Singleton : این اطمینان را ایجاد میکند که از یک کلاس ، تنها یک نمونه وجود دارد و امکاناتی را فراهم میکند که ان یک نمونه به صورت سراسری قابل دسترسی باشد

# آموزش koin

Koin : طبق گفته های سازنده های کوین، این کتابخونه میتونه فقط توی 3 مرحله وابستگی های مختلف رو برای ما تزریق کنه

1. اولین قدم اینه که فایل Module  (ماژول) رو درست کنی و چیزهایی که میخوای تزریق کنی رو اونجا قرار بدی.
2. سپس فایل ماژول رو باید توی کلاس Application  معرفی کنی.
3. درنهایت هم کلاس ها رو به کمک دستور inject توی صفحه ای که میخوای ازشون استفاده کنی تزریق کنی.

Module : ماژول درواقع وابستگی هایی که ما بهشون نیاز داریم رو برای ما تامین میکنه.

وابستگی های مختلف، اعم از کلاس، فانکشن و متغیر رو داخل ماژول تعریف میکنیم و ماژول هم وظیفه تامین کردن اونارو بر عهده داره.

inject : وابستگی هایی که توسط ماژول تامین شدن رو به کمک inject  (اینجکت) میتونیم توی Activity / Fragment  استفاده کنیم.

## مرحله اول

اولین مرحله ساختن ماژول بود.

برای اینکار میتونیم یک فایل جدید درست کنیم و کلاس ها رو داخلش قرار بدیم.

من یک فایل جدید با نام ModuleViewModel درست کردم و کدهای زیر رو هم داخل قرار دادم.

*val viewModelModule* = *module* **{** *viewModel* **{** VmAddClass(get()) **}** *viewModel* **{** VmHomePage(get()) **}** *viewModel* **{** VmAddSession(get()) **}** *viewModel* **{** VmNote(get()) **}** *viewModel* **{** VmProfile(get()) **}** *viewModel* **{** VmSession(get()) **}** *viewModel* **{** VmStatistics(get()) **}** *viewModel* **{** VmStudent(get()) **}** *viewModel* **{** VmAccount(get()) **}  
}**

برای اینکه بتونی از ماژول کوین استفاده کنی باید اون رو داخل یک متغیری بریزی، برای همین هستش که من داخل فایل ModuleViewModel یک متغیر با نام viewModelModule درست کردم و اون رو از نوع  module قرار دادم.

#### انواع روش های تامین وابستگی در ماژول

توی کوین ما 2 روش برای تامین کردن وابستگی ها داریم :

1. single  : به صورت سینگلتون تامین میکنه (یکبار در روند کل اجرای اپلیکیشن)
2. factory  :در هربار صدا کردن یک آبجکت جدید ازش میسازه.

توی فایل بالا یعنی ModuleViewModel من هردو کلاس رو توسط  single تامین کردم.

کلاس VmAddClass چون وابستگی به ریپازیتوری داشت برای همین در ورودی این کلاس get() رو قرار دادیم و در پایین وابستگی کلاس ریپازیتوری را تامین کردیم .

*val repositoryModule* = *module* **{** *fun* provideUserRepository(dao: *RunDao*): RepositoryRoom {  
 *return* RepositoryRoom(dao)  
 }  
  
 single **{** provideUserRepository(get()) **}  
}**

مهم : چون وابستگی کلاس RepositoryRoom رو تامین کردیم، پس خیلی راحت به کمک کلمه کلیدی get  (جز کلمات کلیدی کوین هستش) میتونی وابستگی کلاس های ویومدل رو مثل( VmAddClass ) رو تامین کنیم. درواقع با نوشتن کلمه get ، کوین به صورت خودکار وابستگی مورد نیاز رو (در صورتی که قبلا تامین شده باشه) تامین میکنه.

بعد از اینکه وابستگی ها رو توی ماژول فراهم کردیم، نوبت این رسیده که اون ماژول رو توی کلاسApplication  تعریف کنیم.

## مرحله دوم

برای اینکار باید کلاس اپلیکیشن رو درست کنی و داخل متد  onCreate اون کد زیر رو قرار بدی.

**class** MyApp : Application**()** **{**

override **fun** onCreate**()** **{**

super.onCreate**()**

startKoin **{**

//Context

*androidContext*(*this*@OstadApplication)

//module

modules(*listOf*(*repositoryModule*, *viewModelModule*,*databaseModule*))

//

*androidLogger*(Level.*DEBUG*)  
*androidFileProperties*()

**}**

**}**

برای اینکه بتونی ماژول رو توی کلاس Application فراخوانی کنی، باید توی onCreate متد startKoin رو بنویسی.

با نوشتن startKoin درواقع به کوین میگیم که ماژول هایی که توی این متد نوشته میشن رو به اپلیکیشن تزریق کنه.

مهم : هر چندتا ماژولی که داریم رو باید داخل modules بنویسیم، فقط کافیه که با , از هم جداشون کنیم. مثال :

modules(listOf(repositoryModule, viewModelModule,databaseModule))

از اونجایی که context  جز مواردی هستش که خیلی توی پروژه ها ازش استفاده میشه، کوین برای تزریق اونم یک روشی رو معرفی کرده.

فقط کافیه که داخل startKoin از کد androidContext  استفاده کنی.

یادت نره که کلاس اپلیکیشن رو توی فایل Manifest پروژه تعریف کنی. برای اینکار فقط کافیه کد زیر رو اضافه بکنی به مانیفست

<application  
 *android:name*=".App"

## مرحله سوم

خب الان دیگه نوبت این رسیده که ازشون استفاده کنیم.

برای استفاده کردن از مواردی که وابستگی هاشون رو تامین کردیم، باید وارد Activity  یا Fragment  بشیم و به کمک inject  اون کلاس/متد/متغیر رو تزریق کنیم.

برای اینکار میتونم از کد زیر استفاده کنیم :

*private val* vmLogin: VmAccount *by inject*()

برای ویو مدل ها میتونیم از ویو مدل استفاده کنیم

*private val* vmHome:VmHomePage *by viewModel*()

ولی اکثر مواقع از inject  استفاده میکنند

# SOLID

SOLID (سالید) یک کلمه مخفف برای 5 اصل هست. هدف معرفی این اصول اینه که برنامه‌ها قابل درک‌تر، انعطاف‌پذیر تر و بیشتر قابل نگهداری باشن. به عنوان یک برنامه‌نویس، توسعه‌دهنده و مهندس نرم‌افزار، یادگیری این پنج اصل جزو "باید" ها هست. این اصول میتونن توی هر طراحی شی‌گرایی اعمال بشن.

### **1. اصل تک مسئولیتی (Single Responsibility Principle)**

هر کلاسی که توی برنامه‌ی ما وجود داره، باید یک مسئولیت خاص و مشخص داشته. در واقع این کلاس باید فقط و فقط مسئول یک عملکرد توی برنامه باشه.

### **2. اصل باز - بسته (Open/Closed Principle)**

موجودیت‌های یک نرم‌افزار (کلاس‌ها، ماژول‌ها، توابع و ...) باید برای توسعه داده شدن(extend)، باز و برای تغییر(modifier) دادن، بسته باشن

### **3. اصل جایگزینی لیسکوف (Liskov Substitution Principle)**

اگر S یک زیر کلاس T باشه، آبجکت‌های نوع T باید بتونن بدون تغییر دادن کد برنامه، با آبجکت‌های نوع S جایگزین بشن.

به بیان ساده‌تر کلاس‌های فرزند نباید رفتار و ویژگی‌های کلاس والد رو تغییر بدن

در اصل میگه اگر بجای کلاس پدر از کلاس فرزند استفاده کردیم نباید مشکلی در برنامه ایجاد شود چون کلاس فرزند زیرکلاس پدر است و همه ویژگی های پدر را ارث بری میکند

### **4. اصل جداسازی اینترفیس‌ها (Interface Segregation Principle)**

کلاس‌ها نباید مجبور باشن متدهایی که به اونها احتیاجی ندارن رو پیاده‌سازی کنن.

در واقع این اصل میگه که ما باید اینترفیس (Interface) ها رو جوری بنویسیم که وقتی یک کلاس از اون استفاده میکنه، مجبور نباشه متدهایی که لازم نداره رو پیاده‌سازی کنه. یعنی متدهای بی‌ربط نباید توی یک اینترفیس کنار هم باشن. این اصل شباهت زیادی به اصل اول SOLID داره که میگه کلاس‌ها باید فقط مسئول انجام یک کار باشن.

### **5. اصل وارونگی وابستگی (Dependency Inversion Principle)**

کلاس‌های سطح بالا نباید به کلاس‌های سطح پایین وابسته باشن؛ هر دو باید وابسته به انتزاع (Abstractions) باشن. موارد انتزاعی نباید وابسته به جزییات باشن. جزییات باید وابسته به انتزاع باشن

#### کلاس سطح پایین چیه؟

به کلاس‌هایی گفته میشه که مسئول عملیات اساسی و پایه‌ای توی نرم‌افزار هستن. مثل کلاسی که با دیتابیس یا هارددیسک ارتباط برقرار می‌کنه، کلاسی که برای ارسال ایمیل استفاده میشه و ...

#### کلاس سطح بالا؟

کلاس‌هایی که عملیات پیچیده‌تر و خاص‌تری انجام میدن و برای انجام این کار از کلاس‌های سطح پایین استفاده میکنن. برای مثال کلاس گزارش‌گیری برای ثبت و خوندن گزارش، به کلاس دیتابیس یا هارددیسک نیاز داره. کلاس Users، برای اطلاع‌رسانی به کاربرها به کلاس ایمیل نیاز داره.

# Hilt in Android

## تزریق وابستگی ها Dependency Injection -

وابستگی چیه ؟ اولا که وقتی میگیم Dependency منظورمون اون چیزایی که شما تو فایل gradle می‌نویسید نیست ! هر کلاس می‌تواند به کلاس های دیگر وابسته باشد ، مثلا کلاس “ماشین” وابستگی به کلاس “چرخ” داره و این وابستگی باید به اون تزریق بشه ، یکی از راه ها اینه که ما تو “سازنده” یا “constructor” کلاسِ “ماشین” ، اِلمانِ کلاس “چرخ” رو بیاریم و ما با این کار به نوعی عملِ “تزریق وابستگی رو انجام دادیم” ، ولی اگه بعدا سازنده رو عوض کنیم چی ؟ هزار جای کد باید این تغییر رو اعمال کنیم ! پس بهترین کار اینه که به نحوی بیاییم این عملیات رو در لایه های بالاتری از انتزاع (Abstraction) انجام بدیم .

## اضافه کردن کتابخانه هیلت به برنامه

ابتدایِ امر باید کتاب‌خونه HILT رو به پروژه اضافه کنیم ، اول به gradle اصلی برید :

buildscript **{**

repositories **{**

*// other repositories...*

mavenCentral**()**

**}**

dependencies **{**

*// other plugins...*

classpath 'com.google.dagger:hilt-android-gradle-plugin:2.43.2'

**}**

**}**

و بعد به gradle app برید ، اول دو پلاگین مورد نیاز رو در بالای پروژه اضافه کنید :

apply plugin: 'com.android.application'

apply plugin: 'com.google.dagger.hilt.android'

android **{**

*// ...*

**}**

و بعد dependency های مربوط رو اضافه می‌کنیم :

dependencies **{**

implementation 'com.google.dagger:hilt-android:2.43.2'

kapt 'com.google.dagger:hilt-compiler:2.43.2'

// برای اضافه کردن ویو مدل ها به گراف هیلت باید این کتابخانه را داشته باشیم

*def* nav\_version = "2.5.1"  
 implementation("androidx.navigation:navigation-compose:$nav\_version")

//برای اینکه در فرگمنت ها بتونیم از هیلت استفاده کنیم باید از این نسخه فرگمنت به بالا استفاده کنیم

*def* fragment\_version = "1.5.1"  
 implementation("androidx.fragment:fragment-ktx:$fragment\_version")

}

kapt **{**

correctErrorTypes **true**

**}**

 نکته بسیار مهم: برای استفاده از هیلت نیازی به کد نویسی پیچیده نداریم ما فقط با چند انوتیشن (حاشیه نویشی) همه چیزایی که لازم است را بدست میاریم

## اضافه کردن هیلت به اپ

در این قسمت تنها کاری که باید بکنیم اضافه کردن انوتیشن @HiltAndroidApp به کلاس اپ است

@HiltAndroidApp

class App : Application() {

}

و تمام باقی کارها توسط خودِ HILT هندل میشه و دیگه لازم نیست مثل Dagger کلی کد بنویسید و بعد یه بارم Build بگیرید که کلاس ها ساخته بشن و … ، با همین یه دونه انوتیشن ما کلاسِ App رو وارد HILT کردیم و حالا می‌تونیم توش Dependency ها رو تزریق کنیم !

## تزریق Inject

ما دو نوع تزریق (Inject) وابستگی داریم ، یکی به صورت Field و یکی به صورت Constructor ، در واقع Field Injection حالت عادی قضیه حساب میشه ، مثلا شما در کدهاتون با انوتیشن Inject مشخص می‌کنید که برای این شئ باید Inject صورت بگیره اما در Constructor Injection شما Constructor اون شئ رو با کلمه Inject مشخص می‌کنید و بعد از اون هر زمانی که اون شئ Inject بشه ورودی هاش هم Inject میشند ، اگر درست متوجه نشدید مثال پایین کمک می‌کنه :

class BlawBlaw {

@Inject

lateinit var obj: SomeClass

// Code

}

class OtherBlawBlaw @Inject constructor(var obj: SomeOtherClass) {

// Code

}

در این کد SomeClass به صورتِ Field و SomeOtherClass به صورت Constructor به درون کلاس ها Inject شدند ، حالا فرض کنید بخوایم به درون یک Activity یا … یک شئ رو Inject کنیم ، چه کار باید بکنیم ؟

ما با مشخص کردن HiltAndroidApp برای کلاسِ App این کلاس رو به داخل گرافِ HILT اضافه کردیم ، حالا همین کار رو با انوتیشن دیگه ای باید انجام بدیم و این انوتیشن ، AndroidEntryPoint هست ، این انوتیشن می‌تونه برای Activity ، Fragment ، Service ، BroadcastReceiver و View استفاده بشه ،با این انوتیشن ما به کلاس اجازه میدیم که بتونه از کلاس های دیگه تزریق داشته باشه به کدِ زیر دقت کنید :

|  |
| --- |
| @AndroidEntryPoint |
| class MainActivity : AppCompatActivity() { |
| @Inject lateinit var blawBlaw : BlawBlaw |
|  |
| override fun onCreate(savedInstanceState: Bundle?) { |
| super.onCreate(savedInstanceState) |
| setContentView(R.layout.activity\_main) |
| } |
| } |
|  |
| class BlawBlaw @Inject constructor(){ |
| @Inject |
| lateinit var obj: SomeClass |
| // Code |
| } |
|  |
| class SomeClass @Inject constructor(var someOtherClass: SomeOtherClass) |
|  |
| class SomeOtherClass @Inject constructor() |
|  |

@AndroidEntryPoint  
*class* LoginActivity : AppCompatActivity() {  
  
 *val* vmLogin: VmAccount *by viewModels*()  
  
 @Inject  
 *lateinit var* funcion: Funcion

}

در این کد ما MainActivity رو به گرافِ HILT اضافه کردیم (منظورم اینه که الان MainActivity زیر مجموعه App حساب میشه از نظر HILT) و با اضافه کردنش به گراف ، توانایی Inject کردن رو بهش دادیم ، شئِ BlawBlaw به صورت Field به درون MainActivity تزریق شده ، از طرفی در شیء BlawBlaw ما یک متغیر به نام SomeClass داریم که اون هم باز به صورت Field تزریق شده ، در کلاسِ SomeClass اما دیگه تزریق رو به صورت Field انجام ندادیم ، در SomeClass کلاسِ دیگه ای به اسم SomeOtherClass به صورت Constructor به درون SomeClass تزریق شده ! امیدوارم با این مثال قشنگ تفاوت این دو براتون جا افتاده باشه !

برای استفاده از ویو مدل ها هم تنها کافیه به کلاس ویومدل ها انوتیشن @HiltViewModel را اضافه کنیم و بعد *constructor های کلاس رو اینجکت کنیم با این کار ما این کلاس ویومدل را به گراف هیلت اضافه میکنیم و هر موقع که بخواهیم میتونیم از این کلاس ویومدل استفاده کنیم*

@HiltViewModel  
*class* VmAccount @Inject *constructor*(*val* repositoryRoom: RepositoryRoom) : ViewModel() {

...

{

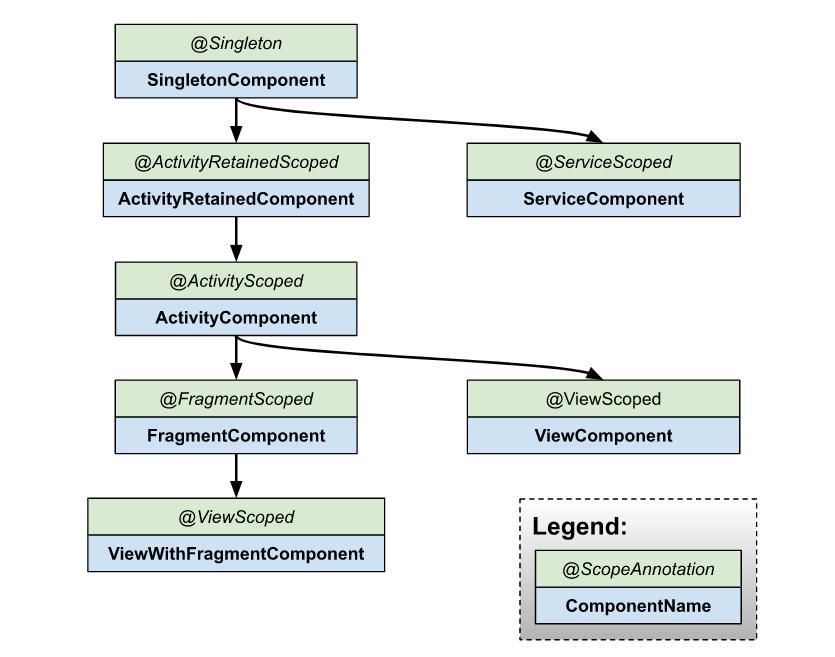
در پایان برای استفاده از ویو مدل تنها کافیست ان را صدا بزنیم و فراموش نکنیم که انوتیشن AndroidEntryPoint را به اکتیویتی خودمان اضافه کنیم و برای صدا زدن به شکل زیر عمل میکنیم

@AndroidEntryPoint  
class LoginActivity : AppCompatActivity() {  
 val vmLogin: VmAccount by viewModels()

{

## ایجاد Component

در Dagger ما مستقیم خودِ Component رو ایجاد می‌کردیم ولی در HILT این‌کار رو نمی‌کنیم ، در HILT ما از یک سری Component از پیش ساخته شده استفاده می‌کنیم ، یعنی به جای اینکه یک کلاس FolanComponent بسازیم فقط میاییم اون Dependency مورد نظر رو با یک انوتیشن مشخص می‌کنیم تا خودش عملیات ساخت Component مورد نظر و وصل کردنش به اون کلاس رو انجام بده ، این انوتیشن ها مشخص کننده Scope و محدوده مورد نظر ما هستند

: 

اگر به شکل دقت کنید می‌بینید یک حالت درخت/ارث بری برای ما کشیده شده ، یعنی چی ؟ فرض کنید ما یک Dependency داریم ، اون رو باید با Scopeای با نام ActivityScope مشخص کنیم ، حالا وقتی این کار رو کردیم اون Dependency فقط می‌تونه از طریق ActivityComponent به پایین (FragmentComponent و ViewWithFragmentComponent) تزریق بشه ، یعنی Dependency ای که متخص Activity هست رو در کلاس Application نمی‌تونید استفاده کنید ولی برعکسش مشکلی نداره . در مورد قسمت Legend هم برای زمانیه که بخواید یک Component دلخواه بسازید که کاری باهاش نداریم . خب پس تا اینجای کار فهمیدیم که لزومی به ساخت Component نیست و از Componentهای از قبل ساخته شده استفاده می‌کنیم ، حالا بریم سراغ ماژول .

## ایجاد Module

فرض کنید یک کلاسِ پیچیده داریم (ComplexModel) که دیگه نمی‌تونیم مثل روش های مثال های قبلی خیلی راحت اونو Inject کنیم بلکه نیاز به تابعی داریم که این کلاس رو برامون تهیه یا Provide کنه ، صورت مساله به این شکله که قصد داریم یک Module به نام ActivityModule بسازیم که وظیفه اش تزریق وابستگی‌ها به کلاس MainActivity باشه ، به کد زیر دقت کنید :

|  |  |
| --- | --- |
|  | @AndroidEntryPoint |
|  | class MainActivity : AppCompatActivity() { |
|  |  |
|  | @Inject lateinit var complexModel: ComplexModel |
|  |  |
|  | override fun onCreate(savedInstanceState: Bundle?) { |
|  | super.onCreate(savedInstanceState) |
|  | setContentView(R.layout.activity\_main) |
|  | Log.d("MyTag",complexModel.someText()) |
|  | } |
|  | } |
|  |  |
|  | @Module |
|  | @InstallIn(ActivityComponent::class) |
|  | class ActivityModule{ |
|  | @ActivityScoped |
|  | @Provides |
|  | fun provideComplexModel() : ComplexModel{ |
|  | val complexModel = ComplexModel() |
|  | return complexModel |
|  | } |
|  | } |
|  |  |
|  | class ComplexModel(){ |
|  | fun someText() : String = "Blaw blaw" |
|  | } |

در این کد اول از همه MainActivity رو داخل گرافِ HILT کردیم (با AndroidEntryPoint) ، بعد از اون ActivityModule رو ساختیم ، وقتی شما می‌خواید مشخص کنید Module مورد نظر مربوط به کدوم Component باشه باید از InstallIn استفاده کنید و توش جنسِ کلاسِ اون Component رو بنویسید (برای اسم های Componentها به عکس های بالاتر نگاه کنید) ، خودِ Module رو هم با انوتیشن Module مشخص می‌کنیم که HILT بفهمه این کلاس یک Module حساب میشه ، حالا برای تهیه Dependency دو راه داریم ، یکی استفاده از Bind یکی استفاده از Provide (که فعلا ما با همین Provide کار داریم) ، تابعی که داره برای ما Provide رو انجام میده provideComplexModel هست که میاد یک ComplexModel رو برای ما برمی‌گردونه ، خودِ این تابع رو هم می‌تونیم Scopeگذاری کنیم (می‌تونیم هم نکنیم چون به صورت پیش فرض این خودش می‌فهمه) و بعد در کلاسِ MainActivity عملیات Inject رو انجام می‌دیم .

نکته : HILT به صورت Compile-Time کدها رو بررسی می‌کنند پس اگر خطایی وجود داشته باشه اجازه اجرای برنامه داده نمیشه و برناه Build نمیشه پس نگران جا به جا نوشتن Scopeها و … نباشید

نکته : در موردِ Bind نمی‌خوام توضیحی بدم ، در همین حد بدونید که اگه از Bind استفاده کنید نیازی نیست بدنه تابع رو بنویسید و به همین خاطر نمیشه همیشه ازش استفاده کرد (مثلا اگه بخواید Retrofit رو Inject کنید) پس بهتره از همون Provide استفاده کنید

@InstallIn(*SingletonComponent*::*class*)  
@Module  
*class* DatabaseModule {  
  
 @Provides  
 @Singleton  
 *fun* funcionModule() = Funcion()  
  
 @Provides  
 @Singleton  
 *fun* provideAppDatabase(@ApplicationContext appContext: Context): DbOstad {  
 *return* Room.databaseBuilder(appContext, DbOstad::*class*.*java*, "daftar")  
 .fallbackToDestructiveMigration()  
 .allowMainThreadQueries()  
 .build()  
 }  
  
 @Provides  
 @Singleton  
 *fun* provideDao( database : DbOstad): *Dao* {  
 *return* database.Dao()  
 }  
  
  
}

# OOP

# Intent-filter

Intent-filter اینگونه عمل می کنند که وقتی از طریق Intent در خواستی برای اندروید فرستاده شد، اندروید همه ی نرم افزار هایی که برای انجام آن در خواست مناسب هستند را تشخیص و برای کاربر لیست می کند، تا کاربر نرم افزار دلخواه خود را برای انجام آن درخواست انتخاب کند. در واقع اندروید به وسیله ی Intent-filter ها تشخیص می دهد که چه نرم افزار هایی امکان اجرای آن درخواست را دارد.

یک Intent-filter مشخص می کند که نرم افزار شما قادر است چه نوع در خواست هایی را اجرا کند. در واقع Intent-filter، قابلیت های نرم افزار شما را تعریف می کند. اگر برای یک اکتیوتی یا سرویس Intent-filter تعریف نشود، آن اکتیویتی فقط به صورت واضح یا (Explicit Intent) قابلیت فراخوانی دارد.

برای تعریف Intent-filter ها دو راه وجود دارد:

**1. تعریف آن ها در AndroidManifest**

**2. استفاده از BroadCastReceiver**

در روش اول وقتی نرم افزار نصب می شود، اندروید تمام Intent-filter هایی که تعریف کرده اید را ثبت می کند. با این کار اگر کاربر درخواستی را ارسال کند که در Intent-filter شما تعریف شده باشد که برنامه شما توانایی انجام درخواست را دارد، برنامه شما به کاربر پیشنهاد داده می شود. نیازی نیست که نرم افزار شما درحال اجرا یا قبلا اجرا شده باشد؛ چرا که قبلا در اندروید ثبت شده که برنامه شما این توانایی را دارد.

به مثال زیر توجه کنید:

<activity

android:name=”.ActivityExample”

android:label=”Example”>

<intent-filter>

<action android:name=”android.intent.action.VIEW” />

<category android:name=”android.intent.category.DEFAULT”/>

<data android:schema=”https”/>

</intent-filter>

</activity>

در مثال بالا نرم افزار به عنوان مرورگر وب در اندروید ثبت می شود؛ حالا تمام ادرس های اینترنتی که با https شروع شود برنامه شما هم به عنوان نرم افزار پیشنهادی به کاربر نمایش داده میشود.

BroadCastReceiver رو اگه بخوایم ساده بیان کنیم به معنی فالگوش وایسادن میشه. وقتی درخواستی توسط کاربر ارسال میشه اندروید با استفاده از BroadCast درخواست رو اعلام میکنه و هر نرم افزاری که Reciver برای اون BroadCast تعریف کرده باشه، به کاربر پیشنهاد داده میشه.

# Retrofit

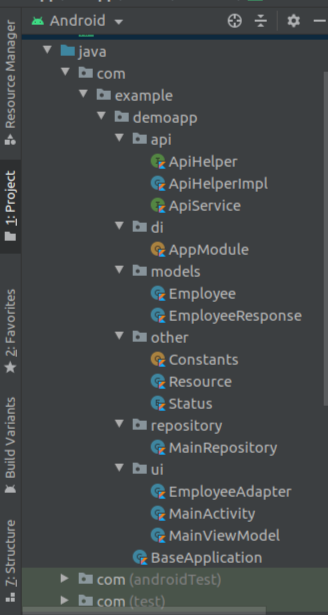
Retrofit یک کلاینت از نوع REST  برای زبان جاوا و اندروید است . این کتابخانه به شما کمک میکند تا به سادگی عمل دریافت و ارسال اطلاعات به وب سرویس هایی که مبتنی بر REST (یا هر نوع ساختار دیگر) هستند را انجام دهید . ارسالا و دریافت اطلاعات با فرمت JSON میباشد . بجای JSON شما میتوانید Retrofit را برای کار دیگر نوع داده ها مثل XML یا نوع های دیگر تنظیم کنید .

Retrofit از کتابخانه ی OkHttp  برای پروتکل HTTP استفاده میکند یعنی یه جورایی میشه گفت retrofit بر پایه ی OkHttp پیاده سازی شده است .

## نحوه ی استفاده از Retrofit

برای کار با Retfofit شما اساسا نیاز به کلاس های زیر خواهید داشت

* یک کلاس مدل که اطلاعات مدل JSON را استفاده میکند
* Interface هایی که عملیات HTTP  را تعریف میکند
* کلاس Retrofit.Builder - برای تعریف خیلی از چیزها مثل URL  مربوط به درخواست HTTP و ... استفاده میشود



## نصب و راه اندازی Retrofit  در محیط android studio

خب برای استفاده از Retrofit باید ابتدا انرا نصب کنید . همچنین ما نیاز به Gson هم داریم که باید آنرا هم نصب کنیم و همچنین converter مربوطه به Gson را !

ابتدا وارد فایل Build.gradle از نوع Module شده و در بخش dependencies سه خط زیر را کپی کرده و دکمه sync را بزنید.

*def* retrofit\_version = "2.9.0"  
*def* httplogging\_version = "4.9.2"  
*def* json\_version = "2.9.1"  
implementation "com.squareup.retrofit2:retrofit:$retrofit\_version"  
implementation "com.squareup.retrofit2:converter-gson:$retrofit\_version"  
implementation "com.squareup.okhttp3:logging-interceptor:$httplogging\_version"  
implementation "com.google.code.gson:gson:$json\_version"

البته برای بارگزاری و نصب شدن این کتابخانه چون کشور ایران تحریم است کاری انجام دهید که IP شما عوض شود مثلا از vpn استفاده کنید .

چون این کتابخانه ها از اینترنت برای برقراری ارتباطات سروری استفاده میکنند شما باید مجوز اینترنت را به اپلیکیشن خود بدهید وگرنه اپ شما کار نمیکند .

مجوزها در فایل AndroidManifest تعریف میشود .

مجوز زیر را بهش اضافه کنید :

<uses-permission android:name="android.permission.INTERNET" />

خب الان کتابخانه ی Retrofit و کتابخانه ی Gson نصب شده اند و مجوز اینترنت هم به اپ شما داده است . یعنی کاملا اماده است که کدنویسی کنید و ارتباط با سرور را برقرار کنید .

## ساخت ماژول رتروفیت

کلاس Retrofit.Builder برای تعریف خیلی از چیزها مثل URL  مربوط به درخواست HTTP و ... استفاده میشود

ساختن ابجکت کلاس با انوتیشن های مختلف مثل @Module, @InstallIn, @Singleton و @Provides که Provides وابستگی هارو فراهم میکنه کلاس ماژول متدهای وابسته لازم را برای ماژول Network فراهم می کند.

* Module : نمونه های مورد نیاز را به عنوان وابستگی برای کلاس های مختلف فراهم می کند. برای اینکه محدوده ماژول را مشخص کنیم از انوتیشن ماژول هیلت با @InstallIn استفاده میکنیم . در اینجا، SingletonComponent::class ظرف singleton را برای کلاس تولید می کند.
* Provides : این انوتیشن اشیا را برای تزریق فراهم میکند
* Singleton : برای ایجاد نمونه singleton از شیء وابستگی و استفاده از آن در سراسر برنامه حاشیه‌نویسی شده است.

برای اینکه ما از هیلت استفاده میکنیم باید مثل مثال زیر رتروفیت رو معرفی کنیم

@Module  
@InstallIn(*SingletonComponent*::*class*)  
*class* ApiModule {

*const val* baseUrl=" https://restcountries.eu/rest/v2/"

@Singleton  
 @Provides  
 *fun* providesHttpLoggingInterceptor() = HttpLoggingInterceptor()  
 .*apply* **{** level = HttpLoggingInterceptor.Level.*BODY* **}** @Singleton  
 @Provides  
 *fun* providesOkHttpClient(httpLoggingInterceptor: HttpLoggingInterceptor): OkHttpClient =  
 OkHttpClient  
 .Builder()  
 .addInterceptor(httpLoggingInterceptor)  
 .build()  
  
 @Singleton  
 @Provides  
 *fun* provideRetrofit(okHttpClient: OkHttpClient): Retrofit = Retrofit.Builder()  
 .addConverterFactory(GsonConverterFactory.create())  
 .baseUrl(baseUrl)  
 .client(okHttpClient)  
 .build()  
  
 @Singleton  
 @Provides  
 *fun* provideApiService(retrofit: Retrofit): *ApiService* = retrofit.create(*ApiService*::*class*.*java*)  
  
 @Singleton  
 @Provides  
 *fun* providesRepository(apiService: *ApiService*) = Repository(apiService)  
  
}

## ساخت ApiService

هر متود از اینترفیس یک درخواست از API را فراهم میکند . منظور از این درخواست ها همان درخواست های HTTP است که میتواند از نوع GET , POST یا هر چیز دیگری باشد .

نتایج درون آبجکت Call بسته بندی میشوند . مثال زیر را ببینید :

## انوتیشن ها (annotation) در Retrofit

انوتیشن ها (annotation) یا به عبارت فارسی حاشیه نوشت ها درواقع برای توصیف نوع درخواست ما و همچنین نگهدارنده ای برای پارامترهای ارسالی و یا Multipart request body (درخواست های چندپاره) ما هستند (نکته مهم اینکه هر درخواست رتروفیت باید با یک annotation مشخص شده باشد)

دررتروفیت ما درکل پنج annotation برای مشخص کردن نوع درخواست رتروفیت داریم

GET, POST, PUT, DELETE, HEAD

که در ادامه هرکدام را به طور واضح توضیح خواهیم داد.

### GET annotation

با انوتیشن @GET درخواست ما تبدیل به یک ریکوست رتروفیت دریافتی میشود بدین صورت که پارامتر ارسالی به سرور ندارد و فقط قصد گرفتن دیتاها را از سرور دارد

برای مثال در مثال زیر درخواست رتروفیت ما از نوع Get هستش و endpoint هم مقدار getUsers.php هستش

@GET("getUsers.php")

اما endPoint چیه؟!

خب آدرس درخواست ها در رتروفیت به دو بخش baseUrl یا آدرس پایه و endPoint یا نقطه انتهایی تقسیم میشن.

یعنی چی ؟ فرض کنید ما یک آدرس api بصورت زیر داریم

https://tejariapp.com/apis/getSomeUsers.php

خب مشخصا تا بخش tejariapp.com که دامنه هستش و تغییر نخواهد کرد

بخش apis هم احتمالا یک فولدر هست که api هارو داخلش قرار دادیم پس احتمال زیاد این بخش هم تغییر نخواهد کرد

اما قسمت اخر که getSomeUsers.php هست اسم api ما هستش و طبیعتا با بقیه api ها تفاوت داره

قسمت هایی از آدرس که برای تمام api های استفاده شده ثابت هستند به baseUrl درخواست رتروفیت تبدیل میشود

و قسمت هایی که برای هر درخواست رتروفیت متغیر است به endPoint تبدیل خواهد شد.

بخش endpoint درخواست رتروفیت داخل حاشیه نوشت ها یا annotation ها قرار میگیره و بخش baseUrl زمان ساخت ریکوست یا درخواست رتروفیت ایجاد میشه

نکته قابل توجه اینکه تمامی endPoint ها یا به اصطلاح تمامی درخواست ها داخل یک interface قرار میگیرن که با یک آبجکت رتروفیت میتونیم به این درخواست ها اشاره و هرکدام را فراخوانی کنیم

### POST annotation

انوتیشن @POST این امکان را به ما میدهد که یک سری پارامتر یا داده نیز همراه با درخواست رتروفیت به api ارسال کنیم (بشدت پرکاربرد) برای مثال

|  |
| --- |
| @POST("createUser.php") |

**نکته:** انوتیشن POST باید حتما یک انوتیشن دیگه با عنوان زیر بهش اضافه کنیم

|  |
| --- |
| @FormUrlEncoded |

اما این annotation چیکار میکنه ؟!  
خب ما میتونیم درخواست های رتروفیت خودمون رو به دوصورت زیر تقسیم کنیم(در مواقعی که میخوایم دیتا همراه با درخواست رتروفیت ارسال شه)  
form-encoded  
برای ارسال دیتا بصورت کلید -> مقدار  
multipart data  
آپلود دیتاهای حجیم مانند عکس و تصویر

که در نهایت درخواست رتروفیت ما بصورت زیر خواهد شد

|  |
| --- |
| @FormUrlEncoded  @POST("createUser.php")  Call<User> updateUser(@Field("first\_name") String first, @Field("last\_name") String last); |

خب همونطور که میبینید ما یک سری پارامتر با انوتیشن@Field داریم پس یعنی درخواست ما جزو form-encoded هاست (کلید -> مقدار ) که کلید first\_name و last\_name و مقادیر همان متغیرهایی ست که مقادیرشان را ما زمان فراخوانی به آنها پاس میدهیم

### PUT annotation

ادامه بحث قبل ما متوجه شدیم که دو حالت ارسال داده به سرور داریم یکی داده های form-encoded و دیتاهای حجیم multipart  
حالا میخوایم نمونه ای از درخواست رتروفیت رو برای داده های multipart ببینیم که با انوتیشن @P درخواست رو مشخص میکنیم

|  |
| --- |
| @Multipart  @PUT("user/photo")  Call<User> updateUser(@Part("photo") RequestBody photo, @Part("description") RequestBody description); |

خب همونطور که میبینید ما از انوتیشن @Part برای مشخص کردن کلید و داده های حجیم استفاده کردیم

خب حالا که یاد گرفتیم یه اینترفیس میسازیم و کد های خودمون رو داخل اون مینویسیم

interface ApiService {

@GET("region/europe")

suspend fun getCountries(): Response<Countries>

}

بعدش ریپازیتوریمونم به برنامه اضافه میکنیم

class Repository(private val apiService: ApiService) {

suspend fun getCountries() = apiService.getCountries()

}

حالا میتونید به راحتی از سرور برای دریافت اطلاعات استفاده کنید

اینترفیس برای این اومد که بتونی با استفاده ازش قرار داد هایی تعیین کنی که هم مطمئن بشی از انجام شدن اون کار و هم ساختار بهتری داشته باشی بر فرض مثال ۳ تا درگاه پرداخت داری درگاه پرداخت هایی که استفاده شده : ملت ،شاپرک ، زیرین پال هس ایده ای که برای پیاده سازی این هس اینه که بیایی یه کلاس به نام درگاه پرداخت بزنی و سه تا متد بزنی براش و هر جا که خواستی بیایی از کلاسش یه اینستنس بگیری و ازش استفاده کنی حالا مشکلی توی این هس که از قوانین کد تمیز درست نیس و قوانین solid رو بهم میزنه راه حل درست چیه راه درس این هس که تو میدونی که درگاه پرداخت ها همه شون کار هایی که انجام میدن یکی هس برای همین میایی از اینترفیس استفاده میکنی به این صورت که در ابتدا یه کلاس درگاه پرداخت مینویسی توش یه متد مینویسی که بیاد علمیات وصل شدن به درگاه پرداخت رو انجام بده و یه اینترفیس میزنی به نام درگاه پرداخت ها که مثلا بر فرض مثلا اسم درگاه پرداخت رو میگیره وصل میشه حالا هربار که نیاز بود فقط میری اینترفیس رو کال میکنی که بره به درگاه پرداخت وصل بشه

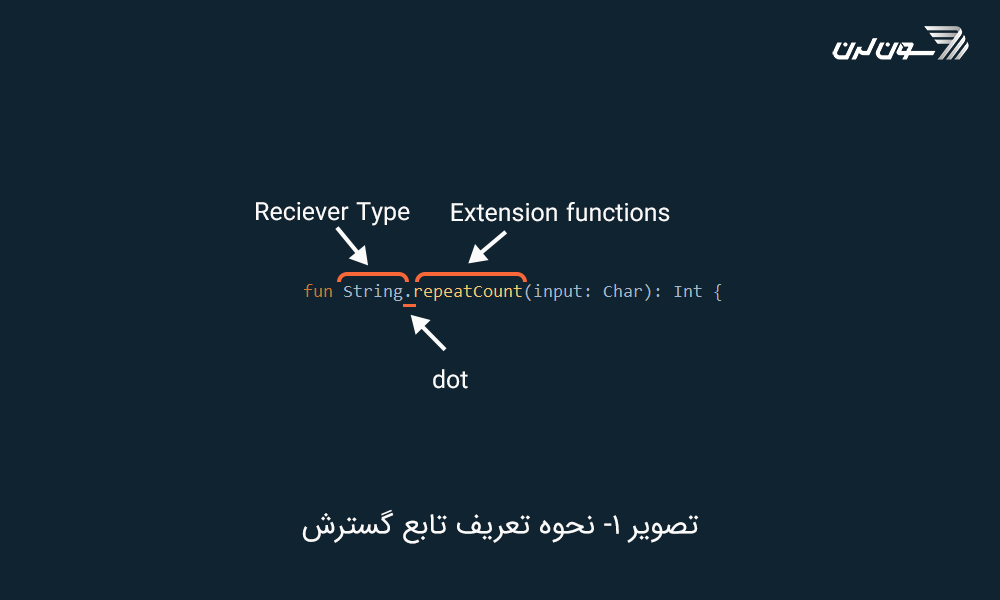
برای ابسترکت مفهومش یکمی متفاوته تفاوت در اینجاس که تو میتونی کلاس های خودت رو مدیریت بکنی اما توی اینترفیس نمیتونی وقتی اینترفیس روی یه کلاس implement میشه متد هایی که هس باید کال بشه اما توی abstract به این شکل نیس شما میتونی تعیین کنی که برای کلاس هایی که extend میشن کدوم متد ها override بشه یا میتونی اصلا بگی کلاسی override نشه اما در مورد مفهوم ابسترکت باید بگم که یک مفهوم کلی هس بر فرض مثال گوشی که دستمون هس ما از قطعات درون اون خبر نداریم یا نمیدونیم کرنل لینوکس توی دیوایس داره به چه صورت کار میکنه یا مثلا وقتی روی دکمه کپی توی گوشی میزنیم عملا چه اتفاقی میوفته صرفا فقط از دانش های قبلی خودمون میدونیم که فلان متن یا فلان عکس کپی میشه توی یه جای از گوشی و میتونی رو پیست کنیم توی جای دیگه در واقعه دید کلی در رابطه با اون داریم حالا توی اندروید نمونه اش لایف سایکل Activity هس که متد های oncreate ,onstop ، و ... داره ما فقط ازشون استفادم میکنیم و برامون فرقی نمیکنه که اون پشت داره چه اتفاقاتی میوفته

# **extension function:**

همان‌طور که گفتیم برای افزودن یک تابع (function) یا قابلیت‌های یکی از کلاس‌های ارائه شده توسط زبان کاتلین مانند String و List و Toast و … می‌توانیم از ویژگی Extension در این زبان استفاده کنیم. به تابعی که به کلاس موجود اضافه می‌شود و توسط آن ویژگی جدید به آن کلاس اضافه می‌شود، تابع گسترش (extension function) می‌گویند. و کلاس مرجعی که تابع گسترش به آن اضافه شده است Receiver Type نامیده می‌شود.

## **ایجاد یک تابع گسترش:**

ایجاد یک تابع گسترش بسیار ساده است و تفاوت چندانی با ایجاد سایر توابع در پروژه ندارد. برای افزودن تابع گسترش به یک کلاس (Receiver Type) کافی است، تابع مورد نظر را در پروژه خود به صورت top level تعریف کنیم و هنگام تعریف تابع ابتدا نام کلاسی که قصد گسترش آن را داریم بیاوریم و سپس یک نقطه گذاشته و نام تابع گسترش را وارد کنیم و تعریف ورودی‌ها و خروجی و بدنه‌ی تابع همانند یک تابع معمولی در کاتلین است. تصویر پایین نحوه‌ی تعریف تابع گسترش و اجزای آن را نشان می‌دهد:



به عنوان مثال قصد داریم تابع گسترشی برای کلاس String ( توسط خود زبان کاتلین ارائه می‌شود.) بسازیم که در ورودی آن یک حرف دریافت و تعداد تکرار آن در متن را به ما برگرداند. برای انجام این کار تابع repeatCount را به شکل زیر با پیشوند String تعریف می‌کنیم:

fun String.repeatCount(input: Char): Int {

var repeatCount = 0

for (c in this) {

if (c.toLowerCase() == input.toLowerCase()) repeatCount++

}

return repeatCount

}

نکته: جهت فراخوانی تابع گسترش، در هنگام ایجاد یک تابع گسترش دقت کنید که آن را به صورت top level تعریف کنید(به طور مستقیم در یک فایل تعریف شود و درون کلاس دیگری قرار نگیرد).

## **فراخوانی تابع گسترش:**

فراخوانی تابع گسترش هم بسیار ساده است و مطابق آنچه در تصویر 2 می‌بینید، همانند سایر توابع داخلی کلاس کافی است بعد از Receiver Object یک نقطه بگذارید و سپس نام تابع گسترش را وارد کنید.



مطابق آنچه در تصویر 2 مشاهده می‌کنید برای استفاده از این تابع و برای اینکه تعداد تکرار حرف ‘d’ را در عبارت “Android studio” بدست آوریم، به صورت زیر عمل می‌کنیم:

fun main() {

val strTest = "Android studio"

val dCount :Int= strTest.repeatCount('D')

println("D repeat $dCount times in $strTest")

}

D repeat 3 times in Android studio خروجی

نکته‌ی قابل توجه هنگام تعریف از توابع گسترش این است که درون بدنه‌ی تابع می‌توانید به property و توابع کلاس مرجع (در مثال ما String) **به طور مستقیم** دسترسی پیدا کنید. **درست مثل اینکه این تابع گسترش داخل همان کلاس تعریف شده است.**اما **نمی‌توانید** به توابع و متغیرهای private و protected که داخل کلاس مرجع تعریف شده‌اند، دسترسی پیدا کنید.

# Object Extension

زبان کاتلین یک روش دیگری برای پیاده سازی مفهوم static جاوا دارد. این روش با استفاده از عبارت companion object قابل استفاده است.

با استفاده از این مکانیسم ، ما میتوانیم یک آبجکت از یک کلاس بسازیم و سپس فقط با اشاره به نام متود میتوانیم آنرا صدا بزنیم.

در مثال زیر نمونه ای از companion object در زبان کاتلین میبینیم :

Live Demo

fun main(args: Array<String>) {

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

}

class A {

companion object {

fun show():String {

return("You are learning Kotlin from avasam")

}

}

}

نمونه کدهای کاتلین بالا نتیجه ی خروجی زیر را در پی خواهد داشت :

Heyyy!!! You are learning Kotlin from avasam

مثال بالا شبیه به مبحث static در زبان برنامه نویسی جاوا است ، در یک زمان واحد ما یک آبجکت از همان کلاس بعنوان یکی از اعضای همان کلاس میسازیم.

در حقیقت با این کار شما در حال گسترش یک شی از همان کلاس هستید تا از بعضی از توابع استفاده کنید.

# Lambda Function

## Lambda

Lambda یک تابع سطح بالا است که به طور قابل توجهی کد های تکراری صفحه را هنگام بیان نمودن یک تابع و تعریف آن، کاهش می دهد.

در مثال زیر یک لامبدا میبینیم که در وریبل تعریف میشود

fun main(args: Array<String>) {

val mylambda :(String)->Unit = {s:String->print(s) }

val v:String = "TutorialsPoint.com"

mylambda(v)

}

در کد بالا، ما lambda  خود را با عنوان “mylambda” ایجاد کرده و یک متغیر را به این lambda  منتقل کرده ایم.  
این lambda یک String است و حاوی مقدار “TutorialsPoint.com” می باشد.

## inline function

مثال فوق الگوی بیان lambda را نشان می دهد که می توان در برنامه کاتلین استفاده نماییم.  
می توان یک lambda را به یک تابع دیگر منتقل نمود تا خروجی مورد نظر را به دست آورد و به این ترتیب، تابع calling function را به یک تابع inline function تبدیل نمود.  
به مثال زیر توجه کنید.

fun main(args: Array<String>){

val mylambda:(String)->Unit = {s:String->print(s)}

val v:String = "TutorialsPoint.com"

myFun(v,mylambda) //passing lambda as a parameter of another function

}

fun myFun(a :String, action: (String)->Unit) { //passing lambda

print("Heyyy!!!")

action(a)// call to lambda function

}

# ویژگی Smart casts

وقتی که بحث cast کردن میشه زبان کاتلین یکمی باهوش عمل میکنه.در خیلی از موارد نیاز نیست که ما خودمون عملیات cast کردن و انجام بدیم بخاطر اینکه کاتلین هنگام کامپایل شدن این کار و خود کار انجام میده.به مثال زیر توجه کنید:

fun demo(x: Any) {

if (x is String) {

print(x.length) //

}

}

دستور is در کاتلین مشابه دستور instanceof در جاوا هست.در کد بالا اگر مقدار x برابر با null نباشد تبدیل به String می شود به طور خودکار.

خب حالا فهمیدیم که کامپایلر هوشمنده و بعضی جاها خودش cast رو انجام میده، ولی اگه بعضی جاها ما خودمون بخوایم این casting رو انجام بدیم چیکار باید بکنیم؟

درواقع با کلیدواژه “as” این کار رو انجام میدیم

var input : Any = 10

fun main(args: Array<String>) {

val str = input as? String

println(str?.length)

}

اینجا گفتیم که میدونیم input از جنس Any هستش، ولی اینجا میخوایم به String کستش کنیم و طولش رو برام بگیریم. تنها کاری که کردیم اون وسط این بود که از کلیدواژه as استفاده کردیم.

# مقدار دهی متغییر ها با lateinit و **lazy**

## کلمه کلیدی lateinit

با این کلمه کلیدی ما میتونیم یه متغییر را تعریف کنیم و بعدا مقدار دهی بکنیم

**نکته مهم در استفاده از lateinit این است که متغییر به صورت mutable باید استفاده شوند که در kotlin با استفاده از کلمه کلیدی var معرفی میشن که در بدنه کلاس تعریف شده اند**

public class Test {

lateinit var mock: Mock

@SetUp

fun setup() {

mock = Mock()

}

@Test

fun test() {

mock.do()

}

}

## **کلمه کلیدی lazy**

**کلمه کلیدی lazy در واقع یک تابع است که ورودی آن lamda هست که ما در آن شیوه مقدار دهی شدن متغییر رو تعریف میکنیم , در ادامه تابع lazy یک نمونه از lazy را برگشت میدهد مه به عنوان یک نماینده جهت مقدار دهی متغییر استفاده میشه.**

**برای اولین باری که get متغییر فراخوانی میشود این تابع فراخوانی شده و متغییر ما را در لحظه مورد استفاده مقدار دهی میکنه! و در بقیه موارد استفاده مقدار قبلی خود را حفط کرده و دوباره مقدار دهی نمیشه و همان مقدار اولیه که برای مقدار دهی استفاده شده را باز میگردونه.**

public class Example {

val name: String by lazy { “mohsen aliakbari” }

}

## کی lazy کجا lateinit

حال بعد از آشنا شدن با این ویژگی های مهم بهتره مواد کاربرد اون رو دقیق بدونیم

* کلمه کلیدی lazy فقط برای متغییر های تغییر ناپذیر یا immutable که در kotlin با val تعریف میشن استفاده میشود و هم چنین برای کلمه کلیدی lateinit از var چرا که به متغییر های final کامپایل نمیشه.
* کلمه کلیدی lateinit در تعریف متغییر هر جایی که بخواهیم مورد استفاده قرار بدیم‌ و اگه از قبلا بدونیم که قراه کجا متغییر ما مقدار دهی بشه بهتره از lateinit استفاده کنیم.

# List ها

## listOf

در لیست ها اضافه یا کم کردن به لیست امکان پذیر نیست برای اینکار باید از mutableListOf استفاده کنیم

val list = listOf(1,2,3,4,5)

## List mutableListOf

val mylist = mutableListOf (1,2,3,4,5)

for(item in mylist) { println(item) }

mylist.add(65)

mylist.remove(2)

در mutableListOf میتوانیم با .add به لیستمون اضافه کنیم و با remove میتوانیم مقدار مورد نظر را حذف کنیم

## arrayListOf

در آرایه چون اندازه آرایه ما ثابت است نمی توانیم مقداری را اضافه  ویا حذف کنیم .به خاطر همین امر از ArrayList استفاده می کنیم . چون اندازه متغیر دارد .

val array= arrayListOf(1,2,3,4,5,6,7,8,9,10)

array.add(11)

for(item in 0 until array.size)

}

println(array[item])

println("\t")

}

## نمایش مقادیر داخل آرایه ها

### سایز آرایه خودمون رو با **array.size** بدست بیاریم

val array= arrayOf("ali",1,2,3)

for (item in 0 until array.size)

{

print(array.get(item).toString()+"\t")

}

ali 1 2 3

### آرایه را به for می دهیم و آیتم های آرایه را تا آخر آرایه می گیرد و برای ما نمایش می دهد .

val array= arrayOf("ali",1,2,3)

for (item in array)

{

print(item.toString()+"\t")

}

ali 1 2 3

### تعدادی از متد های کاربردی در آرایه ها (Array) :

* **()Last**:  برای زمانی هست . که ما می خواهیم آخرین مقدار آرایه مان را بدست بیاوریم.
* **()First:**از این  برای بدست آوردن اولین مقدار آرایه استفاده می کند.
* **()isEmptyو ()isNotEmpty**:این متد ها برای این است که بفهمیم آرایه ما خالی هستو یا نه :
* **متد ()Set :** برای زمانی است که ما بخواهیم یک مقدار از آرایه را عوض کنیم . متد  ()Set دو پارامتر از ما می گیرد اول شماره خانه یا index و پارامتر دوم نیز مقدار آن می باشد .
* **متد ()add :** از این متد برای اضافه کردن یک مقدار به لیست خود اسفاده می کنیم .
* **متد ()remove :  این متد برای پاک کردن یک مقدار در ArrayList می باشد**
* متد ()clear : مقادیر داخل آرایه لیست را به صورت کامل پاک می کند.

## تفاوت set و list و arraylist

* اندازه ی لیست آرایه ها یا همان ArrayList ها متغیر می باشد . یعنی می توانیم به تعداد نامحدود مقادیر در آن اضافه کنیم .ولی برای آرایه ها این گونه نیست و آرایه ها (Array) دارای اندازه ثابت می باشد .
* ما می توانیم در **ArrayList** ها یا همان لیست آرایه هر تعدای مقادیر اضافه کنیم ولی این کار در آرایه ممکن نیست چون دارای اندازه ثابت می باشد .
* یکی از نقاط قوت آرایه نسبت به ArrayList این استکه چون آرایه ها دارای اندازه ثابت هستن و موقع ایجاد کردن آرایه حافظه به آن اختصاص داده می شود و برای همین امرکار با آرایه ها درکاتلین هم سریع تر است .
* **خطای ArrayOutOfBoundException** در لیست آرایه ها یا ArrayList ها اتفاق نمی افتد .یعنی ما اگر یک آرایه تعریف کرده باشیم ولی خالی بزاریم آونو حافظه را هدر دادیم .
* list:میزان حافظه ی تخصیص داده شده ایستاست
* arraylist:میزان حافظه ی تخصیص داده شده پویاست
* در لیست میشود دو مجموعه را مقایسه کرد ولی در ارایه نمیشود

val l1 = listOf("a")

val l2 = listOf("a")

var x = (l1 == l2) // => true

val a1 = arrayOf("a")

val a2 = arrayOf("a")

var y = (a1 == a2) // => false

| **ویژگی** | **Array** | **List** |
| --- | --- | --- |
| Implementation  پیاده سازی | ارایه با استفاده از کلاس  **Array<T>**  پیاده سازی میشود | **از اینترفیس List<T> یا MutableList<T>** برای پیاده سازی لیست در کاتلین استفاده میشود |
| Mutable  قابل تغییر بودن | **Array<T>**  قابل تغییر است یعنی مقادیر ان میتوانند تغییر کنند | **List<T>** ذاتن تغییر ناپذیر است ولی میتوانیم با استفاده از **MutableList<T>**  یک لیست قابل تغییر ایجاد کنیم |
| Size  سایز | آرایه اندازه ثابتی دارد. اندازه آن نمی تواند کم و زیاد شود. | **MutableList<T>** دارای توابع 'add' و 'remove' است به منظور افزایش و یا کاهش اندازه **MutableList** |
| Performance  کارایی | از آن برای عملکرد بهتر استفاده کنید، زیرا آرایه برای انواع داده های اولیه مختلف مانند IntArray[]، DoubleArray[] بهینه شده است. | برای دسترسی بهتر در کد از آن استفاده کنید. از آنجایی که اندازه در طبیعت پویا است، بنابراین مدیریت حافظه خوب است. |

val a = arrayOf(1, 2, 3)

a[0] = a[1] // OK

val l = listOf(1, 2, 3)

l[0] = l[1] // کار نمیکند

val m = mutableListOf(1, 2, 3)

m[0] = m[1] // OK

val a = arrayOf(1, 2, 3)

println(a.size) // همیشه برای این ارایه 3 خواهد بود

val l = mutableListOf(1, 2, 3)

l.add(4)

println(l.size) // 4

val a: Array<Number> = Array<Int>(0) { 0 } // کار نمیکند

val l: List<Number> = listOf(1, 2, 3) // OK

#### تفاوت بین List و Set :

1. [List](https://javapro.ir/news_126020-utab) می‌تواند عناصر تکراری داشته باشد اما Set عناصر یکتا و غیر تکراری می‌گیرد.
2. [List](https://javapro.ir/news_126020-utab)  یک مجموعه مرتب شده است که ترتیب درج را حفظ می کند، به این معنی که با نمایش محتوای لیست، عناصر را به همان ترتیبی که در [List](https://javapro.ir/news_126020-utab)  درج شده اند نمایش می دهد.
3. Set یک مجموعه نامرتب است، هیچ نظمی را حفظ نمی کند. پیاده سازی های کمی از Set وجود دارد که نظم را حفظ می کند مانند LinkedHashSet (عناصر را به ترتیب درج حفظ می کند).
4. [List](https://javapro.ir/news_126020-utab)  به هر تعداد عنصر تهی اجازه می دهد. Set حداکثر می تواند تنها یک عنصر تهی داشته باشد.

|  |  |  |
| --- | --- | --- |
| **Factor** | **List** | **Set** |
| آیا عناصر گروه بندی مرتب شده است؟ | YES | NO |
| دسترسی موقعیتی را با index فراهم می کند؟ | YES | NO |
| آیا می توان عناصر تکراری را ذخیره کرد؟ | YES | NO |
| آیا می توان چندین عنصر تهی را ذخیره کرد؟ | YES | NO |
| زیرمجموعه ها: | ArrayList, LinkedList, Vector, and Stack | HashSet and LinkedHashSet |

# HashMap

ببینین هش مپ ها در کاتلین یه جورایی کارکردی مانند لیست ها دارند ولی با این تفاوت ما برای داده های خودمون آمدیم از Key استفاده کردیم . یعنی هر داده ایی که در HashMap ذخیره می شود. دارای یک کلید (Key) می باشد . وما برای اینکه بخواهیم به اون مقدار دسترسی پیداکنیم . باید از کلید مخصوص به آن استفاده کنیم . مثلا من می خواهم در HashMap یک رشته مثلا “Mykotlin” را ذخیره کنم . باید برای آن یک Key نیز بزارم تا موقعی که می خوام مقدار را برای من برگرداند . با دادن کلید (Key) به آن رشته دسترسی پیدا کنم .

## ویژگی ها

* از ساختار Key و Value برای پر کردن مقادیر داخل map خود پشتیبانی می کند.
* در زمانی که ما می خواهیم داده های خود را ذخیره می کنیم. از Thread های جداگانه و مختلف استفاده می کند.
* موقع ذخیره سازی اطلاعات بسیار سریع عمل می کند.
* Key یا همون کلید هایی که ما استفاده می کنیم . دارای hashcode هایی است که در موقع جستوجو از آن استفاده می کند.
* ما در HashMap خطای Memory Leak نداریم.
* ما در HashMap ها از یک Hashtable به عنوان یک جدولی برای ذخیره اطلاعاتمون استفاده می کنیم.

## کار با HashMap

در داخل HashMap یک دستور به نام to وجود دارد و موقع جایگذاری مقادیر وKey از آن استفاده می کنیم . و اینگونه است که مقداری که سمت چپ to هست همون Key ما هست . و مقداری که در سمت راست آن قرار دارد همون مقدار ما می باشد . همه این چیز هایی که براتون گفتم را در کد زیر برای شما آورده ام .

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", "myKey" to 12)

در کد بالا ما یک هش مپ تعریف کرده ایم که در map خود مقادیر جای داده است که هر کدام از این مقادیر دارای یک key است ولی نکته که در کد بالا هست . این است که ما برای key از رشته هم استفاده کرده ایم و هم عدد و همچنین برای مقادیر نیز این گونه است . یعنی در اینجا DataType برای ما مهم نیست . هر ورودی میتواندبگیرد . ولی اگر شما بخواهید می توانید مانند مبحث آرایه می توانیم DataType های خودمون را محدود کنیم . یعنی بگیم کلید ما Int باشه و مقادیر آن نیز رشته باشد . برای اینکه راحتر بفهمید مثال زیر را مشاهده کنید.

val hashmap= hashMapOf<Int,String>(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

print(hashmap.get(2))

ali

## دستور get برای نمایش دادن مقادیر داخل HashMap

برای نمایش دادن یک مقدار HashMap در کاتلین از این دستور به کار برده می شود . به این صورت که ما Key را به ()get می دهیم . و مقدار مربوط به آن را برمی گرداند. مانند مثال زیر :

val hashmap= hashMapOf<Int,String>(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

print(hashmap.get(2))

ali

ما میتونیم کلیه اطلاعات داخل HashMap را به وسیله یک ساختار تکرار نمایش دهیم . در مثال های پایین ما نمونه هایی از این نمایش ها را برای شما کد نویسی کرده ایم. تا با نحوی کار با حلقه ها در HashMap نیز آشنا شوید .

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

for (item in hashmap.keys) {

print(hashmap.get(item).toString()+"\n")

}

ali

myKey

farshid

reza

در حلقه بالا کلید های ما رو می گیرد . و مقادیر مربوط به آن را برای ما برگرداند. ما می تونیم در کد بالا values ها رو مستقیم نمایش دهیم . مانند مثال زیر :

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

for (item in hashmap.values) {

print(item.toString()+"\n")

}

ali

myKey

farshid

زمانی هم ممکن است ما بخواهیم هم مقادیر و هم Key هارو با هم نمایش دهیم . که در این صورت باید مثل مثال زیر عمل کنید.

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

for (item in hashmap) {

print(item.toString()+"\n")

}

2=ali

12=myKey

5=farshid

7=reza

## استفاده ازدستور entries برای نمایش مقادیر Hashmap در کاتلین

این دستور به گونه ایی است که مقدار Key و مقادیر را در داخل دوتا مقادیر در حلقه ما قرار می دهد . یعنی دو مقدار بر می گرداند. بزارید خیلی آسون برای شما در مثال زیر توضیح بدم .

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

for ((key ,values) in hashmap.entries) {

print("$key = $values \n")

}

2 = ali

12 = myKey

5 = farshid

7 = reza

## استفاده ازدستور put برای اضافه کردمقادیربه Hashmap در کاتلین

اندازه های HashMap ثابت نیست . اگر یادتون باشه ما در آرایه ها همون اول یک آرایه تعریف می کردیم . و لحظه اول نیز برای آن یک حافظه اختصاص میدادیم . ولی Hashmap این گونه نیست و ما می توانیم هر موقع خواستیم مقادیری به آن اضافه می کنیم. مثال :

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

hashmap.put(3,"key1")

hashmap.put(4,"key2")

hashmap.put(6,"key3")

for ((key ,values) in hashmap.entries) {

print("$key = $values \n")

}

2 = ali

3 = key1

4 = key2

5 = farshid

6 = key3

7 = reza

12 = myKey

## استفاده از دستور()removeبرای حذف کردن یک مقدار HashMap

ما با استفاده از دستور ()Put و با استفاده از یک کلید می توانیم مقدار مورد نظر را در HashMap حذف کنیم. مانند نومنه کد زیر :

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

hashmap.remove(2)

hashmap.remove(5)

for ((key ,values) in hashmap.entries) {

print("$key = $values \n")

}

12 = myKey

7 = reza

## استفاده از دستور ()replace برای جایگزینی یک مقدار در به جای مقداری دیگر در HashMap

شاید نیاز داشته باشید یکی از مقادیری که وارد کردید را عوض کنید در این زمان ما باید از دستور ()replace استفاده کنیم.

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

hashmap.replace(2,"farshid")

hashmap.replace(7,"AliReza")

for ((key ,values) in hashmap.entries)

{

print("$key = $values \n")

}

2 = farshid

12 = myKey

5 = farshid

7 = AliReza

## به کار بردن دستور ()size برای نمایش دادن سایز Hashmap در کاتلین

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

print(hashmap.size)

## استفاده از دستور ()contains در Hashmap

کاربرد این دستور برای این است که ما می خواهیم بدانیم . Key یا کلیدی که در نظر داریم در HashMap ما وجود دارد یا نه .

val hashmap= hashMapOf(2 to "as" , 5 to "farshid", 7 to "reza", 12 to "myKey")

print(hashmap.contains(21))

false

## استفاده از دستور clear برای پاک کردن کل HashMap

از این متد زمانی استفاده می کنیم . که میخواهیم HashMapرا کلا پا کنیم .

val hashmap= hashMapOf(2 to "ali", 5 to "farshid", 7 to "reza", 12 to "myKey")

print("Size = "+hashmap.size.toString()+"\n")

hashmap.clear()

print("Size = "+hashmap.size.toString())

Size = 4

Size = 0

## تفاوت HashMap با Map در کاتلین چیست ؟

HashMap و Map مانند همند . تفاوت اصلی آن ها این می باشد. که Map در موقع ساخته شدن ، همون لحظه بهش حافظه اختصاص داده می شود. یعنی دارای حافظه ثابت می باشد . و اینکه دیگه نمی توانیم در طول کارمون بهش یک مقداری اضافه کنیم(Put ) . ویا اینکه یک مقداری از حذف کنیم (()remove). و برای این کار ها ما از HashMap استفاده می کنیم.