ANDROID

ANDROID INTERVIEW GUIDE

Land your dream Android Developer Job

Kotlin

The best way to pass any Android interview is to know your staff very well. This book packs questions, answers, red flags and general guidance for Android developers   to ace technical interviews.

J A M E S   B R I G H T

ANDROID INTERVIEW GUIDE

v 1.0.0
JAMES BRIGHT

The Android Interview Guide v1.0

James Bright

Trademarks

INTRODUCTION

You have to be prepared for a technical Android interview. Android grows and evolves rapidly. So, it is very hard to catch it up. Most of the times you are asked some concepts that you have never heard before.

We have gathered the most common Android interview questions in this book. Not only questions, but also some recommendations and red flags.

When you read this book, you will learn what to say and what not to.

This is not a traditionally typed book. We have provided a human speakable language for answers. You will find lots of “we” and “i” in this book. You are provided real-world questions and real-world answers.

That means you don’t have to think how to evaluate answer to your daily speaking language. Because we have already provided this for you. That is the advantage of this book.

We recommend you to read every question and answer for ten times. Yes, 10 times! Repetitively! That is the best method for mastering. So that you will be ready for every kind of question.

When replying, you will sound confidently. That is going to influence you interviewer. This is the key of success in a job interview.

We have also provided lots of source code for almost every question. We highly recommend you to write down codes on a piece of paper while studying that question. Because you may be supposed to write code on whiteboard in the interview.

Please don’t forget, you should read every question and answer 10 times and write down the code on a paper. And be self-confident.

PART 1

BASIC LEVEL QUESTIONS

CHAPTER 1

ACTIVITY

1. What are the lifecycle methods of an Activity?

onCreate(): This is when the Activity is first created. This is where we create views, get data from bundle and initialize things.

onStart(): Called when the Activity is becoming visible to user. But still it isn’t on foreground yet.

onResume(): Called when Activity is on foreground and it will start interacting with user. Activity is now at the top of activity stack.

onPause(): Called when activity is going into the background. But has not yet been killed. This is the best place to free system resources that used by app.

onStop(): Called when your activity is no longer visible to user.

onDestroy(): Called when activity is finishing.

onRestart(): Called when your activity is stopped, prior to it being started again.

2. Can you explain order of callback methods when an Android device is rotated?

When an Android device is rotated, activity is destroyed and re-created again. So the lifecycle methods are in order of:

onPause() → onStop() → onDestroy() → onCreate() → onStart() → onResume()

3. You do not want your activity to be destroyed and re-created on configuration changes. How can you achieve this?

For this purpose, Android Framework provides us a feature called **onConfigChanges** . We need to define this keyword for our activity in AndroidManifest.xml file.

AndroidManifest.xml

```
< activity

   android :name= ".MyActivity"

   android :configChanges= "orientation"

   android :label= "@string/app_name" >
```

From now on, if device is rotated while our activity is on foreground, it won't be destroyed and re-created. Instead a callback method will be invoked. We may override onConfigurationChanged method of Activity or Fragment to understand configuration changes.

```
override fun onConfigurationChanged(newConfig: Configuration) {
   super .onConfigurationChanged(newConfig)

   // Checks the orientation of the screen
   if (newConfig. orientation === Configuration. ORIENTATION_LANDSCAPE
) {
        Toast.makeText( this , "landscape" , Toast. LENGTH_SHORT ).show()
    } else if (newConfig. orientation === Configuration.
ORIENTATION_PORTRAIT ) {
        Toast.makeText( this , "portrait" , Toast. LENGTH_SHORT ).show()
    }
}
```

4. We have Activity A which is on the foreground. We call Activity B using startActivity() method. Can you explain order of callback methods that are called while Activity A goes to background and Activity B comes to foreground.

First, Activity A goes into background, so **onPause()** and **onStop()** are called. Then Activity B is created and goes into foreground, so **onCreate()** , **onStart()** and **onStop()** are called.

So the order is:

- **onPause()** of Activity A
- **onStop()** of Activity A
- **onCreate()** of Activity B
- **onStart()** of Activity B
- **onResume()** of Activity B

Be aware that **onDestroy()** is not called yet since Activity A hasn’t been killed yet. It is still in activity stack.

5. What are “launch modes”?

Launch Mode determines that activity is going to be stored in activity stack or not. Also it determines the position of it.

There are 4 different launch modes.
Standart: It creates a new instance of an activity in the task from which it was started. Multiple instances of an activity can be created and be added to same or different tasks.

Example: Suppose there is an activity stack of A → B → C.
Now if we launch Activity B again, the new stack will be A → B → C → B.

SingleTop: It is the same as the standard, except if there is a previous instance of the activity that exists in the **top of the stack** , then it will not create a new instance.

Example: Suppose we have a stack of A → B.
Now if we launch C, the new stack will be A → B → C.
And then if we launch C again, the stack will remain same A → B → C since C is already at the top of the stack.

SingleTask: A new task will always be created and a new instance will be pushed to the task as the root one.

Example: Suppose we have a stack of A → B → C → D.
Now if we launch D, the stack will remain same A → B → C → D.
But the *onNewIntent()* method of D will also be invoked.
Then if we launch B, new stack will be A → B. Obvious that C and D are destroyed and *onNewIntent()* of B is invoked.

SingleInstance: Same as SingleTask but the system does not launch any activities in the same task as this activity. If new activities are launched, they are launched in a separate task.

Example: Suppose we have a stack of A → B → C → D.
Now if we launch B again, the new stack will be :
Task 1: A → B → C.
Task 2: D

6. What is “Context”? Why do we need it? How can we gather it?

A Context is a class that is used to resolve resources, obtaining access to databases and preferences, and so on. An Android app has activities. Context is like a handle to the environment your application is currently running in.

Application Context: This context is tied to the lifecycle of an application. The application context can be used where you need a context whose lifecycle is separate from the current context or when you are passing a context beyond the scope of an activity.

```
// How to retrieve applicationContext in an Activity
applicationContext


// How to retrieve applicationContext in a Fragment
context ?. applicationContext ?

// How to use applicationContext
applicationContext .fileList()
```

Activity Context: This context is available in an activity. This context is tied to the lifecycle of an activity. The activity context should be used when you are passing the context in the scope of an activity or you need the context whose lifecycle is attached to the current context.

An Activity itself is a Context, so you don’t have to retrieve it. Almost all of the Context related things are defined in **ContextWrapper** Class which extends **Context** Class.

7. What are the components of an Android App?

A typical Android application has the following components:

- **Activity:** Main component of an Android app. It represents screens and allows us to reach system resources.
- **Fragment:** Introduced for different screen sizes. Can be used for a whole screen or just a little part of it.
- **Application:** A Singleton class to represent app itself.
- **Service:** Used to perform background functionalities. (Some on main thread, some on worker threads.)
- **Intent:** A mechanism to perform inter connection between Activities and passing data.
- **Broadcast:** A message between apps or Activities.
- **Notification:** A message to user’s attention.
- **Content Provider:** Used to share data between applications.

8. How can you save data before an Activity is destroyed?

We can save data using **onSaveInstanceState()** method before an Activity is destroyed. This method is generally called between **onPause()** and **onStop()** . (But no guarantee)

When Activity is recreated after it was previously destroyed, we can recover the saved state from the **Bundle** that system passes to new Activity.

How to save it?

```
override fun onSaveInstanceState(outState: Bundle?) {
   // Save the user's current state
   outState?. run {
      putInt(STATE_ID, userId)
     putString(STATE_NAME, userName)
   }

   // Always call the superclass so it can save the view hierarchy state
   super .onSaveInstanceState(outState)
}
```

How to retrieve back this data?

This Bundle data is passed to both **onCreate()** method and **onRestoreInstanceState()** method.

```
override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
   super .onRestoreInstanceState(savedInstanceState)

   var id = savedInstanceState?.getInt(STATE_ID)
```

```
    var name = savedInstanceState?.getString(STATE_NAME)
}
```

Or in **onCreate()** method.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super .onCreate(savedInstanceState)
   setContentView(R.layout. activity_main )

    var id = savedInstanceState?.getInt(STATE_ID)
    var name = savedInstanceState?.getInt(STATE_NAME)

}
```

9. How can you clear the backstack of an Activity when a new Activity is called using Intent?

- The first approach is to use a **FLAG_ACTIVITY_CLEAR_TOP** flag.

```
val intent = Intent( this , MainActivity:: class . java )
intent. flags = Intent. FLAG_ACTIVITY_CLEAR_TOP
```

- The second way is by using **FLAG_ACTIVITY_CLEAR_TASK** and **FLAG_ACTIVITY_NEW_TASK** in conjunction.

```
val intent = Intent( this , MainActivity:: class . java )
intent. flags = Intent. FLAG_ACTIVITY_CLEAR_TASK and  Intent.
FLAG_ACTIVITY_NEW_TASK
```

10. What’s difference between FLAG_ACTIVITY_CLEAR_TASK and FLAG_ACTIVITY_CLEAR_TOP?

FLAG_ACTIVITY_CLEAR_TASK is used to clear all the activities from the task including any existing instances of invoked Activity. The Activity launched by intent becomes the new root of the otherwise empty task list. This flag has to be used in conjunction with **FLAG_ACTIVITY_NEW_TASK** .

On the other hand, **FLAG_ACTIVITY_CLEAR_TOP** , if set and if an old instance of this activity exists in the task list then barring that all the other activities are removed and that old activity becomes the root of the task list.

Else if there is no instance of that Activity, then a new instance of it is created and made the root of the task list. Using this flag with **FLAG_ACTIVITY_NEW_TASK** is a good practice most of the time.

11. What is the correct way of starting a new Activity with arguments?

When starting a new Activity with arguments, you have to share keys of arguments between activities. That introduces a dependency between activities. So, as a good development practice ***defining a newIntent() method in every activity*** is recommended to deliver an intent for that Activity.

For instance:

```
// Define a newIntent() method in every Activity

class MainActivity  : AppCompatActivity() {

    companion object {
        val ARG_NAME = "ARG_NAME"

        fun newIntent(context: Context, name: String): Intent {
            val intent = Intent(context, MainActivity:: class . java )
           intent.putExtra( ARG_NAME , name);
            return intent
        }

    }
}
```

✓

Important:

You have to pronounce **“a newIntent() method”** when replying this question. That is what interviewer wants to hear.

How can we retrieve this arguments?

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super .onCreate(savedInstanceState)
       setContentView(R.layout. activity_deneme )

        intent ?. apply {
            val name = getStringExtra( ARG_NAME )
        }

    }

}
```

And How to start this Activity?

```
startActivity(MainActivity.newIntent( this , "My name is Khan!" ))
```

12. How can you explain “Constructive first, destructive last” rule?

“ *Constructive first, destructive last* ” means, in constructive methods you should call **super()** method first, then do whatever you want. Oppositely, in destructive methods, **super()** method should be the last line in the overridden method.

Constructive methods are: **onCreate()** , **onStart()** , **onResume()** .

For instance:

```
override fun onCreate(savedInstanceState: Bundle?) {
   super .onCreate(savedInstanceState)

  // Do whatever you want
   readIntentData()

}

override fun onStart() {
   super .onStart()

   // Do whatever you want
   bindUI()
}

override fun onResume() {
   super .onResume()

   // Do whatever you want
   refrehToken ()
}
```

And destructive methods are : **onPause()** , **onStop()** , **onDestroy()** .

For instance:

```
override fun onPause() {
   videoView.pause()
    super .onPause()
}

override fun onStop() {
   clearResources()
    super .onStop()
}
```

The basic idea behind this rule is, if your activity is going to be destroyed, then you should do whatever you want just before it is destroyed.

CHAPTER 2

FRAGMENT

13. Why do we need Fragments?

Fragments has been introduced for different screen sized devices . Using Fragments we can implement a whole screen or just a little part of it.

When designing our apps for tablets, Fragments gives us flexibility.

14. Can you explain lifecycle callbacks of a Fragment?

Yes, sure. Fragment's lifecycle method are similar to that of an Activity. But it has a few additional callbacks also.

onAttach(): The fragment instance is associated with an Activity but they are not fully initialized yet. We can use this method to retrieve a reference to the Activity.

onCreate(): System calls this method when creating the Fragment. We should initialize essential components of the Fragment that we want to retain when your Fragment is resumed.

onCreateView(): This is where our Fragment draws its UI for the first time. We need to return a View component in this method that is the root of our Fragment. If our Fragment doesn't provide a UI, we may return null.

onActivityCreated(): It is called when the host Activity is created. Activity and Fragment have been created as well as view hierarchy of the of the Activity. *We can now initialize components that requires a Context object.*

onStart(): This method is called once the Fragment gets visible.

onResume(): This method called is called when Fragment is active.

onPause(): User is leaving our Fragment. System calls this method to indicate that. We should now commit any changes and clean system resources if required.

onStop(): Fragment is going to be stopped.

onDestroyView(): Our Fragment's UI is going to be destroyed.

onDestroy(): Called to do a final clean up of the Fragment’s state. But it’s not guaranteed to be called every time by Android OS.

onDetach(): Called immediately prior to the Fragment no longer being associated to the activity.

15. How can two Fragments communicate?

We can provide Interface Callbacks in fragments for this purpose. This interface must be implemented by activities that contains this Fragment to allow an interaction in this Fragment to be communicated to the Activity and other Fragments contained in that Activity.

For instance;

```
class FragmentOne : Fragment() {

        interface OnFragmentInteractionListener {
           fun onFragmentInteraction(uri: Uri)
        }
}
```

This is a Callback Interface defined in FragmentOne. Any Activity using this Fragment must implement this interface. When this interface method is invoked, our Activity is invoked also. That way this Activity can invoke other Fragments. This is how Fragments can communicate.

16. What is difference between an Activity and a Fragment?

An Activity is an application component that provides a screen, with which users can interact in order to do something.

A Fragment represents a behavior or a portion of a screen that Activity provides. Fragments have their own lifecycle.

17. What is the correct way of calling a Fragment with arguments?

We need to use a ***factory method*** create a new instance of a Fragment. This is a best practice that was introduced with ***Effective Java*** book of Joshua Bloch.

✓

Important:

You have to pronounce **“Factory Method”** when replying this question. That is what interviewer wants to hear. Also definitely mention **Effective JAVA** book. That will make you seem as an expert. Because this book is one of the most famous books in software industry and Factory Method is item #1 in this book.

For instance;

```
class FragmentOne : Fragment() {

   companion object {
      /**
      * Use this factory method to create a new instance of
      * this fragment using the provided parameters.
      *
      */
      @JvmStatic
      fun newInstance(param1: String, param2: String) =
            FragmentOne(). apply {
               arguments = Bundle(). apply {
                  putString( ARG_PARAM1 , param1)
```

```
                putString( ARG_PARAM2 , param2)
            }
        }
    }
}
```

18. How can a Fragment communicate to an Activity?

We can provide Interface Callbacks in fragments for this purpose. This interface must be implemented by activities that contains this Fragment to allow an interaction in this Fragment to be communicated to the Activity and other Fragments contained in that Activity.

For instance;

```
class FragmentOne : Fragment() {

        interface OnFragmentInteractionListener {
            fun onFragmentInteraction(uri: Uri)
        }
}
```

19. When replacing one Fragment with another, how can you ensure that user can return to previous Fragment by pressing BACK button?

Going back to previous screen is essential for an Android app. That is what Google recommends. Since activities are added to activity stack, we go back to previous Activity when user presses BACK button unless we use NEW_TASK flag.

But on Fragment we need to save each Fragment transaction to the backstack, by calling ***addToBackStack()*** method before you ***commit( )*** that transaction.

```
// Add the new tab fragment
fragmentManager.beginTransaction()
.replace(R.id.container, TabFragment.newInstance())
.addToBackStack(BACK_STACK_ROOT_TAG)
.commit()
```

Important:

The interviewer wants to see if you know standart UX for Android such as what happens when pressing BACK or HOME button etc. That’s why we have also mentioned about activities a bit.

19. What is difference between FragmentPagerAdapter and FragmentStateAdapter ?

FragmentPagerAdapter: The fragment of each page the user visits will be stored in memory, although the view will be destroyed. So when the page is visible again, the view will be recreated but the fragment instance is not recreated. This can result in a significant amount of memory being used.

FragmentPagerAdapter should be used when we need to store the whole fragment in memory. FragmentPagerAdapter calls *detach(Fragment)* on the transaction instead of *remove(Fragment)* .

FragmentStatePagerAdapter: The fragment instance is destroyed when it is not visible to the user, except the saved state of the fragment. This results in using only a small amount of memory and can be useful for handling larger data sets.

FragmentStatePagerAdapter should be used when we have to use dynamic fragments, like fragments with widgets, as their data could be stored in the savedInstanceState.Also it won’t affect the performance even if there are large number of fragments.

CHAPTER 3

ANDROID UI

20. What is ConstraintLayout?

ConstraintLayout is a ViewGroup which allows you to position and size widgets in a flexible way.

Why should we prefer it RelativeLayout?

Because ConstraintLayout provides much more constraints compared to RelativeLayout. These are dimension constraints, chains, circular positioning etc…

Also using ConstraintLayout may make you save from GPU since it can reduce redraw count of pixels to one.

21. What are different orientations for LinearLayout? Which one is default when no orientation is set?

There are two different orientations for LinearLayout: **Vertical** and **Horizontal** .

When no orientation is defined, Horizontal is default.

22. LinearLayout, RelativeLayout and FrameLayout. Can you explain them briefly and when should we prefer them?

LinearLayout: It's the most commonly used layout in Android. We can group views as vertical or horizontal using this layout.

RelativeLayout: It allows us to locate views related to other views in the RelativeLayout. For instance *"keep this button under this edittext"* is a constraint that requires RelativeLayout.

FrameLayout: It is designed to block out an area on the screen to display a single item.

When should prefer one of them?

If your layout contains only one oview then Framelayout may be a good fit. But if you have a more complicated UI, you should try RelativeLayout. For the rest LinearLayout is good.

Android SDK has a tool called **Layout Inspector** that allows us to analyse our layout when our app is running. I generally use this tool to examine to see that if i have to optimize my UI.

✓

Important:

The interviewer wants to see if you know how to design a complicated UI. You should say that you are aware of Layout Inspector tool that comes with Android SDK. For more info please refer to here .

23. What are ViewGroups and how they differ from View?

ViewGroup is the invisible container. It holds View and ViewGroup. For instance LinearLayout is the viewgroup that contains views and other layouts also. ViewGroup is the base class for Layouts.

But a View is an object which is the basic building block of UI elements in Android. A Button, a Textview or an Edittext… All of them are views. You can check Palette section of Android Studio for a complete set of provided views. We can develop our custom views also.

24. What is difference between View.GONE and View.INVISIBLE ?

When a view in INVISIBLE, then it’s invisible on screen but still it keeps occupying some space on layout.

But if a view is GONE, then it’s invisible and it doesn’t occupy any space on layout.

25. How can you create a custom view?

Actually a View is just a base class for other views. So we start creating a class that extends View class.

Then we should override some of the methods from superclass. Some of the superclass methods to override are *onDraw()* , *onMeasure()* , *onLayout()* etc...

Now we are ready to use out new View. Our view can be used in place of the view upon which it was based.

26. What should we do to support different screen sizes?

Android devices comes in all shapes and sizes, so we need to design flexible screens. To achieve this we may do such things:

- Use view dimensions that allow layout to resize instead of hardcoded dimensions.
- Benefit from ConstraintLayout
- Create alternative layouts for different screen sizes. Android allows us to define alternative layout for different screens. So we can provide different XML files for tablets and phones.

For instance we can define same XML with three different versions.

```
res/layout/main_activity.xml           # For handsets (smaller than 600dp available width)
res/layout- sw600dp /main_activity.xml   # For 7" tablets (600dp wide and bigger)

res/layout- sw720dp /main_activity.xml   # For 10" tablets (720dp wide and bigger)
```

27. How do you understand your app is running on tablet or phone?

The best practice for this is to have a field in *bools.xml* file.

We can put XML (bools.xml) files in the following folders

- values-large
- values-xlarge
- values-normal
- v alues-small

For *values-large/bools.xml* and *values-xlarge/bools.xml*

```
<bool name="tablet">true</bool>
```

For *values-normal/bools.xml* and *values-small/bools.xml*

```
<bool name="tablet">false</bool>
```

Then to determine programmatically,

```
boolean isTablet = context.getResources().getBoolean(R.bool.tablet);
```

28. How can we make our apps to look better in tablet devices?

We need to do a bunch of things. First of all we have to make our screens responsive. For this purpose we should design different versions of our XML files and drawables and other assets.

So for standard configuration qualifiers for screen size you can create following configuration:

- *layout-sw480dp drawable-sw480dp*
- *layout-sw600dp drawable-sw600dp*
- *layout-sw720dp drawable-sw720dp*

Then we have to test our app for basic tablet app quality. We should never forget to run it on different tablet emulators.

Tablets have large screens, we should use this extra space as an advantage. That requires a special UI design for tablets.

Phones and tablets typically offer slightly different hardware support for sensors, telephony and camera. So we need to pay attention that our APK does not require any hardware features that are not commonly available on tablets.

```
<uses-feature   android:name = "android.hardware.telephony"
android:required = "false"   />
```

Important:

Designing for tablets is a very important topic in daily life of an Android developer. Every Android developer is required to know key topics of this since apps generally deliver a single APK for phones and tablets. Answer given above is sufficient most of the time but since this is a general topic, you can gather more info from here .

29. Why do we need “Support Library” and “AppCompat Library” ?

Android Support Library provides newer API’s for older releases. That’s why we need it.
Android Support Library is not a single library, but rather a collection of libraries that can roughly be divided into two groups: **Compatibility** and **Component Libraries** . This Compatibility Library can be called as AppCompat Library.

<u>To sum up, Support Library provides newer API’s for older releases and it contains AppCompat Library.</u>

What do you mean by Compatibility and Component Libraries?

Good question! **Compatibility libraries** focus on back porting features newer framework releases so that devices running older versions can take advantage of newer API’s. The major Compatibility libraries are **v4-appcompat** and **v7-appcompat** .

V4-appcompat is well-known for DrawerLayout and ViewPager while v7-appcompat brings ToolBar and ActionBar.

On the other hand **Component libraries** are smaller and more modular libraries that enable developers to add features that are otherwise not part of the standard Framework. Some of this libraries are for *RecyclerView* , *Palette* , *CardView* , *GridLayout* and *MediaRouter* .

Important:

Most of the time Android developers don’t know the difference between Support Library and AppCompat Libraries. They just add Gradle dependency and use it. That is not a good situation for an experienced developer. The interviewer wants to see that you are aware of the basic functionalities of Android since Support Library is the most widely used library in Android world.

You should also be ready to second question i have given. They might require you to name a few components that are provided by these libraries. Just memorize the text i above and repeat it. That will be possibly enough.

30. Why should i use the Android Support Library? Which version of it should i use?

We should use one of the Support Libraries when we need specific framework features that are newer than minSdkVersion of our app that are not available in the standard framework. For instance if we are targeting anything less than API 21 and wish to use Material design UI features then we are forced to use Support Libraries.

Which version of Support Library should i use?

In short, we should use the version our application can support that has the features we need. Some features are modular. For example, to use RecyclerView, simply add the v7-recyclerview dependency identifier to our gradle build script.

If we need one of the compatibility components from v4, we can use v13 instead if our minSdkVersion supports that, since it bundles v4. Otherwise, if we need to support API versions <13 and need a v4 component, we use the v4 support library itself.

CHAPTER 4

CORE JAVA

31. Can you compare Array with LinkedList? Which is better?

An **Array** consists of a group of elements of the same data type. It is stored contiguously in memory and by using its' index we can find the underlying data. Arrays can be one dimensional and multi-dimensional.

A **LinkedList** , just like a tree and unlike an array, consists of a group of nodes which together represent a sequence. Each node contains data and a pointer. The data in a node can be anything, but the pointer is a reference to the next item in the LinkedList. A LinkedList contains both a head and a tail. The "Head" is the first item in the LinkedList, while the "Tail" is the last item.

The runtime complexity for these data structures are as follows,

Algorithm	**Array**	**LinkedList**
Access	O(1)	O(n)
Search	O(n)	O(n)
Insert	O(n)	O(1)
Delete	O(n)	O(1)

Important:

Most of the time they require you to know Big-O values for CRUD operations. Try to memorize them and write them down on a paper while you are being interviewed. That will possibly impress the interviewer.

32. What is difference between Abstract Class and Interface?

Abstract class is a class that can contain both concrete and abstract methods. Abstract method means a method without implementations.

An Interface is a contract that can be implemented by classes. All the methods in an Interface are abstract. They can’t have concrete methods.

- Abstract classes are extended while Interfaces are implemented.
- Abstract classes may have constructors but Interfaces can’t have.
- A class can extend only one class (abstract or not) but can implement as many interfaces as it wants.

Important:

This is one of the oldest and most common interview questions in Java world. If you are a new graduate or a junior developer, be ready for this one.

33. What are Local, Instance and Static variables?

Local variables are the ones declared in methods. Their scope is restricted with that method. They can’t be reached from other methods.

Instance variables are the non-static ones declared in class. They can be reached in every method of the class itself. Why they are called as “Instance variable” is that you need to instantiate an object of the class itself to reach this variable. (In case it’s not private.)

Static variables are the ones that are declared in class with static keyword. You don’t have to instantiate an object of that class to reach that variable. Static variables are stored in Stack.

```
class  MainClass{

        public  int instanceVariable = 0;
        public  static int staticVariable = 5;

        public   void  aMethod(){
                String localVariable = “My name is Khan”;
        }

}
```

34. What is “Call-by-value” and “Call-by-reference” ? Which one is used in JAVA?

<u>JAVA uses call-by-value only.</u> JVM manipulates objects according to their references but passes parameters to methods as **Call-by-Value** .

Call-by-value means, when passing parameters to methods we send their value not the reference of object. Oppositely Call-by-reference means, sending reference of objects to methods.

Let me explain in more details. In Java, method parameters can be primitive types - like *int* , *float* , *double* etc…- or object references. Both are passed as value only but small tricky explanation is here for object references.

When primitive data types are passed as method parameters, they are passed by value (a copy of the value) but in case of object references, the reference is copied and passed to the called method. That is, object reference is passed as a value. So, original reference and the parameter copy both will refer the same Java object.

As both refer the same object, if the calling method changes the value of its reference -passed from calling method-, the original object value itself changes. Note that object itself is not passed, but it’s references is passed.

✓

Important:

This is one of the core topics of Java language. If you don’t sound confident when replying this question, then it is going to be a big minus for your interview.

35. What is “Mutable” and “Immutable” objects?

Immutable objects are simply objects whose state can’t be changed after construction. I mean object’s data by state. Examples of Immutable objects from the JDK includes String and Integer.

Oppositely mutable objects are the ones that their state can be changed after construction.

For Instance;

```
String myString = new String( "old String" );
System. out .println( myString );
myString.replaceAll( "old" , "new" );
System. out .println( myString );

// Result

old String
old String
```

Since String is immutable, we can’t change its value and as you can see, output of code given above is same: “ **old String** ”. Because String is immutable.

But if we assign result of `myString.replaceAll( "old" , "new" )` method to a new String object, then output changes. But then we have a new String object. The old one is still has same value.

```
myString = myString.replaceAll( "old" , "new" );
System. out .println( myString );

// Result
```

```
new String
```

36. Can you explain difference between objects and primitive types?

A **primitive data type** uses a small amount of data to represent a single item of data. There are eight primitive data types in Java: *byte* , *short* , *int* , *long* , *float* , *double* , *char* and *boolean* . Primitives are stored in **Stack** .

All data of the same primitive types are the same size. For instance, all variables of int type uses 32 bits.

Here are size of primitive types:

byte	short	char	int	float	double	long	boolean
1 byte	2 bytes	2 bytes	4 bytes	4 bytes	8 bytes	8 bytes	Not precisely defined

An **object** is a large chunk of memory that can potentially contain a great deal of data and methods. Every object derives from *Object.java* class . Objects are stored in **Heap** .

They don't have a defined size. It can be large or small.

✓

Important:

I have indicated that primitive types are stored in **Stack** while objects are in **Heap** . It may sound as a detail but experienced Java guys sees this as an essential topic.

Especially if your interviewer is older than 40 years, he will probably pay attention to that. You should definitely mention from **Stack** and **Heap** .

37. What is difference between “Overloading” and “Overriding” ?

Overloading is using same method name with different parameter signature. Key point here is that return type doesn’t matter. It may be same or different. No matter!

And overloading may occur in same class or subclass.

On the other hand, **Overriding** is defining a method in a subclass that is already defined in parent class with same name, same return type and same parameter signature.

Overriding can’t be done in the same class.

```
// OVERLOADING: Here we have two different methods with same name
but different parameter signature. Both methods are in same class

fun aMethod(param1: String, param2: String){

}

fun aMethod(param1: String, param2: String, param3: String){

}

// OVERRIDING: Here we are overriding onStart() method of Activity class.

override fun onStart() {
    super .onStart()
}
```

38. Why should inner classes be static? What is disadvantage of non-static inner classes?

In Java, inner classes have a direct reference to the outer class. That prevents Garbage Collector from collecting some objects in some cases. That is the disadvantage.

For instance, if we have an AsyncTask as an inner class in our Activity, Android Studio warns us : “This AsyncTask class should be static or leaks might occur.”.

We should avoid to have inner classes unless they are static. That is a good software engineering practice.

✓

Important:

This question brings Core Java and practical Android experience together. You should give this answer. It’s short, complete and to-the-point. How lucky of you!

39. What is difference between Serializable and Parcelable ?

First of all **Serializable** is a Java specific interface. It is used to represent an object as a sequence of bytes. It is implemented by JVM and according to Google, it leaves a lot of garbage behind and performance is not good.

Hence, Android has provided **Parcelable** interface as a alternative. Parcelable provides a new mechanism. Parcelable is for classes whose instances can be written to and restored from a *parcel* .

Classes implementing the Parcelable interface must also have a non-null static field called *CREATOR* of a type that implements the *Parcelable.Creator* interface.

A simple Parcelable implementation is;

```
public class MyParcelable implements Parcelable {
    private int mData;

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(mData);
    }

    public static final Parcelable.Creator<MyParcelable>
CREATOR
            = new Parcelable.Creator<MyParcelable>() {
        public  MyParcelable createFromParcel(Parcel in) {
            return new MyParcelable(in);
        }

        public  MyParcelable[] newArray(int size) {
            return new MyParcelable[size];
        }
    };

    private  MyParcelable(Parcel in) {
        mData = in.readInt();
    }
}
```

40. What is benefit of SparseArray ?

SparseArray is a Android specific data structure that maps integers to objects and, unlike a regular array of objects, its indices can contain gaps.

SparseArray is intended to be more **memory-efficient** than a *HashMap* , because it avoids auto-boxing keys and its data structure doesn't rely on extra an extra entry object for each mapping.

An example of SparseArray is;

```
val sparseArray= SparseArray<String>()
sparseArray.append( 10 , "The First" )
sparseArray.append( 15 , "The Second" )

val item = sparseArray.get( 10 )
```

Benefits of SparseArray:

- Allocation-free
- No boxing
- Memory efficient

Disadvantages:

- Generally slower, not indicated for large collections
- It doesn't work in a non-Android project.

CHAPTER 5

KOTLIN

41. How many constructors are available in Kotlin?

In Kotlin we are allowed to have two different kinds of constructors: **Primary Constructor** and **Secondary Constructor** . We also have **init{}** block which is executed right after primary constructor.

Primary Constructor: A Kotlin class may have only one Primary Constructor. It's part of class header definition, it goes after the class name.

Primary Constructor is a good way for declaring global variables and initializing them.

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

   init {
      print ( "First name is $ firstName " )
      age = 25
   }
}
```

Secondary Constructor: Kotlin also enables us defining new constructors using `constructor` keyword.

```
class Person( val firstName : String) {

   init {
      print ( "First name is $ firstName " )
  }

   constructor (firstName: String, lastName: String) : this (firstName) {}
   constructor (firstName: String, lastName: String, age: Int) : this (firstName)
{}
}
```

A class may have multiple secondary constructors and if it also has a primary constructor, every secondary constructor has to delegate to the primary constructor.

41. What is purpose if init{} block in Kotlin? Can you explain when it is called?

We can't write code in primary constructors in Kotlin, so we have **init{}** blocks. Kotlin has a concise syntax for defining properties with primary constructor and **init{}** block is executed right after the primary constructor.

```
class Person( val firstName : String) {

   init {
      print ( "First name is $ firstName " )
  }

   init {
      print ( "This is the second init{} block." )
  }

   constructor (firstName: String, lastName: String) : this (firstName) {
      print ( "This is a secondary constructor." )
  }

}
```

When this code is executed, execution order is:

- Primary Constructor
- The first init{} block
- The second init{} block
- Secondary Constructor

This order is important and always same. We may have multiple init{} blocks and they are executed in the same order as they appeared in the class body.

Important:

Kotlin is the new language of Android. You need to know basics of this language. Constructors and init{} blocks are very fundamental topics of Kotlin.

I have given the order of execution, please pay attention to that, it is important. Companies assumes a new developer with extensive Kotlin ability may be a potential investment for future.

42. What is “Type Inference”? Why it’s so useful?

Despite Kotlin is a strongly typed language, we don’t always need to define types explicitly. The compiler attempts to figure out the type of an expression from the information included in the expression. This mechanism is called **type inference** .

For instance;

```
var x = 10 // Compiler knows that x is Integer
var name = “Uncle Bob” // Compiler knows that name is a String
```

43. What is Extension Method (Extension Function)?

Extension Method is a handy way of extending existing classes with new functionality without using inheritance or any forms of Decorator pattern.

This is done via special declarations called ***extensions*** . Kotlin supports **Extension Functions** and **Extension Properties** .

Here we define an extension function and add *swap* functionality to **MutableList** class:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
   val tmp = this [index1] // 'this' corresponds to the list
   this [index1] = this [index2]
   this [index2] = tmp
}
```

Now we can use swap method like it is defined in MutableList interface.

```
val l = mutableListOf ( 1 , 2 , 3 )
l.swap( 0 , 2 ) // 'this' inside 'swap()' will hold the value of 'l'
```

44. What is “Null Safety” ? How does Kotlin provides this?

Kotlin’s type system is aimed to eliminate *NullPointerExceptions* . For a variable to be null, it must be *Nullable* , otherwise no variable is allowed to be null. That is how Kotlin provides **Null Safety** .

For instance, below given code is a compilation error:

```
var a : String = "abc"
a = null // compilation error
```

But if we really want it to be null, then we have to modify code.

```
var a : String? = "abc"
a = null // OK.
```

Did you notice the difference? There is a question mark (?) in second code. String? means a nullable String. It may have a value of null.

But then when calling a nullable operator we need to be careful. We have to use **?** or **!!** operators with it. **?** means “if it’s null, the don’t execute it”. **!!** means “I accept NullPointerException if it’s null, execute it.”

```
a ?. length   // If a is null, it doesn’t execute
a !!. length // It executes, no matter a’s value
```

Important:

Null safety is an important concept for modern languages like Kotlin. You need to be intrigidently aware of this concept. Please refer to here for more detailed info.

45. Can you name a few important Kotlin features that Java doesn’t have?

Kotlin provides a bunch of features that Java doesn’t have. For instance:

- **Null safety:** A safe way of getting rid of NullPointerExceptions.
- **Extension functions:** A new way of adding new functionality to existing classes without inheritance.
- **Primary Constructors:** A more practical syntax
- **Type inference for variable and property types**
- **Range expressions**
- **Operator overloading**
- **Data classes**
- **Companion objects**
- **Separate interfaces for read-only and mutable collections**

Important:

The Interviewer wants to see if you are eager to use Kotlin. You should show them you have reasons to chose Kotlin over Java. These items are pretty good enough for you.

46. How can we extend a class and override a method in Kotlin?

By default all classes in Kotlin are **final** . So we have to use `open` keyword for making classes open to inheritance.

For instance;

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

As you can see, **Base** class is defined with `open` keyword. Otherwise it can't be extended.

For overriding methods , we need to declare methods with `open` keyword as well. For instance;

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}

class Derived () : Base() {
    override fun v() { ... }
}
```

Here we have `open fun v()` and this indicated that it's open to be overridden. The other function `fun nv()` can't be overridden. That's how we override methods and extend classes in Kotlin.

Another important detail is that if a class is not open, then we can't define open functions in that class. And a overridden method is also open to be overridden. If we want it to be final, then we need to declare it with `final` keyword.

Basic idea behind this is , **Effective Java** book Item 19 says that: “ *Design and document for inheritance or else prohibit it.* ” We in Kotlin we stick to making things explicit.

Important:

Please pay extra attention to last paragraph. Effective Java book is a very famous one and referencing to one of its items will impact your interviewer. Please keep in mind that such references will make your interviewer think that you are enthusiastic for good software engineering and endeavour to be a good developer.

47. What is “Interface Delegation” in Kotlin?

Delegation is like inheritance done manually through object composition. The Delegation pattern has proven to be a good alternative to implementation inheritance, and Kotlin supports it natively requiring zero boilerplate code. A class Derived can implement an interface Base by delegating all of its public members to a specified object:

```
interface Base {
    fun print()
}

class BaseImpl( val x : Int) : Base {
    override fun print() { print ( x ) }
}

class Derived(b: Base) : Base by b

fun main(args: Array<String>) {
    val b = BaseImpl( 10 )
   Derived(b).print()
}
```

The ***by*** clause in the supertype list for **Derived** class indicates that ***b*** will be stored internally in objects of **Derived** class and the compiler will generate all the methods of **Base** that forward to ***b*** .

48. What is “var” and “val” in Kotlin?

We use `var` and `val` keywords to define new variables is Kotlin. The difference is that `val` is for immutable objects while `var` is for mutable ones.
When you defined an object with val, then you can’t neither change it nor assign a new value to it. For instance;

```
class Person( val firstName : String) {

    var a : String? = "abc"
    var mutable = Person( "Tom Cruise" )
    val imMutable = Person( "Daniel Craig" )

    fun test(){
        mutable . a = "Mission Impossible"   // OK
        imMutable = Person( "Uncle Bob" )   // ERROR
        mutable .firstName = "Kent Beck"   // ERROR
    }

}
```

49. What is “Companion Object” ? Which Android best-practice can be implemented benefiting from “Companion Objects” ?

If we want to create static members in Kotlin, we use **companion object** . Members inside **companion objects** can be called like static members. For instance;

```
class My Activity : AppCompatActivity() {

    companion object {
        val TAG = "MyActivity"
    }

}
```

Actually at runtime, even though the members of **companion objects** look like *static* members in other languages, at runtime those are still instance members of **real objects** , and can, for example, implement interfaces.

However, on the JVM you can have members of companion objects generated as real static methods and fields, if you use the **@JvmStatic** annotation.

Which Android best-practice can be implemented benefiting from Companion Objects?

Actually, we can use companion objects for writing a **newIntent()** method and defining a **TAG** string for logging. Let me show you an example:

```
companion object {
   val TAG = MyActivity:: class . java . simpleName
   val ARG_NAME = "ARG_NAME"

   fun newIntent(context: Context, name: String): Intent {
      val intent = Intent(context, MyActivity:: class . java )
     intent.putExtra( ARG_NAME , name);
      return intent
  }

}
```

50. What are the visibility modifiers in Kotlin? Which one is default?

There are four visibility modifiers in Kotlin: **private** , **protected** , **internal** , **public** .

public : means that your declarations will be visible everywhere. If you don't specify any visibility modifier, **public** is used by default .

private : means visible inside this class only, including all its members.

internal : If you mark it internal, it is visible everywhere is the same module.

protected : same as **private** , plus, visible in *subclasses* too.

```kotlin
open class Outer {
    private val a = 1
    protected open val b = 2
    internal val c = 3
    val d = 4   // public by default

    protected class Nested {
        public val e : Int = 5
    }
}
```

51. What is “lateinit” keyword in Kotlin?

We use **lateinit** for initializing field later. Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient.

For example, properties can be initialized through **Dependency Injection** , or in the *setup()* method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class.

To handle this case, you can mark the property with the **lateinit** modifier:

```
class Main Activity : AppCompatActivity() {

    lateinit var textView : TextView
    override fun onCreate(savedInstanceState: Bundle?) {
        super .onCreate(savedInstanceState)
       setContentView(R.layout. activity_main )

        textView = findViewById(R.id. text_view )

   }
}
```

The most common case to use **lateinit** is finding views using `findViewById()` method.

52. What is “lazy” keyword in Kotlin?

lazy is a function to delegate the implementation of the property. For instance;

```
val toolbar: Toolbar by lazy {
   findViewById(R.id.toolbar) as Toolbar
}
```

The **lazy** block won’t execute until the *toolbar* variable is called for the first time. Now you can initialize at the property level before the context is ready.

```
// Now the lazy block is executed and stored
toolbar.setTitle( "I love Kotlin!" )
```

A common scenario to use **lazy** is finding views using `findViewById()` method.

CHAPTER 6

INTENT AND BROADCAST

53. What is Implicit Intent and Explicit Intent?

Explicit Intent is just a normal *Intent* that we use to start an *Activity* or start a *Service* or deliver a *Broadcast* . We will typically use it to start a component in our app, because we know the class name of the Activity or Service .

```
// An Explicit Intent
val intent = Intent( this , SecondActivity:: class . java )
startActivity(intent)
```

Implicit Intent does not require a component name but instead declare a general action to perform. That allows a component from another app to handle it.

```
// An Implicit Intent
val sendIntent = Intent()
sendIntent. action = Intent. ACTION_SEND
sendIntent.putExtra(Intent. EXTRA_TEXT , textMessage)
sendIntent. type = "text/plain"
```

The main difference is that , Explicit Intent requires name of component to launch but Implicit Intent requires an action to perform. Explicit Intents are used to start a component in our app but Implicit Intents are used to launch another app's component.

Important:

Difference between these Intents are important. Since we need Intents to do lots of things, the interviewer may require you know the difference between implicit and explicit intents.

54. What is wrong with the given code?

```
// Create the text message with a string
val sendIntent = Intent()
sendIntent. action = Intent. ACTION_SEND
sendIntent.putExtra(Intent. EXTRA_TEXT , textMessage)
sendIntent. type = "text/plain"

startActivity(sendIntent)
```

OK, in this code we see that developer wants to share a text message and he calls an implicit intent for this purpose. But the problem here is we don't know if there exists any activity to resolve this intent.

<u>So, we should always check resolver activities before starting a component with implicit intent.</u> We should change last line as follows:

```
// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null ) {
  startActivity(sendIntent);
}
```

OK, then what happens when we run this code?

When we run this code, there are three scenarios.

- There is no resolver Activity, so nothing happens thanks to if block we have added.
- There is only one resolver Activity, then Android OS automatically starts that Activity.
- There are more than one resolver Activity. Android OS pops up a list of resolver activities and user selects one of them.

Important:
The interviewer wants to test that are you aware of common mistakes in Android and can make defensive programming against runtime exceptions.

The answer is a bit long but if you try to follow codes and speak about it, this texts comes out of your mouth as a script.

Don't memorize the text, just try to explain code, you will see that your brain is going to produce same words while explaining.

55. What is a Sticky Intent?

A **Sticky Intent** is an intent that is used with *sticky broadcast* , is called as sticky intent. This intent will stick with android system for future broadcast receiver requests.

For example if **BATTERY_LOW** event occurs then that Intent will stick with Android so that any future requests for **BATTERY_LOW** , will return the Intent.

56. What is a Pending Intent?

Pending Intent is actually an object which wraps an Intent to do some future work by another app.

If we want some component to perform any Intent operation at future point of time on behalf of us, then we will use Pending Intent.

```
// Creating a pending intent and wrapping our intent
val pendingIntent = PendingIntent.getActivity( this , 1 , intent, PendingIntent.
FLAG_UPDATE_CURRENT )

try {
    // Perform the operation associated with our pendingIntent
    pendingIntent.send()
} catch (e: PendingIntent.CanceledException) {
   e.printStackTrace()
}
```

57. What does Android Oreo recommends about dynamic and static BroadcastReceivers?

Broadcast is simply a message between apps and we use **BroadcastReceivers** to catch these messages.

If we declare our BroadcastReceiver in our *AndroidManifest.xml* file then it is a **Static BroadcastReceiver** . But if we define it in code, using `Context.registerReceiver()` method, then it is a **Dynamic BroadcastReceiver** .

When we are doing *dynamic registration* (i.e. at run time) it will be associated with lifecycle of the app. If we do it *static registration* (i.e. on compile time) and our app is not running, a new process will be created to handle the broadcast.

Android Oreo no longer supports Static BroadcastReceivers for battery life and security issues.

Here we have a sample of Dynamic BroadcastReceiver. Please notice that we unregister it in **onDestroy()** method. Static BroadcastReceivers don't allow us to do that.

```
class MainActivity : Activity() {

   internal lateinit var filter1 : IntentFilter

   override fun onCreate(savedInstanceState: Bundle?) {
      super .onCreate(savedInstanceState)
      filter1 = IntentFilter(
"android.bluetooth.BluetoothDevice.ACTION_ACL_CONNECTED" )
     registerReceiver( myReceiver , filter1 )
  }

   public override fun onDestroy() {
        unregisterReceiver( myReceiver )
         super .onDestroy()
    }
}
```

58. How can we catch only a specific kind of Broadcast message?

We use **IntentFilter** for this. **IntentFilter** is a class provided by Android to catch only a specific kind of **Broadcast** messages.

Here we have an example of typical **BroadcastReceiver** and **IntentFilter** used with it.
This code is listening to Bluetooth connection events. Its IntentFiler is filtering incoming broadcast messages and comparint its action with “ **ACTION_ACL_CONNECTED** ”.

That’s how a Broadcast message is handled through an IntentFilter.

```
class MainActivity : Activity() {

   internal lateinit var filter1 : IntentFilter

   //The BroadcastReceiver that listens for bluetooth broadcasts
   private val myReceiver = object : BroadcastReceiver() {
      override fun onReceive(context: Context, intent: Intent) {
         if (intent. action !!. equals (
"android.bluetooth.BluetoothDevice.ACTION_ACL_CONNECTED" ,
ignoreCase = true )) {
            Log.d( TAG , "Bluetooth connect" )
         }
      }
  }

   override fun onCreate(savedInstanceState: Bundle?) {
      super .onCreate(savedInstanceState)
      filter1 = IntentFilter(
"android.bluetooth.BluetoothDevice.ACTION_ACL_CONNECTED" )
     registerReceiver( myReceiver , filter1 )
  }
```

```
    public override fun onDestroy() {
      unregisterReceiver( myReceiver )
       super .onDestroy()
   }
}
```

CHAPTER 7

SERVICE

59. What is Service and Intent Service? What is difference?

A **Service** is an application component that can perform long-running operations in the background, and it doesn't provide a user interface.

Another application component can start a service, and it continues to run in the background even if the user switches to another application.

<u>One important detail is that Services run on main thread.</u>

Intent Service is a special kind of Service that <u>runs on a background thread</u> . That is the difference between Service and Intent Service.

60. Can you explain Android Bound Services?

A bound service is a service that allows other android components (like activity) to bind to it and send and receive data. A bound service is a service that can be used not only by components running in the same process as local service, but activities and services, running in different processes, can bind to it and send and receive data.

When implementing a bound service we have to extend Service class but we have to override *onBind()* method too. This method returns an object that implements **IBinder** , that can be used to interact with the service.

A typical Service example would be;

```
class LocalService : Service() {
    // Binder given to clients
    private val mBinder = LocalBinder()

    /**
    * Class used for the client Binder.
      Because we know this service always runs in the same process as   its
clients, we don't need to deal with IPC.
    */
    inner class LocalBinder : Binder() {
        internal // Return this instance of LocalService so clients can call public
methods
        val service : LocalService
            get () = this @LocalService
    }

    override fun onBind(intent: Intent): IBinder {
        return mBinder
    }
}
```

As we can see it has a onBind() method. Clients have to use this method in order to bind to Service.

Now let’s look at an Activity that wants to bind this service.

```
class BindingActivity : Activity() {
   internal lateinit var mService : LocalService
   internal var mBound = false

  override fun onCreate(savedInstanceState: Bundle?) {
      super .onCreate(savedInstanceState)
     setContentView(R.layout.main)
  }

   override fun onStart() {
      super .onStart()
      // Bind to LocalService
      val intent = Intent( this , LocalService:: class . java )
     bindService(intent, mConnection , Context. BIND_AUTO_CREATE )
  }

   override fun onStop() {
     unbindService( mConnection )
      mBound = false
     super .onStop()

   }


   /** Defines callbacks for service binding, passed to bindService()  */
   private val mConnection = object : ServiceConnection {

      override fun onServiceConnected(className: ComponentName,
                                      service: IBinder) {
         // We've bound to LocalService, cast the IBinder and get LocalService
instance
         val binder = service as LocalBinder
         mService = binder.getService()
         mBound = true
```

```
        }

        override fun onServiceDisconnected(arg0: ComponentName) {
            mBound = false
        }
    }

}
```

Let’s explain code line by line. At first we create an instance of **Service** . Since it isn’t initialized yet we have defined it with `lateinit` keyword. We also have a boolean value to keep binding status.

Then in `onStart()` method we create an Intent and call `bindService()` to bind the service. We also provide a `mConnection` object, but let me come to that later.

As a good development practice we unbind service calling `unbindService()` method. Then we update value of `mBound` to false. We always pay attention to close resources in destructive methods.

Finally we have a connection object of `ServiceConnection` . This is a special class of Android. It has two callback methods that are triggered when connection status of service changes.

`onServiceConnected()` is called when the connection is established and it brings an **IBinder** object with itself. **IBinder** is important because we get our Service instance from that **IBinder** .

Respectively `onServiceDisconnected()` method is triggered once the connection is lost. Here we have nothing to do more than updating our `mBound` object to false.

That is roughly how a bound service is working. It’s simple and elegant.

61. How would you update UI of an Activity from a Service?

Actually there are a few ways of achieving this.

- The first solution comes to my mind is that, i bound to the service in activity. Since it's a bound service, i get a direct reference of Service and so that i can assign a listener object to the service. That is called Observer pattern of Hollywood Principle. If the service wants to update activity's UI then all it has to is just to trigger listener callback methods.

- Second method is also yet another common way of messaging in Android: Broadcasts. Yes, i would definitely send a broadcast in service when i want to update UI of activity. Of course we should add extra info to broadcast's bundle for UI. Then in the activity, i would catch that broadcast and update UI.

- One another way is using a third party library: EventBus. EventBus is a very simple and efficient way of messaging between components. Service can send messages to activity using EventBus.

- One last way of this is maybe we can use RxJava as an addition to the first approach. The only difference is we observe a field of service using Rx instead of setting a listener.

If you still insist on me to chose one of them, my choice would be the first one since it doesn't require any third party libraries. But still that may vary from project to project. Sometimes coding conventions or the principles of the team may require different approaches.

✓

Important:

Be ready to that kind of questions. In software engineering industry there isn’t an absolute way of getting things done. Most of the times there are tones of different ways of doing something. The interviewer wants to see that if you are open to look for alternative solutions to some general problems.

Try to mention every solution that comes into your mind and don’t hesitate to mention pros and cons for every idea. This is how you are going to have a good impression.

62. How does Android Oreo restricts Services? What are our alternatives to use service functionality?

Android Oreo (API 26) imposes limitations on what can apps do while running in the background.

When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services. At the end of that window, the app is considered to be idle. At this time, the system stops the app's background services, just as if the app had called the services' *Service.stopSelf()* methods.

While an app is in the foreground, it can create and run both foreground and background services freely.

Possible solutions:

- First of all we may migrate some services to JobIntentServices.

- Secondly, while the app is in the background, we should use the *startForegroundService()* method instead of *startService()* .

- If the service is noticeable by the user, we should make it a foreground service. For example, a service that plays audio should always be a foreground service. We should create the service using the *startForegroundService()* method instead of *startService()* .

- Firebase Cloud Messaging (FCM) is a good alternative to wake our application up when network events occur, rather than polling in the background.

✓

Important:

The interviewer wants to see if you are following latest releases of Android versions. Every year, Google announces different restrictions or functionalities at Google I/O event. As an Android developer you are supposed to follow these new staff and be comfortable with it.

I have provided a detailed answer to the question. So you don’t have to worry about it. Just read the answer a few times, it is gonna cover you.

63. What is a JobScheduler? Why do we need it?

JobScheduler is Android API to scheduling tasks or work.

The framework will be intelligent about when it executes jobs, and attempt to batch and defer them as much as possible. Typically if we don't specify a deadline on a job, it can be run at any moment depending on the current state of the JobScheduler's internal queue.

Advantage of JobScheduler is that while a job is running, the system holds a wakelock on behalf of our app. For this reason, we do not need to take any action to guarantee that the device stays awake for the duration of the job.

We do not instantiate this class directly; instead, retrieve it through `Context.getSystemService(Context.JOB_SCHEDULER_SERVICE)` .

For instance;

```
val jobScheduler = getSystemService(Context. JOB_SCHEDULER_SERVICE )
as JobScheduler
jobScheduler.schedule(JobInfo.Builder(LOAD_ARTWORK_JOB_ID,
      ComponentName( this , DownloadArtworkJobService:: class . java !!))
      .setRequiredNetworkType(JobInfo. NETWORK_TYPE_ANY )
      .build())
```

PART 2

SENIOR LEVEL QUESTIONS

CHAPTER 8

BACKGROUND JOBS

64. How can we perform long-running operations in background?

Android provides a few ways of this depending our needs. To name a few of them: **ThreadPool** , **Foreground Services** and **WorkManager** .

Let's explain them one by one.

ThreadPools:
For work that should only be done when our app is in the foreground, we use **ThreadPools** . **ThreadPools** provide a group of background threads that accept and enqueue submitted work. If we need to monitor system triggers during this time, we use dynamically-registered broadcast receivers to monitor OS state and triggers.

Foreground Services:
For work that must execute to completion, if we need the work to execute immediately, we use a **foreground service** . Using a foreground service tells the system that the app is doing something important and they shouldn't be killed. Foreground services are visible to users via a non-dismissible notification in the notification tray.

WorkManager:
For work that must execute to completion and is deferrable, we use **WorkManager** . **WorkManager** is an Android library that gracefully runs deferrable background work when the work's triggers (like appropriate network state and battery conditions) are satisfied. **WorkManager** uses the framework **JobScheduler** whenever possible, to help optimize battery life and batch jobs. On devices below Android 6.0 (API level 23), WorkManager attempts to use *Firebase JobDispatcher* if it's already an included dependency of our app; otherwise, WorkManager falls back to a custom

AlarmManager implementation to gracefully handle our background work.

65. What happens if Activity dies while AsyncTask is running?

Lifecycle of an Activity and that of an AsyncTask are separated . That means if Activity dies while AsyncTask is still running, the AsyncTask won't stop. It will keep working. But in `onPostExecute()` method if **AsyncTask** tries to update UI, we get into trouble. Since Activity is already dead, system throws an exception:

```
java.lang. IllegalArgumentException : View not attached to window manager
```

To get rid of this, we should always check if Activity is alive before updating UI in `onPostExecute()` method of **AsyncTask** .

```
protected fun onPostExecute(file_url: String) {
   if ( this @YourActivity .isDestroyed()) {
     // or call isFinishing() if min sdk version < 17
      return
   }

  dismissProgressDialog()
  something(note)
}
```

66. How can we benefit from Handlers?

Handlers are objects for managing threads. It receives messages and writes code on how to handle the message. They run outside of the activity's lifecycle, so they need to be cleaned up properly or else you will have thread leaks.

```
Handler(Looper.getMainLooper()).post( Runnable {
    kotlin. run {
        //This runs in the main thread
    }
} )
```

Handlers allow communicating between the background thread and the main thread.

```
val handler = object : Handler() {
    override fun handleMessage(msg: Message) {
        // process incoming messages here
       // this will run in the thread, which instantiates it
    }
}

// Let's send a message to Handler
val msg = Message()
msg. obj = "Ali send message"
handler.sendMessage(msg)
```

A Handler class is preferred when we need to perform a background task repeatedly after every x seconds/minutes or to be executed after a delay..

```
Handler().postDelayed(
      Runnable { kotlin. run {
         Log.d( TAG , "This will run after 2 seconds." ) }
      } ,
      2000 )
```

67. What is difference between Service, Intent Service, AsyncTask & Threads?

First of all these are the classes that we use to perform tasks on background. In real world applications there are some really complicated scenarios that we need to take care of. We need to think judiciously to select which structure to use.

To explain them briefly;

Android service is a component that is used to perform operations on the background such as playing music. It doesn't has any UI (user interface). The service runs in the background but be aware that it runs on the main thread. Android Oreo has restricted services while app is not on foreground but there are a few solutions for this.

AsyncTask allows us to perform asynchronous work on our user interface. It performs the blocking operations in a worker thread and then publishes the results on the UI thread, without requiring you to handle threads and/or handlers yourself. We should deny it for long running operations.

IntentService is a base class for Services that handle asynchronous requests (expressed as Intents) on demand. Clients send requests through startService(Intent) calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.

A **thread** is a single sequential flow of control within a program. Threads can be thought of as mini-processes running within a main process. This is the base of all background jobs and it's specific to Java. So, we should deny using it unless it's the last option for us.

68. What are disadvantages of AsyncTask?

AsyncTask is flexible and easy to use. But in some cases there are some problems with it. We should avoid using AsyncTask for long running operations. Some problems in AsyncTask are;

- When you rotate your screen, Activity gets destroyed, so AsyncTask will not have a valid reference to publish data from onPostExecute(). In order to retain it, you need to usesetRetainState(true) if calling from fragment or onConfigChanges() if calling from activity method of an activity.

- When the Activity is restarted, your AsyncTask's reference to the Activity is invalid, so onPostExecute() will have no effect on the new Activity.

- If activity gets finished, AsyncTask execution will not cancelled automatically, you need to cancel them else they will keep on running in the background.

- If any exception occurs while performing network task, you need to handle them manually.

69. How can you manage multiple Threads running at the same time?

To allow multiple tasks to run at the same time, we need to provide a managed collection of threads. To do this, we use an instance of **ThreadPoolExecutor** , which runs a task from a queue when a thread in its pool becomes free. To run a task, all we have to do is add it to the queue.

A thread pool can run multiple parallel instances of a task, so we should ensure that our code is thread-safe. Enclose variables that can be accessed by more than one thread in a synchronized block. This approach will prevent one thread from reading the variable while another is writing to it. Typically, this situation arises with static variables, but it also occurs in any object that is only instantiated once.

For instance, let’s assume you want to synchronise threads for downloading photos. You need to create an instance of `ThreadPoolExecutor` class.

```
object PhotoManager {
  ...
   // Sets the amount of time an idle thread waits before terminating
   private const val KEEP_ALIVE_TIME = 1L
   // Sets the Time Unit to seconds
   private val KEEP_ALIVE_TIME_UNIT = TimeUnit. SECONDS
   // Creates a thread pool manager
   private val mDecodeThreadPool : ThreadPoolExecutor =
ThreadPoolExecutor(
          NUMBER_OF_CORES,      // Initial pool size
           NUMBER_OF_CORES,      // Max pool size
           KEEP_ALIVE_TIME ,
           KEEP_ALIVE_TIME_UNIT ,
          mDecodeWorkQueue
  )
```

70. What are advantages of AsyncTask over Thread?

Thread should be used to separate long running operations from main thread so that performance is improved. But it can’t be cancelled elegantly and it can’t handle configuration changes of Android. We can’t update UI from Thread.

```
val thread = Thread( Runnable {
    val bitmap = processBitMap( "image.png" );
   mImageView.post( Runnable () {
       mImageView.setImageBitmap(bitmap);
    } )
} ).start()
```

AsyncTask can be used to handle work items shorter than 5ms in duration. With AsyncTask, we can update UI unlike Java *Thread* . But many long running tasks will choke the performance.

```
class DownloadFilesTask: AsyncTask<URL, Integer, Long>() {

    override fun onPreExecute() {
        super .onPreExecute()
   }

    override fun doInBackground( vararg url: URL?): Long {
        return 0
    }

    override fun onProgressUpdate( vararg values: Integer?) {
        super .onProgressUpdate(*values)
   }
```

```
    override fun onPostExecute(result: Long?) {
        super .onPostExecute(result)
    }
}


// How to call
DownloadFilesTask().execute(url1, url2, url3);
```

The three types used by an asynchronous task are the following:

- Params, the type of the parameters sent to the task upon execution.
- Progress, the type of the progress units published during the background computation.
- Result, the type of the result of the background computation.

71. What is onTrimMemory() method? When does it called?

onTrimMemory(): Called when the operating system has determined that it is a good time for a process to trim unneeded memory from its process. This will happen for example when it goes in the background and there is not enough memory to keep as many background processes running as desired.

Android can reclaim memory for from our app in several ways or kill our app entirely if necessary to free up memory for critical tasks. To help balance the system memory and avoid the system's need to kill our app process, we can implement the ComponentCallbacks2 interface in our Activity classes. The provided onTrimMemory() callback method allows our app to listen for memory related events when our app is in either the foreground or the background, and then release objects in response to app lifecycle or system events that indicate the system needs to reclaim memory.

For instance;

```
public class MainActivity : AppCompatActivity(), ComponentCallbacks2 {


    override fun onTrimMemory(level: Int) {
        super .onTrimMemory(level)

        // Determine which lifecycle or system event was raised.
        when (level) {

            ComponentCallbacks2. TRIM_MEMORY_UI_HIDDEN -> {
                // Release any UI objects that currently hold memory.
            }
            ComponentCallbacks2. TRIM_MEMORY_RUNNING_MODERATE ,
            ComponentCallbacks2. TRIM_MEMORY_RUNNING_LOW ,
            ComponentCallbacks2. TRIM_MEMORY_RUNNING_CRITICAL -> {
                // Release any memory that your app doesn't need to run.
             }

            ComponentCallbacks2. TRIM_MEMORY_BACKGROUND ,
            ComponentCallbacks2. TRIM_MEMORY_MODERATE ,
            ComponentCallbacks2. TRIM_MEMORY_COMPLETE -> {
                // Release as much memory as the process can.
             }


             else -> { //  Release any non-critical data structures.}

             }
        }

    }
}
```

72. How do you find memory leaks in an Android application?

First of all, in Android Studio we have Memory Profiler tool. It helps us identify memory leaks and memory churn that can lead to stutter, freezes, and even app crashes. It shows a real time graph of our app's memory use, lets us capture a heap dump, force garbage collections, and track memory allocations.

Besides that we have the famous LeakCanary library. It is really easy to use and it again helps us to find memory leaks.

But while developing apps our priority should be to avoid things that cause memory leaks more than trying to find leaks after development. So, the question should be “How to avoid memory leaks?”.

Ok?

We have a bunch of things to avoid memory leaks in our apps. Let me roughly mention a few of them.

- Never hold the reference of ui specific object in a background task, as it leads to memory leak.
- Do not use static views as it is always available, static views never get killed.
- Never use context as static. Context is an expensive object.
- Use application context if possible and use activity context only if required.

```
class MainActivity : AppCompatActivity() {
    companion object {

        private val button : Button? = null   //NEVER USE LIKE THIS
        private val context : Context? = null //NEVER USE LIKE THIS
    }

}
```

- Do not forget to unregister your listeners in onPause() / onStop() / onDestroy() method. By not unregistering, it always keeps the activity alive and keeps waiting for listeners.

- If you are using an inner class, use this as static because static class does not need the outer class implicit reference. Using inner class as non static makes the outer class alive so it is better to avoid.

- And if you are using views in static class, pass it in the constructor and use it as a weak reference.

- Avoid putting views in collections that do not have clear memory pattern.

```
class LocationListenerActivity : Activity(), LocationListener {

    public override fun onStart() {
      LocationListener.get().register( this )
   }

    override fun onLocationChanged(location: Location?) {

   }

    public override fun onStop() {
      LocationListener.get().unregister( this )
   }

}
```

73. What are ways of scheduling jobs intelligently?

Android SDK provides several facilities to help us schedule work intelligently. First of the is **JobScheduler** .

JobScheduler:
JobScheduler does scheduling work for us that it improves app's performance along with aspects of the system health such as battery life.

JobScheduler is implemented in the platform, which allows it to collect information about jobs that need to run across all apps. This information is used to schedule jobs to run at, or around, the same time.

The JobScheduler API allows us to specify robust conditions for executing tasks, along with centralized task scheduling across the device for optimal system health. It is also suitable for small tasks like clearing a cache, and for large ones such as syncing a database to the cloud.

AlarmManager API:
The **AlarmManager API** is another option that the framework provides for scheduling tasks. This API is useful in cases in which an app needs to post a notification or set off an alarm at a very specific time.

You should only use this API for tasks that must execute at a specific time, but don't require the other, more robust, execution conditions that JobScheduler allows you to specify, such as device idle and charging detect.

Firebase JobDispatcher:

Firebase JobDispatcher is an open-source library that provides an API similar to JobScheduler in the Android platform. Firebase JobDispatcher serves as a JobScheduler-compatibility layer for apps targeting versions of Android lower than 5.0 (API level 21).

74. What is ANR? When it happens? How to avoid it?

First of all ANR means Application Not Responding. It happens when application is blocked, unable to work etc… The operating system interrupts and says “This is an ANR” and shuts the app down.

According to the official documentation of Android, there are two scenarios for an ANR to happen.

First , if there is no response for an input event for 5 seconds, then this is an ANR. For instance you have pressed a key on keyboard and for five seconds there isn’t any response, Android OS interrupts and throws an ANR.

Secondly , assume that you have a BroadcastReceiver. By the time it receives a broadcast message, then it has 10 seconds to finish its execution. Otherwise it is yet another ANR.

75. Have a look at the below code. Does it cause ANR?

```
class MainActivity : AppCompatActivity() {

   override fun onCreate(savedInstanceState: Bundle?, persistentState:
PersistableBundle?) {
       super .onCreate(savedInstanceState, persistentState)

      Thread.sleep( 15000 ) // sleep for 15 seconds
   }

}
```

Ok, this a very tricky question. Let’s have a look to the code. There seems an Activity and in onCreate() method there is a Thread.sleep(15000) code that makes thread sleep for 15 seconds.

Does it cause ANR?

First of all, we need to know the definition of ANR. Google’s official documentation says that if there is no response for an input event for 5 seconds, then this is an ANR. But the tricky part here is it must be an input event, such as pressing a key on keyboard.

Since this code makes thread sleep more than five second, it is not an input event. So, the answer is NO, it doesn’t cause an ANR.

CHAPTER 9

RECYCLERVIEW, LOADING DATA WITH PERFORMANCE

76. How does RecyclerView differ from Listview?

Recyclerview implements the **ViewHolder** pattern whereas it is not mandatory in a ListView. A RecyclerView recycles and reuses cells when scrolling .

What is a ViewHolder Pattern?
A ViewHolder object stores each of the component views inside the tag field of the Layout, so we can immediately access them without the need to look them up repeatedly.

In ListView, the code might call findViewById() frequently during the scrolling of ListView, which can slow down performance. Even when the Adapter returns an inflated view for recycling, we still need to look up the elements and update them. A way around repeated use of findViewById() is to use the **ViewHolder** design pattern.

In a ListView, the only type of view available is the vertical ListView. A RecyclerView decouples list from its container so we can put list items easily at run time in the different containers (linearLayout, gridLayout) by setting **LayoutManager** .

ListViews are lacking in support of good animations, but the RecyclerView brings a whole new dimension to it. RecyclerViews use **Item Animators** for supporting animations.

To sum up, ViewHolder pattern, LayoutManager and Itam Animators are the basic advantages of RecyclerView over ListView.

77. What is SnapHelper? Can you briefly explain how to use it?

SnapHelper is a helper class that helps in snapping any child view of the RecyclerView. For example, you can snap the firstVisibleItem of the RecyclerView as you must have seen in the play store application that the firstVisibleItem will be always completely visible when scrolling comes to the idle position

```
val snapHelper = LinearSnapHelper()
snapHelper.attachToRecyclerView(yourRecyclerView)
```

78. How can we efficiently update RecyclerView when only a small amount of data is updated?

When adapter’s data is changed we have to notify adapter in order to update RecyclerView. The most common way of this is calling *notifyDataSetChanged()* method.

But the problem is this method causes adapter to update all the indices in data. That is going to slow down our app. If we are writing an adapter it will always be more efficient to use the more specific change events if we can. We should rely on *notifyDataSetChanged()* as a last resort.

There are overloaded versions of this methods:

notifyItemChanged(int position)
notifyItemInserted(int position)
notifyItemRemoved(int position)
notifyItemRangeChanged(int positionStart, int itemCount)
notifyItemRangeInserted(int positionStart, int itemCount)
notifyItemRangeRemoved(int positionStart, int itemCount)

The most appropriate method is *notifyItemRangeChanged (int positionStart, int itemCount)* .

This is an item change event, not a structural change event. It indicates that any reflection of the data in the given position range is out of date and should be updated. The items in the given range retain the same identity.

```
void notifyItemRangeChanged (int positionStart, int itemCount)
```

79. How does RecyclerView work?

RecyclerView is designed to display long lists of items. Say we want to display 100 row of items. A simple approach would be to just create 100 views, one for each row and lay all of them out.

But that would be wasteful because at any point of time, only 10 or so items could fit on screen and the remaining items would be off screen.

So RecyclerView instead creates only the 10 or so views that are on screen. This way you get 10x better speed and memory usage.

What if happens when you start scrolling and need to start showing next views?

Again a simple approach would be to create a new view for each new row that you need to show.
But this way by the time you reach the end of the list you will have created 100 views and your memory usage would be the same as in the first approach. And creating views takes time, so your scrolling most probably wouldn't be smooth.

This is why RecyclerView takes advantage of the fact that as you scroll, new rows come on screen also old rows disappear off screen.

Instead of creating new view for each new row, an old view is recycled and reused by binding new data to it.

This happens inside the `onBindViewHolder()` method. Initially you will get new unused view holders and you have to fill them with data you want to display. But as you scroll you will start getting view holders that were used for rows that went off screen and you have to replace old data that they held with new data.

CHAPTER 10

DATABASE

80. Why do we use SQLite instead of SQL in mobile apps?

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.

Advantage of SQLite:

- An SQLite database can also be queried and the data retrieval is much more robust.
- The android.database and android.database.sqlite packages offer a higher-performance alternative where source compatibility is not an issue.

- Android-databases created in Android are visible only to the application that created them
- There is no file parsing and generating code to write and debug.
- Content can be accessed and updated using powerful SQL queries, greatly reducing the complexity of the application code.
- Extending the file format for new capabilities in later releases is as simple as adding new tables or new columns to existing tables.

- The application file is portable across all operating systems, 32-bit and 64-bit and big- and little-endian architectures.
- The application only has to load as much data as it needs, rather than reading the entire application file and holding a complete parse in memory. Startup time and memory consumption are reduced.
- Small edits only overwrite the parts of the file that change, not the entire file, thus improving performance and reducing wear

on SSD drives.

- Multiple processes can attach to the same application file and can read and write without interfering with each other.
- In many common cases, loading content from an SQLite database is faster than loading content out of individual files. See Internal Versus External BLOBs for additional information.

81. What are the SQLite storage classes?

Each value stored in an SQLite database (or manipulated by the database engine) has one of the following storage classes:

- **NULL** . The value is a NULL value.
- **INTEGER** . The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL** . The value is a floating point value, stored as an 8-byte IEEE floating point number.
- **TEXT** . The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB** . The value is a blob of data, stored exactly as it was input.

82. What is DAO Pattern? How do we use it and what is its advantage?

Data Access Object Pattern or **DAO** pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).

```
interface Dao {
    fun getAllStudents(): List<Student>;
}
```

Data Access Object concrete class - This class implements DAO interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.

```
class DummyDaoImpl :Dao{
    override fun getAllStudents(): List<Student> {

    }

}

class NetworkDaoImpl :Dao{
    override fun getAllStudents(): List<Student> {

   }

}
```

Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class

```
data class Student( val name : String)
```

Main advantage of DAO pattern is that we can quickly change source of data in our app. While developing apps, we may want usu dummy datas before using real data. DOA pattern gives us this advantage.

ROOM persistence library also uses this pattern for a flexible structure.

83. How can you separate your data layer from presentation?

Android's framework team recently has introduced a new architecture. They want us to separate presentation layer and data layer. For this purpose they recommend us to use Repository Pattern.

Repository is a pattern and it is the single source of truth for all the app data. It has a clean API for UI to communicate with.

Whenever we need data, we go to Repository. It doesn't matter where the data comes from. From local db or network or file storage or SharedPrefs or somewhere else.

Here we have a simple Repository example:

```
class UserRepository( val userApi : UserApi, val userDao : UserDao) {

   fun getUsers(): Observable<List<User>> {
      return Observable.concatArray(
           getUsersFromDb(),
           getUsersFromApi())
  }


   fun getUsersFromDb(): Observable<List<User>> {
      return userDao .getUsers().filter { it.isNotEmpty() }
            .toObservable()
           .doOnNext {
               Log.d( "Dispatching ${ it. size }  users from DB..." )
            }
   }

   fun getUsersFromApi(): Observable<List<User>> {
      return userApi .getUsers()
```

```
            .doOnNext {
                Log.d( "Dispatching ${ it. size }  users from API..." )
               storeUsersInDb(it)
            }
    }

    fun storeUsersInDb(users: List<User>) {
      Observable.fromCallable { userDao .insertAll(users) }
             .subscribeOn(Schedulers.io())
            .observeOn(Schedulers.io())
            .subscribe {
                Log.d( "Inserted ${ users. size }  users from API in DB..." )
            }
    }

}
```

Here we we clearly see the advantage of the Repository Pattern. This repository returns list of users from local db or network. It also have a method to persist users.

These methods can increment against your needs. There is no limit here. But we should design our repository so clean that it should be very easy to use.

As you can see, it’s compatible to be used with RxAndroid. Reactive programming in repository is a latest trend in Android world.

Yet another benefit is, repository takes API and Dao as a parameter and work with them. That makes our app more testable.

84. Do you use any ORM libraries in your apps?

Before REALM, there wasn’t any successful ORM libraries in Android. I have used this library for a while in my projects. It’s first advantage is it is very easy to use as it should be. But there were some problems also with it. You have to extend RealmObject class in your POJO’s. That is a disadvantage for inheritance. It also increases APK size tremendously in some cases.

Later then, ROOM has reached out. As a part of Android Architecture Components, ROOM is the only library that i use for persistence in my apps.

It provides compile-time validation for SQL. It’s unbelievable! SQL is very open to typos and ROOM saves us.

Secondly it is very easy to use. All you need is a DAO interface. You don’t even have to implement the methods. That means no more boilerplate code in database methods.

For instance;

```
@Dao
interface UserDao {
  @Query( "SELECT * FROM user" )
  List<User> getAll();

  @Query( "SELECT * FROM user WHERE uid IN (:userIds)" )
  List<User> loadAllByIds(int[] userIds);

  @Query( "SELECT * FROM user WHERE first_name LIKE :first AND "
       + "last_name LIKE :last LIMIT 1" )
  User findByName(String first, String last);

  @Insert
  void insertAll(User... users);
```

```
   @Delete
   void delete(User user);
}
```

AppDatabase class:

```
@Database( entities = { User. class } , version = 1 )
abstract class AppDatabase: RoomDatabase {
    public abstract UserDao userDao();
}
```

After creating the files above, you get an instance of the created database using the following code:

```
var db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase:: class . java , "database-name" ).build()
```

85. You are supposed to build a “URL Shortener” app. How do you design your database?

Ok, when designing database tables we should always design according to objects. So the question should always be “What objects we need?” rather than “What tables we need?”.

Let’s begin with designing our model objects which are called as entity.

Let's call our main entity a **Link** . A Link is a mapping between a **shortLink** on our app, and a **longLink** , where we redirect people when they visit the shortLink.

So, our data class is,

```
data class Link( val shortLink : String, val longLink : String)
```

Here we are benefitting from Data Class syntax of Kotlin. We don’t have to annotate it as an entity or define serialized names. They are optional.

Then of course we need a DAO. I always use DAO pattern in my apps. Since ROOM is the standard library for ORM, we are going to use it.

Here is our DAO object;

```
@Dao
interface UrlShortenerDao{
  @Query( "SELECT * FROM Link WHERE shortLink LIKE :shortLink  LIMIT
1 " )
  Link get(shortLink: String);

  @Insert
```

```
    void insert(link: Link);

    @Delete
    void delete(link: Link);
}
```

We have three methods: get(), insert(), and delete(). Since this is the first version of our app, we have saved update() method for later.

Actually, it seems pretty simple and efficient. ROOM requires us to write a small query of SQL in our get() method but the rest is pretty easy.

Now we need to write a database class.

```
@Database( entities = { Link. class } , version = 1 )
abstract class AppDatabase : RoomDatabase {
    public abstract UrlShortenerDao urlShortenerDao();
}
```

Now, we are ready to to. All we need is to write a Repository class and call regarding DAO methods from that Repository.

You know, Repository is a pattern. It is the single source of truth for all the app data and it has a clean API for UI to communicate with.

I have to also underline that, i would design my app as three different layers: Presentation, Data and Domain. So far, we have designed our Data layer.

This architecture is called as Clean Architecture and i try to use it in all of my apps. It helps us to satisfy Separation of Concerns Pattern and provides a more robust, scalable, testable structure.

✓

Important:

System design questions like this are usually intentionally left open-ended, so you have to ask some questions and make some decisions about exactly what you're building to get on the same page as your interviewer.

Do not hesitate to ask questions to your interviewer and also make your own decisions. This is supposed to be an interactive question.

You should also mention about architecture and three layers we have mentioned. This is a big plus for you.

CHAPTER 11

ANDROID STUDIO

86. What is ADB? Why do we need it and how can we benefit from it?

ADB is Android Debug Bridge. It is a versatile command-line tool that lets you communicate with a device. Using ADB we can install, uninstall and debug apps on real devices. It basically provides access to a Unix shell that we can run a variety of commands.

The most important commands are as follows:

To connect a device over Wi-Fi:

```
$ adb tcpip 5555
$ adb connect device_ip_address
```

To list of attached devices:

```
$ adb devices
```

To install an APK on device:

```
$ adb -s emulator-5555 install helloWorld.apk
```

To setup port forwarding:

```
$ adb forward tcp:6100 tcp:7100
```

To copy files to/from a device:

```
$ adb forward tcp:6100 tcp:7100
```

87. What are the most used shortcuts of Android Studio you use?

Our job requires to be working on a computer throughout each standard 8-hour workday, then using keyboard shortcuts can save us 8 entire work days every year. This is equal to 3.3% of our total productivity!

My favorite shortcuts are for formatting and navigation. Here are my favorite shortcuts that i use oftenly.

Shift+Shift — Search Everywhere
Ctrl+F — Find Text with in a Single File
Ctrl+Shift+F -Find Text in All Files
Ctrl+N — Navigate to Class
Ctrl+Shift+N — Navigate to a File
Ctrl+E — Recent Files
Ctrl+Shift+BackSpace — Jump to Last Edited Location
Ctrl + W — Extend selection (selects a word->line->method->Class)
Ctrl + / — Comment / uncomment with line comment
Ctrl+Alt+L — Reformat code
Ctrl + Alt + T — Surround with…

During a long workday, we repeat some common tasks over and over again. These shortcuts saves us time and makes us more productive.

✓

Important:

You don’t have to memorize these shortcuts. But you should keep in mind some shortcuts for yourself. When you are asked this question, you should at least say “I use shortcuts for, running, debugging, running tests, opening classes etc...”

That is true during a long workday, we repeat some common tasks over and over again. These shortcuts saves us time and makes us more productive.

This is going to help you after interview also :)

88. What is Android Profiler ? How can we use it?

The **Android Profiler** provides real time data for our app's CPU, memory, and network activity . We can perform sample-based method tracing to time our code execution, capture heap dumps, view memory allocations, and inspect the details of network-transmitted files.

Before Android Studio 3.0 there was Android Monitor tools. Android Profiler has replaced that tools.

In android Studio, it is located under **View** > **Tool Windows** > **Android Profiler** (we can also click Android Profiler in the toolbar).

89. How do you protect your source codes from being decompiled?

Actually there is no way to stop reverse engineering completely. But we use Proguard for obfuscating our code.

An APK can always be decompiled but if it is obfuscated, things are much complicated to understand. Method names and variable names are obfuscated thanks to Proguard.

Here is how we enable Proguard:

```
android {

  buildTypes {

      release {

          minifyEnabled false

           proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
'proguard-rules.pro'

       }

  }

}
```

But in some cases obfuscation may cause some RunTime Exceptions. Our app may throw ClassNotFoundException interestingly. Generally for libraries and third party classes. We need to keep these classes out of obfuscation.

For this we use:

```
-keep public  class  MyClass
```

90. You are supposed to deliver two different APK's of your app. (For instance a car booking app for drivers and riders) How can you manage it in same project?

Android Studio provides us **Product Flavors** for this purpose. Creating product flavors is similar to creating build types: add them to the productFlavors block in your build configuration and include the settings you want. The product flavors support the same properties as defaultConfig —this is because defaultConfig actually belongs to the **ProductFlavor** class.

For instance we have a car sharing app and we want to deliver two different APK's for riders and drivers. We should create two product flavors in our project: Driver and Rider.

The following code sample creates a flavor dimension named " version " and adds "driver" and "rider" product flavors. These flavors provide their own applicationIdSuffix and versionNameSuffix :

```
android {
  ...
  defaultConfig {...}
  buildTypes {
     debug{...}
     release{...}
  }
   // Specifies one flavor dimension.
   flavorDimensions "version"
   productFlavors {
     driver {
         // Assigns this product flavor to the "version" flavor dimension.
        // This property is optional if you are using only one dimension.
         dimension "version"
```

```
            applicationIdSuffix ".driver"
            versionNameSuffix "-driver"
        }
       full {
           dimension "version"
            applicationIdSuffix ".rider"
            versionNameSuffix "-rider"
        }
   }
}
```

91. How do you analyze your apk? Why its size is larger than expected?

Android Studio includes an **APK Analyzer** that provides immediate insight into the composition of our APK after the build process completes.

I generally use this tool to reduce the time i spend debugging issues with DEX files and resources within my app, and help reduce my APK size. It's also available from the command line.

It provides us a bunch of functionalities such as:

- View the absolute and relative size of files in the APK, such as the DEX and Android resource files.
- Understand the composition of DEX files.
- Quickly view the final versions of files in the APK, such as the AndroidManifest.xml file.
- Perform a side-by-side comparison of two APKs.

I see how much space /res folders occupy. If there is a problem with size of resources we should use shrinkResources command to delete unused resources.

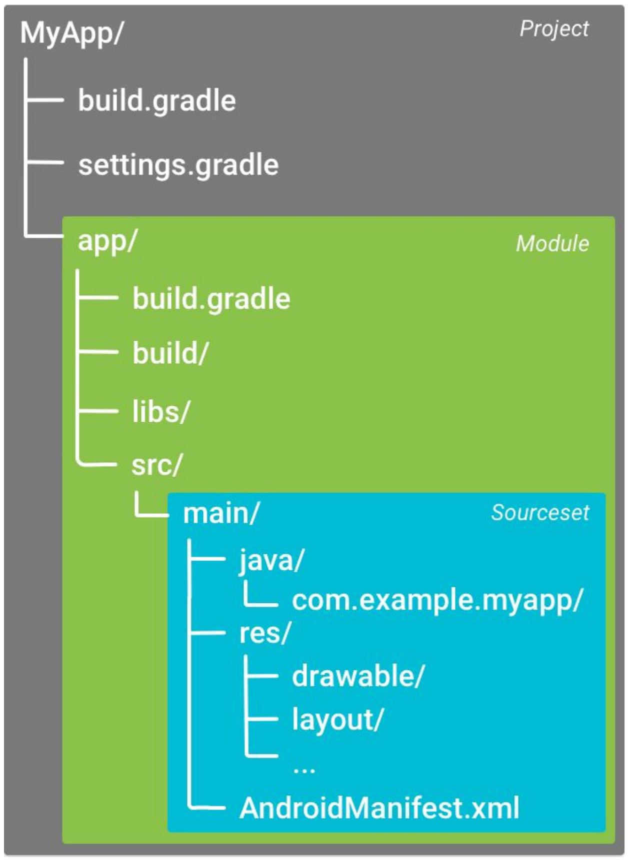
We may also use WebP file format instead of PNG. There are a tons of ways that Google recommends to reduce APK size. These are just a few of them.

92. What do you know about Gradle? How does it work?

Gradle is an advanced build toolkit to automate and manage build process. It allows us to define custom build configurations. Android Studio uses Gradle for build automation. We define dependencies of our project and configure details of our build process.

It was first developed by Gradleware Inc and open-sourced then.

Gradle and the Android plugin run independent of Android Studio. This means that we can build your Android apps from within Android Studio, the command line on our machine, or even on machines where Android Studio is not installed such as continuous integration servers .



In an Android project we have build.gradle file for entire project and a *build.gradle* file for every module.

The *settings.gradle* file, located in the root project directory, tells Gradle which modules it should include when building our app.

The top-level *build.gradle* file, located in the root project directory, defines build configurations that apply to all modules in our project.

To add a dependency to our project, we specify a dependency configuration such as implementation in the dependencies block of our *build.gradle* file.

We are also allowed to add extra properties to the ext block in the top-level *build.gradle* file. So that we can define version numbers as a property and use it for different dependencies that use same version. Such as support libraries.

Creating custom build configurations requires us to make changes to one or more build configuration files, or build.gradle files. These plain text files use **Domain Specific Language (DSL)** to describe and manipulate the build logic using Groovy, which is a dynamic language for the Java Virtual Machine (JVM). But we don't have to know Groovy to build using Gradle. It's straightforward most of the time.

A simple Gradle file should be as follows:

```
apply plugin: 'com.android.application'

android { ... }

dependencies {

   // Dependency on a local library module

   implementation project( ":mylibrary" )

   // Dependency on local binaries

   implementation fileTree(dir: 'libs' , include: [ '*.jar' ])

   // Dependency on a remote binary

   implementation 'com.example.android:app-magic:12.3'

}
```

CHAPTER 12

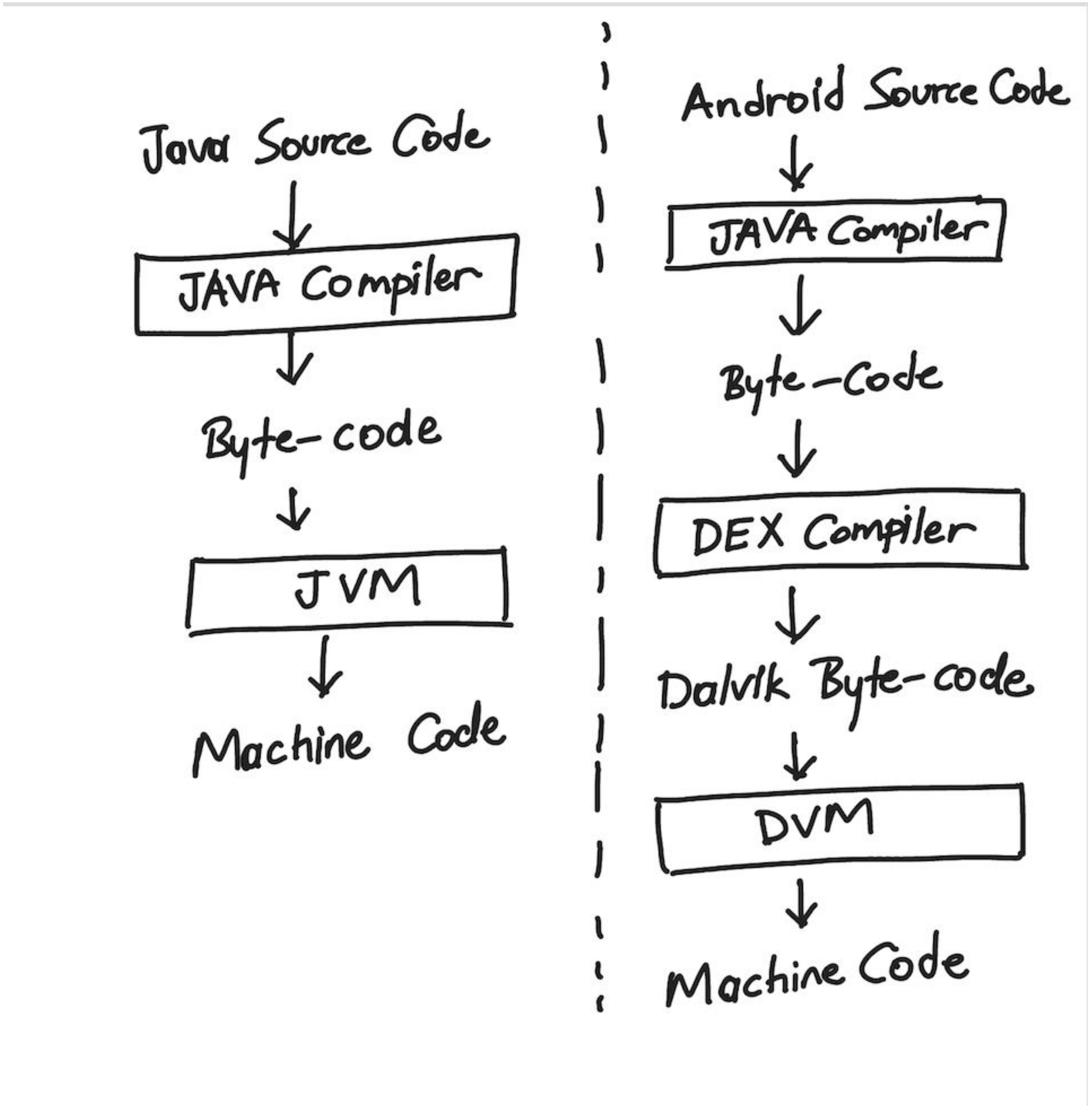
ANDROID SYSTEM INTERNAL

93. How Android code is compiled?

Java code is compiled to an intermediate format called **byte-code** . Then this byte-code is parsed by JVM and translated to machine code.

Android SDK has a tool called **dx** that converts Java byte-code to **Dalvik byte-code** . It is located under build-tools/android-{version} .

Then Dalvik VM moves in and translates Dalvik byte-code to binary machine code.



Maybe the most significant difference between the both is that JVM is stack-based while DVM is register-based.

What is stack based and register based?

Ok, let me briefly explain:

In **Stack Based VM's** , all the arithmetic and logic operations are carried out via Push and Pop operands and results in the **stack** .

On the other hand, in **Register Based VM's** , the data structure where the operands are stored is based on the **registers of CPU** .

Important:

Main point behind this question is that your interviewer wants to measure you Computer Science background.

You should definitely know how Android code is compiled. I have provided you a diagram of it. Try to draw it on a paper 10 times. Then when you are asked, take a pen and draw it on a whiteboard or at least on a paper.

Describe the process with drawing. That is the main point of a successful answer in such questions. They want to see if you are able to document solutions with diagrams and teach others easily.

Albert Einstein says that "If you can't explain it simply, you don't understand it enough.".

94. What is DEX code?

Dex files are Dalvik executable files for both ART and Dalvik runtimes. They combine the power of our four protagonists creating the bytecode that runs on Android devices.

The dex file is most known for the infamous 65k method count limit but it contains more than just a method table. More precisely, it aggregates content from the app module and all of its dependencies. The dex file format contains the following elements:

1. File Header
2. String Table
3. Class List
4. Field Table
5. Method Table
6. Class Definition Table
7. Field List
8. Method List
9. Code Header
10. Local Variable List

But the road to the dex file is long and winding. First, the Java source code is compiled using javac (part of the JDK) to create the .class files. Afterwards, using the dx tool (part of the Android SDK build tools), the Java bytecode .class files are translated to the .dex files.

✓

Important:

For more details about DEX code, please refer to “Question 93. How Android code is compiled?”

95. What is ART?

Android runtime (ART) is the managed runtime used by applications and some system services on Android. ART and its predecessor Dalvik were originally created specifically for the Android project. ART as the runtime executes the Dalvik Executable format and Dex bytecode specification.

ART and Dalvik are compatible runtimes running Dex bytecode, so apps developed for Dalvik should work when running with ART. However, some techniques that work on Dalvik do not work on ART.

Here are some of the major features implemented by ART.

- Ahead-of-time (AOT) compilation
- Improved garbage collection
- Support for sampling profiler
- Support for more debugging features
- Improved diagnostic detail in exceptions and crash reports

ART also provides improved context information in app native crash reports, by including both Java and native stack information.

96. How can you call Garbage Collector in your app?

Nope, you can’t call Garbage Collector (GC). Simply, Java doesn’t allow us to do this.

In fact, you shouldn't be trying to force GC - if you are running low on memory then you have a memory leak somewhere. Forcing GC at that point won't help, because if you are holding a reference to the object then it still won't be garbage collected.

CHAPTER 13

ADVANCED KOTLIN

97. What is difference between “==” and “===” in Kotlin?

In Kotlin there are two types of equality:

- Structural equality
- Referential equality

Structural equality is checked by the **==** operation. By convention, an expression like *a == b* is translated to:

```
a?.equals(b) ?: (b === null )
```

And its negated counterpart **!=** .

For **referential equality** , we use the **===** symbol which allows us to evaluate the reference of an object if it’s pointing to the same object. This is an equivalent of “==” operator in Java.

Let’s say we have two integers defined:

```
val a = Integer( 10 )
val b = Integer( 10 )
```

and we check them both by doing *a === b* , which will return false because they’re two separate objects, each pointing to a different location in memory.

98. How can we define a Singleton object in Kotlin?

In Java, we make constructor private, write a *static getInstance()* method that returns a *private static final* instance of that class.

But in Kotlin, things are much simpler with a much better syntax.

In Kotlin the best way to define a **Singleton** is to create an object . Here it is:

```
object Singleton {
   init {
      println ( "This ( $this ) is a singleton" )
  }

   var b :String? = null
}
```

This object is a Singleton. Whenever we instantiate it, we always have the same object over and over again.

```
var first = Singleton     //This is a singleton
first.b = "hello singleton"
var second = Singleton
println(second.b)        //Prints “hello singleton” again.
```

✓

Important:

Singleton is a commonly used Design Pattern, also easy to implement and explain. But keep in mind that singleton is hard to test.

99. What are “Inline Functions” in Kotlin?

inline annotation means that function as well as function parameters will be expanded at call site that means it helps reduce call overhead.

Let me explain it more clearly. High order functions are stored as objects may make the use disadvantageous at times because of the memory overhead. **Inline functions** are introduced for this purpose.

Declaring a function inline is simple, just add inline keyword to the function like in below code snippet :

```
inline fun someMethod(a: Int, func: () -> Unit):Int {
   func()
    return 2 *a
}
```

100. How to declare mutable and immutable objects in Kotlin?

In Kotlin variables are declared using `var` or `val` , provided they are **mutable** or **immutable** .

`var` means this object is mutable, it can be re-assigned, its value can be changed.

On the other hand, `val` means this object is immutable. Its value can not be changed. It’s like `final` keyword in Java.

An example of mutable objects:

```
var x = 7
var y : String = "my String"
var z = View( this )
```

But in Kotlin **immutable** objects are always preferred.

```
val x : Int = 20
val y : Double = 21.5
val z : Unit = Unit
```

Due to convention of using val whenever possible, Kotlin provides much more safety.

101. What is a “Higher Order Function” in Kotlin?

A **higher-order function** is a function that takes functions as parameters, or returns a function.

Kotlin functions are *first-class*, which means that they can be stored in variables and data structures.
For instance here we have a *higherOrderFunction()* that takes another function as a parameter. The thing is we need to define input and output parameters of incoming function. Then inside *higherOrderFunction()* we can use this *fn* function.

```
fun higherOrderFunction(fn: () -> Unit){
   fn()
}
```

How can we benefit this?

We can change behavior of methods at runtime. For instance, we can send different methods with different algorithms to be executed at a higher order function.

Here we have two simple methods: **add** and **subtract**. Both of them takes two *Integers* and return an *Integer* as a result. Their parameter signature is same.

```
val add = fun (num1: Int, num2: Int): Int{
    return num1 + num2
}

val subtract = fun (num1: Int, num2: Int): Int{
    return num1 - num2
}
```

In Kotlin, we can assign functions to variables.

Then we can write a higher order function to take one of this functions with two integers. Then it’s going to execute incoming function with incoming parameters.

Here our higher order function:

```
fun higherOrderFunction(fn: (param1: Int, param2: Int) -> Int, num1: Int, num2: Int){
    val result = fn(num1, num2)
    print ( "Result is $ result " )
}
```

As you can see, it defined parameter signature of incoming function: takes two Integers and return an Integer.

`fn: (param1: Int, param2: Int) -> Int`

Then we can call this method using both of the methods.

```
higherOrderFunction( add , 15 , 10 ) // prints 25
higherOrderFunction( subtract , 13 , 8 ) // prints 5
```

This how can we benefit from higher order functions.

102. What are “open”, “final” and “abstract” modifiers in Kotlin?

open : In Kotlin, classes are closed to be extended. That means by default, all classes are `final` . We have to use `open` keyword to make classes extendable.

`open` keyword can also be used with functions and properties. A function or property with `open` keyword can be overridden. Otherwise they are not allowed to be overridden.

Extending an open class:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

Overriding a function:

```
open class Base {
    open fun v() { ... }
}


class Derived () : Base() {
    override fun v() { ... }
}
```

Overriding a property:

```
open class Foo {
    open val x : Int get () { ... }
}

class Bar1 : Foo() {
    override val x : Int = ...
}
```

By default, all classes in Kotlin are final, which corresponds to **Effective Java** book Item 19: " *Design and document for inheritance or else prohibit it.* ".

final : Forbids overriding a member.

We have said classes and members are already closed to be extended. But why we need it?

Answer lies here: when a function or a property is overridden, it is open to inheritance. We use final keyword to close it to be overridden again.

For instance:

```
open class Base {
    open fun v() { ... }
}

open class Derived () : Base() {
    final override fun v() {   }
}

class DerivedSub () : Derived() {
    override fun v() { ... }   // function v() is final and cannot be overridden
}
```

Base class is `open` and it has . Since it is defined `open` , it can be overridden.

Derived class is also `open` and it extends *Base* . Derived has also overridden `open fun v()` . `override` keyword in `override fun v()` makes this method open to inheritance. But when this method is `final override` , then it's close to inheritance.

DerivedSub class tries to `override fun v()` but compiler gives error: "*function v() is final and cannot be overridden* "
To sum up, a member with `final` keyword cannot be overridden.

Abstract : An `abstract` class is a class that cannot be instantiated. We create abstract classes to provide a common template for other classes to extend and use.

Functions and properties can also be declared `abstract` . Those members have to be overridden by concrete subclasses.

An `abstract` class is also `open` . We don't need to use `open` keyword, it is `open` by default.

```
abstract class Device( val name : String,
                       val color : String,
                       val weight : Double) {   // Concrete (Non Abstract) Properties

  // Abstract Property (Must be overridden by Subclasses)
   abstract var maxSpeed : Double

   // Abstract Methods (Must be implemented by Subclasses)
   abstract fun start()
   abstract fun stop()

   // Concrete (Non Abstract) Method
   fun displayDetails() {
      println ( "Name: $ name , Color: $ color , Weight: $ weight , Max
Speed: $ maxSpeed " )
   }
}
```

One important detail is and **abstract** class may have **abstract** and non-abstract methods. And we can override a non-abstract open member with an **abstract** one:

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

Red Flag:

Interviewer wants to see if you are comfortable with **Object Oriented Programming** concepts.

Don't you even dare to mix meanings of **open** , **final** and **abstract** . These are basic keywords for **Object Oriented Programming** . We use these keywords for implementing **Encapsulation** , **Inheritance** and **Polymorphism** . Actually, Encapsulation is more related with access modifiers but these keywords are very important for other two concepts: Inheritance and Polymorphism.

Interviewer may also ask you these three concepts, you should be prepared.

For more detailed information i can provide you a nice article. Please refer to https://medium.freecodecamp.org/object-oriented-programming-concepts-21bb035f7260

103. What are Data Classes in Kotlin? What makes them so useful?

Data Classes are classes that holds data. Their only purpose is this. Pure, simple and elegant. We use them for model classes in our projects.

```
data class Book( val name : String, val publisher : String, var reviewScore :
Int)
```

Data classes use primary constructor and there must be at least one primary constructor parameter.

Parameters can be val or var. They may also have visibility modifiers such as **private** .

We instantiate a Data class as a normal object:

```
val book = Book( "1984" , "George Orwell" , 10 )
```

Kotlin also provides us a `copy()` function to easily copy data classes.

```
val book = Book( "1984" , "George Orwell" , 10 )

val otherbook = book .copy( name = "Animal Farm" )
```

Here we see, `copy()` method has copied book object and we have changed one of its properties. The rest is the same but it’s a new object with a new place on memory heap.

104. What is “Pair” and “Triple” in Kotlin?

Kotlin gives us **Pair** and **Triple** classes for holding more than one data in an object. Using Pair or Triple objects, our functions can return two or three values at the same time .

That saves us from writing a new class for data pairs. Let’s assume we have a method that returns a user’s name and age. How can you do it?

We definitely write a User class which has name and age fields and our function returns a User object.

But thanks to Kotlin, we no longer have to define a new class. Because we have **Pair** and **Triple** classes.

Here how to use it:

```
val user = Pair( "John" , 25 )
println (user. first )     // prints John
println (user. second )     // prints 25
```

Kotlin by default names pair items as `first` and `second` . We have to use this syntax to reach values.

We are also allowed to define Pair objects. User object created above can be used as:

```
val (name, age) = user
println (name)       // prints John
println (age)        // prints 25
```

Triple is same as Pair. The only difference is Pair can hold two values while Triple can three.

An example of Triple would be:

```
val actor = Triple( "John" , "Wick" , 25 )
println (actor. first )     // prints John
println (actor. second )     // prints Wick
println (actor. third )     // prints 25
```

To sum up, if we want to hold more than one data in an object without creating a special class for it, Kotlin provides us **Pair** and **Triple** classes.

105. What is difference between “lazy” and “lateinit” ?

Using **lateinit** , the initial value does not need to be assigned. However, we have to be careful to assign our lateinit var a value before we use it. Otherwise, a lateinit property acts as if we performed !!: it will crash the app on a null value.

A property defined via by **lazy** is initialized using the supplied lambda upon first use, unless a value had been previously assigned.

Lazy is a good fit for properties that may or may not be accessed. If we never access them, we avoid computing their initial value. They may work for Activities, as long as they are not accessed before setContentView is called. They are not a great fit for referencing views in a Fragment, because the common pattern of configuring views inside onCreateView would cause a crash. They can be used if view configuration is done in onViewCreated.

With Activities or Fragments, it makes more sense to use **lateinit** for their properties, especially the ones referencing views. While we don’t control the lifecycle, we know when those properties will be properly initialized. The downside is we have to ensure they are initialized in the appropriate lifecycle methods.

CHAPTER 14

COMMON MISTAKES & ERRORS

106. How can you reduce APK size?

We should remove unused resources . There is a tool in Android Studio called **lint** for this purpose. The **lint** tool detects resources in our *res/* folder that our code doesn't reference.

Libraries that we use may include unused resources . We may eliminate them with **shrinkResources** in our app's *build.gradle* file.

```
android {
   // Other settings

   buildTypes {
     release {
         minifyEnabled true
          shrinkResources true
          proguardFiles getDefaultProguardFile( 'proguard-android.txt' ),
'proguard-rules.pro'
       }
   }
}
```

We may support only specific densities . If we know that only a small percentage of your users have devices with specific densities, we may bundle those densities into our app. Android automatically scales existing resources originally designed for other screen densities.

We may prefer XML drawables or WEBP file format over PNG or JPG ones. The WebP format provides lossy compression as well as transparency but can provide better compression than either JPEG or PNG.

We can use vector graphics to create resolution-independent icons and other scalable media. Using these graphics can greatly reduce our APK footprint.

We should use **AnimatedVectorDrawableCompat** to create animated vector drawables instead of **AnimationDrawable** . AnimationDrawable requires that we include a separate bitmap file for each frame of the animation, which drastically increase the size of our APK.

On the code side, we may remove unnecessary generated code or avoid enumerations . A single enum can add about 1.0 to 1.4 KB of size to our app's *classes.dex* file. These additions can quickly accumulate for complex systems or shared libraries.

107. Do you know what is inside of an APK file?

An APK contains the following directories:

META-INF/: Contains the CERT.SF and CERT.RSA signature files, as well as the MANIFEST.MF manifest file.

assets/: Contains the app's assets, which the app can retrieve using an AssetManager object.

res/: Contains resources that aren't compiled into resources.arsc.

lib/: Contains the compiled code that is specific to the software layer of a processor. This directory contains a subdirectory for each platform type, like armeabi, armeabi-v7a, arm64-v8a, x86, x86_64, and mips.

An APK also contains the following files. Among them, only *AndroidManifest.xml* is mandatory.

resources.arsc: Contains compiled resources. This file contains the XML content from all configurations of the res/values/ folder. The packaging tool extracts this XML content, compiles it to binary form, and archives the content. This content includes language strings and styles, as well as paths to content that is not included directly in the resources.arsc file, such as layout files and images.

classes.dex: Contains the classes compiled in the DEX file format understood by the Dalvik/ART virtual machine.

AndroidManifest.xml: Contains the core Android manifest file. This file lists the name, version, access rights, and referenced library files of the app. The file uses Android's binary XML format.

108. Your app is not looking good on tablets. How can you solve this problem?

First of all we need to create a flexible layout . The best way to create a responsive layout for different screen sizes is to use **ConstraintLayout** as the base layout in our UI.

ConstraintLayout allows us to specify the position and size for each view according to spatial relationships with other views in the layout. This way, all the views can move and stretch together as the screen size changes.

We should create stretchable nine-patch bitmaps. This is a special kind of images that works fine on Android.

We should avoid hard-coded layout sizes. Instead we should use *wrap_content* or *match_parent* .

We should create alternative layouts . The app should provide alternative layouts to optimise the UI design for certain screen sizes. For eg: different UI for tablets.

We should use the smallest width qualifier. For instance, we can create a layout named *main_activity.xml* that's optimised for handsets and tablets by creating different versions of the file in directories as follows:

res/layout/main_activity.xml — For handsets (smaller than 600dp available width)
res/layout-sw600dp/main_activity.xml — For 7" tablets (600dp wide and bigger).

The smallest width qualifier specifies the smallest of the screen's two sides, regardless of the device's current orientation, so it's a simple way to specify the overall screen size available for our layout.

We should test text and graphics on *ldpi* and *mdpi* screen densities to ensure that our text and graphics work well on low- and medium-density screens.

Important:

Yet another generic question. As an Android developer, your daily life will be dominated with effort of making sure your app is looking good on different devices. You are developing for phones and tablets. Interviewer wants to see that you are capable of handling UI issues with tablets. Most of the times, apps are responsive and it may take a bit time to ensure this responsiveness. Your employer wants to be sure that you are good enough to handle that.

Just repeat the given answer. That will be well enough.

109. How can you optimize your app’s battery consumption?

There are several steps we can take to help make sure that our app is only consuming battery power when it needs to, and that it's not consuming more power than necessary.

Our app should minimize its activity when in the background and when the device is running on battery power.

Sensors, such as GPS sensors, can drain battery significantly. We should avoid issues by using the **FusedLocationProvider API** to manage the underlying location technology. It provides a simple API so that you can specify requirements such as high accuracy or low power at a high level. It also optimizes the device's use of battery power by caching locations and batching requests across apps.

We should avoid using wake locks because they prevent the device from going into low-power states.

We may batch network activity to reduce the number of device wake-ups.

We should use **GcmNetworkManager** to perform non-essential background activity when the device is charging and is connected to an unmetered network.

GcmNetworkManager schedules tasks and lets Google Play services batch operations across the system. This greatly simplifies the implementation of common patterns, such as waiting for network connectivity, device charging state, retries, and backoff.

110. How do you protect your API keys in your project? So that no one sees them in commited code on repo.

We should never hard code keys in code. We should always add the key to our *gradle.properties* file in our home directory under *.gradle* directory.

Then we may import the key as a *buildConfigField* / *resValue* in our module-level *build.gradle* file.

Then we are ready to use the key in our code or XML files as we need.

For instance, build.gradle file:

```
buildTypes {

  debug {
     buildConfigField 'String' , "ApiKey" , My_ApiKey
     resValue 'string' , "api_key" , My_ApiKey
  }

  release {
     buildConfigField 'String' , "ApiKey" , My_ApiKey
     resValue 'string' , "api_key" , My_ApiKey
  }
}
```

Then in Kotlin code, we may get the key as follows:

```
val apiKey: String = BuildConfig. ApiKey ;
```

PART 3

EXPERT LEVEL QUESTIONS

CHAPTER 15

DESIGN PATTERNS, PRINCIPLES, ARCHITECTURES

111. What design patterns are commonly used in Android apps?

Design patterns and principles are huge topics, i can't explain them one by one because it may take days even weeks. But i can briefly explain used design patterns in Android.

Builder
Builder pattern separates construction of a complex object from its representation. In Android, the Builder pattern appears when using objects like AlertDialog.Builder.

```
AlertDialog.Builder( this )
      .setTitle( "Mahatma Gandhi " )
      .setMessage( "Be the change you wanna see in the world." )
      .setNegativeButton( "No thanks" , { dialogInterface, i ->
          // "No thanks" button was clicked
       } )
      .setPositiveButton( "OK" , { dialogInterface, i ->
          // "OK" button was clicked
       } )
      .show()
```

Dependency Injection
Dependency injection has you provide any objects required when you instantiate a new object; the new object doesn't need to construct or customize the objects itself.

Dagger 2 is the most popular open-source dependency injection framework for Android and was developed in collaboration between Google and Square.

```
@Module
class AppModule {
  @Provides fun provideSharedPreferences(app: Application):
SharedPreferences {
       return app.getSharedPreferences( "prefs" , Context. MODE_PRIVATE )
  }
}


//--------------------------------------------------------------------------------
------

@Inject lateinit var sharedPreferences: SharedPreferences
```

Singleton

The Singleton Pattern specifies that only a single instance of a class should exist with a global point of access. **Application** class of Android is a good example of Singleton.

Adapter

Adapter Pattern matches interfaces of different classes. That is the most common pattern used in Android. Because we use it to bind RecyclerViews. We have adapter object for them.

```

class MyAdapter( private val data : List<Data>) :
RecyclerView.Adapter<MyViewHolder>() {
   override fun onCreateViewHolder(viewGroup: ViewGroup, i: Int):
MyViewHolder {
       val inflater = LayoutInflater.from(viewGroup. context )
       val view = inflater.inflate(R.layout.row, viewGroup, false )
       return MyViewHolder(view)
  }

   override fun onBindViewHolder(viewHolder: MyViewHolder, i: Int) {
      viewHolder.bind( data [i])
  }

   override fun getItemCount() = data . size
}
```

Facade
The Facade pattern provides a higher-level interface that makes a set of other interfaces easier to use. **Retrofit** helps us implement the Facade pattern; we create an interface to provide API data to client classes like so:

```
interface BooksApi {
  @GET( "books" )
   fun listBooks(): Call<List<Book>>
}
```

Command
Command pattern lets you issue requests without knowing the receiver. **EventBus** is a popular Android framework that supports this pattern in the following manner:

```
class MySpecificEvent { /* Additional fields if needed */ }

eventBus.register( this )

fun onEvent(event: MySpecificEvent) {
   /* Do something */
}

eventBus.post(event)
```

Observer
Observer Pattern is a way of notifying change to a number of classes. The **RxAndroid** framework lets us implement this pattern throughout our app. A sample usage would be:

```
apiService.getData(someData)
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
```

```
.subscribe ( /* an Observer */ )
```

MVVM (Model View ViewModel)
The MVVM pattern is trending upwards in popularity but is still a fairly recent addition to the pattern library. Google recently introduced this pattern as part of its **Architecture Components** library.

✓

Important:

Design Patterns are very important for senior developers. You should be aware of widely used patterns in Android world. I have provided a good answer to this question. I recommend you to read the answer 10 times and repeat it to yourself. This way you are going to master it easily.

112. What is MVP?

The MVP pattern allows separating the presentation layer from the logic so that everything about how the UI works is agnostic from how we represent it on screen.

MVP is not an architecture by itself, it’s only responsible for the presentation layer.

MVP only models the presentation layer, but the rest of layers will still require a good architecture if you want a flexible and scalable App.

An example of a complete architecture could be Clean Architecture, though there are many other options.

In MVP, we have four components: Activity/Fragment, View and Presenter interfaces and PresenterImpl class.

Our Activity/Fragment implements View interface. View interface contains methods related to events to occur on UI.

Presenter interface contains methods regarding to actions that UI may trigger. Presenter interface is implemented by PresenterImpl class.

That PresenterImpl class that implements Presenter interface. PresenterImpl keeps logic related to UI. It determines what will happen on screen.

And PresenterImpl keeps a reference to View, so that it can update Activity/Fragment whenever it needs.

A simple code would be as follows...

View Interface:

```
interface View{
    fun showLoading()
    fun hideLoading()
    fun onLoginSuccess(user: User)
    fun onLoginError(message: String)
}
```

Presenter Interface:

```
interface Presenter{
    fun login(userName:String, password:String)
}
```

PresenterImpl class:

```
class PresenterImpl( val view : View): Presenter{
    override fun login(userName: String, password: String) {

        if (userNameAndPasswordCoorect(userName, password)){
            view .hideLoading()
            view .onLoginSuccess(user)
       } else {
            view .hideLoading()
            view .onLoginError( "Incorrect username or password" )
       }
   }

}
```

And Activity or Fragment:

```
class MVPActivity: AppCompatActivity(), View{

    private val presenter = PresenterImpl( this )
    override fun onLoginSuccess(user: User) {

   }

    override fun onLoginError(message: String) {
   }

}
```

Separating interface from logic in Android is not easy, but the MVP pattern makes it easier to prevent our activities end up degrading into very coupled classes consisting of hundreds or even thousands of lines. In large Apps, it is essential to organize our code well. Otherwise, it becomes impossible to maintain and extend.

Nowadays, there are other alternatives like MVVM, let’s wait and see what happens.

113. What is MVVM?

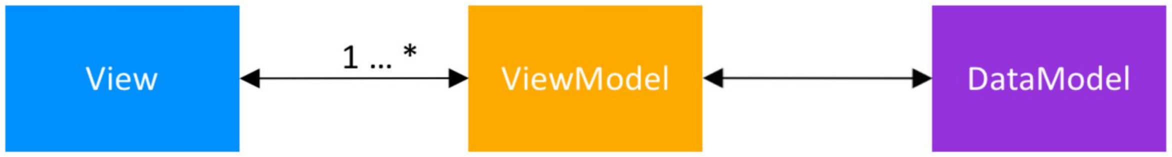
MVVM stands for **Model-View-ViewModel** . MVVM has three components:

The View: That informs the ViewModel about the user's actions.
The ViewModel: Exposes streams of data relevant to the View.
The Model: Abstracts the data source. The ViewModel works with the Model to get and save the data.

Basically, MVVM was proposed by Google with declaration of Architecture Components. Android Studio now provides a template for Fragment+ViewModel.

We are supposed to create a ViewModel for every screen. In the ViewModel we keep all screen related data in LiveData objects. Whenever this data is changed, it automatically triggers our UI components such as Fragments or Activities.

Let's visualize it with a simple diagram:



Every view is connected to a ViewModel. ViewModel keeps screen related logic and all the data that UI requires. It determines what will happen on the UI section. And ViewModel uses data classes while executing its job.

When compared to MVP, MVVM looks quite similar. The only difference is that, ViewModel replaces Presenter class of MVP.

Basic advantage is that, MVP may brings some lifecycle issues on runtime. Because we have to give a Context object and a View object to Presenter to do its job. But when Presenter tries to update UI, context or view object may no longer be alive. That causes some lifecycle problems potentially.

On the other hand, MVVM does not require to get an instance of view or context. Because this is handled by Architecture Components.

Let’s explain MVVM with a sample code.

Here is a simple ViewModel. It keep screen related data in LiveData. LiveData is an observable data holder. Whenever data is changed, it notifies observers.

```
class MyViewModel : ViewModel() {
   private lateinit var users : MutableLiveData<List<User>>

   public fun getUsers(): LiveData<List<User>> {
      if ( users == null ) {
         users = MutableLiveData<List<User>>();
        loadUsers();
     }
      return users ;
  }

   private fun loadUsers() {
      // Do an asynchronous operation to fetch users.
   }
}
```

User object is our Model class but for brevity i am not providing source code for it. It is quite simple to imagine what are the fields of a simple User class.

Now we have ViewModel and its data in LiveData. Now we need observers, i mean screens: Fragments or Activities.

Here is a sample Activity that uses ViewModel and observes data inside it.

```
class MyActivity: AppCompatActivity() {
   public override fun onCreate( savedInstanceState: Bundle) {
       // Create a ViewModel the first time the system calls an activity's onCreate()
method.
      // Re-created activities receive the same MyViewModel instance created by
the first activity.

       val model = ViewModelProviders.of( this ).get(MyViewModel. class );
      model.getUsers().observe( this , users -> {
          // update UI
       } );
  }
}
```

Architecture Components provides us `ViewModelProviders` class to get an instance of ViewModel. We just provide class name of ViewModel to `ViewModelProviders` .

Then as you can see, we observe to users in ViewModel with line of `model.getUsers().observe()` . Whenever user data is changed, this callback is triggered immediately. That is how we keep screen related data in ViewModel and observe it via Observer Pattern. This pattern is simply called as Hollywood Principle.

To sum up, View is an activity or Fragment that observes data. ViewModel is a special class that holds UI data and logic. Model is simple data classes that we know.

MVVM is highly trending, i am aware of that and i am a big fan of it.

114. What are the common layers of responsibility in a well structured Android app?

In a well structured Android app, common layers would be **Presentation** , **Data** and **Domain** layers.

This structure is commonly used by Google's sample projects on Github.

Presentation is the layer that we keep everything related to UI. Activites, Fragments, ViewModels, custom UI components etc.

Data is the layer that fetches data. It may fetch from different sources such as network, web service, local db, file storage, SharedPrefs or cache or anywhere else. It doesn't matter. The only thing matters is that data layer is the only where that fetches data. It's the single source of truth all the app data.

Data layer may contain Retrofit related staff, some model classes with Retrofit annotations and some mapper classes if needed.

Domain is the layer that keeps business logic of our app. It has some classes called Use Case. Every use case contains an algorithm of a user scenario in the app. For instance, "Fetch shops in coordinates of X" is a scenario and it has a corresponding use case class in domain section.

Domain section is not allowed to reach other layers: Presentation or Data. It can't reference a class or an object in other layers. It's also not allowed to use any third party libraries. It must be pure Kotlin or Java. Main idea behind this is it should be testable.

Domain data should be tested without any mocks or dependencies. That is why we try to reduce dependencies of this layer as much as possible.

Actually this architecture is named as Clean Architecture and proposed by Uncle Bob. His real name is Robert C. Martin, a famous guy in software world. Recently, Google also recommends this architecture or its derivatives.

When it comes to the architecture, i try to remember things that i have learned from famous video of Uncle Bob: Architecture, The Lost Years. Basically he says that we have wasted many years to find the perfect architecture but only a few us were lucky enough to find out that there is no perfect architecture for every application.

The point is that most of the time commonly accepted solutions work well. We don't have to reinvent the wheel but should be able to create elegant solutions to new problems.

Important:

Ability of architecting a good architecture for your app is the essential difference between an expert developer and the others. This kind of questions are very general and it has many different approaches.

I have provided an answer with a very concrete structure and it is well enough for you to prove your advanced skills.

115. What are SOLID principles? Can you given an example of each in Kotlin?

SOLID is an acronym for five design principles. These are:

Single Responsibility Principle: A class or method should have only one responsibility.

Open-Closed Principle: A class should be open to be extended, closed to be changed.

Liskov's Substitution Principle: Derived classes must be substitutable for their base classes.

Interface Segregation: Make fine grained interfaces that are client specific.

Dependency Inversion: Depend on abstractions not on concretions.

When we bring the first letters together, it makes: **SOLID** . Let me explain them one by one.

Single Responsibility Principle

Single Responsibility Principle (SRP) states that a class or a method should only be doing one thing and shouldn't be any doing anything related. A class should have only one reason to change. If a class or a method is doing more than one thing, then it must be splitted.

For instance suppose that we have a `Person` class to keep user info such as name and email. Since `Person` is just a model class, it can't have logic of email validation. Let's see a badly written `Person` class:

```
class Person( val email : String){
    init {
       validateEmail( email )
   }

    fun validateEmail(email: String){
        // Email validation code
    }

}
```

Here we see that Person class has a function `validateEmail( email )` to validate emails. Since Person's only responsibility is to keep person's data, it can not have a function to validate it. Because it is yet another responsibility. It violates SRP.

We need to split it and have 2 classes for two responsibilities: one for holding email data, one for validating it. Let's name second class as .

```
class Person( val email : String){

    init {
       EmailValidator.validateEmail( email )
   }

    // We have removed validateEmail() function
}

// We have created this class for email validation
class EmailValidator{

    companion object {

        fun validateEmail(email: String){
            // Email validation code
        }
   }
}
```

Let’s pass to the most famous one.

Open Closed Principle

What it means though is that we should strive to **write code that doesn’t have to be changed every time the requirements change** . How we do that can differ a bit depending on the context, such as our programming language. When using Kotlin or some other statically typed language the solution often involves inheritance and polymorphism .

Let’s assume we are writing a class for calculating area of rectangle shapes. Here we have a `Rectangle` class and an `AreaCalculator` class that sums area of incoming rectangles with `calculateAreas()` .

```
class Rectangle( val width : Double, val height : Double) {

   fun area() = width * height

}

class AreaCalculator{

   fun calculateAreas(shapes: Array<Rectangle>): Double{
      var totalArea = 0.0

      for (rectangle in shapes){
        totalArea += rectangle.area()
     }
      return totalArea
  }
}
```

At first glance everything seems fine but when we are requested to also calculate area of triangles with rectangles things get messy.

An inexperienced developer would go to `calculateAreas(shapes: Array<Rectangle>)` method and add some code to calculate areas of triangles using if block. That means you have changed the code. But you forgot, our code should be closed to be changed, open to be changed.

Let’s see it on codes:

```
class AreaCalculator{

   fun calculateAreas(shapes: Array<Rectangle>, triangles: Array<Triangle>):
Double{
       var totalArea = 0.0

       for (rectangle in shapes){
          totalArea += rectangle.area()
       }

       for (triangle in triangles){
          totalArea += triangle.area()
       }

       return totalArea
   }
}
```

Liskov Substitution Principle (LSP)

It means child classes should never break the parent class’ type definitions.

As simple as that, a subclass should override the parent class’ methods in a way that does not break functionality from a client’s point of view. Here is a simple example to demonstrate the concept.

Let’s consider an `Animal` parent class.

```
interface Animal {
    fun makeNoise()
}
```

Now let’s consider the `Cat` and `Dog` classes which extends Animal.

```
class Dog : Animal {
    override fun makeNoise() {
        println ( "bow wow" )
    }
}

class Cat : Animal {
    override fun makeNoise() {
        println ( "meow meow" )
    }
}
```

Now, wherever in our code we were using Animal class object we must be able to replace it with the `Dog` or `Cat` without exploding our code. What do we mean here is the child class should not implement code such that if it is replaced by the parent class then the application will stop running. For example if the following class is replace by `Animal` then our app will crash.

```
internal class DeafDog : Animal {
    override fun makeNoise() {
        throw RuntimeException( "I can't make noise" )
    }
}
```

If we take Android example then we should write the custom RecyclerView adapter class in such a way that it still works with

RecyclerView. We should not write something which will lead the RecyclerView to misbehave.

Interface Segregation Principle

This principle suggests that “many client specific interfaces are better than one general interface”. This is the first principle which is applied on interface, all the above three principles applies on classes. Let’s take following example to understand this principle.

In other words ake fine grained interfaces that are client specific. The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

Let me give an example from Android. As you know, the Android View class is the root superclass for all Android views.

```
interface OnClickListener {
    fun onClick(view: View);
    fun onLongClick(view: View);
    fun onTouch(view: View, event: MotionEvent);
}
```

As you can see, this interface contains three different methods, assuming that we wanna get click from a button:

```
// Violation of Interface segregation principle
val button: Button = findViewById(R.id.button);
button.setOnClickListener( View.OnClickListener {
    override fun void onClick(view: View) {
        // TODO: do some stuff...

    }

    override fun void onLongClick(view: View) {
        // we don't need to it
    }

    override fun void onTouch(view: View, event: MotionEvent) {
        // we don't need to it
    }
} );
```

The interface is too fat because it’s forcing to implement all the methods, even if it doesn’t not need them. Let’s trying to fix them using **ISP** :

```
// Fix of Interface Segregation principle
interface OnClickListener {
    fun onClick(v: View)
}

interface OnLongClickListener {
    fun onLongClick(v: View)
}

interface OnTouchListener {
    fun onTouch(v: View, event: MotionEvent)
}
```

Now we can use the interface without implement some messy methods. Because it satisfies **Interface Segregation Principle** .

Dependency Inversion Principle

It means high-level modules should not depend on low-level modules. Both should depend on abstractions.

Let me explain with a simple code:

```
// violation of Dependency inversion principle
// Program.java
class Program {

   fun   work() {
      // ....code
   }
}


// Engineer class
class Engineer ( var program: Program){


   fun manage() {
     program.work()
  }
}
```

This code seems pretty familiar to us because most of the time we write this kind of classes.

The problem with the above code is that it breaks the Dependency Inversion Principle : <u>High-level modules should not depend on low-level modules. Both should depend on abstractions.</u>

We have the `Engineer` class which is a high level class, and the low level class called `Program` . And `Engineer` depends on `Program` .

Solution is to define interfaces for `Program` and let `Engineer` to depend on that interface not concrete class.

```

// Dependency Inversion Principle - Good example
interface IProgram {
   fun work()
}

class Program : IProgram {
   override fun work() {
      // ....code
   }
}

class SuperProgram : IProgram {
    override fun work() {
        //....code
    }
}

class Engineer( var program : IProgram) {

    fun manage() {
        program .work()
   }
}
```

Here we see how better it is.

Main idea behind design principles is that we make our code open to improvements and lots of developers can work it on at the same time. Robert C. Martin has brought these principles together and made us a big fever. Design principles are more abstract compared to design patterns and must be our guide while making decisions.

Important:

OK, this is pretty long answer but it should be. Because principles and patterns are core topics for expert developers. If you are claiming that you are an expert developer, the first thing that you are going to be asked is a pattern or principle.

This is a very generic and huge topic. But we have summarized it all well with nice explanations and code examples.

We recommend you, again, to read the answer 10 times and then you will be able to explain it easily. You should also try to write code on whiteboard or a piece of paper.

Assume that you are asked a principle or pattern. You have explained its meaning well, then your interviewer is probably going to ask you how to implement it. So, you should be ready to show some code on whiteboard.

You don’t need an extra resource to answer this question. This answer is pretty enough for a hard interview.

116. What is Functional Programming and Reactive Programming?

Functional programming is a programming paradigm where we model everything as a result of a function that avoids changing state and mutating data.

On the other side, **Reactive programming** is the practice of programming with asynchronous data streams or event streams.

An event stream can be anything like keyboard inputs, button taps, gestures, GPS location updates, accelerometer, and iBeacon. You can listen to a stream and react to it accordingly.

Let’s compare them on code examples. Functional programming is quite straightforward.

```
val numbers = arrayOf ( 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 )
val numbersLessThanFive = numbers. filter { it < 5 }
```

But when it comes to Reactive programming, we program with event streams. For instance:

```
var twoDimensionalArray = arrayOf ( arrayOf ( 1 , 2 ), arrayOf ( 3 , 4 ), arrayOf
( 5 , 6 ))

val flatArray = twoDimensionalArray. flatMap {
  it . map { x * 2 }
}
print (flatArray)

Output : [ 2 , 4 , 6 , 8 , 10 , 12 ]
```

In Android, we have RxAndroid library for reactive programming. It is flexible to use Rx with Retrofit.

CHAPTER 16

CLEAN CODE, CODING CONVENTIONS, BEST PRACTICES

117. What is clean code? How do you write clean code?

Actually, Clean Coding is famous thanks to Uncle Bob. He has a book with same name: Clean Code. Most of the developers have learned clean coding from him.

Clean code is simply writing simple, human readable and good code. Uncle Bob says “ bad code smells. ”.

I generally pay attention to such things:

- Methods should be small. Uncle Bob says a method should be 3-4 lines but i think that in real world 10-15 lines is enough. If a method is more than 10-15 lines, i think of how to split it.

- Naming conventions is very important. Method names, variable names and class names should be understandable. It must propose its intention. It doesn’t matter how long they are, but must be named comprehensively.

- Indentation is yet another topic. But i don’t have to pay effort for this. Just Cmd+Shift+F and Android Studio formats code for me.

- Code must be testable. This is a large topic, but in short, try to take things as parameters. Use Dependency Injection if possible. Reduce dependencies. Stay away from Singletons and static variables.

- Methods should do only one job. Actually this is Single Responsibility Principle. Code also should satisfy SOLID Design Principles.

- Always refactor. I generally review codes that i have written previously and think of how to improve it.
- Don’t repeat yourself. This is a famous principle. Repeating codes should be one method and used when required.

- Be careful when using external libraries. Just use the libraries that have proven themself. For small tasks, i try to implement it myself rather than using a library.

- Restrict method parameter count to three. If more, i write a request class for that method.

Actually this is a huge topic but most of the time, i care about such things. Of course this list can be extended.

118. Which coding conventions you follow when developing an Android app?

Google provides lots of conventions for us. Let me mention just a few of them.

- Don't ignore exceptions. We must handle every exception in our code in some principled way.

- Don't catch generic exception.

- We don't use finalizers. There are no guarantees as to when a finalizer will be called.

- Fields should be defined at the top of the file and they should follow the naming rules listed below.

    - Private, non-static field names start with m.
    - Private, static field names start with s.
    - Other fields start with a lowercase letter.
    - Static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

- Class member ordering should use the following order:

    - Constants
    - Fields
    - Constructors
    - Override methods and callbacks (public or private)
    - Public methods
    - Private methods
    - Inner classes or interfaces

- When programming for Android, it is quite common to define methods that take a Context. If we are writing a method like this, then the Context must be the first parameter.

- Callback interfaces that should always be the last parameter of s method.

- When an Activity or Fragment expects arguments, it should provide a public static method that facilitates the creation of the relevant Intent or Fragment. This method is called as `newIntent()` method.

- Resource IDs and names are written in **lowercase_underscore** .

Important:

Google provides lots of coding conventions to Android developers. We are supposed to follow these rules.

You can see the full list from this url: https://source.android.com/setup/contribute/code-style

119. What are your best practices as an Android developer?

I have created a lot of Android projects and crafted robust architectures. So far i have learned some best practices. Let me summarize a few of them.

- I always try to follow recommended architecture. Google recommends us to split Data and Presentation layers. I also add a Domain layer.

- I always use Fragments. I never design my screens on activities.

- I always try to satisfy SOLID design principles and design patterns.

- I always create a TAG string and a `newIntent()` or `newInstance()` method for every Activity and Fragment.

- Unit testing is a must for me. I always include automated tests. I know TDD is hard, so most of the time i don't do TDD but i always add tests for some edge cases and essential scenarios.

- I use Proguard in my release version. This will remove all unused code, which will reduce APK size.

- I always run my code on different emulators for testing different Android versions and different screen sizes. For instance a 4 inch phone, a normal phone, 7 inches, 9 inches and 10 inches tablets.

- I never hard-code strings. Strings.xml file for this. It also provides i18n for different languages.

- I use shapes, selectors and webp format instead of images as much as possible. This further reduces APK size.

- I avoid deep levels in layouts. A deep hierarchy of views makes our UI slow, not to mention making it harder to manage your layouts. I use Constraint Layout for this.

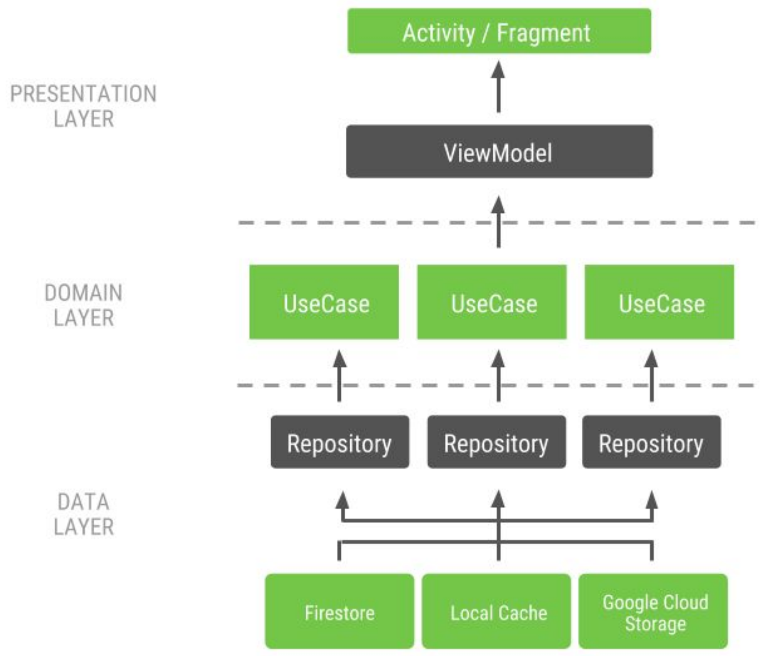
- I perform file, network and database operations off the UI thread. Also every heavy calculation should be done off the UI thread.

120. What is Clean Architecture?

Clean Architecture has been proposed by Uncle Bob. The app is divided into a three layer structure:

- Presentation layer (Views and ViewModels)
- Domain layer (use cases)
- Data layer (repositories, user manager)

The presentation layer cannot talk to the data layer directly. A ViewModel can only get to a repository through one or more use cases. This limitation ensures independence and testability. It also brings a nice opportunity to jump to a background thread: all use cases are executed in the background guaranteeing that no data access happens on the UI thread.



Presentation layer: Views + ViewModels + Data Binding

ViewModels provide data to the views via LiveData. The actual UI calls are done with Data Binding, relieving the activities and fragments from boilerplate.

We deal with events using an event wrapper, modeled as part of the UI’s state.

Domain layer : UseCases
The domain layer revolves around the UseCase class, which went through a lot of iterations. In order to avoid callback hell we decided to use LiveData to expose the results of the UseCases.

By default use cases execute on a **DefaultScheduler** (a Kotlin object) which can later be modified from tests to run synchronously. We found this easier than dealing with a custom Dagger graph to inject a synchronous scheduler.

Data layer
The app dealt with 3 types of data, considering how often they change:

- Static data that never changes: map, agenda, etc.
- Data that changes 0–10 times per day: schedule data (sessions, speakers, tags, etc.)
- Data that changes constantly even without user interaction: reservations and session starring

121. You are supposed to review someone’s code. How do you judge it?

The first thing i care about is commit message. We developers use source control for collaborating. We use Git for it. So every commit on Git must be clear enough to expose its intent. Commit messages must be clean and neat.

Secondly, i pay attention to version code, fix numbers, Git tag etc… Such things are also important. If there is no problem, then i look at the code.

If a commit is too large, i deny it. Commits must be small. Google recommends maximum of 30 lines of code per commit but we can extend it to 50-60 lines at our company.

Looking at code, i ask these questions to myself:

- Is code formatted?
- Variable and method names are clear enough?
- SOLID design principles are violated?
- Is there a better way of implementing this?
- Is the code readable, short and clean?

I judge the code against to these criterion.

Lastly, is unit tests are included? I definitely reject commits without unit tests.

CHAPTER 17

ARCHITECTURE COMPONENTS

121. What are Architecture Components?

Android architecture components are a collection of libraries that help you design robust, testable, and maintainable apps. They are a part of Jetpack.

It provides us classes for managing our UI component lifecycle and handling data persistence. Basically, it provides four components.

Lifecycle: We use Lifecycle to manage our app's lifecycle with ease. New lifecycle-aware components help us manage our activity and fragment lifecycles. Survive configuration changes, avoid memory leaks and easily load data into our UI.

LiveData: We use LiveData to build data objects that notify views when the underlying database changes.

ViewModel: ViewModel Stores UI-related data that isn't destroyed on app rotations.

Room: Room is an a SQLite object mapping library. We use it to avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.

122. How does ViewModel survives configuration changes?

ViewModel objects are scoped to the Lifecycle passed to the ViewModelProvider when getting the ViewModel . The ViewModel remains in memory until the Lifecycle it's scoped to goes away permanently: in the case of an activity, when it finishes, while in the case of a fragment, when it's detached.

We should never use constructor to get an instance of a ViewModel . Because that defeats the ViewModel 's purpose of surviving configuration changes. Android provides us ViewModelProvider factory class for instantiating a ViewModel.

ViewModelProvider uses a HashMap in its internal implementation and provides us same instance of ViewModel everytime.

A typical example would be:

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super .onCreate(savedInstanceState)
        var model = ViewModelProviders.of( this ).get(MyViewModel:: class . java
)
       model.getUsers().observe( this , users -> {
           // update UI
        } )
   }

}
```

123. What is advantage of LiveData?

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

LiveData considers an observer, which is represented by the Observer class, to be in an active state if its lifecycle is in the **STARTED** or **RESUMED** state. LiveData only notifies active observers about updates. Inactive observers registered to watch LiveData objects aren't notified about changes.

That is the advantage of LiveData over a regular observable.

124. Can you explain relationship between LifeCyle Owner and LifeCycle Observer?

LifecycleOwner is a single method interface that denotes that the class has a Lifecycle. It has one method, *getLifecycle()* , which must be implemented by the class.

This interface abstracts the ownership of a Lifecycle from individual classes, such as Fragment and AppCompatActivity , and allows writing components that work with them. Any custom application class can implement the LifecycleOwner interface.

Components that implement **LifecycleObserver** work seamlessly with components that implement **LifecycleOwner** because an owner can provide a lifecycle, which an observer can register to watch.

125. What do you know about ROOM?

Room is an a SQLite object mapping library . The Room persistence library provides an abstraction layer over SQLite to allow for more robust database access while harnessing the full power of SQLite.

The library helps us create a cache of our app's data on a device that's running our app. This cache, which serves as our app's single source of truth, allows users to view a consistent copy of key information within our app, regardless of whether users have an internet connection.

We should use it to avoid boilerplate code and easily convert SQLite table data to Java objects. Room provides compile time checks of SQLite statements and can return RxJava, Flowable and LiveData observables.

CHAPTER 18

THIRD PARTY LIBRARIES

126. What is RxAndroid? How can we benefit from it?

RxAndroid is a reactive programming library for Android. **Reactive Programming** is basically event-based asynchronous programming. Everything we see is an asynchronous data stream, which can be observed and an action will be taken place when it emits values.

We can create data stream out of anything; variable changes, click events, http calls, data storage and errors. When it says asynchronous, that means every code module runs on its own thread thus executing multiple code blocks simultaneously.

A typical example of RxAndroid would be:

```
myObservable // observable will be subscribed on i/o thread
        .subscribeOn(Schedulers.io())
       .observeOn(AndroidSchedulers.mainThread())
       .map { /* this will be called on main thread... */ }
        .doOnNext { /* ...and everything below until next observeOn */ }
        .observeOn(Schedulers.io())
       .subscribe { /* this will be called on i/o thread */ }
```

Rx also provide RxKotlin, which is a wrapper of RxJava. We can safely use it.

127. What is Dependency Injection? How do you implement it in your project?

Dependency injection is a very powerful technique, where you relay the task of providing object with its' dependencies on instances of other objects to a separate class.

This allows for fewer constructors, setters, factories and builders as all those functions are taken care of by the DI framework that we use.

The pattern seeks to establish a level of abstraction via a public interface and to remove dependencies on components by supplying a 'plugin' architecture. This means that the individual components are tied together by the architecture rather than being linked together themselves. The responsibility for object creation and linking is removed from the objects themselves and moved to a factory.

Another example of DI usage is unit-testing - it allows to conveniently inject all needed dependencies and keep the amount of written code at a lower level.

In Android, we use **Dagger** library for dependency injection. Dagger is a fully static, compile-time dependency injection framework for both Java and Android. It is an adaptation of an earlier version created by Square and now maintained by Google.

Dagger 2 uses annotation processing not reflection. That is why it is so fast and successful. In order to use it, we add the dependency to our Gradle file and write a component interface and a module class for ith. Then we use it with `@Inject` annotation and it provides us the object we want.

For instance;

```
class MyActivity : AppCompatActivity() {

  @Inject lateinit var navigator : Navigator
  @Inject lateinit var moviesAdapter : MoviesAdapter

}
```

128. What is advantage of Dagger 2 against other DI libraries?

Dagger 2 works on **Annotation processor** . So all the code generation is traceable at the compile time. Hence we have no performance overhead and its errors are highly traceable.

That is the main advantage of Dagger 2 over other dependency injection frameworks. That is why it’s so fast and successful.

Annotation processors are the code generators that eliminate the boilerplate code, by creating code for us during the compile time. Since it’s compile time, there is no overhead in the performance.

Dagger provides three different usage of Inject annotations: **Field Injection** , **Constructor Injection** , **Method Injection** .

```
class MyActivity {
   /**
   * Explaining different usage of Inject annotations of dagger
   **/

  //Feild injection
   @Inject Allies allies;

   //Constructor injection
   @Inject
   public MyActivity()
  {
      //do something..
   }

   //Method injection
   @Inject
   private void someMethod()
  {
      //do something..
   }

}
```

129. How does ButterKnife works? Can you explain its working mechanism?

ButterKnife works thanks to **Annotation Processing** . Annotation processing is a tool build in *javac* for scanning and processing annotations at compile time.

When we compile our Android project ButterKnife Annotations Processor *process()* method is executed doing the following:

- First, it scans all java classes looking for ButterKnife annotations: *@InjectView* , *@OnClick* , etc.

- When it find a class with any of these annotations it creates a file called: *<className>$$ViewInjector.java*

- This new *ViewInjector* class contains all the necessary methods to handle annotation logic: *findViewById()* , *view.setOnClickListener()* , etc.

- Finally, during execution, when we call *ButterKnife.inject(this)* each ViewInjector *inject()* method is called.

130. What are commonly used libraries in a typical Android app?

I pay attention to use commonly used libraries because they give us power and flexibility. To name just a few of them:

RxJava: reactive programming for Java and Android. Also supports Kotlin.

Dagger2: The most popular Dependency Injection framework. Maintained by Google.

jUnit 4: Official test framework of Java and Android.

Mockito: Mockito allows you to create and configure mock objects. The most popular mock framework which can be used in conjunction with JUnit.

Robolectric: A unit test framework that de-fangs the Android SDK jar so we can test-drive the development of our Android app.

Espresso: A library to write concise, beautiful, and reliable Android UI tests.

Retrofit: A must use tool for connecting to a web service.

Fresco: Fresco is a powerful system for displaying images in Android applications.
Fabric - Crashlytics: The most powerful, yet lightest weight crash reporting solution.

Google Analytics: Helps us to measure what is happening in our app.

Architecture Components: Helps us design more robust, testable and maintainable apps.

Glide: For downloading and caching images.

EventBus: Simplifies communication between Activities, Fragments, Threads, Services, etc. Less code, better quality.

LeakCanary: We use it to find memory leaks.

These technologies are kinda toolbelt for a well grounded Android developer. I efort to use these latest technologies effectively.

131. What do you know about Version Control? Have you ever used Git?

Version control systems are a category of software tools that help a software team manage changes to source code over time.

Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

The most popular version control system is Git. I have been using it for years.

Git is a distributed version control system. Every dev has a working copy of the code and full change history on their local machine, by Linus Torvalds.

Using Git, we are able to collaborate on the same project, version our code, commit and push to repository.

Let me briefly explain its working mechanism...

When working with Git, we have a local repository and a remote repository. We pull the source code from remote repository, make changes and commit code to local repository then we push to the remote repository. Then other teammates can pull the changes that we made on project to their locale machines.

Git provides us ability of creating branches. So that we can work on different versions of the project without mixing up the codes.

Let me show you most used Git commands.

To create an empty git repo or reinitialize an existing one

```
$ git init
```

To c lone the foo repo into a new directory called foo:

```
$ git clone https://github.com/<username>/foo.git foo
```

To see a list of the repositories (remotes) whose branches we track:

```
$ git remote -v
```

To create and checkout a new branch:

```
$ git checkout -b <new_branch_name>
```

To see what files have changed:

```
$ git status
```

To commit a change:

```
$ git commit -m "Updated README"
```

To push a local branch for the first time:

```
$ git push --set-upstream origin <branch>
$ git push
```

132. What is Continuous Integration?

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

By integrating regularly, we can detect errors quickly, and locate them more easily.

We use **Jenkins** as our CI server. We have a bunch of unit and integration tests. Whenever we submit a build request, Jenkins runs all the automated tests and emails us a report.

Martin fowler says that “Continuous Integration doesn’t get rid of bugs, but it does make them dramatically easier to find and remove.”

CHAPTER 19

UNIT TESTS & INSTRUMENTATION TESTS

133. Uncle Bob has a famous acronym for tests: FIRST. Do you know what does it mean?

First is an acronym. It stands for Fast, Independent, Repeatable, Self-Validating and Timely.

Let me explain them one by one.

Fast
Unit tests should be fast otherwise they will slow down our development/deployment time and will take longer time to pass or fail.

Independent
Never ever write tests which depend on other test cases. We should be able to run any one test at any time, in any order.

By making independent tests, it’s easy to keep our tests focused only on a small amount of behavior.

Repeatable
A repeatable test is one that produces the same results each time you run it. To accomplish repeatable tests, we must isolate them from anything in the external environment not under our direct control.

Self-validating
Tests must be self-validating means – each test must be able to determine that the output is expected or not. It must determine it is failed or pass. There must be no manual interpretation of results.

Timely
Practically, we can write unit tests at any time. We can wait upto code is production ready or we’re better off focusing on writing unit

tests in a timely fashion.

134. What is difference between Unit Test and Instrumentation Test?

Unit tests run tests without a device or an emulator attached. Thay are referred to as “local tests” or “local unit tests”.

Instrumentation tests run on a device or an emulator. In the background, our app will be installed and then a testing app will also be installed which will control our app, launching it and running UI tests as needed.

Unit tests cannot test the UI for our app without mocking objects such as an Activity.

Instrumentation tests can be used to test none UI logic as well. They are especially useful when we need to test code that has a dependency on a context.

JUnit is the only thing we need for unit testing but we need Espresso for instrumented tests.

Lastly, unit test focuses on objects behaviour but instrumented tests are generally for testing integration between components such as Activity or Fragments etc…

135. What is TDD (Test Driven Development) ?

Test-driven development (TDD) (Beck 2003; Astels 2003), is an approach to development which you write a test before you write just enough production code to fulfill that test and refactoring.

TDD has a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

The following sequence of steps is generally followed:

- Add a test
- Run all tests and see if the new one fails
- Write some code
- Run tests
- Refactor code
- Repeat

TDD forces us to think through our requirements or design before we write our functional code. TDD is both an important agile requirement and agile design technique. The goal of TDD is to write clean code that works.

Actually TDD is very hard to follow. Recently there is a rumor that TDD is dying because almost no one doing it. You need to practice well to master TDD. At that point you are gonna benefit it too much but until that point you are going to suffer.

I personally support TDD but i think we need much more time, a better deadline for project delivery if we are following TDD.

136. How can you test a timeout? Write a simple unit test method for it.

There are couple solutions how we can test a timeout.

The first solution is to use timeout parameter in @Test annotation. The time is in milliseconds.

```
@Test ( timeout= 100 )
public void myTestFunction() { ... }
```

The other solution is to use @Rule annotation. This is really handy if we want to setup timeout for the whole test case for every test in the file.

```
class MyTests {

  @Rule
   public Timeout globalTimeout = Timeout.seconds( 10 );

   @Test
   public void test1() throws Exception {
     ...
  }

   @Test
   public void test2() throws Exception {
     ...
  }
}
```

If the timeout is reached, it will throw TimeoutException .

137. How can you test an Exception? Write a simple unit test method for it.

Actually i know three different ways of this. The first is to use @Test (expected)
annotation.

```
@Test ( expected = IndexOutOfBoundsException:: class )
fun empty() {
  ArrayList<Any>()[ 0 ]
}
```

That is good for short test methods. The above test will pass if any code in the method throws *IndexOutOfBoundsException* .

Secondly, for longer test methods it's recommended to use the *ExpectedException* .

```
@Rule
var thrown = ExpectedException.none()

@Test
@Throws (IndexOutOfBoundsException:: class )
fun shouldTestExceptionMessage() {
   val list = ArrayList<Any>()

   thrown .expect(IndexOutOfBoundsException:: class . java )
   thrown .expectMessage( "Index: 0, Size: 0" )
  list[ 0 ] // execution will never get past this line
}
```

This rule lets you indicate not only what exception we are expecting, but also the exception message we are expecting.

Lastly, we may use **Try/Catch** idiom for testing purposes. A simple explanation would be:

```
@Test
fun testExceptionMessage() {
    try {
      ArrayList<Any>()[ 0 ]
      fail( "Expected an IndexOutOfBoundsException to be thrown" )
   } catch (anIndexOutOfBoundsException: IndexOutOfBoundsException) {
      assertThat(anIndexOutOfBoundsException. message , `is`( "Index: 0,
Size: 0" ))
   }

}
```

138. We want to test business logic of our app but it’s implemented in an activity or fragment. How do you solve it?

First of all we need to remove business logic from our UI components. There shouldn’t be any logic in an Activity or Fragment.

I hold UI logic and UI data in **ViewModels** . So we may write a ViewModel class for every Fragment in our app and move UI logic to there.

Additionally, we need an extra layer for our business logic. This layer may be called as domain layer.

Clean Architecture recommends us to divide our project into three layers : Presentation, Domain and Data.

We keep UI related classes in **Presentation** . These are Activity, Fragment or ViewModel. There shouldn’t be any business logic here. Only UI related things are allowed.

We use data related things in **Data** layer. Retrofit, ROOM, repositories etc…

And we have **Domain** layer for business logic. We are not allowed to reach other layers from this layer. It’s only pure Kotlin or Java.

Since we have sealed our business logic in domain layer, we can easily test our business logic simply using only JUnit. We don’t need any real device or emulator.

That is also satisfies **Separation of Concerns** principle.

139. How can you effectively deal with legacy code?

Michael Feathers has written a famous book for this : **Working Effectively with Legacy Code** . We all need to read this.

He recommends a few steps. Let me summarize them one by one.

1. We should write characterization tests
Our goal isn't to figure out what the system should do but rather what it actually does, hence the name characterization. We just there to observe and categorize.

2. We should bring in tooling to help our testing and characterization efforts
Writing characterization tests is great in principle. And it can work well in practice when a couple of things are true:

- We have a relatively isolated bit of functionality, like the aforementioned, hypothetical *Add()* method.
- The scope of our change is relatively limited and narrow.

3. We should keep changes to a minimum and boy-scout as you go
We should avoid changing anything we don't need to, but when we have to change something, we should try also to improve it a little. After all, we're probably not done with this codebase after one single change.

4. We should work gradually toward modularity and better design
Well, because if we make the codebase modular enough, we can start to migrate parts of it to newer platforms, or modernize it with language/framework upgrades.

140. What is advantage of automated testing? Why do we write it?

Automated testing ensures that you haven’t broken any functionality previously implemented. We constantly keep adding new code or making changes to our codebase. Hence we should be sure that we haven’t broken any functionality that is currently working. Automated tests gives us confidence.

Automated testing cuts down the time to run repetitive tests from days to just a few hours. After all less time spent equals smaller development bill to foot.

Additionally, we don’t need to keep a dedicated manual testing team to handle QA. One automated testing engineer can have us covered at all times.

141. How do you structure your unit test methods?

I generally use **Given-When-Then** structure for my tests.

We define variables in **Given** section.
When section is where the action happens. Methods to be tested are called here.
Then section is where we check result and decide pass or fail.

This approach is recommended in **Clean Code** book of Uncle Bob.
A simple example would be as follows:

```
@Test
fun getName_returnsName() {
    // GIVEN
    val expectedName = "linda"
    testSubject = Customer(expectedName)

    // WHEN
    val actualName = testSubject.getName()

    // THEN
    assertTrue(expectedName == actualName)
}
```