Null Safety . Java Vs Kotlin

→ In java, we dont know whether a variable will give value or a null value, If that will give null value, then it will give null pointer Exception. So we need to specify or give check that variable not equal to null, then only execute statement.

→ In Kotlin, If we assign null to nullable non nullable variable then in IDE only it will give Error.
So we need to specify nullable or make that variable as nullable before assigning to null.

Java

```
Class Main{
  public static void main(String[] args)
  {
    String hi = "hello";
    printlength(hi);
    hi = null;
    printlength(hi); }
public static void printlength(String param)
  {
    if(param ≠ null)
      { sopln( param.length());
      }
  }
}
```

Kotlin

```
fun main() {
    var hi: String? = "Hi"
    printlength(hi)
    hi = null
}

fun printlength(param: String?) {
    if(param ≠ null) {
    println(param.length)
    }
```

use if you want to operate on a non null obj

use if you want to just execute lambda expression on nullable obj & avoid NPE

If you want to operate on a nullable obj and avoid NPE use run

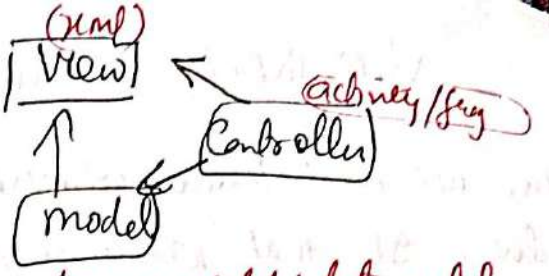
with	let	run
Return → lambda result Content obj → this	Return → lambda Content obj → it	Return: lambda Content obj → this

use if you want to modify an obj

apply	also
Return: Content obj Content obj: this	Return: Content obj Content obj: it

use if you want to do some additional obj configuration oprs

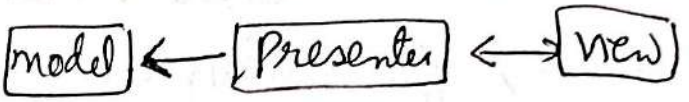
MVP Vs MVVM



MVC model view Controller

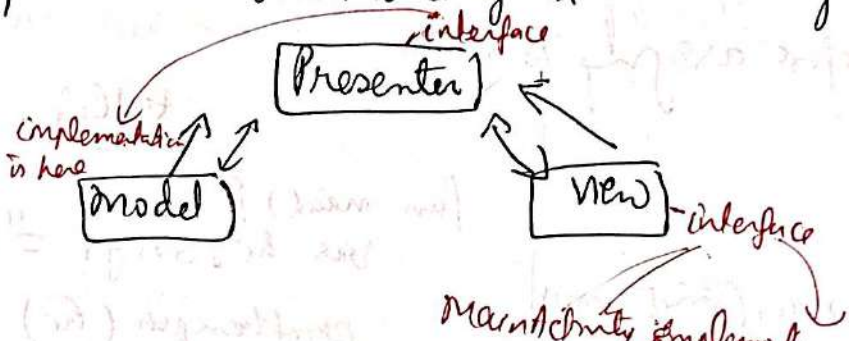
xml will tell to Controller that (view) btn is pressed. Now Controller will decide what action to take place.

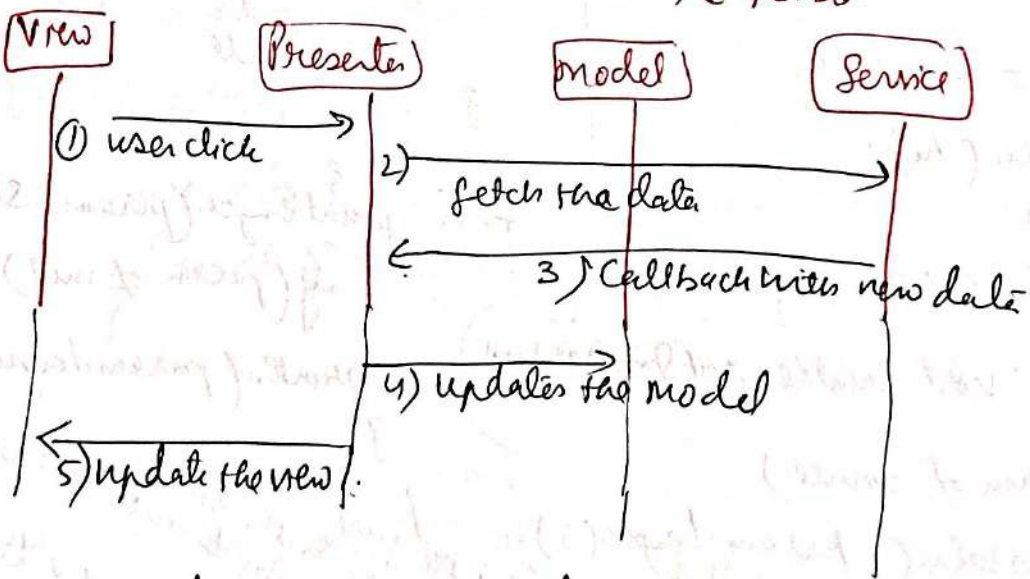
MVP model View Presenter



→ There is tight coupling b/w Presenter & view.

→ It Separates the datamodel from view through presenter.





→ MVP makes it Easier to test your presentation logic.

→ huge amt of Interfaces for interaction b/w layers.

→ Code size quite Excessive.

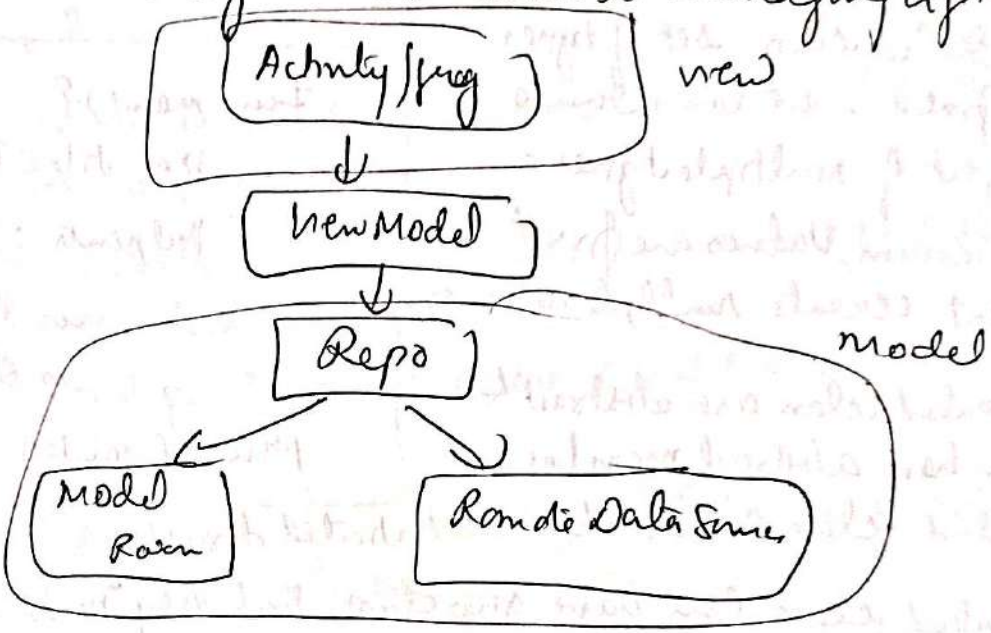
MVVM (model view ViewModel).

→ It supports two-way data binding b/w view and viewModel

→ MVVM Can map many views to one ~~view~~ View Model. ViewModel has no reference to the view while in MVP the view knows the presenter.

MVVM Advantages

1) UI Components are kept away from business logic.
2) Business logic is kept away from the database ops.
3) Easy to understand and read.
4) A lot less to worry when it comes to managing life-cycle events.



Differences b/w data class and normal class

→ A class is a blueprint for an object.
→ A data class is a simple class used to hold data/state and contains standard functionality.
→ a data keyword is used to declare data class. It reduces boilerplate code. no getter setter ~~see~~ method needed to define. It is already there by default.
→ A data class must be declared with at least one primary constructor parameter which is declared with var or val. A normal class can or cannot contain primary constructor
→ A data class cannot be extended by another class. They are final classes by default.
→ A data class cannot be sealed, open, abstract or inner.

Sealed class Vs Enum

```
Enum class DAY(val no:Int)
{ SUNDAY(1),
  ⋮
}
```

```
Sealed class Tile {
}
class Red(val type:Str, val points:Int) : Tile()
class Blue(val points:Int) : Tile()
```

→ In Sealed class, set of types are fixed. we can create object of multiple types.
In Enum, values are fixed. can't create multiple instances.

→ Sealed class are abstract & can have abstract members.

```
fun main() {
    val tile : Tile = Red("Mushroom", 2)
    val points : Int = when(tile) {
        is Red → tile.points * 2
        is Blue → tile.points * 5
    }
    Print(points)
```

→ Sealed class cannot be instantiated directly.

→ Sealed class can have sub class, but they must either be in the same file or nested inside of sealed class declaration.

RecyclerView

→ It is a View Group to show items in list. It is advance Version of list.
→ Efficient way to create a Scrollable List.
→ It has 2 methods, OnCreateViewHolder & OnBindViewHolder.
OnCreateViewHolder → creates object which store/hold views object.
Each view has one viewholder that stores view that it can be recycled.
OnBindViewHolder → will get one viewholder & one position that bind data in views according to given position
getItemCount → returns total no. of items in data source.
→ Adapter → It adapts your data in a format that can be consumed by recyclerview.
Recyclerview requests data from adapter & adapter provides the data inside OnCreateViewHolder, layoutInflater converts xml to java object

LayoutManagers → define how items are arranged in the recyclerview.
- → LinearLayoutManager → arranges items in row or column
- → GridLayoutManager → grid with specified column
- → StaggeredGridLayoutManager → irregular grid cell

How can you implement endless scrolling in a recyclerview using pagination?

Step1 → detect when the user has reached the end of the list by listening for scroll events.

Step2 → Trigger a data load (Eg. fetching the next page of data) when the end is reached.

Step3 → Add a loading indicator at the end of list while data is being loaded

Step4 → Append the new data to existing dataset and notify the adapter about the data change.

RecyclerView.OnScrollListener is used to achieve this.

```
OnScrolled{Method
}
```

How to call multiple APIs in Parallel and in sequence with the help of Rxjava operators.

```
(First observable
 .flatMap(second observable)
 .flatMap(third observable)
 .subscribe(
  {response -> }
  ,
  {error -> }
 )
```

flatMap opr for multiple observable opr sequentially

```
Zip(First Observable,
  Second observable,
  Third observable,
  Mapper)
 .subscribe(
  {response -> },
  {error -> }
 )
```

Zip opr for multiple observable opr parallelly

```
fun callThreeAPIsSequentially() {
    var list: MutableList<String> = ArrayList()
    DisposableManager.add(
        apiService.getAlbumPhotos()
            .subscribeOn(Schedulers.io())
            .flatMap { photos ->
                list.add(photos.toString())
                apiService.getFirstPost() }
            .flatMap { post ->
                list.add(post.toString())
                apiService.getComments() }
            .map { response ->
                response }
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                { result ->
                    list.add(result.toString())
                    Log.i(TAG, "list -> $list")
                    // log.i(TAG, "result -> $result")
                }
                { error ->
                    Log.i(TAG, "error -> $error.message)")}
            )
    }
```

determines the thread where observable starts its work

determines thread where downstream operations are executed

initiates subscription to the observable, triggering the execution of its logic.

```
fun callThreeAPIsParallely(){
    val errorPhotoList : MutableList<Photo> = ArrayList()
    errorPhotoList.add( Photo( 1, 2, "a", "b", "c"))

DisposableManager.add(
    Single.zip( apiService.getAllPhotos, onErrorReturn{errorPhotoList},
        apiService.getFirstPost,
        apiService.comments,
    Function3<List<Photo>, Post, List<Comment>, List<String>>
    { type1, type2, type3 →
            val list : MutableList<String> = ArrayList()
            list.add( type1[0].toString())
            list.add( type2.toString())
            list.add( type3[0].toString())
            list
        })
    .subscribeOn( Schedulers.io())
    .observeOn( AndroidSchedulers.mainThread())
    .subscribe(
        { result →
            Log.i( TAG, "result → $result")},
        { error →
            Log.i( TAG, "error → $error.message")}
    ))
}
```

How to call 2 apis parallely in coroutines

```
fun startLongRunningTaskInParallel() {
    viewModelScope.launch {
        val resultOneDeferred = async { doLongRunningTaskOne() }
        val resultTwoDeferred = async { doLongRunningTaskTwo() }
        val combineResult = resultOneDeferred.await() +
                            resultTwoDeferred.await()
    }
}
```

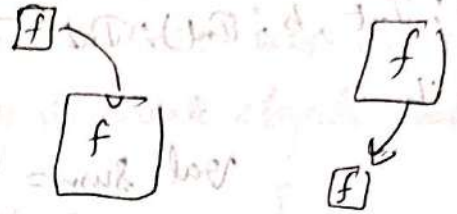
```
fun startLongRunningTaskInParallel() {
    viewModelScope.launch {
        launch { doLongRunningTaskOne() }
        launch { doLongRunningTaskTwo() }
    }
}
```

```
private suspend fun doLongRunningTaskTwo(): Int {
    return withContext(Dispatchers.Default) {
        delay(2000)
        return@withContext 10
    }
}
```

```
private suspend fun doLongRunningTaskTwo() {
    withContext(Dispatchers.Default) {
        delay(2000)
    }
}
```

Higher Order functions

⇒ The function that accepts functions as argument or return functions or both.




```
fun main() {
    var fn: (a: Double, b: Double) -> Double = ::sum      // func as type
    println (fn (2.0, 4.0))
    calculator (5.0, 5.0, ::sum)
}
fun sum (a: Double, b: Double) : Double {
    return a+b
}

fun calculator(a: Double, b:Double, gn: (Double,Double) -> Double){
    val result : Double = gn(a,b)
    println(result)
}
```

Why to send function as a parameter..? what is use? Why not to call function as normal.

Reason. Scenario where we send parameter and its functionality to Execute in runtime, if particular Condition persists. Execute parameter with functionality 1, 2.. etc.

Lambdas Expression

↓

is anonymous function no name.

```
fun main() {
    var fun: (a: Int, b: Int) -> Int = ::sum
    val lambda1 : (Int, Int) -> Int = { x: Int, y: Int -> x+y }
    val multilinelambda: () -> Int = {
        println("Hello")
        val a: Int = 2+3
        2
    }
    println(multilinelambda())
}
```

Hello
2

Lambdas

```
fun Sum(a: Int, b: Int): Int
{
    return a+b
}
```

```
val sum = {a: Int, b: Int -> a+b}
```

Variations.

```
val singleParamLambda : (Int) -> Int = {x: Int -> x * x}
val lambda2 : (Int) -> Int = {x -> x * x}

val sayHi : (String) -> Unit = {msg: String -> println("Hello $msg")}
val sayHi2 : (String) -> Unit = {msg -> println("Hello $msg")}
```

```
val simplifyLambda2 : (Int) -> Int = {it * it}

calculator(1, 2) {a, b -> a+b}
```

ViewModel and ViewModel factory.

→ ViewModel is class to store and manage UI-related data, it allow data survive in device configuration.

→ ViewModelFactory is class to instantiate and return ViewModel.

→ If we use ViewModel instance using ViewModel class, object will create everytime when fragment is re-created, for the better result we can instance ViewModel using ViewModelProvider.

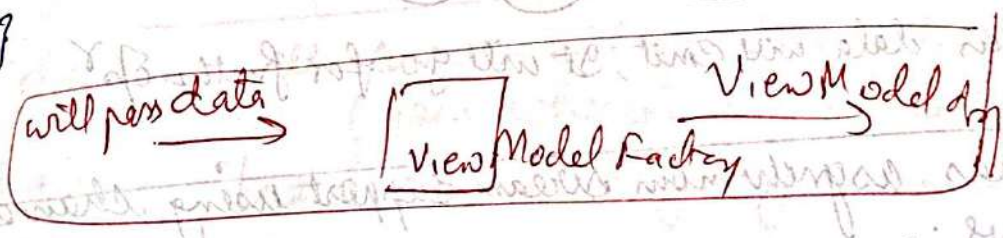
→ ViewModelProvider() return Existing ViewModel(if Exists) or create a new one if not Exist will create ViewModel instance in association with given scope.

viewModel = ViewModelProviders.of(this).get(GameViewModel::class.java)

ViewModelFactory. Sometimes the value must pass to ViewModel at initialization part. In that case use ViewModelFactory. ViewModelFactory uses factory to create objects.

Example

```
class MainActivity . . . {
    lateinit var mainViewModel : MainViewModel
    OnCreate
    {
    // mainViewModel = ViewModelProvider(this).get
                        (MainViewModel::class.java)
    mainViewModel = ViewModelProvider(this, MainViewModelFactory(10)).
                    get(MainViewModel::class.java)
    }

class MainViewModel(val initialValue : Int)
                    : ViewModel(){
    var count : Int = initialvalue
    fun increment(){
        count ++
    }
}
```




```
class MainViewModelFactory(val counter : Int) : ViewModelProvider.Factory{
    override fun<T : ViewModel?> create(modelClass : Class<T>): T{
        return MainViewModel(counter) as T
    }
}
```

Note: we can actually use hilt or dagger (DI) for that because in ViewModelFactory we can write some unwanted boilerplate code to do same functionality in dependency injection

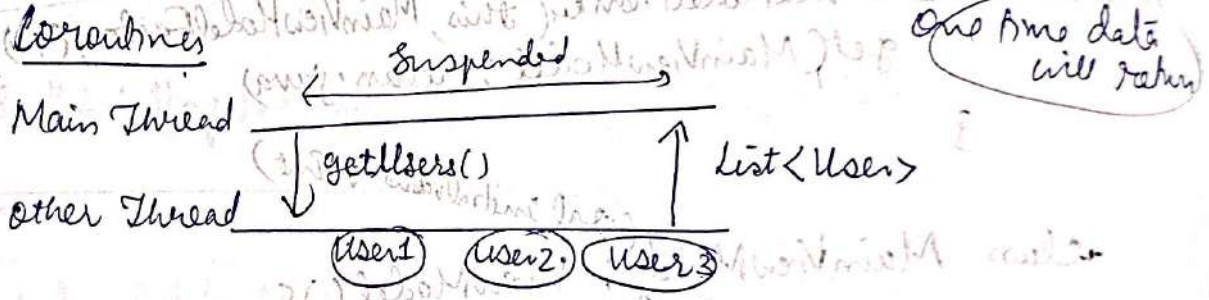
Flow

Coroutines save us from callback hell by giving us the possibility to run asynchronous code as if it was synchronous.
Flow takes this a step further by adding streams.

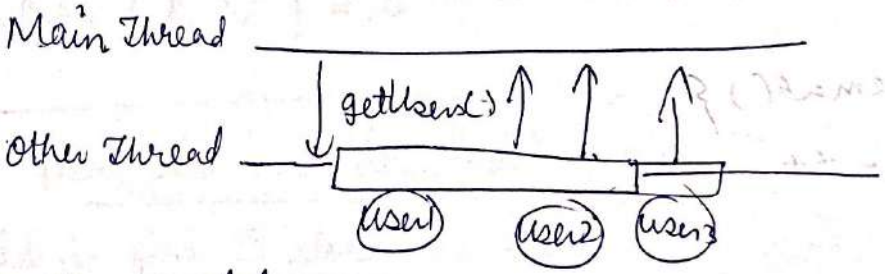
→ Flows are cold, so we receive the data once we start to collect it.

→ Coroutines helps to implement asynchronous, non blocking code. for this we use Suspend functions.

Coroutines



Streams.

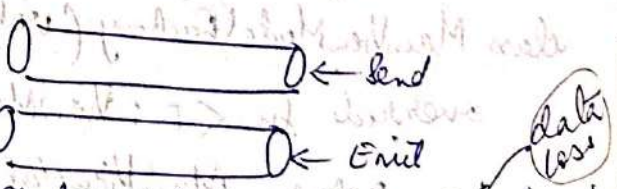


as soon as data will emit, it will go for further ops

→ Kotlin has asynchronous stream support using channels and flows.

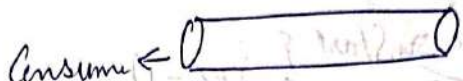
→ Channels (Send & Receive)

→ Flows (Emit & Collect)



Channels are Hot - (eg. Radio Station) (always data is emitted)
Flows are mostly cold (data emitted if there is consumer)
↳ will get data from start.

Observer ← ← Observable
(Consumer) (producer)



Producer is fast
Consumer is fast } bottleneck

↓

To resolve this here thread is not blocked we suspend coroutines at respective end.

```
class MainActivity -- . {
    onCreate . {

val job = GlobalScope . launch { val data : Flow<Int> = producer()
        data . collect {
            Log . d ( "HELLO", it . toString() )
        }
    }
}

fun producer () = flow<Int> {
    val list = listOf ( 1, 2, 3, 4, ---- 10)
    list . forEach { delay (1000)
        emit ( it )
    }
}
```

GlobalScope . launch { delay (3500) job . cancel ( ) }

Cold Case → all consumer will get ~~from~~ data from starting. (join wherever doesn't matter)

Hot Case → lost data will not get by consumer if joined late.

Flow operators.

```
GlobalScope.launch(Dispatchers.Main){
    producers()
        .onStart { emit(-1) }
        .onCompletion { emit(6) }
        .onEach {
            .collect { }
        }
}
```

Terminal Operator

producer().first() → Terminal operator

↓ this will return first element of list.

producer().toList() → Terminal operator

↓ this will convert flow into list & return

this is suspend function

To start a flow, we need terminal operator like collect.

```
GlobalScope.launch(Dis..Ma){
    producer()
        .map { it * 2 }
        .filter { it < 8 }
        .collect {
            log.d("....", it.toString())
        }
}
```

map will convert one flow of data into another

this will filter based on condition. It returns either flow

this will consume data.

```
{ producer()
    .buffer(3)          // capacity
    .collect {
        delay(1500)
    }
}
```

In flow, context switching is done using flowOn operator (not with context in coroutine)

(Eg producer getting data from n/w should operate on IO thread, consumer should get data in Main thread (Collector))

flowOn operator works upstream.

```
GlobalScope.launch(Dispatchers.Main) {
    producer()
        .map { it * 2 }
        .flowOn(Dispatchers.IO)
        .filter { it < 8 }
        .flowOn(Dispatchers.Main)
        .collect {
        }
}
```

`catch` → operator to catch Exception.

```
private fun producer(): Flow<Int> {
    return flow<Int> {
        val list = listOf(1,2,3,4,5)
        list.forEach { delay(1000)
            emit(it)
            throw Exception("Error in Emitter")
        }
    }.catch { Log.d("Exception", "Emitter catch - ${it.msg}") }
}
```

SharedFlow

It is of hot nature. It does not maintain state, & multiple consumer can consume the emitted data.

```
private fun producer(): Flow<Int> {
    val mutableSharedFlow = MutableSharedFlow<Int>(1)   // replay
    GlobalScope.launch {
        val list = listOf<Int>(1,2,3,4,5)
        list.forEach {
            mutableSharedFlow.emit(it)
            delay(1000)
        }
    }
    return mutableSharedFlow
}
```

StateFlow

It is also of hot nature like SharedFlow but it maintain the state, any late consumer will get last updated value.

```
private fun producer(): StateFlow<Int> {
    val mutableStateFlow = MutableStateFlow(10)
    GlobalScope.launch {
        delay(2000)
        mutableStateFlow.emit(20)
        delay(2000)
        mutableStateFlow.emit(30)
    }
    return mutableStateFlow
}
```

LiveData Vs StateFlow

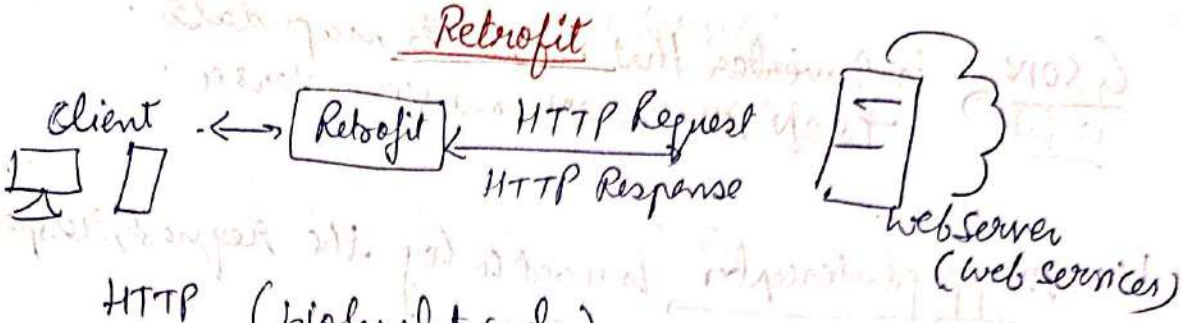
→ In StateFlow, we have flow operator to execute in background thread

→ Transformations on Main thread (LiveData Execute on main thread fully all work on main thread)

→ operators → Limited operator in LiveData

→ lifecycle Dependent on activity, frag → LiveData is lifecycle aware it works only but suppose in repository there you can use StateFlow, LiveData can't be used.

Retrofit



HTTP (boilerplate code)

↓

Volley (Limited Rest Specific features, Poor authentication layer, Meagre documentation, Smaller Community)

↓

Retrofit (active Community
↳ Easier troubleshooting
↳ Expressive code with more abstraction.
↳ Manage resources Efficiently
↳ Background thread
↳ Async calls and queues
↳ Automatic JSON parsing using GSON lib.
↳ automatic Error handling Callbacks.
↳ Built in User authentication Support.

Status code Range	Meaning
100	Informational
200	Success
300	Redirects
400	Client Error
500	Server Error

Main HTTP Methods → Get, Post, Put, Patch, delete

HTTP → hyper text transfer protocol.

Restful Web Services

↳ A web service is restful when it provides stateless ops to manage data using different HTTP methods and structured URLs.

JSON → Javascript Object Notation (Light weight & Extensively used)

SOAP → Simple object Access protocol.

GSON is Lib → is Converter that helps to map data. JSON to GSON and Vice Versa.

HTTPLoggingInterceptor is used to log the request, response body.

PATH Parameters

same

```
@GET("destination/{id}")
fun getDestination(@Path("id") id : Int): Call<Destination>
```

http://base-url/destination/47

QUERY Parameters (If you want to sort or filter items at backend)

Eg: www.abc.com/users?occupation=doctor

http://...../destination?country=India

```
@GET("destination")
fun getDestinationList(@Query("country") country : String):
                    Call<List<Destination>>
```

QUERYMAP PARAMETER

(When you have a lot of Query Params, then use QueryMap)

www.abc.com/users?count=3&Country=India

```
@GET("destination")
fun getDestinationList(@QueryMap filter: HashMap<String, String>):
                    Call<List<Destination>>
```

Send data to the Web Service

→ in request Body of HttpRequest.

→ Data format → JSON
→ FormUrlEncoded

```
@POST("destination")
fun addDestination(@Body newDest: Destination):
                                    Call<Destination>
```

Static / dynamic header

```
@Headers("x-device-type: Android", "x-foo: bar")
@GET("destination")
fun getDestinationList(
    @QueryMap filter: HashMap<String, String>,
    @Header("Accept-Language") language: String
): Call<List<Destination>>
```

dynamic header

```
// create a Custom Interceptor to apply Headers application wide.
val headerInterceptor: Interceptor = object: Interceptor {
override fun intercept(chain: Interceptor.Chain): Response {
    var request: Request = chain.request()
request = request.newBuilder()
        .addHeader("x-device-type", Build.DEVICE)
        .build()
val response: Response = chain.proceed(request)
return response
}
}
```

```
@FormUrlEncoded
@PUT("destination/{id}")
fun updateDestination(
        @Path("id") id: Int,
        @Field("city") city: String,
        @Field ...
)
```

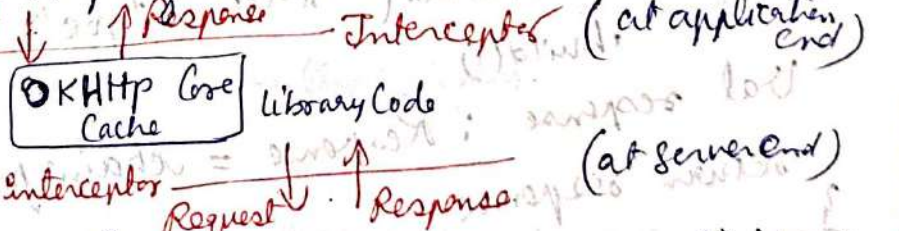
By Default, in retrofit we have 10 sec timeout for getting response. But manually also we can set timeout in OkHttpClient.Builder:

```
.callTimeout(5, TimeUnit.SECONDS)
```

Cancelling Requests

```
val requestCall: Call<List<Destination>> = destinationService.
        getDestinationList(filter)
requestCall.cancel()
requestCall.isCancelled
```

Interceptors are a powerful mechanism that can monitor, rewrite, and retry the API call. So basically, when we do some API call, we can monitor the call or perform some tasks.



Use Case

(1) Logging the errors centrally
(create errorInterceptor) when(response.code) 400 →
.addInterceptor(ErrorInterceptor()) throwing 401 →
⋮

(2) Caching the response

(3) Adding the header like Access token centrally.

Kotlin Revision.

| 3 Reasons - why developers should learn Kotlin?

① Kotlin is concise → can write lot things in less code.

② Kotlin is null-safe → less no. of null pointer

③ Kotlin is interoperable → Can call java from Kotlin
Kotlin from java.

first class lang. backed by Google.

---

val and var → general/mutable variable in kotlin
Can be assigned multiple times
↳ like final modifier in java (immutable)
↳ Can be initialized only one time

---

const and val
↳ both are immutable property.

const is compile time constant. no running assignment of values is allowed in const variables.

val can be initialized at runtime also.

Eg const val companyName = "Abc" // ✓
val companyName = "Abc" // ✓ will work

const val companyName = getAbc() ✗ will not work
val companyName = getAbc() // ✓ will work

Safe calls (?.) Vs Null check (!!)

(!!) → use this when property can't have a null value

(?.) → If not sure, use this to avoid NPE.

```
var name: String = "abc"
name = null          // compilation Error
```

```
var name: String? = "abc"      // no Error
name = null                    // no Error
```

Lateinit and Lazy keywords

"Lateinit" keyword is used for late initialization of variable.

↳ you should be sure that before accessing it, you must initialized it else you will get:
UninitializedPropetyAccessException.

Eg
```
private lateinit var courseName: String

fun fetchCourseName(courseId: String) {
    courseName = courseRepository.getCourseName(courseId)
}
```

Usage of lateinit

① for late initialization of variables.
② for injecting an object using Dagger.

Lazy keyword

There are certain classes whose object initialization is very heavy and so much time taking that it results in the delay of the whole class creation process.

Eg
```
class SomeClass {
    private val heavyObject: HeavyClass = HeavyClass()
}

class SomeClass {
    private val heavyObject : HeavyClass by lazy {
        HeavyClass()
    }
}
```

to check if lateinit variable has been initialized

```
class Person {
    lateinit var name: String

    fun initializeName() {
        println(this::name.isInitialized)
        name = "abc"
        println(this::name.isInitialized)
    }

    fun main(args: Array<String>) {
        Person().initializeName()
    }
}
```

o/p false
true

Companion object

In java, we used to access classname.funname directly by using static keyword, but in Kotlin there is no static keyword. we can use this method by using Companion object.

Java

```
public class MyClass {
    public static void myMethod() {
    }
}
```

MyClass.myMethod();

Kotlin

```
class ToBeCalled {
    companion object {
        var someInteger: Int = 10
        fun callMe() = println("You are calling me!")
    }
}

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

ToBeCalled.callMe()

Visibility Modifiers → private, Protected, ...

① public (Visible to All)
② Private (Visible inside file/class)
③ Protected (Visible to SubClass)
④ Internal (Visible inside Module (project))

Eg

```
open private class Student() {
    protected open fun getCount() = 1000
}
private class StudentManager : Student() {
    override fun getCount() = 2000
}
```

Data Class. is a container of data.

- ↳ primary constructor needs to have one parameter.
- ↳ All primary constr.. parameter need to be marked as val or var.
- ↳ data class cannot be abstract, open, sealed or inner

```
data class Abc ( val name: String, val age : Int)
```

Singleton Class is a class that is defined in such a way that only one instance of the class can be created & used Everywhere. Eg. NetworkService, DatabaseService.

Properties ① have only one instance.
② Globally accessible.

```
fun main() {
    val profile = UserProfile()
    val profileTwo = UserProfile()
    println(profile.toString())
    println(profileTwo.toString())
}
```

```
class UserProfile {
}
```

diff address

O/P { UserProfile@26...
UserProfile@24... } diff inst of object

```
main() {
    println( UserProfile.toString() )
    "                "
}
```

// It will create single instance.

object UserProfile {

}

Used for Singleton

object → can have fun, variables, init block.

doesn't have Constructor, Cannot instantiate

Open keyword

To make a class inheritable to the other classes, you must mark it with the open keyword Otherwise you will get an Error saying "type is final so Can't be 'inherited'".

By default, all classes in Kotlin are final.

Eg

```
open class BaseClass {
    open val courseId : Int = 0
    open fun courseName() { }
}

class ABC : BaseClass() {
    override val courseId : Int = 1
    override fun courseName() { }
}
```

In Kotlin, new classes, the functions, variables are final in nature by default. So to make it inheritable use the open keyword with class, function and variable name.

Scope Functions. with, let, run, apply,

↳ It makes your code concise and readable

Eg1 "With"

```
fun main() {
    val person = Person()
    val bio : String = with(person) {
        println(name)
        println(age)
        age + 5
        "He is a freak who loves to teach in his own way"
    }
    println(bio)
}
```

Property1 : Refer to content object by using "this"

Property2 : return value is "lambda result"

Eg2 "apply"

Return : Content object
Content object : this

```
fun main() {
    val person = Person().apply {
        name = "Abc"
        age = 26
    }
    with(person) {
        println(name)
        println(age)
    }
}
```

Eg3 also

Return : Content object
Content object : it

also

~~fun duplicateNum~~

```
Val numberList : MutableList<Int> = mutableListOf
                                    (1,2,3)

nos = numberslist . also {
        println ("The list Elements are : $it")
        it . add(4)
        println ("The list elements after adding an
                  Elemat : $it")
      it . remove (2)
    println ("The list Elements after removing an Element:
              $it")
}
println ("nos is : $nos")
println ("Orginal no is : $numberlist")
```

} Same only

Solid principle of android
Clean Architecture !

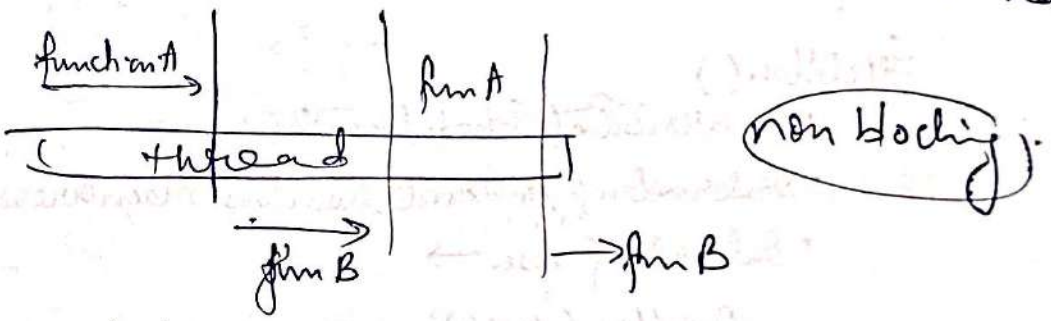
Synchronous and Asynchronous programming

→ Synchronous communications are scheduled, real time interactions by phone, video, or in-person.

→ Asynchronous communication happens on your own time and doesn't need scheduling.

→ Async programming is about non-blocking Execution between functions, and we can apply async using single-threaded or multithreaded programming. So, multithreading is one form of asynchronous programming.

Coroutines Co + Routines means when func. cooperate with each other, we call it Coroutines.
↓ Cooperation ↓ function.



Threads are managed by OS and coroutines by the users as it can execute a few lines of function by taking advantage of the cooperation.

Coroutines are light weight threads → means it doesn't mapped the native thread, it doesn't require context switching on the processor, so they are faster.

Stackless, Stackful → types of coroutines.

```
fun fetchAndShowUser() {
    val user = fetchUser()
    showUser(user)
}
fun fetchUser(): User {
    // make n/w call
    // return user
}
fun showUser(user: User) {
    //show user
}
```

→ this will throw NetworkOnMainThreadException as the n/w call is not allowed on the main thread.

To solve this, use any of below —
1. use callback
2. using Rx java
3. using coroutines.

① Using Callback

```
fun fetchAndShowUser() {
    fetchUser { user ->
        showUser(user)
    }
}

fun fetchUser(callback: (User) -> Unit) {
    // make n/w call on background thread to get user
    // callback with user
    callback(user)
}

fun showUser(user: User) {
    // show user
}
```

② Using RxJava. Reactive world approach (we can get rid of nested callback)

```
fetchUser()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe { user ->
        showUser(user)
    }

fun fetchUser(): Single<User> {
    // make network call
    // Emit User
}

fun showUser(user: User) {
    // show user
}
```

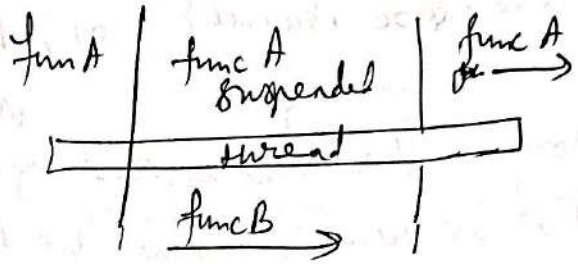
③ using Coroutines

```
fun fetchAndShowUser() {
    GlobalScope.launch(Dispatchers.Main) {
        val user = fetchUser()   // fetch on IO thread
        showUser(user)           // back on UI thread
    }
}

suspend fun fetchUser(): User {
    return withContext(Dispatchers.IO) {   // make n/w call on IO thread & return user
    }
}
```

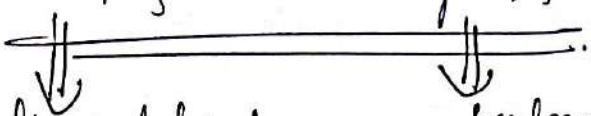
coroutines, the code looks synchronous, but it is asynchronous, by writing the launch, we launch a coroutine to do a task.

Dispatchers IO → for doing n/w ops & disk related work,
Default → CPU intensive work
Main → UI thread of android.

Suspend is a function that could be started, paused, & resume.



Launch { } Vs async { }



fire and forget — perform a task and return a result.

```
GlobalScope.launch(Dispatchers.Default){
}
```

```
val deferred = GlobalScope.async(Dispatchers.Default){
    // do something and return result, eg 10 as a result
    return@async 10
}
val result = deferred.await() // result=10
```

Note

1. Both **launch** & **async** are used to launch a coroutine. This Enables us to do tasks in parallel. They are not Suspend func.
2. **Async** can be used to get the result that is not possible with the launch.
3. withContext does not launch a coroutine & it is just a Suspend func, used for shifting the context of the Existing coroutine.

Scopes in Kotlin Coroutines

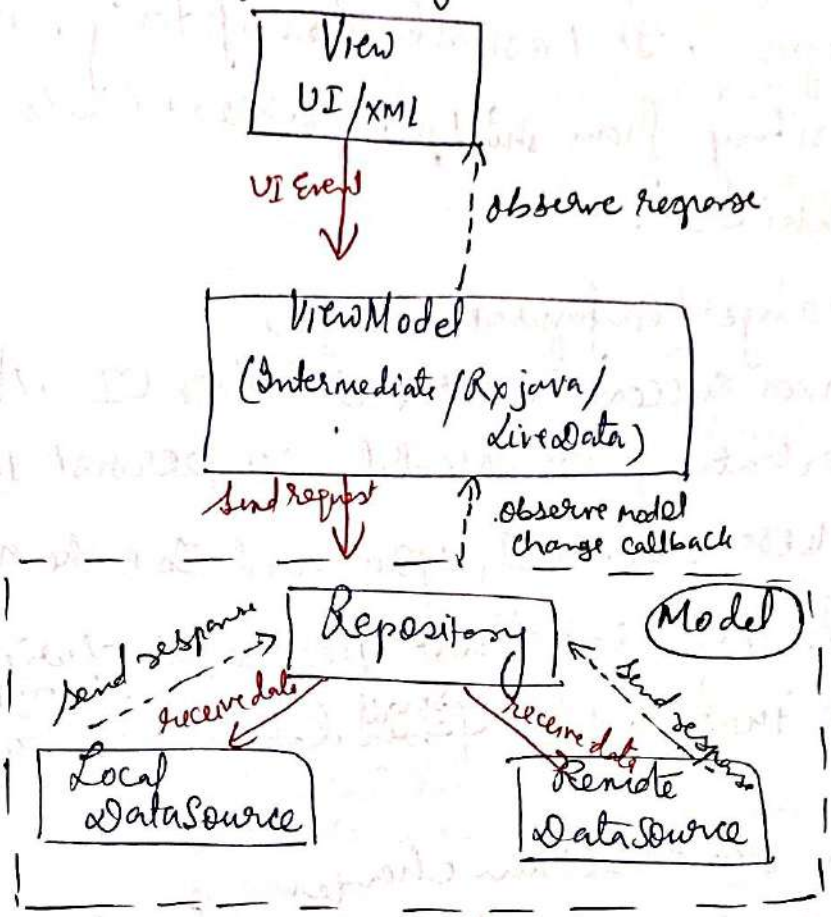
↳ is useful because we need to cancel the background ... soon as the activity is destroyed.

Activity → lifecycleScope.launch { } — Can be used. It is associated with activity, bind to lifecycle of activity

ViewModel → viewModelScope.launch { } — It is bind to lifecycle of viewModel.

↳ Means as soon as ViewModel is destroyed, the task will get canceled if it is running because we have used the ... Scope which is bound to the lifecycle of the ViewModel.

MVVM is a Model View ViewModel architecture that removes the tight coupling b/w each component.



Model → It represents the data and the business logic of app. It consists of the business logic – local and remote data source, model classes, repository.

View → It ~~is a bridge b/w~~ consists of the UI code (Activity, Fragment), XML. It sends the user action to the ViewModel but does not get the response back directly. To get the response, it has to subscribe to the observables which ViewModels exposes to it.

ViewModel → It is a bridge b/w the View and Model (business logic). It does not have any clue which View has to use it as it does not have a direct reference to the View. It should not be aware of the view who is interacting with. It interacts with the Model and exposes the observables that can be observed by the view.

ViewModel is a class that holds the state of the UI and Exposes the State to the UI. Also the ViewModel Encapsulates the business logic. It basically does offering. It uses multiple repository from multiple sources. Combine & filter & then into the UI.

Orientation changes (Configuration changes)

Eg n/w call gives success result & it sends to UI. After that change the orientation, ViewModel will persist the data. If you don't use ViewModel, you need to make n/w call again.

Earlier we were using SaveInstanceState, RestoreInstanceState during orientation changes (bundle thing) but that handles small data. There was limitation in that.

1. It handles the orientation changes.
2. Lifecycle awareness → It will only get destroyed when activity gets destroyed.

How to Cancel Coroutine?

⇒ To cancel a coroutine, you simply need to call cancel on the Job object that represents the coroutine.

⇒ for that the coroutine is in a Cancellable State.

Some blocking ops, such as delay & withContext, are Cancellable while others, such as sleep and blocking, are not.

```
val job = GlobalScope.launch {
        //do some work
    }
    job.cancel()
```

lateinit It is useful in a scenario when we do not want to initialize a variable at the time of the declaration and want to initialize it at some later point in time, but we make sure that we initialize it before use.

```
private lateinit var mentor: Mentor
```

`::isInitialized` to check it is initialize or not.

If not initialize, it will throw UninitializedPropertyAccessException

Note
1. Can be only used with var keyword.
2. Can be only used with a non-nullable variable.
3. Should be used if the variable is mutable & can be initialized later.
4. Should be used if you are sure about the initialization before use.

Lazy is useful in a scenario when we want to create an object inside a class, but that object creation is expensive and that might lead to a delay in the creation of the object that is dependant on that expensive object.

```
class Session {
    private val mentor: Mentor = Mentor()
}
```

Session obj dependent on Mentor obj
Mentor is expensive

(II)

```
class Session {
    private val mentor: Mentor by lazy { Mentor() }
}
```

Note
1. by using lazy, mentor will get initialized only when it is accessed for first time.
2. It will lead to fast creation of Session object because Mentor obj will not get initialized unnecessarily during creation of session object.

③ lazy can be only used with val keyword, (read only)
④ we want variable to be initialized only if we need it.

Elvis Operator. (?:) is used to return the not null value. Even the conditional Expression is null. It is also used to check the null safety of values.

Const Vs Val (Both read only) Var → Read + write

Val → known at runtime
↓
Contents can be mutated

known at compile time
↓
It should be declare at top
↓
or at inside Companion object

Const val gfg = "Hi" ✓
Val coursename = "android" ✓

~~final~~ Immutable : Strings or primitive Const val gfg = getgfgname() ✗
Val coursename = getcoursename() ✓

```
private val mylist = mutableListOf<Int>()
private const val DEFAULT_NICKNAME = "brainne"

fun main() {
        mylist.add(10)
    ;
}
```

Const Advantages → As the value has been inlined, there will be no overhead to access that variable at runtime. Hence it will lead to a better performance of the application.

Scoped functions – Utility functions (apply, let, with, run)

```
fun main() {
    val emp = Employee()
    emp.age = 20
    emp.name = "John"

    emp.apply { this: Employee
        age = 30
        name = "Sean"
    }
    println(emp)
}

data class Employee(var name: String = "", var age: Int = 18)
```

returns unit by default or last value

```
println(emp.age)
println(emp.name)
emp.let { it: Employee
    println(it.name)
    println(it.age)
}
```

return value will be obj only

```
val emp: Employee? = null

emp?.age = 20
emp?.name = "John"

emp?.let {
    it.age = 20
    it.name = "John"
}
```

```
val emp: Employee = Employee()

emp.let { it: Employee
    it.age = 20
    it.name = "John"
}
```

⇒

calling with externally on this obj emp reference

```
with(emp) { this: Employee
    age = 30
    name = "XYZ"
}
```

```
emp.run { this: Employee
    age = 35
    name = "PDR"
}
```

run is → combination of with & let

- **with** → If you want to operate on a non null object.
- **let** → If you want to just execute lambda Expression on a nullable object and avoid NPE.

run → If you want to operate on a nullable object, Execute S. Expression and avoid NPE.

apply → If you want to initialize or Configure an object.

also → If you want to do Some additional object configuration or operations.

```
/** Scope fn' : 'also'
    property 1 : Refer to content object by using 'it'
    property 2 : The return value is the 'content object' */
val numbersList: MutableList<Int> = mutableListOf(1,2,3)

numbersList.also { it: MutableList<Int>
println("The list elements are : $it")
    it.add(4)
    println("The list Elements after adding : $it")
}
```

```
/* Property 1: Refer to Content object by using 'this'
   Property 2 : The return value is the Content object' */
val person = Person().apply { this: Person
                name = "abc"
                age = 26
            }
with(person) { this: Person
        println(name)
    } println(age)
person.also { it: Person
    it.name = "def"
    println("new name : it.name")
} }
```

LiveData

It is basically a data holder & used to observe the changes of a particular view and then update the corresponding change.

LiveData used to make the task of implementing ViewModel easier. View will be updated with data in the foreground.

Advantages of LiveData

→ No memory leaks
→ View always get the up to date data
→ No Crash due to Stopped activities.

Eg 2 Methods of LiveData

(1) SetValue()
↓
Using Main thread to set/change the data of mutableLiveData
↓
If you call this 2 times data will be updated 2 times

(2) postValue()
↓
using background thread to change, use postValue() method of mutableLiveData class.
↓
If you call this 2 times, before execution of main thread observer will receive notify only once and updated with last. (latest data)

Extension function: The ability to add more functionality to the existing classes, without inheriting them. This is achieved through a feature known as Extensions.

When a function is added to an existing class it is known as a Extension function.
(defined outside of a class)

```
fun <class_name>.<method_name>(){
    function body
}
```

Eg of Extension functions.

```
fun main(){
var agesStage = AgesStage()
println(" ABC : " + agesStage.hasBabyChild(2))
}
```

```
fun AgesStage.hasBabyChild(age : Int) : Boolean {
    if (age > 0 && age < 3){
        return true
    }else {return false
    }
}
```

```
class AgesStage(){
    fun hasAdult(age : Int) : Boolean{
        if(age > 18) {return true
        }else {return false
        }
    }
}
```

Sealed functions used to prevent users from inheriting a class.
It can be sealed using the sealed keyword.

```
Sealed class Gadget(val s : String)
{ class Laptop: Gadget("Laptop")
  class Phone : "Phone")
}
fun display(g: Gadget) = when(g){
                    is Gadget.Laptop -> println("Laptop")
                    is Gadget.Phone -> println("Phone")
fun main(args : Array<String>) }
{   val obj1 = Gadget.Laptop()    display(obj1)
    " obj2 = "  .Phone()         display(obj2) }
```

else part not needed here

above Sealed class, we have 2 types → Laptop and Phone. we are creating an object in class and assigning its type at runtime.

Lambda Expression

It is a Concise, unnamed function Enclosed in braces, used for defining code blocks that can be passed as values or stored as variables.

```
Val myVariable : (Int, String) → String = { a: Int, b: String →
                                            "$a + $b" }
```

Eg1

```
fun main(args: Array<String>) {
    Val product = { a: Int, b: Int → a*b }
    Val result = product(9, 3)
    println(result)
}
```

(parameters, body)

O/P 27

Eg2

```
fun main(args : Array<String>){
    Val greeting = { println("Hello!") }
    // invoking function
    greeting()
}
```

O/P Hello!

Here the function (lambda Expression) is invoked as greeting()

How branching Strategy works ?

There are Separate master and develop branches for Each application. (active branches of all time)

(1) Sprint branches to be created from ~~respective Sprint~~ develop branch.

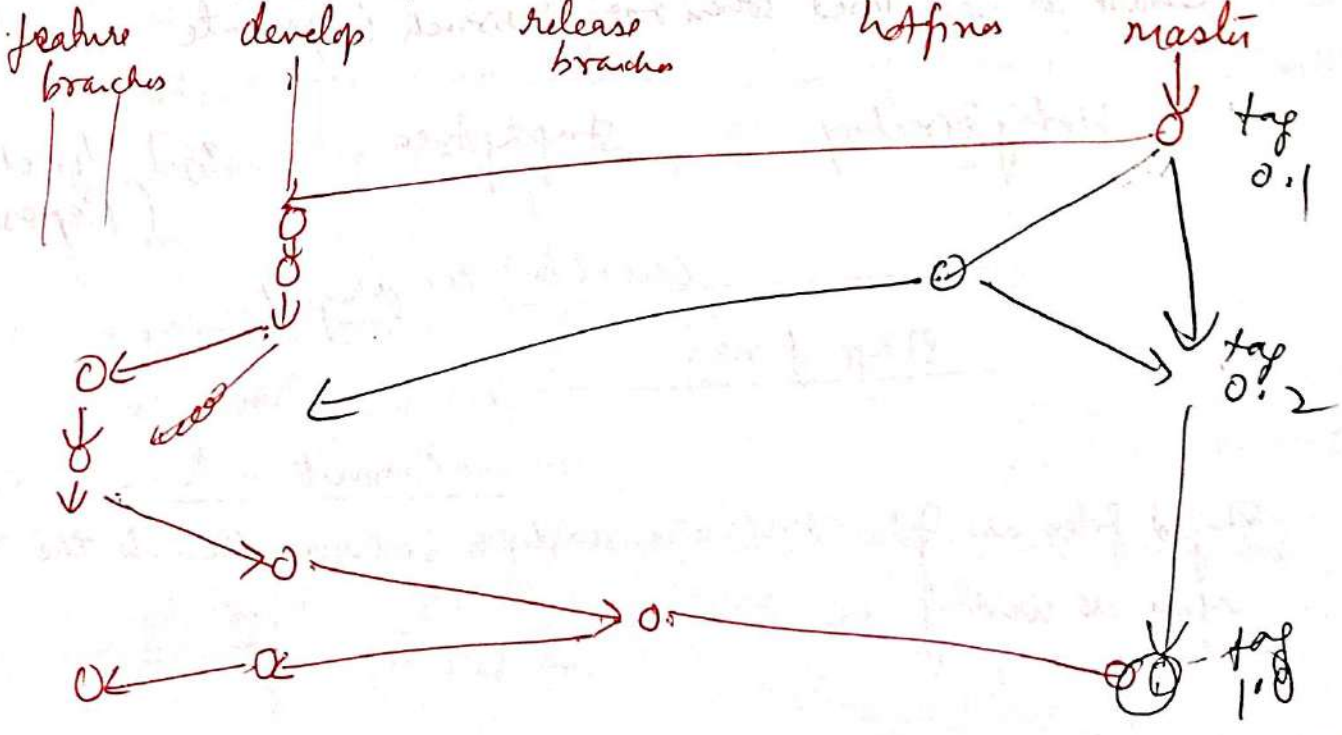
(3) Features branches to be created from respective Sprint branch

(4) After Completing Sprint, the branch will be merged to develop branch.

(5) All release branches to be created from develop. All UAT and critical bugs reported by QA team will be fixed to release branch itself and will be merged to develop & respective Sprint branch again.

(6) After Completing UAT, APKs will be released for production and code will be merged to ~~devel~~ master with TAGs and release branches will be deleted.

(7) If there is critical bug from production, hotfix branch to be created from master, after fixing bug in hotfix, code will be merged to master as well as develop.



Git Commands

⇒ git init

⇒ git remote add origin "https- - - "

⇒ git pull origin master

⇒ git status

⇒ git add edu.txt

⇒ clear

⇒ ~~git bra~~ (master) > git branch dev

⇒ git branch

⇒ git checkout dev

⇒ git checkout -b abc (//checkout & create branch both)

⇒ git status

⇒ git add multiply.java

⇒ git commit -m "added multiply.java"

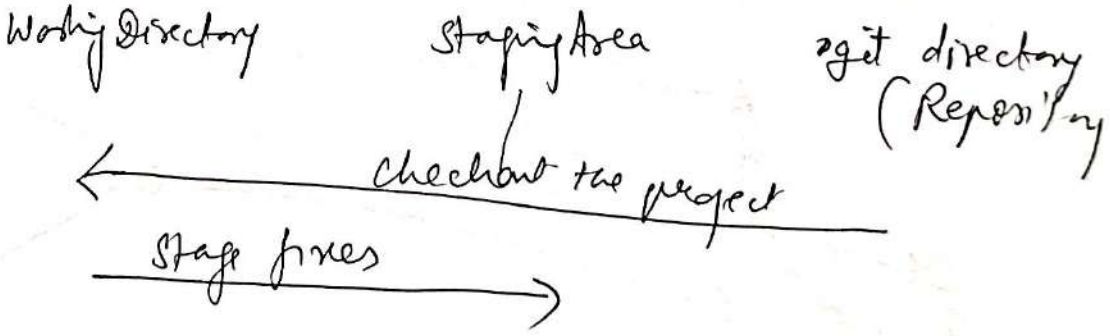
⇒ git log.

Merge Vs Rebase

→ Merge preserves history. Rebase rewrites history.

Merge is best used when target branch is supposed to be shared.

Rebase is best used when target branch is private.



Staged files are files that are ready to be committed to the repo you are working on.

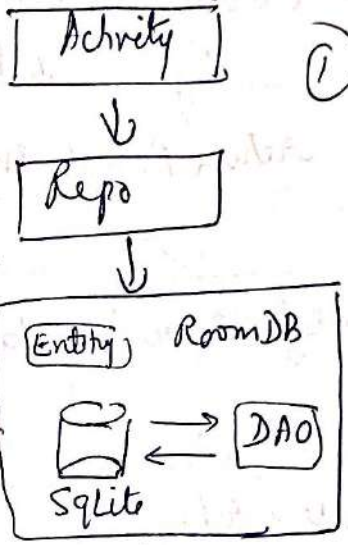
Room & Sqlite

(Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.) ⑨

Sqlite is a type of relational database (RDBMS)

Room is a SQLite object mapping library. Room takes care of mundane tasks that you used to handle with an SQLiteOpenHandler.

Room persistence library allows developers to easily convert SQLite table data into java objects. Room provides compile time checks for SQLite statements and can return Rxjava, Flowable and LiveData observables



① Entity → will define rows of SQLite db. Kind of model of data you want to store

If you want to replace row in case of conflicts

```
@Insert(onConflict =
    onConflictStrategy.REPLACE)
```

```
@Dao
public interface NoteDao{

@Insert
long[] insertNotes(Note... notes);

@Query("Select * from notes")
LiveData<List<Note>> getNotes();

@Delete

@Update
}
```

② DAO (Data Access Objects)

Here you specify SQL queries and associate them with method calls. The compiler checks the SQL, generates the queries from convenience annotations for common queries, such as @Insert, @Update, @Delete, @Query.

③ Database: This class will contain code to instantiate the SQLite database object. We need one instance of Room db, so we will be implementing the Singleton pattern.

```
@Entity(tableName = "notes")
public class Note {

@PrimaryKey(autoGenerate = true)
private int id;

@ColumnInfo(name = "title")
private String title;
}
```

Advantages of using Room

1. Compile time verification of queries.
2. Reduces Boilerplate code
3. Easy to understand and use.
4. Easy integration with Rxjava, LiveData and Kotlin Coroutines

Annotations: @Entity, @Database, @Dao → interface

Work Manager

To Cancel a task
WorkManager.cancelWorkById(workId)

It is a part of jetpack Component. Its a library which makes Easy to Schedule deferrable, Asynchronous tasks that are Expected to run Even if the app Exists or device restart.

→ Create Workerclass which Extend Worker.

```
@Override fun doWork(): Result {
    // you work here
    return Result.success()   //task result
}
```

responsible for Executing task in background.

Result.failure()
Result.retry()

we can pass input also in workManager like user id
.setInputData(userId)

⇒ WorkRequest → OneTimeWorkRequest
→ PeriodicWorkTimeRequest (after 1hr / 2hr / 3hr interval)

⇒ we can add Specific Constraints in WorkRequest

```
val constraint = Constraints.Builder()
    .setRequiresCharging(true)
    .setRequiresBatteryNotLow(true)
    .build()

val yourWorkRequest = OneTimeWorkRequestBuilder<YourWorkClass>()
    .setConstraints(constraints)
    .build()
```

⇒ Chaining of Tasks

Chaining in series (work1, work2) / parallel chaining

```
WorkManager.getInstance(context)
    .beginWith(listOf(work1, work2, work3))
    .then(work4)
    .enqueue()
```

Navigation Components

Jetpack Compose : Navigation

NavController NavHost

```
onCreate {
setContent {
ComposeNavigationApp {
  composeNavigation()
```

Its a Controller Central API of the Navigation Component. So to navigate to any destination, use NavController.

```
val navController = rememberNavController()
```

→ NavController is stateful and it will have all information about the back stack.

→ NavController should be associated with only one NavHost.

→ NavHost is area where you draw your screen. With the help of NavController you can navigate to screens.

```
fun ComposeNavigation() {
val navController = rememberNavController()
NavHost ( navController = navController,
          startDestination = "first_screen"
        )
{ composable ("first_screen")
        { FirstScreen(navController = navController)
        }
  composable ("second_screen")
        { SecondScreen(navController = navController)
        }
}
}
@Composable
fun FirstScreen(navController : NavController) {
    Column( modifier = Modifier.fillMaxSize(),
          ) {
        Text(
      , onClick { navController.navigate ("second_screen") } }
```

JetPack Navigation Component (graph)

Create (nav_graph) inside res directory

```
<navigation xmlns: . . .
        id = "@+id/nav_graph"
    startDestination = "@id/nav_first_frag" >

  <fragment
        id = "@+id/nav_first_frag"
    name = "app.navigationComponentEg.FirstFragment"
      layout = "@layout/frag_first" >

    <action
        id = "@+id/action_first_to_second"
      destination = "@id/nav_second_fragment" />
  </fragment>

  <fragment
        id = "@+id/nav_second_frag"
    name = . . . . .SecondFragment
    layout = "@layout/fragment_second" >

    <argument
          name = "arg1"
        argType = "Integer"
        defaultValue = "0" />

    <argument
          name = "arg2"
        argType = "String"
        defaultVal = "default" />

  </fragment>
</navigation>
```

Safe arguments) add → classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.2.2"

```
...setOnClickListener {
    val direction = FirstFragmentDirections.actionFirstFragmentToSecondFragment(arg1 = 123, arg2 = "abcd")
    findNavController().navigate(direction)
}
```

```
button.setOnClick {
    findNavController().navigate(R.id.nav_second_frag)
}
```

using destination id

Using ClickListener

```
button.setOnClickListener {
    Navigation.createNavigateOnClickListener(R.id.nav_second_frag, null)
}
```

Using NavigationAction

```
button.set... {
    findNavController().navigate(R.id.action_first_to_second)
}
```

in activity xml, add below.

```
<fragment
    android:id="@+id/nav_host_fragment"
    layout_width
    height
    name="androidx.navigation.fragment.NavHostFragment"
    navGraph="@navigation/nav_graph"
    app:defaultNavHost="true" />
```

defines the NavHostFragment used by NavController

DataBinding Vs ViewBinding

ViewBinding DataBinding

Earlier we were using findViewById → boilerplate code
crashes was happening
as it maps at runtime.

Use databing or ViewBinding
Dated to avoid crash, reduces boilerplate code, maps At Compile Time.

DataBinding → Binding data (from code) to Views + ViewBinding (Binding Views to code)
→ <layout tag _ has to include inside Xml

ViewBinding → layout tag not needed
↳ only binding views to code.

ViewBinding is faster than DataBinding
It reduces app sizes.

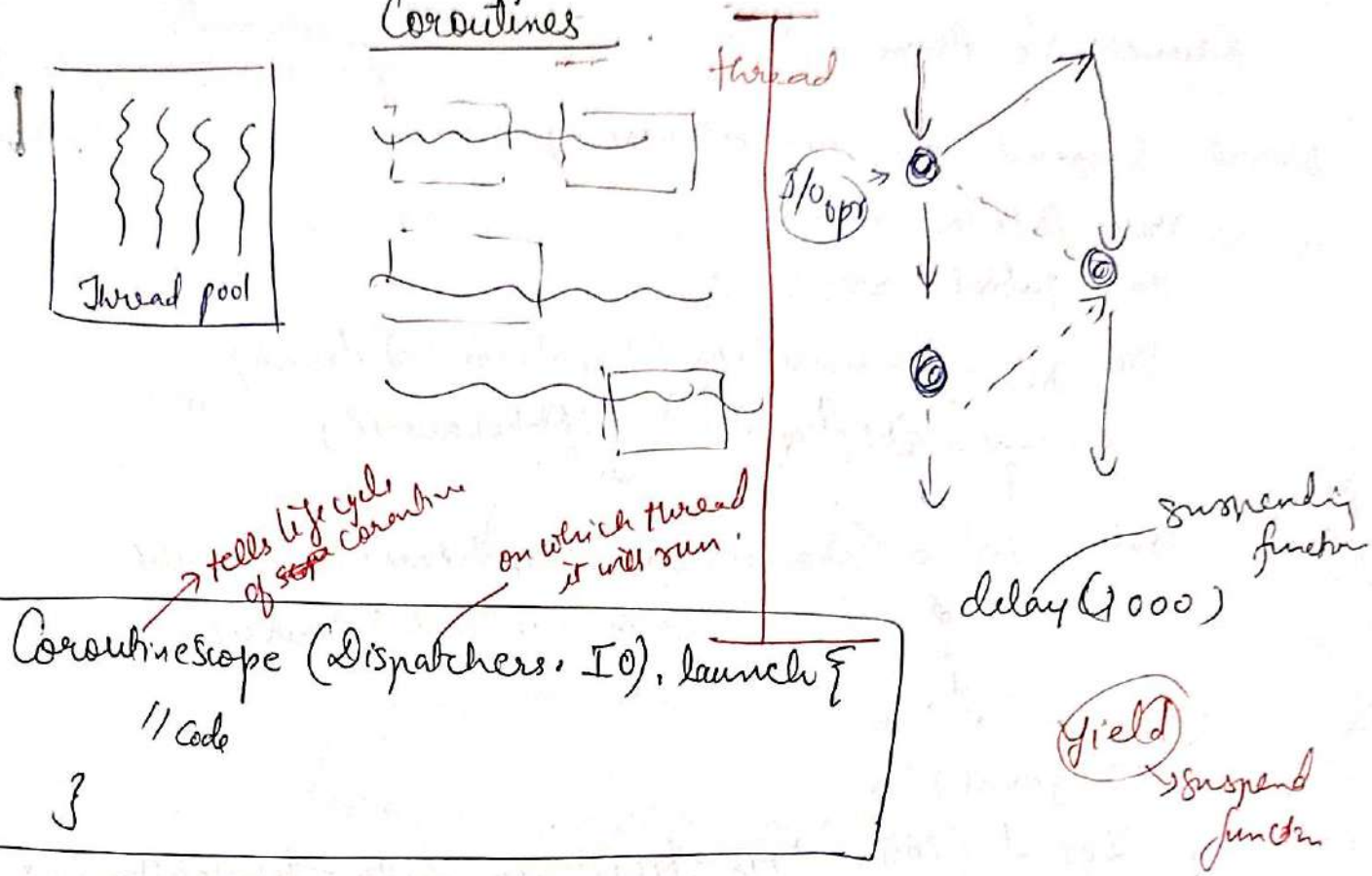
```
apply Plugin 'kotlin-kapt'
android {
    dataBinding {
        enabled = true
    }
}
```

```
android {
    viewBinding {
        enabled = true
    }
}
```

```
// DataBinding
binding = DataBindingUtil.setContentView(this, R.layout.activity_main
                                         as ActivityMainBinding.

// ViewBinding
binding = ActivityMainBinding.inflate(layoutInflater)
```

first_activity.xml → ActivityFirstBinding
second_fragment.xml → FragmentSecondBinding
item_movie_layout.xml → ItemMovieLayoutBinding

Coroutines



→ Suspending functions must be called from Either Coroutines or Other suspending functions.

→ Helps coroutines to suspend the computation at a particular point.

```
Suspend fun task1()
    { Log.d(TAG, "Starting task1")
      delay(1000)
      Log.d(TAG, "Ending task1")
    }

Suspend fun task2()
    { Log.d(TAG, "Starting task2")
      delay(2000)
      Log.d(TAG, "Ending task2")
    }
```

O/P

Starting task1
Starting task2
Ending task1
Ending task2

Launch Vs Async.

o/p FB-54, In...

```
private suspend fun printFollowers() {
    var fbFollowers = 0
    var instaFollowers = 0
    val job = CoroutineScope(Dispatchers.IO).launch {
        fbFollowers = getFbFollowers()
    }
    val job2 = CoroutineScope(Dispatchers.IO).launch {
        instaFollowers = getInstaFollowers()
    }
    job.join()
    job2.join()
    Log.d(TAG, "FB - $fbFollowers, Insta - $instaFollowers")
}
```

both will give same o/p.

```
private suspend fun printFollowers() {
    val fb = CoroutineScope(Dispatchers.IO).async {
        getFbFollowers()
    }
    val insta = CoroutineScope(Dispatchers.IO).async {
        getInstaFollowers()
    }
    Log.d(TAG, "Fb - ${fb.await()}, Insta - ${insta.await()}")
}
```

```
private suspend fun printFollowers() {
    CoroutineScope(Dispatchers.IO).launch {
        var fb = getFbFollowers()
        var insta = getInstaFollowers()
        Log.d(TAG, "Fb - $fb, Insta - $insta")
    }
}
```

```
private suspend fun printFollowers() {
    CoroutineScope(Dispatchers.IO).launch {
        var fb = async { getFbFollowers() }
        var insta = async { getInstaFollowers() }
        Log.d(TAG, "Fb - ${fb.await()},
            Insta - ${insta.await()}")
    }
}
```

Coroutines Jobs & Cancellation

```
ate  suspend fun Execute() {
     var parentJob = GlobalScope.launch(Dispatchers.Main) {
          Log.d(TAG, "Parent Started")
     var childJob = launch(Dispatchers.IO) {
          try {
               Log.d(TAG, "Child job started")
               delay(5000)
               Log.d(TAG, "Child job Ended")
          } catch (e: CancellationException) {
               Log.d(TAG, "Child job Cancelled")
          }
     }
     delay(3000)
     childJob.cancel()
     Log.d(TAG, "Parent Ended")
}
parentJob.join() // once child job completed then only Parent job will complete
Log.d(TAG, "Parent Completed")
}
```

Note: If ParentJob.Cancel(), then all of its child job will automatically get cancelled.

```
Instead fun Execute() {
     val parentJob = CoroutineScope(Dispatchers.IO).launch {
          for (i in 1..1000) {
               if (isActive) {
                    EnterLongRunningTask()
                    Log.d(TAG, i.toString())
               }
          }
     }
}
```

```
        delay(100)
        Log.d(TAG, "Cancelling job")
        parentJob.cancel()
        parentJob.join()
        Log.d(TAG, "Parent Completed")
    }
```

here Even if you
It will run (Execution
Resu

So use check
if (isActive)

means if coroutine is in active, then only Execute it.

---

withContext & runBlocking functions

It is Of blocking nature

```
private suspend fun executeTask(){
    Log.d(TAG, "Before")
    GlobalScope.launch {
        delay(1000)
        Log.d(TAG, "Inside")
    }
    Log.d(TAG, "After")
}
```

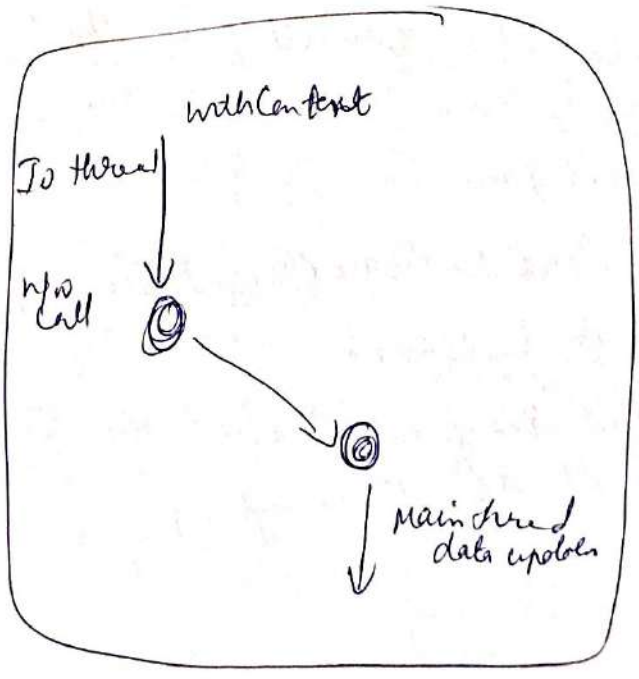
non blocking, it executes in Separate thread

```
withContext(Dispatchers.IO){
    delay(1000)
    Log.d(TAG, "Inside")
}
```

It blocks coroutine not thread

It is of blocking nature
It will suspend
Execute then only next
line will execute

O/P
Before
After
Inside



O/P
Before
Inside
After

```
MainActivity : AppCompatActivity() {
    lateinit var viewModel : ViewModel
    override fun onCreate(savedInstanceState : Bundle?) {
    }
    viewModel = ViewModelProvider(this).get(MainViewModel::class.java)
    lifecycleScope.launch {
        delay(2000)
        val intent = Intent(this@MainActivity, AnotherActivity::class.java)
        startActivity(intent)
        finish()
    }
}
```

```
class MainViewModel : ViewModel() {
    private val TAG : String = "KOTLINFO"

    init {
        viewModelScope.launch {
            while(true)
                delay(500)
            Log.d(TAG, "Hello from")
        }
    }
    override fun onCleared() {
        super.onCleared()
        Log.d(TAG, "View Model Destroyed")
    }
}
```

o/p KOTLIN Hello from
.
.
.
View Model Destroyed

runBlocking :

```
fun main() {
    GlobalScope.launch {
        delay(1000)
        println("World")
    }
    println("Hello")
    Thread.sleep(2000)
}
```

O/P Hello

```
fun main() {
    runBlocking {
        GlobalScope.launch {
            delay(1000)
            println("World")
        }
        println("Hello")
    }
}
```

O/P Hello
World

```
fun main() {
    runBlocking {
        GlobalScope.launch {
            delay(1000)
            println("World")
        }
        println("Hello")
    }
}
```

O/P Hello
World

---

runBlocking → generally can be used in test case
↳ it will make thread alive until / coroutine completes its task

---

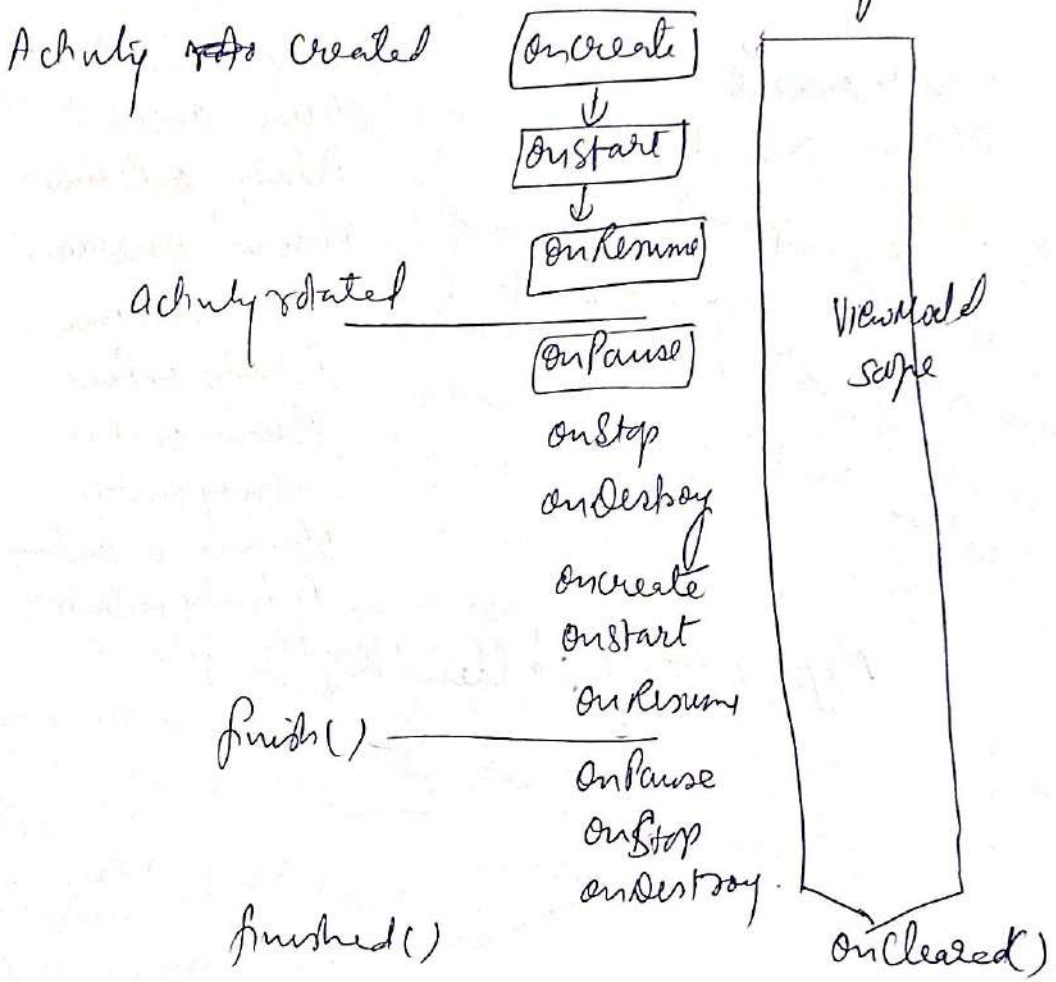
ViewModelScope & LifecycleScope

- CoroutineScope attached with ViewModels.
- Coroutines in this scope will be cancelled automatically when ViewModel is cleared! we don't need to manually cancel the coroutines

CoroutineScope attached with lifecycle (activity or frag)
Coroutine in this scope will be cancelled automatically when lifecycle is destroyed + we don't need to manual cancel the coroutines

ViewModel Lifecycle

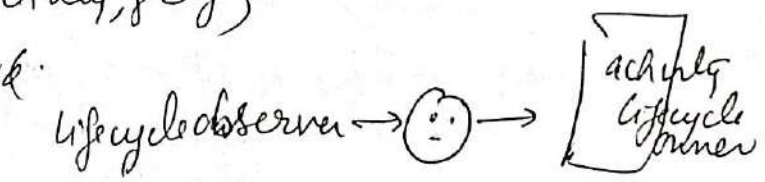
The lifecycle/lifetime of ViewModel is Exists from when you first request a ViewModel (usually on creat of activity) until the activity is finished and destroyed. onCreate may be called several times during the life of an activity, such as app is rotated, but the ViewModel survives throughout.



Lifecycle Aware Components

- Most of code is written inside activity lifecycle methods — onCreate, onResume, onPause. etc. Due to this, Activity has multiple responsibilities.
- But there are scenarios where we want to take actions based on activity lifecycle. Eg. Access User's location, Playing Video, Downloading images.

→ lifecycleowner (activity, frag)
→ Lifecycle observer.



Eg

```
Class MainActivity : App... {
    onCreate {
        lifecycle.addObserver(observer())
        Log.d("Main", "ActivityOnCreate")
    }
}
```

```
class Observer : LifecycleOb...
@OnLifecycleEvent(Lifecycle.Eve...
fun onCreate() {
    Log.d("Main", "Observer-onCreate")
}
```

O/P

Activity onCreate
Observer-onCreate

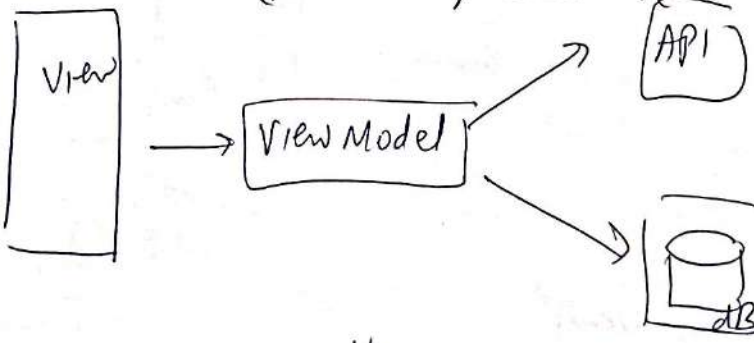
Benefit

It reduces use of code of activity, moving to different component using observer. Observer will observe the lifecycle of activity, frag.

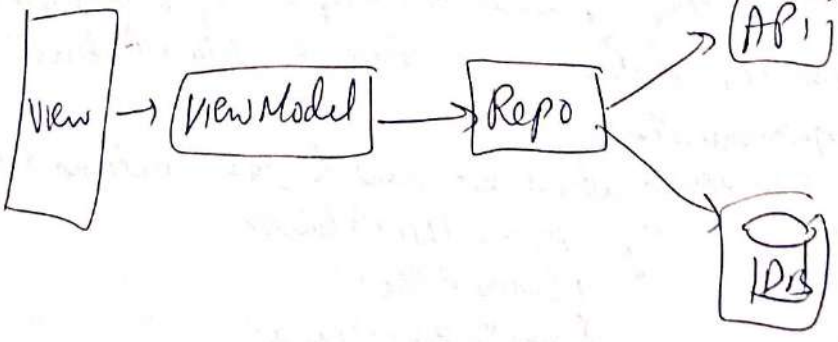
sequence of o/p.

Activity OnCreate
Observer onCreate
Activity onResume
Observer onResume
Observer onPause
Activity onPause
Observer onStop
Activity onStop
Observer onDestroy
Activity onDestroy

Repository Pattern (Third design pattern)

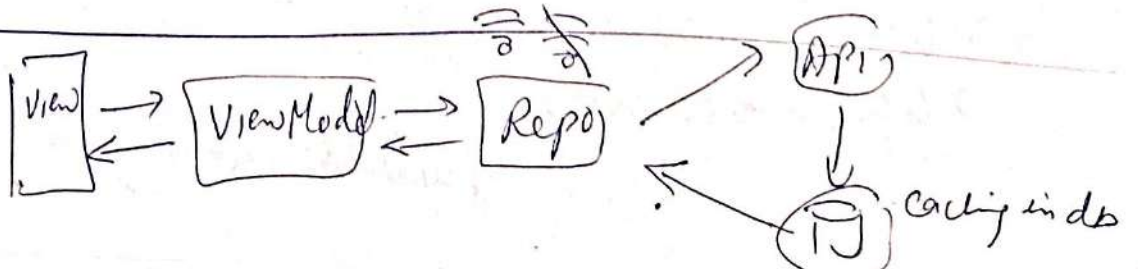


In future if there will be change in API or DB, then you will need to make changes in Every ViewModel. ViewModel can be more



Here in this pattern you will need to do changes in only one repo

Use Case



Database Migration

we are migrating from Sqlite to Room, then need to update Version and provide migration (empty are)

→ If you don't provide version, app will crash.

→ If you provide version and no migration provided – app crashes (IllegalStateException)

→ version increased, fallback to destructive migration enabled – database is cleared.
(It will not crash but data will get lost)

```
database = Room.databaseBuilder(context.getApplicationContext(),
        UserDatabase.class, "Sample.db")
    .fallbackToDestructiveMigration()
    .build()
```

→ Version increased, migration provided – data is kept

```
@Database(entities = {User.class}, version = 2)
public abstract class UserDatabase extends RoomDatabase {

    static final Migration MIGRATION_1_2 = new Migration(1,2) {
        void migrate( ) {
            // don't add anything
        }
    }

database = - - - -
    - - -
    .addMigrations(MIGRATION_1_2)
    .build();
```

Case 2 — Migration with Simple Schema changes

(increase Version, add Migration)

① @Database (entities = {User.class}, version = 3)
public abstract class UserDb extends RoomDatabase

② · Migration from 2 to 3

```
static final Migration MIGRATION_2_3 = new Migration(2,3)
{
    @Override
    public void migrate( . . ) {
        database.execSQL ("ALTER TABLE users "
                + " ADD COLUMN last_update INTEGER");
    }
};
```

③ Add migration to Room database builder.

```
database = Room. -
    . . . .
    .addMigrations (MIGRATION_1_2, MIGRATION_2_3)
    .build()
```

Note: User who is using DB version 1 want to use DB version 4 (upgrade app)

then

```
.addMigrations ( MIGRATION_1_2, MIGRATION_2_3,
    MIGRATION_3_4, MIGRATION_1_4)
    .build();
```

→ cold nature (if not Consume, it will not produce)

SharedFlow → hot nature (If not Consume, it will still produce)
(read lately)

→ There will be multiple Consumer, all will get same amount data at that time.
If any Consumer will come late, it that lost data will not be provided)

StateFlow (hot nature)
It maintain state of last value (that is latest value)

```
GlobalScope.launch(Dispatchers.Main){
    val result = producer()
    Log.d("<--"; result.value.toString())
}
```

```
private fun producer(): StateFlow<Int>{
val mutableStateFlow = MutableStateFlow(10)
    GlobalScope.launch{
        delay(2000)
        mutableStateFlow.emit(20)
        delay(2000)
        mutableStateFlow.Emit(30)
    }
    return mutableStateFlow
}
```

Hilt

@HiltAndroidApp (added at application class)

@AndroidEntryPoint (Components like activity, fragment, View become ready to be injected)

@HiltViewModel (derive from ViewModel)

@InstallIn (we need to specify scope DI Containers)
eg SingletonComponent, FragmentComponent, ActivityComponent.

```
interface UserRepository{
    fun saveUser(email: String, password: String)
}

class Repo1 @Inject constructor(): UserRepository{
    @override fun saveUser(                    ){
                }
}

class Repo2 : UserRepository{
    @Override fun saveUser(
    }
}

@InstallIn(FragmentComponent::class)
@Module
class UserModule{

    @Provides
    fun providesUserRepository(): UserRepository{
    } ? return Repo2()
```

SingletonComponent

eg want to access in activity

```
@AndroidEntryPoint
class MainFrag : Fragment(){
    @Inject
    lateinit var userRep: UserRepo
    .
    .
    .
    userRep.saveUser("","")
}
```

SOLID PRINCIPLE

1. Single Responsibility Principle
2. Open-closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

Single Responsibility : A class should have one, and only one, reason to change.

Can give Eg onBindViewHolder should avoid adding extra code.

The open-closed Principle : Sw Entities (classes, modules, functions, etc...) should be open for Extension, but closed for modification.

```
class MainRepository(
    private val auth: FirebaseAuth
){
    suspend fun loginUser(email: String, password: String){

        try{
            auth.signInWithEmailAndPassword(email, password).await()
        }
        catch(e: Exception){
            val file = File("errors.txt")
            file.appendText(
                text = e.message.toString()
            )
        }
    }
}
```

Single Responsibility

This violates Single Respon... you we can separate this to new class, method & access here.

```
class FileLogger {
    fun logError(error: String) {
        val file = File("errors.txt")
        file.appendText(
            text = error
        )
    }
}
```

This class has responsibility to logging.

```
class MainRepository(
    private val auth: FirebaseAuth,
    private val fileLogger: FileLogger
) {
    suspend fun loginUser(email: String, password: String) {
        try {
            auth.signInWithEmailAndPassword(email, password).await()
        } catch (e: Exception) {
            fileLogger.logError(e.message.toString())
        }
    }
}
```

Open-closed Principle

Classes should be open for Extension & closed for modification.

→ Calling class (Parent class) dont do modification in same class. Better to create an interface or abstract class and add the common method and SubClass will override those method in their respective class. So SubClass can Extend and parent or calling class can be closed for modification..

```
open class FileLogger {
    open fun logError(error: String) {
        val file = File("errors.txt")
        file.appendText(
            text = error
        )
    }
}
class CustomErrorFileLogger : FileLogger() {
    override fun logError(error: String) {
        val file = File("my_custom-error-file.txt")
        file.appendText(
            text = error
        )
    }
}
```

Liskov Substitution

Child classes should never break the parent class type definitions

(If there is anything to add, add in child class, avoid to add in parent class as it changes will reflect everywhere ~~child class which extend parent~~.

Interface Segregation Principle (ISP)

The Interface-Segregation States that no client should be forced to depend on methods it does not use.

Dependency Inversion Principle (DIP)

→ High-level modules should not depend on low level modules. Both should depend on abstractions.

→ Abstractions should not depend upon details. Details should depend upon abstractions.

// violation of DIP

```
// Program.java
class Program {
        public void work() {
                //...code
        }
}

// Engineer.java.
class Engineer {
        Program program;
        public void setProgram(Program p) {
                program = p;
        }
        public void manage() {
                program.work();
        }
}
```

high level module it depends on low level (Program)

```
interface IProgram {
        public void work();
}
class Program implements IProgram {
        public void work() {
                //--code
        }
}
class SuperProgram implements IProgram {
        public void work() {
                //--code
        }
}
class Engineer {
        IProgram program;
        public void setPr(IProgram p) {
                prog = p
        }
}
```

all changes only when needed not in Engineer class (high level module)

Composite Disposables

Subscriptions to streams need to be disposed at the right time in order to avoid memory leaks. This can be done very simply. Hold a reference to the disposable when created and call the dispose function at the right moment in one of the lifecycle hooks like onStop or onDestroy.

```
abstract class BaseViewModel : ViewModel(){
    var disposable : CompositeDisposable = CompositeDisposable()
```

in ViewModel:

```
fun getLoad() {
    disposable += usecase.getUser()
        .whileSubscribed{}
        .subscribeOn(...)
        .observe...
        .subscribe...
    }
}

override fun onCleared(){
    super.onCleared()
    disposable.clear()
}
```

```
class RegisterFragment : Fragment() {
private var _binding : FragmentRegisterBinding ? = null
private val binding get() = _binding !!
@Override fun onCreateView(
                (inflater, ... ) : View?
{
_binding = FragmentRegisterBinding . inflate (inflater, container, false)
binding . txtRedirect . setOnCl... {
        findNavController() . navigate(R. id . action_signdi_to_loginfrag)
    }
return binding root
/*
    val view = inflater . inflate (R.layout. fragment_signin, container, false)
    val txtRedirect = view . findViewById<TextView>(R.id. txtRedirect)
    txtRedirect . setOnCl {

    }
    return view
*/
}
override fun onDestroyView() {
        super. onDestroyView()
    } _binding = null
}
```

```
<frag...                    Passing
                            arguments
  <argument
      name = "arg1"
      argType = "integer"
      defaultValue = "0" />
</frag>

val directions = FirstFragmentDirections . actionFirstToSecond
                    (arg1 = 1234)
findNavController() . navigate (directions)
```

activity_main.xml

```
<?xml version = ...
<androidx . fragment . app . FragmentContainerView
    xmlns ...        /android
    xmlns:          res auto
    id = "@+id/ nav_host_fragment"
    name = "androidx . navigation . fragment . NavHostFragment"
        width = match_parent
        height = "1
    defaultNavHost = "true"
    app: navGraph = "@navigation / nav_graph" />
```

Kotlin Collections

Array : Mutable but has fixed size.

Collections
- ↳ Immutable collections: Read only operations.
  - Immutable List: listOf
  - Immutable Map: mapOf
  - Immutable Set: setOf
- ↳ Mutable Collections: Read and Write Both
  - Mutable List: ArrayList, arrayListOf, mutableListOf
  - Mutable Map: HashMap, hashMapOf, mutableMapOf
  - Mutable Set: mutableSetOf, hashSetOf.

```
// Elements:   32  0  0  0  0  0
// Index:      0  1  2  3  4  5
var myArray = Array<Int>(6) { 0 }   // mutable, fixed Size
myArray[0] = 32
println(myArray[0])
for (element in myArray) {
        println(element)
    }
for (index in 0..myArray.size-1)
    { println(myArray[index])
    }
```

```
var list = listOf<String>("Yogi", "Manmohan", "Vajpayee")
                // Immutable, fixed Size, Read only
list[2] = " "  // error

for (element in list) {
    println(element)
}

var list = listOf<Int>(0, 1, 2)
```

mutableListOf, arrayListOf, ArrayList → same only.

```
var list = mutableListOf<String>() // mutable, no fixed size, can use
                                   or remove elements
var list : MutableList<String> = mutableListOf<String>()

list.add("Yogi")    //0
list.add("Manmohan") //1
list[1] = "Modi"
```

```
var myMap = mapOf<Int, String>() // Immutable, fixed size,
                                 Read only.
var myMap = mapOf<Int, String>( 2 to "Yogi", 43 to "Manmohan",
                                7 to "Vajpayee")

for (key in myMap.keys) {
    println(myMap[key])          // myMap.get(key)
    }

println("Element at key: $key = ${myMap[key]}")
```

```
var myMap = HashMap<Int, String>()  // Mutable, Read/Write
                                    no fixed size
myMap.put(4, "Yogi")
myMap.put(43, "ManMohan")
myMap.put(7, "Vajpayee")

myMap.replace(43, "Modi")  // myMap.put(43, "M
```

HashMap
mutableMapOf
hashMapOf

} all same only difference is mutableMapOf returns LinkedHashMap

Set

// "Set" contains unique Elements, Value in sequence (print)

// "HashSet" also contains unique Elements but sequence is not guaranteed

```
var mySet = setOf<Int>(2, 54, 3, 1, 0, 9, 9, 9, 8)
                    // Immutable, Read only
```

```
var mySet = mutableSetOf<Int>(2, . . . . . 8)
mySet.remove(54)                    // mutable Set, Read & write both
mySet.add(100)
```

```
var mySet = hashSetOf<Int>(2, 54, 3, 1, 0, 9, 9, 9, 8)
```

---

filter and Map

```
val myNumbers: List<Int> = listOf(2, 3, 4, 6, 23, 90)
val mySmallNums: List<Int> = myNumbers.filter { it < 10 }
                    // or { num -> num < 10 }

val mySquaredNums = myNumbers.map { it * it }
                    // or { num -> num * num }

for (num in mySquaredNums) {
    println(num)
}

val mySmallSquaredNums = myNumbers.filter { it < 10 }.map
                    { it * it }
```

Example

```
var people = listOf<Person>(Person(10, "Priyank"),
                    Person(23, "Annie"), Person(17, "Sam"))
var names = people.filter { it.name.StartsWith("S") }.map
                    { it.name }
```

Filter → filter our desired elements from a collection.
map → Perform operations, modify Elements.
Predicates: Or a condition that returns True or False.
→ "all": Do all elements satisfy the predicate / condition?
→ "any": Do any element in list satisfy the predicate?
→ "count": Total elements that satisfy the predicate.
→ "find": Returns the FIRST element that satisfy predicate.

```
val myNumbers = listOf(2, 3, 4, 6, 23, 90)
val check1 = myNumbers.all({it > 10})  // are all elements greater than 10?
println(check1)

val check2 = myNumbers.any({it > 10})
// Does any of these elements satisfy the Predicate?
println(check2)

val totalCount : Int = myNumbers.count{it > 10}
// no. of elements that satisfy the predicate

val num = myNumbers.find{it > 10}  // returns the first number that matches the predicate.
```

Abstract

By default in Kotlin, all classes, methods are public & final. To inherit class, make it open. To override any class function, make function as open.

→ By default, all the abstract methods are open in nature so no need to explicitly make it open.

→ abstract fun method dont contain body.

→ If method is abstract then you need to make that class as abstract.

→ we cant create / instantiate object of abstract class.

→ If there is abstract method in Super class, then you must override that abstract method in Subclass.

But if there is open method in Super class, then you need not create override method in the Subclass (its optional)

```
abstract class Person {
        abstract var name: String
        abstract fun eat()          // abstract prop are "open" by default
        open fun getHeight() { }    // open fun ready to be overridden
        fun goToSchool() { }        // public & final by default.
    }

class Indian : Person() {           // Sub class or Derived class.
    override var name : String = "dummy"
    override fun eat() {
                    }
}
```

⇒ The Role of abstract class is to just provide set of methods and properties. (no body when declared)

Interface

```
interface MyInterfaceListener {
    var name: String    // properties in interface are abstract by default
    fun onTouch()       // methods in interface are abstract by default.
    fun onClick() {     // Normal methods are public and open by default
                        // Not final.
    }
}
```

```
class MyButton : MyInterfaceListener {
    override var name: String = "dummy"
    override fun onTouch() {
        println("Btn was touched")
    }
    override fun onClick() {    // optional to override
        println("Btn was clicked")    super.onClick()
    }
}
```

Design Patterns

Its basically a solution where to resolve problem which we get over an over in programming.

① Singleton Pattern
↓
Single instance

```
object Singleton {
    fun doSomething() {
    }
}
```

② Factory Pattern

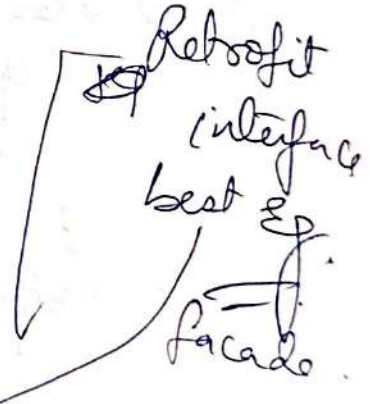
Eg ViewModelFactory (Depending on the class type of ViewModel, it will return particular ViewModel.)

```
enum class DialogType {
    DIALOG_CREATE_CHAT,
    DIALOG_DELETE_MESSAGE,
    DIALOG_EDIT_MESSAGE,
}

sealed class Dialog {
    object CreateChatDialog : Dialog()
    " DeleteMessageDialog : "
    " EditMessageDialog : "
}

object DialogFactory {
    fun createDialog(dialogType: DialogType): Dialog {
        return when(dialogType) {
            DialogType.DIALOG_CREATE_CHAT -> Dialog.CreateChatDialog
            DialogType.DIALOG_DELETE_MESSAGE -> Dialog.DeleteMessageDialog
        }
    }
}
```

3) Builder Pattern is a Pattern that allows you to produce types and representations of an object using the same Constr...

```
class Hamburger private constructor(
    val cheese : Boolean,
    val onions : Boolean
){
    class Builder {
        private var cheese : Boolean = true
                "    "  onions :   "   "
        fun cheese(value : Boolean) = apply { cheese = value }
        fun onions(value : Boolean) =  "    { onions = value }
        fun build() = Hamburger(cheese, onions)
    }
}
```

```
class MainActivity - - - {
    onCreate . . .
    val hamburger = Hamburger.Builder()
        .cheese(true)
        .onions(false)
        .build()
}
```



---

4) Facade . Hide code behind how retrofit & network work.

```
interface MyApi {
    @GET("hamburger")
    suspend fun getHamburgers(): List<Hamburger>
}
```

Dependency Injection.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Singleton
    @Provides
    fun provideHamburger(): Hamburger {
        return Hamburger.Builder()
            .cheese(true)
            .onions(false)
            .build()
    }
}
```

DI → is passing of dependency to a dependent object

---

(6) Adapter Pattern. Eg Recyclerview adapter

Adapter → OnCreateViewHolder, onBindViewHolder, return item
getItemCount.

Memory Leaks

because it is singleton in nature

It is a failure to release unused objects from the memory.

Means that there are unused objects in the application that the GC cannot clear from memory because

↓

Garbage Collector

Solution: (1) use getApplicationContext()

(2) use WeakReference when ↳ GC will understand to clean it

Types of memory Leaks

- Application terminates (permanent)
- when method terminates (temporary)
- (3) findViewById as it has reference to activity. Causes memory leak. So better to dereference it in onDestroy method

RAM

- Stack (used for static memory allocation)
- Heap (used for dynamic memory allocation)

A memory leak can easily occur in android when Asynctask, handlers, Singletons, Thread, and other components are used incorrectly.

Example: Suppose you have class MainActivity, inside that class you have another class ABC in which there is a reference of MainActivity. In this class ABC you have one method of downloading one song for 20 sec.

Now once it starts downloading song, before 20 sec you rotated the device or close the application. Then in that case activity will be still alive because its class ABC (inner class) holding a reference of activity class, here GC cannot clear it from the heap memory.

So, we have a memory leak, the task still running in the background and the activity still alive.

Example 2 The memory leak could happen when we initialize the Singleton from the activity and pass a long-lived activity-context reference to the Singleton constructor when we initialized it.

```
onCreate(...){
    SingletonManager.getInstance(this);
}
```

How to avoid memory leak here?

Use application-Context instead of activity-Context

```
SingletonManager.getInstance(getApplicationContext());
```

Do not keep long-lived references to a Context-activity

In situation in which activity-Context is required. Remove the Context reference when it is no longer needed. Eg when the activity is destroyed.

Example 3, using the activity-Context to show a Toast is a common mistake that can easily lead to a leak.

```
Toast.makeText(applicationContext, text, duration)
```

Why Should we Care about Memory Leaks?

1. Lags in the app.
2. ANR (Application not Responding) (which takes more than 5 sec to respond) (when UI thread is blocked for long)
3. Crash - OutOfMemoryError Exception. (as JVM can not allocate an object because it is out of memory)

Tools that can help you identify Leaks

1. **LeakCanary** → great tool for detecting memory leaks in an app
2. **Profiler** → View the Java heap and memory allocations with memory profiler.

Unit Testing.

→ we used MockitoJunitRunner library. @Mock used to mock the specific class, services.

@Before annotation is used to ~~set~~ do setup, (like to open Mocks)

```
MockitoAnnotations.openMocks(this)
RxAndroidPlugins.setInitMainThreadSchedulerHandler
```

@Test is used to write unit testCase functions.

@After is used tearDown() to reset

```
RxAndroidPlugins.reset()
```

Whenever → use it when you want the mock to return particular value when particular method is called.

```
Whenever(.....).thenReturn(Observable.just(response))
```

You can create TestObserver then Subscribe ~~it to~~ request to it

```
request.subscribe(testObserver)
testObserver.await()
            .assertComplete()
            .assertNoErrors()
            .assertValue(response)
```

assertValue → Assert that this TestObserver/TestSubscriber received Exactly one onNext value which is Equal to the given value to with respect to Objects.equals (Object, Object)

Parameters : Value - the value to Expect

Returns: this

Example

```
@Test
fun test_getLoyaltyDetail () {
    val response = LoyaltyDetail()
    val request = loginViewModel.getLoyaltyDetail(response)
val testObserver = TestObserver<LoyaltyDetail>()
request.subscribe(testObserver)
testObserver.await()
testObserver.assertComplete()
testobserver.assertNoErrors()
testObserver.assertValue(response)
}
```

Agile It is a continuous iteration lifecycle model used for developing and testing the s/w (Using SDLC)
The process of development divided into Sprints.

Waterfall It is a linear Sequential lifecycle model for developing and testing the s/w.

Advantages of agile

Customer Satisfaction, welcome changes, deliver frequently, work together, Simplicity, working s/w.

Scrum → Implementation of agile used that need to handle constant changing requirement * Daily standup helps with collaboration.

Product backlog, **Sprint backlog**, **Retro meeting**, **Sprint planning meeting**

- Product backlog: product owner keeps items related to product
- Sprint backlog: Contains items related to Sprint
- Retro meeting: involve whole team 2-3 hrs
- Sprint planning meeting: to discuss on items from product backlog.

Kanban It is nothing but a Kanban Board. It helps to optimize the flow of task b/w different teams. Kanban process visualizes the workflow which is easy to understand.

DevOps (Development + Operations)
(Plan, Code, Build, Test, Release, Deploy, Operate, Monitor)
DevOps is possible beg of CI / CD

CI → Continous Integration, CD → Continuous delivery.

Continuous Integration → It is a practice that integrates code into a shared repository. It uses automated verifications for the early detection of problems. It doesn't eliminate bug but helps in finding and removing them Easily.

Continuous Delivery is phase in which changes are made in code before deploying. Team decides what is to be deployed to customers and when.

Azure Devops, Jenkins are tools (CI / CD) tools.
pipeline → code checkout + build creation + run test cases + Sonar qube + merging into upper branch + pushing to cloud/appcenter + push to playstore.

Proguard Vs R8

8.5% reduction (10% reduction)

→ tool that converts java byte code into an optimized dex code.

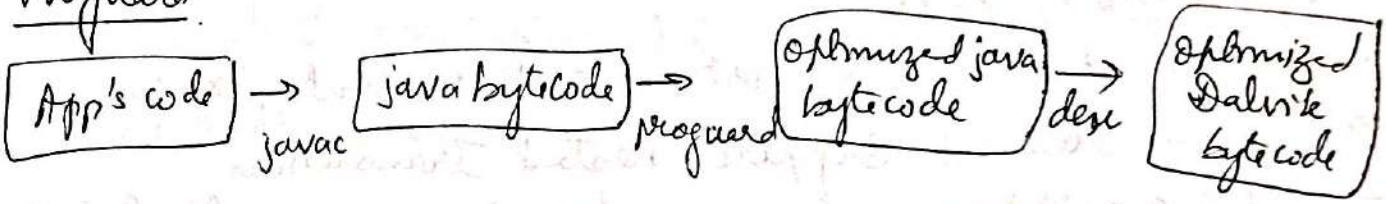
→ It removes unused classes, methods. It runs on Compile time.

→ R8 uses Proguard rules to modify its default behaviour

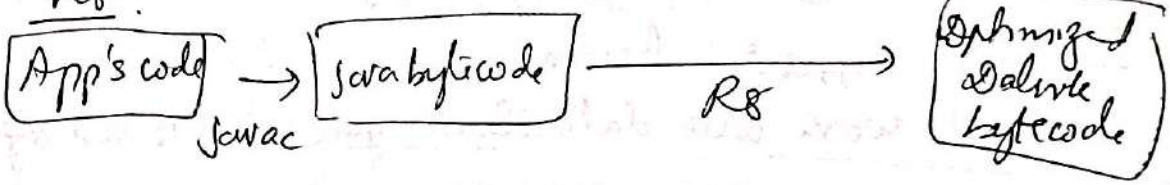
→ Gradle plugin above 3.4.0 or more, project uses R8 by default

→ R8 has more Kotlin support compared to Proguard.

Proguard



R8



If you want aggressive obfuscation in R8, then add below

android.enableR8.fullMode = true

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
    }
}
```

Security → To provide security, R8 provides code obfuscation. This means that it will take all class names, variables and functions in your app, renamed to short unreadable names before building the release version of app.

Sqlite → is a database Engine written in C.
To interact with db, you need to run queries.

Realm → is a NOSQL db. data is mapped to objects in a realm file. classes are used to define the schema (It increases size of app) (faster than Room)

Room → Room is not a db. but simply a layer on top of Sqlite and bought a huge array of customizations that Sqlite lacked in its purest form. Room is part of jetpack Components.

Limitations of Realms

→ Realm does not support auto-incrementing of id.
→ Realm does not support nested transactions.
→ It's a nightmare when we have to merge multiple tables in Realm.
→ Realm does not support inheritance.
→ Realm does not work with data classes you need to use regular open classes.

→ If we access the realm database to forget to close it, it may produce manipulated data. So, the realm is less reliable from the Room.
→ Room accomplishes about 100% unit test coverage ability.

Android launch modes

1. Standard
2. SingleTop
3. SingleTask
4. SingleInstance

Before launch
A → B → C → D

after launch
A → B → C → D → B

B has Standard Launch mode

new instance of B is created again

Before launch → A → B → C → D

→ if we launch C, new instance of C is not on top

A → B → C → D → C

→ If we launch C now, a new instance of C will not be created.

A → B → C → D → C

SingleTask (singleton)

before A → B → C

launch D A → B → C → D

launch B ⇒ A → B (as B already present)
all above B will get destroyed.

SingleInstance

Case 1 before A → B → C

D has SingleInstance launch D
task 1 ⇒ A → B → C
task 2 ⇒ D

add E
task 1 ⇒ A → B → C → E
task 2 ⇒ D

Case 2

~~before~~ before task 1 ⇒ A → B → C
task 2 ⇒ D

launch D
↳ again D will call old instance will get called.

Custom Views

```
class CustomToolTip : LinearLayout {
private var b... CustomToolTipBinding? = null
:
constructor(   )
constr..
con... init()
private init(   ) {

}
```

XML call like below

```
< com.noteaid.android.customView..
    id..

    text
    :
/>
```

XML

```
<include
    id
    layout = "@layout/layout_ren.."
    :
/>
```

Databinding

```
<layout ... >
  <data>
   <variable
      name="viewModel"
      type="task.ui.view ... LeanViewModel" />
   <import type="android.view.View" />
  </data>
  <a ... constraintLayout
  :
   <include
android:visibility = "@{viewModel.toasterVisibility ? View.VISIBLE :
                                                     View.GONE}"
</layout>
```

ViewModel

```
var toasterVisibility = ObservableField<Boolean>(false)
var collectionItemText = ObservableField("")
```

toasterVisibility.set(false)

```
} collectionItemText.
set(activityCont.getString
    (TranslationKeys:...))
```

xml "@{viewModel.collectionItemText}"

Map Vs FlatMap

Map → returns a list containing the results of applying the certain set of transformation to each element in the original collection.

Map → is used to transform a list based on certain conditions.

FlatMap → is used to combine all the items of lists into one list.

```
val numbers = listOf(1,2,3,4,5)
val squaredNumbers = mutableListOf<Int>()
numbers.forEach {
    squaredNumbers.add(it * it)
}
```

```
val numbers = listOf(1,2,3,4,5)
val squaredNumbers = numbers.map
    { it * it
    }
```

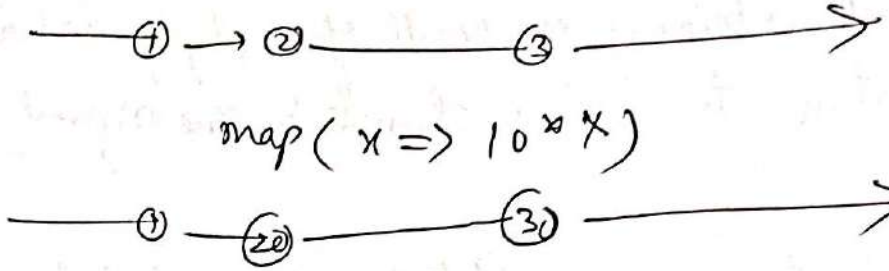
same o/p [1, 4, 9, 16, 25]

```
data class Motor(val name: String, val model: Int)

val cars = listOf(
        Motor("Swift", 2016), , )

val bikes = listOf(
        Motor("R-15", 2018), Motor( ) )

val allVehicles = mutableListOf<Motor>()

allVehicles.addAll(cars)
"                (bikes).
```

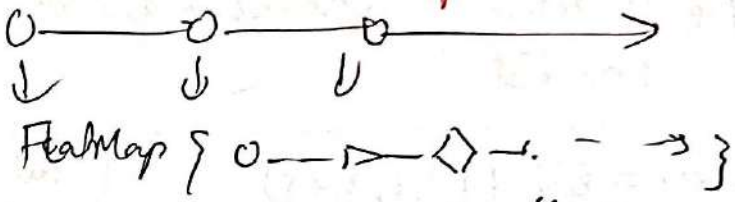
Using flatMap

```
val vehicles = listOf(cars, bikes)
val allVehicles = vehicles.flatMap { it }
```

Map transforms the items Emitted by an Observable by applying a function to Each item.

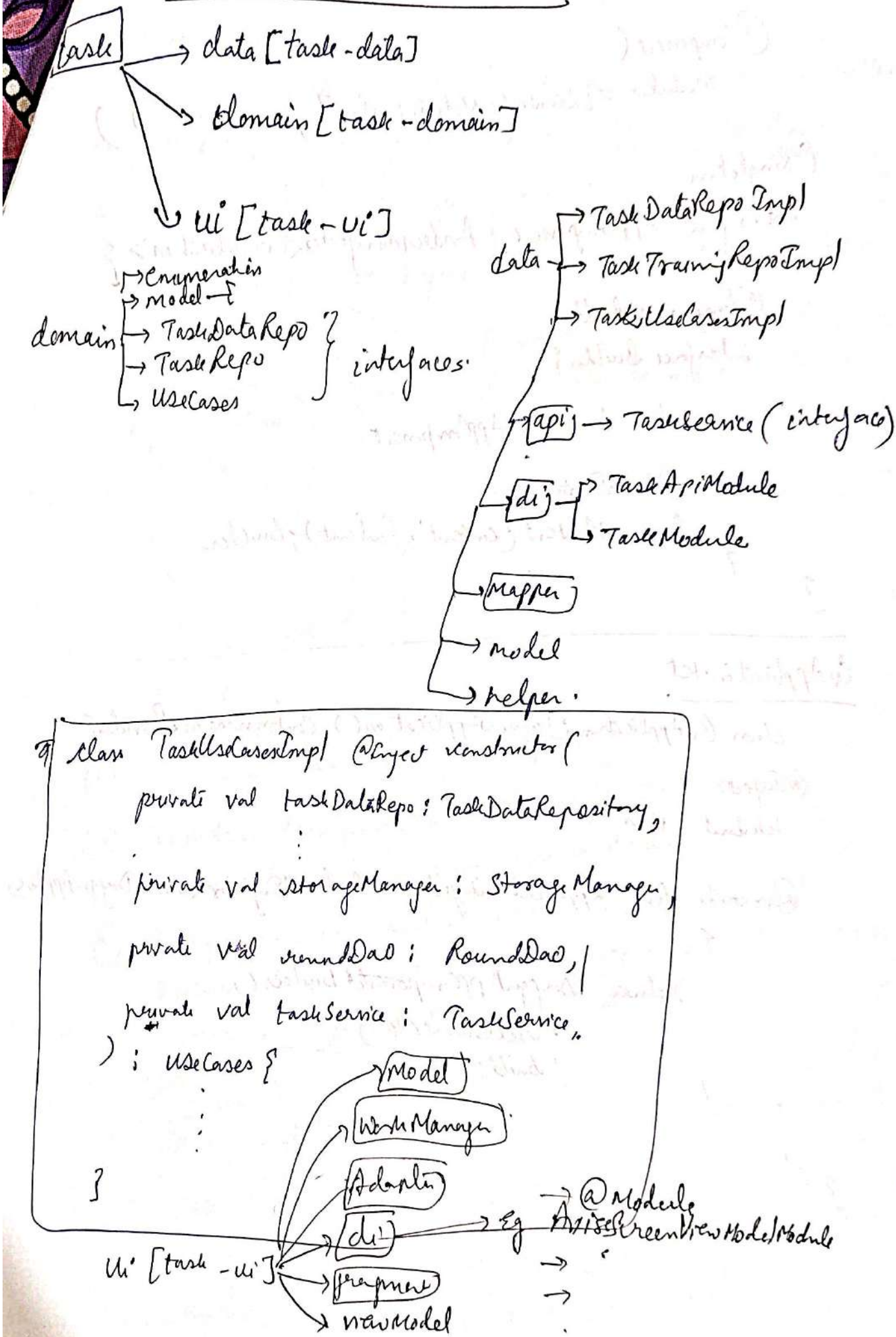


FlatMap transforms the items Emitted by an Observable into Observables.



FlatMap mapper returns an Observable itself, so it is used to map over asynchronous opr.

Clean Code Architecture



app-Module

AppComponent.kt

```
@Component(
    modules = [LeanLockerModule::class, ..... ])
@Singleton
interface AppComponent : AndroidInjector<CvApplication> {

    @Component.Builder
    interface Builder {

        fun build(): AppComponent

        @BindsInstance
        fun setContext(context: Context): Builder
    }
}
```

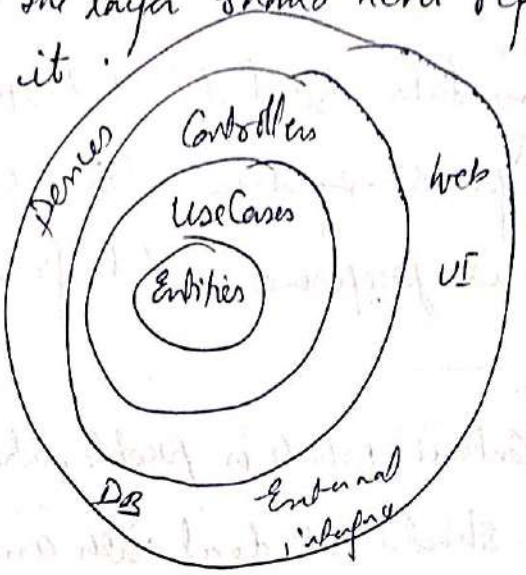
CvApplication.kt

```
class CvApplication : DaggerApplication(), Configuration.Provider {

@Inject
lateinit var ...

@override fun applicationInjector(): AndroidInjector<out DaggerApplication>
    {
    : return DaggerAppComponent.builder()
                .setContext(this)
                .build()
    }

}
```

Clean Architecture layer as a sphere that fades in as we go towards the center.

The center has no concrete details of how any specific task is handled. It just defines the rules/policies that should be followed by the S/w (that is business logic of the S/w)

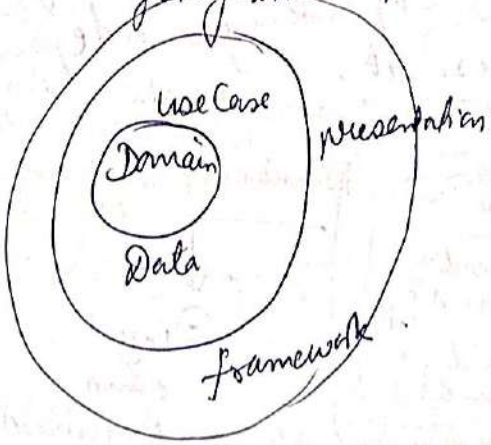
→ The layer should never reference anything in the layer above it.



As we go inside the center, the details should become abstract.

Advantages

1. It makes it easy for system to change.
2. The code is decoupled so that you can treat every layer as a black box, helping in testing code.
3. Every module defines its purpose rather than its details. This helps what an app does rather than going in-depth into technical details.



ViewModel Vs AndroidViewModel

⇒ AndroidViewModel requires a ApplicationContext, can be used to access resources or services that require a Context, such as SharedPreferences, Resources or ContentResolver.

⇒ AndroidViewModel is a subclass of ViewModel.

⇒ ViewModel is used for storing data related to UI state,

⇒ AndroidViewModel is used for storing data related to application state such as user preferences or data from Database

---

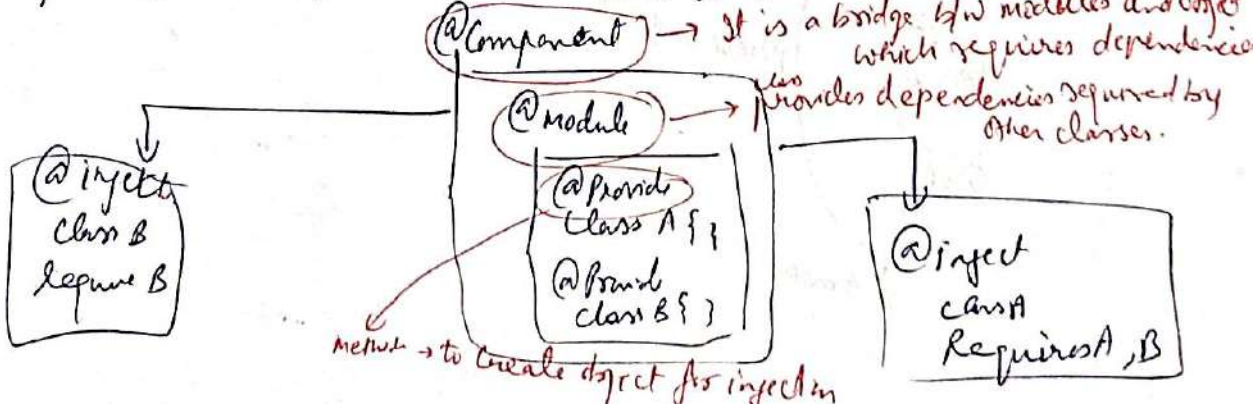
⇒ AVM provides application Context which is problematic for unit testing. Unit tests should not deal with any of Android Lifecycle such as Context.

Dagger2

⇒ It is an Dependency Injection framework maintained by Google
new version is Hilt.

⇒ 3 building blocks → Modules, Provides, Components
Dagger2 uses generated code rather than reflection (like in Dagger1)

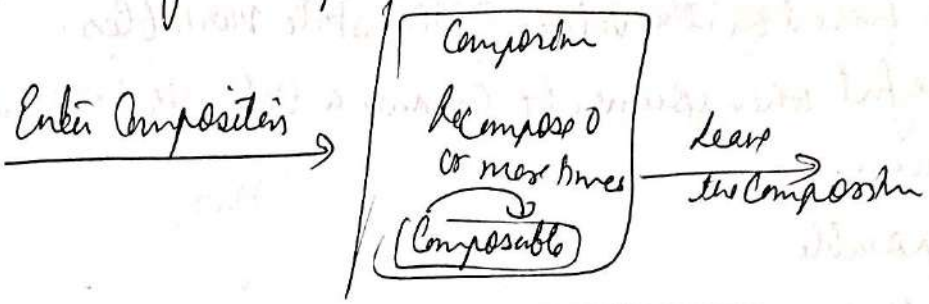
⇒ An injection is the passing of a dependency to a dependent object (a client) that would use it.



Compose

Composition is when Jetpack Compose re-Executes the Composables that may have changed in response to state changes, and then updates the Composition to reflect any changes.

A Composition can only be produced by an initial composition and updated by recomposition.



Remember is a Composable function that can be used to cache expensive opr.

```
Val state: Int = remember { 1 }

                                  can change
Val state: MutableState<Int> = remember { mutableStateOf(1) }

Val state: MutableState<Int> = rememberSaveable
                                { mutableStateOf(1) }
```

Survive Config changes

Side Effects in Compose

It is a change of state of the app that happens outside the scope of composable function.

=) ① LaunchedEffect is a Composable fun that is used to launch a coroutine inside the scope of Composable, when LaunchedEffect Enter Composition, it launches coroutine & cancels when it leaves composition.

clean code architecture.

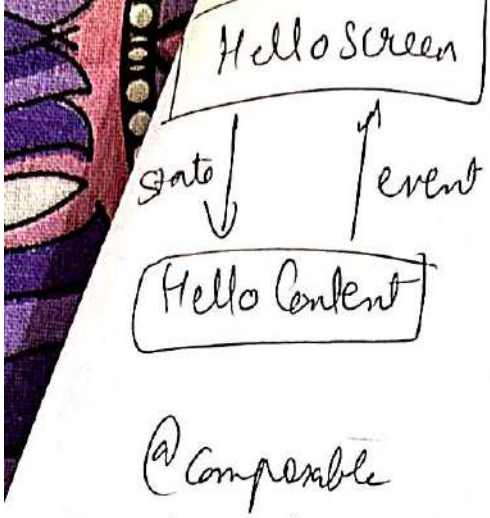
SnapshotFlow
rememberCoroutineScope
rememberUpdatedState
DisposableEffect.
SideEffect
ProduceState
derivedStateOf → is a Composable that can be used to derive new state based on the values of other state variables.
It is useful when you need to compute a value that depends on other values.

Eg
```
@Composable
fun MyComponent() {
    var firstName by remember { mutableStateOf("") }
    var lastName by remember { mutableStateOf("") }
    val fullName = derivedStateOf {
        "$firstName $lastName"
    }
    Text(text = "Full Name: $fullName")
}
```

Stateful → A Composable that uses remember to store an object creates internal state, making the Composable stateful.

State hoisting → A stateless Composable is a Composable that doesn't hold any state. An easy way to achieve stateless is by using state hoisting.

State hoisting → is a pattern of moving state to a Composable caller to make a composable stateless.




```
@Composable
fun HelloScreen() {
  var name by rememberSaveable{mutableStateOf
                                ("")}
  HelloContent(name = name, onNameChange =
                                {name = it})
}
```

```
@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit){
    Column(          ){
    Text(
    ,
    on - ( value = name, onValueChange = onNameChang - - )
} }
```

Column Hello Android

Row Hello And

Unit testing and UI testing

→ we did unit testing using the Mockito ~~JUnitRunner~~ lib.

~~Where we can~~ first we need to create a test class under src/test

@Test annotation ~~on~~ is used for creating test functions.

assertTrue, assertFalse } method to check/verify methods used in the app.

@Before → tells that we must set up the dependency Even before running the test.

@Mock annotation, we can mock any class in Mockito, by mocking a specific class, we create a mock object of that class.

whenever( ) . thenReturn(...)

TestObserver

request . subscribe (testObserver)
. await ()
. assertComplete ()
. assertNoErrors
. assertValue (response)

Comparison
Expected & mocked object

Q. How to use rest api in the android app?

→ To use rest api, we have retrofit lib. Add that in dependency in gradle.

1. Create ApiService interface having all methods like GET, Post..
2. then create retrobuilder class to intantiate the retrofit.

```
val retro = Retrofit.Builder()
        .baseUrl(BASEURL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
}
```

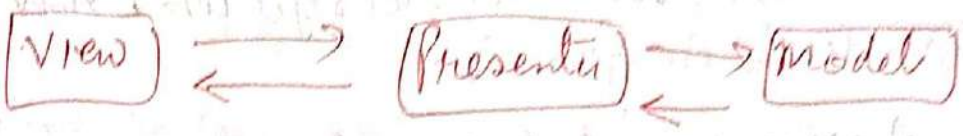
```
Val apiService : ApiService by lazy {
    retrofit.create(ApiService::class.java)
}
```

3. class MainRepo {
    suspend fun getUsers(): GetUsers {
        return RetrofitBuilder.apiService.getUsers()
    }
}

ApiService

```
interface ApiService {
    @GET(ServerConst.GETUSER)
    suspend fun getUsers(): GetUsers
}
```

MVP Vs MVVM



(1) Tight Coupling. For Each View (Activity / frag) we require a presenter. Presenter holds a reference to the View and view holds a reference to the presenter.

~~Testing is difficult.~~

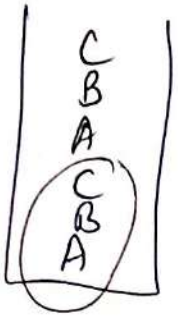
As the Complexity of view increases so does the maintenance and handling of this relationship.

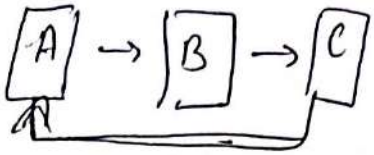
(2) Testability Once Presenter are hard bound to Views, unit tests becomes slightly difficult as there is a dependency of the View.

ViewModel are Even more Unit test friendly as they just expose the State and hence can be independently tested without requiring the need for testing how data will be consumed. There is no dependency of the View.

Navigation Graph Component

popUpTo & popUpToInclusive

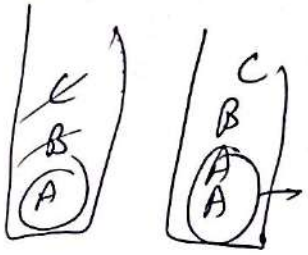




C to A => popUpTo : A

=popUpToInclusive : true

popUpTo : A
popUpToInclusive : false



Team Lead Role Questions

What is the difference b/w a team leader and a team manager?
A team leader inspires and motivates team members to achieve their goals together, while a team manager handles tasks and responsibilities of the team and make sure that others get their work done.

When do you consider success for you as a leader?
I measure success in terms of the goals that the team achieves. If a team member is successful, then it indicates success for my leadership.

What is the most difficult part of being a leader?
Being a team leader, sometimes we can feel alone in some ways in spite of being part of a team. The leader's responsibility is to see the end goal and objectives of an organization and guide others towards it. When others are not on the same track, the leader has to be the only one to ensure they are back on board.

What qualities should a team leader have?
Acknowledgment, appreciation and giving due credit, active listening, showing commitment, having a clear vision, investing in the team's future, acting with integrity, activity objectively, motivating others.

What is your greatest weakness?
Sometimes I must delegate tasks to others that I know I can do better. But if I do not delegate, I could end up with my hands too full of work to handle myself. I have learnt time management strategies to figure out how to effectively manage tasks to overcome this weakness.

What is your greatest strengths?

I believe I can lead and inspire a team to perform their best and try to accomplish goals. I can achieve this through relationship building, being motivated about the goals, and influencing others around me.

How would your teams members and Colleagues describe you?

How to resolve conflicts in team as an Android lead?

Resolving conflicts as an android lead involves Effective communication, active listening and a collaborative approach.

1) Understand Perspectives → Take the time to understand each team member's perspective on the conflict. Actively listen to their concerns.
2) We can have Private discussions. to get a clearer understanding of the issues.
3) Stay Neutral, remain neutral and avoid taking sides. We should focus on finding a resolution rather than blaming them.
4) Encourage Open Communication → I will encourage them to communicate directly with each other to resolve misunderstandings.
5) Set Clear Expectations Ensure everyone understands their roles and responsibilities.
6) Follow Up, follow up with involved parties to ensure that conflict has been fully addressed.
7) Learn & Improve, Use conflicts as learning opportunities for both individuals and team.

By taking a proactive and empathetic approach to conflict resolution, we can create a positive team environment and promote effective collaboration.

Difference between Mvvm and Clean Code Architecture

Clean architecture actually separates a whole app into presentation, domain and data layers.
However, mvvm on the other hand, is more just a set of rules that determines how data that is coming from domain should be displayed on the UI.

How to lead an android development team?

I think leading an android development team involves a combination of technical expertise, effective communication and strong leadership skills.

Some of the important skills are clear communication and setting clear goals. Lead should be able to clearly communicate with the team about the goals. As a lead, first I do analysis of stories/goals and then do make a rough road map. we do have a grooming session where we all go through the stories, understand it and if any blocker, we make a note of it, so that in sprint planning meeting, we ~~can give~~ are clear about story and can ask anything which is having doubt. So, In this way we set our clear Goals.

Apart from this, as a lead we do delegate the task Effectively based on teams members strength and skills. we do have internal team meeting where we discuss about daily task and we address challenges promptly. we seek input from team members when problem-solving. As a team, we do code review of code each other. If there is any doubt, we discuss about that.

We also keep an eye on firebase crashlytics, if there is any prod bug. we make a story of it and assign it to developers based on priority.

Staying technically proficient is also important. I try to be stay updated on latest trends & technologies in android development.

we should be adaptable also → If there is any change in feature/story, we do take it positively and adjust our strategies accordingly.

Empower team members → we encourage our team members to take ownership of their tasks.

from time-time we also celebrate our achievement

I also believe in team members well-being → I encourage a healthy work environment, Enhance productivity and Job-Satisfaction.