Desenvolvimento para dispositivos móveis

Introdução à Orientação a Objetos em Kotlin – Recursos Avançados

Professor Msc. Fabio Pereira da Silva
E-mail: fabio.pereira@faculdadeimpacta.com.br

Introdução

- Teve início em 2010, lançamento em 2016 e última release há 3 meses (1.3);
- Iniciativa da JetBrains em trazer novos conceitos a linguagem Java;
- Foco em ser concisa, expressiva, ferramental e na interoperabilidade;
- Multiparadigma (Funcional e Orientada à objetos);
- Inspirada em Java, Scala, C# e Groovy;
- Plataformas alvo:
  - JVM;
  - Js;
  - Android;
- Feita para uso interno mas diversas empresas utilizam, tais como American Express GitHub, Netflix, NBC News Digital, Uber

Convenções

- Segue as convenções de Java;
- camelCase para métodos e atributos;
- UpperCase para tipos, nomes de classes e objetos;
- Ponto e vírgula é opcional (requerido somente para separar atributos de funções em enum class);
- caminho reverso dos pacotes;
- múltiplas classes por arquivo;
- pacotes não precisam ser idêntico ao caminho;

Definições importantes

- Packages: são utilizados para a organização do código em pacotes, ou seja, em subsistemas.
- Imports: são utilizados para realizar a importação de bibliotecas que serão trabalhadas durante o desenvolvimento da aplicação
- Annotations: são metadados, por meio dele é possível relacionar classes, métodos, atributos, parâmetros e variáveis, e podem ser utilizados em tempo de execução.

Sintaxe

- Pacotes e importações

```
// everything in 'org.example' becomes
accessible  import org.example.*

import org.example.Message // Message is
accessible
// testMessage stands for
'org.test.Message'  import org.test.Message
as testMessage
```

Exemplo

```
import kotlin.time.Duration
import kotlin.time.ExperimentalTime
@ExperimentalTime //uso de annotation
fun main() {
//sampleStart
val duration = Duration.milliseconds(120000)
println("There are ${duration.inWholeSeconds} seconds in
${duration.inWholeMinutes} minutes") //sampleEnd
}
```

Identificadores e palavras chaves

- Hard keywords que sempre são interpretados como keywords;
- Soft e Modifier keywords podem ser usadas como identificadores dependendo do contexto

Kotlin keywords List

as	break	class	continue	do	else
false	for	fun	if	in	interface
is	null	object	package	return	super
this	throw	true	try	typealias	typeof
val	var	when	while		

Amarração

- Estaticamente tipada para JVM;

```
val dyn: dynamic = … ( para Web
)
```

- Escopo aninhado e estático;
- Inferência de tipos durante a compilação;

Variáveis Globais

- Kotlin suporta a declaração de variáveis de nível elevado (top-level), ou seja, fora de uma função ou classe:

```
val preco = 2.99f
fun main() {
    val desconto = 0.5f
        println(preco - preco * desconto)   //Saída: 1.495
}
```

- A variável de nível elevado “preco” pode ser utilizada em qualquer lugar no projeto, inclusive em outros arquivos, enquanto a variável local “desconto ”pode ser utilizada somente dentro da função onde foi declarada

Variáveis Globais

- Caso uma variável local, declarada dentro de uma função, tenha o mesmo nome de uma variável de nível elevado, a variável de nível elevado será sombreada dentro da função.

```
val preco = 2.99f
fun main() {
val preco = 4.99f
val desconto = 0.5f
println(preco - preco * desconto)
}
```

```
//Saída:
2.495
```

Exercícios

- 1) Crie uma variável global que indique o percentual de reajuste de um determinado produto, que deve ser calculado a partir do método main definido na função. Calcule o novo salário e exiba-o em console.

Variáveis estáticas

- não possui “static” como em Java mas permite o uso de “companion object” para produzir o efeito semelhante;

```
class Car(val horsepowers: Int) {
companion object Factory {
val cars = mutableListOf<Car>()
fun makeCar(horsepowers: Int):
    Car {  val car =
    Car(horsepowers)
   cars.add(car)
return car
}}}
fun main(){
val car = Car.makeCar(150)
println(Car.Factory.cars.size)}
```

Endereçamento de memória

- Kotlin rodando na JVM não permite endereçamento de memória, entretanto rodando nativo no Android é possível;
- Kotlin não possui suporte a variáveis anónimas.

```
// Kotlin Native v0.5
import
kotlinx.cinterop.*


fun main(args: Array<String>) {
    val intVar =
    nativeHeap.alloc<IntVar>()
    intVar.value = 42
    with(intVar) { println("Value is $value, address is $rawPtr") }
}
    nativeHeap.free(intVar) //Value is 42, address is 0xc149f0
```

Coletor de lixo

- Utiliza o que está implementado na JVM (similar a Java com sistema de gerações), geralmente descarta os objetos sem referência.

Memória e Companion Object

- Memória primária segue o padrão de pilha e monte como em Java;
- Memória secundária para ações de I/O e serialização basta importar as classes de Java para tal.
- Companion Object permite que um método ou função de uma classe seja chamado diretamente sem necessariamente criar uma instância do objeto.

Exemplo sem Companion Object

```
class ToBeCalled {
    fun callMe() = println("You are calling me :)")
}


fun main(args: Array<String>) {
    val obj = ToBeCalled()

    // calling callMe() method using object obj
    obj.callMe()
}
```

Exemplo com Companion Object

```
class ToBeCalled {
    companion object Test {
        fun callMe() = println("You are calling me :)")
    }
}


fun main(args: Array<String>) {
    ToBeCalled.callMe()
}
```

Exercícios

2) A classe Aluno possuí os atributos id, nome e curso. Receba todos os parâmetros no método construtor e exiba-os em console, sem a utilização de métodos estáticos.
3) A classe Aluno possuí os atributos id, nome e curso. Receba todos os parâmetros no método construtor e exiba-os em console. Adicione um companion Object com a mensagem chamando método estático, que deve ser acessado sem realizar a instância direta da classe.

Tipagem Estática vs. Dinâmica

- Por ser uma linguagem com tipagem estática, Kotlin ainda precisa interoperar com ambientes sem tipo, como o ecossistema JavaScript.
- Para facilitar esses casos de uso, o tipo **dynamic** está disponível na linguagem

```
val dyn: dynamic = ...
```

- O tipo **dynamic** não é suportado no código destinado à JVM
- O tipo **dynamic** desliga o checador de tipos do Kotlin

Tipos Básicos - Números

Tipo	Tamanho (bits)	Menor Valor	Maior Valor
Byte	8	-128	127
Short	16	-32768	32767
Int	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
Long	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

```
val int = 1
val long = 1L
val byte: Byte = 1
```

Tipos Básicos - Números

Tipo	Tamanho (bits)	Bits Significativos	Bits do Expoente	Dígitos Decimais
Float	32	24	8	6-7
Double	64	53	11	15-16

```
val float_pi = 3.14f                  //Float
val double_pi = 3.14                  //Double
val notaçãoCientífica = 1.235e10 //Double
```

Tipos Básicos - Números

- Representações:

  Decimal: 25
  Hexadecimal: **0x**19
  Binário: **0b**11001
  Octal não é suportado

- Pode-se usar underscores para maior legibilidade:

```
val bytes = 0b11010010_01101001_10010100_10010010
val long = 1234_5678_9012_3456L
val hex = 0xFF_EC_DE_5E
```

Tipos Básicos - Números

- **NaN** é considerado igual a ele mesmo
- **NaN** é considerado maior que qualquer outro elemento, incluindo POSITIVE_INFINITY
- **-0.0** é considerado menor que 0.0

```
fun main(){
val nan:Double = Double.NaN
val inf:Double = Double.POSITIVE_INFINITY
println(nan.compareTo(nan))
    println(nan.compareTo(inf))

println(0.0.compareTo(-0.0))
}
//Saída:  0
          1
          1
```

Tipos Básicos - Characters

- São representados com aspas simples

```
var c = 'a'
```

- Caracteres especiais são representados com barra invertida

\t	Inserts tab	\’	Inserts single quote character
\b	Inserts backspace	\”	Inserts double quote character
\n	Inserts newline	\\	Inserts backslash
\r	Inserts carriage return	\$	Inserts dollar character

- Para outros caracteres utiliza-se a sintaxe Unicode

```
var c = '\uFF00'
```

Tipos Básicos - Characters

- Diferente de linguagens como C, não podem ser tratados diretamente como números

```
fun check(c: Char) {
    if  (c == 1) {//...} //ERROR: Operator '==' cannot
                         be applied to 'Char' and 'Int'
}
```

- Podem ser convertidos explicitamente para inteiros

```
fun check(c: Char) {
    if  (c.toInt == 1) {//...}
}
```

Tipos Básicos - Booleans

- Só assumem os valores **true** ou **false**
- Operadores em Booleanos:
  - || - OU
  - && - AND
  - ! - negação
- || e && são operadores mais utilizados

Tipos Básicos - Array

- Arrays em Kotlin são **Invariantes**

- var vetor = Array(5) { i -> (i * i).toString() }
  //[“0”, ”1”, ”4”, ”9”, ”16”]

- **arrayOf** permite criar array com tipos diferentes

```
var outroVetor= arrayOf ("1",     2,   '3',  4.0, 5.0f,     6L)
```

- Pode-se fazer operações **get** e **set** com o
  operador **[ ]** outroVetor[0] = 7.0
  println(outroVetor[0]) //printa: 7.0

Tipos Básicos - Array

- Kotlin possui classes especializadas para representar arrays de tipos primitivos: intArrayOf, doubleArrayOf, charArrayOf, …

```
/ Array de int com tamanho 5 e valores [0, 0, 0, 0, 0]
val arr = IntArray(5)
/ Array de int com tamanho 5 e valores [42, 42, 42, 42, 42]
val arr2 = IntArray(5) { 42 }
/ Array de int com tamanho 5 e valores [0, 2, 4, 6, 8]
var arr3 = IntArray(5) { it * 2 }
```

Vetores em Kotlin

```
fun main(){
    // Array com 5 elementos do tipo inteiro [0,0,0,0,0]
    //var vetor1 = IntArray(5)
    // Array com 5 elementos do tipo iguais a 12 [12,12,12,12,12]
    //var vetor1 = IntArray(5,{12})
    // Array com 5 elementos do tipo iguais a [10,20,30,40,50]
    //var vetor1 = IntArray(5) {it}
    var vetor1 = IntArray(5){10*(it+1)}
    //vetor1[3] = 17
    for (valor in vetor1){
        println(valor)
    }
    for ( (indice, valor) in vetor1.withIndex()){
        println("o elemento em vetor [$indice] é $valor")
    }
}
```

Vetores em Kotlin

```
import java.util.*

fun main(args: Array<String>) {
    val valores: IntArray = intArrayOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    val expressaoLambda = Array(10, { i -> i })
    println(Arrays.toString(valores))
    println(Arrays.toString(expressaoLambda))

}
```

Tipos Básicos - Unsigned integers

Tipo	Tamanho (bits)	Intervalo
UByte	8	0 - 255
UShort	16	0 - 65535
UInt	32	0 - 2^{32} - 1
ULong	64	0 - 2^{64} - 1

```
val s: UShort = 1u   // UShort
val l: ULong = 1u    // ULong
val a = 1UL          // ULong
val a1 = 42u         // UInt é o padrão
```

Tipos Básicos - Strings

- São imutáveis
- Os elementos da string são **char** e podem ser acessados pelo operador **[ ]**

```
var str = "Olá"
println(str[0])      //Saída: O
```

- Strings podem ser iteradas com **for**

```
for (c in  str) { println(c) }
```

- Podem ser concatenadas com o operador **+**

```
str = str + ", Mundo!"
println(str)

//Saída: Olá, Mundo!
```

Tipos Básicos - Strings

- Escaped Strings: pode-se utilizar caracteres especiais **\n, \t, ** ...

```
val s = "Olá, Mundo!\n"
```

- Raw String: é delimitada por três aspas duplas (“””) e não contém caracteres especiais

```
val text = """
    for (c in "foo")
        print(c)
    """
```

Tipos Básicos - Strings

- Pode-se remover os espaços em branco com a função **trimMargin()** e marcando o início das linhas com uma barra vertical (**|**)

```
fun main(){
val text = """
|Tell me and I forget.
|Teach me and I remember.
|Involve me and I learn.
|(Benjamin Franklin)  """.trimMargin()

println(text)
}
```

Tipos Básicos - Strings

- Strings podem conter pedaços de código que são avaliados e cujos resultados são concatenados na string. É utilizado o caracter **$** no início das expressões

```
fun main(){
val i = 10
println("i = $i")
val s = "abc"
println("$s.length is ${s.length}")
}
```

```
//Saída:    i = 10
            abc.length
            is 3
```

Intervalos

- Pode-se criar intervalos de valores utilizando a função **rangeTo()** e o operador **..**

```
var intervalo = 0..10  //equivale à 0<= intervalo <=10
```

- Pode ser utilizados em expressões condicionais junto com **in** ou **!in**

```
var num = 5
if(num in 0..10)   //se num >= 0 && num <= 10
        println(num)
```

- Intervalos de tipos Integral (IntRange, LongRange, CharRange) podem ser iterados

```
for (i in 'a'..'e') print(i)     //Saída: abcde  for (i in
4 downTo 1) print(i) //Saída: 4321
```

Checagem de Tipos

- A checagem de tipo são feitas com os operadores **is** e **!is**
- Operador **is** compara o tipo da variável e retorna um booleano **true** se os tipos combinam. O operador **!is** retorna o inverso do operador **is**

```
fun main(){
   var forma: Any? = null
    forma = Circulo()
if   (forma is    Circulo) {
print("é um Circulo")
}    else if  (forma is    Quadrado) {
    print("é um Quadrado")
}    else if  (forma is    Retangulo) {
    print("é um Retângulo")
}
}
class Circulo(){
}
class Quadrado(){
}
class Retangulo(){
}
```

Exercícios

- 4) Crie três classes chamadas, Aluno, Turma e Professor. Na função main, crie um atributo chamado obj instanciando uma das classes, valide se o atributo é do tipo da Classe Aluno, Turma ou Professor e exiba em console o tipo de dado que ele pertence.
- 5) Com base na resolução do exercício 4, se o atributo informado for do tipo Aluno, instancie o objeto com o tipo da Classe Turma. Ao final, apresente o último tipo do objeto instanciado.

Funções - Declaração e Chamada

- Funções em kotlin são declaradas com a palavra-chave **fun**

```
fun dobro(x: Int): Int {
    return 2 * x
}
```

- Utiliza-se a abordagem tradicional para chamar as funções

```
val resultado = dobro(4)
println(resultado)
//Saída: 8
```

Funções - Parâmetros

- Os parâmetros de função são definidos usando a notação Pascal, ou seja, **nome: tipo**. Parâmetros são separados usando vírgulas. Cada parâmetro deve ser explicitamente tipado

```
fun powerOf(number: Int, exponent: Int) { /*...*/ }
```

- Os parâmetros de função podem ter valores padrão, que são usados quando um argumento correspondente é omitido

```
fun read(b: Array<Byte>, inicio: Int = 0, fim: Int = b.size) {}
```

Funções - Single-expression

- Quando uma função retorna uma única expressão, as chaves podem ser omitidas e o corpo é especificado após o símbolo **=**

```
fun dobro(x: Int): Int = x * 2


fun main(){
println(dobro (5))
}
```

Funções - Escopo

- Além das funções de nível superior, as funções do Kotlin também podem ser locais e funções de membro
- Funções Locais: uma função dentro de outra função

```
fun printArea(width: Int, height: Int): Unit {
fun calculateArea(width: Int, height: Int): Int =  width * height
val area = calculateArea(width, height)
println("A área é $area")
}
fun main(){
printArea(5, 2)
 }
```

Funções - Escopo

- Além das funções de nível superior, as funções do Kotlin também podem ser locais e funções de membro
- Funções Membro: é uma função definida dentro de uma classe ou objeto

```
class MinhaClasse() {
    fun ola() {
    print("Olá, Mundo!")
    }
}
    fun main(){
     var obj = MinhaClasse()
      obj.ola()
}
```

Expressões lambda

```
fun main(args: Array<String>){
    val product = { a: Int, b: Int -> a * b }

    println(product(2,3))
}
```

Exercícios

- 6) Crie uma expressão lambda que realize a soma de 2 valores e exiba os resultados em console.
- 7) Crie uma expressão lambda que realiza a subtração de 2 valores e exiba os resultados em console.
- 8) Crie uma expressão lambda que realiza a divisão de 2 valores e exiba os resultados em console.

Operadores Unários

Expressão	Traduzido para
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

```
fun main(){
    var x = 2
println(+x)

var y = 2
println(-y)
println(y.unaryMinus())

var z = true
println(z.not())
println(!z)
println(!!z)

}
```

Operadores Binários

Expressão	Traduzido para
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

```
fun main(){
println(10 == 50)
println(10 != 50)
println(10> 50)
println(10< 50)
println(10 >= 50)
println(10 <= 10)
}
```

Operadores Binários

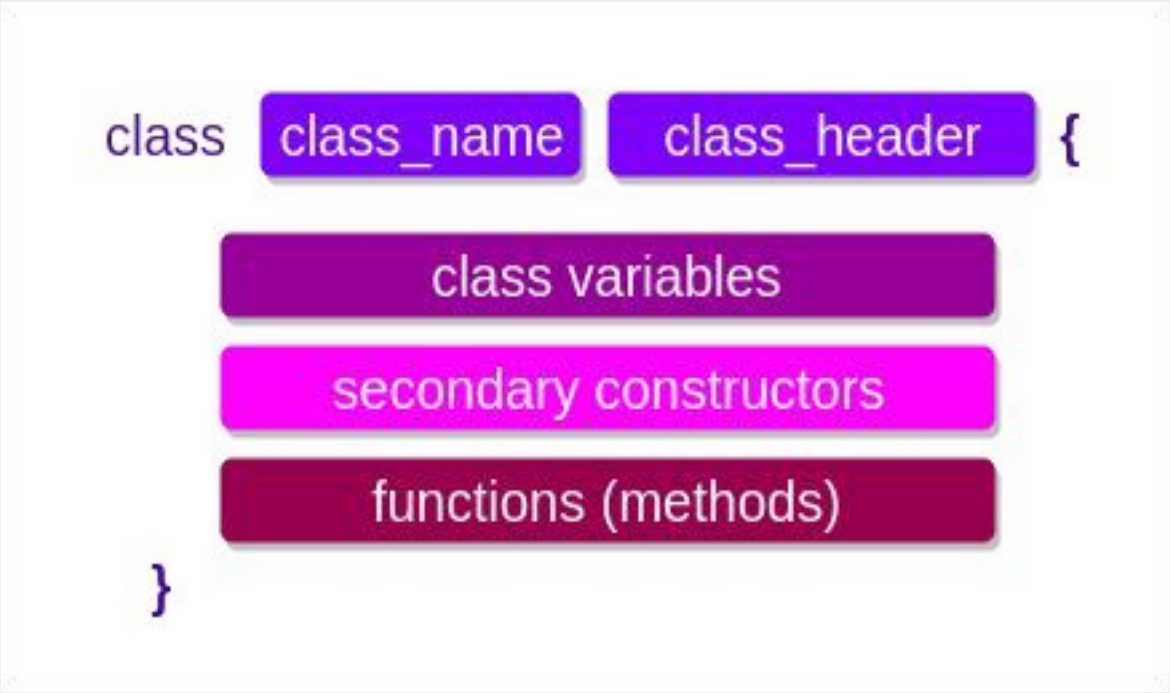
Expressão	Traduzido para
a in b	b.contains(a)
a !in b	!b.contains(a)
a[i]	a.get(i)
a[i] = b	a.set(i, b)

```
fun main(){
var x =       10
var y =arrayOf(0,1,5,8,13,20)
println(x in  y)      //Saída: false
println(x !in y) //Saída: true
println( y[2] )       //Saída: 5

y[2]= x
println( y[2] )
}
```

Classes

- Palavra reservada class



Construtores

- Construtor primário- parte do cabeçalho da classe, não pode conter nenhum código,logica de inicialização feita no bloco init
- Construtor secundário - uso palavra chave constructor

Classes

- Para criar uma classe basta utilizar a palavra reservada class
- O construtor padrão é definido na mesma linha da classe

```
class Aluno(nome: String, idade: Int) {
val nome: String
val idade: Int
init {
// bloco de inicialização
this.nome = nome
this.idade = idade
}
override fun toString(): String {
return "Aluno: $nome, idade: $idade"
}
}
fun main(args: Array<String>) {
val a1 = Aluno("Fabio", 28)
println(a1)
println("Aluno: ${a1.nome}, idade: ${a1.idade}")
}
```

Classes - construtor primário

```
class Person(firstName: String, lastName: String, yearOfBirth: Int)
    {
val fullName = "$firstName $lastName"
var age: Int

init {
age = 2018 - yearOfBirth
}
}


fun main(){
    var person = Person ("Fabio", "Pereira", 1993)
    println(person.age)
}
```

Classes

Declarando uma classe no Kotlin

```
class Person(val name: String, private val birthYear: Int) {  init {
    println("Initializing Person with name $name")
  }

  constructor(name: String, birthYear: Int, isMarried: Boolean) : this(name, birthYear) {
    this.isMarried = isMarried
  }

  // Definindo uma propriedade com o get customizado
  val age: Int
    get() {
      return LocalDateTime.now().year - birthYear
    }

  var isMarried: Boolean = false
    private set // Atribuição apenas dentro da classe
}
```

Construtor Primário com Propriedades

Bloco de inicialização

Construtor Secundário sem propriedades

Propriedade com get customizado

Propriedade com set privado

Classes
Enum

- Possuem características de classe
- Cada constante enum é um objeto
- Como cada enum é uma instância da classe enum, eles podem ser inicializados

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

fun main() {
var c = Color.RED
    println(c.name)
    println(c.ordinal)
    println(c.rgb)
}
```

Saída
RED
0
16711680

Exercícios

- 9) Crie uma classe ENUM contendo 4 notas dos alunos durante o semestre letivo. Exiba cada uma das propriedades em console. Repare que as notas devem ser inicializadas como double.

Data Class

- Lidar com dados e não referências
- toString(), equals(), hashCode(),copy() construídos implicitamente.

```
data class Movie(var name: String, var studio: String, var
rating:  Float)
```

Data Class

```
data class User(var name:String, var id:Int)
fun main(args: Array<String>){
var user1=User("Joao",10)
var user2=User("Joao",10)
println(user1.toString())
if(user1==user2){
println("Iguais")
}
else{
println("Diferentes")
}

var user3=user1.copy()
   println(user3)
}
```

Instancia de classes

Criando uma instância

```
val person1 = Person("Felipe", 1987)

val person2 = Person("Bruna", 1989, false)
```

Modificadores de Visibilidade

Modificador	Visibilidade
public	Visível de qualquer local. É o modificador de visibilidade padrão
protected	Visível apenas para Classes e Sub-Classes
private	Visível apenas para classe
internal	Visível apenas no módulo

Modificador Public

```
// Exemplo de classe em Kotlin e de construtores
class Curso(){
// Atributos da Classe
public var nome: String = "“
public var turma: String = ""
public var predio: String = ""
}

fun main(){
var c1: Curso = Curso()
c1.nome = "Veterinário“
c1.turma = "Zebra"
c1.predio = "Zoológico“

println(c1.nome)
println(c1.turma)
println(c1.predio)
}
```

Mais sobre modificadores de acesso

```
// Operadores de visibilidade
// private >> torna o recurso visivel apenas na classe
// public >> torna o recurso visivel para todas as sub-classes e objetos.
// protected >> torna o recurso visivel para todas as sub-classes, mas não para os objetos.
// internal >> Apenas classes deste arquivo podem utilizar esse recurso.
//  o comando open permite que Carro4 se torne uma super classe.
open class Carro5{
    // Atributos da Classe
    private var modelo: String
    private var ano: Int
    protected var velocidade: Int
    public var cor : String
    // Construtor
    constructor(novoModelo:String, novoAno:Int) {
        println("Iniciando construtor 1 ...")
        this.modelo = novoModelo
        this.ano = novoAno
        this.velocidade = 0
        this.cor = "Branco"
    }
```

Mais sobre modificadores de acesso

```
 fun mudaModelo(novoModelo:String){
        this.modelo = novoModelo
    }
    fun escreveModelo(){
        println(this.modelo)
    }
    fun escreveAno(){
        println(this.ano)
    }
    // permite que o método tipoTracao possa ser sobre escrito.
    open fun tipoTracao(): String{
        return "4X2"
    }
}
```

Mais sobre modificadores de acesso

```
class Jeep(modelo:String, ano:Int, angulo:Int): Carro5(modelo,ano)   {
    var anguloDeEntrada: Int= angulo
    override fun tipoTracao(): String{
        return "4X4"
    }
   fun escreveVelocidade(){
        println(velocidade)
   }
}
```

Mais sobre modificadores de acesso

```
fun main(){
    var c: Carro5 = Carro5("Onix",2019)
    var j = Jeep("Troller",2013,47)
    println("--------")
    c.escreveModelo()
    c.escreveAno()
    //println(c.velocidade)
    println(c.cor)
    println(c.tipoTracao())
    println("--------")
    j.escreveModelo()
    j.escreveAno()
    println(j.tipoTracao())
    println(j.anguloDeEntrada)
    // println(j.velocidade) >> nao estah mais visivel
}
```

Exercícios

10) Dado o exemplo anterior, contendo o modelo, ano, velocidade e cor do Carro, crie uma nova propriedade chamada valor do tipo Double. O atributo valor deve ser do modificador de acesso private. O conteúdo do atributo deve ser fornecido em funções separadas fora do método construtor. O atributo valor deve ter seu conteúdo enviado por meio de uma função setValor() e exibido na função getValor() a partir da função main.

Herança

- Para utilizar herança basta utilizar : e o nome da superclasse

```
open class Pessoa(nome: String, idade: Int) {
val nome: String
val idade: Int
init {// bloco de inicialização
this.nome = nome
this.idade = idade
}
open fun adicionarDesconto(desconto: Int) {
println("Desconto para a pessoa de $desconto")
}
override fun toString(): String {
return "Pessoa: $nome, idade: $idade"
}
}
class Aluno(nome: String, idade: Int): Pessoa(nome, idade) {
override fun adicionarDesconto(desconto: Int) {
println("Desconto para o aluno de $desconto")
}
}
fun main(args: Array<String>) {
val a1 = Aluno("Fabio", 28)
println(a1)
println("Aluno: ${a1.nome}, idade: ${a1.idade}")
a1.adicionarDesconto(10)
}
```

Herança

Declarando extensões

```
open class Animal() {
open var colour: String = "White"
}

class Dog: Animal() {
override var colour: String = "Black"
fun sound() {
println("Dog makes a sound of woof woof")
}
}

fun main(){
var animal:Dog = Dog()
animal.sound()
}
```

Exercício

11) Crie uma nova classe com base no exemplo anterior chamada Cat e implemente o método definido na classe Animal. Apresente valores em console da mesma forma que ocorre na Classe Dog, para a nova classe implementada.

Classes abstratas

- Métodos abstratos devem ser marcados com abstract e deverão ser substituídos em suas subclasses
- Podem ter métodos não abstratos

Classes abstratas

```
abstract class Employee(val name: String,val experience: Int) {  abstract var
salary: Double
abstract fun dateOfBirth(date:String)
fun employeeDetails() {
println("Name of the employee: $name")
 println("Experience in years: $experience")
   println("Annual Salary: $salary")
}
}
class Engineer(name: String,experience: Int) :  Employee(name,experience) {
override var salary = 500000.00
override fun dateOfBirth(date:String){
println("A data de nascimento é: $date")
}
}
fun main(){
    var eng = Engineer ("Fabio", 12)
    eng.dateOfBirth("19/01/1993")
}
```

Classes Abstratas

Declarando uma classe abstrata

```
abstract class Person(name: String) {
    init {
        println("Meu nome é $name.")
    }
    fun displaySSN(ssn: Int) {
        println("Meu numero é: $ssn.")
    }
    abstract fun displayJob(description: String)
}
class Teacher(name: String): Person(name) {
    override fun displayJob(description: String) {
        println(description)
    }
}
fun main(args: Array<String>) {
    val jack = Teacher("Jack Smith")
    jack.displayJob("Eu sou professor.")
    jack.displaySSN(233333)
}
```

Interfaces

Declarando e implementando uma interface

```
interface Autenticavel {
    val password: String
}

class Usuario : Autenticavel {
    override val password: String = "123"
}

fun main(){
val user = Usuario()
println(user.password)
}
```

Interfaces

Declarando e implementando uma interface

```
interface Usuario {
   fun imprimir()
}
class Leitura:Usuario{
    override fun imprimir(){
       println("Implementando uma função")
    }
}
fun main(){
val user = Leitura()
user.imprimir()
}
```

Exercícios

- 12) Dado a interface do exemplo anterior, realize a implementação de duas classes concretas chamadas Aluno e Professor que devem implementar o método imprimir. Apresente os resultados em console.
- 13) Crie um novo método na interface, chamado estudo. Implemente em todas classes concretas o novo método, contendo uma simples mensagem e apresente os resultados em console.

Polimorfismo Ad Hoc

- Coerção - não permite polimorfismo de coerção para atribuir valores a variáveis e constantes.

```
fun main(args: Array<String>){

val inteiro : Int = 2

val long : Long

long = inteiro }
```

```
error:type mismatch: inferred type is Int b
but long was expected
```

Ad Hoc

- Sobrecarga- possível fazer sobrecarga de operadores

```
data class Point(val x: Int, val y: Int)
operator fun Point.plus(other: Point) = Point(x + other.x, y + other.y)

fun main() {
val p1 = Point(0, 1)
val p2 = Point(1, 2)
println(p1 + p2)
}

//Point(x=1, y=3)
```

Ad Hoc

- Sobrecarga-permite sobrecarga de funções

```
fun main() {
 println(square(34))
 println(square(62.34))
 }

 fun square(num: Int) = num * num
 fun square(num: Double) = num * num
```

Universal - paramétrico

```
class Company<T> (text : T){
var x = text
init{
println(x)
}
}
fun main(args: Array<String>){
var name: Company<String> =
    Company<String>("teste")
var rank: Company<Int> = Company<Int>(12)
 }
```

Universal - inclusão

- Herança- toda classe em Kotlin possui um supertipo chamado Any
- Any tem métodos equals(), hashCode() e toString() definido para toda classe
- Uma classe herda somente de uma classe base

```
open class Food(val price: Double)
class Hamburger(price: Double) : Food(price)
```

Universal - inclusão

```
open class Animal() {
open var colour: String = "White"
}

class Dog: Animal() {
override var colour: String = "Black"
fun sound() {
println("Dog makes a sound of woof woof")
}
}
fun main(){
    var viraLata = Dog()
    viraLata.sound()
}
```

Modificador Data

Modificador Data

```
// Modificador data “cria” automaticamente os métodos “equals”, “hashCode” e “toString”
data class Human(val name: String, val birthYear: Int, val isMarried: Boolean = false)
fun main(){
data class Human(val name: String, val birthYear: Int, val isMarried: Boolean = false)
val human = Human("Felipe", 1987)
val human2 = Human("Felipe", 1987)
val human3 = Human("Fulano", 1999)

println(human) // Imprime Human(name=Felipe, birthYear=1987, isMarried=false)

println(human == human2) // Imprime true
println(human == human3) // Imprime false
}
```

Listas

Listas Imutáveis

```
var list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

println("List elements: $list")
println("Is list empty? ${list.isEmpty()}")
println("List size: ${list.size}")
println("Print element at position 3: ${list[3]}")

list = listOf() // Cria uma lista vazia
```

Listas Mutáveis

```
val list = mutableListOf(1, 2, 3, 4, 5, 6, 7, 8, 9)

list[0] = 2 // [2,2,3,4,5,6,7,8,9]
list.add(10) // [2,2,3,4,5,6,7,8,9,10]
list += 15 // [2,2,3,4,5,6,7,8,9,10,15]
list.removeAt(0) // [2,3,4,5,6,7,8,9,10]
list.remove(9) // [2,3,4,5,6,7,8,10]
list.clear() // []
```

Sets

São similares às Lists, porém não permitem a existência de dois elementos duplicados no seu conjunto.

Sets não permitem o acesso de seus elementos através de índices.

Generics

```
class Company<T> (text : T){  var x = text
init{
println(x)
}
}

fun main(args: Array<String>){
var name: Company<String> =
        Company<String>("Doze")
    var rank: Company<Int> = Company<Int>(12)
}
```

Exceções

- Unchecked (checada em tempo de execução), capturada em tempo de execução
- Derivadas da classe Throwable
- Não precisam ser declaradas explicitamente nas assinaturas de funções, como em Java

Exceções

```
fun main (args: Array<String>){
 try {
var int = 10/0
println(int)
} catch (e:
    ArithmeticException) {
 println(e)
}
    finally {
println("This block always
    executes")
}
}
```

Exceções

```
fun test(password: String) {
if (password.length < 6)
throw ArithmeticException("Senha Fraca")
else
println("Senha Forte")
}

fun main(args: Array<String>) {
 test("abcd")
}
```

Exercício

14) Dado o exemplo anterior, modifique a exceção para que seja considerado como senha fraca um valor menor do que 3 caracteres.

Classe nothing

- Utilizada para representar "um valor que nunca existe"
- Se uma função tem o tipo de retorno Nothing, significa que ela nunca retorna
- Loop infinito, lançar erro

Classe nothing

```
fun main(){
    forever()
}

fun forever():String{
while(true){
Thread.sleep(1)
    println("oi")
}
}
```

Ex.:

```
fun throwException(message: String): Nothing {  throw
        IllegalArgumentException(message)
}
```

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Aplicabilidade	Sim	Sim	Parcial	Parcial
Confiabilidade	Não	Não	Sim	Sim
Aprendizado	Não	Não	Não	Não
Eficiência	Sim	Sim	Parcial	Parcial
Portabilidade	Não	Não	Sim	Sim

- JVM;
- Aplicações Android.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Sim
Custo	Depende	Depende	Depende	Depende

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Método de Projeto	Estruturado	Estruturado & OO	OO	Estruturado, OO & Funcional
Evolutibilidade	Não	Parcial	Sim	Sim
Reusabilidade	Parcial	Sim	Sim	Sim
Integração	Sim	Sim	Parcial	Parcial
Custo	Depende	Depende	Depende	Depende

- 100% interoperável com Java.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Programador	Sistema	Sistema

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Escopo	Sim	Sim	Sim	Sim
Expressões & Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Programador	Programador	Sistema	Sistema

- Coletor de lixo.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Persistência dos Dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	JSON, biblioteca de classes, serialização
Passagem de Parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por cópia.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Persistência dos Dados	Biblioteca de funções	Biblioteca de classes e funções	JDBC, biblioteca de classes, serialização	JSON, biblioteca de classes, serialização
Passagem de Parâmetros	Lista variável e por valor	Lista variável, default, por valor e por referência	Lista variável, por valor e por cópia de referência	Lista variável, default, por valor e por cópia.

- *var* (mutável) & *val* (imutável).

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos, exceto Coerção

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Encapsulamento e Proteção	Parcial	Sim	Sim	Sim
Sistema de Tipos	Não	Parcial	Sim	Sim
Verificação de Tipos	Estática	Estática / Dinâmica	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Coerção e Sobrecarga	Todos	Todos	Todos, exceto Coerção

- Não há a conversão implícita de tipos.

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca de funções)	Sim	Sim

Avaliação

Critérios Gerais	C	C++	Java	Kotlin
Exceções	Não	Parcial	Sim	Parcial
Concorrência	Não (Biblioteca de funções)	Não (Biblioteca de funções)	Sim	Sim

- Coroutines;
- Threads;
- Semáforos;
- Monitores.

Referências

- https://kotlinlang.org/docs/home.html
- http://www.inf.ufes.br/~vitorsouza/archive/2020/wp-content/uploads/teaching-lp-20192-seminario-kotlin.pdf
- https://pt.slideshare.net/felipe_pedroso/aprendendo-kotlin-na-prtica