica: ${e.message}")  
} finally {  
    println("Este bloco é sempre executado.")  
}
```

Exception Handling

Pontos Importantes

- Em Kotlin, ao contrário do Java, não há exceções verificadas. Isso significa que você não é obrigado a capturar ou declarar exceções.
- A eficácia do tratamento de exceções depende da precisão com que as exceções são capturadas e tratadas. Capturar `Exception` genérico pode não ser a melhor prática, pois pode ocultar erros inesperados e tornar o diagnóstico mais difícil.
- O uso adequado do tratamento de exceções pode ajudar a tornar seu programa mais robusto e confiável ao lidar com situações de erro.

Exception Verificadas (Checked)

- **Definição:** São exceções que são verificadas em tempo de compilação. O compilador exige que essas exceções sejam tratadas (capturadas) ou declaradas no método.
- **Propósito:** Elas são usadas para lidar com erros recuperáveis e situações onde você espera que o programa possa continuar a operar de forma significativa após a exceção. Por exemplo, um erro de leitura de arquivo ou problemas de conexão de rede.
- **Exemplos em Java:**
 - `IOException`: Exceções relacionadas a operações de I/O (como leitura/escrita de arquivos).
 - `SQLException`: Exceções relacionadas a operações de banco de dados.

Exception Não Verificadas (Unchecked)

- **Definição:** São exceções que não são verificadas em tempo de compilação, o que significa que o compilador não exige que sejam tratadas ou declaradas.
- **Propósito:** Geralmente são erros de programação que não são esperados para serem recuperados, como divisão por zero, acesso a um objeto nulo, ou índices de array fora dos limites.
- **Exemplos em Java:**
 - `NullPointerException`: Ocorre quando um programa tenta usar uma referência que tem o valor nulo.
 - `ArrayIndexOutOfBoundsException`: Ocorre quando um programa tenta acessar um índice de array que está fora de seus limites.
 - `ArithmeticException`: Por exemplo, uma divisão por zero.

Kotlin e Exceções

É importante notar que, em Kotlin, a distinção entre exceções verificadas e não verificadas não é aplicada da mesma maneira que em Java. Kotlin não tem exceções verificadas. Todas as exceções em Kotlin são não verificadas, o que significa que o compilador não exige que sejam declaradas ou tratadas explicitamente. Isso torna o código Kotlin mais conciso e reduz a necessidade de tratamento de exceções que muitas vezes tornam o código Java mais verboso.

Conclusão

As exceções verificadas promovem uma programação robusta ao forçar o tratamento de certos tipos de erros, enquanto as exceções não verificadas permitem que os desenvolvedores evitem o tratamento de erros onde não é razoavelmente esperado ou útil fazê-lo. A escolha entre usar ou não exceções verificadas depende da filosofia de design da linguagem de programação.