Incluí um guia completo sobre Kotlin 1.9

Desenvolvimento Android com

Jetpack Compose

Jetpack Compose 1.6

{ PROFESSOR CHOREN }

Prof. Esp. Anderson Coimbra Choren

Sobre o Autor

Olá, me chamo Anderson Coimbra Choren e gostaria de compartilhar com você, caro leitor, um pouco da minha vida profissional, para que possamos nos conhecer melhor.

Sou um profissional com mais de 17 anos de atuação na área de Tecnologia da Informação. Sendo graduado em Análise e Desenvolvimento de Sistema e pós-graduado em Especialização em Formação de Professores na Docência do Ensino Profissional e Superior, ambos pelas Escolas e Faculdades QI do Brasil, atualmente atuando como professor do curso técnico em informática na mesma instituição, lecionando em disciplinas relacionadas a programação e análise de sistemas. Atuante também nas áreas de UX e UI, áreas essas onde possui formações livres.

FACEBOOK.COM/GROUPS/431311894533110
@PROFESSORCHOREN
@PROFESSORCHOREN
PROFESSORCHOREN@GMAIL.COM
ANDERSONCHOREN.COM.BR

Sumário

Capítulo 1 - Introdução à linguagem Kotlin

Figura 1.1 Logo da Linguagem Kotlin

Se você, assim como eu, é um “velho de guarda” nesse universo do desenvolvimento, seja ele mobile, web ou desktop, sabe muito bem que o primeiro passo rumo à excelência na linguagem a qual estamos aprendendo, é conhecer os princípios básicos da linguagem, como a tipagem de variáveis, tipos de dados, paradigmas suportados e etc. E com o Kotlin não seria diferente, já que ele pode ser considerado o filho do Java, ou mesmo seu descendente.

Neste capítulo falaremos sobre a linguagem Koltin, linguagem utilizada no desenvolvimento de aplicações Android

Kotlin é uma linguagem de programação orientada a objetos criada pelo JetBrains, sendo lançada oficialmente no ano de 2010.

Um fator importante sobre Kotlin é o fato de possuir uma tipagem estática e forte, assim como seu ancestral, o Java.

OK, mas o que tudo isso quer dizer afinal? 🤔 Beleza, como diria Jack Estripador “Vamos por partes” 😅

Orientação a Objetos: Acho que esse dispensa apresentações. Mas, em todo caso, vamos às explicações. Orientação a Objetos é um paradigma de programação que parte do princípio que todo nosso programa é dividido em partes chamadas de objetos, esses objetos são as entidades que representam o sistema. Bom, como o foco desta publicação não orientação a objetos, não vou me aprofundar muito nesse conceito. Mas se quiser, você pode consultar um artigo da Wikipédia para se inteirar do assunto. Pois O.O é primordial para seus estudos de Kotlin e Flutter.

Tipagem estática: Basicamente, isso quer dizer que no momento que uma variável é declarada com um tipo de dado, esse tipo não poderá ser modificado. Ou seja, se eu criar uma variável com o tipo de dado inteiro, ou mesmo, não declarar seu tipo e atribuir um valor inteiro a ela, ela será declarada como inteira, e não poderá receber valores de outros tipos.

Existe uma forma de declarar uma variável com tipagem dinâmica, mas isso é assunto para outra hora.

Tipagem forte: Basicamente, estamos falando da forma como a linguagem tratará operações entre tipos de dados diferentes. Por exemplo: Se eu tentar somar um valor inteiro 5 a uma variável que contém um texto contendo o número 5, a linguagem irá informar que não é possível somar um inteiro com uma string, sendo necessário uma conversão de tipo de dado (Cast) para tal operação.

Bom, agora que já conhecemos um pouco mais sobre “o Kotlin”, vamos entender um pouco mais sobre a sintaxe e outros aspectos importantes sobre essa linguagem de programação.

1.1 Uso de comentários no código

Bom, nesse quesito, o dart é muito parecido com praticamente todas as linguagens de programação existentes hoje em dia. Mas, em todo caso, vamos às formas de declarar comentários em nossos códigos.

👨‍💻 Mão no código

```
fun main(){
    // Comentário de unica linha
    /*
    Comentário
    de
    múltiplas
    linhas
    */
    /**
     * Comentário de documentação
     */
}
```

Como podemos perceber no código apresentado acima, podemos fazer uso de três tipos de comentários, usando **//** para comentários de única linha, **/* */** para comentários de múltiplas linhas e **/** */** para comentários de documentação, ou seja, comentários que nós criamos para documentar nossos códigos-fonte.

1.2 O método main

Se você é um desenvolvedor mais experiente, já deve saber do que se trata essa função e poderá pular este tópico. Mas caso esse seja seu primeiro contato com programação, deixe-me explicar o que é essa função e qual sua importância.

O método **main** é o principal método do Kotlin, sendo o método que a aplicação irá chamar no momento de sua inicialização. É nele que declaramos nossos códigos iniciais, e as chamadas de outros métodos de nosso sistema.

👨‍💻 Mão no código

```
fun main(){
    // Aqui fica seu código
}
```

1.3 Nosso 'Hello World' de cada dia

Bem, como dizem os gurus da programação: "Se seu primeiro contato com a linguagem não foi um **Olá mundo**, você está aprendendo errado 😅". Então vamos aprender agora a dar nosso "Olá mundo em Kotlin". O que não é nada difícil, basta chamarmos a função main, dentro dela utilizarmos o comando **println**, passando como parâmetro o valor que desejamos imprimir no console.

👨‍💻 Mão no código

```
fun main() {
    println("Olá mundo em Kotlin")
}
```

🖨️ Resultado

```
Olá mundo em Kotlin
```

1.4 Declaração de variável

A declaração de um variável, talvez seja o segundo passo em qualquer linguagem, logo atrás do nosso comando de impressão. E com o Kotlin não seria diferente.

Ao declararmos uma variável no Kotlin, podemos utilizar duas estratégias:

- A primeira é declarar a variável utilizando a palavra reservada “var”. Dessa forma, a variável receberá o tipo de dado do mesmo tipo do valor informado. Ou seja, se eu criar uma chamada **valor1** e utilizar a palavra reservada “var”, poderei colocar qualquer tipo de valor nela, seja **um número**, **um texto** ou mesmo um **valor booleano**.
- A segunda estratégia seria inferir o tipo de dado da variável no momento da criação da mesma.

👨‍💻 Mão no código

```
fun main(){
    var valor1 = "Um texto qualquer"
    var valor2:String = "Outro texto qualquer"
    println(valor1)
    println(valor2)
}
```

🖨️ Resultado

```
Um texto qualquer

Outro texto qualquer
```

Um aspecto importante sobre o Kotlin é o fato de **não possuir tipos de dados primitivos**. Ou seja, todos os tipos de dados são referências de objetos do pacote **dart.core**.

👨‍💻 Mão no código

```
fun main(){
    var valor1 = 15.10
    var convertido:Int = valor1.toInt()
    println(convertido)
}
```

🖨️ Resultado

```
15
```

1.5 Regras de nomenclatura

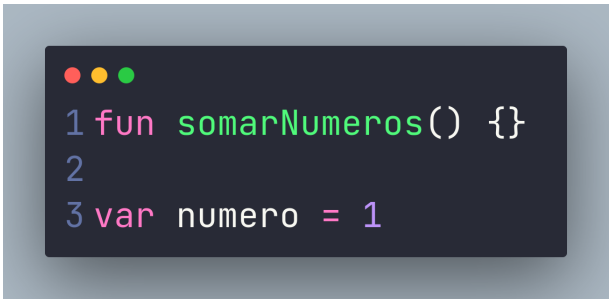
Uma das principais coisas a se conhecer em uma linguagem de programação é a sua sintaxe, porém existe outro aspecto importante sobre qualquer linguagem de programação, as suas regras e boas práticas, também conhecidas pelo nome de “Convenções de código”. Com o Kotlin não seria diferente, o mesmo possui uma extensa conversão de código destinada a mostrar aos desenvolvedores o que fazer e o que não fazer em seus códigos.

Para esse momento focaremos no fato de nomenclatura, ou seja, quais são as boas práticas para nomearmos nossas variáveis, constantes, namespaces, classes e arquivos.

1.5.1 Funções, propriedades e variáveis

Nomes de funções, propriedades e variáveis locais começam com uma letra minúscula e usam maiúsculas e minúsculas e sem sublinhados como podemos ver no exemplo abaixo:

👨‍💻 Mão no código

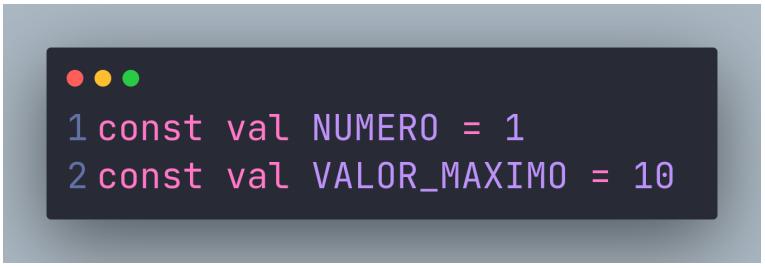


```
fun somarNumeros() {}

var numero = 1
```

1.5.2 Constantes

Nomes em constantes devem ter todas as letras em letras maiúsculas. Já em constantes com nomes compostos devemos separar os nomes por sublinhado maiúsculo (screaming snake case). Vejamos um exemplo da declaração de constantes no exemplo abaixo:



```
const val NUMERO = 1
const val VALOR_MAXIMO = 10
```

1.5.3 Namespaces

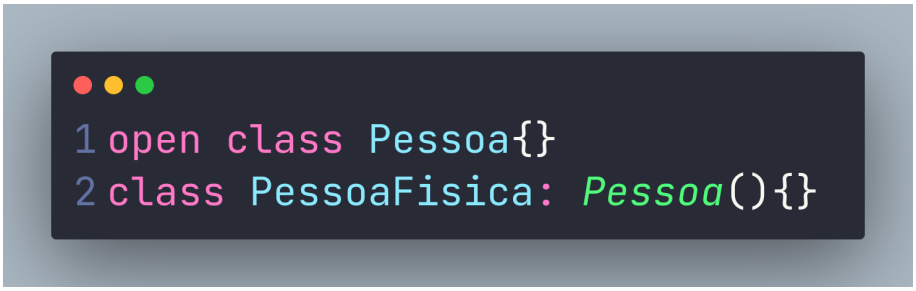
Os nomes dos pacotes são sempre minúsculos e não usam sublinhados (br.com.empresa.projeto). O uso de nomes com várias palavras geralmente é desencorajado, mas se você

precisar usar várias palavras, poderá concatená-las ou usar o caso camel (br.com.empresa.exemploKotlin).

1.5.4 Classes

Nomes de classes e objetos começam com uma letra maiúscula e fazem uso do padrão Camelcase como podemos ver no exemplo abaixo:

👨‍💻 Mão no código



Não se preocupe em entender o conceito apresentado no exemplo acima, compreenderemos melhor no **capítulo 8**, onde trataremos de Orientação a Objetos no Kotlin. Por enquanto se atente ao uso do Camelcase.

1.5.5 Arquivos

Se um arquivo Kotlin contém uma única classe, seu nome deve ser o mesmo que o nome da classe, com a extensão **.kt**. Se um arquivo contiver várias classes ou apenas declarações de nível superior, escolha um nome que descreva o que o arquivo contém e nomeie o arquivo de acordo. Use maiúsculas e minúsculas com uma primeira letra maiúscula, por exemplo, **CarrinhoDeCompras.kt** (Contém as classes Item e CarrinhoDeCompras).

O nome do arquivo deve descrever o que o código no arquivo faz. Portanto, você deve evitar usar palavras sem sentido como “Util” em nomes de arquivos.

1.6 Declaração de constantes

A declaração de constante em Kotlin é tão simples quanto a criação de variáveis. Para declararmos uma constante em nosso código basta adicionarmos a palavra reservada “**val**” antes da declaração da tipagem da variável. Isso dirá ao Kotlin que aquela será uma variável com valor imutável, ou seja, ao qual seu valor não pode ser modificado.

👨‍💻 Mão no código

```
1 // Uso em variáveis globais
2 const val PI = 3.141592653589793
3
4 fun main() {
5     // Uso em variáveis locais
6     val NUMERO_DE_TENTATIVAS = 3
7     println(PI)
8     println(NUMERO_DE_TENTATIVAS)
9 }
10
```

🖨️ Resultado

```
3.141592653589793
3
```

1.7 Tipos de dados

Um aspecto muito importante sobre qualquer linguagem são os tipos de dados suportados. E no Kotlin podemos dizer que estamos vivenciando um sonho, pois o Kotlin possui uma variedade de tipos de dados e como já foi comentado, todos eles são objetos, já que o Kotlin não possui tipos primitivos, muitos herdados de seu ancestral, o Java.

Dentre os principais tipo de dados que o Kotlin possui, podemos ressaltar os seguintes:

- **String:** Representa uma sequência de **caracteres alfanuméricos**, podendo conter letras, números e caracteres especiais (!,@,#,$,%,&, e etc.);
- **Double:** Representa um **valor com fração de até 64 bits**;
- **Float:** Representa um **valor com fração de até 32 bits**;
- **Boolean:** Representa um valor booleano **true** ou **false**;
- **Byte:** Representa um **valor inteiro, com sinal,** de até 8 bits;
- **Short:** Representa um **valor inteiro, com sinal,** de até 16 bits;
- **Int:** Representa um **valor inteiro**, **com sinal,** de até 32 bits;
- **Long:** Representa um **valor inteiro, com sinal,** de até 64 bits;
- **UByte:** Representa um **valor inteiro, sem sinal,** de até 8 bits;
- **UShort:** Representa um **valor inteiro, sem sinal,** de até 16 bits;

- **UInt:** Representa um **valor inteiro**, **sem sinal,** de até 32 bits;
- **ULong:** Representa um **valor inteiro, sem sinal,** de até 64 bits;
- **List:** Representa uma coleção de objetos indexados por um valor inteiro;
- **Set:** Representa uma coleção de objetos, os quais não se repetem na coleção, sendo eles indexados por um valor inteiro;
- **Map:** Representa uma coleção de objetos indexados por um conjunto de chave valor/ valor;
- **Object: É a classe base para todos os objetos Kotlin.** Como Object é a raiz da hierarquia de classes Kotlin, todas as outras classes Kotlin são subclasses de Object;
- **Any:** Como o próprio nome já sugere, representa um valor dinâmico. Ou seja, um tipo de dado que pode ser de qualquer tipo e pode receber objetos de qualquer tipo. Enfim, **dinâmico**.

Existem outros objetos no core do Kotlin, neste tópico tratamos dos principais e mais utilizados em nosso cotidiano de desenvolvimento. Caso deseje conhecer mais sobre os demais tipos de dados do Kotlin, consulte a documentação oficial da linguagem.

1.8 Operadores matemáticos

O Kotlin, assim como praticamente qualquer linguagem de programação existente, possui uma série de operadores matemáticos, indo desde as **4 operações básicas** da matemática até operadores complexos como **resto da divisão** e **pré-incremento** e **pós-incremento**.

Na tabela 1.1 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.1 - Operadores aritméticos no Kotlin

Operador	Significado
+	Adição
–	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão, também conhecido como módulo da divisão

👨‍💻 Mão no código

```kotlin
fun main(){
    val primeiro_numero = 12
    val segundo_numero = 10
    val soma = primeiro_numero + segundo_numero
    val subtracao = primeiro_numero - segundo_numero
    val multiplicacao = primeiro_numero * segundo_numero
    val divisao = primeiro_numero / segundo_numero
    val modulo = primeiro_numero % 2
    println("A soma é $soma")
    println("A subtração é $subtracao")
    println("A multiplicação é $multiplicacao")
    println("A divisão é $divisao")
    println("O resto da divisão é $modulo")
}
```

🖨️ Resultado

```
A soma é 22
A subtração é 2
A multiplicação é 120
A divisão é 1
O módulo é 2
```

1.9 Operadores relacionais

Outro aspecto muito comum e primordial em qualquer linguagem de programação é a capacidade de relacionar e comparar valores. E com o Kotlin não é diferente, pois com ele é possível comparar valores de forma simples e extremamente prática.

Na tabela 1.2 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.2 - Operadores relacionais no Kotlin

Operador	Significado	Objetivo prático
==	Igual	Verificar se o valor à esquerda do operador é igual ao que está à direita do mesmo.
!=	Não igual ou diferente	Verificar se o valor à esquerda do operador é diferente do que está à direita do mesmo.
>	Maior que	Verificar se o valor à esquerda do operador é maior que está à direita do mesmo.
<	Menor que	Verificar se o valor à esquerda do operador é menor que está à direita do mesmo.
>=	Maior ou igual à	Verificar se o valor à esquerda do operador é maior ou igual que está à direita do mesmo.

<=	Menor ou igual à	Verificar se o valor à esquerda do operador é menor ou igual que está à direita do mesmo.

👨‍💻 Mão no código

```
fun main(){
    val primeiro_numero = 12
    val segundo_numero = 10
    // Igualdade
    println(primeiro_numero == segundo_numero)
    // Diferença
println(primeiro_numero != segundo_numero)
    // Maior que
    println(primeiro_numero > segundo_numero)
    // Menor que
    println(primeiro_numero < segundo_numero)
    // Maior ou igual
    println(primeiro_numero >= segundo_numero)
    // Menor ou igual
    println(primeiro_numero <= segundo_numero)
}
```

🖨️ Resultado

```
false
true
true
false
true
false
```

1.10 Operadores Lógicos

Outro grupo de operadores muito importante para qualquer linguagem de programação são os operadores lógicos. Esses operadores são capazes de comparar conjuntos de operadores relacionais, resultando em um valor booleano.

Na tabela 1.3 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.3 - Operadores lógicos no Kotlin

Operador	Significado
!*expressão*	**Inverte** o resultado lógico (Verdadeiro se torna falso e falso se torna verdadeiro)
||	Retorna verdadeiro caso **ao menos uma das comparações tiverem um valor verdadeiro**. Caso contrário retorna **falso**
&&	Retorna verdadeiro caso **todas as comparações tiverem um valor verdadeiro**. Caso contrário retorna **falso**

👨‍💻 Mão no código

```
1 fun main() {
2     val primeiro_numero = 12
3     val segundo_numero = 10
4     val terceiro_numero = 5
5     // O primeiro é maior que os demais
6     println(primeiro_numero > segundo_numero && primeiro_numero > terceiro_numero)
7     // O segundo é maior o primeiro OU menor que o terceiro
8     println(segundo_numero > primeiro_numero || segundo_numero < terceiro_numero)
9     // O primeiro NÃO é maior que os demais
10    println(primeiro_numero > segundo_numero && primeiro_numero > terceiro_numero)
11}
12
```

🖨️ Resultado

```
true
false
true
```

1.11 Operadores de teste

Os operadores de teste são muito úteis quando necessitamos realizar verificações e associações em tempo de execução.

Esses operadores também podem ser utilizados para realizar uma conversão explícita de dados, o que pode ser muito útil em determinados casos e operações do cotidiano de nossas aplicações.

Na tabela 1.4 podemos ver esse operadores, assim como uma breve explicação sobre seu significado na linguagem.

Tabela 1.4 - Operadores de teste no Kotlin

Operador	Significado
`is`	Retorna ***True*** caso um objeto seja de um determinado tipo
`!is`	Retorna ***False*** caso um objeto seja de um determinado tipo

👨‍💻 Mão no código

```kotlin
fun main() {
    val primeiro_valor:Any = "Kotlin"
    val segundo_valor: Any = 10
    println(primeiro_valor is String)
    println(segundo_valor !is String)
}

```

🖨️ Resultado

```
true
true
```

1.12 Estruturas de decisão

Muitas são as vezes em que necessitamos comparar valores e objetos em nossos sistemas. E como vimos, o Kotlin está mais que preparado para tal. Mas até o momento só foi possível comparar tais valores, não não podemos tomar decisões com base nesses resultados. É aí que entram as estruturas de decisão. Também conhecidas como estruturas condicionais ou estruturas de fluxo.

Uma estrutura de decisão, como o próprio nome já sugere, nos permite tomar uma decisão com base em um resultado booleano, seja ele vindo de uma comparação ou mesmo do valor de uma variável ou constante do sistema.

1.12.1 Estrutura IF

A estrutura **IF** é a mais comum e usual estrutura de decisão, e está presente em basicamente 100% das linguagens de programação. Conhecer seu funcionamento é fundamental para qualquer programador, seja ele um iniciante ou experiente no universo do desenvolvimento.

Uma estrutura IF é composta por duas palavras chave, são eles: **if** e **else**. O uso destas duas palavras chave, assim como suas "sub combinações", nos permitem decidir como nosso sistema irá reagir a uma determinada situação. Por exemplo, com uso desta estrutura podemos decidir se um determinado bloco de comandos ou cálculo será executado ou não.

A criação de uma estrutura IF é bastante simples, podemos fazer uso da palavra chave if, onde passamos a condição para sua execução, e caso essa condição seja verdadeira, o bloco de comandos referentes a essa estrutura será executado.

É importante ressaltar que apesar da estrutura condicional IF normalmente estar associada a estrutura else, seu uso não é obrigatório.

No exemplo abaixo podemos ver um exemplo básico do uso de uma estrutura condicional IF.

👨‍💻 Mão no código

```
fun main() {
    val primeiro_valor = 12
    val segundo_valor = 10
    val e_maior = primeiro_valor > segundo_valor
    if(e_maior) {
        println("O primeiro valor é maior")
    } else {
        println("O segundo valor é maior")
    }
}

```

🖨️ Resultado

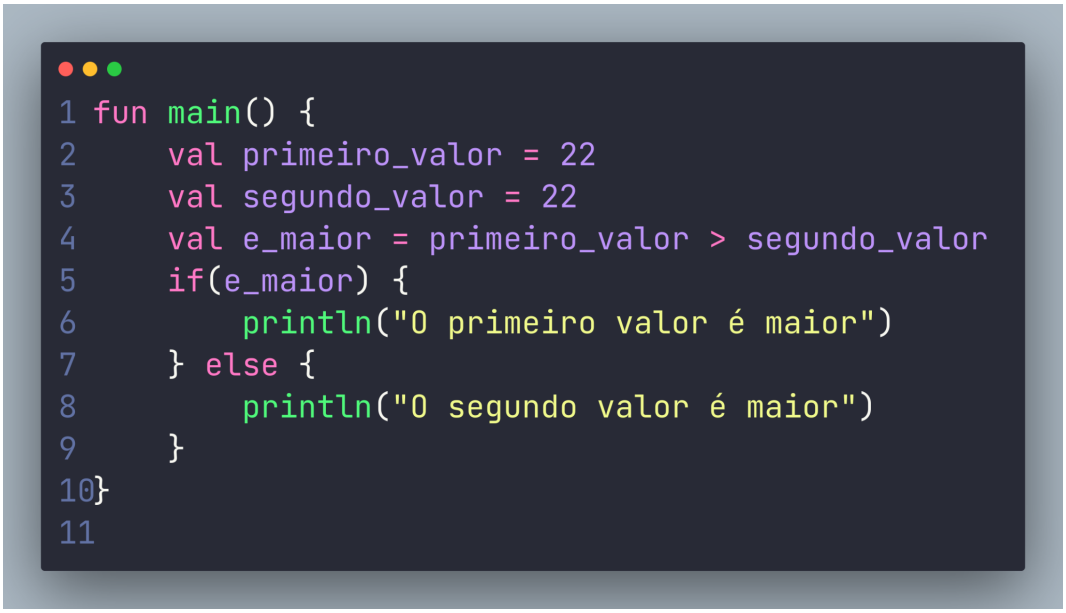
```
O primeiro valor é maior
```

Outro aspecto importante a ressaltar é o fato da estrutura “IF” só ser executada se o valor informado como condição seja **True**. Caso contrário, a estrutura “Else” será executada.

Outro aspecto que vale ressaltar é que podemos combinar as duas palavras chaves que compoẽm a estrutura IF para formar mais cenários possíveis.

Pense comigo, e se os valores forem iguais, o que aconteceria? 🤔. Vejamos o que aconteceria neste determinado cenário.

👨‍💻 Mão no código



```
fun main() {
    val primeiro_valor = 22
    val segundo_valor = 22
    val e_maior = primeiro_valor > segundo_valor
    if(e_maior) {
        println("O primeiro valor é maior")
    } else {
        println("O segundo valor é maior")
    }
}

```

🖨️ Resultado

```
O segundo valor é maior
```

Logo temos um pequeno problema, pois o segundo valor não é maior que o primeiro, pois ambos são iguais. Mas o que podemos fazer para resolver isso? 🤔

E a resposta é bem simples, vamos verificar a possibilidade de ambos serem iguais. Para isso vamos adicionar mais uma condição à estrutura, essa condição será a palavra chave ”else if”.

👨‍💻 Mão no código

```
fun main() {
    val primeiro_valor = 22
    val segundo_valor = 22
    val e_maior = primeiro_valor > segundo_valor
    val sao_iguais = primeiro_valor == segundo_valor
    if(e_maior) {
        println("O primeiro valor é maior")
    } else if(sao_iguais) {
        println("Os valores são iguais")
    } else {
        println("O segundo valor é maior")
    }
}

```

🖨️ Resultado

```
Os valores são iguais
```

Outro aspecto importante a ressaltar sobre a estrutura IF, é o fato de somente um dos blocos ser executado. Pois quando a estrutura encontra um resultado verdadeiro, o bloco de comandos referente a parte da estrutura é executado e a execução da estrutura como um todo é finalizada.

1.12.2 When

Essa estrutura de decisão é muito parecida com a estrutura IF, se destacando pelo fato de possuir uma estrutura voltada para validação de diversos cenários distintos. Podemos dizer que a estrutura When **permite uma bateria de testes de forma organizada e simplificada**, uma ótima alternativa para momentos em que temos diversos cenários para validar.

O When faz uso de uma estrutura de casos, onde cada caso representa um cenário possível. Sendo que existe também um caso padrão para a possibilidade do valor testado não se enquadre em nenhum dos dados previstos na estrutura do When.

Outro aspecto importante sobre essa estrutura é o fato de não possuir um conjunto de delimitadores para seus cenários, ou seja, não utilizamos chaves “{}” para delimitar o encerramento de um cenário, assim como faríamos no IF.

Também é válido ressaltar que o bloco de cenário padrão (else) não é obrigatório, mas recomendado para informar que nenhum cenário previsto é válido para o valor informado.

No primeiro exemplo encontramos uma validação de strings representando uma estrutura de menu, onde o usuário poderá escolher entre uma gama limitada de opções previamente definidas, onde cada opção representa uma operação dentro do sistema.

Vejamos um exemplo do uso dessa estrutura. Nesse exemplo é simulada a escolha do usuário dentro uma gama limitada de opções. Logo que a escolha é feita, a estrutura When entra em ação para verificar a operação escolhida pelo usuário.

👨‍💻 Mão no código

```kotlin
fun main() {
    // Opção escolhida pelo usuário
    val opcao = 2
    when (opcao) {
        1 → println("Escolheu a opção cadastrar")
        2 → println("Escolheu a opção editar")
        3 → println("Escolheu a opção deletar")
        4 → println("Escolheu a opção listar")
        else → println("Opção inválida")
    }
}
```

🖨️ Resultado

```
Escolheu editar
```

1.13 Expressões em Kotlin

Um aspecto importante sobre o Kotlin é o fato de condicionais como o **IF/Else** e o **When** serem expressões. Ou seja, podemos armazenar o resultado de uma dessas estruturas em uma variável. Ficou confuso? 😕 Não se preocupe, vou explicar de forma simples e prática 😁.

Imagine que você gostaria de verificar o valor de uma variável e atribuir um valor a uma determinada variável com base no resultado de tal operação, no Kotlin podemos fazer uma atribuição direta da condicional. Vejamos um exemplo prático:

👨‍💻 Mão no código

```
fun main() {
    val primeiroValor = 10
    val segundoValor = 20

    val resultado =
        if (primeiroValor > segundoValor)
        println("O primeiro valor é maior")
        else println("O segundo valor é maior")
    println(resultado)
}
```

🖨️ Resultado

```
O segundo valor é maior
```

O mesmo pode ser feito utilizando a expressão When, vejamos um exemplo prático, para tal mudaremos nosso exemplo anterior para receber o resultado da verificação realizada.

👨‍💻 Mão no código

```
fun main() {
    val opcao = 2

    val resultado = when(opcao){
        1 -> "Escolheu a opção cadastrar"
        2 -> "Escolheu a opção editar"
        3 -> "Escolheu a opção deletar"
        4 -> "Escolheu a opção listar"
        else -> "Opção inválida"
    }
    println(resultado)
}
```

🖨️ Resultado

Escolheu a opção editar

1.14 Estruturas de repetição

Neste tópico aprenderemos a utilizar as estruturas de loop. Estruturas estas que nos permitem realizar uma mesma operação uma determinada quantidade de vezes, a qual é controlada por uma variável de controle booleana.

Esse tipo de estrutura é muito utilizada para a manipulação de listas e mapas, devido à estrutura em que eles operam. Porém esse não é seu único uso, apesar de ser o mais comum.

Outro aspecto importante a se ressaltar sobre as estruturas de loop é justamente o fator que as torna o que que são, os loops (também conhecidos com repetições ou voltas). Uma estrutura de loop possui uma condição de parada, e até que essa condição seja atendida, ela não cessa sua execução, o que pode levar a uma execução infinita, também conhecido como loop eterno.

1.14.1 Estrutura de loop While

O **While** é uma estrutura de loop similar ao IF. Da mesma forma, possui uma condição para executar um bloco de comandos. Porém se distingue dele pelo fato de executar o bloco de comandos até que a condição imposta torna-se falsa. Até esse momento chegar, o bloco de comandos previsto é executado.

👨‍💻 Mão no código

```kotlin
fun main() {
    var contador = 1
    while(contador < 10) {
        println("Volta n° $contador")
        contador++
    }
}
```

🖨️ Resultado

```
Volta n° 1
Volta n° 2
Volta n° 3
Volta n° 4
Volta n° 5
Volta n° 6
Volta n° 7
Volta n° 8
Volta n° 9
```

Talvez você tenha notado algo incomum em nosso comando “println” usual. Foi introduzido um símbolo de cifrão ($). Isso informa ao Kotlin que desejamos combinar uma String literal ao valor de uma variável ou constante. Muito útil quando desejamos imprimir texto juntamente com referências de variáveis e constantes.

1.14.2 Estrutura de loop Do while

A estrutura de loop **Do While** é praticamente igual a estrutura do **While**, a grande diferença se encontra no modo como a função valida a condição de parada do loop. Enquanto o **While** valida a condição antes de iniciar o loop, o **Do While** o faz ao final da

volta. Isso possibilita a execução do loop ao menos uma vez, dando a oportunidade de inicializar o controlador do loop após o início do mesmo. É importante ressaltar que esse tipo de implementação pode ocasionar problemas na execução do loop, como por exemplo torná-lo eterno, ou seja, onde a condição de parada nunca é alcançada.

👨‍💻 Mão no código

```
fun main() {
    var contador = 1
    do {
        println("Contador: $contador")
        contador++
    } while (contador ⩽ 5)
}

```

🖨️ Resultado

```
Volta nº 1
Volta nº 2
Volta nº 3
Volta nº 4
Volta nº 5
```

Você deve estar se perguntando “OK, mas afinal quando devo usar o While e quando utilizar o Do While? 🤔”. E a resposta é bem simples: Depende 😌.

Eu vou explicar melhor, calma 😅. Caso você tenha uma condição fixa para a execução do loop, ou seja, um ponto de parada calculável, poderá utilizar ambos. Agora se a condição de parada venha a mudar durante a execução do loop, então utilize o Do While, assim poderá verificar se deverá continuar sempre

que a volta anterior tiver terminado. Outro cenário em que o Do While é necessário é quando a condição de parada deve ser definida durante o loop e não antes dele, devido a diversos fatores, como por exemplo, ele for obtido por uma entrada do usuário ou retorno de uma função.

1.14.3 Estrutura de loop For

O **For** é uma estrutura de controle que estabelece um loop de repetição baseado em um contador numérico, diferenciando-se dos demais pelo fato de não utilizar uma condição booleana externa, já que a estrutura da sua sintaxe prove tudo que essa função necessita ser executada, impedindo de ações externas ao loop afetem a execução do mesmo.

Outro fator importante a se levar em consideração é que essa estrutura se baseia em uma contagem e não simplesmente em um valor booleano. Isso quer dizer que a repetição será executada um número limitado de vezes, devido a sua estrutura interna de contagem (incremento ou decremento).

👨‍💻 Mão no código

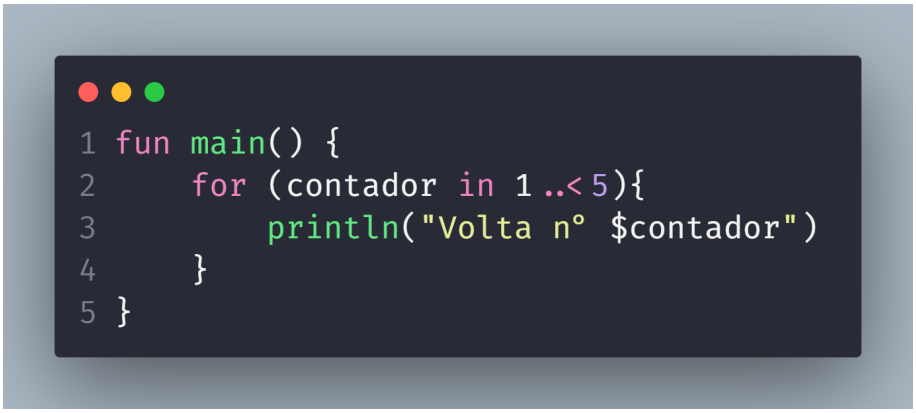
```
fun main() {
    for (contador in 1..5) {
        println("Volta n° $contador")
    }
}
```

🖨️ Resultado

```
Volta nº 1
Volta nº 2
Volta nº 3
Volta nº 4
Volta nº 5
```

Você deve ter notado que o valor **5** também foi impresso, pois a contagem vai do número 1 até o número 5. “Mas e se eu precisar imprimir todos números partindo do primeiro, mas excluindo o último? 🤔. Bom é bastante simples, podemos utilizar o operador **..<**. Vejamos um exemplo prático:

👨‍💻 Mão no código



```
fun main() {
    for (contador in 1..<5){
        println("Volta nº $contador")
    }
}
```

🖨️ Resultado

```
Volta nº 1
Volta nº 2
Volta nº 3
Volta nº 4
```

1.14.3.1 Estrutura For com coleções

A estrutura de loop **for** também pode ser utilizada para gerenciar as voltas do loop baseada nas posições de uma coleção (Listas, sets e mapas). Isso faz dele uma estrutura de loop perfeita também para interações em coleções.

👨‍💻 Mão no código

```
fun main() {
    val numerosPrimos = listOf<Int>(2,3,5,7,11)// ...
    for (numero in numerosPrimos) {
       println(numero)
    }
}
```

🖨️ Resultado

```
2
3
5
7
11
```

Capítulo 2 - Introdução ao uso de funções

Outro aspecto muito importante em uma linguagem de programação é a declaração de funções, e no Kotlin não seria diferente, por esse motivo destinamos um capítulo dedicado ao seu uso, para que possamos nos aprofundar nos mais diversos conceitos que elas possuem em Kotlin.

No Kotlin todas as funções são iniciadas com a palavra reservada “fun”, como podemos ver quando utilizamos a função **main** nos capítulos anteriores.

2.1 Declaração de funções

Uma função do Kotlin é formada por três partes fundamentais, sendo elas:

- Nome
- Parâmetros
- Tipo de retorno

Na figura 2.1 podemos ver um diagrama apresentado as partes que compõem uma função no Kotlin.

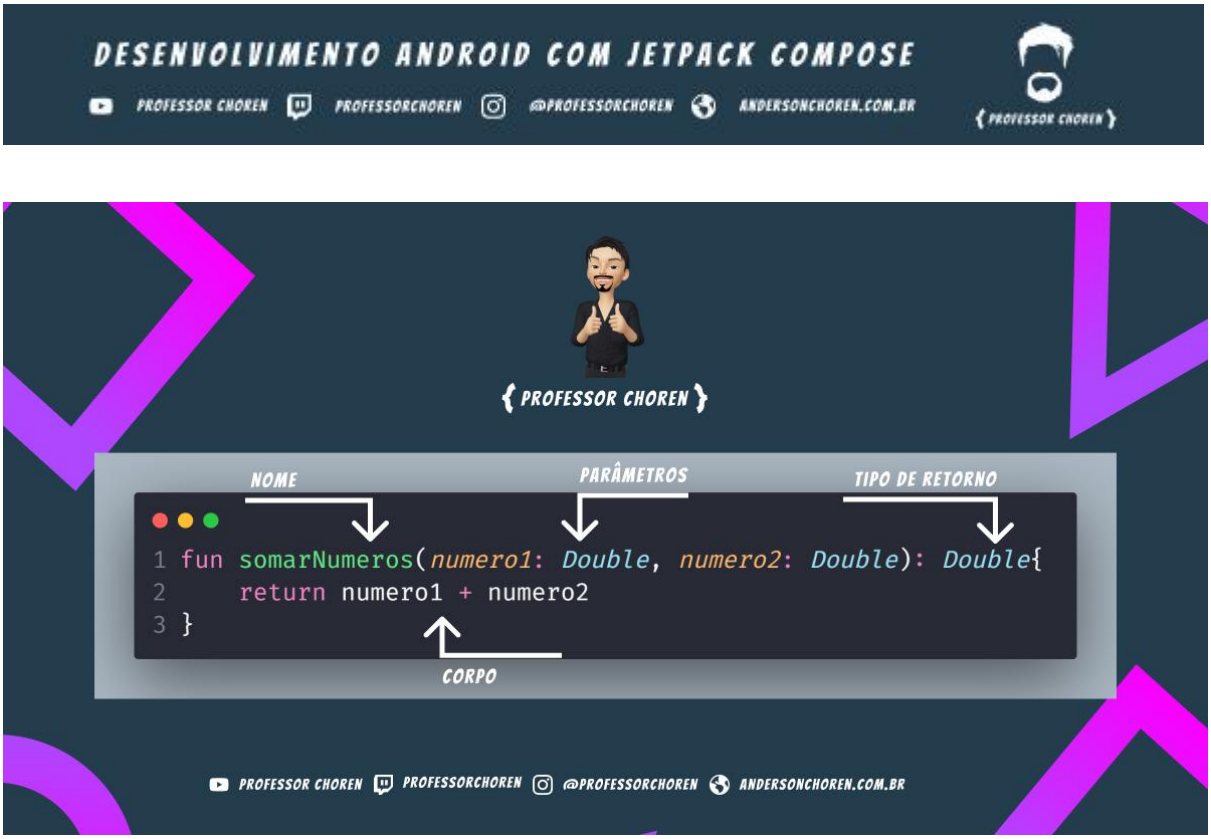


Figura 2.1 - Partes de uma função

É importante ressaltar que uma função pode ou não possuir parâmetros, tudo dependerá da finalidade da função e de sua necessidade de informações externas a seu escopo.

2.2 Declaração de parâmetros

Como vimos anteriormente, uma função pode ou não possuir parâmetros. Isso dependerá da razão de sua existência e da necessidade de informações externas ao seu escopo. Porém existem alguns aspectos importantes a serem falados sobre o uso de parâmetros.

2.2.1 Parâmetros opcionais

No exemplo apresentado na figura 2.1 vimos a declaração de uma função para a soma de dois números, a qual recebe como parâmetro dois valores numéricos, porém ambos são obrigatórios, ou seja, ambos devem obrigatoriamente ser informados, caso contrário sua execução acarretará um erro em

tempo de compilação. Mas e se necessitarmos declarar parâmetros opcionais, ou seja, que podem ou não ser utilizados.

Para esse exemplo, vamos recriar nossa função de soma, agora para que ela permita um terceiro número, mas este sendo opcional.

👨‍💻 Mão no código

```
fun somarNumeros(numero1: Double, numero2: Double,numero3:Double?): Double {
    if(numero3 != null){
        return numero1 + numero2 + numero3
    }
    return numero1 + numero2
}

```

Como podemos ver no exemplo, utilizamos um símbolo de interrogação (?) para declarar que o terceiro parâmetro é opcional. No corpo da função incluímos uma condicional IF para verificar se devemos utilizar o terceiro parâmetro ou não. Dessa forma evitando um erro em tempo de execução devido ao fato do parâmetro estar nulo.

2.2.2 Parâmetros nomeados

Outro aspecto interessante sobre o Kotlin, é a possibilidade de nomear os parâmetros de uma função. Dessa forma podemos informar nossos parâmetros na ordem em que desejarmos, ou até mesmo não informá-los, de forma a criar o mesmo efeito do parâmetro opcional. Mas antes de entendermos como podemos fazer isso, é importante entender como o Kotlin interpreta os parâmetros de uma função.

O Kotlin, por padrão, faz uso da declaração posicional de parâmetros. Ou seja, ao chamarmos um método, devemos passar seus parâmetros na mesma ordem em que eles foram declarados na criação da mesma, pois caso contrário, o Kotlin irá colocar os valores nos locais errados e isso acarretará em um erro na execução da função. Vejamos um exemplo.

👨‍💻 Mão no código

```kotlin
fun main() {
    // No ordem correta
    println(imprimirSaudacao("Anderson", "M"))
    // Com parâmetros invertidos
    println(imprimirSaudacao("F", "Michele"))
}

fun imprimirSaudacao(nome:String,genero:String): String {
    if(genero == "M"){
        return "Bem-vindo Sr. ${nome}!"
    }
    return "Bem-vinda Sra. ${nome}!"
}
```

🖨️ Resultado

```
Bem-vindo Sr.Anderson
Bem-vindo Sra.F
```

Como pode perceber, ao inverter a ordem dos parâmetros houve um pequeno erro na impressão dos dados. Apesar de não haver um erro de compilação devido a ambos os parâmetros serem Strings, a ordem pode afetar a execução correta da função.

É aí que entram os parâmetros nomeados. Com eles podemos dar nomes a cada um dos parâmetros da função, dessa forma podemos informá-los na ordem em que desejarmos, pois o Kotlin irá

associar o nome do parâmetro e não sua posição declarativa. Vejamos como podemos mudar nossa função para aceitar parâmetros nomeados.

👨‍💻 Mão no código

```
fun main() {
    println(imprimirSaudacao(nome="Anderson", genero="M"))
    println(imprimirSaudacao(genero="F", nome="Michele"))
}

fun imprimirSaudacao(nome:String,genero:String): String {
    if(genero == "M"){
        return "Bem-vindo Sr. ${nome}!"
    }
    return "Bem-vinda Sra. ${nome}!"
}

```

🖨️ Resultado

```
Bem-vindo Sr. Anderson
Bem-vinda Sra. Michele
```

Perceba que desta vez obtivemos o resultado esperado, pois estamos nomeando nossos parâmetros ao invés de utilizarmos de forma posicional.

Você já deve ter notado que no momento da passagem dos parâmetros para nossa função, chamamos os mesmo utilizando um símbolo de igualdade (=). Isso informa para o Kotlin que nossos parâmetros **nome** e **gênero** são os nomes dos parâmetros aos quais desejamos associar os valores informados. Assim podemos passá-los na ordem em que desejarmos.

É importante ressaltar que é possível utilizar tanto parâmetros nomeados quanto posicionais em uma mesma função. Ficando assim de sua preferência como chamá-los.

2.3 Lambda expression

Outro aspecto importante a ser ressaltado no Kotlin é o fato do suporte à *"Lambda expression"*, também conhecida como *"Arrow function"* ou função anônima. Para maiores detalhes, temos um ótimo artigo na Wikipédia sobre esse assunto.

Mas a critério de resumo, esse tipo de declaração nos permite criar funções com uma sintaxe mais simples e limpa, o que deixa nosso código menor e consequentemente mais fácil de entender e de dar manutenção.

Para nosso exemplo, vamos recriar nosso exemplo anterior, desta vez utilizando arrow function e operador ternário. Dessa forma poderemos reduzir as linhas de nossa função, tornando-a mais simples e fácil de compreender.

👨‍💻 Mão no código

```
fun main() {
    val saudacao = { nome: String, genero: String →
        if (genero == "M") {
            "Bem-vindo Sr. ${nome}!"
        } else {
            "Bem-vinda Sra. ${nome}!"
        }
    }
    println(saudacao("Anderson", "M"))
    println(saudacao("Michele", "F"))

}

```

🖨️ Resultado

```
Bem-vindo Sr. Anderson
Bem-vinda Sra. Michele
```

2.4 Funções de escopo

A biblioteca padrão Kotlin contém várias funções cujo único propósito é executar um bloco de código dentro do contexto de um objeto. Quando você chama essa função em um objeto com uma expressão lambda fornecida, ela forma um escopo temporário. Nesse escopo, você pode acessar o objeto sem seu nome. Essas funções são chamadas de funções de escopo. Existem cinco deles: deixe, execute, com, aplique e também. Basicamente, essas funções fazem o mesmo: executam um bloco de código em um objeto. A diferença é como esse objeto fica disponível dentro do bloco e qual é o resultado de toda a expressão.

2.4.1 Função de escopo let

A diretiva **let** pode ser usada para invocar uma ou mais funções em resultados de cadeias de chamadas. Vejamos um exemplo abaixo:

👨‍💻 Mão no código

```kotlin
fun main() {
    val saudacao = { nome: String, genero: String →
        if (genero == "M") {
            "Bem-vindo Sr. ${nome}!"
        } else {
            "Bem-vinda Sra. ${nome}!"
        }
    }
    saudacao("Anderson", "M").let { texto →
        println(texto.uppercase())
    }
}

```

🖨️ Resultado

```
BEM-VINDO SR. ANDERSON!
```

2.4.2 Função de escopo apply

A diretiva **apply** é usada para blocos de código **que não retornam um valor** e operam principalmente nos membros do objeto receptor. O caso comum para apply é a configuração do objeto. Essas chamadas podem ser lidas como “aplicar as seguintes atribuições ao objeto”. Vejamos um exemplo:

👨‍💻 Mão no código

```
fun main() {
    val pessoa = Pessoa()
    pessoa.apply {
        nome = "Anderson"
        idade = 30
    }
    println(pessoa.nome)
}
```

🖨️ Resultado

```
Anderson
```

O Kotlin é repleto de diretivas de outras funções de escopo, tratamos aqui das duas principais, porém existem outras que podem ter seu valor em determinados casos. Sendo assim, vale a pena estudar as demais diretamente pela documentação oficial do Kotlin.

Capítulo 3 - Manipulação de Strings

Como vimos no **capítulo 1**, uma string é um conjunto de valores alfanuméricos, onde podemos manter tanto letras como números.

Neste capítulo nos aprofundaremos mais nesse tipo de dado, abordando exemplos de uso, assim como seus métodos nativos presentes no core do Kotlin para a manipulação deste tipo de dado.

3.1 Método isEmpty

O método **isEmpty**, como o próprio nome já sugere, retorna um valor booleano informando se **o objeto está em vazio ou não.**

👨‍💻 Mão no código

```
fun main() {
    val nomeDoUsuario = ""
    println(nomeDoUsuario.isEmpty())
}
```

🖨️ Resultado

```
true
```

3.2 Método isNotEmpty

O método **isEmpty**, como o próprio nome já sugere, retorna um valor booleano informando se **o objeto não está em vazio.**

👨‍💻 Mão no código

```
fun main() {
    val nomeDoUsuario = "Anderson"
    println(nomeDoUsuario.isNotEmpty())
}
```

🖨️ Resultado

```
false
```

3.3 Método Count

O método **count**, como o próprio nome já sugere, retorna o **número de caracteres** que uma string possui.

👨‍💻 Mão no código

```
fun main() {
    val nomeDoUsuario = "Anderson Choren"
    println("A string possui ${nomeDoUsuario.count()} caracteres")
}
```

🖨️ Resultado

```
A string possui 15 caracteres
```

3.4 Método Contains

O método **Contains**, como o próprio nome já sugere,tem como objetivo descobrir se uma string possui ou não uma outra

string como parte de seu conteúdo. Caso encontre a string ele retornará **true**, caso contrário retornará **false**.

✍️ Parâmetros

Parâmetro	Descrição
string	String a qual deseja localizar

👨‍💻 Mão no código

```
fun main() {
    val conteudo = "Essa é uma string qualquer"
    println(conteudo.contains("string"))
}
```

🖨️ Resultado

```
true
```

3.5 Método startsWith

O método **startsWith**, como o próprio nome já sugere,tem como objetivo descobrir se uma string **começa** ou não uma outra string. Caso encontre a string ele retornará **true**, caso contrário retornará **false**.

✍️ Parâmetros

Parâmetro	Descrição
string	String a qual deseja localizar no início da string alvo
[ignoreCase]	Define se desejamos ou não desconsiderar as letras maiúsculas ou não. O valor padrão é

	false

👨‍💻 Mão no código

```
fun main() {
    val conteudo = "Google é o motor de busca mais usado no mundo"
    println(conteudo.startsWith("google"))
    println(conteudo.startsWith("google",true))
}
```

🖨️ Resultado

```
false
true
```

3.6 Método endsWith

O método **endsWith**, é muito parecido com o método startsWith, se diferenciando pelo fato de descobrir se uma string **termina** ou não uma outra string. Caso encontre a string ele retornará **true**, caso contrário retornará **false**.

✍️ Parâmetros

Parâmetro	Descrição
string	String a qual deseja localizar no final da string alvo
[ignorar case]	Define se desejamos ou não desconsiderar as letras maiúsculas ou não. O valor padrão é false

👨‍💻 Mão no código

```
1 fun main() {
2     val conteudo = "Google é o motor de busca mais usado no mundo"
3     println(conteudo.endsWith("google"))
4 }
```

🖨️ Resultado `false`

3.7 Método substring

O método **substring** retorna parte do conteúdo de acordo com as configurações de **ponto de início** até o **ponto de término.**

✍️ Parâmetros

Parâmetro	Descrição
ínicio	Caractere inicial a ser lido
término	Caractere final a ser lido

👨‍💻 Mão no código

```
1 fun main() {
2     val conteudo = "Google é o motor de busca mais usado no mundo"
3     val prefixo = "O Bing é um"
4     val sufixo = conteudo.substring(11,26)
5     println("$prefixo $sufixo")
6 }
```

🖨️ Resultado

```
O Bing é um motor de buscas
```

3.8 Método split

O método split transforma um string em uma lista, separando os elementos por meio de um separador, ao qual determinamos.

✍️ Parâmetros

Parâmetro	Descrição
separador	Caractere separador que desejamos utilizar

👨‍💻 Mão no código

```
fun main() {
    val personagens = "Homem de ferro, Capitão América, Thor, Hulk"
    val lista_de_personagens = personagens.split(",")
    println(lista_de_personagens)
}
```

🖨️ Resultado

```
[Homem de ferro,  Capitão América,  Thor,  Hulk]
```

3.9 Método uppercase

O método **toUpperCase** transforma todas o conteúdo (números, letras e caracteres especiais) de uma string para maiúsculo.

👨‍💻 Mão no código

```
fun main() {
    val nomeDoUsuario = "Anderson Choren"
    println(nomeDoUsuario.uppercase())
}
```

🖨️ Resultado

```
ANDERSON CHOREN
```

3.10 Método lowercase

O método **toLowerCase** transforma todas o conteúdo (números, letras e caracteres especiais) de uma string para minúsculo.

👨‍💻 Mão no código

```
1 fun main() {
2     val nomeDoUsuario = "Anderson Choren"
3     println(nomeDoUsuario.lowercase())
4 }
```

🖨️ Resultado

```
anderson choren
```

3.11 Método trim

O método **trim** retorna uma string com os espaços retirados do início e do final de string.

👨‍💻 Mão no código

```
1 fun main() {
2     val nomeDoUsuario = "    Anderson Choren        "
3     println(nomeDoUsuario.trim())
4 }
```

🖨️ Resultado

```
Anderson Choren
```

É importante ressaltar que o Kotlin possui ainda as funções **trimStart** e **trimEnd**, que removem, respectivamente, apenas os espaços em branco da direita e esquerda de uma string.

3.12 Método Replace

O método **replace**, como o próprio nome já sugere, substitui uma palavra por outra, muito útil quando desejamos substituir um determinado caractere ou palavra.

👨‍💻 Mão no código

```
fun main() {
    val frase = "O rato roeu a roupa do rei de roma"
    val letra = "r"
    val substituta = "p"
    val novaFrase = frase.replace(letra
, substituta)
    print(novaFrase)
}
```

🖨️ Resultado

```
O pato poeu a poupa do pei de poma
```

3.13 Objeto StringBuilder

O objeto **StringBuilder** gera uma string de maneira programática. Um **StringBuilder** não gera um novo objeto String até que o método **toString()** seja chamado.

Para a inserção de uma string única podemos utilizar o método append(), o qual recebe a string que desejamos inserir no

objeto. O qual possui dois parâmetros opcionais que permitem especificar a posição inicial e a posição final da inserção.

👨‍💻 Mão no código

```
fun main() {
    val builder = StringBuilder()
    builder.append("Heróis da Marvel: ")
    builder.append("Homem de ferro, ")
    builder.append("Capítão américa, ")
    builder.append("Thor, ")
    builder.append("Hulk")
    println(builder.toString())
}
```

🖨️ Resultado

```
Heróis da marvel: Homem de ferro, Capitão América, Thor, Hulk
```