ZUP

Kotlin Básico

Kotlin

Kotlin é uma linguagem de programação criada para, como diz a própria documentação, fazer os programadores felizes.

Kotlin foi criada para utilizar a própria máquina virtual do java (JVM - Java Virtual Machine) fazendo dela uma linguagem multiplataforma, o que significa que pode ser utilizada para criar programação independente do sistema operacional ou plataforma destino como smartphones, TVs, android, windows etc.

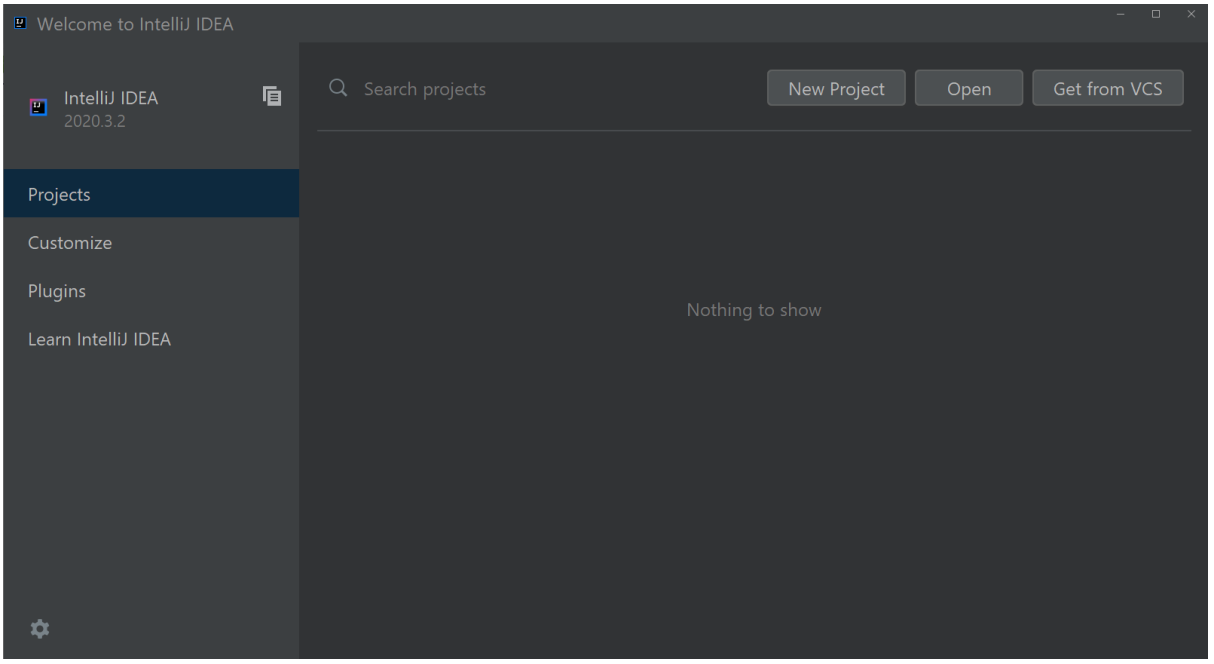
Uma das suas características mais fortes é a possibilidade do programador utilizar o paradigma de orientação a objetos ou paradigma funcional, ao contrário do Java que ficamos limitados à orientação a objetos.

Como Criar um Projeto Kotlin - Intellij

Neste curso vamos utilizar a IDE Intellij para nos auxiliar com o processo de criação de projetos. O Intellij também é capaz de ler nosso código e fazer sugestões inteligentes de coisas que podemos usar ou até mesmo apontar erros que podem passar despercebidos.

Primeiro passo

Ao abrir pela primeira vez a IDE Intellij teremos uma janela inicial de boas vindas. Nessa janela podemos selecionar a opção de criar um novo projeto. Navegue pela tela utilizando a tecla TAB ou clicando diretamente no botão.



Segundo passo

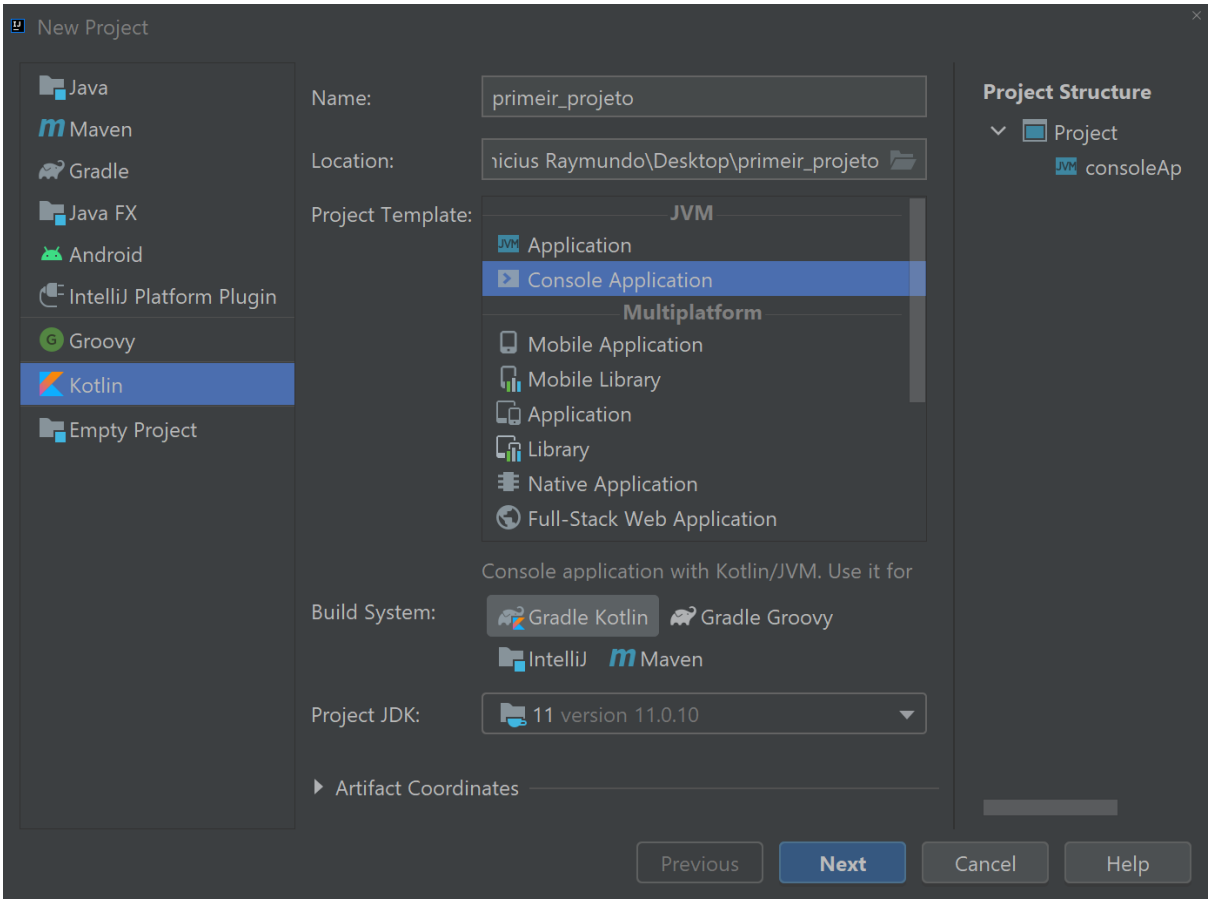
Após clicar no botão New Project uma nova janela será aberta para que possamos selecionar as opções desejadas para o projeto. Vale dizer que o Intellij é uma IDE para programação em diversas linguagens e arquiteturas, por isso temos opções como Java para linguagem que será usada.

No caso do nosso curso usaremos Kotlin. Com a tecla direcional para baixo podemos navegar por essa opções até chegar em Kotlin, ou clicando diretamente nesta opção.

Logo em seguida uma nova aba com opção do lado direito será aberta, podemos chegar até lá usando a tecla TAB do teclado. Essas são as opções para criar a estrutura inicial do nosso projeto.

Entre todas as opções vamos seguir as seguintes opções configuração:

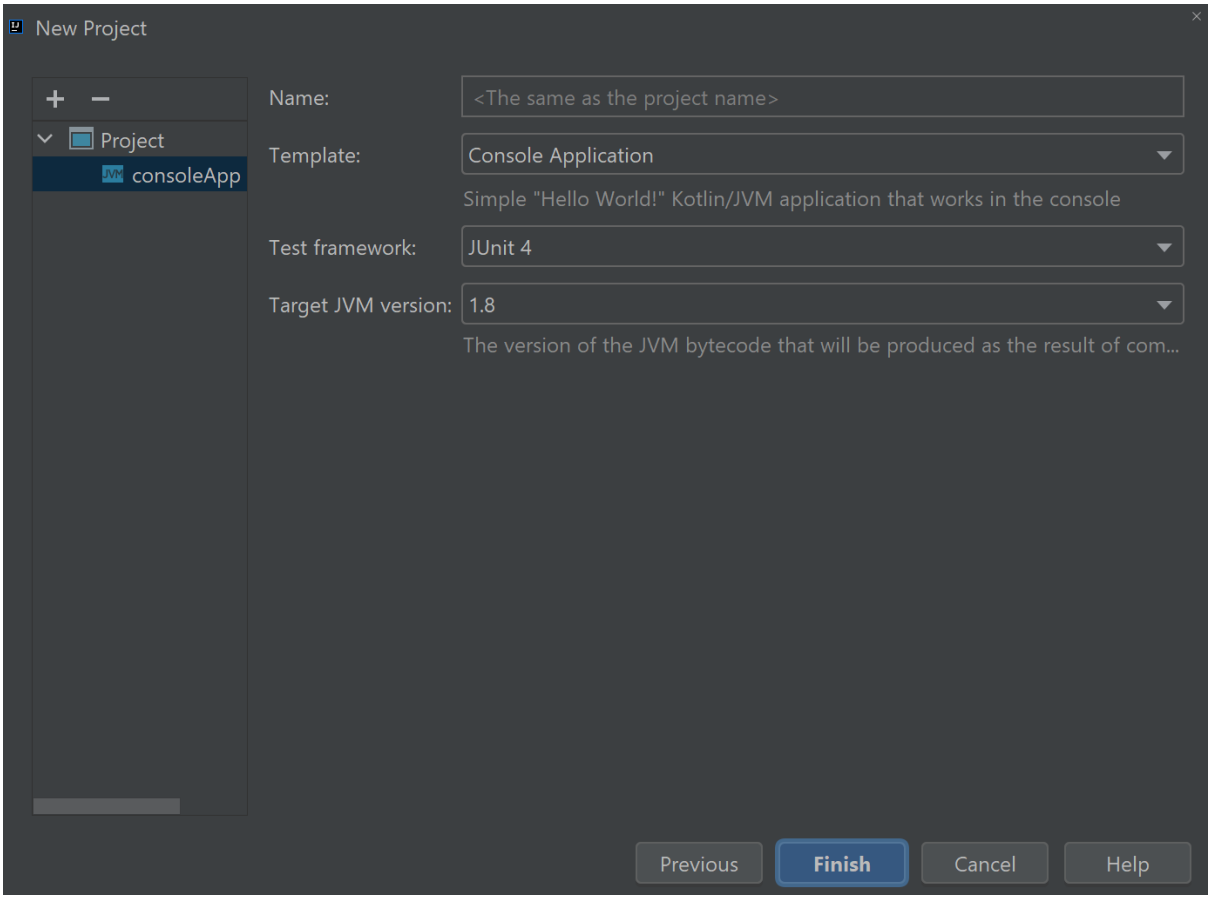
- **Nome**: primeir_projeto
- **Location**: será a pasta onde o Intellij vai criar o projeto. Vamos selecionar o Desktop
- **Project Template**: Console Application
- **Build System**: Gradle (falaremos mais sobre ele no futuro)



Após preencher e selecionar todos os campos vamos navegar com o TAB até o next ou clicar diretamente no botão de cor azul escrito Next para seguir para o próximo passo.

Terceiro passo

Agora para finalizar o processo temos uma nova tela onde podemos confirmar as opções inseridas anteriormente. Caso esteja tudo certo podemos apenas selecionar o botão Finish.



Selecionando finish o projeto será criado e carregado na tela.

Importante: O processo de criação do projeto pode demorar alguns minutos, pois o Intellij vai realizar o download e instalação de ferramentas e bibliotecas que serão necessárias para o projeto. O Gradle também pode realizar o mesmo processo caso seja necessário.

Sintaxe

Kotlin é uma linguagem compilada e assim como toda linguagem compilada o código é executado a partir de um método ou função principal chamada de main (main, do inglês, significa principal).

Em Kotlin todo o código do sistema pode ser executado dentro dessa função. Para exemplificar o código vamos criar o famoso "Hello World” (que, traduzindo do inglês, significa “Olá Mundo!”).

```
fun main() {
    println("Hello World!")
}
```

Chaves

O Kotlin, assim como o Java, utiliza blocos de execução para separar e organizar quais comandos devem ser executados na ordem correta, para demarcar esses blocos de execução devemos usar chaves ({ }), a todo momentos que podemos notar um chave se abrindo podemos considerar que um bloco de execução foi aberto, e sempre que notamos uma chave fechando podemos considerar que o contexto daquele bloco de execução foi encerrado.

Parênteses

Os parênteses no Kotlin servem para que possamos passar argumentos, parâmetros ou recursos para uma função funcionar.

No exemplo abaixo vemos um simples if que vai verificar se 1 é maior que 2, para poder perguntar isso vemos passar como parâmetros o número 1 o operador de comparação de maior (>) e o número 2 tudo isso dentro dos parênteses.

```
if(1 > 2){

}
```

Os parênteses também são usados para dar ordem de prioridade para cálculos da mesma forma que na matemática convencional.

Exemplo:

```
var resultado = 2 + 3 * (3 - 2)
```

No exemplo acima notamos que vamos armazenar em uma variável o resultado do cálculo. Para o Kotlin a prioridade está na subtração presente dentro dos parênteses, então ele primeiro irá executar o cálculo de 3 menos 2 e depois multiplicar por 3 seguindo a ordem de prioridade natural da matemática.

Ponto e Vírgula

Para o Kotlin o uso de ponto e vírgula é totalmente opcional, porém, por boa prática, recomenda-se NÃO utilizar.

Função main

```
fun main() {
    println("olá mundo")
}
```

Notamos no código anterior como podemos escrever um "olá mundo" no console. Podemos usar esse exemplo para analisar a sintaxe da linguagem Kotlin.

Logo no começo encontramos a palavra reservada "*fun*" que é usada para declarar (ou criar) uma função. "*Fun*" é abreviação de *function* que significa função em inglês.

Palavras reservadas são todas as palavras usadas pela linguagem para realizar operações ou declarações, existem muitas outras palavras reservadas e vamos ver cada vez mais ao longo do curso.

Em seguida da palavra *fun* temos o nome da função que no caso precisa ser chamada de *main* para que seja reconhecida como função principal do projeto. Quando estamos chamando ou criando uma função sempre colocamos logo em

seguida os sinais de abre e fecha parênteses pois entre eles podem ficar os parâmetros necessários para o funcionamento da função. Especificamente quando estamos criando uma função nova após os parênteses abrimos e fechamos chaves e entre elas ficarão o código que será executado após a compilação do código.

Em nosso exemplo a função main chama uma outra função chamada *println* que naturalmente existe na biblioteca do Kotlin. Essa função *println* precisa receber um texto (*String*), essa função fará com que o texto apareça no console.

Variáveis

Variáveis são como caixas para armazenar conteúdo do nosso código que pode ou não ser alterado durante a execução do programa. Com elas temos o poder de alterar valores e criar novos valores como resultados de contas ou tratamento de texto. Em Kotlin temos 2 formas de criar variáveis.

VAL

São variáveis imutáveis que uma vez preenchidas com um valor esse valor continuará o mesmo.

```
fun main() {
    val texto = "Ola mundo"
    println(texto)
}
```

No exemplo acima armazenamos o "Olá mundo” em uma *val* chamada texto e usamos essa variável para entregar o texto à função println.

Caso você tente alterar o valor de uma variável declarada como val uma exceção será executada no sistema durante a compilação.

```
Val cannot be reassigned
```

A tradução da mensagem seria “Val não pode ser reescrita”.

VAR

A segunda forma de declarar uma variável é usando a palavra reservada *var.* Seu funcionamento é semelhante ao do *val,* porém variáveis declaradas com *var* podem ser alteradas a qualquer momento.

```
fun main() {
    var texto = "Ola mundo"
    println(texto)
    texto = "Tchau mundo"
    println(texto)
}
```

Tipagem

Toda variável armazena um único tipo de dado, seja esse tipo texto (String) ou números (Int, Float. Double …).

Sempre que declaramos uma variável podemos indicar de maneira explícita ou implícita qual é o tipo de dados que será armazenado nesta variável.

A maneira implícita fazemos tudo de forma direta, criamos a variável e atribuímos um valor, a partir desse momento durante a compilação o kotlin vai atribuir um tipo para essa variável e não será alterado.

Já a explícita, além de criar a variável, antes de atribuir um valor a ela, definimos o seu tipo.

```
fun main() {

    //Declaração implícita
    var texto1 = "Ola mundo"

    //Declaração explícita
    var texto2: String = "Ola mundo"
}
```

Ao escolher trabalhar com a declaração explícita, é importante lembrar que, diferentemente do Java, o Kotlin tem apenas as variáveis do tipo wrapper, ou seja, NÃO temos as variáveis do tipo primitivas.

Os wrappers são tipagens que possuem métodos para auxiliar no desenvolvimento além de armazenar valores NULL.

Tipo	Descrição
Boolean	Apenas valores **true** ou **false**
Byte	Números inteiros de até 1 bytes
Short	Números inteiros de até 2 bytes
Int	Números inteiros de até 4 bytes
Long	Números inteiros de até 8 bytes
Double	Números com vírgula de até 8 bytes
Float	Números com vírgula de até 4 bytes
String	Textos inteiros
Char	Apenas 1 letra

Null Safety

Uma das armadilhas do Java é o acesso a variável com referência nula que resulta na NullPointerException. Se trata do caso em que tentamos fazer acesso a um atributo ou método de um objeto que está nulo.

Para eliminar esse tipo de problema, a linguagem kotlin trabalha com o conceito Null Safety (ou, do inglês, nulo seguro), por padrão as variáveis não são nulas a menos que se especifique que ela pode ser nula utilizando o ? (ponto de interrogação) como indicador dessa possibilidade de ser nulo.

Caso tente indicar uma variável explícita sem o ?, o código não irá nem compilar e o intellij indicará o erro como:

```
fun main() {
    var texto: String = null
    println(texto)
}
```

Null can not be a value of a non-null type String

A tradução da mensagem seria “Nulo não pode ser o valor de um não-nulo tipo String”. Esse erro vale para todos os tipos de variáveis declaradas de forma explícita.

Dica: Rode o trecho de código na sua máquina para ver funcionando.

Para resolver esse erro basta apenas incluir o ? logo após a palavra String.

```
fun main() {
    var texto: String? = null
    println(texto)
}
```

É importante ressaltar que essa validação de nulo ou não-nulo só será feita em variáveis declaradas de maneira explícita, as variáveis implícitas ainda correm o risco de ocorrer o NullPointerException.

```
fun main() {
    var texto = null
    println(texto)
}
```

Por boa prática, o ideal é sempre trabalhar com variáveis declaradas de forma explícita, pois assim teremos maior controle sobre a verificação do nulo e reduzimos a possibilidade da aplicação apresentar erros do tipo NullPointerException.

Dica: Rode o trecho de código na sua máquina para ver funcionando.

Operador de Atribuição

No exemplo anterior nós apenas criamos as variáveis, porém para preencher esse espaço de memória criado com algum valor precisamos usar um operador de atribuição, esse operador é o final de igual da matemática (=).

Exemplo:

```
var numero: Int = 10
var texto: String = ""
texto = "Agora sua variável tem algum valor"
```

Nesse exemplo podemos notar que temos 2 variáveis criadas porém são preenchidas em tempos diferentes.

A variável **"número"** é criada e ao mesmo tempo já usamos o operador de atribuição para preenchê-la com o valor 10.

Já a segunda variável é criada em um primeiro momento com o vazio e apenas na próxima linha de execução preenchemos ela com um valor.

Todas as variáveis em Kotlin devem ser inicializadas, mesmo que seu valor seja vazio (aspas duplas sem nada no meio no caso de String), zero (para qualquer tipo de número) ou *false* (no caso de booleanos).

É importante mencionar que os valores dentro de cada variável pode ser facilmente alterado a qualquer momento durante a programação caso o programador queira, basta utilizar o nome da variável seguido do operador de atribuição e por fim o novo valor.

Estruturas de Tomada de Decisão

As estruturas de tomada de decisão são justamente para preparar essas condições. Em Kotlin usamos os **if**, **else** e **else if** para tomar decisões específicas de acordo com condições pré programadas.

Exemplo:

```
val sexta = 13

if (sexta != 13) {
   println("Dia de sorte")
} else {
   println("Hoje é uma sexta-feira 13. Bruxa está solta")
}
```

No exemplo acima temos uma variável que armazena um número inteiro a condições que preparei dentro do **if** vai analisar o valor armazenado dentro da variável chamada sexta, caso o valor lá dentro seja **diferente de 13** então o programa exibirá **“Dia de sorte”**.

Ainda temos a possibilidade de usar o if como expressão, onde uma variável recebe o valor do if de acordo com a sua condição.

```
val maiorValor = if (2 >1 ) "Maior valor é 2" else "Maior valor é 1"
```

No exemplo acima, temos a variável maiorValor onde fazemos a verificação de qual número é maior se é o 2 ou o 1, se 2 for maior que 1, então a variável receberá o valor “Maior valor é 2”, caso contrário receberá o valor “Maior valor é 1”.

Dica: Rode o trecho de código na sua máquina para ver funcionando.

Operadores de Comparação

Para realizar comparações de valores, usamos operadores capazes de dizer se uma comparação é verdadeira ou falsa, estes ponderadores literalmente vão literalmente comparar dois valores de retornar como resposta um true ou false que podemos armazenar em uma variável ou usar diretamente dentro de um if como vimos no exemplo acima.

Operador	Descrição
==	se são exatamente iguais
===	se são iguais por referência
!=	se são diferentes
<	se um é menor que outro
>	se um é maior que outro
<=	se um é menor ou igual a outro
>=	se um maior ou igual a outro

Portas Lógicas

Além dos operadores de comparação, também podemos utilizar dentro de um if uma porta lógica para realizar mais de uma comparação ao mesmo tempo.

Exemplo:

```
val idade = 18
val ingresso = true

if (ingresso && idade >= 18) {
   println("Pode Entrar")
}
```

No exemplo acima foram criadas 2 variáveis que representam 2 valores diferentes que devem ser verificados dentro de um **IF**. O primeiro valor é a idade e o segundo é um boolean dizendo se a pessoa possui um ingresso, dentro do IF vamos verificar se o valor da variável ingresso é true e se a idade é maior ou igual a 18 anos. Caso uma delas seja falsa o **IF** não deve ser executado, é por isso que usamos o sinal de **E** comercial (&) que corresponde a porta lógica **“and”**.

Com a porta lógica **AND** vamos garantir que o **IF** seja executado apenas quando as duas condições forem verdadeiras.

Existem outras portas lógicas além da **AND**, observe na tabela abaixo.

Operador	Porta	Descrição
&&	and (e)	Apenas verdadeiro quando ambas as condições são verdadeiras
||	or (ou)	Apenas verdadeiro quando pelo menos uma condição for verdadeira
!	not	Inverte o resultado da saída. Exemplo: se o resultado for verdadeiro ele troca para falso. Se o resultado for falso ele troca para verdadeiro

Estruturas de Repetição

Em determinados momentos na programação nós precisamos repetir diversas vezes uma mesma função ou algoritmo, nesses casos precisamos de estruturas de repetição que vão literalmente repetir em um número limitado de vezes aquele algoritmo.

Isso é extremamente vantajoso evitar a repetição desnecessária de código e para criar programação que só param sua execução quando atingem um objetivo específico.

While

Uma das estruturas de repetição é o while que poderia ser traduzido para a palavra “enquanto”. O while funciona a partir de uma condição que for verdadeira, enquanto essa condição continuar verdadeira o código dentro do contexto do while será executado,

Isso é muito importante saber, pois se essa condição nunca mudar para falso o while nunca irá parar criando assim um looping infinito.

Exemplo:

```
var contador: Int = 0;

while (contador <= 10) {
   println(contador);
   contador++;
}
```

No exemplo acima, criamos um while que espera que a condição do valor dentro da variável contador seja menor ou igual a 10, enquanto a condição for verdadeira o código dentro do bloco do while será executado, sendo assim vamos exibir na tela o valor contido dentro da variável contador e vai somar mais 1 ao valor dentro da variável.

Por fim, quando o valor dentro da variável for 11 ele a condição de menor ou igual a 10 vai ser falsa e o while vai para sua execução.

IMPORTANTE: lembre-se que o while vai funcionar apenas enquanto a condição for verdadeira (true), e só para enquanto a condição for falsa (false).

For

Uma outra estrutura de repetição que podemos usar é o **FOR** que podemos traduzir para **"para cada"**.

Embora o for tenha a mesma função do **while** de gerar repetição no código e facilitar nossa vida como programador ele tem uma maneira diferente de trabalhar.

Exemplo:

```
for (contador: Int in 0..10) {
   println(contador)
}
```

No exemplo acima tem um for que realizará a mesma tarefa do exemplo com while visto anteriormente, porém com o for podemos construir as condições para seu funcionamento dentro da sua própria assinatura, isso de certa forma cortamos o caminho e no final das contas temos o mesmo resultado que é contar até 10 e exibir na tela.

For em listas e vetores

O interessante é que com o for também podemos percorrer listas e vetores para facilitar o trabalho na hora de trabalhar com conjunto massivo de dados.

Com o for podemos acessar separadamente cada item dentro de um vetor e trabalhar exclusivamente com esse item e partir para o próximo com facilidade.

Exemplo:

```
val nomes = arrayOf("Vinícius", "José", "Saulo", "Miriana",
"Diandra")

//Leia-se para cada nome em nomes

for (nome: String in nomes) {
   println(nome.uppercase(Locale.getDefault()))
}
```

No exemplo acima usamos um for para percorrer um vetor de nomes, nesse caso o for vai pegar um nome de cada vez dentro da variável **nomes** e colocar dentro da variável **nome** assim podem trabalhar com cada um separadamente. Nesse exemplo nós estamos imprimindo na tela do usuário cada um dos nomes dentro do vetor com todas as letras maiúsculas.

Vetores

Armazenar informações durante a execução do código é sempre importante e para facilitar e centralizar informações contamos com vetores, porém ele são imutáveis e não são dinâmicos.

Para criar um vetor, usamos a função *arrayOf* passando os valores que queremos dentro do vetor.

```
val nomes = arrayOf("Vinícius", "José", "Saulo", "Miriana",
"Diandra")
```

Para apenas alocar memória e ir inserindo os valores dentro do vetor, basta fazer conforme abaixo:

```
var numeros = IntArray(5)
numeros[0] = 1
print(numeros[0])
```

Funções

Conforme mencionado no início desse material, Kotlin é uma linguagem que pode seguir tanto o paradigma orientado a objetos como o paradigma funcional. Por conta disso, ao invés de utilizarmos métodos, utilizamos funções (que é a mesma coisa que os métodos em Java).

Funções em Kotlin têm o mesmo comportamento que os métodos em Java, elas podem receber parâmetros, retornar valores ou serem do tipo void (ou seja, não retornam nada).

Abaixo temos como é as estruturas básicas que uma função pode ter.

Exemplos:

```
fun nomeDaFuncaoSemRetornoESemParametro(){
   //logica
   print("Eu sou uma função")
}

fun nomeDaFuncaoSemRetornoEComParametro(nomeParametro:
tipoParametro){
   //logica
   print(nomeParametro)
}

fun nomeDaFuncaoComRetornoESemParametro():TipoRetorno{
   //logica
   return valorDoRetorno
}

fun nomeDaFuncaoComRetornoEComParametro(nomeParametro:
tipoParametro):TipoRetorno{
   //logica
   return valorDoRetorno
}
```

Exemplos para você rodar na sua máquina:

```
fun nomeDaFuncaoSemRetornoESemParametro(){
   print("Eu sou uma função")
}

fun nomeDaFuncaoSemRetornoEComParametro(nome:String){
   print("Olá! eu me chamo $nome")
}


fun nomeDaFuncaoComRetornoESemParametro():Int{
   var total = 2 + 2
   return total
}


fun nomeDaFuncaoComRetornoEComParametro(x: Int, y: Int): Int {
   var soma = x + y
   return soma
}
```