Udemy

Kotlin for Beginner

Learn Programming with Kotlin

By: Peter Sommerhoff

Jorge Gonçalves – 20091502

Índice

Introdução

Kotlin é uma linguagem de programação multiplataforma desenvolvida pela JetBrains que compila para a JVM (Java Virtual Machine) para correr os programas. Todas as bibliotecas de Java funcionam com Kotlin, uma vez que o principal objetivo da linguagem é a simplificação desta. Também pode ser utilizado para JavaScript.

Para se poder utilizar é necessário o JDK (Java Development Kit) assim como um IDE(Intellij, Android Studio, etc).

Instalação

É necessário o JDK da Oracle para a versão do Sistema Operativo utilizado:

https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Relativamente ao IDE (Integrated Development Environment) o Intellij é desenvolvido pela mesma empresa que desenvolveu a linguagem:

https://www.jetbrains.com/idea/download/#section=windows

REPL – Read Eval-print loop, ferramenta que permite ter acesso a alguns comandos da linguagem.

Operações

Atenção que Kotlin faz divisão inteira, é preciso alterar a forma como são dados os dados para o caso de se querer utilizar floats.

Exemplo no REPL:

7/4 = 1

7.0/4 = 1.75

Getting Started With Kotlin!

Fundamentos básicos

Para declarar variáveis em Kotlin, utilizamos a keyword **var** e não é necessário declara tipo.

```
var string = "Kotlin"
```

Para declarar variáveis constante temos a keyword val(é boa pratica para garantir que valores que não podem ser alterados, não o são:

```
val myName = "Jorge"
```

Arrays são chamas de listas e declaradas como:

```
listOf(1, 2, 3, 4, 5)
```

Para aceder a variáveis numa posição de uma lista:

```
var list = listOf(1, 2, 3, 4, 5)
list[0]
```

Tipos de dados Primitivos & Strings

https://kotlinlang.org/docs/reference/basic-types.html

Strings

String são introduzidas entre “”.

```
val str: String = "Kotlin"
```

Concatenação +.

```
"Kotlin" + " Udemy"
```

Booleans

Well..

```
3<4
true

4<3
false
```

Primitivas Inteiras

Por default Kotlin irá atribuir o tipo integer a variáveis do tipo numérico que não sejam definidas com um tipo de dado.

A definição do tipo de variável numérico pode aumentar a performance do programa no geral por reduzir a memória ocupada no computador.

```
val b: Byte = 127
```

Byte varia entre -128 e 127 (8 bits).

```
val s: Short = 32767
```

Short varia entre -32768 e 32767 (16 bits).

```
val i: Int = 2147483647
```

Int varia entre -2147483648 e 2147483647 (32 bits). Default.

```
val l: Long = 9223372036854775807
```

Long varia entre -9223372036854775808 e 9223372036854775807 (64 bits).

Floating-point e outros

Atenção a utilização de f no final de variáveis decimais para float.

```
val f: Float = 3.73f
```

Float pode ter entre 6 e 7 dígitos significativos (32 bits).

```
 val d: Double = 3.1415
```

Double pode ter entre 16 e 17 dígitos significativos (64 bits). **Default**

```
val c:  Char = 'C'
```

Char permite guardar um único caracter até 16 bits (atenção as aspas simples).

```
true
```

Boolean (8 bits).

```
var f2: Float = 3.4f
var d2: Double = 0.0
d3 = f2.toDouble()
```

Se quisermos juntar valores, por exemplo de uma variável que era float para um double, temos de chamar uma função do Kotlin para evitar erro.

Float não são muito utilizadas porque não são tão precisas como **Doubles**.

Expressions vs Statements

Expressions -> bocados de código que têm uma valor em específico.

Exemplo:

```
3

listOf(1, 2, 3, 4)
```

Statements -> código que não tem uma valor de retorno (apesar de imprimir no REPL, é um statment por ser uma chamada de função).

```
print("Kotlin")
```

Statment Assignment -> well… do lado direito há sempre uma expressão.

```
var a = 47
```

Nullable Variables

https://kotlinlang.org/docs/reference/null-safety.html

Por default Kotlin evita que sejam criadas variáveis nulas mas podemos fazê-los através de:

```
var string: String? = null
```

O ponto de interrogação irá definir um tipo de dado nullable sobre o qual não podem ser chamadas funções, a não ser que quando a variáveis é chamada, seja utilizado uma safety call na forma de ?, isto irá permitir executar uma função, caso a variável tenha um valor diferente de null.

```
string = "Jorge"
string?.length
```

O resultado a da função continua a ser nullable, para o poder guardar é necessário guardar numa variável também nullable.

!! Indica ao Kotlin que temos a certeza que a variável nunca vai ser null e pode chamar a função. Apenas assim é possível causar **null Pointer Exception** em Kotlin.

```
string!!.length
```

Kotlin File

Após ter-se a aplicação configurada, para se criar ficheiros novos, fazemo-lo na pasta **src** – **New** – **Kotlin File/Class**.

Para se criar uma aplicação em Kotlin é necessário a definição de uma main function que será onde Kotlin irá começar a correr a aplicação(ponto de entrada da aplicação):

```
fun main() {
    var str: String = "Hello World!"
    println(str)
}
```

Conditional Statements

Pode ser necessario alterar a pasta onde o ficheiro está por ambiguidade derivado a haver dois **fun main** no mesmo src, para resolver isto, simplesmente queriar um package e mover o ficheiro para lá.

Conditional Statements Using "if"

Exercício: Simular que para registar uma pessoa ela precisa de ter mais de 18 anos.

```
package conditionals

fun main() {
    val age: Int = 28
    if (age < 18) {
        println("You cannot register.")
    } else if (age < 21) {
        println("Well, maybe you can register.")
    } else if (age < 27) {
        println("You should be able to register.")
    } else {
        println("You're good to go!")
    }

    println("The end.")
}
```

Conditional Statements Using "when"

Equivalente ao **switch**, onde o **else** é o equivalente ao **default**.

```
package conditionals

fun main() {
    val mode: Int = 2

    when (mode) {
        1 -> println("THe mode is lazy.")
        2 -> {
            println("The mode is 2.")
            println("So the mode is busy.")
        }
        3 ->println("The mode is super-productive.")
        else -> println("I don't know that mode")
    }
}
```

When to Use "if" vs "when"

A regra comum é que o if é normalmente o utilizado por se poder colocar qualquer tipo de código arbitrário dentro dos parênteses.

O when statement pode ser sempre substituído por um if statement mas em casos em que a execução pode mudar por causa de um determinado conjuntos de valores finitos que uma variável é melhor usar o primeiro(como no caso em cima que temos um número finito de modos).

Conditional Expressions

```
package conditionals

fun main() {
    val mode: Int = 2

    val result = when (mode) {
        1 -> "THe mode is lazy."
        2 -> {
            "The mode is 2."
            "So the mode is busy."
        }
        3 -> "The mode is super-productive."
        else -> "I don't know that mode"
    }
    println(result)

    val x = if (mode < 2) {
        println("Mode is less then 2")
        17
    }else {
        42
    }
    println(x)
}
```

More Advanced "when" Constructs

```
package conditionals

fun main() {
    val x = 10
    when (x) {
        5 -> println("x is 5")
        3*12 -> println("x is 3*12")
        "Hey there".length -> println("x is the length of the string 'Hey
there'")
        in 1..10 -> println("x is between 1 and 10")
        !in 1..9 -> println("x is not between 1 to 9")
    }
    println(x)
}
```

Arrays and Lists

Arrays vs. Lists

Arrays são regra geral melhores para situações em que sabemos previamente quando elementos vamos ter (Países de um continente). Facilmente se faz um loop a um array para fazer algo com os elementos que ele contém. Têm um comprimento fixo. Não tem métodos como add e afins.

List são melhores para situações (rule of thumb) em que queremos adicionar e remover elementos(ver melhor immutable list). Não tem métodos como add e afins.

Arraylist, combina o melhor dos dois mundos. Temos alguns métodos, como **add**.

Arrays e lists são muito similares, mas trabalham-se de formas diferentes.

```
package collections

fun main() {
    val array = arrayOf("Texas", "Iowa", "California")

    val list = listOf("Orange", "Apple", "Bananas")

    val arrayList = arrayListOf("Patrick", "Michael", "Sarah")
    arrayList.add("Sandra")
}
```

Arrays in Kotlin

Podemos criar arrays de um tipo fixo de dados ou de dados misturados.

Podemos aceder ao conteúdo com [ ].

Strings também pode ser tratados como Arrays.

Também é possível concatenar Arrays, mas têm de ser do mesmo tipo.

Também conseguimos saber se um Array contém um determinado elemento.

```
// Arrays
val array = arrayOf("Texas", "Iowa", "California")
val mixed = arrayOf("Kotlin", 17, 3.1, false)
```

```
val numbers = intArrayOf(1, 2, 3, 4, 5)
println(mixed[0])
mixed[2] = 3.1415
println(mixed[2])
val str = "Udemy"
println(str[0])

val states = arrayOf("Nevada", "Florida")
val allStates = array + states
println(allStates)

//comprimento do array
println(allStates.size)

//retorna um boolean
val empty: Boolean = numbers.isEmpty()

if (states.contains("Kentucky")){
    println("It contains Kentucky")
}else{
    println("Is does not")
}
```

Lists in Kotlin

Tem funções muito similares a Arrays.

Funções de adicionar e remover, também retornar um Boolean.

Também é possível com listas criar uma sublista entre determinados índex, sendo o segundo índex indicado excluído(3-1=2 elementos).

```
// Lists
val arrayList = arrayListOf("Patrick", "Michael", "Sarah")
val list = arrayListOf("Peter")
println(arrayList[0])
println(arrayList + list)
println(arrayList.size)
println(arrayList.isEmpty())
println(arrayList.contains("Sarah"))

arrayList.add("Josh")
val changed = arrayList.add(1, "Jack")
println(arrayList)

val removed: Boolean = arrayList.remove("Josephine")
println(arrayList)
println(removed)

val subList = arrayList.subList(1, 3)
println(subList)
```

Lucky Loop

"for" Loops

```
fun main() {
    var sum = 0
    for (i in 1..100) {
        sum = sum + i
    }
    println(sum)

    val list = listOf("Java", "Kotlin", "Python")

    for (element in list) {
        println(element)
    }
    //withIndex pode ser chamado em arrays também
    for ((index, value) in list.withIndex()) {
        println("Element at $index is $value")
    }
}
```

"while" Loops

Muito utilizado na leitura de ficheiros.

```
fun main() {
    var x = 9
    while (x >= 0) {
        println(x)
        x--
    }
    var i = 1
    while ( 1<= 10) {
        println(i)
        i++
    }
    while (user.isOnline()){

    }
}
```

Using "break" and "continue" Statements

"break" salta para o fim do loop.

"continue" só faz skip a um iteração.

Usados moderadamente e ponderadamente pode aumentar a performance.

```
fun main() {
    for (char in "Python") {
        if (char == 'o') {
            break
        }
        print(char)
    }

    println()
    val list = listOf("Book", "Table", "Laptop")
    for (str in list){
        if (!str.contains('o')){
            continue
        }
        //calculations..
        println(str)
    }
}
```

Naming loops

Pode ser útil por causa de nested loops.

```
fun main() {
    for (i in 1..10) {
        for (j in 1..10) {
            if (i - j == 5) {
                break
                //neste caso irá fazer skip ou inner loop
            }
            println("$i - $j")
        }
    }
}
```

```
fun main() {
    outer@ for (i in 1..10) {
        inner@ for (j in 1..10) {
            if (i - j == 5) {
                break@outer
                //neste caso irá fazer skip ou inner loop
            }
            println("$i - $j")
        }
    }
}
```

Functions

Functions

```kotlin
package functions

import java.util.*

// No parameters, no return value
fun helloWorld() {
    println("Hello World!")
}

// "Kotlin" -> "K o t l i n "
// ! parameter, no return value
fun printWithSpaces(text: String){
    for (char in text) {
        print ("$char ")
    }
    println()
}
// em funções que retornam um valor temos que especificar o tipo de dados
// No parameters, returns Data
fun  getCurrentDate(): Date {
    return Date()
}

// 2 parameters, return Int
fun max(a : Int, b: Int): Int {
    if ( a >= b) {
        return a
    }else {
        return b
    }

}

fun main() {
    helloWorld()
    printWithSpaces("Kotlin")
    println(getCurrentDate())
    println(max(17, 42))
}
```

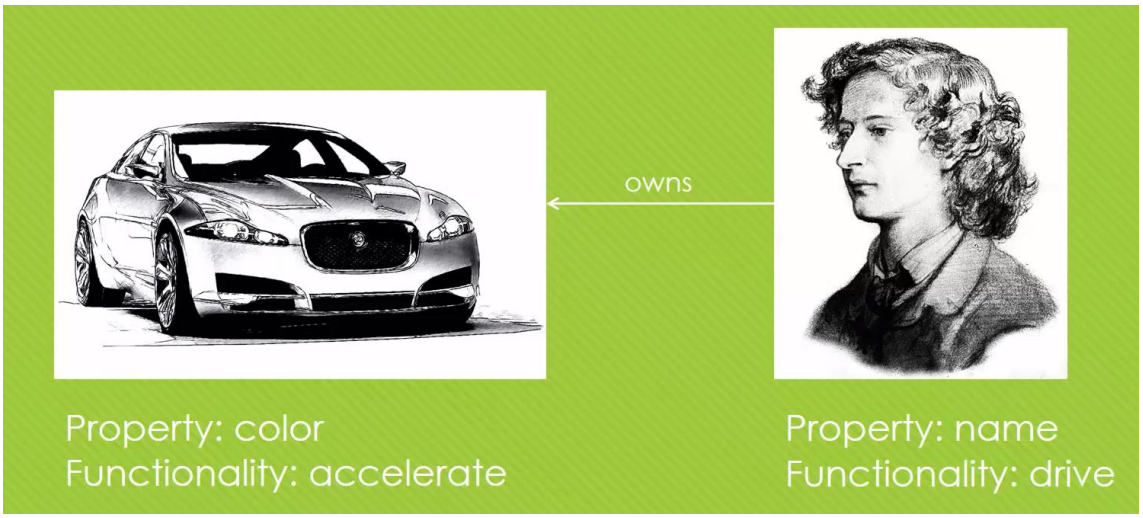
Code Along: Reversing a List

```
package loops

fun main() {
    val numbers = listOf(1, 2, 3, 4, 6, 8, 9)
    println(reverse((numbers)))
    println(reverse2(numbers))
}

fun reverse(list: List<Int>): List<Int> {
    val result = arrayListOf<Int>()
    for (i in 0..list.size - 1) {
        result.add(list.get(list.size - 1 - i))
    }
    return result
}

fun reverse2(list: List<Int>): List<Int> {
    val result = arrayListOf<Int>()
    for (i in list.size -1 downTo 0) {
        result.add(list.get(i))
    }
    return result
}
```

Object-Oriented Programming -- Part I

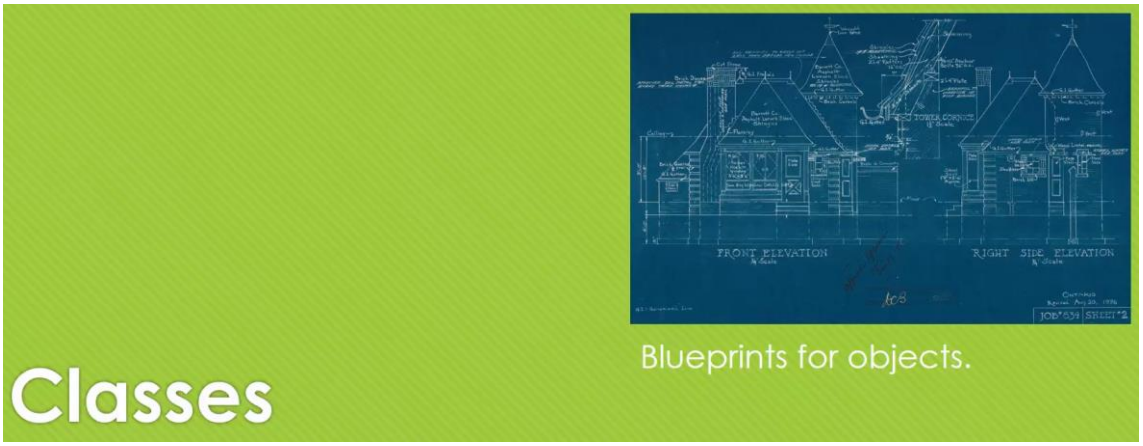
Starting with Object-Orientation

Modelar o mundo real para apenas utilizar(abstração do mundo real). Para modelar apenas temos de dar a uma classe propriedades(cor ou nome) e capacidades(aceleração, conduzir). Também é possível definir relacionamento entre as classes.

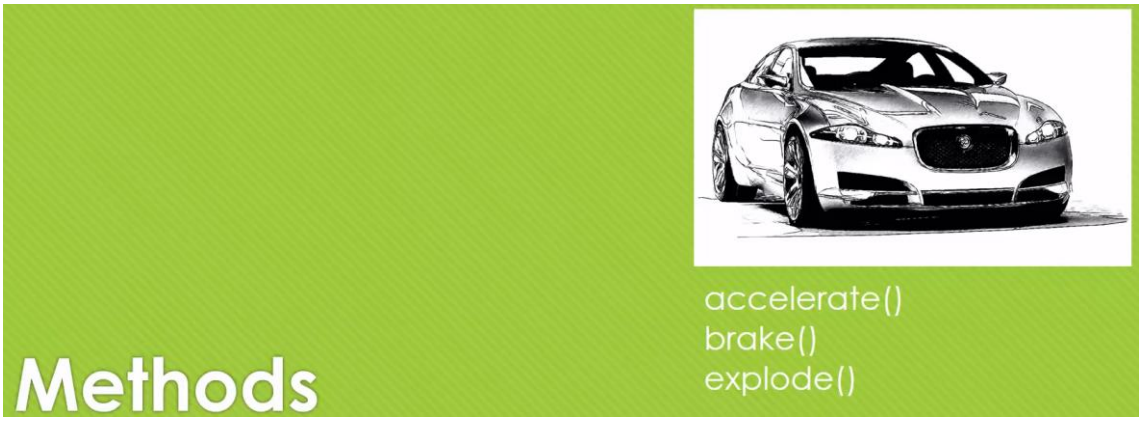


Um carro em específico ou uma pessoa em específico é um objeto. Um objeto pode ter um conjunto de propriedades e capacidades(métodos).

Classes são basicamente a blueprints através das quais podemos criar os nossos objetos. Defini quais os componente que um objeto irá ter mas deixa "espaço" variações em específico, como a cor de um carro ou o nome de uma pessoa.



Com as propriedades definimos a parte dos dados.



Com os métodos definimos as funcionalidades.

Basicamente definem um "contrato" que as classes podem decidir aderir. Todas as classes que forem implementarem esta interface, têm que fornecer um método drive(), como no exemplo acima.

Your First Class

Por convenção as classes, não apenas em Kotlin, são instanciadas com capitalização.

```
class Person {
    var name: String = "Sarah"
    var age: Int = 42
}

fun main() {
    val person = Person()

    println(person.name)
    println(person.age)

    person.name = "Peter"
    println(person.name)
}
```

Methods

```
class Person {
    var name: String = "Sarah"
    var age: Int = 42
    fun speak() {
        println("Hello!")
    }
    fun greet(name: String) {
        println("Hello $name!!")
    }
//    fun getYearOfBirth(): Int {
//        return 2016 - age
//    }
    fun getYearOfBirth() = 2016 - age
}

fun main() {
    val person = Person()

    person.speak()
    person.greet("world")
    println(person.getYearOfBirth())

    println(person.name)
    println(person.age)
}
```

Constructors

```
// no Kotlin não precisamos de inicializar as propriedades como no Java
// para evitar o boilerplate code, inicializa-se diretamente na declaração da
class
class Person(name: String, age: Int) {
    val name: String
    var age: Int

    //apenas será chamado quando o objeto for criado
    init {
        this.name = name
```

```
        this.age = age
        println("Object was created")
    }

    fun speak() {
        println("Hello!")
    }
    //neste caso irá se referir ao name local(refere-se sempre a scope mais
próxima
    fun greet(name: String) {
        println("Hello $name!!")
    }

    fun getYearOfBirth() = 2019 - age
}

fun main() {
    val person = Person("Jack", 17)

    person.speak()
    person.greet("world")
    println(person.getYearOfBirth())

    println(person.name)
    println(person.age)
}
```

```
class Person(val name: String, var age: Int) {

    //apenas será chamado quando o objeto for criado
    init {
        println("Object was created")
    }

    fun speak() {
        println("Hello!")
    }
    //neste caso irá se referir ao name local(refere-se sempre a scope mais
próxima
    fun greet(name: String) {
        println("Hello $name!!")
    }

    fun getYearOfBirth() = 2019 - age
}
```

Named Parameters & Default Values

```
// atenção ao valor default em price
class House(var height: Double, val color: String, val price: Int = 50000) {
    fun print() {
        println("House [height=$height, color=$color, price=$price]")
    }
}

fun main() {
    val house = House(4.5, "Blue", 150000)
    val redHouse = House(color = "Red", price = 200000, height = 5.0)
    val yellowHouse = House(color = "Yellow", height = 2.5)

    house.print()
    redHouse.print()
    yellowHouse.print()
}
```

Open Classes and Inheritance

Uma classe ao herdar de outra, irá herdar todas as propriedades e métodos da parent class neste caso, assim não temos de especificar outra vez os métodos e propriedades herdado pelo child class.

Para tal, temos de especificar que uma classe irá ser herdada o pode ser, com a keyword **open**(também é necessário passar o construtor)(ver **abstract**). Os parâmetros que uma determinada classe tenha(que são usados no construtor) têm de ser também open para poderes ser subscritos(**override**). Uma propriedade que seja de valor fixo no construtor não será necessário declarar como open mas o valor deverá de ser atribuído na "extensão da classe".

Isto permite-nos escrever menos código, evitando duplicação, só sendo necessário escrever o código que define a diferença entra cada classe.

```
open class Person(open val name: String, open var age: Int) {

    //apenas será chamado quando o objeto for criado
    init {
        println("Object was created")
    }

    fun speak() {
        println("Hello!")
    }

    //neste caso irá se referir ao name local(refere-se sempre a scope mais
próxima
    fun greet(name: String) {
        println("Hello $name!!")
    }

    fun getYearOfBirth() = 2019 - age
}

class Student(override val name: String, override var age: Int, val
studentID: Long) : Person(name, age) {
    fun isIntelligent() = true
}

class Employee(override val name: String, override var age: Int) :
Person(name, age) {
    fun receivePayment() {
        println("Received payment.")
    }
}

fun main() {
    val student = Student("John", 25, 3647284)
    student.speak()
    println(student.isIntelligent())

    val employee = Employee("Mary", 32)
    println(employee.getYearOfBirth())
    employee.receivePayment()
}
```

Abstract Classes

Para evitar que uma classe que define propriedades e métodos comuns entre outras classes tenha a possibilidade de criar também objetos, em vez de definirmos como open, definimos como **abstract** o que irá fazer que não seja possível criar objetos a partir desta, uma classes abstact é implicitamente **open**.

Muitas vezes não iremos utilizar todos os métodos de uma parent class, mas as childs class's deveram ter um método com determinado nome(encapsular semelhanças), para tal também podemos declarar um método como **abstract**. Isto irá fazer com que o método deva ser definido em todas as classes que o herdarem mas apenas nelas implementado, sendo nelas feito a implementação com as devidas diferenças.

```
abstract class Person(open val name: String, open var age: Int) {

    //apenas será chamado quando o objeto for criado
    init {
        println("Object was created")
    }

    abstract fun speak()

    //neste caso irá se referir ao name local(refere-se sempre a scope mais
próxima
    fun greet(name: String) {
        println("Hello $name!!")
    }

    fun getYearOfBirth() = 2019 - age
}

class Student(override val name: String, override var age: Int, val
studentID: Long) : Person(name, age) {

    fun isIntelligent() = true
    override fun speak() {
        println("Hi there, I'm a student!")
    }
}

class Employee(override val name: String, override var age: Int) :
Person(name, age) {
    fun receivePayment() {
        println("Received payment.")
    }

    override fun speak() {
        println("Hi I'm an employee!")
    }
}
```

Classes astract ou open, não implica que apenas se possa implementar propriedades e métodos com este atributo, também é possível definir propriedades e métodos com "normais".

Open vs. Abstract

Se não declarar-mos como nem **open** ou **abstract**, a Kotlin irá reconhecer a class como **final** levando a que não seja possível herdar desta.

A **open** keyword permite criar child class's da parent class assim como instanciar objetos da parent class. Permite a herança.

A **abstract** keyword apenas permite criar child class's da parente, mas não permite instanciar objetos da parent class por estas não ser concreta o suficiente (pode conter métodos que estão definidos, com nome, mas apenas são implementados na child class). Requer que seja herdado.

Uma class **abstract** pode ter um método **open** o que permite que esta seja alterada. O contrário já não é possível (numa **class open** declarar um método **abstract**).

Interfaces

Interfaces definem um contrato para as class's (exemplo da Interface drivable define que cada class que implemente está Interface deve de ter um método chamado drive()). Isto permite mudar entre diversas class's desde que estas tenham um método para a Interface.

Na sua forma mais simples, uma Interface define diferentes métodos que uma class tem de oferecer quando a implementar (cada class que implementar esta Interface deverá ter o/s métodos necessários).

Interfaces são **abstract** assim como os métodos que esta podem requerer que sejam implementados (override). Uma interface também pode ter métodos **final**(apesar de não ser recomendado) (foi implementado em Java 8 para estender Interfaces que já existe há mais de 10 anos sem crashar código que depende disto) ou até mesmo propriedades, mas esta última não pode ter nenhum estado (valor).

Uma Interface também pode ser um tipo para a criação de um objeto, o que irá fazer que apenas se tinha acesso aos métodos disponíveis nesta Interface, mesmo que Car tenho outras Interfaces ou métodos.

```
val car: Driveable = Car("blue")
```

Dependendo da class em concreto que estamos a instanciar, os objetos irão executar diferentes tarefas (polimorfismo).

Uma class pode implementar múltiplas Interfaces.

Interfaces têm o nível mais alto de abstração.

```
interface Driveable {
    fun drive()

}

interface Buildable {
    val timeRequired: Int
    fun build()
}

class Car(val color: String) : Driveable, Buildable {
    override val timeRequired: Int = 120
```

```
    override fun drive() {
        println("Driving car...")
    }

    override fun build() {

        println("Built a shiny car.")
    }
}

class MotorCycle(val color: String) : Driveable {
    override fun drive() {
        println("Driving motorcycle...")
    }
}

fun main() {
    val car: Driveable = Car("blue")
    car.drive()
    val motorCycle: Driveable = MotorCycle("red")
    motorCycle.drive()
}
```

Object-Oriented Programming -- Part II

Overriding Rules

Em Kotlin, uma child class pode fazer override de uma class do parent, mesmo que o parent já tenha feito.

```
abstract class Course(val topic: String, val price: Double) {
    open fun learn() {
        println("Learning a $topic course.")
    }
}

open class KotlinCourse() : Course("Kotling", 999.99) {
    override fun learn() {
        println("I'm one of the first people to learn Kotlin!")
    }
}

class SpecialKotlinCall() :KotlinCourse() {
    override fun learn() {
        println("Learning special Kotlin course")
    }
}
```

Se quisermos que não seja possível fazer override, no parent que faz o override, apenas temos de declarar final.

```
open class KotlinCourse() : Course("Kotling", 999.99) {
    final override fun learn() {
        println("I'm one of the first people to learn Kotlin!")
    }
}
```

Quando se herda propriedades ou métodos com o mesmo nome (no caso de haver múltiplas class que estendemos ou Interfaces com o mesmo nome para métodos) é necessário fazer override. Se quisermos fazer override mas implementar o método de um dos parent, podemos usar a keyword **super** (refere-se à parent class) especificando qual das class's queremos que forneça o método.

```
abstract class Course(val topic: String, val price: Double) {
    open fun learn() {
        println("Learning a $topic course.")
    }
}
interface Learnable{
    fun learn() {
        println("Learning...")
    }
}

open class KotlinCourse() : Course("Kotling", 999.99), Learnable {
    final override fun learn() {
        super<Course>.learn()
        println("I'm one of the first people to learn Kotlin!")
    }
}
```

Data Classes

Preferível para situações onde apenas queremos guardar dados.

Já implementam muitos métodos que facilitam o tratamento de dados. Também pode ser implementado métodos.

Estas data class's implementam automaticamente um método toString que nos facilita a apresentação dos dados sem ter de fazer override ao toString.

Também é muito mais simples fazer a comparação de objetos de data class's.

```
val dataBook = DataBook("Super Book", "John Doe", 2017, 99.99)
val dataBook2 = DataBook("Super Book", "John Doe", 2017, 99.99)
println(dataBook.equals(dataBook2))
```

Copiar também é direto, podem ser alterado o valor de qualquer dos campos.

```
val dataBook3 = dataBook.copy()
```

Também é possível desconstruir uma data class.

```
val (title, author, year, price) = dataBook
```

Também é mais simples a criação de hashsets (onde não é suposto haver conjuntos de dados iguais). Os conjuntos duplicados, apenas iram ter um representante.

```
class Book(val title: String, val author: String, val publicationYear: Int,
var price: Double) {
    override fun toString(): String {
        return "Book[title=$title, author=$author, publicationYear=
$publicationYear, price=$price]"
    }
}


data class DataBook(val title: String, val author: String, val
publicationYear: Int, var price: Double) {

}

fun main() {
    val book = Book("Super Book", "John Doe", 2017, 99.99)
    val book2 = Book("Super Book", "John Doe", 2017, 99.99)
    val dataBook = DataBook("Super Book", "John Doe", 2017, 99.99)
    val dataBook2 = DataBook("Super Book", "John Doe", 2017, 99.99)
    val dataBook3 = dataBook.copy(price = 89.99)

    val (title, author, year, price) = dataBook

    val set = hashSetOf(dataBook, dataBook2, dataBook3)
    val set2 = hashSetOf(book, book2)

    println(set)
    println(set2)

    println(book)
    println(dataBook)
    println(dataBook3)
    println(title)

    println(dataBook.equals(dataBook2))
}
```

Objects / Singletons

Singletons são class' sobre os quais apenas pode haver um único objeto instanciado a um determinado momento. É definido pela keyword **object**.

```
// object declaration
object Cache {
    val name = "HyperCache"

    fun retrieveData(): Int {
        return 101010
    }
}
```

Basic Enums

Enums = Enumerations.

Não deve de ser possível introduzir valores que não sejam espectáveis (validação). Enums permitem definir quais são os inputs validos (listas de inputs) para um determinado parâmetro, por exemplo: definir que um parâmetro de uma class é do tipo direcao e apenas pode ser Norte, Sul, Este e Oeste.

```
enum class Color {
    RED, GREEN, BLUE
}
```

```
class Car(val color: Color) : Driveable, Buildable {
    override val timeRequired: Int = 120

    override fun drive() {
        println("Driving car...")
    }

    override fun build() {

        println("Built a shiny car.")
    }
}
fun main() {
    val car = Car(Color.GREEN)
    println(car.color)
}
```

Packages

São uma forma básica de categorizar os source files e por conseguinte contêm as class's em unidades logicas.

Por convenção os nomes do packages são todos em lowercase.

Para os utilizar basta apena fazer o import do package. Em alternativa pode-se utilizar o nome qualificado do package o que não é muito prática uma vez que tem de se colocar o prefixo onde se utilizar o package. Alguns packages já são importados por default.

Os packages também podem ser nested.

Imports

É possível importar uma package em específico ou todos de um determinado grupo.

```
import java.math.BigInteger
import java.util.*
```

Também se pode chamar funções que estão dentro objetos(singletons), uma vez que estes não recebem parametros.

```
object CarFactory {
    fun produceCar(){
        println("Porduced a car.")
    }
}
```

Não é possível importar class's em concreto porque pode não estar disponível todas as variáveis necessários para executar uma função.

É possível importar enums também.

```
import oo.Color.BLUE
```

Utilizando o IntelliJ, este era otimizar os imports e retirar ou acrescentar o código necessário para o import.

Nomes de packages devem ser sempre únicos. Para tal, regra geral, coloca-se o nome do package em ordem reversa: pt.jasmimdesing.functions

Binary & Hexadecimal Numbers

Hexadecimal Numbers & The Color Enum

Os números (8 se for hexadecimal, 10 para binário) são elevados à potência da sua posição e multiplicados pelo quantidade de vezes que é superior a 1.

$$2137 = 2 \times 10^3 + 1 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

Hexadecimal

$$0..9, A(10), B(11), C(12), D(13), E(14), F(15)$$

$$0x1F \rightarrow 1 \times 16^1 + F \times 16^0 = 31$$

$$0xFF \rightarrow F \times 16^1 + F \times 16^0 = 255$$

Em Kotlin, os números hexadecimais, têm como prefixo 0x.

$$0x10 \rightarrow 16\ em\ decimal$$

```
//cores em hexadecimal são representadas e grupos de dois
println(0x000000) // black
println(0xFFFFFF) // white
println(0xFF0000) // red
println(0x00FF00) // green
println(0x0000FF) // blue
println(0x000088) // dark blue
```

Binary Numbers & The Color Enum

Similar ao anterior, apenas temos dois números que podem surgir (0 e 1), multiplicamos número de ocorrências pelo número de hipóteses (2) elevado à potencia na sua posição.

Em Kotlin o prefixo para binário é 0b.

$$0b10 \rightarrow 2\ em\ decimal$$

$$0b1010 \rightarrow 1 \times 2^3 + 1 \times 2^1 = 10$$

```
    // binary numbers
    // 0..1 -> 0b10
    // podem ser utilizados 0 para ter os binarios com o mesmo
comprimento(byte)
    println(0b00001010) // -> 10
    println(0b11111111) // -> 255
    println(0b11111111_00000000_00000000) // red

    // 1 and 1 == 1
    // 1 and 0 == 0
    // 0 and 1 == 0
    // 0 and 0 == 0
    //      1101
    //  and 1011
    // ------------
    //  =   1001
    println(0b1101 and 0b1011)
}
```

```
RED(0xFF0000), GREEN(0x00FF00), BLUE(0x0000FF), YELLOW(0xFFFF00);

fun containsRed(): Boolean {
    return this.rgb and 0xFF0000 != 0
}
// 1 and 1 == 1
// 1 and 0 == 0
// 0 and 1 == 0
// 0 and 0 == 0
//      1101
//  and 1011
// ------------
//  =   1001
println(0b1101 and 0b1011) // -> 9

println(Color.RED.containsRed())
println(Color.GREEN.containsRed())
println(Color.BLUE.containsRed())
println(Color.YELLOW.containsRed())
```

Bitwise Operators

```
// bitwise operators

// 1 or 1 == 1
// 1 or 0 == 1
// 0 or 1 == 1
// 0 or 0 == 0
//      1101
// or   1000
// ------------
//  =   1101
println(0b1101 or 0b1000) // 13

// 1 xor 1 == 0
// 1 xor 0 == 1
// 0 xor 1 == 1
// 0 xor 0 == 0
//      1101
// xor  1000
// ------------
//  =   0101
println(0b1101 xor 0b1000) // 5

// inverse
// 0b10011.inv() == 0b...01100
// 0x0000001F == 0b00..0011111
println(0b1011.inv() and 0x0000001F) // 0b01100
//relacionado com o facto de ser do tipo Int e ter 32bits
```

Object-Oriented Programming -- Part III

The Principle of Information Hiding

Quando usamos uma class importada, ou importamos a nossa, não deve de ser possível ver ou ter acesso a toda a lógica dentro dessa class. Apenas deve de ser possível trabalhar com a class e não alterar os seus métodos ou propriedades. **Private** vs **Public**.

Restringir o acesso ao exterior ao conteúdo interno da class.

Properties II: Getters and Setters

```
class Animal {
    // isto define um getter e um setter
    // getter -> retorna o valor de uma variável
    // setter -> define um novo valor à variável
    var age: Int = 0
    // var's têm Getter's e Setter's
        get() = field
        // é vantagoso por permitir criar restrições para a definição dos
dados na raiz do objeto
        set(value) {
            if (value >= 0) {
                field = value
            }
        }
}

fun main() {
    val animal = Animal()
    // val's têm apenas Getter's porque não se pode redifinir a variável
    // não acede diretamente à propriedade
    // internamente chama o setter do Kotlin
    animal.age = 8
    // internamente chama o getter do Kotlin
    println(animal.age)
}
```

Visibilities

Private -> é o modificador mais restrito de privacidade só será visível na sua scope(class). Boa pratica utilizar por default.

```
private var age: Int = 0
```

Protected -> Pode ser acedido dentro da própria class e subclasses da class onde é declarado (apenas faz sentido dentro de class's **open**).

```
open class Animal {
    private var age: Int = 0
    protected  var name = "Sam"
}

class Vertebrate : Animal() {
    fun introduceYourself () {
        println(this.name)
    }
}
```

Internal -> podem ser acedidos dentro de dos mesmo módulos ("projeto").

```
internal val isDangerous = true
```

public -> qualquer class pode aceder ao método ou propriedade do exterior. Não precisa de ser uma subclasse nem no mesmo modulo. Regra geral é utilizado para bem definir um contrato de uma Interface. É preciso ter cuidado se uma função tiver parametros pois um utilizador poderá introduzir valores não expectáveis (fazer validação de dados).

```
public fun isOld(): Boolean {
    return age > 12
}
```

Generics

```
// DRY = don't repeat yourself
```

Uma boa pratica é utilizar loops.

Situações onde temos propriedades e métodos comuns entre class's, pode ser boa pratica definir uma class com esses parâmetros que irá ser herdada.

```
class Stack<E> {
    fun push () {
    }
    fun pop() {
    }
}
```

<E> permite-nos declarar que iremos criar uma class que não tem um tipo de dados definido, mas sim é genérica, podendo ser criados objetos desta class de qualquer tipo. Útil quando temos lógica que não depende do tipo de dados (exemplo da Stack, push, pop).

A Generic Stack

Muito importante para estruturas de dados que podem variar de tipo.

```
// DRY = don't repeat yourself
// vararg permite adicionar quanto elementos de um tipo quisermos.
// não é necessario criar mais uma lista para passar mais que um elemento.
class Stack<E>(vararg val items: E) {

    val elements = items.toMutableList()

    fun push(element: E) {
        elements.add(element)
    }

    fun pop(): E? {
        if (!isEmpty()) {
            return elements.removeAt(elements.size - 1)
        }
        return null
    }

    fun isEmpty(): Boolean {
        return elements.isEmpty()
    }
}

fun main() {
    val stack = Stack(3, 5, 2, 8)
    stack.push(11)
    println(stack.pop())
    println(stack.pop())
    println(stack.pop())
    println(stack.pop())
    println(stack.pop())
}
```

Generic Functions

Similar ao método **arrayListOf** que irá criar uma lista de arrays do tipo dos elementos introduzidos.

```
// seja qual for o tipo de T a Stack criada será desse tipo
fun <T> stackOf(vararg elements: T): Stack<T> {
    // spread operatior * para converter elements (que esta em array) para
elementos
    return Stack(*elements)
}

fun main() {
    val stack = Stack(3, 5, 2, 8)
    stack.push(11)

    val stack2 = stackOf("Hi", "Hey", "Hello")
    for (i in 1..3) {
        println(stack2.pop())
    }
}
```

I/O -- Input and Output

Introduction to I/O

Input -> dar entrada de dados num Software/Hardware. Estas operações podem ser simplesmente pedir dados a um utilizador ou ler dados de uma base de dados.

Output -> dar saída de determinados dados num Software/Hardware. Estas operações estão relacionadas com dar informação ao utilizador ou outros dispositivos.

A Little Console Game

CTRL + B permite ir diretamente para a class importada.

O método readline() permite ficar à escuta de informação por parte do utilizador.

```
        fun main() {
    val number = Random().nextInt(100) + 1
    var input: String?
    var guess = -1

    while (guess != number){
        print("Guess a number between 1 and 100: ")
        input = readLine()

        if (input != null) {
            // can throw NumberFormatException
            guess = input.toInt()
        }
        if (guess < number) {
            println("Too low")
        }else if ( guess > number) {
            println("Too high")
        }else{
            println("You guessed!")
        }
    }
}
```

Code Along: Hangman Game - Part I

```
package io

fun main() {
    println("Enter the word to guess: ")
    val word: String? = readLine()

    if (word == null) {
        println("No word given, game ended.")
        return
    }
    for (i in 1..100) {
        println()
    }
    // normalização
    // ao converter para HashSet, são retiradas os duplicadas da array
    // Tree
    // tree
    // ['t','r','e','e']
    // {'t','r','e'}
    val letter = word.toLowerCase().toCharArray().toHashSet()
    // mutable significa que se pode acrescentar novos elementos ao set
    val correctGuesses = mutableSetOf<Char>()
    var fails = 0

    while (letter != correctGuesses) {
        printExploredWord(word, correctGuesses)
        println("#Wrong guesses: $fails")

        print("Guess a letter: ")
        val input = readLine()

        if (input == null) {
            continue
        } else if ( input.length != 1) {
            println("Please enter one letter")
            continue
        }
        if (word.toLowerCase().contains(input.toLowerCase())){
            correctGuesses.add(input[0].toLowerCase())
        } else{
            fails++
        }

    }
    printExploredWord(word, correctGuesses)
    println("\n#Wrong guesses: $fails\n")
    println("Well done!")
}
fun printExploredWord(word: String, correctGuesses: Set<Char>) {
    for (character in word.toLowerCase()) {
        if (correctGuesses.contains(character)) {
            print("$character ")
        } else {
            print("_ ")
        }
    }
}
```

Reading From a File

Deve de ser evitado se possível porque requer muito poder computacional e pode levar algum tempo.

Assim como em outras linguagens, o Kotlin, para acesso a feito, considera a pasta do projeto, como src.

Atenção ao importe de java.io.File

```
import java.io.File

fun main() {
    var lineNumber = 0
    File("../inputfile.txt").forEachLine {
        // implicitamente, Kotlin irá chamar a cada linha it
        ++lineNumber
        println("#$lineNumber: $it")
    }
}
```

Challenge Preparation: Maps

```
fun main() {
    // Pair cria objetos
    val namesToAges = mapOf(Pair("Peter", 24), Pair("Roger", 42))
    // Outro tipo de notação para criação de mapas
    // to é infix (também existe o método .to("parametro"))
    val namesToAges2 = mapOf(
        "Peter" to 24,
        "Roger" to 42
    )
    println(namesToAges == namesToAges2)

    println(namesToAges.keys)
    println(namesToAges.values)
    println(namesToAges.entries)

    val countrytoInhabitants = mutableMapOf(
        "Germany" to 80_000_000,
        "USA" to 300_000_000
    )
    // no caso de ser igual(a key) .put subescreve
    countrytoInhabitants.put("Australia", 23_000_000)
    // apenas irá acrescentar se não existir
    countrytoInhabitants.putIfAbsent("USA", 320_000_000)
    println(countrytoInhabitants)
    println(countrytoInhabitants.contains("Australia"))
    println(countrytoInhabitants.containsKey("France"))
    println(countrytoInhabitants.containsValue(20_000_000))
    // .get, retorna o valor guardado na chave
    println(countrytoInhabitants.get("Germany"))
    // na ausencia, podemos definir um valor default
    println(countrytoInhabitants.getOrDefault("France", 0))

    namesToAges.entries.forEach {
        println("${it.key} is ${it.value}")
    }
}
```

Challenge Solution: Find the Most Frequent IP Address

```
import java.io.File

fun main() {
    val ipToCount = mutableMapOf<String, Int>()
    File("../ips.txt").forEachLine {
        val previous = ipToCount.getOrDefault(it, 0)
        ipToCount.put(it, previous + 1)
//        if (ipToCount.contains(it)) {
//            val previous = ipToCount.get(it)!!
//            ipToCount.put(it, previous + 1)
//        } else {
//            ipToCount.put(it, 1)
//        }
    }
    // permite procurar qual é o número maximo de ocorrencias
    // acedemos a todas as entradas(entries) e maxBy retorna apenas o que
ocorre mais vezes
    val (maxIp, maxCount) = ipToCount.entries.maxBy { it.value }!!
//    var maxIp = ipToCount.keys.first()
//    var maxCount = 0
//    //podemos deconstruir no key value pair
//    for ((ip, count) in ipToCount.entries) {
//        if (count > maxCount) {
//            maxCount = count
//            maxIp = ip
//        }
//    }
    println("Most frequent IP adress is $maxIp, which occurred $maxCount
times.")
}
```