# ANDROID INTERVIEW QUESTION PART 2

## 1) What is Android Architecture?

The main components of the Android architecture are the following:-

- Applications
- Application Framework
- Android Runtime
- Platform Libraries
- Linux Kernel

## 2) What are the core building blocks of an Android application?

*The basic components are as follows:*

**Activity**: An activity is a subclass of the "ContextThemeWrapper" class. Since almost all activities interact directly with the user, it is often helpful to think of an activity as the screen for a particular action, such as logging in or taking a picture.

**View**: The view is everything you can see on the screen of the app — think of the individual UI elements like buttons, labels, and text fields.

**Intent**: The main purpose of intent is to invoke individual components. Common uses include starting the service, launching activities, displaying a list of contacts, dialing a phone number, or displaying a web page.

**Service**: A service is a background process that can either be local or remote. Local services may be accessed from within the application while remote services are intended to be used by other applications running on the same device.

**Content Provider**: Content providers share data between applications.

**Fragment**: Fragments are best thought of as parts of an activity — you can display more than one fragment on the screen at the same time.

**Android Manifest**: The AndroidManifest.xml file provides essential information about the app required for it to run on the Android operating system. All Android apps have this file in their root directory.

### 2) What is activity in Android?

Activities are basically containers or windows to the user interface.

### 3) What is Context in Android?

*Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents, etc.*

- It allows us to access resources.
- It allows us to interact with other Android components by sending messages.
- It gives you information about your app environment.
  Firstly, let's look at 3 most used functions for retrieving the Context:
- **getContext()** — returns the Context which is linked to the Activity from which is called,
- **getApplicationContext()** — returns the Context which is linked to the Application which holds all activities running inside it,
- **getBaseContext()** — is related to ContextWrapper, which is created around existing Context and lets us change its behavior. With *getBaseContext()* we can fetch the existing Context inside ContextWrapper class.

### 4) What is the intent?

It is a kind of message or information that is passed to the components. It is used to launch an activity, display a web page, send SMS, send email, etc. There are two types of intents in Android:

1. **Implicit Intent** (*Implicit intent is when you call the system default intent like send email, send SMS, dial a number*)
2. **Explicit Intent** (*Explicit intent is when you call an application activity from another activity of the same application.*)

### 5) What is the Lifecycle of an Activity?

- ***OnCreate():*** This is when the view is first created. This is normally where we create views, get data from bundles, etc.
- ***OnStart():*** Called when the activity is becoming visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
- ***OnResume():*** Called when the activity will start interacting with the user. At this point, your activity is at the top of the activity stack, with user input going to it.
- ***OnPause():*** Called as part of the activity lifecycle when an activity is going into the background, but has not (yet) been killed.
- ***OnStop():*** Called when you are no longer visible to the user.
- ***OnRestart():*** Called after your activity has been stopped, prior to it being started again
- ***OnDestroy():*** Called when the activity is finishing

### 6) What is the difference between onCreate() and onStart()?

- *The* ***onCreate()*** method is called once during the Activity lifecycle, either when the application starts, or when the Activity has been destroyed and then recreated, for example during a configuration change.
- *The* ***onStart()*** method is called whenever the Activity becomes visible to the user, typically after onCreate() or onRestart().

### 7) What is the Scenario in which only onDestroy is called for an activity without onPause() and onStop()?

If **finish()** is called in the OnCreate method of an activity, the system will invoke onDestroy() method directly.

### 8) What is the activity A to activity B lifecycle in Android?

Let's Say Activity **A is starting Activity B by Calling StartActivity(Intent)** method then the lifecycle call be like this:-

- A onCreate()
- A onStart()
- A onResume()

**Now Button click for startActivity(intent)**

- A onPause()
- B onCreate()
- B onStart()
- B onResume()
- A onStop()

**….. If you press the back button from Activity B then the lifeCycle call will be …..**

- B onPause()
- A onRestart()
- A onStart()
- A onResume()
- B onStop()
- B onDestory()

**Now one more scenario** "From Activity A start Activity B by calling StartActivity(Intent) on button click and use finish() method inside onstart() method on Activity B"

- A onPause()
- B onCreate()
- B onStart()
- A onResume()
- B onStop()
- B onDestory()

### 9) What are the fragments?

A fragment is a UI entity attached to an Activity. Fragments can be reused by attaching to different activities. Activity can have multiple fragments attached to it. The fragment must be attached to an activity and its lifecycle will depend on its host activity.

### 10) What is the lifecycle of the fragment?

- ***onAttach() :*** The fragment instance is associated with an activity instance. The fragment and the activity are not fully initialized. Typically you get in this method a reference to the activity which uses the fragment for further initialization work.
- ***onCreate() :*** The system calls this method when creating the fragment. You should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
- ***onCreateView() :*** The system calls this callback when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View component from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
- ***onActivityCreated() :*** The onActivityCreated() is called after the onCreateView() method when the host activity is created. Activity and fragment instances have been created as well as the view hierarchy of the activity. At this point, the view can be accessed with the findViewById() method. example. In this method, you can instantiate objects which require a Context object
- ***onStart() :*** The onStart() method is called once the fragment gets visible.
- ***onResume() :*** Fragment becomes active.
- ***onPause() :*** The system calls this method as the first indication that the user is leaving the fragment. This is usually where you should commit any changes that should be persisted beyond the current user session.
- ***onStop() :*** Fragment going to be stopped by calling onStop()
- ***onDestroyView() :*** Fragment view will destroy after call this method
- ***onDestroy() :*** called to do the final clean up of the fragment's state but Not guaranteed to be called by the Android platform.

### 11) What lifecycle methods will trigger when FragmentB is added on top of FragmentA?

`FragmentA` is not affected when we inflate `FragmentB` with add

```
B-> onAttach
B-> onCreate
B-> onCreateView
B-> onActivityCreated
B-> onStart
B-> onResume
```

**What lifecycle methods will trigger when FragmentB is pop-backed?**

Since fragment B was added on top of A, fragment A is not affected by the removal of B.

```
B-> onPause
B-> onStop
B-> onDestroyView
B-> onDestroy
B-> onDetach
```

**What lifecycle methods will trigger when FragmentB replaces FragmentA?**

*When Fragment B replaces Fragment A, Fragment A is destroyed and Fragment B is created. However, in case the transaction that had added Fragment A was saved using the addToBackStack method, then the backstack is holding a reference to that fragment from the previous transaction and hence only its view is destroyed. i.e. onDestroy and onDetach method of Fragment A will not be invoked.*

```
B-> onAttach
B-> onCreate
A-> onPause
A-> onStop
A-> onDestroyView
A-> onDestroy
A-> onDetach
B-> onCreateView
B-> onActivityCreated
B-> onStart
B-> onResume
```

**What happens if FrgametnB is pop-backed?**

```
B -> onPause
B -> onStop
B -> onDestroy
B -> onDestroyView
B -> onDetach
A-> onAttach
A-> onCreate
A-> onCreateView
A-> onActivityCreated
A-> onStart
A-> onResume
```

## 12) When should you use a fragment rather than an activity?

- When there are UI components that are going to be used across multiple activities.
- When there are multiple views that can be displayed side by side (viewPager tabs)
- When you have data that needs to be persisted across Activity restarts (such as retained fragments)

## 13) Difference between adding/replacing fragments in the backstack?

- ***replace*** removes the existing fragment and adds a new fragment. This means when you press the back button the fragment that got replaced will be created with its onCreateView being invoked.

- ***add*** retains the existing fragments and adds a new fragment which means the existing fragment will be active and they wont be in 'paused' state hence when a back button is pressed onCreateView is not called for the existing fragment(the fragment which was there before new fragment was added).

- *In terms of the fragment's life cycle events onPause, onResume, onCreateView, and other life cycle events will be invoked in case of replace but they won't be invoked in case of add.*

### 14) What are retained fragments?

By default, Fragments are destroyed and recreated along with their parent Activities when a configuration change occurs. Calling ***setRetainInstance(true)*** allows us to bypass this destroy-and-recreate cycle, signaling the system to retain the current instance of the fragment when the activity is recreated.

### 15) Difference between FragmentPagerAdapter vs FragmentStatePagerAdapter?

- ***FragmentPagerAdapter:*** *t*he 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. 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 a large number of fragments.

### 16) How does the activity respond when the user rotates the screen?

When the screen is rotated, the current instance of an activity is destroyed a new instance of the Activity is created in the new orientation. The onRestart() method is invoked first when a screen is rotated. The other lifecycle methods get invoked in a similar flow as they were when the activity was first created.

### 17) How to prevent the data from reloading and resetting when the screen is rotated?

- The most basic approach would be to use a combination of ***ViewModels*** and `onSaveInstanceState()` . So how do we do that?

- Basics of ***ViewModel****:* A ViewModel is ***LifeCycle-Aware.*** In other words, a ViewModel will not be destroyed if its owner is destroyed for a configuration change (e.g. rotation). The new instance of the owner

will just be re-connected to the existing ViewModel. So if you rotate an Activity three times, you have just created three different Activity instances, but you only have one ViewModel.

- So the common practice is to store data in the ViewModel class (since it persists data during configuration changes) and use OnSaveInstanceState to store small amounts of UI data.
- For instance, let's say we have a search screen and the user has entered a query in the Edittext. This results in a list of items being displayed in the RecyclerView. Now if the screen is rotated, the ideal way to prevent resetting of data would be to store the list of search items in the ViewModel and the query text user has entered in the OnSaveInstanceState method of the activity.

### 18) What are content providers?

A *ContentProvider* provides data from one application to another when requested. It manages access to a structured set of data. It provides mechanisms for defining data security. ContentProvider is the standard interface that connects data in one process with code running in another process.

When you want to access data in a ContentProvider, you must instead use the ContentResolver object in your application's Context to communicate with the provider as a client. The provider object receives data requests from clients, performs the requested action, and returns the results.

### 19) How do access data using Content Provider?

Start by making sure your Android application has the necessary read-access permissions. Then, get access to the ContentResolver object by calling *getContentResolver()* on the Context object, and retrieving the data by constructing a query using *ContentResolver.query()*.

The *ContentResolver.query()* method returns a Cursor, so you can retrieve data from each column using Cursor methods.

### 20) What are services?

A ***Service*** is an application component that can perform long-running operations in the background, and it doesn't provide a user interface. It can run in the background, even when the user is not interacting with your application.

*These are the three different types of services:*

- ***Foreground Service:*** A foreground service performs some operation that is noticeable to the user. For example, we can use a foreground service to play an audio track. A Notification must be displayed to the user.

- ***Background Service:*** A background service performs an operation that isn't directly noticed by the user. In Android API level 26 and above, there are restrictions to using background services and it is recommended to use WorkManager in these cases.

- ***Bound Service:*** A service is bound when an application component binds to it by calling ***bindService()***. A bound service offers a client-server interface that allows components to interact with the service, send requests, and receive results. A bound service runs only as long as another application component is bound to it.

### 21) Difference between Service & Intent Service?

- ***Service*** is the base class for Android services that can be extended to create any service. A class that directly extends Service runs on the main thread so it will block the UI (if there is one) and should therefore either be used only for short tasks or should make use of other threads for longer tasks.

- ***IntentService*** is a subclass of Service that handles asynchronous requests 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.

### 22) Difference between AsyncTasks & Threads?

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

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

### 23) Difference between Service, Intent Service, AsyncTask & Threads?

- ***Android service*** is a component that is used to perform operations in the background such as playing music. It doesn't have any UI (user interface). The service runs in the background indefinitely even if the application is destroyed.

- ***AsyncTask*** allows you to perform asynchronous work on your 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.

- ***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.

### 24) What are 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.

- Handlers allow communication between the background thread and the main thread.
- A Handler class is preferred when we need to perform a background task repeatedly after every x seconds/minute.

### 25) What is Android Bound Service?

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 the Service class but we have to override the onBind method too. This method returns an object that implements IBinder, which can be used to interact with the service.

### 26) Difference between Serializable and Parcelable?

***Serializable*** is a standard Java interface.

- Serialization is the process of converting an object into a stream of bytes in order to store an object in memory so that it can be recreated at a later time, while still keeping the object's original state and data.
- In this approach, you simply mark a class Serializable by implementing the interface and Java will automatically serialize it.
- Reflection is used during the process and many additional objects are created. This leads to plenty of garbage collection and poor performance.

***Parcelable*** is an Android-specific interface.

- In this approach, where you implement the serialization yourself.
- Reflection is not used during this process and hence no garbage is created.

- Parcelable is far more efficient than Serializable since it gets around some problems with the default Java serialization scheme. Also, it is faster because it is optimized for usage in the development of Android, and shows better results.

### 27) How would you update the UI of an activity from a background service?

We need to register a LocalBroadcastReceiver in the activity. And send a broadcast with the data using intents from the background service. As long as the activity is in the foreground, the UI will be updated from the background. Ensure to unregister the broadcast receiver in the onStop() method of the activity to avoid memory leaks. We can also register a Handler and pass data using Handlers.

### 28) What is a Sticky Intent?

*Sticky Intents allows communication between a function and a service.* `sendStickyBroadcast()` *performs a sendBroadcast(Intent) known as sticky,* ***i.e.*** the Intent you are sending stays around after the broadcast is complete so that others can quickly retrieve that data through the return value of `registerReceiver(BroadcastReceiver, IntentFilter)`. ***For example,*** if you take an intent for ACTION_BATTERY_CHANGED to get battery change events: When you call registerReceiver() for that action — even with a null BroadcastReceiver — you get the ***Intent that was last Broadcast for that action***. Hence, you can use this to find the state of the battery without necessarily registering for all future state changes in the battery.

### 29) What is a Pending Intent?

If you want someone to perform any Intent operation at a future point of time on behalf of you, then we will use Pending Intent.

### 30) What are intent Filters?

Specifies the type of intent that the activity/service can respond to.

### 31) Launch modes in Android?

- ***Standard****:* It creates a new instance of an activity in the task from which it was started. Multiple instances of the activity can be created and multiple instances can be added to the same or different tasks.
  ***Eg:*** *Suppose there is an activity stack of A -> B -> C.*
  *Now if we launch B again with the launch mode as* ***"standard"****, 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 at the **top** of the stack, then it will **not** create a new instance but rather send the intent to the existing instance of the activity.
  ***Eg:*** *Suppose there is an activity stack of A -> B.*

*Now if we launch C with the launch mode as **"singleTop"**, the new stack will be A -> B -> C as usual.*
*Now if there is an activity stack of A -> B -> C.*
*If we launch C again with the launch mode as **"singleTop"**, the new stack will still be A -> B -> C.*

- ***SingleTask**: A new task will always be created and a new instance will be pushed to the task as the root one. So if the activity is already in the task, the intent will be redirected to onNewIntent() else a new instance will be created. At a time only one instance of activity will exist.*
  ***Eg:** Suppose there is an activity stack of A -> B -> C -> D.*
  *Now if we launch D with the launch mode as **"singleTask"**, the new stack will be A -> B -> C -> D as usual.*
  *Now if there is an activity stack of A -> B -> C -> D.*
  *If we launch activity B again with the launch mode as **"singleTask"**, the new activity stack will be A -> B. Activities C and D will be destroyed.*

- ***SingleInstance**: Same as a single task but the system does not launch any activities in the same task as this activity. If new activities are launched, they are done so in a separate task.*
  ***Eg:** Suppose there is an activity stack of A -> B -> C -> D. If we launch activity B again with the launch mode as **"singleInstance"**, the new activity stack will be:*
  *Task1 — A -> B -> C*
  *Task2 — D*

### 32) What is Toast in Android?

Android Toast can be used to display information for a short period of time. A toast contains a message to be displayed quickly and disappears after sometime.

### 33) What are Loaders in Android?

Loader API was introduced in API level 11 and is used to load data from a data source to display in an activity or fragment. Loaders persist and cache results across configuration changes to prevent duplicate queries.

### 34) Difference between margin & padding?

***Padding*** will be space added inside the container, for instance, if it is a button, padding will be added inside the button.

***Margin*** will be space added outside the container.

### 35) What is a View Group? How are they different from Views?

***View:*** View objects are the basic building blocks of User Interface(UI) elements in Android. The view is a simple rectangle box that responds to the user's actions. Examples are EditText, Button, CheckBox, etc. View refers to the `android.view.View` class, which is the base class of all UI classes.

***ViewGroup:*** ViewGroup is the invisible container. It holds View and ViewGroup. For example, LinearLayout is the ViewGroup that contains Button(View), and other Layouts also. ViewGroup is the base class for Layouts.

### 36) What is View Lifecycle?

Activity and Fragment lifecycles are the most frequently asked for any level of developer. But when it comes to view the lifecycle, senior Android developers must learn about it, for juniors it's optional but it's good to know. When compared to the above two view life cycle is a bit simple, have a look:

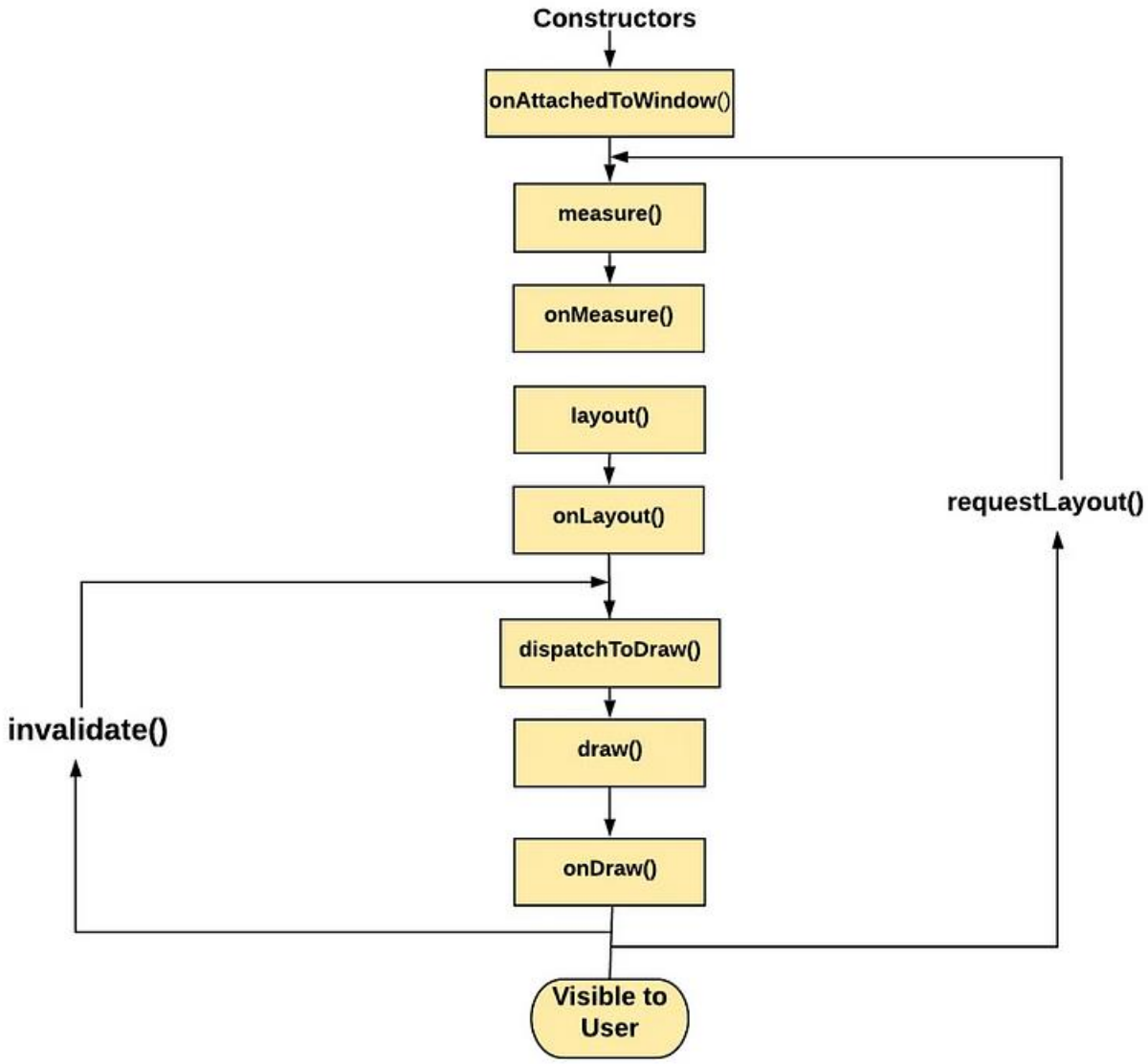


Image: View Life Cycle

- ***onAttachedToWindow()*** Called when the view is attached to a window. View class have its opposite method onDettachToWindow()

- ***onMeasure(int, int)*** Called to determine the size requirements for this view and all of its children. But we don't use it. we use **setMeasuredDimension(800, 300);**
- ***onLayout( int, int, int, int)*** Called when this view should assign a size and position to all of its children.
- ***onDraw(android.graphics.Canvas)*** is Called when the view should render its content. It provides Canvas as an argument, we draw anything on Canvas using the Paint class Instance.

### 37) Difference between RelativeLayout and LinearLayout?

***Linear Layout*** — Arrange elements either vertically or horizontally. i.e. in a row or column.

***Relative Layout*** — Arranges elements relative to parent or other elements.

### 38) What is ConstraintLayout?

It allows you to create large and complex layouts with a flat view hierarchy (no nested view groups). It's similar to RelativeLayout in that all views are laid out according to relationships between sibling views and the parent layout, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.

### 39) What is the use of FrameLayout?

***Frame Layouts*** are designed to contain a single item, making them an efficient choice when you need to display a single View.

If you add multiple Views to a FrameLayout then it'll stack them one above the other, so FrameLayouts are also useful if you need overlapping Views. ***for example****, if you're implementing an overlay or a HUD element.*

### 40) What are Adapters?

An ***adapter*** responsible for converting each data entry into a View that can then be added to the AdapterView (ListView/RecyclerView).

### 41) How to support different screen sizes?

- ***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 your UI. ConstraintLayout allows you 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.
- ***Create stretchable nine-patch bitmaps***

- ***Avoid hard-coded layout sizes*** — Use wrap_content or match_parent.
- ***Create alternative layouts*** — The app should provide alternative layouts to optimize the UI design for certain screen sizes. For eg: different UI for tablets
- ***Use the smallest width qualifier*** — *For example,* you can create a layout named main_activity that's optimized 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 your layout.

## 42) What is an Application Not Responding (ANR) error, and how can you prevent them from occurring in an app?

An ANR dialog appears when your UI has been unresponsive for *more than 5 seconds*, usually because you've blocked the main thread. To avoid encountering ANR errors, you should move as much work off the main thread as possible.

The following measures can be taken to avoid ANR:

- An application should perform lengthy database or networking operations in separate threads to avoid ANR.
- For background task-intensive applications, you can lessen pressure from the UI thread by using the IntentService.

## 43) What is a singleton class in Android?

A singleton class is a class which can create only an object that can be shared with all other classes.

**Properties of Singleton Class**

The characteristics of a typical singleton class are listed below:

- ***Only one instance:*** The singleton class only has one instance, which is accomplished by offering a class instance inside another class. Additionally, it should be made impossible for outside classes and subclasses to create the instance.
- ***Globally accessible:*** Each class should be able to use the singleton class instance as it should be globally accessible.

**Rules for Making a Class Singleton**

To create a Singleton class, the guidelines listed below must be adhered to:

- A private constructor
- A static reference of its class
- One static method
- Globally accessible object reference
- Consistency across multiple threads

```
public class singleTonExample {
 // private static instance variable to hold the singleton instance
 private static volatile singleTonExample INSTANCE = null;

 // private constructor to prevent instantiation of the class
 private singleTonExample() {}

 // public static method to retrieve the singleton instance
 public static singleTonExample getInstance() {
  // Check if the instance is already created
  if(INSTANCE == null) {
   // synchronize the block to ensure only one thread can execute at a time
   synchronized (singleTonExample.class) {
    // check again if the instance is already created
    if (INSTANCE == null) {
     // create the singleton instance
     INSTANCE = new singleTonExample();
    }
   }
  }
  // return the singleton instance
  return INSTANCE;
 }
}
```

## 44) What's the difference between commit() and apply() in SharedPreferences?

***commit()*** writes the data synchronously and returns a boolean value of success or failure depending on the result immediately.

***apply()*** is asynchronous and it won't return any boolean response. Also if there is an apply() outstanding and we perform another commit(). The commit() will be blocked until the apply() is not completed.

## 45) How does RecyclerView work?

- ***RecyclerView*** is designed to display long lists (or grids) of items. Say we want to display 100 rows 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 in time, only 10 or so items could fit on the 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.

- ***But what happens when you start scrolling and need to start showing the 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 a 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.

### 46) How does RecyclerView differ from ListView?

- ***ViewHolder Pattern:*** Recyclerview implements the ViewHolders 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 you 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, you still need to look up the elements and update them. A way around repeated use of `findViewById()` is to use the "view holder" design pattern.

- ***LayoutManager:*** 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.

- ***Item Animator:*** ListViews are lacking in support of good animations, but the RecyclerView brings a whole new dimension to it.

### 47) What is AAPT?

AAPT stands for Android Asset Packaging Tool. It is a build tool that gives the ability to developers to view, create, and update ZIP-compatible archives (zip, jar, and apk). It parses, indexes, and compiles the resources into a binary format that is optimized for the platform of Android.

### 48) What is AIDL? Which data types are supported by AIDL?

***AIDL***(Android Interface Definition Language) is a tool that handles the interface requirements between a client and a service for interprocess communication(IPC) to communicate at the same level.

The process involves dividing an object into primitives that are understood by the Android operating system. Data Types supported by AIDL are as follows:

- String
- List
- Map
- CharSequence
- Java data types (int, long, char, and boolean)

### 49) What are broadcast receivers? How is it implemented?

A broadcast receiver is a mechanism used for listening to system-level events like listening for incoming calls, SMS, etc. by the host application. It is implemented as a subclass of BroadcastReceiver class and each message is broadcasted as an intent object.

```
public class MyReceiver extends BroadcastReceiver
{
    public void onReceive(context,intent){}
}
```

### 50) What is the difference between compileSdkVersion and targetSdkVersion?

**compileSdkVersion:**

- The *compileSdkVersion* is the version of the API the application is compiled against. You can use Android API features involved in that version of the API (as well as all previous versions).
- For example, if you try and use API 15 features but set compileSdkVersion to 14, you will get a compilation error. If you set compileSdkVersion to 15 you can still run the app on an API 14 device as long as your app's execution paths do not attempt to invoke any APIs specific to API 15.

**targetSdkVersion:**

- The *targetSdkVersion* indicates that you have tested your app on (presumably up to and including) the version you specify. This is like a certification or sign-off you are giving the Android OS as a hint to how it should handle your application in terms of OS features.

- For example, setting the targetSdkVersion value to “11” or higher permits the system to apply a new default theme (Holo) to the application when running on Android 3.0 or higher. It also disables screen compatibility mode when running on larger screens (because support for API level 11 implicitly supports larger screens).

### 51) What is JobScheduler?

The JobSchedular API is used for scheduling different types of jobs against the framework that will be executed in your app’s own process. This allows your application to perform the given task while being considerate of the device’s battery at the cost of timing control.

The JobScheduler supports batch scheduling of jobs. The Android system can combine jobs for reducing battery consumption. JobManager automatically handles the network unreliability so it makes handling uploads easier.

Here is some example of a situation where you would use this job scheduler:

- Tasks that should be done when the device is connected to a power supply.
- Tasks that require a Wi-Fi connection or network access.
- Tasks should run on a regular basis as a batch where the timing is not critical.

### 52) What are the different storage methods in Android?

Android offers several options to see persistent application data. They are:

- ***Shared Preferences*** **—** Store private primitive data in key-value pairs
- ***Internal Storage*** **—** Store private data on the device's memory
- ***External Storage*** **—** Store public data on the shared external storage
- ***SQLite Databases*** **—** Store structured data in a private database

### 53) What is a work Manager in Android?

Android WorkManager is a background processing library that is used to execute background tasks that should run in a guaranteed way but not necessarily immediately. With WorkManager we can enqueue our background processing even when the app is not running and the device is rebooted for some reason.

***WorkManager Features:***

- It is fully backward compatible so in your code, you do not need to write **if-else** for checking the Android version.
- With WorkManager we can check the status of the work.
- Tasks can be chained as well, for example when one task is finished it can start another.
- And it provides guaranteed execution with the constraints, we have many constrained available that we will see ahead.

## 54) Difference between Volley and Retrofit?

***Volley-***

- No automatic parsing
- Caching Mechanism
- volley we can set a retry policy using the setRetryPolicy method
- inbuilt image loading support

***Retrofit-***

- Automatic JSON parsing
- No Caching mechanism
- Retrofit does not support any retrying mechanism
- no Image loading

## 55) Difference between Picasso and Glide?

***Picasso:***

- Slow loading big images from the internet into ListView
- Tinny size of the library
- The small size of the cache
- Simple to use
- UI does not freeze
- WebP support

***Glide:***

- Big size library
- Tinny size of the cache
- Simple to use
- GIF support
- WebP support
- Fast loading big images from the internet into ListView
- UI does not freeze
- BitmapPool to re-use memory and thus lesser GC events

### 56) Difference between MVC & MVP & MVVM?

- **MVC** is the *Model-View-Controller* architecture where the model refers to the data model classes. The view refers to the XML files and the controller handles the business logic. The issue with this architecture is unit testing. The model can be easily tested since it is not tied to anything. The controller is tightly coupled with the Android APIs making it difficult to unit test. Modularity & flexibility is a problem since the view and the controller are tightly coupled. If we change the view, the controller logic should also be changed. Maintenance is also an issue.
- **MVP architecture**: *Model-View-Presenter architecture*. The ***View*** includes the XML and the activity/fragment classes. So the activity would ideally implement a view interface making it easier for unit testing (since this will work without a view).

- **MVVM**: *Model-View-ViewModel Architecture*. The ***Model*** comprises data, tools for data processing, and business logic. The ***View Model*** is responsible for wrapping the model data and preparing the data for the ***view***. It also provides a hook to pass events from the view to the model.

### 57) What is RxJava?

RxJava is a JVM library for doing asynchronous and executing event-based programs by using observable sequences. Its main building blocks are triple O's, Operator, Observer, and Observables. And using them we perform asynchronous tasks in our project. It makes multithreading very easy in our project. It helps us to decide on which thread we want to run the task.

### 58) What is RxAndroid?

RxAndroid is an extension of RxJava for Android which is used only in Android applications.

RxAndroid introduced the **Main Thread** required for Android.

To work with the multithreading in Android, we will need the **Looper and Handler** for Main Thread execution

RxAndroid provides **AndroidSchedulers.mainThread()** which returns a scheduler that helps in performing the task on the main UI thread that is mainly used in the Android project. So,
here **AndroidSchedulers.mainThread()** is used to provide us access to the main thread of the application to perform actions like updating the UI.

### 59) What is ProGuard?

ProGuard is a free Java tool in Android, which helps us to do the following,

- Shrink(Minify) the code: Remove unused code in the project.
- Obfuscate the code: Rename the names of class, fields, etc.
- Optimize the code: Do things like inlining the functions.

In short, ProGuard makes the following impact on our project,

- It reduces the size of the application.

- It removes the unused classes and methods that contribute to the 64K method count limit of an Android application.
- It makes the application difficult to reverse engineer by obfuscating the code.

**How it is useful for our application?**

In Android, proguard is very useful for making a production-ready application. It helps us to reduce the code and make apps faster. Proguard comes out of the box by default in Android Studio and it helps in a lot of ways, a few of them are mentioned below,

- It obfuscates the code, which means that it changes the names to some smaller names like for **MainViewModel** it might change the name to **A**. Reverse Engineering of your app becomes a tough task now after obfuscating the app.
- It shrinks the resources i.e. ignorers the resources that are not called by our Class files, not being used in our Android app like images from drawable, etc. This will reduce the app size by a lot. You should always shrink your app to keep it light weighted and fast.