# Extensão

Usando a extensão, poderemos adicionar ou remover algumas funcionalidades do método, mesmo sem herdá-las ou modificá-las. As extensões são resolvidas estatisticamente. Na verdade, ele não modifica a classe existente, mas cria uma função que pode ser chamada que pode ser chamada com uma operação de ponto.

# Extensão de Função

Na extensão da função, o Kotlin permite definir um método fora da classe principal. No exemplo a seguir, veremos como a extensão é implementada no nível funcional.

class Alien {

var skills : String = " null "

fun printMySkills() {

print(skills)

}

}

fun main() {

var a1 = Alien()

a1.skills = "JAVA"

a1.printMySkills()

var a2 = Alien()

a2.skills = "SQL"

//a2.printMySkills()

var a3 = Alien()

a3.skills = a1.addMySkills(a2)

a3.printMySkills()

}

fun Alien.addMySkills(a:Alien):String{

var a4 = Alien()

a4.skills = this.skills + " " +a.skills

return a4.skills

}

No exemplo acima, não temos nenhum método dentro da classe "Alien" denominada "addMySkills ()", no entanto, ainda estamos implementando o mesmo método em outro lugar fora da classe. Essa é a mágica da extensão.

O trecho de código acima gerará a seguinte saída no navegador.

**JAVA** SQL

# Extensão de Objeto

Kotlin fornece outro mecanismo para implementar a funcionalidade estática do Java. Isso pode ser alcançado usando a palavra-chave “objeto complementar”. Usando esse mecanismo, podemos criar um objeto de uma classe dentro de um método de fábrica e, posteriormente, podemos chamar esse método usando a referência do nome da classe. No exemplo a seguir, criaremos um "objeto complementar".

fun main() {

println("Heyyy!!!"+A.show())

}

class A {

companion object {

fun show():String {

return("You are learning Kotlin")

}

}

}

O trecho de código acima produzirá a seguinte saída no navegador.

Heyyy!!!You are learning Kotlin

O exemplo acima parece estático em Java, no entanto, em tempo real, estamos criando um objeto como um variável membro dessa mesma classe. É por isso que ele também está incluído na propriedade extension e pode ser chamado alternativamente como uma extensão de objeto. Você basicamente estende o objeto da mesma classe para usar algumas das funções de membro.

# Kotlin - Classes de dados

Uma classe pode ser marcada como uma classe de dados sempre que marcada como "dados". Este tipo de classe pode ser usado para manter os dados básicos separados. Fora isso, ele não fornece nenhuma outra funcionalidade.

Todas as classes de dados precisam ter um construtor primário e todo o construtor primário deve ter pelo menos um parâmetro. Sempre que uma classe é marcada como dados, podemos usar algumas das funções incorporadas dessa classe de dados, como “toString ()”, ”hashCode ()”, etc. Qualquer classe de dados não pode ter um modificador como abstrato, aberto ou interno. A classe de dados também pode ser estendida para outras classes. No exemplo a seguir, criaremos uma classe de dados.

fun main() {

val book: Book = Book(" Kotlin ", "Editora X", 5)

println("Name of the Book is--"+book.name) // " Kotlin "

println("Puclisher Name--"+book.publisher) // "Editora X"

println("Review of the book is--"+book.reviewScore) // 5

book.reviewScore = 7

println("Printing all the info all together--"+book.toString())

//using inbuilt function of the data class

println("Example of the hashCode function--"+book.hashCode())

}

data class Book(val name: String, val publisher: String, var reviewScore: Int)

O trecho de código acima produzirá a seguinte saída no navegador, onde criamos uma classe de dados para armazenar alguns dados e, a partir da função principal, acessamos todos os seus membros de dados.

Name of the Book is-- Kotlin

Puclisher Name--Editora X

Review of the book is--5

Printing all the info all together--Book(name= Kotlin , publisher=Editora X, reviewScore=7)

Example of the hashCode function---729312998

# Kotlin - Classe Selada

Este tipo de classe é usado para representar uma hierarquia de classes restrita. Selado permite que os desenvolvedores mantenham um tipo de dados de um tipo predefinido. Para criar uma classe selada, precisamos usar a palavra-chave "selada" como um modificador dessa classe. Uma classe selada pode ter sua própria subclasse, mas todas essas subclasses precisam ser declaradas dentro do mesmo arquivo Kotlin junto com a classe selada. No exemplo a seguir, veremos como usar uma classe selada.

sealed class MyExample {

class OP1 : MyExample() // MyExmaple class can be of two types only

class OP2 : MyExample()

}

fun main() {

val obj: MyExample = MyExample.OP2()

val output = when (obj) { // defining the object of the class depending on the inuputs

is MyExample.OP1 -> "Option One has been chosen"

is MyExample.OP2 -> "option Two has been chosen"

}

println(output)

}

No exemplo acima, temos uma classe selada chamada "MyExample", que pode ser de apenas dois tipos - um é "OP1" e outro é "OP2". Na classe principal, estamos criando um objeto em nossa classe e atribuindo seu tipo em tempo de execução. Agora, como essa classe "MyExample" é selada, podemos aplicar a cláusula "when" em tempo de execução para implementar a saída final.

Na classe selada, não precisamos usar nenhuma declaração "else" desnecessária para complexar o código. O trecho de código acima produzirá a seguinte saída no navegador.

**option** Two has been chosen

# Kotlin - Tratamento de exceções

**O que é uma exceção?**

Segundo o site dicio, temos a seguinte definição:

“Ruptura de uma regra ou norma; desvio de um padrão estabelecido; rompimento do que se considera normal: não há regra sem exceção.”

Na computação, temos o tratamento de exceção e segundo a wikipedia (enciclopédia livre):

É o mecanismo responsável pelo tratamento da ocorrência de condições que alteram o fluxo normal da execução de programas de computadores. Para condições consideradas partes do fluxo normal de execução.

**Como faço para lançar uma exceção?**

É muito simples, como podemos ver no exemplo abaixo, basta usar a palavra reservada throw antes da exceção.

fun main() {

throw Exception("Testando uma exceção")

}

As classes de exceção (exception) no Kotlin são descendentes da classe java.lang.Throwable e é possível passar algumas informações para instanciar um novo objeto, como:

* Uma mensagem (detailMessage), que serve para detalhar o erro ocorrido para o desenvolvedor;
* Uma causa (cause) vinculada, ou seja, uma outra exceção que deu origem ao erro. Esse valor pode ser nulo e isso significa que a causa é desconhecida ou inexistente.

**Como faço para capturar uma exceção lançada?**

A chamada da função deve estar dentro do bloco try, para que seja possível captura-lá e em seguida declarar o bloco catch com o tratamento desejado.

fun main() {

try {

// trecho de código com divisão por zero

} catch (ex: ArithmeticException) {

// tratando a exceção NumberFormatException

}

}

Também podemos tratar várias exceções para um mesmo bloco de código. Isso é possível usando um bloco catch para cada tipo de exceção, sendo que as classes mais genéricas devem ficar após as classes mais específicas. Por exemplo, Exception depois de NumberFormatException.

fun main() {

try {

// função de processamento de notas

} catch (ex: ArithmeticException) {

// tratando a exceção ArithmeticException

} catch (ex: NumberFormatException) {

// tratando a exceção NumberFormatException

} catch (ex: Exception) {

// tratando as exceções mais genéricas

}

}

**Expressões com try**

O bloco try pode ser tratado como uma expressão em Kotlin e isso significa que ele devolverá um resultado.

val resultado: Int? = try { converter(variavel) } catch (e: NumberFormatException) { null }

O valor retornado sempre será a última da expressão do bloco try/catch. Para exemplo acima temos:

Para os casos de sucesso = o retorno da função converter(variavel)

Para os casos de falha = null