

# Sintaxe

## I Variáveis

Para declaração de variáveis em Kotlin podemos utilizar 2 palavras chaves: **val** e **var**

Utilizamos **val** para variáveis imutáveis, ou seja, uma vez definido seu valor este não poderá mais mudar. Ex:

```
val nome:String = "João"
```

## ! Variáveis

Utilizamos a palavra **var** para variáveis mutáveis, ou seja, que seu valor poderá mudar ao longo do tempo. Ex:

```
var idade:Int = 15
```

Via de regra, procure sempre utilizar variáveis imutáveis, utilize variáveis mutáveis somente quando há necessidade.

## I Variáveis

Utilizamos a palavra **var** para variáveis mutáveis, ou seja, que seu valor poderá mudar ao longo do tempo. Ex:

```
var idade:Int = 15
```

Via de regra, procure sempre utilizar variáveis imutáveis, utilize variáveis mutáveis somente quando há necessidade.

## ! Variáveis

**Repare que Kotlin é uma linguagem tipada, ou seja, toda variável tem um tipo definido. O tipo da variável é definido em sua declaração, veja:**

```
var idade:Int = 15
```



## I Variáveis

**Porem, informar o tipo da variável em sua declaração é opcional, caso você não informe a variável irá assumir o tipo do valor que ela está recebendo, veja:**

```
var idade = 15
```

# FUNÇÕES

## I Funções em Kotlin

Para declarar uma função utilizamos a palavra chave *fun* seguindo do nome da função, argumentos e retorno, veja exemplo:

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



## I Funções em Kotlin

Para funções com somente 1 linha de execução, podemos fazer de uma declaração simplificada:

```
fun somar(a: Int, b: Int): Int = a + b
```

## I Funções em Kotlin

Para chamar a função:

```
val resultado:Int = somar(4, 5)
```

## I Funções “void”

Algumas funções não possuem retorno, em Java são as funções “void”. Aqui no Kotlin temos o tipo “Unit” que é equivalente ao “void” do Java. Então, podemos fazer uma função sem retorno da seguinte maneira:

```
fun imprimir(): Unit {  
    print("Ola FIAP")  
}
```

## I Funções “void”

No entanto, dificilmente você verá um código em Kotlin escrito assim, isso porque quando uma função não possui retorno é opcional a utilização da palavra “Unit”.

```
fun imprimir(){  
    print("Ola FIAP")  
}
```

# Tipos Nullables

## I Tipos Nullable

Por padrão, uma variável em kotlin não pode receber valores nulos. Para isso, devemos indicar explicitamente que a variável pode receber nulos utilizando o operador **?** a frente do tipo da variável em sua declaração. Ex:


```
var variavelNula:String? = "Teste FIAP"
```

Neste exemplo, estamos declarando uma String com valor, porem estamos dizendo que em algum momento ela pode

## I Tipos Nullables

Quando trabalhamos com tipos nullable precisamos tomar alguns cuidados ao acesso a este tipo, por exemplo para saber o length de uma String nullable não podemos acessar a propriedade diretamente, veja:

variavelNula.length



O compilador nos mostra um erro dizendo que não podemos acessar essa propriedade diretamente. Como faremos então?

---

## I Tipos Nullables

Neste caso, temos opção de acesso seguro utilizando o operador **?** ou forçar acesso a referencia da variável com operador **!!** (Não recomendável)

```
var variavelNula:String? = "Teste FIAP"  
val tamanho = variavelNula?.length
```

Caso a variável seja nula, ele simplesmente ignora a execução do comando não deixando lançar erros de referências nulas.



# Arrays e Listas

## I Arrays

Para criar um Array em Kotlin é muito simples:

```
val arrayNumeros = arrayOf(1, 2, 3, 4, 5)
```

Para acessar o valor de cada posição podemos fazer:

```
val pos0 = arrayNumeros[0]  
val pos1 = arrayNumeros[1]
```

## I Listas

Para criar uma lista em Kotlin é muito simples:

```
val listaNumeros = listOf(1, 2, 3, 4, 5)
```

Para acessar o valor de cada posição podemos fazer:

```
val pos0 = listaNumeros[0]  
val pos1 = listaNumeros[1]
```

## I Listas Mutáveis

Uma lista em Kotlin por padrão é imutável. Isto é, não conseguirmos adicionar ou remover elementos de uma lista depois de sua criação. Para isso, precisamos utilizar uma lista mutável:

```
val listaMutavel = mutableListOf(1, 2, 3, 4)
```

Assim, temos disponível os métodos de add e remove:

```
//Adiciona um item  
listaMutavel.add(5)
```

```
//remove um item em determinada posição  
listaMutavel.removeAt(1)
```

## I Estruturas de repetição

Em Kotlin temos muitas, **MUITAS** formas de fazer uma estrutura de repetição. Vamos as mais comuns:

**FOR:**

```
for(i in 0 until 10){  
    print("$i")  
}
```

## I Estruturas de repetição

FOR em uma lista ou Array:

```
val lista = listOf(0, 1, 2, 3)

for(i in lista){
    print("$i")
}
```

## I Estruturas de repetição

**ForEach:** ForEach é uma função lambda disponível em listas ou Arrays em Kotlin e pode ser usado da seguinte maneira:

```
val lista = listOf(0, 1, 2, 3)
```

```
lista.forEach {  
    print("$it")  
}
```

No contexto do “ForEach” a variável “it” é sempre o elemento em questão.



## I Função Filter

**Filter:** Toda lista ou array também tem disponível a função “filter” que tem pro objetivo filtrar a lista a partir de algum critério. Ex:

```
val lista = listOf(0, 1, 2, 3, 4)

val listaPares = lista.filter {
    it % 2 == 0
}
```

Aqui utilizamos a função filter para filtrar somente os números pares da lista.



## I Função Map

Map: O Map transforma os elementos da lista em outros elementos através de um mapeamento. Ex:

```
val lista = listOf(0, 1, 2, 3, 4)
```

```
val listaQuadrado = lista.map {  
    it * it  
}
```

Aqui utilizamos a função map para elevar ao quadrado todos elementos da lista.

# Classes

A criação de classes em Kotlin é muito simples. Para criar uma classe básica a sintaxe é a seguinte:

```
class ClasseBasica {  
  
  
  
  
  
  
}
```

E para criar uma instancia dessa classe basta:

## I Class

Vamos para um exemplo mais prático, vamos supor que necessitemos de uma classe “Pessoa” com 2 atributos: nome e idade.

```
class Pessoa {  
    var nome:String = ""  
    var idade: String = ""  
}
```

## I Class

E então, poderíamos instanciar um objeto da classe Pessoa e definir valores para suas propriedades.

```
val p = Pessoa()  
p.nome = "Kassiano"  
p.idade = 30
```

## I Class

Essa definição de classe, apesar de funcionar, tem um problema. Precisamos definir suas propriedades como “var” porque precisamos mudar seus valores posteriormente. E por isso perdemos o recurso da imutabilidade.

```
val p = Pessoa()  
p.nome = "Kassiano"  
p.idade = 30
```

## I Construtores

Para resolver esse problema, podemos definir os atributos da classe como “val” e recebemos no construtor, fazendo assim uma única inicialização.

```
class Pessoa(  
    val nome:String,  
    val idade:Int  
)
```

## I Construtores

Desta forma, devemos passar os valores na construção do objeto:

```
val p = Pessoa("Kassiano", 30)
```



## I Construtores

Ainda sobre construtores. A forma completa de uma declaração de classes com construtor é usando a palavra “constructor”. Porém para construtores primários, como no exemplo anterior isso é opcional.

```
class Pessoa constructor(  
    val nome:String,  
    val idade:Int  
)
```

## I Getters e Setters

Perceba que não precisamos criar métodos de get e set para as propriedades, isto porque a linguagem cria os gets e sets internamente e não precisamos deixar explicito. No entanto, as vezes precisamos sobrescrever o get e set. Isso é possível da seguinte maneira:

```
var nome:String = ""  
    get() = field  
    set(value) {  
        field = value  
    }
```

## I Modificadores de acesso

Em kotlin, basicamente temos os mesmos modificadores de acesso do Java:

```
private var nome:String = ""
protected var sobreNome:String = ""
var idade:Int = 0
```

# Herança de classes

## I Herança

Um conceito muito importante em orientação a objetos é a herança de classes. Este recurso também está disponível em Kotlin. Para fazer herança de uma classe basta usar : em frente a declaração da classe e informar a classe Pai.  
Ex.

```
open class Veiculo {  
    fun acelerar(){ }  
    fun frear() {}  
}  
  
class Barco: Veiculo() {  
    fun levantarLeme() { }  
}
```

## I Herança

Detalhes para prestar atenção. A classe pai deve ser marcada como “open” senão a herança não é permitida.



```
open class Veiculo {  
    fun acelerar(){ }  
    fun frear() {}  
}  
  
class Barco: Veiculo() {  
    fun levantarLeme() { }  
}
```

## I Herança: Outras considerações

Quando uma classe pai, recebe parâmetros em seu construtor, esses parâmetros devem ser passados na hora da herança. Ex:

```
open class Colaborador(val nome:String)
```

```
class Funcionario(  
    nome:String,  
    val cargo:String  
): Colaborador(nome)
```

## I Casting

Vamos analisar esse trecho de código:

```
open class Veiculo
class Barco: Veiculo()
class Automovel: Veiculo()

fun obterVeiculoAtual(): Veiculo {
    if( nextBoolean() ){
        return Barco()
    }else{
        return Automovel()
    }
}
```



## I Casting

A função “*obterVeiculoAtual*” retorna um tipo genérico “Veiculo”. Para saber se o veiculo é um Barco ou Automóvel, podemos utilizar o comando “when”:

```
val veiculoAtual = obterVeiculoAtual()
when(veiculoAtual) {
    is Barco -> {
        println("0 veiculo é um barco")
    }
    is Automovel -> {
        println("0 veiculo é um automóvel")
    }
}
```

# Interfaces

## I Interfaces

A ideia de uma interface em orientação a objetos é de criar um contrato em que a classe que implementa a interface é obrigada a seguir.

```
interface OnClickListener {  
    fun onClick()  
}
```

```
class Botao: OnClickListener {  
    override fun onClick() {  
  
    }  
}
```

## I Interfaces

Também é possível atribuir a implementação de uma interface para uma variável, veja:

```
val click = object : OnClickListener{  
    override fun onClick() {  
  
    }  
}
```

**Data class, Enum Class e  
Sealed Class**

## I Data Class

O conceito de data classe é bem simples, uma “data class” é uma classe em que seu principal propósito é armazenar dados. Basta incluir a palavra “data” ao nome da classe:

```
data class User(val name: String, val age: Int)
```

As data class também implementam por padrão as funções:

```
equals()/hashCode()  
toString()  
componentN()  
copy()
```

## I Enum Class

Os Enums em Kotlin são muito parecidos com os do Java:

```
enum class ClasseSocial {  
    BAIXA, MEDIA, ALTA  
}
```

```
val classeBaixa = ClasseSocial.BAIXA
```

## I Objects

Os objects, diferentes de classes são objetos que não possuem instancia. Um objeto estático. (static em Java)

```
object DatabaseProvider{  
  
    fun getDatabase(){  
        //exemplo  
    }  
}  
  
fun main(){  
  
    DatabaseProvider.getDatabase()  
  
}
```



## | Sealed Class

As classes seladas são classes que só podem ser herdadas por classes que estão no mesmo escopo. Veja:

```
sealed class ViewState {  
    class Loading : ViewState()  
    class Success : ViewState()  
    class Error : ViewState()  
}
```

## | Sealed Class

E podemos utilizar o `when` para verificar seu tipo:

```
val state: ViewState = ViewState.Loading()
when(state){
    is ViewState.Loading->{

    }
    is ViewState.Success->{

    }
    is ViewState.Error->{

    }
}
```

# 1. Nullable Types

## | Nullables Types

**Em Kotlin, por padrão, nenhuma variável pode ser nula. No entanto, podemos habilitar que determinada variável pode receber valores nulos. Pra isso, basta acrescentar o operador “?” na frente do tipo da variável. Ex:**

```
var nome:String? = null  
var soma:Int? = null  
var resultado:Double? = null
```

## I Nullables Types

Quando temos uma variável nullable, assumimos o risco de ela ter um valor ou não. Sendo assim, o compilador do Kotlin nos obriga a tomar certos cuidados ao trabalhar com variáveis nullable. Veja o seguinte caso:

```
val idade:Int? = getIdade()


if(idade > 18){ ← Erro de compilação
    println("Maior de Idade")
}else{
    println("Menor de idade")
}
```

## | Checagem com if

A primeira forma de se trabalhar com nullables é fazer a checagem utilizando um IF. Com o IF checamos se a variável é diferente de nula, e caso verdadeiro, temos a garantia de acessar aquele valor.

```
val idade:Int? = getIdade()
```

```
if(idade != null) {  
    if (idade > 18) {  
        println("Maior de Idade")  
    } else {  
        println("Menor de idade")  
    }  
}
```



Compilação OK

## Utilizando a função let

A função let em conjunto com o operador “?” também poderia ser utilizada neste cenário. Veja:

```
val idade:Int? = getIdade()

idade?.let {
    if (idade > 18) {
        println("Maior de Idade")
    } else {
        println("Menor de idade")
    }
}
```

## I Utilizando a função let

No caso da função let, em seu escopo podemos acessar o valor da variável de forma segura através do nome da variável e também através do “it” :

```
val idade:Int? = getIdade()

idade?.let {
    if (it > 18) {
        println("Maior de Idade")
    } else {
        println("Menor de idade")
    }
}
```



## I Var e Nullables

Nos exemplos anteriores estávamos trabalhando com variáveis imutáveis, e isso com certeza facilita as coisas. Quando estamos trabalhando com variáveis mutáveis precisamos tomar ainda outros cuidados. Veja o seguinte exemplo:

```
data class Pessoa(val nome:String, var idade:Int?)

fun main() {

    val pessoa = Pessoa("p1", 0)

    if(pessoa.idade != null){
        pessoa.idade += 10 ← Erro de compilação
    }
}
```

## | Var e Nullables

Isso acontece porque o compilador não pode garantir que um var vai manter seu valor, mesmo protegido por um IF. Temos algumas opções nesses casos:

```
persona.idade?.let {  
    pessoa.idade = it + 10  
}
```

## | Var e Nullables

No entanto, se a propriedade idade estivesse nula, ela nunca receberia valor algum porque o código do let seria ignorado. Nesse caso, um IF seria uma boa saída:

```
if(pessoa.idade == null){  
    pessoa.idade = 10  
}else{  
    pessoa.idade = pessoa.idade?.plus(10)  
}
```

# Operador Elvis

## I Operador Elvis

Ainda no problema anterior, podemos também fazer uso do operador Elvis para resolver. O operador Elvis é como se fosse um if ternário, sua sintaxe é “?:” ele faz uma verificação de nulo e atribui um valor a variável caso ela seja nula. Ex:

```
val saldo:Int? = null  
val novoSaldo = saldo?:0
```

## I Operador Elvis

Se a idade for nula ele considera 0 e então podemos fazer a operação de soma sem problemas.

```
data class Pessoa(val nome:String, var idade:Int?)

fun main() {

    val pessoa = Pessoa("p1", 0)
    pessoa.idade = pessoa.idade?:0 + 10
}
```

## I Operador Elvis + Extensions Functions

Uma ideia legal é combinar o operador Elvis com uma extension Function. Para esse problema, poderíamos criar uma extension de Int? chamada: orZero que irá retornar um Int não nulo

```
fun Int?.orZero():Int {  
    return this?:0  
}  
  
fun main() {  
    val pessoa = Pessoa("p1", 0)  
    pessoa.idade = pessoa.idade.orZero() + 10  
}
```



# Recursos avançados da linguagem Kotlin

- Extensions Function
- Higher Order Function
- First class Function
- Infix Function
- Parâmetros e argumentos



# Extensions Function

## I Extensions Function

Extensions Function, ou Função de extensão, é um recurso que possibilita estender as funcionalidades de uma classe sem alterar seu código base. Ex:

```
fun Int.quadrado(): Int {  
    return this * this  
}
```

## I Extensions Function

Com a extensão criada, todos os Int do projeto passarão a ter a função “quadrado”.

```
val idade = 20
```

```
val idadeQuadrado = idade.quadrado()
```

## Extensions Function

Podemos fazer extension para tudo, por exemplo poderíamos fazer uma extension do algoritmo que soma os algarismos de um número inteiro:

```
fun Int.sumDigits():Int{  
    var n = this  
    var sum = 0  
    while (n > 0){  
        sum += n % 2  
        n /= 2  
    }  
    return sum  
}
```

## I Extensions Function

E a chamada ficaria assim:

```
fun main(){  
    val lista = listOf(12, 34, 25, 9)  
    lista.map {  
        it.sumDigits()  
    }.forEach {  
        println("$it")  
    }  
}
```

# **Funções de alta ordem - Higher Order Functions**

## I Higher Order Functions

Uma função de alta ordem, ou mais conhecida como *Higher Order Function*, é aquela que possui a capacidade de receber outras funções como argumento, retornar funções ou fazer ambas as coisas! A princípio, parece um pouco complicado, mas a prática é muito simples e seu uso pode resolver problemas complexos.

## I Callbacks

O exemplo mais comum do uso desse tipo de função são as funções de *callbacks*. Chamamos de função de callback uma função que será executada logo após o retorno de uma função principal. Por ex:

O software faz uma requisição a uma API Web através da seguinte função:

```
fun executarRequisicao(){  
    //  
}
```



## I Callbacks

E essa requisição retornará alguma resposta, no entanto essa resposta não chegará na mesma *tread*, dessa forma eu não conseguiria fazer algo assim:

```
val resposta = executarRequisicao()
```

## I Callbacks

Não temos controle de quando a resposta chegará, mas eu preciso executar alguma ação quando essa resposta chegar. Por isso, passamos um *callback* pra dentro da função *executarRequisicao* e então, internamente quando chegar a resposta ele executará o *callback*.

## I Recebendo Callbacks por parâmetro

Devemos preparar a função para receber um callback.  
Veja o exemplo:

```
fun executarRequisicao(callback: ()-> Unit){  
    //executar a chamada ...  
    callback.invoke()  
}
```

## I Passando o Callbacks por parâmetro

Podemos criar uma variável que guarda a referência de uma função e passa-la por parâmetro. Esse recurso é chamado de *first class function*, ou seja a linguagem trata funções com o mesmo peso de variáveis.

```
val callback: ()->Unit = {  
    println("função de callback")  
}  
val resposta = executarRequisicao(callback)
```

## I Passando o Callbacks por parâmetro

Uma outra forma é ter uma função realmente e passar sua referência por parâmetro:

```
fun callback(){  
    println("função de callback")  
}  
  
fun main(){  
    val resposta = executarRequisicao(::callback)  
}
```

## I Passando o Callbacks por parâmetro

Uma terceira forma de se passar a função é em forma de DSL. Se uma função recebe como último parâmetro uma outra função, podemos fazer a chamada dessa forma:

```
fun main() {  
    val resposta = executarRequisicao() {  
        println("função de callback")  
    }  
}
```

# Infix Fucntion

## I Infix Fucntion

Infix é a capacidade de invocar uma função de uma classe uma função de extensão sem a necessidade de utilizar o “.” Por exemplo, imagine que criemos uma função de extensão para multiplicar números inteiros:

```
fun Int.multiplicar(n:Int): Int {  
    return this * n  
}  
  
fun main(){  
    val resultado = 20.multiplicar(4)  
    println("$resultado")  
    //irá printar: 80  
}
```



## I Infix Fuction

Infix é a capacidade de invocar uma função de uma classe uma função de extensão sem a necessidade de utilizar o “.” Por exemplo, imagine que criemos uma função de extensão para multiplicar números inteiros:

```
fun Int.multiplicar(n:Int): Int {  
    return this * n  
}  
  
fun main(){  
    val resultado = 20.multiplicar(4)  
    println("$resultado")  
    //irá printar: 80  
}
```

## I Infix Fuction

Poderíamos marcar a função *multiplicar* como *infix* e então sua chamada ficaria assim

```
infix fun Int.multiplicar(n:Int): Int {  
    return this * n  
}
```

```
fun main(){  
    val resultado = 20 multiplicar 4  
    println("$resultado")  
    //irá printar: 80  
}
```

## ■ Notas: Infix Function

- Funções *infix* devem ser obrigatoriamente funções de extensão ou membros de classe.
- Funções *infix* devem ter obrigatoriamente receber um único parâmetro.

# **Parâmetros e argumentos de função - Conceito**

## I Argumentos de funções

Argumentos de funções são valores fornecidos na chamada de uma função. Veja um exemplo:

```
val argumento1 = 10
```

```
val argumento2 = 5
```

```
somar(argumento1, argumento2)
```

## I Parâmetros de funções

Parâmetros são variáveis nomeadas dentro da função que recebem os valores fornecidos pelos argumentos, veja:

```
fun somar(parametro1:Int, parametro2:Int): Int{  
    return parametro1 + parametro2  
}
```

## I Argumentos nomeados

Um recurso interessante da linguagem é a possibilidade de passagem de argumentos nomeados. Veja o exemplo:

```
fun somar(number1:Int, number2:Int): Int{  
    return number1 + number2  
}
```

```
somar(  
    number1 = 10,  
    number2 = 20  
)
```

## I Parâmetros opcionais

Também podemos ter funções que recebem parâmetros opcionais. Neste, caso um parâmetro opcional receberá um valor padrão caso esse valor não seja fornecido:

```
fun somar(number1:Int, number2:Int = 0): Int{  
    return number1 + number2  
}
```

```
somar(number1 = 10)
```



# Funções da StandardLib

## I Função let

Uma função largamente utilizada no kotlin é a função *let*. *let* é uma função de escopo que pode ser aplicada em qualquer tipo do kotlin. O *let* simplesmente abre um escopo fechado para trabalhar com algum valor, ex:

```
val nome = "FIAP"
```

```
nome.let {  
    println("$it")  
    ...  
}
```

## I Função let

Olhando para o exemplo anterior não parece algo muito útil. No entanto, e se a variável “nome” fosse um Nullable, olha como poderíamos acessa-la de forma segura sem fazer um IF

```
val nome:String? = "FIAP"
```

```
nome?.let {  
    println("$it")  
}
```



## I Função apply

Função apply é outra função de escopo e assim como o let está disponível a qualquer tipo do kotlin. A diferença é que usaremos o *apply* para modificar valores de um objeto. Por ex:

```
class Pessoa{  
    var nome:String = ""  
    var sobreNome:String = ""  
}
```

```
val p = Pessoa().apply {  
    this.nome = "Kassiano"  
    this.sobreNome = "Resende"  
}
```

## I Função *also*

A função *also* é muito parecida com a função *let* com a diferença que a função *also* não retorna nenhum valor e a função *let* retorna. Ex:

```
fun somar(number1:Int, number2:Int): Int{  
    return (number1 + number2).also {  
        println("somando valores..")  
    }  
}
```

[https://github.com/macedoraf/FIAP SI 1S 2023](https://github.com/macedoraf/FIAP_SI_1S_2023)