KOTLIN

A picture containing screenshot, diagram

Description automatically generated

// Variables

var - reasignabila

val - nu e reasignabila

// Data Types - Primitive data types

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

String nu e primitive, are special language support

// assign a value  from if condition to a var

Ultima expresie din if condition se va asigna catre variabila

// when statement - cu argument si fara argument. Branch-urile care au conditii comune, se vor executa in ordinea declarată

// null — very hard to get - toate variabilele sunt non-nullable la început

val text: String = null; // nu merge

val text: String? = null; // ? mark at the end of the type

println(text); // null — nu primim exceptia

println(text.length) // eroare la compile time, trebuie facut safe check: text?.length sau text!!.length care daca e null iti arunca exceptia, daca e valoare o printează

// functions, short version of function with just like of code with =, functions overloading, default values, call functions with the order of params different, varargs

// break and continue - break outer - ex while in while, in al doilea while daca conditia e indeplinita dam break la tot

// arrays - muttable lists and immutable lists

listOf, mutableListOf, setOf, mutableSetOf, mapOf, mutableMapOf

// mapping (.map, .mapKeys, .mapValues, .mapIndexedNotNull)

// group (.group, .groupBy)

// zipping (.zip, a zip b, .unzip)

// association (.associateWith, .associateBy)

// flatten

// string representaton (.joinToString, .joinTo)

// filtering (.filter, .filterIndexed, .filterNot, .filterIsIstance, .partition)

// .any, .none, .all

// slice, take, takeLast, drop, dropLast, takeWhile, takeLastWhile, dropWhile, dropLastWhile, chunked

// .elementAt, .first, .last, .random, .isEmpty

// .sum, .count, .average, .maxOrNull, .minOrNull, .sumOf

// + and - to add or remove in a mutableListOf

// OOP

* in classes properties can not be uninitialised
* metodele sunt de fapt funcții
* primary constructor - also: putem declara properties direct in constructor
* secondary constructor - automat va apela primary, dar se poate folosi ca un class overloading constructor
* initializer blocks - se va executa cand o clasa va fi creată si e buna cand avem mai mult cod ca sa executam pentru a salva intr-o proprietate
* getters and setters - override - ca sa le facem override trebuie sa definim pe acestea imediat dupa ce definim proprietatea in clasa
* singletone - object sau cu companion object
* lateinit
* companion object - Ex: Calculator.sum()
* lazy initialization
* enum classes - enum
* inner classes- ex: ListView si ListViewItem
* sealed classes
* abstract classes - can assign some vars, in interfaces can not. Also abstarct fun must be empty
* data class
* Interfaces - particular to each class. Ex: button on click => onclick e particular pentru fiecare button
* object expression - onclick defined based by interface pentru fiecare la inițializarea
* Delegation

// Comparable - pentru clase (override fun compareTo). Ex: sa poti sorta o lista de acelasi tip de clase dupa un anume criteriu, suprascrie comparable global al clasei

// Comparator  - override fun compare - sa poti face sa compari dupa anumite chestii, nu ceva global al clasei

compareBy - creaza un comperator dar cu o lambda expression

// sealed class - compile time sunt cunoscute colectiile de date, nu la run time

// inheritance - clasele sunt closed by default(final), putem sa le extidem doar daca sunt declarate open sau abstract

// serializare si deserializare - reflection - transportare de date

// Coroutines Dispatchers

Main - UI operations, change UI

Default - by default - complex and long running calculations which blocks ui thread

IO - data operations, writing to db, writing to files

Unconfined - starts a coroutine in the caller thread, but only the first suspension point. After suspension it resumes the coroutine in the thread that is fully determined by the suspending function that was invoked. Does not matter where executing is.

// launch { // context of the parent }

// Functions in Kotlin always return a value. If you skip the return value, the function is indeed returning Unit. Unit is similar to void in Java, though this is, in fact, an object

// Unless a visibility modifier is applied, classes, functions or properties are public by default

// Characters (Char) cannot directly be used as numbers. We can, however, convert them to a number when we need it: 1 val c: Char = 'c' 2 val i: Int = c.toInt()

// Kotlin

val bitwiseOr = FLAG1 or FLAG2

val bitwiseAnd = FLAG1 and FLAG2

// data class

data class - storage mai mult

// Data Class Copy

val f1 = Forecast(Date(), 27.5f, "Shiny day”)

val f2 = f1.copy(temperature = 30f)

// Declaration destructuring

val f1 = Forecast(Date(), 27.5f, "Shiny day”)

val (date, temperature, details) = f1

// Inline functions

Inline functions are a bit different from regular functions. An inline function is substituted by its code during compilation, instead of doing the real call to a function. That reduces memory allocations and runtime overhead in some situations. For instance, if we have a function as an argument, a regular function internally creates an object that contains that function. On the other hand, inline functions substitute the code of the function in the place where it is called, so it does not require an internal object for that.

inline function - in momentul incare este apelata, nu se mai fac jump-uri sau call-uri catre ea. nu se mai aloca nimic, compilatorul copiaza codul acolo unde se apelează

// scope coroutines - musai sa alba - lifecycle care are si isActive

// A lambda expression is a simple way to define an anonymous function.

Lambdas can substitute any function that receives an interface with a single function. So we can rewrite setOnclickListener with a lambda function.

view.setOnClickListener(object : OnClickListener {

override fun onClick(v: View) {

toast("Click”)

}

})

->

view.setOnClickListener({ view -> toast("Click")})

view is not used so ->

view.setOnClickListener({ toast("Click") }) -> 

view.setOnClickListener() { toast("Click") } -> 

view.setOnClickListener { toast("Click") }

// Kotlin

getters and setters sunt implicite pentru proprietăți

// Coroutines

* Suspend
* Contexts
* runBlocking
* jobs, waiting, cancelation
* async and await

// AsyncTask - To do some work on the background and then update elements in your UI with the result of that background work. . You have to make sure to properly handle cancelling a running AsyncTask when your Activity or Fragment shuts down, since you could otherwise get a NullPointerException when the AsyncTask tries to update UI elements that don't exist anymore.

// Broadcast vs ConflatedBroadcast - se folosesc daca ai mai multi consumeri ca sa nu intervina race condition

Broadcast emite si old items

ConflatedBroadcast emite doar recentul

// Observable

This delegate helps us detect changes on any property we need to observe. It executes the declared lambda expression every time the set function is called. So after the new value is assigned, we receive the delegated property, the old value, and the new one.

class ViewModel(val db: MyDatabase) {

var myProperty by Delegates.observable("") { \_, \_, new ->   db.saveChanges(this, new) }  }

// Vetoable

This delegate is similar to observable, with the difference that it runs before the value is assigned, and lets you decide whether the value must be saved or not. It can be used to check some conditions before saving a value.

// Generics - reified types

Generic programming consists of writing algorithms without the need of specifying the exact type the code is going to use.

class TypedClass<T>(parameter: T) {

val value: T = parameter

}

ex:

val t1 = TypedClass("Hello World!”)

val t2 = TypedClass(25)

// Variance

clasa user

clasa male: user

clasa female: user

val a: listOf<male> = …

val b: listOf<female> = …

val c: listOf<user> = a;

// Contravariance

clasa user

clasa male: user

clasa female: user

class userCompareTo

class maleCompareTo

Nu voi putea face: val userComp: Comparable<User> = maleComparator;

Dar voi putea face: val maleComp: Comparable<Male> = userComparator;

Putem folosi user comparator fiindca e super type ca sa comparam maleComparator, fiindca e un subtype.

// object

object MyClass {} // folosit pentru singletone, returneaza aceeași instanță.

// Dependency injection

Decat sa  faci initializari de alte clase in clasa unde ai nevoie, mai bine le dam c parametru. Gen clasa theatre are nevoie de clasa film, clasa proiector, clasa sala. Putem da parametrii astia in constructor ca sa fie si dinamici in functie de ce proiector e nevoie si etc. Asa este si testabila.

Sa faci fiecare modul independent de altul.

Cand o clasa foloseste alta care se initializeaza in clasa respectiva si nu se poate testa functia folosita. Se da clasa ca si parametru atunci. Sau cand se doreste sa se extinda functionalitatile unei clase.

class Car(private val engine: Engine) {

    fun start() {

        engine.start()

    }

}

fun main(args: Array) {

    val engine = Engine()

    val car = Car(engine)

    car.start()

}

// Nested classes

We can define classes inside other classes. By default, an inner class will not be able to access the members of the outer class.

class Outer {

 private val bar: Int = 1

class Nested {

 fun foo() = 2

 }

}

 val demo = Outer.Nested().foo() // == 2

If we want to access to the members of the outer class, we need to declare it as an inner class:

class Outer {

private val bar: Int = 1

 inner class Inner {

  fun foo() = bar

}

}

val demo = Outer().Inner().foo() // ==

// Delegation

interface CanFly {

fun fly()

 }

class Bird(f: CanFly) : CanFly by f

We can indicate that a bird can fly by using the interface, but the way the bird uses to fly is defined by a delegate that comes through the constructor, so we can have different birds with different flying methods. The way an animal with wings flies is defined in another class: class AnimalWithWings : CanFly {

val wings: Wings = Wings()

 override fun fly() = wings.move()

}

An animal with wings moves its wings to be able to fly. So now we can create a bird that flies using wings:

 val birdWithWings = Bird(AnimalWithWings())

 birdWithWings.fly()

But now wings can be used with other animals that are not birds. If we assume that bats always use wings, we could instantiate the object directly where we specify the delegation:

 class Bat : CanFly by AnimalWithWings()

...

 val bat = Bat()

 bat.fly()

// Extension functions

Bune cand nu detii clasa, gen o importi sau ceva si ca sa nu faci inheritance la clasa respectiva pe una noua, poti sa extinzi clasa respectiva.

Ex:

fun String.getAllWord() {return this.split(“ “)}

val a: String = “how are you?”;

val words = a.getAllWords();

// by keyword

val abc by map; by means provided by

// doAsync

**doAsync** {

*// do things in the background 1*

}

**(1)** In here, you can read or write to large files, download a file from the internet or do a task that will take a long time to complete. This block will execute in a background thread

It provides a doAsync function that executes its code in another thread, with the option to return to the main thread by calling uiThread.

doAsync() {

 Request(url).run()

 uiThread { longToast("Request performed") }

}

// with

inline fun <T, R> with(**receiver**: T, **block**: T.() -> R): R  
[(source)](https://github.com/JetBrains/kotlin/tree/30788566012c571aa1d3590912468d1ebe59983d/libraries/stdlib/src/kotlin/util/Standard.kt#L66)

Calls the specified function [block](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/with.html#kotlin$with(kotlin.with.T,%20kotlin.Function1((kotlin.with.T,%20kotlin.with.R)))/block) with the given [receiver](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/with.html#kotlin$with(kotlin.with.T,%20kotlin.Function1((kotlin.with.T,%20kotlin.with.R)))/receiver) as its receiver and returns its result.

The context object is avaiable as receiver ( this ).

The return value is the lambda result.

// let

The context object is available as an argument ( it ).

The return value is the lambda result.

The let function helps you with null-checks and creates a new local scope to safely perform operations. 

fun printCar(car: Car?) {

val isCoupe = car?.let { (it.doors <= 2) }

if (isCoupe == true) { println("Coupes are awesome") } }

let poate fi folosit pentru a invoca una sau mai multe funcții pe rezultatele lanțurilor de apeluri. De exemplu, următorul cod imprimă rezultatele a două operații pe o colecție: 

val numbers = mutableListOf("one", "two", "three", "four", "five”)

val resultList = numbers.map { it.length }.filter { it > 3 }

println(resultList)

Cu let, puteți rescrie exemplul de mai sus, astfel încât să nu atribuiți rezultatul operațiunilor din listă unei variabile:

val numbers = mutableListOf("one", "two", "three", "four", "five")

numbers.map { it.length }.filter { it > 3 }.let {

    println(it)

    // and more function calls if needed

}

Dacă blocul de cod transmis la let conține o singură funcție cu ea ca argument, puteți utiliza referința la metodă (::) în loc de argumentul lambda:

numbers.map { it.length }.filter { it > 3 }.let(::println)

let este adesea folosit pentru a executa un bloc de cod care conține valori non-nule. Pentru a efectua acțiuni asupra unui obiect non-null, utilizați operatorul de apel sigur ?. pe ea și cheamă let cu acțiunile în lambda sa.

val str: String? = "Hello"

//processNonNullString(str)       // compilation error: str can be null

val length = str?.let {

    println("let() called on $it")

    processNonNullString(it)      // OK: 'it' is not null inside '?.let { }'

    it.length

}

// run

The context is available as a context ( receiver ).

The result is a lambda result.

run face la fel ca with, dar este implementat ca o funcție de extensie. Deci, cum ar fi let, îl puteți apela pe obiectul context folosind notația punct.

val service = …;

val result = service.run { }

run este util atunci când lambda inițializează obiecte și calculează valoarea returnată.

Puteți, de asemenea, să invocați rularea ca funcție non-extensie. Varianta non-extensie a run-ului nu are obiect context, dar returnează totuși rezultatul lambda. Executarea fără extensie vă permite să executați un bloc de mai multe instrucțiuni în care este necesară o expresie.

val hexNumberRegex = run {

    val digits = "0-9"

    val hexDigits = "A-Fa-f"

    val sign = "+-"

    Regex("[$sign]?[$digits$hexDigits]+")

}

// apply

The context object is available as a receiver ( this ).

The return value is the object itself.

Deoarece apply returnează obiectul context în sine, vă recomandăm să îl utilizați pentru blocuri de cod care nu returnează o valoare și care operează în principal asupra membrilor obiectului receptor. Cel mai frecvent caz de utilizare pentru aplicație este configurarea obiectelor. Astfel de apeluri pot fi citite ca „aplicați următoarele atribuții la obiect”.

val adam = Person("Adam").apply {

    age = 32

    city = "London"

}

println(adam) // Person(name=Adam, age=32, city=London)

// also

The context object is available as an argument ( it ).

The return value is the object itself.

Also este util pentru efectuarea unor acțiuni care iau obiectul context ca argument. Utilizați also pentru acțiuni care necesită o referință la obiect, mai degrabă decât proprietățile și funcțiile acestuia, sau când nu doriți să umbriți this referință dintr-un domeniu exterior.

Când vedeți also în cod, îl puteți citi ca „și, de asemenea, faceți următoarele cu obiectul”.

val numbers = mutableListOf("one", "two", "three")

numbers

    .also { println("The list elements before adding new one: $it") }

    .add("four”)

// we also have the ability to create new threads or thread pools for coroutines. This is done by creating a new Executor. Executors are objects that execute given tasks. They are usually tied with Runnables, since they wrap the task in a runnable, which needs executing

// Toast - footer care apare putin in partea de jos si dispare foarte rapid. Eroare pentru devs sau debugging

// facade, adapter, proxy, observer, factory patterns

// Observer are 3 states: onNext - when emits a new item to the Observer, onComplete - notifies all the observers that the observable is done with its task, onError - notifies the observer that the Observable experienced an error, onSubscribe

// Live Data

LiveData is a lifecycle-aware component that wraps around objects you wish to emit in a reactive way, much the same way the Observable object does in RxJava. LiveData is “live” in the sense that when the underlying data updates, anything observing that data will also receive the updates. For example, when paired with Room, LiveData retrieved from the Room database automatically updates when the data in the database changes.

Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

LiveData considers an observer, which is represented by the [Observer](https://developer.android.com/reference/androidx/lifecycle/Observer) class, to be in an active state if its lifecycle is in the [STARTED](https://developer.android.com/reference/androidx/lifecycle/Lifecycle.State#STARTED) or [RESUMED](https://developer.android.com/reference/androidx/lifecycle/Lifecycle.State#RESUMED) state. LiveData only notifies active observers about updates. Inactive observers registered to watch [LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData) objects aren't notified about changes.

/// MVVM - Model-View-ViewModel

* View displays the UI and informs the other layers about user actions.
* ViewModel exposes information to the View.
* Model retrieves information from your datasource and exposes it to the ViewModels

Thanks to Google's new Architecture Components, you now have a special class to build your ViewModels called ViewModel. The ViewModel class is specially designed to manage and store information in a lifecycle-aware manner. This means that the data stored inside it can survive configuration/lifecycle changes like screen rotations.

The ViewModel remains in memory until the lifecycle object to which it belongs has completely terminated. This behavior applies to activities when they finish and in fragments when they are detached.

View -> ViewModel -> Model - au referinta directa, invers nu, nu detin referinta catre părinte, ceea ce inseamna ca model si viewModel vor expune date care vor putea fi observate de catre părinți.

// ANR - application not responding dupa 5s blocked pe main thread

// context

iti da info de orice resursa ce sistemul de operare o da la dispoziție

informatii globale despre un mediu de aplicatie - permite accesul la reurse si clase specifice aplicației

Contexts contain the following information that views require:

* device screen size and dimensions for converting dp,sp to pixels
* styled attributes

Application Context and an Activity Context, which last for the duration of their respective lifecycle. Most [Views](https://guides.codepath.com/android/Defining-Views-and-their-Attributes) should be passed an Activity Context in order to gain access to what themes, styles, dimensions should be applied. In most cases, you should use the Activity Context

// coroutines context

Contextul corutinei este un set de diverse elemente. Elementele principale sunt Job-ul corutinei si dispecerul său

The coroutine's [Job](https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) is part of its context, and can be retrieved from it using the coroutineContext[Job]

A coroutine itselfis represented by a [Job](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html). It is responsible for coroutine’s lifecycle, cancellation, and parent-child relations.

Coroutine context is immutable, but you can add elements to a context using [plus](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.coroutines/-coroutine-context/plus.html) operator, just like you add elements to a set, producing a new context instance

// public, private, internal(ca si public, doar ca in packet), protected(mostenire, nu sunt visibile in exterior, dar la copii lor)

// activitatea unui UI - onCreate - perform basic application startup logic that happens only once for the entire life of the activity. , onStart - This call makes the activity visible to the user as the app prepares for the activity to enter the foreground and become interactive, onResume - activitatea e interactionabila, onPause - activitatea nu e interactionabila(As long as the activity is partially visible but not in focus, it remains paused - multi window), onStop - activitatea nu mai e vizibila, onDestroy - activitatea e distrusă

private

The private modifier is the most restrictive we can use. It indicates it is visible in its file. So if we declare a class as private, we are not able to use it outside the file where it was defined. On the other hand, if we use private inside a class, the access is restricted to that class. Even classes that extend it cannot use it. So first level classes, objects, interfaces… (known as package members) declared as private are only visible inside the file where they are declared, while everything defined inside a class or interface is only visible by that class or interface.

protected

This modifier only applies to members of a class or an interface. A package member cannot be protected. Inside a member, it works the same way as in Java: it can be used by the member itself and the members that extend it (for instance, a class and its subclasses).

 internal

An internal member is visible inside the whole module if it is a package member. If it is a member of another scope, it depends on the visibility of the scope. For instance, if we write a private class, the access to an internal function is limited to the visibility of the class. We can use internal classes from any other class in the same module, but not from another module.

public

As you may guess, this is the less restrictive modifier. \*\* It is the default modifier\*\*, and a member declared as public is visible anywhere, only restricted by its scope. A public member defined in a private class will not be visible outside the scope where the class is visible

// flows

flow  ca si coroutine cu emit si apoi le iau cu collect sau collectLatest. flow-ul nu face nimic atata timp cat nu sunt subscriberi - asta e cold flow

un hot flow - emite desi nu sunt colectori, adica subscriberi

-collect (le ia pe rand, chit ca dureaza si are delay-uri collect-ul)

-collectLatest (daca are delay si ii tot vin pe collect, anuleaza tot ce are si ia ultimul colect si tot asa)

Reduce,filter, flatmap - pe flows

Buffer, conflate cu collect ca sa nu se faca delay in aceeasi coroutine, practic vor rula pe coroutine separate. Buffer le ia pe toate, cu conflate daca sunt 2 trimise, va lua pe ultima

state flow - se bazeaza pe o valoare, isi notifica toti colectorii, si persista si la rotate pentru ca isi va notifica din nou colectorii ( hot flow). Trimite aceeasi valoare la toti subcriberii

repeatOnLifecycle - sa ii zici pe ce lifecycle sa funcționeze

shared flow - one time emission, nu va notifica iar si la rotate de device. daca nu sunt colectori, eventul va fi pierdut daca e primit ( ca si hot flow ). Daca ii dai replay = number, poti face cache de number events si cand noi colectori fac subscribe la flow, ei primesc events-urile. Trimite aceeasi instanță  la toti subcriberii

// flow-uri, state flow-uri - functioneaza pe baza de coroutine, fata de livedata care foloseste thread-uri