# Inferência automática de tipos;

O Kotlin ele sabe automaticamente qual o tipo da variável, seja String, int, float...

# Como definir uma variável em Kotlin

Var = (mutável) > posso mudar o valor

Val = (imutável) > variável que não posso mudar o valor (constantes)

package Variaveis  
  
fun main() {  
 *println*("Hello Word!!!")  
 var nome = "Alleph";  
 val cpf = 15679025701;  
}

Segue o exemplo, nome podemos mudar, CPF é único.

# Paradigmas

Kotlin trabalha com Orientação a Objetos e Funcional.

# Como iniciar o Kotlin

Assim como o Java vamos iniciar com uma função main

package Variaveis  
  
fun main() {  
 *println*("Hello Word!!!")  
 var nome = "Alleph"  
 val cpf = 15679025701  
}

# Concatenando no Kotlin

fun main() {  
 *println*("Hello Word!!!")  
 var nome = "Alleph"  
 val cpf = 15679025701  
  
 *println*("Seja bem-vindo $nome $cpf")  
}

Saída:

Hello Word!!!

Seja bem-vindo Alleph 15679025701

# Variáveis – Tipos e Instância

fun main() {  
 val nome: String = "Alleph"  
 val cpf: Int = 12345678910  
}

## Tipos das variáveis

Byte, Short, Int, Long, Float, Double, Boolean, Char.

## Constantes

A constante ela deve ser utilizada de duas maneiras, dentro da uma função.

fun main() {  
 val nome: String = "Alleph"  
 val cpf: Long = 12345678910 // Tempo de execução  
}

Fora de uma função

package Variaveis  
  
const val *nomeUsuario*: String = "Alleph Nogueira" // Constante global

// Tempo de compilação  
fun main() {  
 val nome: String = "Alleph"  
 val cpf: Long = 12345678910  
}

Uma variável global, ela está disponível para todas as funções dentro do arquivo.

## Tempo de compilação x Tempo de execução

Tempo de compilação: O computador já tem que saber o valor da variável antes de tudo.

Tempo de execução: O valor pode ser adicionado a uma variável val na hora de compilar.

## Instância com Kotlin

package Variaveis  
  
class Usuario {}  
  
fun main() {  
 val usuario = Usuario()  
 // Lembrar que quando eu faço a instancia com val o usuario não pode ser mudado.  
  
}

# Null Safety e Chamada segura

Imagina que você tem um objeto carro, e aí tudo certo, mas você também pode não conseguir instanciar o objeto e aí ficaria nulo, isso geraria um erro e fecharia a aplicação. Ou seja, você não pode deixar de hipótese nenhuma um item nulo, mas para isso o Kotlin tem uma solução; Null Safety (Nulo seguro) podemos colocar dentro de um objeto um valor nulo, ou seja, não existe o carro ainda, mas em algum momento o carro vai existir.

Exemplo a baixo do Kotlin;

fun main() {  
 var nome: String = "alleph" // Variavel correta  
 var sobrenome: String? // esse ? indica que essa variavel pode receber um valor nullo, ou ate mesmo um texto.  
}

## Usando Java que não tem o Null Safaty

### Null Safaty (Erro em Java)

package Variaveis.NullSafaty;  
  
  
class Carro{  
 String cor = "Prata";  
  
 void acelerar (){}  
}  
  
public class NullSafaty {  
 public static void main(String[] args) {  
  
 // Imagina que não instanciamos o carro  
 Carro carro = null;  
  
 // Imagina que o carro vai ser instanciado em outro local, mas acabou não sendo instanciado  
  
  
 // E ai tentamos usar o carro  
 carro.acelerar();  
  
 // E ai iria gerar o erro NullPointerExecption  
 }  
}

### Null Safaty (Erro tempo de execução)

Imagina que temos um botão, e o erro vai acontecer quando o usuário clicar nele, isso se chama tempo de execução é quando o programa está em funcionamento

package Variaveis.NullSafaty;  
  
  
class Carro{  
 String cor = "Prata";  
  
 void acelerar (){}  
}  
  
public class NullSafaty {  
 public static void main(String[] args) {  
  
 }  
   
 void cliqueBotao(){  
  
 // Imagina que não instanciamos o carro  
 Carro carro = null;  
  
 // Imagina que o carro vai ser instanciado em outro local, mas acabou não sendo instanciado  
  
  
 // E ai tentamos usar o carro  
 carro.acelerar();  
  
 // E ai iria gerar o erro NullPointerExecption  
 }  
}

## Usando o Kotlin que tem o Null Safaty (ERRO)

package Variaveis.NullSafaty  
  
class Carro {  
 var cor = "Prata"  
  
  
 fun acelerar(){}  
}  
  
fun main() {  
  
 // Aqui estou dizendo que essa variavel é do tipo CARRO(Class)  
 var carro: Carro  
  
 // Mas isso daria um erro  
 *println*(carro.cor)  
}

Agora vamos imaginar fazendo da forma correta, onde vamos colocar que o carro pode ser nulo, mas ele vai receber um valor à medida que o código vai crescendo.

package Variaveis.NullSafaty  
  
class CarroK {  
 var cor = "Prata"  
  
  
 fun acelerar(){}  
}  
  
fun main() {  
  
 // Aqui estou dizendo que essa variavel é do tipo CARRO(Class)  
 var carro: CarroK? = null  
  
 // Em algum lugar do codigo o carro vai receber algo e vai deixar de ser nullo  
 carro = CarroK()  
  
 *println*(carro?.cor)

// Pra que serve a chamada? Aqui ele vai verificar se o carro esta nulo, se estiver nulo ele não faz nada, se o carro tiver algo ele chama normalmente a variavel  
}

## Verificando se a variável esta nulo para evitar erro em execução **(Chamada SEGURA > ?)**

package Variaveis.NullSafaty  
  
class CarroK {  
 var cor = "Prata"  
  
  
 fun acelerar(){}  
}  
  
fun main() {  
  
 // Aqui estou dizendo que essa variavel é do tipo CARRO(Class)  
 var carro: CarroK? = null  
  
 *println*(carro?.cor)  
}

Observa que ele verifica o carro é nulo antes com o **?** Se for nulo ele não vai fazer nada, se não for ele segue o processo.

## Uma forma de não exibir o Null se a variável estiver nula

class CarroK {  
 var cor = "Prata"  
  
  
 fun acelerar(){}  
}  
  
fun main() {  
  
 // Aqui estou dizendo que essa variavel é do tipo CARRO(Class)  
 var carro: CarroK? = null  
  
 // Se o carro for nulo e eu quiser exibir uma cor padrao?  
 // ?: Isso é chamado e Elvis Operation  
 // 1 Primeiro vamos testar se o carro é nulo carro?.cor  
 // 2 Se o carro for nulo ele não vai acessar a cor do carro  
 // 3 Então agora ele vai usar a segunda opção que é "Cor padrao"  
 // Funciona bem parecido com o Operador Ternario, ele testa a primeira opção e tem a segunda "Cor padrao"  
 val cor = carro?.cor ?: "Cor padrão"  
  
 *println*(cor)  
   
 *println*(carro?.cor) // Aqui se o carro for nulo vai exibir o nulo.  
}

SAIDA:

Cor padrão

null

# Kotlin – Class – Atributo – Método – Instância

package Variaveis.NullSafaty  
  
class Carro { // Classe  
 var cor = "Prata" // Atributo  
  
  
 fun acelerar(){} // Metodo  
}  
  
fun main() {  
 var nome: String = "alleph" // Variavel correta  
 var sobrenome: String? // esse ? indica que essa variavel pode receber um valor nullo, ou ate mesmo um texto.  
  
  
 var carro = Carro(); // Minha instancia em Kotlin  
}

# Templetes em Kotlin

fun main() {  
 var contagem = 0  
  
 while (contagem < 10) {  
 contagem++  
 *println*("Contagem $contagem")  
 }  
}

# Estruturas condicionais (If, Else)

package EstruturasCondicionais  
  
fun main() {  
 var conta = 200  
 var despesas = 0  
 var sol = false  
  
 if (conta > despesas && sol) {  
 *println*("Você pode ir a praia")  
 }else {  
 *println*("O tempo não esta legal para praia hoje.")  
 }  
}

Outro teste usando apenas a variável condição

val notaAluno = 7  
val condicao = notaAluno >= 6 // Nota do aluno é maior ou igual a 6? isso vai retornar um True ou False  
// e logo em baixo vamos confirmar se devemos entrar no if ou não  
if (condicao) {  
 *println*("Aprovado")  
}else{  
 *println*("Reprovado")  
}  
  
*println*(condicao)

Out: Aprovado

True

## If com IN

//////////////////////// IF COM IN ////////////////  
  
// Observa que o IN ele diz assim  
// Se a opcao estiver em(IN) 1 ate 3  
var opcao = 4  
if (opcao in 1..3) {  
 *println*("Cartao de credito")  
}else if (opcao in 4..5){  
 *println*("Pagamento com dinheiro")  
}else {  
 *println*("Opcao invalida.")  
}

Saída: Pagamento com dinheiro

## WHEN Lembrar do Switch

package EstruturasCondicionais  
  
fun main() {  
 val opcao = 6  
  
 when(opcao){ // Opcao que vamos testar.  
 1 -> *println*("Cartao de credito")  
 2 -> *println*("Extrato bancario")  
 in 3..5 -> *println*("Saldo") // Também podemos usar o in em WHEN como você pode ver.  
 else -> *println*("Opcao não encontrada")  
 }  
}

\*\*\*\* OPCAO = 6 \*\*\*\*

Out: Opcao não encontrada

# Array em Kotlin

package Arrays  
  
fun main() {  
 // Lembrar que com val, voce pode alterar os dados dentro do array, ou seja, mudar o nome das pessoas  
 // Mas nao pode recriar o array, ou seja, refazer um novo array nomes.  
 val nomes = *arrayOf*("Alleph", "Fernanda", "Crixus", "Amora")  
  
 *println*(nomes[1])  
  
 // mudando valor do array  
 nomes[1] = "Anastacia"  
  
 *println*(nomes[1])  
}

# While

Vamos executar aquele bloco de código enquanto aquela condição for verdadeira

Aqui o programa vai contar de 1 ate 10

package Arrays  
  
fun main() {  
 var contagem = 0  
  
 while (contagem < 10) {  
 contagem++  
 *println*(contagem)  
 }  
}

# For

package ArraysELoops  
  
fun main() {  
  
 for (numero in 1..5){  
 *println*("Contagem $numero")  
 }  
}

Observa a facilidade do for em Kotlin, não precisamos nem colocar uma variável fora do for, o próprio Kotlin já entende que numero é uma variável.

# For com arrays

package ArraysELoops  
  
fun main() {  
 val postagens = *arrayOf*(  
 "Ir a praia",  
 "Comprei um carro",  
 "Fui ao cinema",  
 "Comprei um hamburguer"  
 )  
  
  
 for (postagem in postagens){  
 *println*(postagem)  
 }

Oque o código esta fazendo? Ele ta pegando cada item do postagens e colocando dentro do Postagem

E ai imprimimos postagem

Seria mais ou menos assim

For 1 = postagem[0]

For 2 = postagem[1]

For 3 = postagem[2]

Muito mais pratico.

Agora vamos imaginar que eu queira exibir o índice também

## For de array com índices

for ((indice, postagem) in postagens.*withIndex*()){  
 *println*("Titulo: $indice $postagem")  
}

Saida:

Titulo: 0 Ir a praia

Titulo: 1 Comprei um carro

Titulo: 2 Fui ao cinema

Titulo: 3 Comprei um hamburguer

Repara que agora ele me da o índice primeiro e depois a postagem.

Aqui esta as 3 formas

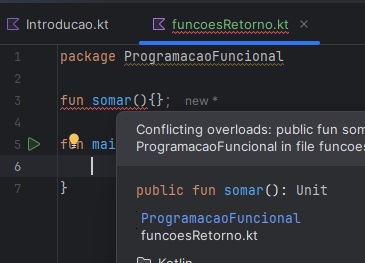
// Pegando somente a postagem  
for (postagem in postagens){  
 *println*(postagem)  
}  
  
// Pegando somente o indice  
  
for (postagem in postagens.*withIndex*()) {  
 *println*("Pegando somente os indices $postagem")  
}  
  
  
// Pegando o indice mais a postagem  
for ((indice, postagem) in postagens.*withIndex*()){  
 *println*("Titulo: $indice $postagem")  
}

# Programação funcional / Funções

A programação funcional ela é orientada a funções.

fun somar(){  
 *println*(2+5)  
}  
  
fun main() {  
 *println*(*somar*())  
}

Lembrar que não podemos ter 2 funções com o mesmo nome dentro do projeto.



## Função com retorno

Vamos precisar colocar o tipo de dado que vai retornar: Int

Fun = para dizer que vamos criar uma função

Soma() = o nome da função

: Int = o tipo de dado que essa função vai retornar

fun soma() :Int{  
 return 5+5  
}  
  
fun main() {  
  
 *println*(*soma*())  
}

fun soma() : Int {  
 return 10+20+30  
}  
  
fun main() {  
  
 val retorno = *soma*();  
 *println*(retorno \* 2)  
}

## Função sem retorno

O Unit ele não e obrigatório, mas ele fica por padrão como se fosse uma função sem retorno.

Lembrando que não precisamos escrever o Unit.

fun soma() : Unit{  
 *println*(5+5)  
}  
  
fun main() {  
  
 *println*(*soma*())  
}

\*\*SEM UNIT\*\*

fun soma() {  
 *println*(5+5)  
}  
  
fun main() {  
  
 *println*(*soma*())  
}

Vai da no mesmo com o Unit ou sem Unit

fun subtrair() {  
 *println*(10-8)  
}  
  
fun main() {  
 *subtrair*()  
}

## Funções inline (Funções em linha)

### **Sem retorno**

fun subtrair() = *println*(10-8)  
  
  
fun main() {  
 *subtrair*()  
}

Saida: 2

### **Com retorno**

Reparar que não precisamos utilizar a palavra RETURN

fun subtrair() : Int = 10-8  
  
  
fun main() {  
  
 *println*(*subtrair*())  
  
}

Saida: 2

## Funções com parâmetros (Normal e Inline)

package ProgramacaoFuncional  
  
// Funçao normal com parametro  
fun somarParametro(a: Int, b:Int) : Int {  
 return a+b  
}  
  
// Funçao inline com parametro  
fun somarParametroInline(a: Int, b : Int) : Int = a+b  
  
fun main() {  
 val somar = *somarParametro*(10,20)  
 *println*(somar) // Saida 30  
  
  
 val somarInline = *somarParametroInline*(20,30)  
 *println*(somarInline) // Saida 50  
}

### Parâmetros nomeados

Se você não colocar o nome do parâmetro, você vai ter que fazer na ordem da função, mas podemos nomear os parâmetros para mudar a ordem da função.

package ProgramacaoFuncional  
  
// Funçao normal com parametro  
fun somarParametro(a: Int, b:Int) : Int {  
 return a+b  
}  
  
// Funçao inline com parametro  
fun somarParametroInline(a: Int, b : Int) : Int = a+b  
  
fun main() {  
 val somar = *somarParametro*(b = 20, a = 10) // Mudamos a ordem e colocamos o nome do parametro  
 *println*(somar) // Saida 30  
  
  
 val somarInline = *somarParametroInline*(b = 20, a = 30) // Observa que mudamos a ordem e colocamos o nome do parametro  
 *println*(somarInline) // Saida 50  
}

### Parâmetro com valor padrão

package ProgramacaoFuncional  
  
fun calculo(numero1 : Int, numero2: Int = 0) : Int = numero1 + numero2  
  
fun main() {  
 val calcular = *calculo*(10)  
 *println*(calcular) // Saida 10  
}

Observa que só passamos um valor como parâmetro, poderíamos ter passado 2 valores, mas deixamos se caso o usuário só digitasse um valor o outro seria padrão 0

# Orientação a objetos com Kotlin

## Classes

class Jogador {  
   
}

## Atributos/Propriedades

class Jogador {  
 **var nome: String = ""  
 var nivel: Int = 0  
 var personagem: String = "Mago"**  
   
   
   
}

## Métodos

class Jogador {  
 var nome: String = ""  
 var nivel: Int = 0  
 var personagem: String = "Mago"  
  
  
 fun soltarMagia(){  
 *println*("Meteoro")  
 }  
}

## Instância

fun main() {  
  
 val jogador1 = Jogador()  
 jogador1.soltarMagia()  
  
}

## Val ou Var para OBJETOS (Atenção)

Quando definimos VAL é para o objeto então dizemos que o objeto não vai mudar, mas os atributos sim.

fun main() {  
  
 val jogador1 = Jogador()  
 jogador1.nome = "Alleph"  
 jogador1.nivel = 150  
 jogador1.personagem = "Mago"  
  
 jogador1.soltarMagia()  
  
}

Imagina o seguinte, temos um objeto caixa 10cm por 10cm, usando o val essa caixa vai ser sempre as mesmas dimensões, mas os objetos que tem dentro dela não tem nada haver, então eles podem sim ser alterados, desde que; caibam dentro da caixa.

Tentar sempre usar o VAL para instância objetos, porque você pode se enganar e sem perceber instanciar outro objeto no meio do programa e aquela “caixa com aquelas dimensões pode se perde”

Usando var, também pode abrir brechas para pessoas más intencionadas, elas podem criar um novo objeto no meio do programa.

Outra atenção; se você utilizar var jogador1 = Jogador() você vai guarda seus atributos dentro de jogador1, como: poder, ataque, defesa, armadura..

Mas se você instancia novamente jogador1 = Jogador(), você vai perde todos os atributos que você tinha colocado, tudo vai embora para da espaço para o novo jogador.

## Métodos com parâmetro

class Jogador {  
 var nome: String = ""  
 var nivel: Int = 0  
 var personagem: String = ""  
 var ataque: Int = 0  
  
  
 fun soltarMagia(){  
 *println*("Meteoro")  
 }  
  
 **fun atacar(forcaAtaque: Int){  
 *println*("Atacando... \nSeu ataque foi de: $forcaAtaque" )**  
 }  
}

fun main() {  
  
 val jogador1 = Jogador()  
 jogador1.atacar(50)

## Métodos com parâmetro padrão

fun atacar(**forcaAtaque: Int = 0){**  
 *println*("Atacando... \nSeu ataque foi de: $forcaAtaque" )  
}

fun main() {  
  
 val jogador1 = Jogador()  
  
 jogador1.atacar()

Você pode ou não passar um parâmetro, se você não passar ele coloca o valor como 0

## Métodos com retorno

// Metodo com retorno  
fun defender(totalDefesa: Int = 0) : String {  
 return "Total de defesa foi $totalDefesa"  
}

fun main() {  
  
 val jogador1 = Jogador()

val defesa = jogador1.defender(120)  
*println*(defesa)

}

## Método com retorno inline

A função aqui espera receber um INTEIRO ( totalDefesa )

E espera uma saída de String (Total de defesa foi...)

Então como ler a função

Usuário tem uma defesa(inteiro) e vamos colocar um texto de saída (String) para mostrar isso na tela.

fun defender(totalDefesa: Int = 0) : String = "Total de defesa foi $totalDefesa"

fun main() {  
  
 val jogador1 = Jogador()  
  
 val defesa = jogador1.defender(120)  
 *println*(defesa)  
  
 jogador1.soltarMagia()  
  
}

## Sobrecarga de métodos (Polimorfismo)

Polimorfismo = Quando o mesmo item tem comportamentos diferentes.

Podemos escrever o mesmo método com o mesmo nome, mas precisamos mudar os parâmetros, assim a própria IDE vai entender qual é o método que é para ele pegar.

Observa que nas 3 opções ele consegue entender qual método ele precisa chamar.

package POO.Classes  
  
  
class Usuario {  
  
  
  
 fun logar(email: String, senha: String){  
 *println*("Usuario logado com: $email e senha: $senha")  
 }  
  
 fun logar(email: String, senha: String, telefone: String) {  
 *println*("Usuario logado com: $email, senha: $senha e telefone: $telefone")  
 }  
  
 fun logar(email: String, telefone: Long) {  
 *println*("Usuario logado com: $email e telefone: $telefone")  
 }  
  
}  
  
fun main() {  
  
 val usuario = Usuario();  
  
 usuario.logar("allephn@hotmail.com.br", "1234")  
 usuario.logar("allephn@hotmail.com.br", "12345", "21975575694")  
 usuario.logar("allephn@hotmail.com.br", 21975575694)  
  
}

Assinatura de um método

fun logar(email: String, senha: String, telefone: String) {

Assinatura é composta por nome, parâmetro e seus tipos.

Então devemos sempre quando for usar a sobrecarga criar métodos com assinatura diferentes (nomes e parâmetros)

## Construtores

Oque é um construtor? Ele permite que você obrigue o usuário a passar parâmetros inicias para dentro de uma instancia.

Ou seja, o usuário vai inicializar uma instancia com informações já obrigatórias.

Lembrar que podemos ter vários construtores e o primeiro chamamos de primário.

### Init = inicializar

Ele serve para inicializar algo quando fazemos a instancia do objeto, ali você pode colocar por exemplo para checar algo, iniciar um banco de dados...

class NewUsuario constructor() { // Construtor primario, porque podemos ter um primeiro construtor e um segundo.  
  
  
 var nome: String = ""  
 var sobrenome: String = ""  
 var cpf: String = ""  
  
  
 // init = inicializar  
 init {  
 *println*("Objeto inicializado.")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario()  
  
}

Saida: Objeto inicializado.

### Criando o construtor

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor(nomeP: String, sobrenomeP: String, cpfP: String) {  
  
  
 var nome: String = ""  
 var sobrenome: String = ""  
 var cpf: String = ""  
  
  
 // init = inicializar  
 init {  
 *println*("Objeto inicializado.")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
  
}

### Construtor com dados padrões

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor(nomeP: String = "", sobrenomeP: String = "", cpfP: String = "") {  
  
  
 var nome: String = ""  
 var sobrenome: String = ""  
 var cpf: String = ""  
  
  
 // init = inicializar  
 init {  
 *println*("Objeto inicializado.")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
 val usuario2 = NewUsuario()  
  
}

### Construtor nomeado

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor(nomeP: String = "", sobrenomeP: String = "", cpfP: String = "") {  
  
  
 var nome: String = ""  
 var sobrenome: String = ""  
 var cpf: String = ""  
  
  
 // init = inicializar  
 init {  
 *println*("Objeto inicializado.")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
 val usuario2 = NewUsuario()  
 val usuario3 = NewUsuario(sobrenomeP = "Nogueira")  
  
}  
Observa que o resto dos dados vão ser padrão, eu defini apenas o sobrenome

### Configurando o dados do construtor nos atributos(variáveis)

Lembra do init? Então ele e inicializado logo após o construtor ser criado.

* Não preciso do construtor para usar o init, mas ele geralmente é usado com o construtor

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor(nomeP: String = "Não preenchido", sobrenomeP: String = "Não preenchido", cpfP: String = "Não preenchido") {  
  
  
 var nome: String = ""  
 var sobrenome: String = ""  
 var cpf: String = ""  
  
  
 // init = inicializar  
 init {  
 this.nome = nomeP  
 this.sobrenome = sobrenomeP  
 this.cpf = cpfP  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenomeP, cpf: $cpf")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
 val usuario2 = NewUsuario()  
 val usuario3 = NewUsuario(sobrenomeP = "Nogueira")  
  
}

Objeto inicializado.

**nome: alleph, sobrenome: nogueira, cpf: 15679025701**

Objeto inicializado.

**nome: Não preenchido, sobrenome: Não preenchido, cpf: Não preenchido**

Objeto inicializado.

**nome: Não preenchido, sobrenome: Nogueira, cpf: Não preenchido**

Atenção não é necessário a palavra constructor já que estávamos usando o init

### Criando atributos direto no construtor

Lembrar de usar o var, para o Kotlin entender que isso é uma propriedade(atributo)

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor(   
 var nome: String = "",   
 var sobrenome: String = "",  
 var cpf: String = "") {  
   
   
 // init = inicializar  
 init {  
  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenome, cpf: $cpf")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
 val usuario2 = NewUsuario()  
 val usuario3 = NewUsuario(sobrenome = "Nogueira")  
  
}

Objeto inicializado.

**nome: alleph, sobrenome: nogueira, cpf: 15679025701**

Objeto inicializado.

**nome: , sobrenome: , cpf:**

Objeto inicializado.

**nome: , sobrenome: Nogueira, cpf:**

Observa como o código fica bem menor.

### Construtor secundário

Lembrar que se você tiver um construtor primário você vai precisar chamar ele, se você não tiver você pode seguir com ele.

class NewUsuario constructor( var nome: String = "", var sobrenome: String = "", var cpf: String = "") {  
  
  
 init {// init = inicializar  
  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenome, cpf: $cpf")  
 }  
  
 constructor(nomeParametro: String): this(nome, sobrenome = "", cpf = "") {//Construtor secundario  
  
 }

Repara que: Criamos o construtor secundário

Vamos passar para o primário como parâmetro apenas o nome

Em this(nome) fazemos a referencia para o construtor primário, ou seja, para onde o nomeParametro vai jogar os dados, que no caso vai ser para this.nome (atributo), o resto vamos deixar como padrão porque não queremos passar sobrenome e cpf.

Agora chamando o construtor secundário

// Construtor primario  
class NewUsuario constructor( var nome: String = "", var sobrenome: String = "", var cpf: String = "") {  
  
  
 init {// init = inicializar  
  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenome, cpf: $cpf")  
 }  
  
 constructor(nome: String): this(nome, sobrenome = "", cpf = "") {//Construtor secundario  
 *println*("Construtor secundario")  
 }  
}  
  
fun main() {  
 val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
 val usuario2 = NewUsuario()  
 val usuario3 = NewUsuario(sobrenome = "Nogueira")  
  
 val usuarioConstrutorSecundario = NewUsuario("alleph")  
  
}

Agora sim, estamos fazendo uma sobrecarga de construtores, o próprio sistema vai identificar qual assinatura está chegando e vamos chamar o construtor ideal, no caso o construtor secundário ele precisa de uma assinatura de (nome String)

Saída:

Objeto inicializado.

**nome: alleph, sobrenome: nogueira, cpf: 15679025701**

Objeto inicializado.

**nome: , sobrenome: , cpf:**

Objeto inicializado.

**nome: , sobrenome: Nogueira, cpf:**

Objeto inicializado.

**nome: alleph, sobrenome: , cpf:**

**Construtor secundario**

**Objeto inicializado =** isso é nosso init, ele vai ser chamando sempre independente do nosso construtor, mas se tiver um construtor primeiro ele chama o construtor e depois o init.

### Logica do construtor secundário

Agora como é na logica

Passamos os dados para o construtor secundário, o construtor secundário passa os dados para o construtor primário(this), o init entra e logo após chama o construtor secundário.

Como você pode ve a baixo.

Primeiro passa os dados como referencia para o construtor 1

class NewUsuario constructor( var nome: String = "", var sobrenome: String = "", var cpf: String = "") {

Após chama o init já com os dados

init {// init = inicializar  
  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenome, cpf: $cpf")  
}

Após isso que vamos chamar o construtor 2

constructor(nome: String): this(nome, sobrenome = "", cpf = "") {//Construtor secundario  
 *println*("Construtor secundario")  
 }

Objeto inicializado.

nome: alleph, sobrenome: , cpf:

Construtor secundário

CODIGO COMPLETO:

package POO.Classes  
  
  
// Construtor primario  
class NewUsuario constructor( var nome: String = "", var sobrenome: String = "", var cpf: String = "") {  
  
  
 init {// init = inicializar  
  
 *println*("Objeto inicializado.")  
 *println*("nome: $nome, sobrenome: $sobrenome, cpf: $cpf")  
 }  
  
 constructor(nome: String): this(nome, sobrenome = "", cpf = "") {//Construtor secundario  
 *println*("Construtor secundario")  
 }  
}  
  
fun main() {

val usuario = NewUsuario("alleph", "nogueira", "15679025701")  
val usuario2 = NewUsuario()  
val usuario3 = NewUsuario(sobrenome = "Nogueira")  
  
val usuarioConstrutorSecundario = NewUsuario("alleph")

}

Lembrar que; só vai passar os dados para o construtor2, dependendo da assinatura dos métodos, é onde vai fazer a sobrecarga de construtores, a própria IDE vai designar qual construtor vai usar, dependendo da sua entrada de dados.

**No caso da instancia do “alleph” apenas ele entende que esta entrando apenas 1 dado e ele vai procurar o construtor que só quer receber apenas 1 dado**

## Herança com Metodos

Vamos herda tudo que vem de outra classe, objeto..

**Open x final**

Open é para abrir a classe, você pode modificar

Final é para fechar.

Imagina uma conversa, FINAL é quando acabamos a conversa é o ponto final e acabou, é isso e ninguém muda.

Podemos levar em consideração isso para a classe, onde colocamos open e oque esta ali esta aberto para ser herdado.

O mesmo vale para os métodos, devemos colocar o open nos métodos que vamos deixar os herdeiros usarem.

Nesse exemplo estamos usando a classe com Open, estamos usando o polimorfismo,construtor e o toString

package POO.Classes  
  
open class Animal(var cor: String, var nome: String, var tamanho: String) {  
  
 // Open = aberto para ser usado  
 open fun somAnimal() = ""  
  
 // Override reescrevendo um metodo  
 override fun toString(): String {  
 return "$nome (cor: $cor, tamanho: $tamanho)"  
 }  
  
  
}  
// Repara que quando a classe pai possui um construtor, a classe filha também deve ter.  
class Cachorro(cor: String, nome: String, tamanho: String) : Animal(cor, nome, tamanho) {  
  
  
 // Override estamos reescrevendo o metodo  
 override fun somAnimal(): String { // Herdando com polimorfismo  
 return "Cachorro lantido..."  
 }  
}  
  
class Gato(cor: String, nome: String, tamanho: String) : Animal(cor, nome, tamanho) {  
  
 // Override estamos reescrevendo o metodo  
 override fun somAnimal(): String {  
 return "Gato miando..."  
 }  
}  
  
  
fun main() {  
 val cachorro = Cachorro("Branco/Amarelo", "Crixus", "Pequeno")  
 *println*(cachorro)  
  
  
 val gato = Gato("Marom", "Anastacia", "Pequeno")  
 *println*(gato)  
  
}

Repara que se você coloca o construtor na classe pai, devemos colocar em todas as classes filhas também.

### Herança com polimorfismo

No polimorfismo precisamos usar a palavra chave override, quando quiser sobrescrever um método, ficando dessa maneira.

open class Animal(var cor: String, var nome: String, var tamanho: String) {  
  
 // Open = aberto para ser usado  
 open fun somAnimal() = ""  
  
 // Override reescrevendo um metodo  
 override fun toString(): String {  
 return "$nome (cor: $cor, tamanho: $tamanho)"  
 }  
  
  
}  
  
class Cachorro(cor: String, nome: String, tamanho: String) : Animal(cor, nome, tamanho) {  
  
  
 // Override estamos reescrevendo o metodo  
 override fun somAnimal(): String { // Herdando com polimorfismo  
 return "Cachorro lantido..."  
 }

Repara que estamos usando o método com o mesmo nome, mas com formas diferentes.

Lembrar que também precisamos colocar o open em somAnimal() para informar que ele pode ser sobrescrito.

* Por padrão todo método vai ser final, você precisa alterar os que você quer reutilizar.

### SUPER recuperando uma parte do código

package POO.Classes  
  
open class Animal(var cor: String = "", var nome: String = "", var tamanho: String = "") {  
  
 // Open = aberto para ser usado  
 open fun somAnimal() = *println*("Som..")  
  
 // Override reescrevendo um metodo  
 override fun toString(): String {  
 return "$nome (cor: $cor, tamanho: $tamanho)"  
 }  
  
  
}  
// Repara que quando a classe pai possui um construtor, a classe filha também deve ter.  
class Cachorro(cor: String, nome: String, tamanho: String) : Animal(cor, nome, tamanho) {  
  
  
 // Override estamos reescrevendo o metodo  
 override fun somAnimal(){ // Herdando com polimorfismo  
 super.somAnimal()  
 *println*("latindo...")  
 }  
}  
  
class Gato(cor: String, nome: String, tamanho: String) : Animal(cor, nome, tamanho) {  
  
 // Override estamos reescrevendo o metodo  
 override fun somAnimal() {  
 super.somAnimal()  
 *println*("miando...")  
 }  
}  
  
  
fun main() {  
 val cachorro = Cachorro("Branco/Amarelo", "Crixus", "Pequeno")  
 *println*(cachorro.somAnimal())  
 *println*(cachorro)  
  
  
 val gato = Gato("Marom", "Anastacia", "Pequeno")  
 *println*(gato.somAnimal())  
 *println*(gato)  
  
}

Saida:

**Crixus (cor: Branco/Amarelo, tamanho: Pequeno)**

**Som..**

**latindo...**

Imagina que no método tivesse 50 linhas de código e queríamos recuperar essas linhas e apenas adicionar mais algumas coisas, então em vez de rescrever todo o código, vamos apenas da um SUPER.METODO, e recuperar todos esses dados e incrementar oque quiser.

## Classes concretas e abstratas

Temos uma classe chamada Pessoa > Homem e Mulher

Então não faz sentido a gente instanciar uma classe Pessoa, ela é uma classe genérica o correto e instanciar um classe Homem e Mulher, com os dados genéricos da classe Pessoa(nome, sexo, cpf)

Então mudamos o OPEN pelo Abstract

package POO.Classes  
  
abstract class Pessoa (var nome: String = "", var cpf: String = "", var sexo: String = "") {  
  
 open fun opcaoBanheiro() = *print*("Vou ao banheiro ")  
  
 override fun toString(): String {  
 return "Pessoa(nome='$nome', cpf='$cpf', sexo='$sexo')"  
 }  
}  
  
class Homem(nome: String, cpf: String, sexo: String) : Pessoa(nome, cpf, sexo) {  
  
 override fun opcaoBanheiro() {  
 super.opcaoBanheiro()  
 *println*("masculinho")  
 }  
  
}  
  
  
class Mulher(nome: String, cpf: String, sexo: String) : Pessoa(nome, cpf, sexo) {  
  
 override fun opcaoBanheiro() {  
 super.opcaoBanheiro()  
 *println*("toalete")  
 }  
}  
  
fun main() {  
  
 val homem = Homem("alleph", "15679025701", "Masculinho")  
 *println*(homem)  
 homem.opcaoBanheiro()  
  
}

Então nesse código vamos ter

1 Classe abstrata

2 Classes concretas(Homem, Mulher)

### Métodos abstratos

Também podemos ter métodos abstratos, métodos que não podem ser inicializado apenas herdados para uso em outros locais.

Metodos abstrato não podem ter um corpo e toda a classe que herda vai ser obrigado a usar esse método abstrato

abstract class Pessoa (var nome: String = "", var cpf: String = "", var sexo: String = "") {  
  
 open fun opcaoBanheiro() = *print*("Vou ao banheiro ")  
  
 override fun toString(): String {  
 return "Pessoa(nome='$nome', cpf='$cpf', sexo='$sexo')"  
 }  
  
  
 abstract fun dormir() // Método sem corpo.  
}

Classe completa (Exemplo com método abstract)

package POO.Classes  
  
abstract class Pessoa (var nome: String = "", var cpf: String = "", var sexo: String = "") {  
  
 open fun opcaoBanheiro() = *print*("Vou ao banheiro ")  
  
 override fun toString(): String {  
 return "Pessoa(nome='$nome', cpf='$cpf', sexo='$sexo')"  
 }  
  
  
 abstract fun dormir()  
}  
  
class Homem(nome: String, cpf: String, sexo: String) : Pessoa(nome, cpf, sexo) {  
  
 override fun opcaoBanheiro() {  
 super.opcaoBanheiro()  
 *println*("masculinho")  
 }  
  
 override fun dormir() {  
 *println*("Indo dormir com barriga para baixo")  
 }  
}  
  
  
class Mulher(nome: String, cpf: String, sexo: String) : Pessoa(nome, cpf, sexo) {  
  
 override fun opcaoBanheiro() {  
 super.opcaoBanheiro()  
 *println*("toalete")  
 }  
  
 override fun dormir() {  
 *println*("Indo dormir com barriga para cima")  
 }  
  
  
}  
  
fun main() {  
  
 val homem = Homem("alleph", "15679025701", "Masculinho")  
 *println*(homem)  
 homem.opcaoBanheiro()  
 homem.dormir()  
  
}

## Encapsulamento (Modificadores de acesso = Public, Private, Protected, Internal, Padrão)

Serve para esconder detalhes da sua aplicação, dando mais segurança, serve para controla o acesso a parte do código, lembrando do public, private, protect...

* **Public** = Visível em todos os lugares
* **Private** = Visível apenas dentro da classe
* **Protected** = Visível apenas dentro da classe ou subclasses (com herança)
* **Internal** = visível em todo lugar dentro de um modulo (Ele é visível somente dentro dos itens que estão dentro do modulo
* **Padrão** = caso não tenha sido definido um modificador vai ficar como o padrão o public

Exemplos práticos

### Public ou Padrão

package POO.Classes  
  
class Carro {  
 var modelo = ""  
  
 fun acelerar(){  
 *println*("Acelerar o carro")  
 }  
}

Essa classe ela está como padrão então eu posso usar ela em outro local

Repara que estamos utilizando a classe em outro local

package POO.Classes  
  
fun main() {  
 val carro = Carro()

carro.modelo = “C4”  
 carro.acelerar()  
}

### Private

Vamos imaginar agora que estamos utilizando o private no atributo, e ai vamos poder acessar a classe pq ela e public, mas o atributo não porque ele é private.

Outro problema é se uma classe for private, você não vai conseguir herda nada dela, mesmo ela estando aberta (Open) o private diz que ela é restrita.

Agora imagina que queremos um método que o usuário do carro não possa alterar por questões de segurança. Vamos imaginar que se ele alterar a quantidade de combustível ele pode fazer o carro pegar fogo.

package POO.Classes  
  
open class Carro {  
 var modelo = ""  
  
 fun acelerar(){  
 *println*("Acelerar o carro")  
 }  
   
 private fun quantidadeDeCombustivel(){  
 val quantidadeDeCombustivel = 10  
 }  
}

#### Utilizando o private

Dessa forma o usuário não vai poder controlar a quantidade de combustível quando ele instanciar a classe, mas o carro precisa adicionar combustível, e quando vamos adicionar? Quando ele chamar o método acelerar, porque o private pode ser acessado somente de dentro da própria classe.

open class Carro {  
 var modelo = ""  
  
  
 private fun quantidadeDeCombustivel(){  
 *println*("Adicionando combustivel")  
 val quantidadeDeCombustivel = 10  
 }  
  
 fun acelerar(){  
 quantidadeDeCombustivel()  
 *println*("Acelerando o carro")  
 }  
  
  
}

### Protected

Imagina que temos o Carro e o método injetar combustível, mas esse método ele deve ser protegido, ai ninguém consegue alterar, mas imagina se você tem um Gol e um C4, eles tem injeções de combustível diferente certo? Um vai injetar mais que o outro, então para isso vamos ter que utilizar esse método em outra classe, sendo private não vamos conseguir, mas com o protected vamos conseguir, porque ele pode ser utilizado apenas em classes que vao herda, no casso C4 e Gol herdam de Carro.

Segue exemplo a baixo.

package POO.Classes  
  
open class Carro {  
 var modelo = ""  
  
  
 protected open fun quantidadeDeCombustivel(){  
 *println*("Adicionando combustivel")  
 }  
  
 fun acelerar(){  
 quantidadeDeCombustivel()  
 *println*("Acelerando o carro")  
 }  
}  
  
  
class C4 : Carro() {  
  
 override fun quantidadeDeCombustivel(){  
 *println*("Adicionando 10 ml de combustivel")  
 }  
}  
  
class Gol: Carro() {  
 override fun quantidadeDeCombustivel(){  
 *println*("Adicionando 5ml de combustivel")  
 }  
  
}  
  
fun main() {  
 val carro = Carro()  
 carro.acelerar()  
}

Saida:

Adicionando 10 ml de combustivel

Acelerando o carro

Adicionando 5ml de combustivel

Acelerando o carro

Com o protected diferente do private conseguimos utilizar em classes que estão herdando e ainda sobrescrever o método.