2022

Android Development

MOBILE APPLICATION DEVELOPMENT

ASHIRBAD SWAIN

SILENTCODER | Odisha, India

Android Roadmap

- Programming
- Java Programming
- Kotlin Programming
- XML

Android Studio

File Structure

- AndroidManifest.xml file
- Java file/Kotlin file
- Drawable file
- Layout file
- Mipmap file
- Colors.xml file
- Strings.xml file
- Styles.xml file
- Build.gradle (Module: app) file

Android Studio Overview

- Create a new project
- Reopen, close, save the project
- Create a new activity, classes, drawable resource files
- Run the app on AVD of Emulator or in a real device etc.

Android Components

- Activity
- Activity Life Cycle
- Handle Activity State Changes
- Understand Tasks and Back Stack
- Processes and Application Lifecycle

- Services
- Types of Android Services
- The Life Cycle of Android Services

- Content Provider
- Content URI
- Operations in Content Provider
- Working of the Content Provider
- Creating a Content Provider

- Broadcast Receiver
- Implicit Broadcast Exceptions

Simple UI Design

- Explore Different Layouts
- Frame
- Linear
- Relative
- Constraint

- View Elements
- TextView
- EditText
- Buttons
- ImageView

- Intent
- Implicit
- Explicit
- Intent Filter

Complex UI Design

- ListView
- RecyclerView
- Fragments
- Dialogs
- Toast
- Bottom Sheets
- Navigation Drawer
- Tabs
- Material Design
- Some Interesting Animations

Storage

- Shared Preferences
- File System
- Database (RoomDB, SQLite)

Build

- Gradle
- Debug/Release Configuration

Threading

- Threads
- Looper

Debugging

- Exceptions
- Error Handling
- Logging
- Memory Profiling

Memory Leaks

- Cause of memory leaks
- Detecting and fixing memory leaks
- Context

Third – Party Libraries

- Image Loading
- Glide
- Picasso

- Video Streaming
- ExoPlayer

- Dependency Injection
- Dragger

- Networking
- Retrofit

- Multithreading (Reactive Programming)
- Coroutines
- RxJava
- RxAndroid

- View Binding
- Butterknife
- Android Data Binding Library

- Testing
- Junit
- Mockito
- Espresso

- Custom Fonts
- Calligraphy
- Custom Fonts with Support Library

- Job Scheduling
- Android Job

- Security
- Auth0

Android Jetpack

- AppCompat library
- Architecture Components
- Animation and Transitions
- Android Ktx
- Navigation
- Paging
- Slices
- Work Manager

Android Architecture

- MVVM (Model – View – ViewModel)
- MVI (Model – View – Intent)
- MVP (Model View Presenter)

Firebase

- FCM (Firebase Cloud Messaging)
- Analytics
- Remote Config
- App Indexing

Unit Testing

- Local Unit Testing
- Instrumentation Testing

Security

- Encrypt/Decrypt
- Proguard

App Release

- Signed APK
- Play Store

Android SDK

Android Interview Questions

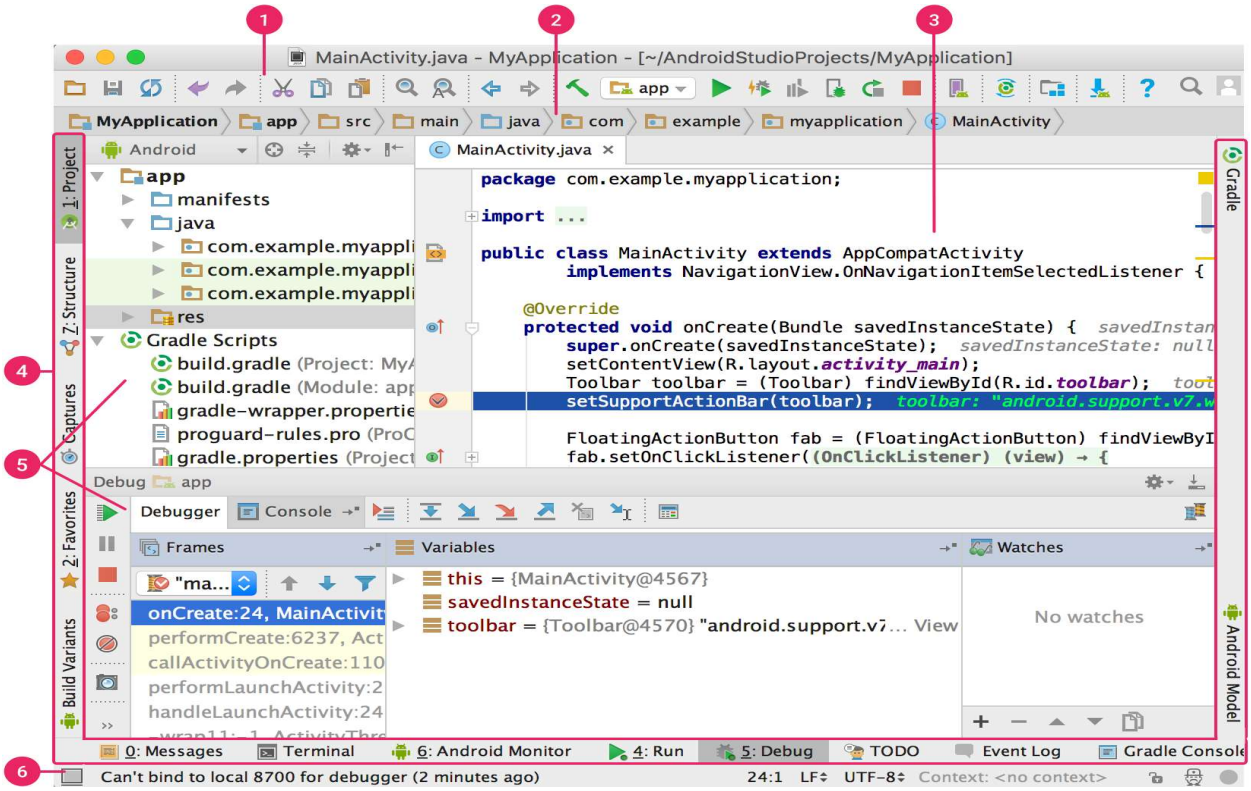
Different Versions of Android

JSON

Android Studio

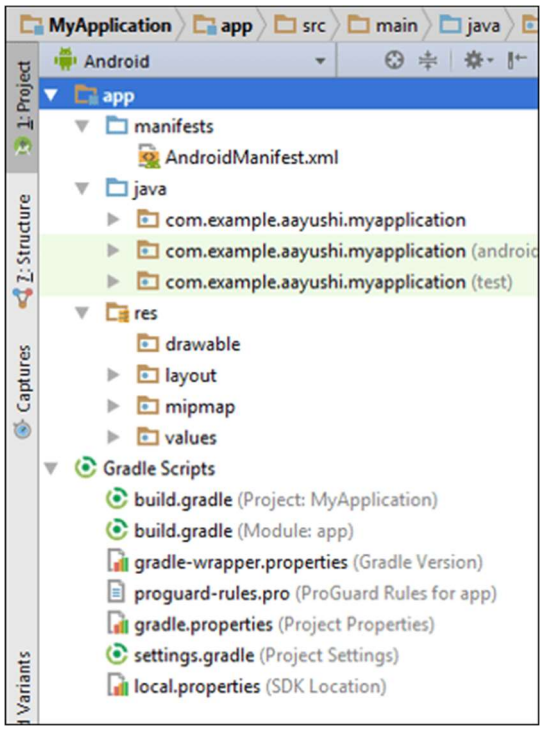
Android Studio IDE Overview

Android Studio is the official Integrated Development Environment (IDE) for Android app development, based on IntelliJ IDEA.



- The toolbar lets you carry out a wide range of actions, including running your app and launching android tools.
- The navigation bar helps you navigate through your project and open files for editing. It provides a more compact view of the structure visible in project window.
- The editor window is where you create and modify code. Depending on the current file type, the editor can change. For example, when viewing a layout file, the editor displays the Layout Editor.
- The tool window bar runs around the outside of the IDE window and contains the buttons that allows you to extend to collapse individual tool windows.
- The tool windows give you access to specific tasks like project management, search, version control and more. You can expand them and collapse them.
- The status bar displays the status of your project and the IDE itself, as well as any warnings or messages.

Project Structure



app

- it describes the fundamental characteristics of the app and defines each of its components.

Manifest file

- Manifest file plays an integral role as it provides the essential information about your app to the android system, which the system must have before it can run any of the app's code.
- Manifest file performs various tasks such as:
- It names the Java package for the app as the package name serves as a unique identifier for the application.
- It protects the application by declaring permission in order to access protected parts of the API and interact with other applications.
- Manifest file declares the minimum level of the Android API and lists the libraries which is linked with the application.
- Manifest file list the instrumentation classes. These classes provide profiling and other information as the application runs, but this information is removed as soon the application is publishes. It remains only till the application is in development mode.
- The Manifest includes many types of information, the main ones are:
- Package name
- Components of the app, such as activities, fragments and services
- Permissions needed from the user

Java

- This folder contains the .java/.kt source files for your project. By default, it includes an MainActivity.java source file.
- Under this, you create all the activities which have .java extensions and all the code behind the application.
- MainActivity.java is the actual file which gets converted to a dalvik executable and runs your app.

res Directory

- It is a directory for files that define your app's user interface. You can add TextView, Button etc. to build the GUI and use its various attributes like android:layout_width, android:layout_height etc which are used to set its width and height.
- The res directory is where you put things such as images, strings and layouts. It's included in every android project, and you can see it in Android Studio res directory.
- Inside of the res directory, are sub folders for the following types of resources. You may have a subset of these directories, depending on the types of the resources you are using in your app.
- Different Resources Directories

Name	What's Stored Here
values	XML files that contain simple values, such as string or integers
drawable	A bunch of visual files, including Bitmap file types and shapes
layouts	XML layouts for your app
mipmap	Drawable files for launcher icon
animator	XML files for property animations
anim	XML files for tween animations
color	XML files that define state list colors
menu	XML files that define applications menus
raw	Resource file for arbitrary files saved in their raw form. For example, you could put audio files here.
xml	Arbitrary XML; if you have XML configuration files, this is a good place to put them.

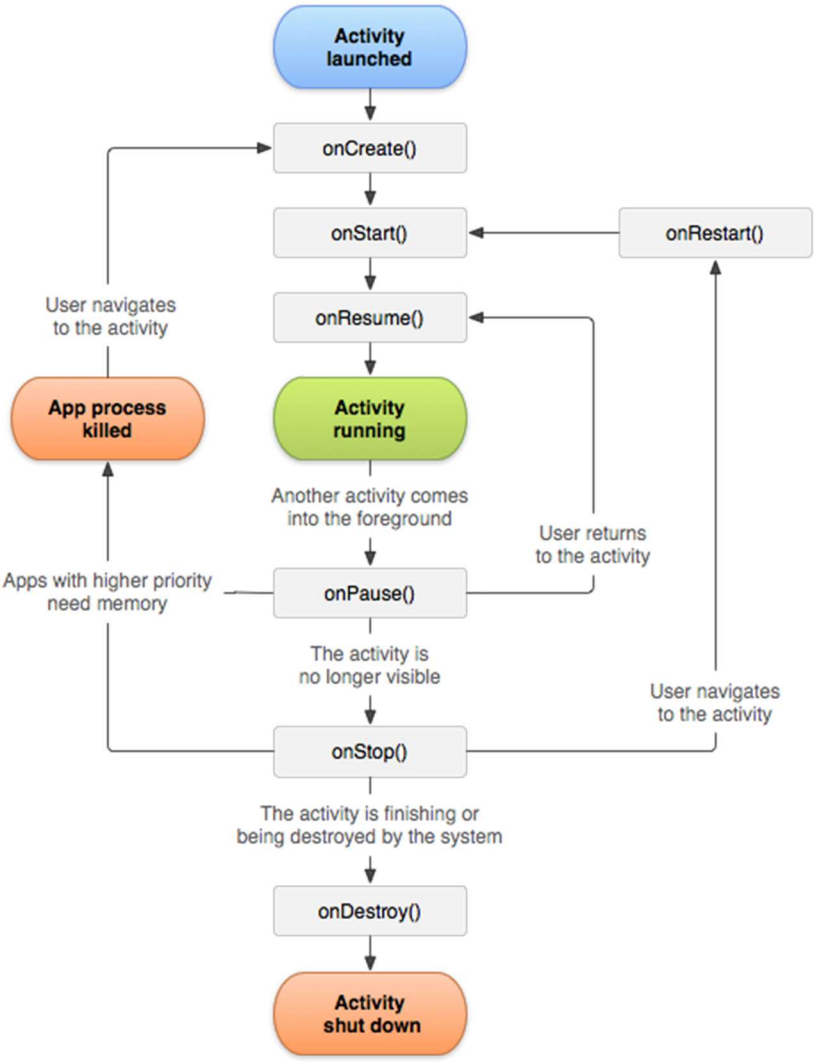
Gradle Scripts

- This is an auto generated file which contains compileSdkVersion, buildToolsVersion, applicationID, minSdkVersion, targetSdkVersion, versionCode and versionName.

Android Components

Activity

- An activity represents a single screen with a user interface just like window or frame in an application. Android activity is the subclass of ContextThemeWrapper class.
- It is very similar to a single window of any desktop application. An android app consists of one or more screens or activities.



Each activity goes through various stages or a lifecycle and is managed by activity stacks. So, when a new activity starts, the previous one always remains below it. There are 4 stages of an activity.

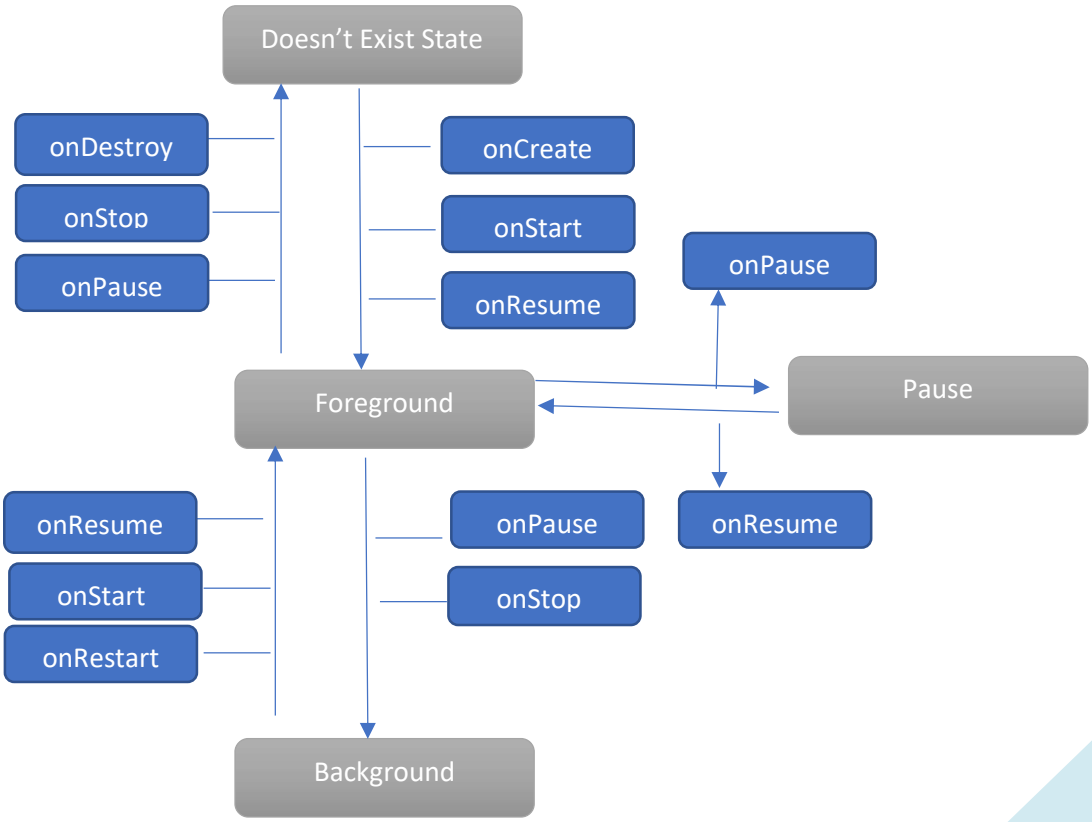
- Doesn't exists
- Foreground
- Background
- Pause

There are 3 key loops you may be interested in monitoring within your activity:

- The entire lifetime of an activity happens between the first call to onCreate (Bundle) through to a single final call to onDestroy ( ). An activity will do all setup of "global" state on onCreate ( ), and release all remaining resources in onDestroy ( ).
- For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate ( ) and then stop the thread in onDestroy ( ).

- The visible lifetime of an activity happens between a call to onStart ( ) until a corresponding call to onStop ( ). During this time the user can see the activity on-screen.

- The foreground lifetime of an activity happens between a call to onResume ( ) until a corresponding call to onPause ( ). During this time the activity is in visible, active and interacting with the user.



For each stage, android provides us with a set of 7 methods that have their own significance for each stage in the life cycle.

onCreate ( )

It is called when the activity is first created. This is where all the static work is done like creating views, binding data to lists, etc. This method also provides a Bundle containing its previous frozen state, if there was one.

onStart ( )

It is invoked when the activity is visible to the user. It is followed by onResume ( ) if the activity is invoked from background. It is also invoked after onCreate ( ) when the activity is first started.

onRestart ( )

It is invoked after the activity has been stopped and prior to its starting stage and thus is always followed by onStart ( ) when any activity is revived from background to on-screen.

onResume ( )

It is invoked when the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, with a user interacting with it. Always followed by onPause ( ) when the activity goes into the background or is closed by the user.

onPause ( )

It is invoked when an activity is going into the background but has not yet been killed. It is a counterpart to onResume ( ). When an activity is launched in front of another activity, this callback will be invoked on the top activity (currently on screen). The activity, under the active activity, will not be created until the active activity's onPause ( ) returns, so it is recommended that heavy processing should not be done in this part.

onStop ( )

It is invoked when the activity is not visible to the user. It is followed by onRestart ( ) when the activity is revoked from the background, followed by onDestroy ( ) when the activity is closed or finished, and nothing when the activity remains on the background only.

Note that this method may never be called, in low memory situations where the system does not have enough memory to keep the activity's process running after its onPause ( ) method is called.

onDestroy ( )

The final call received before the activity is destroyed. This can happen either because the activity is finishing (when finish ( ) is invoked) or because the system is temporarily destroying this instance of the activity to save space. To distinguish between these scenarios

How to handle activity state changes in android?

As we know that every Android app has at least one activity associated with it. When the application begins to execute and runs, there are various state changes that activity goes through. Different events some user-triggered and some system triggered can cause an activity to do the transition from one state to another. All the state changes associated with the activity are part of the activity lifecycle in android.

When a certain event occurs, each activity has to go through different stages of the android lifecycle. The state of activity is managed by maintaining the activity stacks, in which the new activity is on the top of the stack, and the rest of the activity is below in stack in the order of the time they have been put in the stack.

Imagine 3 activities, one is in a running state, the second activity is just minimized and is running in the background, and the 3rd activity is also running in the background but has gone from foreground to background earlier than the second activity. So, here the 1st activity will be on the top of the stack, 3rd activity will be at the bottom of the stack and 2nd activity will be at the middle of the stack.

Events leading to configuration change/state change

Changing the Orientation of the app

Let's say the user is viewing the app in portrait mode and suddenly the user changes the orientation of the application to landscape mode. Since changing the orientation of the application will eventually change the view, which will lead to a configuration change in the app and the activity associated with the portrait mode is destroyed and new activity associated with the landscape mode is created. The running/active activity associated with the portrait mode will have onPause (), onStop (), and onDestroy () callbacks triggered. When the user changes the orientation, the new activity is created and new activity will have onCreate (), onStart (), and onResume () callbacks triggered.

Switching between the apps in MultiWindow Screen

The common example of changing the configuration of the app is when the user resizes one activity with respect to another activity. Your activity can handle the configuration to change itself, or it can allow the system to destroy the activity and recreate it with the new dimensions.

In multi-window mode, although there are two apps that are visible to the user, there is only the one app with which the user is interacting is in the foreground and has focus is in the onResume() state and another app that is in the background and visible to the user is in a Paused state. So, it can be said that the activity with which the user is interacting is the only activity which is in the resumed state, while all the other activities are started but not resumed. When the user switches from one app to another app, the system calls onPause() lifecycle state on the running app, and onResume() lifecycle state on another app which is previously not in the active state. It switches between these two methods each time the user toggles between apps.

It can be seen that when there are multiple screens running at the same time, every time when the user resizes one screen relative to another screen, the original activity gets destroyed and new activity gets created corresponding to the new screen size

Activity or dialog appears in the foreground

Let's suppose when an activity is running and is in progress, but at the same time, a new activity appears in the foreground, and partially covering the activity in progress, and take the focus and causing the running activity to enter into the Paused state by calling onPause() on the running activity and the new activity enter into the onResume() state. When the covered activity returns to the foreground and regains focus, it calls onResume(), whereas the running activity again enters into the onPause() state.

If a new activity or dialog appears in the foreground, taking focus and completely covering the activity in progress, the covered activity loses focus and enters the Stopped state. The android operating system then immediately calls onPause() and onStop() in succession. When the same instance of the covered activity comes back to the foreground, the system calls onRestart(), onStart(), and onResume() on the activity. If it is a new instance of the covered activity that comes to the background, the system does not call onRestart(), only calling onStart() and onResume().

The user taps the Back button

- If an activity is in the foreground and is in running state, and the user taps the Back button, the activity transitions through the onPause(), onStop(), and onDestroy() callbacks. In addition to being destroyed, the activity has also removed the stack, which is used for storing the activities in order of the time put in the stack. It is important to note that, by default, the onSaveInstanceState() callback is not called in this case as it is assumed that the user tapped the Back button with no expectation of returning to the same instance of the activity and so there is no need to save the instance of the activity. If the user overrides the onBackPressed() method, it is still highly recommended that the user should invoke super.onBackPressed() from the overridden method, else if super.onBackPressed() is not invoked then it may lead to ambiguous behavior of the application and may lead to bad user experience. See the Toast messages to see the activity state changes.

Tasks and Back stack in Android

Tasks

- A task is a collection of metadata and information around a stack of activities.
- So, when you tap the launcher icon for your app, what the system is actually doing is looking for a previously existing task (determined by the Intent and Activity it points to) to resume – getting you back to exactly where you were. If no existing task is found, then a new task is created with your newly launched activity as the base activity on the task's back stack.
- A task holds the activities, arranged in a stack called Back Stack. The stack has LIFO structure and stores the activities in the order of their opening. The activities in the stack are never rearranged. The navigation of the back stack is done with the help of the Back Button.

Back Stack

- As you might imagine, a task's back stack is tied together with the back button, but it goes both ways. When you start a new activity using startActivity(), that is (by default) pushing a new activity onto your task, causing the previous Activity to be paused (and stopped if the new activity fully obscures the previous activity).

- The back button (by default) then pops the stack, calling finish() on the topmost activity, destroying it and removing it from the back stack and taking you back to the previous activity. This repeats until there's nothing left in the back stack and you are back at the launcher.

We will now go through the default behavior of the Task and Back Stack.

- The application launcher creates a new Task with the main activity created and placed in the root of the back stack (It has another role that we will review later).
- When the current activity starts another activity, then the new activity is pushed on top of the stack and takes focus.
- The previous activity moves below this new activity in the back stack and is stopped. The system retains the current state of this activity's user interfaces like text in the form, scroll position etc.
- The subsequent new activities keep on piling the back stack.
- When the back button is pressed then the current activity is popped from the back stack. This destroys the activity and the previous activity resumes with its state restored.
- The back button then keeps on popping the current activities and restoring the previous activities. When the last activity is removed from the back stack, then the task terminates to the screen that was last running before the creation of the task (in our case the launcher screen).
- When the back stack goes empty then the task ceased to exist.
- Activities of different applications invoked by the intent are put into the same task.

Task has few important properties:

- It goes into the background when a new stack is created or when the Home button is pressed.
- It is then brought to the foreground by clicking the launcher icon (this is another role of the launcher icon that we mentioned earlier) or by selecting the task from the recent screens.
- When multiple tasks are in the background or when the user leaves the tasks for a long time, the system in order to recover memory clears the task of all the activities except the root activity. When the user returns to the task again, only the root activity is restored (this behavior can be overridden).

Processes & Android Application life cycle

Android is a perfect example of true multi-tasking i.e.; you can perform more than one tasks at a particular instant of time. Actually, we are not performing multiple tasks at a particular instant of time, rather it seems that we are performing more than one tasks at a time. In a very simple way, at any particular instant of time, only one app can be in the running state and other apps will be in the background state. All these processes that we are doing more than one task at some instant of time.

But the problem that arises here is that, a particular Android device has some limited amount of space and processing speed, and in order of have a fluent flow or to provide a fluent use of all the applications, to give a better experience to the users, android pushes the application that is least used in some cache. By doing so, the app that is not being used for a longer period of time, will be pushed in the background and possibly it's onStop() method will be called to stop the activity of that app.

LRU Cache

LRU or Least Recently Used Cache is the cache used by the Android OS to push the applications that are least used in the nearer time. For example, if you are running the music app along with the Email, Facebook, Instagram and WhatsApp application then the app which you have not used for a long time will be placed at the header or the front of th cache and the application that is used recently will be put in the back of LRU cache queue.

For example, if the Email app is least used and Facebook app is the most used app in the mobile phone at a particular time, then the Email app will be placed on the front of the queue of the LRU cache and the Facebook app will be placed at the back of the LRU cache queue.

Also, if the application is restarted or opened again, then it will be placed at the back of the LRU cache queue. For example, if you open the Email app again, then that Email app will be placed at the back of the LRU cache, not at the front of the LRU cache.

Priorities of Android Application

In order to have a proper memory and battery management in the android device, android pushes or kills the applications that are having less priority. In order to free up some space from the android device, android uses some sets of rules and assign priorities to th applications based on the current running states of the applications.

Following are the process status that are associated with the android application:

Foreground Process

A process is said to be in the foreground state if a user interacts with that process. For example. If you are watching some video on the YouTube app, then the YouTube app will be called to be in the foreground state because that app is currently being used by the user. So, the applications that are on the foreground have the highest priority.

Visible Process

A process is said to be in the visible state when the activity of the application can be visible but not in the fort. For example, whenever you are using some application that requires some kind of permission then you are using the visible process.

For instance, let's take the example of Instagram application, when you want to upload some images from your device then the app will ask you to give permission of the storage. At this time, your Instagram activity is visible but not in the foreground because, in the foreground, you are having the permission box asking for storage permission.

Service Process

A process is said to be Service process if it is currently running but it does not come under the above two categories i.e., the foreground and the visible process. This is helpful for those applications that performs some background tasks such as downloading some data or uploading some data. One example of the service process is the uploading of files in Google drive where uploading of files is done in background.

Background Process

A process is said to be in background state if it's onStop () method is being called by the android. Suppose you are using some application and you suddenly press the home button of your mobile then at that time, your application will go to the background state from the foreground state. Also, the application will be placed in the LRU cache so that it will be called whenever the user reopens the app. This is done so because starting from scratch is very difficult as compared to start from an intermediate state.

Empty Process

A process is said to be in the empty state if it doesn't come under the category of the above four mentioned process states. In an empty process, there is no active component of the application i.e., each and every component of the process will be in stop state. The application can be put in the LRU for better caching purpose but is memory is not present or low in amount, then that application will be removed from the cache also.

Why consider application life cycle?

- In most cases, every Android application runs in its own Linux process. This process is created for the application when some of its code needs to be run, and will remain running until it is no longer needed and the system needs to reclaim its memory for use by other applications.
- What if your app takes or uses a lot of battery? Will it be preferred by the users to use your app? The answer is no. while making an android application, you should handle each and every application state of the android app.
- For example, if you are using some data receiving and sending operation in your app and you want to manage and hold the session whenever the app is running then you should handle the operations that are to be done when your application will be in the background state. If your app is in the background state then you should not hold the session because this will result in more resource utilization and this, in turn, will use more battery and memory. So, in the onPause () state, you should release the session and in the onResume () state, you should again build the session and do the rest operations.
- By doing so, your application will have a good impact on your users and your users will recommend the app to their friends. So, android application life cycle is very important to make an android application.

Services

A service in Android is a background process which is used to perform long-running operations. Let's say, a service of location is active in the background while user is in a different application. So, it does not disturb the user interaction with an activity. Now, services are classified into two types, namely:

- **Local**: This service is accessed from within the application.
- **Remote**: This service is accessed remotely from other applications running on the same device.

It is implemented as a subclass of Service class as follows:

```
public class MyService extends Service {

}
```

What is Service in android and what are their types?

A Service is an application component that can perform long-running operations in the background and it does not provide a user interface.

These are the 3 different types of services:

- Foreground Service
- Background Service
- Bound Service

Foreground Services

Services that notify the user about its ongoing operations are termed as Foreground Services. Users can interact with the service by notifications provided about the ongoing task. Such as in downloading a file, the user can keep track of the progress in downloading and can also pause and resume the process.

Background Services

Background services do not require any user intervention. These services do not notify the user about ongoing background tasks and user also cannot access them. The process like schedule syncing of data or storing of data fall under this service.

Bound Services

This type of android service allows the components of the application like activity to bound themselves with it. Bound services perform their task as long as any application component is bound to it.

More than one component is allowed to bind themselves at a time. In order to bind an application component with a service bindService () method is used.

Can you explain by any real time example when to use bindService or startService?

Suppose, I want to play music in the background, so call startService ( ) method. But I want to get information of the current song being played, I will bind the service that provides information about the current song.

The life cycle of Android Services

In android, services have 2 possible paths to complete its life cycle namely Started and Bounded.

Started Service (Unbounded Service)

By following this path, a service will initiate when an application component calls the startService () method. Once initiated, the service can run continuously in the background even if the component is destroyed which was responsible for the start of the service.

Two option are available to stop the execution of service:

- By calling stopService () method
- The service can stop itself by using stopSelf () method

Bounded Service

It can be treated as a server in a client – server interface. By following this path, android application components can send requests to the service and can fetch results.

A service is termed as bounded when an application component binds itself with a service by calling bindService () method. To stop the execution of this service, all the components must unbind themselves from the service by using unbindService () method.

NOTE

To carry out a downloading task in the background, the startService () method will be called. Whereas to get information regarding download progress and to pause or resume the process while the application is still in the background, the service must be bounded with a component which can perform these tasks.

What is difference between Service and Intent Service?

Service is the base class for all services. Once the service is started the onStartCommand (intent) method in the service is called. It passes in the Intent object from the startService (intent) call. If startService (intent) is called while the service is running, each time its onStartCommand ( ) is also called. Therefore, it's important to create a new thread each time in onStartCommand in which the service can complete all of its work for that particular intent received.

While IntentService is a subclass of Service. Create a work queue that passes one intent at a time to your onHandleIntent ( ) implementation, so you never have to worry about multi-threading.

Example of Android Services

Playing music in the background is a very common example of services in android. From the time when a user starts the service, music plays continuously in the background even if the user switches to another application. The user has to stop the service explicitly in order to pause the music.

Content Providers

Content providers manage access to a structured set of data. It is the standard interface that connects data in one process with code running in another process.

They encapsulate the data and provide mechanism for defining data security.

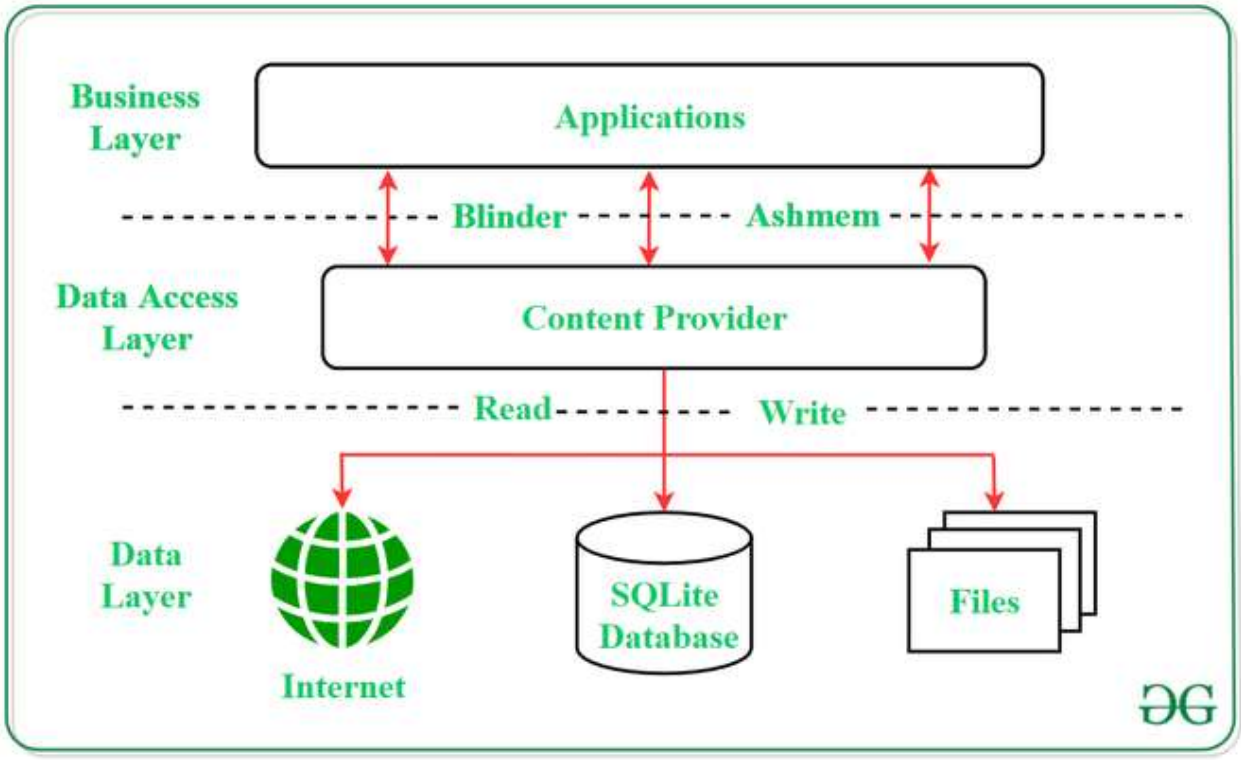
Content providers are used to share the data between different applications.

It is implemented as a subclass of ContentProvider class and must implement a standard set of APIs that enable other applications to perform transactions.

```
public class MyContentprovider extends ContentProvider {

public void onCreate(){}

}
```

The role of the content provider in the android system is like a central repository in which data of the applications are stored, and it facilitates other applications to securely access and modifies that data based on the user requirements.

Android system allows the content provider to store the application data in several ways. Users can manage to store the application data like images, audio, videos and personal contact information by storing them SQLite Database, in files, or even on a network. In order to share the data, content providers have certain permissions that are used to grant or restrict the rights to other applications to interface with the data.



Content URI

Content URI (Uniform Resource Identifier) is the key concept of Content providers. To access the data from a content provider, URI is used as a query string.

Structure of a Content URI:

Content://authority/optionalPath/optionalID

Details of different parts of Content URI:

Content:// - Mandatory part of the URI as it represents that the given URI is a Content URI.

Authority – signifies the name of the content provider like contacts, browser etc. This part must be unique for every content provider.

OptionalPath – specifies the type of data provided by the content provider. It is essential as this part helps content providers to support different types of data that are not related to each other like audio and video files.

optionalID – It is a numeric value that is used when there is a need to access a particular record.

If an ID is mentioned in a URI, then it is an id-based URI otherwise a directory-based URI.

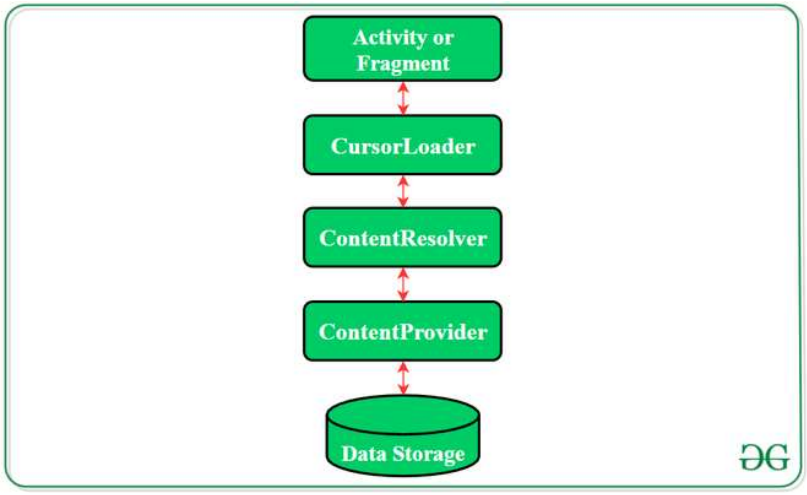
Operations in Content Provider

Four fundamental operations are possible in Content Provider namely Create, Read, Update and Delete. These operations are often termed as CRUD operations.

- **Create**: Operation to create data in a content provider.
- **Read**: Used to fetch data from a content provider.
- **Update**: To modify existing data.
- **Delete**: To remove existing data from the storage.

Working of the Content Provider

UI components of android applications like activity and fragment use an object CursorLoader to send query requests to ContentResolver. The ContentResolver object sends requests (Like create, read, update and delete) to the ContentProvider as a client. After receiving a request, ContentProvider process it and returns the desired result.



Creating a Content Provider

Following are the steps which are essential to follow in order to create a Content Provider:

- Create a class in the same directory where the **MainActivity** file resides and this class must extend the ContentProvider base class.
- To access the content, define a content provider URI address.
- Create a database to store the application data.
- Implement the six abstract methods of ContentProvider class.
- Register the content provider in AndroidManifest.xml using <provider> tag.

Following are the six abstract methods and their description which are essential to override as the part of ContentProvider class:

- **Query ()**	- **A method that accepts arguments and fetches the data from the desired table. Data is retired as a cursor object.**
- **Insert ()**	- To insert a new row in the database of the content provider. It returns the content URI of the inserted row.
- **Update ()**	- This method is used to update the fields of an existing row. It returns the number of rows updated.
- **Delete ()**	- This method is used to delete the existing rows. It returns the number of rows deleted.
- **getType ()**	- This method returns the Multipurpose Internet Main Extension (MIME) type of data to the given Content URI.
- **onCreate ()**	- As the content provider is created, the android system calls this method immediately to initialize the provider.

Example

The prime purpose of a content provider is to serve as a central repository of data where users can store and can fetch the data. The access of this repository is given to other applications also but in a safe manner in order to serve the different requirements of the user. The following are the steps involved in implementing a content provider. In this content provider, the user can store the name of persons and can fetch the stored data. Moreover, another application can also access the stored data and can display the data.

Broadcast Receiver

- It responds to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents.
- Broadcast Receiver is a mechanism using which host application can listen for System level events.
- Broadcast receiver is used by the application whenever they need to perform the execution based on system events. Like listening for incoming calls, SMS etc.
- Broadcast receivers helps in responding to broadcast messages from other application or from the system.
- It is used to handle communication between Android operating system and applications.
- It is implemented as a subclass of BroadcastReceiver class and each message is broadcast as an Intent object.

```
public class MyReceiver extends BroadcastReceiver {

Public void onReceive(context,intent){}

}
```

- Broadcast in android is the system-wide events that can occur when the device starts, when a message is received on the device or when incoming calls are received, or when a device goes to an airplane mode, etc.
- Broadcast Receivers are used to respond to these system-wide events. Broadcast Receivers allow us to register for system and application events, and when that event happens, then the register receiver get notified.

There are mainly two types of Broadcast Receivers:

Static Broadcast Receivers

These types of Receivers are declared in the manifest file and works even if the app is closed.

Dynamic Broadcast Receivers

These types of receivers work only if the app is active or minimized.

The two main things that we have to do in order to use the broadcast receiver in our application are:

Creating the Broadcast Receiver:

```
class AirplaneModeChangeReceiver:BroadcastReceiver() {

    override fun onReceive(context: Context?, intent: Intent?) {

        // logic of the code needs to be written here

    }

}
```

Registering a Broadcast Receiver:

```
IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED).also {

            // receiver is the broadcast receiver that we have registered

            // and it is the intent filter that we have created

            registerReceiver(receiver, it)

    }
```

Following are some of the important system-wide generated intents:

Intent	Description of Event
android.intent.action.BATTERY_LOW	Indicates low battery condition on the device.
android.intent.action.BOOT_COMPLETED	This is broadcast once after the system has finished booting.
android.intent.action.CALL	To perform a call to someone specified by the data.
android.intent.action.DATE_CHANGE	Indicates that the date has changed
android.intent.action.REBOOT	Indicates that the device has been a reboot
android.net.conn.CONNECTIVITY_CHANGE	The mobile network or WIFI connection is changed (or reset)
android.intent.ACTION_AIRPLANE_MODE_CHANGED	This indicates that airplane mode has been switched on or off.

Static User Interface (Simple UI Design)

View Elements

- Button
- ImageView
- TextView
- EditText etc.

ViewGroup (Different Layouts)

- Linear Layout
- Relative Layout
- Frame Layout
- Constraint Layout etc.

A ViewGroup is a special view that can contain other views. The ViewGroup is the base class for Layouts in Android, like LinearLayout, RelativeLayout, FrameLayout etc.

ViewGroup acts as an invisible container in which other Views and Layouts are placed. Means, a layout can hold another in it, or in other words a ViewGroup can have another ViewGroup in it.

Types of View

UI Components

There are two major categories of views. The first type are UI components that are often interactive.

Here are a few examples:

Class Name	Description
TextView	Creates text on the screen; generally, non-interactive text.
EditText	Creates a text input on the screen
ImageView	Creates an image on the screen
Button	Creates a button on the screen
Chronometer	Creates a simple timer on screen

Container View

The second are views called “Layout” or “Container” views. They extend from a class called ViewGroup. They are primarily responsible for containing a group of views and determining where they are on screen. Means that a view will be nested inside the tag of another view.

A few examples of common container views are:

Class Name	Description
LinearLayout	Display views in a single column or row
RelativeLayout	Displays views positioned relative to each other and this view
FrameLayout	A ViewGroup meant to contain a single child view

ScrollView	A FrameLayout that is designed to let the user scroll through the content in the view
ConstraintLayout	This is a newer viewgroup, it positions views in a flexible way

Android Layouts (ViewGroup)

Android **Layout** is used to define the user interface which holds the UI controls or widgets that will appear on the screen of an android application or activity.

Generally, every application is combination of **View** and **Viewgroup**. As we know, an android application contains a large number of activities and we can say each activity is one page of the application. So, each activity contains multiple user interface components and those components are the instances of the View and ViewGroup.

A **View** is defined as the user interface which is used to create an interactive UI component such as TextView, EditText, Radio Button etc. and it responsible for event handling and drawing.

A **ViewGroup** act as a base class for layouts and layouts parameters which hold other Views and ViewGroups and to define the layout properties.

The android framework will allow us to use UI elements or widgets in two ways:

- Use UI elements in XML file
- Create elements in Kotlin file dynamically

Type of Android Layout

Linear Layout: Linear Layout is a ViewGroup subclass, used to provide child view elements one by one either in a particular direction either horizontally or vertically based on the orientation property.

Relative Layout: Relative Layout is a ViewGroup subclass, used to specify the position of child View elements relative to each other like (A to the right of B) or relative to the parent (fix to the top of parent).

Constraint Layout: Constraint Layout is a ViewGroup subclass, used to specify the position of a layout constraints for every child view relative to other views present. A constraint layout is similar to a Relative Layout, but having more power.

Frame Layout: It is used to specify the position of View elements it contains on the top of each other to display only single view inside the Frame Layout.

Table Layout: It is used to display the child View elements in rows and columns.

Web View: WebView is a browser which is used to display the web pages in our activity layout.

List View: ListView is a ViewGroup, used to display scrollable list of items in single column.

Grid View: GridView is a ViewGroup which is used to display scrollable list of items in grid view of rows and columns.

ScrollView: ScrollView is a kind of layout that is useful to add vertical or horizontal scroll bars to the content which is larger than the actual size of layouts such as Linear Layout, Relative Layout, Frame Layout etc. The Android ScrollView can hold only one direct child.

Android Intents

An intent is to perform an action on the screen. It is mostly used to start activity, send broadcast receiver, start services and send message between two activities.

Intent is a messaging object which passes between components like services, content providers, activities etc.

Some of the general functions of intent are:

- Start service
- Launch activity
- Display web page
- Display contact list
- Message broadcasting

Methods are used to deliver intents to different components:

- context.startActivity ( ) – This is to launch a new activity or get an existing activity to be action.
- context.startService ( ) – This is to start a new service or deliver instructions for an existence service.
- context.sendBroadcast ( ) – This is to deliver the message to broadcast receivers.

Intent Classification

Implicit Intent:

Implicit intent is when the target component is not defined in the intent and the android system has to evaluate the registered component on the intent data.

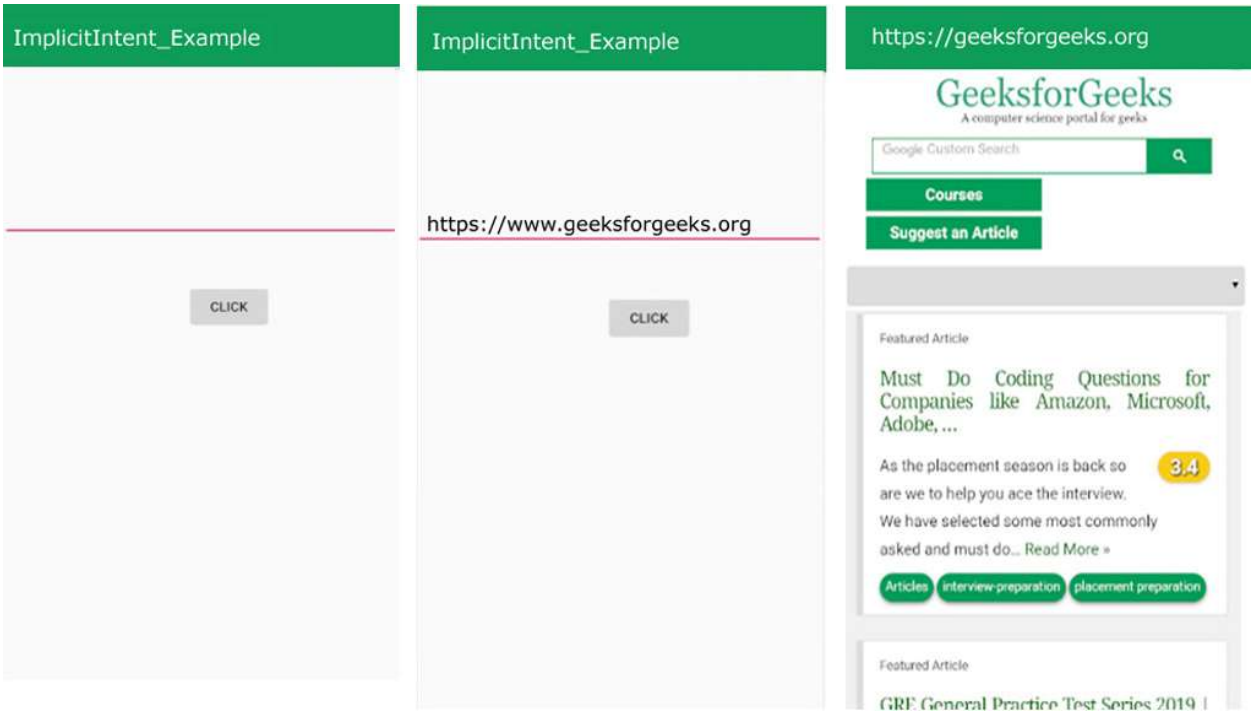
Using implicit intent, component can't be specifying. An action to be performed is declared by implicit intent. Then android operating system will filter out component which will response to the action.

Implicit Intent is used whenever you are performing an action. For example, send email, SMS, dial number or you can use a Uri to specify the data type. For example:

```
Intent i = new Intent(ACTION_VIEW,Uri.parse("<a href="http://www.edureka.co">http://www.edureka.co</a>"));

startActivity(i);
```

Example

In the below example, no component is specified, instead an action is performed i.e., a webpage is going to be opened. As you type the name of your desired webpage and click on 'CLICK' button. Your webpage is opened.



Explicit Intent:

Explicit intent is when an application defines the target component directly in the intent.

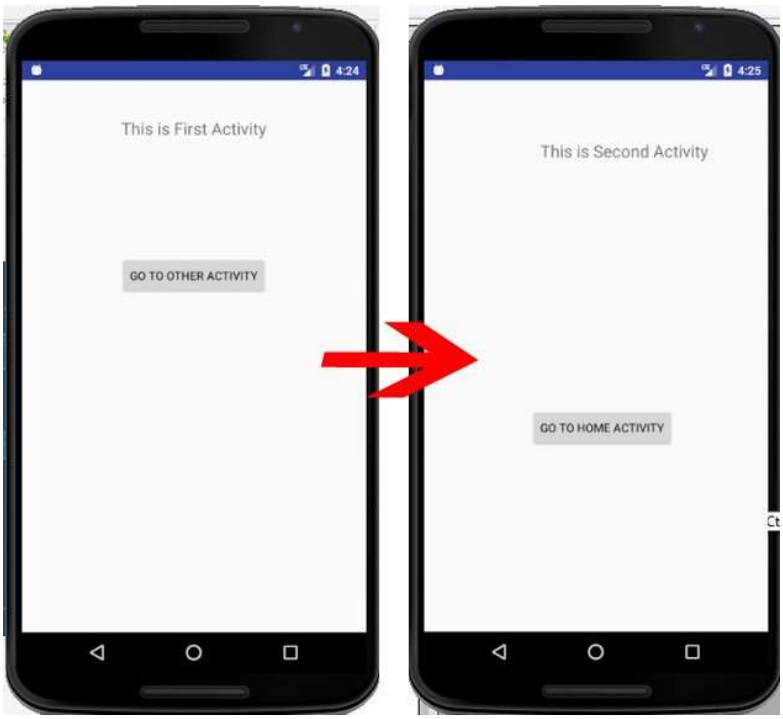
Explicit, on the other hand, helps you to switch from one activity to another activity (often known as the target activity). It is also used to pass data using putExtra method and retrieved by other activity by getIntent ( ).getExtras ( ) methods.

When you know about the target activity, and the task is performed is known as explicit intent. Using explicit intent any other component can be specified. In other words, targeted component is specified by explicit intent. So, only the specified target component will be invoked.

```
Intent i = new Intent(this, Activitytwo.class); #ActivityTwo is the target component

i.putExtra("Value1","This is ActivityTwo");

i.putExtra("Value2","This Value two for ActivityTwo");

startactivity(i);
```

Example

In the below example, there are two activities (First Activity, Second Activity). When you click on 'GO TO OTHER ACTIVITY' button in the first activity, then you move to second activity. When you click 'GO TO HOME ACTIVITY' button in the second activity, then you move to the first activity. This is getting done through explicit intent.



Intent Filter

An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with certain kind of intent.

Main Points:

- Implicit intent uses the intent filter to serve the user request.
- The intent filter specifies the types of intents that an activity, service or broadcast receiver can respond.
- Intent filters are declared in the Android manifest file.
- Intent filter must contain <action>

Most of the intent filter are described by its,

- <action>
- <category>
- <data>

What's the difference between the intent and intent filter in Android?

An intent is an object that can hold the OS or other app activity and its data in Uri form. It is started using startActivity (intent-obj) … whereas IntentFilter can fetch activity information on OS or other app activities.

Complex UI Design (Dynamic UI Design)

ListView

Android ListView is a ViewGroup which is used to display the list of items in multiple rows and contains an adapter which automatically inserts the items into the list.

The main purpose of the adapter is to fetch data from an array or database and insert each item that placed into the list for the desired result. So, it is main source to pull data from **string.xml** file which contains all the required strings in Kotlin or XML files.

Android Adapter

Adapter holds the data fetched from an array and iterates through each item in data set and generates the respective views for each item of the list. So, we can say it act as an intermediate between the data sources and adapter views such as ListView, GridView.

Different types of Adapter

ArrayAdapter

It always accepts an array or List as input. We can store the list items in the strings.xml file also.

CursorAdapter

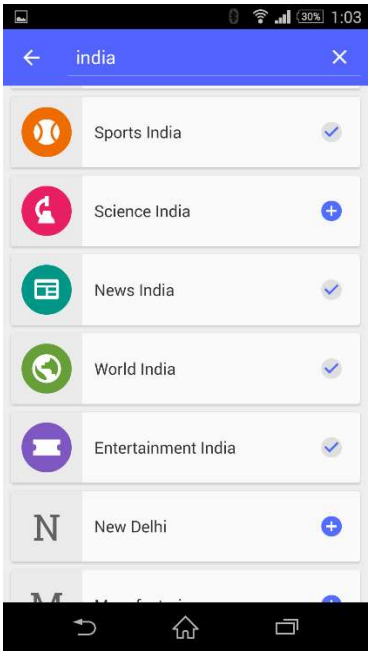
It always accepts an instance of cursor as an input.

SimpleAdapter

It mainly accepts a static data defined in the resources like array or database.

BaseAdapter

It is a generic implementation for all three adapter types and it can be used in the views according to our requirements.



RecyclerView

The RecyclerView is a widget that is more flexible and advanced version of GridView and ListView.

It is a container for displaying large datasets which can be scrolled efficiently by maintaining limited number of views. You can use RecyclerView widget when you have data collections whose elements change at runtime depend on network event or user interaction.

It has been created to make possible construction of any lists with XML layouts as an item which can be customized vastly while improving on the efficiency of ListView and GridView. This improvement is achieved by recycling the views which are out of the visibility of the user.

For example, if a user scrolled down to a position where items 4 and 5 are visible; items 1, 2 and 3 would be cleared from the memory to reduce memory consumption.

Implementation

To implement a basic RecyclerView three sub parts are needed to be constructed which offer the user the degree of control they require in making varying designs of their choice.

- **The Card Layout**: The card layout is an XML layout which will be treated as an item for the list created by the RecyclerView.
- **The ViewHolder**: The ViewHolder is a Java class that stores the reference to the card layout views that have to be dynamically modified during the execution of the program by a list of data obtained either by online databases or added in some other way.
- **The Data Class**: The Data class is a custom java class that acts as a structure for holding the information for every item of the RecyclerView.

To click on recycler item

To click on item of recycler view pass the instance of click interface in constructor of adapter.

The Adapter

The adapter is the main code responsible for RecyclerView. It holds all the important methods dealing with the implementation of RecyclerView.

The basic methods for a successful implementation are:

- **onCreateViewHolder**: which deals with the inflation of the card layout as an item for the RecyclerView.

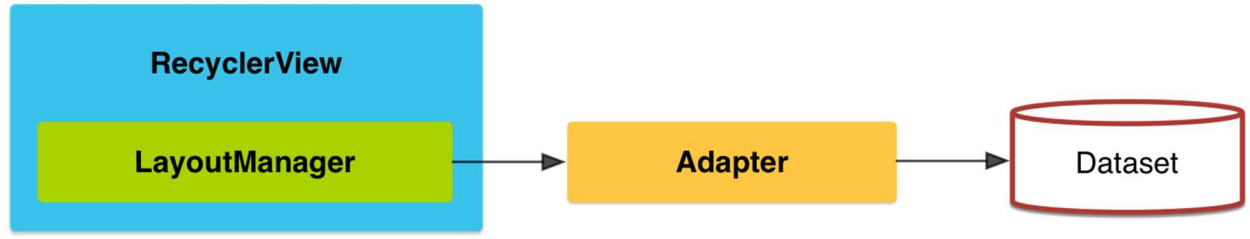
- **onBindViewHolder**: which deals with the setting of different data and methods related to clicks on particular items of the RecyclerView.

- **getItemCount**: which returns the length of the RecyclerView.

- **onAttachedToRecyclerView**: which attaches the adapter to the RecyclerView.

If you want to use a RecyclerView, you will need to work with the following:

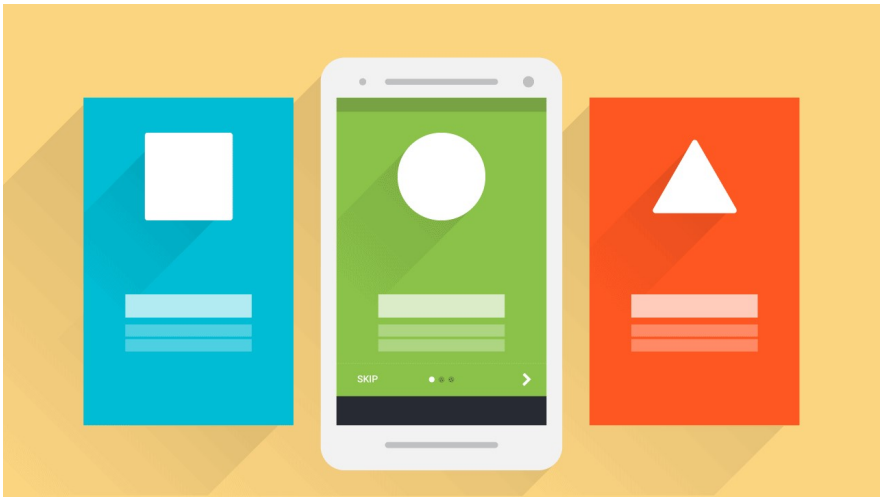
- *RecyclerView.Adapter*: To handle the data collection and bind it to the view
- *LayoutManager*: Helps in positioning the items
- *ItemAnimator*: Helps with animating the items for common operations such as Addition or Removal of item.



ViewPager

ViewPager in Android allows the user to flip left and right through pages of data or between fragments.

Android ViewPager widget is found in the support library and it allows the user to swipe left or right to see an entirely new screen.



Android Spinner

Spinner provide a quick way to select one value from a set.

In the default state, a spinner shows its currently selected value.

Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one.

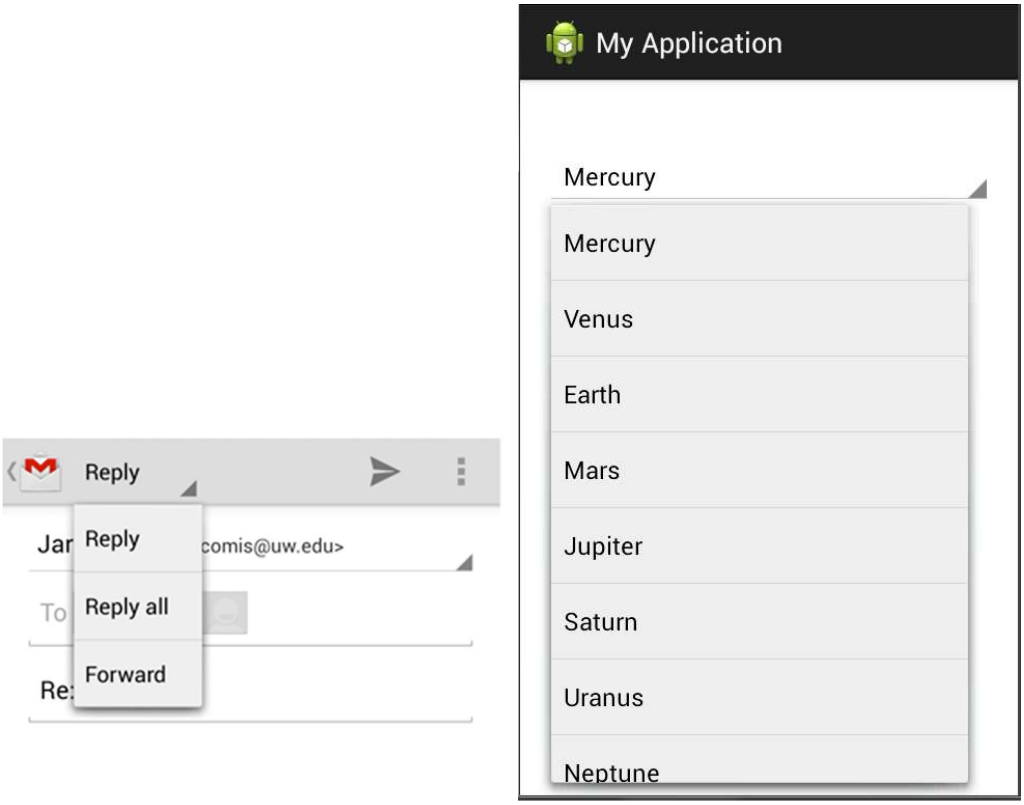
You can add a spinner to your layout with the Spinner object and to display the value we need to use ArrayAdapter class.

Key classes of Spinner are:

- *Spinner*
- *SpinnerAdapter*
- *AdapterView.OnItemSelectedListener*

Example:

When you are using Gmail application you would get drop down menu as shown below, you need to select an item from a drop-down menu.



CustomView

Sometimes you want to show a certain type of data and there is already a suitable view in the basic widget set. But if you want UI customization or a different user interaction, you may need to extend a widget.

Suppose that there was no Button widget in the basic widget set in Android SDK and you want to make one. You would extend the TextView class to get all the capabilities related to the text like setting text, text color, text size, text style and so on. Then you will start your customization work, to give your new widget the look and feel of a button. This is what happens in the Android SDK the Button class extends the TextView class.

Canvas & Bitmap & Paint

- Canvas is a class in Android that performs 2D drawing of different objects onto the screen. It is basically, an empty space to draw onto.
- Everything that is drawn in Android is a Bitmap. We can create a Bitmap instance, either by using the Bitmap class which has methods that allows us to manipulate pixels in the 2D coordinate system, or we can create a Bitmap from an image or a file or a resource by using the BitmapFactory class.
- The Paint class holds the style and color information about how to draw geometrics, texts and bitmaps.
- The Canvas class holds "draw" calls. To draw something, you need 4 basic components: A Bitmap to hold the pixels, a Canvas to host the draw calls (writing into the bitmap), a drawing primitive (e.g., Rect, Path, text, Bitmap), and a Paint (to describe the colors and styles for the drawing).

UI Resources

Resources are the additional files and static content that your code uses, such as Bitmaps, layout definitions, user interface strings, animation instructions and more.

Drawables Resources

A drawable resource is a general concept for a graphic that can be drawn to the screen. Drawables are used to define shapes, colors, borders, gradients etc. which can then be applied to views within an activity.

This is typically used for customizing the view graphics that are displayed within a particular view or context. Drawables tend to be defined in XML and can then be applied to a view via XML or Java or Kotlin.

There are at least 17 types of drawables but there are five that are most important to understand:

- Shape Drawables – Defines a shape with properties such as stroke, fill and padding
- StateList Drawables – Defines a list of drawables to use for different states
- LayerList Drawables – Defines a list of drawables grouped together into a composite result
- NinePatch Drawables – A PNG file with stretchable regions to allow proper resizing
- Vector Drawables – Defines complex XML-based vector images

String Resources

Android String Resource provides text strings for application. We have an option to format text and style it as well.

We have three types resources:

- String Array: It is an XML resource which provides an array of strings
- String: This is an XML resource which provides a single string
- Quantity Strings: It is an XML resource. It carries the strings for pluralization

Style Resources

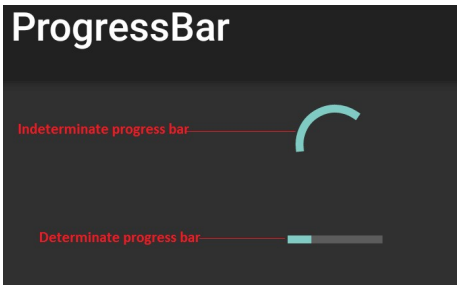
Style resources as the name suggests it is going to define the format and look of user interface.

An individual view can have a specific style. An entire activity or an application can be stylized by Manifest file. It is nothing but a resource which has to be referenced properly.

Support User Interface

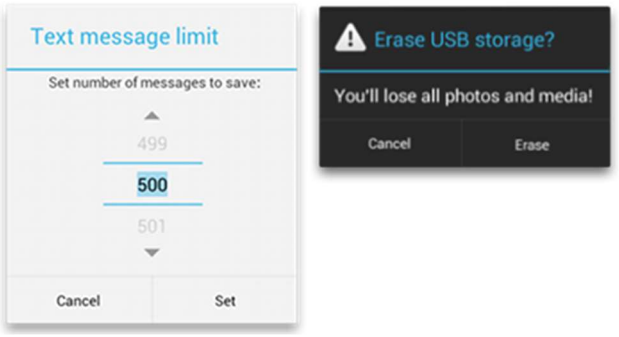
ProgressBar

- In Android, ProgressBar is used to display the status of work being done like analyzing status of work or downloading a file etc.
- In Android, by default a ProgressBar will be displayed as a spinning wheel but if we want it to be displayed as a horizontal bar then we need to use style attribute as horizontal.



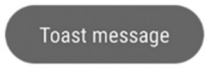
Dialogs

- A dialog is a small window that prompts the user to make a decision or enter additional information.
- A dialog does not fill the screen and is normally used for modal events that requires users to take an action before they can proceed.
- The Dialog class is the base class for dialogs.

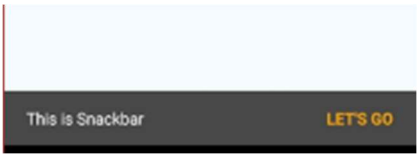


Toast & SnackBar

Toast is an Android UI component that is used to show message or notification that does not require any user action. It is independent to the activity in which it is being shown and disappears automatically after the set duration.



SnackBar is an Android material design UI component. It is used to show popup message to user that requires some user action. It can disappear automatically after set time or can be dismissed by the user.



Difference between Toast and SnackBar

Toast	SnackBar
Toast is an Android UI component present since API 1	It was later introduced with Material Design in API 23
It is not linked to an activity	It is linked with parent activity
User input can't be taken	User input can be taken
It can't be shown indefinitely	It can be shown for indefinite duration
It can't be closed by user action	It can be closed by user action
It is mostly used to show feedback message	It is used show message that need user action
It is used for showing info messages to user	It is used for showing warning/info type messages to user that needs attention
It can't be closed by swiping	It can be closed by swipe

Fragments

A fragment is a piece of an activity which enable more modular activity design. A fragment encapsulates functionality so that it is easier to reuse within activities and layouts

Android devices exists in a variety of screen sizes and densities. Fragments simplify the reuse of components in different layouts and their logic. You can build single-pane layouts for handsets (phones) and multi-pane layouts for tablets. You can also use fragments also to support different layout for landscape and portrait orientation on a smartphone.

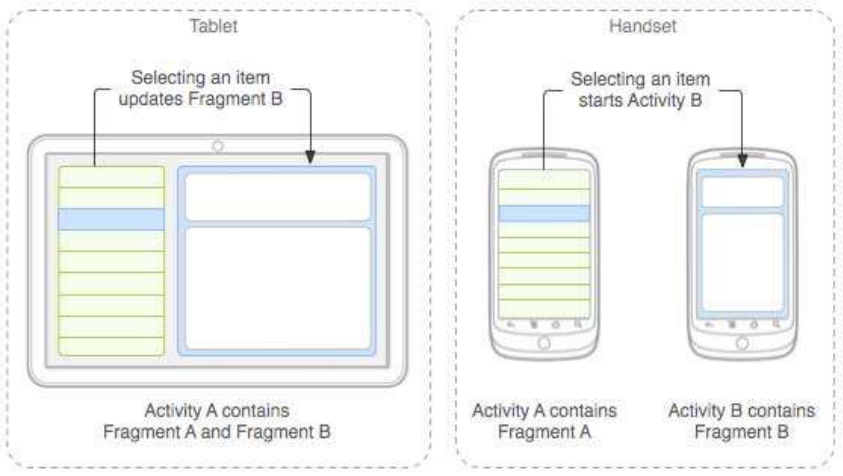
Fragment is the part of activity; it is also known as sub-activity. There can be more than one fragment in an activity.

Following are important points about fragment:

- Fragment represent multiple screens inside one activity.
- A fragment has its own layout and its own behaviour with its own life cycle call-backs.
- You can add or remove fragments in an activity while the activity is running.
- You can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- Fragment lifecycle is closely related to the life cycle of its host activity which means when the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behaviour that has no user interface component.
- Fragments were added to the Android API in Honeycomb version of Android which is API **version 11.**
- You can create fragments by extending **Fragment class** and you can insert a fragment into your activity layout by declaring the fragment in the activity's layout file, as a <fragment> element.

Example:

This is a typical example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design.



The application can embed two fragments in Activity A, when running on a tablet sized device. However, on a handset-sized screen, there's not enough room for both fragments, so activity A includes only the fragment for the list of articles, and when the user selects an article, it starts Activity B, which includes the second fragment to read the article.

Types of Android Fragments

Basically, fragments are divided as three stages:

Single frame fragments

Single frame fragments are using for hand hold devices like mobiles, here we can show only one fragment as a view.

Single frame fragment is designed for small screen devices such as mobiles and it should be above Android 3.0 version.

List fragments

Fragments having special list view is called as list fragment.

This fragment is used to display a ListView from which the user can select the desired sub activity. The menu drawer of apps like Gmail is the best example of this kind of fragment.

Fragment Transaction

Using with fragment transaction. We can move one fragment to another fragment.

Introduced in **KitKat** version.

The transition framework provides a convenient API for animating between different UI states in an application. The framework is built around two key concepts:

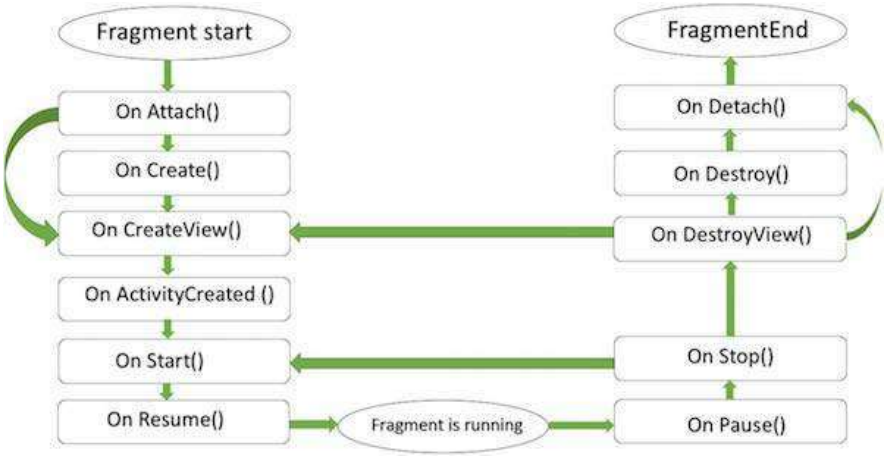
- Scenes
- Transitions

A **scene** defines a given state of an applications UI, whereas a **transition** defines the animated change between two scenes.

This kind of fragments supports the transition from one fragment to another at run time. Users can switch between multiple fragments like switching tabs.

Fragment Lifecycle

Each fragment has its own lifecycle but due to the connection with the activity it belongs to, the fragment lifecycle is influenced by the activity's lifecycle.



Methods of the android Fragment

Methods	Description
onAttach ()	The very first method to be called when the fragment has been associated with the activity. This method executes only once during the lifetime of a fragment.
onCreate ()	This method initializes the fragment by adding all the required attributes and components.
onCreateView ()	System calls this method to create the user interface of the fragment. The root of the fragment's layout is returned as the View component by this method to draw the UI.
onActivityCreated ()	It indicates that the activity has been created in which the fragment exists. View hierarchy of the fragment also instantiated before this function call.
onStart ()	The system invokes this method to make the fragment visible on the user's device.
onResume ()	This method is called to make the visible fragment interactive.
onPause ()	It indicates that the user is leaving the fragment. System calls this method to commit the changes made to the fragment.
onStop ()	Method to terminate the functioning and visibility of fragment from the user's screen.
onDestroyView ()	System calls this method to clean up all kinds of resources as well as view hierarchy associated with the fragment.
onDestroy ()	It is called to perform the final cleanup of fragment's state and its lifecycle.
onDetach ()	The system executes this method to disassociate the fragment from its host activity.

Handling the Fragment Lifecycle

A fragment exists in three states:

- **Resumed**: The fragment is visible in the running activity.

- **Paused**: Another activity is in the foreground and has focus, but the activity in which this fragment lives is still visible (the foreground activity is partially transparent or doesn't cover the entire screen).

- **Stopped**: The fragment is not visible. Either the host activity has been stopped or the fragment has been removed from the activity but added to the back stack. A stopped fragment is still alive (all state and member information is retained by the system). However, it is no longer visible to the user and will be killed if the activity is killed.

Fragment Manager

The FragmentManager class is responsible to make interaction between fragment objects.

A FragmentManager manages Fragments in android, specifically it handles transactions between fragments. A transaction is a way to add, replace or remove fragments.

This class was deprecated in API level 28.

Example of Android Fragment

Fragments are always embedded in activities i.e.; they are added to the layout of activity in which they reside. Multiple fragments can be added to one activity. This task can be carried out in 2 ways:

- **Statically**: Explicitly mention the fragment in the XML file of the activity. This type of fragment can not be replaced during the run time.

- **Dynamically**: FragmentManager is used to embed fragments with activities that enable the addition, deletion or replacement of fragments at run time.

Almost all android app uses dynamic addition of fragments as it improves the user experience.

How to use Fragments?

This involves number of simple steps to create Fragments:

- First of all, decide how many fragments you want to use in an activity. For example, let's we want to use two fragments to handle landscape and portrait modes of the device.
- Next based on number of fragments, create classes which will extend the Fragment class. The Fragment class has above mentioned call-back functions. You can override any of the functions based on your requirements.
- Corresponding to each fragment, you will need to create layouts files in XML file. These files will have layout for the defined fragments.
- Finally modify activity file to define the actual logic of replacing fragments based on your requirement.

Android BottomSheet

The Bottom Sheet is seen in many of the applications such as Google Drive, Google Maps and most of the applications used the Bottom Sheet to display the data inside the application.

Bottom Sheet is a component of the Android design support library that is used to display different actions in an expandable UI design. It is an expandable widget that opens from the bottom of the android device on clicking on a specific Button or View.

Type of Bottom Sheet

There are two different types of Bottom Sheet as,

- Persistent Bottom Sheet
- Model Bottom Sheet

Persistent Bottom Sheet

A Persistent Bottom Sheet will be displayed on the bottom of the screen in our Android application. This sheet will be displayed at the bottom of the screen making some portion of the content visible.

The elevation of this bottom sheet is the same as that of the app. Users can be able to navigate to both the app along with the bottom sheet at a time.

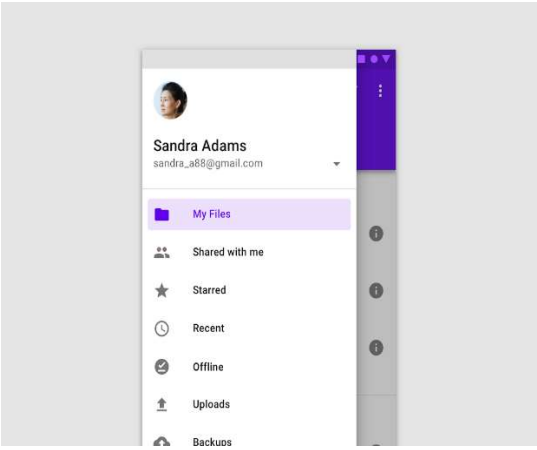
Modal Bottom Sheet

Modal Bottom Sheet will also be displayed on the bottom of the screen, but the difference is the user will not be able to use the app's background content when the bottom sheet is open.

This type of bottom sheet is having an elevation slightly higher than that of the app. When this bottom sheet is open, the user will not able to access the app's content.

Android Navigation Drawer

- The Navigation Drawer is UI panel that shows your app's main navigation menu. It is also one of the important UI elemnts, which provides actions preferable to the users like example changing user profile, changing settings of the application etc.
- The navigation drawer slides in from the left and contains the navigation destinations for the app.
- The user can view the navigation drawer when the user swipes a finger from the left edge of the activity. They can also find it from the home activity by tapping the app icon in the action bar. The drawer icon is displayed on all top-level destinations that use a DrawerLayout.



Android Storage

Android provides many kinds of storage for applications to store their data. These storage applications are

- Shared preferences
- Internal and external storage
- SQLite storage
- Storage via network connection.

To store the data permanently (until deleted) for future reference, we use some kind of storage in Android. These storage systems are called Android Storage System.

Internal Storage

- When you install some application on your phone, then the Android system will provide you with some kind of private internal storage where the application can store its private data. This data can't be accessed by any other application. When you uninstall the application, then all the data related to that application will be deleted.
- On order to save some file in the storage, you have to get the file from the internal directory. In order to do so, you can use the *getFilesDir ( )* method or *getCacheDir ( )* method.
- The *getFilesDir ( )* method returns the absolute path to the directory on the file system where files are created. The getCacheDir ( ) returns the absolute path to the application specific cache directory on the file system.

When to use Internal Storage?

- When you want to have some private data for your application, then you can use internal storage.
- If your application is storing some data that can be used by other application, then you shouldn't use internal storage because when you uninstall the app, all your data will be deleted and other apps will never get the data.

Example:

- If your application is downloading some PDF or storing image or video that can be used by other applications, then you shouldn't use the internal storage.

External Storage

- Most of the Android devices have very less internal storage. So, we use some kind of external storage to store our data. These storage unit are accessed by everyone i.e.; it can be accessed by all the applications in your device. Also, you can access the storage just by connecting the mobile device to some PC.
- In order to get access to the external storage, you have to take the READ_EXTERNAL_STORAGE permission from the user. So, any application having this permission can access your application's data.

When to use External Storage?

- If the data that is being stored by your application can be used by other application, then you can use external storage. Also, if the file is stored by your application is very large as in case of video, then you can store the file in the external storage. If you want the data even after uninstalling the application, then you can use external storage.
- If your app is storing some private data that is of no use when the app is not there on the phone, then you should avoid using the external storage.

Database: RoomDB & SQLite

Database are organized collection of data that are stored for future reference. You can store any kind of data in your Database by using some Database Management System.

All you need to do is just create the database and do all the operations i.e., insertion, deletion, searching etc. with the help of one query. You will pass the query and the database will return the desired output from the database.

SQLite database is one such example of a database in Android.

When to use Database?

- When you want to store some structured data, then you can use a database. You can store any kind of data in a database.
- So, if the data size is very big and you want to access the data in a very easy way, then you can use a database and store your data in a structured format.

Shared Storage

- Every application in the device has some private storage in the internal memory and you can find this in android/data/your_package_name directory. Apart from this internal storage, the rest of the storage is called the Shared Storage i.e., every application with the storage permission can access this part of the memory. This includes media collections and other files of different applications.
- In applications that store some data that is being used by some other application, we can use shared storage. But in many cases, other applications only need a small part of the shared storage and rest of the storage is of no use for other applications. So why to provide them access to full shared storage? To solve this, the concept of Scoped Storage is introduced in Android 10.

When to use Shared Storage?

- When the data stored by your application can be accessed by some other application then you can use shared storage. For example, the image stored by your app can be viewed or edited by some other application present in your phone.
- Similarly, the PDF stored or generated by your application can be viewed from another application also. So, in these cases, you can use shared storage.

Shared Preferences

If you have a small amount of data to store and you don't want to use the internal storage, then you can use the shared preferences. Shared preferences are used to store the data in the form of key-value that is you will be having one key and based on that key, the corresponding data or value will be stored.

The data stored in the shared preferences will remain with the application until the application is present on your mobile phone. If you uninstall the application, then all the shared preferences will be removed from the device.

When to use Shared Preferences?

- When the data you want to store is very small, then you can use the shared preference in your application. It is recommended not to store more than 100kb of data in shared preferences.
- Also, if you want to store small and private data, then you can use the shared preferences in Android.

Android File System

File System (fs) controls how data is stored and retrieved.

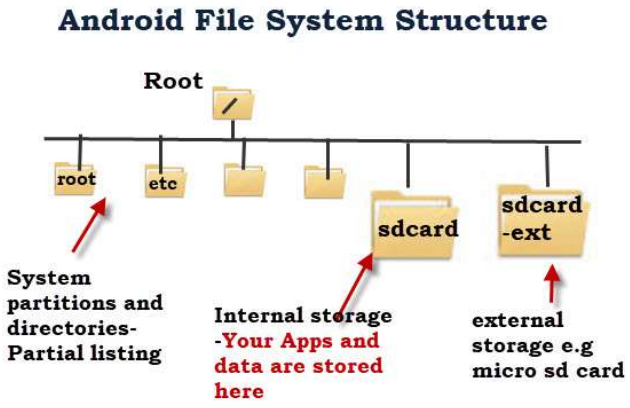
Usually, the file system used in Android is YAFFS (Yet Another Flash File System).

Since, Android is a Linux-based operating system, your handset features a Linux-esque file system structure. This system consists of six main partitions which form the structure of the entire file storage.

The six partitions are:

- Boot
- System
- Recovery
- Data
- Cache
- Misc

MicroSD cards also count as their own memory partition.



RoomDB

Room is a persistence library that provides an abstraction layer over the SQLite database to allow a more robust database. With the help of room, we can easily create the database and perform CRUD operations very easily.

Room is now considered as a better approach for data persistence than SQLite database. It makes it easier to work with SQLite database objects in your app, decreasing the amount of boilerplate code and verifying SQL queries at compile time.

Why uses Room?

- Compile-time verification of SQL queries. Each @Query and @Entity is checked at the compile time, that preserves your app from crash issues at runtime and not only it checks the only syntax, but also missing tables.
- Boilerplate code
- Easily integrated with other Architecture components like LiveData.

Major problems with SQLite usage are:

- There is no compile-time verification of raw SQL queries. For example, if you write a SQL query with a wrong column name that does not exist in real database then it will give exception during run time and you cannot capture this issue during compile time.
- As your schema changes, you need to update the affected SQL queries manually. This process can be time-consuming and error-prone.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects (POJO).

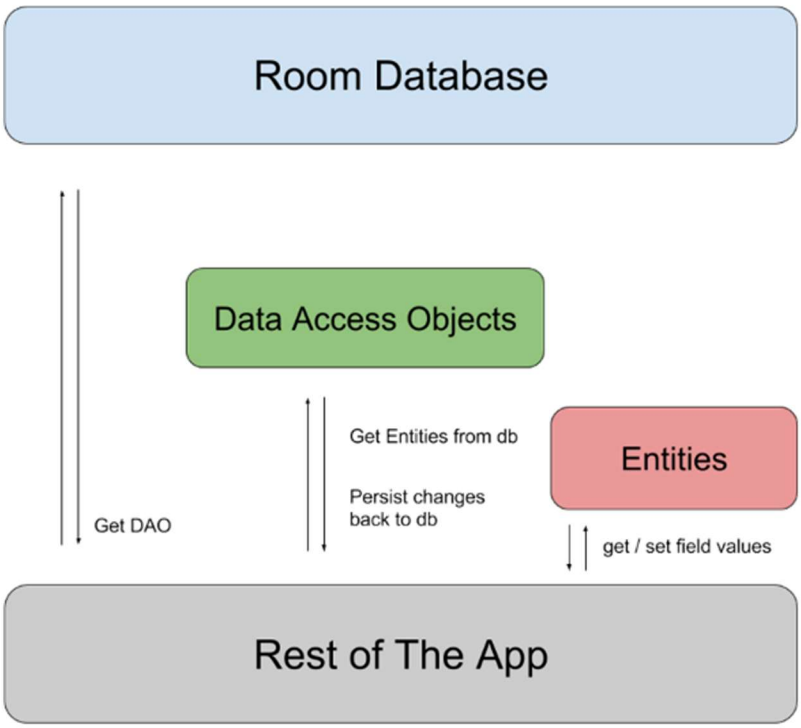
Room vs SQLite

Room is an ORM, Object Relational Mapping Library. In other words, Room will map our database objects to Java objects. Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

Difference between SQLite and Room persistence library:

- In the case of SQLite, there is no compile-time verification of raw SQLite queries. But in Room, there is SQL validation at compile time.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But Room maps our database objects to Java Object without boilerplate code.
- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
- Room is built to work with LiveData and RxJava for data observation, while SQLite does not.

Components of Room DB



Room has three main components of Room DB:

- **Entity**
- **Dao**
- **Database**

Entity

Represents a table within the database. Room creates a table for each class that has @Entity annotation, the fields in the class correspond to columns in the table. Therefore, the entity tends to be small model classes that don't contain any logic.

Entities annotations:

@Entity – Every model class with this annotation will have a mapping table in DB.

- ForeignKeys: names of foreign keys
- Indices: list of indicates on the table
- primaryKeys: names of entity primary keys
- tableName

@PrimaryKey – as its name indicates, this annotation points the primary key of the entity. autoGenerate – if set to true, then SQLite will be generating a unique id for the column.

```
@PrimaryKey (autoGenerate = true)
```

@ColumnInfo – allows specifying custom information about column.

```
@ColumnInfo (name = "column_name")
```

@Ignore – field will not be persisted by Room.

@Embeded – nested fields can be referenced directly in the SQL queries.

Dao (Data Access Objects)

DAOs are responsible for defining the methods that access the database. In the initial SQLite, we use the `Cursor` objects. With Room, we don't need all the `Cursor` related code and can simply define our queries using annotations in the `DAO` class.

Database

- Contains the database holder and serves as the main access point for the underlying connection to your app's persisted, relational data.
- To create a database, we need to define an abstract class that extends `RoomDatabase`. This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.
- The class that's annotated with `@Database` should satisfy the following conditions:
- Be an abstract class that extends `RoomDatabase`.
- Include the list of entities associated with the database within the annotation.
- Contain an abstract method that has 0 arguments and returns the class that is connected with `@Dao`.
- At runtime, you can acquire an instance of `Database` by calling `Room.databaseBuilder ( )` or `Room.inMemoryDatabaseBuilder ( )`.

Implementation of Room

Step 1: add the Gradle dependencies

To add it to your project, open the project level `build.gradle` file and add the highlight line as shown below:

```
allprojects {
    repositories {
        jcenter()
        maven { url 'https://maven.google.com' }
    }
}
```

Open the `build.gradle` file for your app or module and add dependencies:

```
implementation "android.arch.persistence.room:runtime:1.0.0"

annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
```

Step 2: Create a Model Class

Room creates a table for each class annotated with `@Entity`; the fields in the class correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don't contain any logic. Our `Person` class represents the model for the data in the database. So, let's update it to tell Room that it should create a table based on this class.

Annotate the class with `@Entity` and use the `tableName` property to set the name of the table.

Set the primary key by adding the `@PrimaryKey` annotation to the correct fields – in our case, this is ID of the user.

Set the name of the columns for the class fields using the `@ColumnInfo (name = "column_name")` annotation. Feel free to skip this step if you field already have the correct column name.

If multiple constructors are suitable, add the `@Ignore` annotation to tell Room which should be used and which not.

```
@Entity(tableName = "person")
public class Person {
    @PrimaryKey(autoGenerate = true)
    private int id;
    @ColumnInfo(name = "name")
    private String name;
    @ColumnInfo(name = "city")
    private String city;

    public Person(int id, String name, String city) {
        this.id = id;
        this.name = name;
        this.city = city;
    }

    @Ignore
    public Person(String name, String city) {
        this.name = name;
        this.city = city;
    }

}
```

Step 3: Create Data Access Objects (DAOs)

DAOs are responsible for defining the methods that access the database.

To create a DAO, we need to create an interface and annotated with `@Dao`.

```java
@Dao
public interface PersonDao {
    @Query("Select * from person")
    List<Person> getPersonList();
    @Insert
    void insertPerson(Person person);
    @Update
    void updatePerson(Person person);
    @Delete
    void deletePerson(Person person);
}
```

Step 4: Create the database

To create a database, we need to define an abstract class that extends `RoomDatabase`. This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.

```java
@Database(entities = Person.class, exportSchema = false, version = 1)
public abstract class PersonDatabase extends RoomDatabase {
    private static final String DB_NAME = "person_db";
    private static PersonDatabase instance;

    public static synchronized PersonDatabase getInstance(Context context) {
        if (instance == null) {
            instance = Room.databaseBuilder(context.getApplicationContext(), PersonDatabase.class,
DB_NAME)
                    .fallbackToDestructiveMigration()
                    .build();
        }
        return instance;
    }

    public abstract PersonDao personDao();
}
```

Step 5: Managing Data

To manage the data, first of all, we need to create an instance of the database. Then we can insert delete and update the database.

Make sure all the operation should execute on a different thread.

Query:

```
PersonDatabase appDb = PersonDatabase.getInstance(this);


appDb.personDao().getPersonList();
```

Insert:

```
PersonDatabase appDb = PersonDatabase.getInstance(this);

Person person = new Person("Ashish","Noida");

appDb.personDao().insertPerson(person);
```

Delete:

```
PersonDatabase appDb = PersonDatabase.getInstance(this);

appDb.personDao().deletePerson(person);
```

Update:

```
PersonDatabase appDb = PersonDatabase.getInstance(this);

Person person = new Person("Ashish Rawat","Noida");

appDb.personDao().updatePerson(person);
```

Executing with Executor:

```
AppExecutors.getInstance().diskIO().execute(new Runnable() {
            @Override
            public void run() {
                final List<Person> persons = mDb.personDao().loadAllPersons();
                mAdapter.setTasks(persons);
            }
        });
```

Android Build

Gradle for Android Studio

In Android Studio, Gradle is used for buliding our android application projects, hence playing the role of a build system.

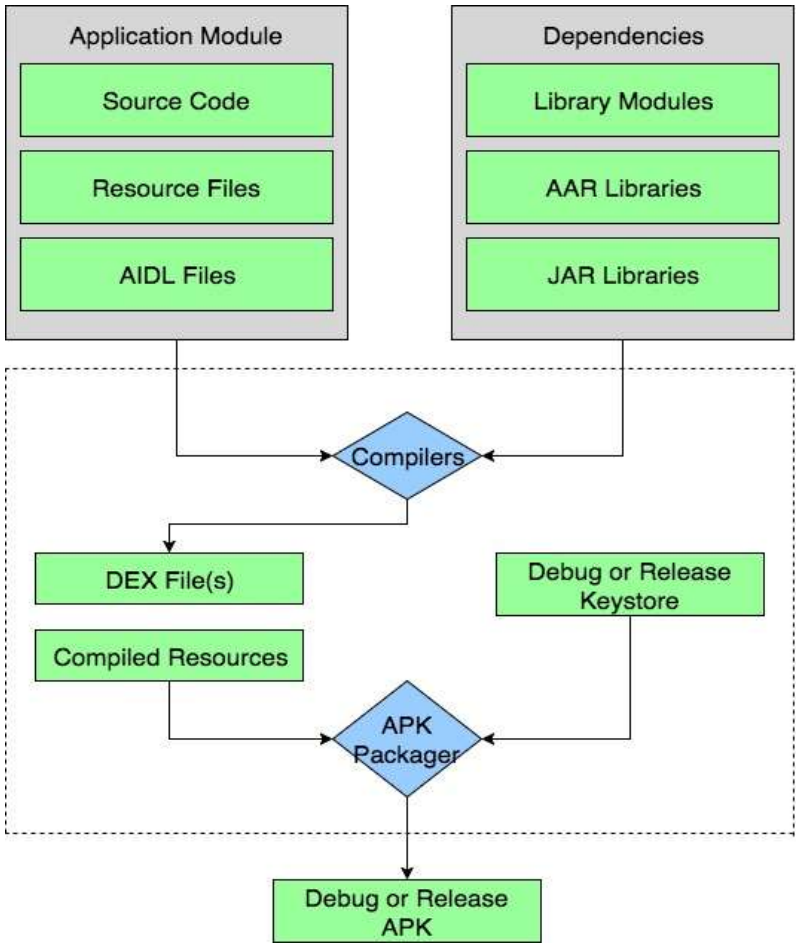
Gradle is a build system, which is responsible for code compilation, testing, deployment and conversion of the code into .dex files and hence runinng the app on the device.

As Andorid Studio comes with Gradle system pre-installed, there is no need to install additional runtime softwares to build our project. Whenever you click on Run button in android studio, a gradle task automatically triggers and starts building the project and after gradle completes its task, app starts running in AVD on in the connected device.

A build system like Gradle is not a compiler, linker etc, but it controls and supervises the operation of compilation, linking of files, running test cases, and eventually bundling the code into an apk file for your android application.

There are two build.gradle files for every android studio project of which, one is for Application and other is for project level (Module level) build files.

The build process works as shown in the below diagram,



In the build process, the compiler takes the source code, resources, external libraries JAR files and AndroidManifest.xml (which contains the meta-data about the application) and convert then into .dex (Dalvik Executable files), which includes bytecode. That bytecode is supported by all android devices to run your app.

Then APK Manager combines the .dex files and all other resources into single apk file. APK Packager signs debug or release apk using respective debug or release keystore.

Debug apk is generally used for testing purpose or we can say that it is used at development stage only. When your app is complete with desired features and you are ready to publish your application for external use then you require a Release apk signed using a release keystore.

setting.gradle

The setting.gradle (Gradle setting) file is used to specify all the modules used in your app.

build.gradle (Project level)

The Top level (module) build.gradle file is project level build file, which defines build configurations at project level. This file applies configurations to all the modules in android application project.

build.gradle (Application level)

The Application level build.gradle file is located in each module of the android project. This file includes your package name as applicationID, version name (apk version), version code, minimum and target sdk for a specific application module.

When you are including external libraries (not the jar files) then you need to mention it in app level gradle file to include them in your project as dependencies of the application.

If a certain application is developed in variations for individual modules like, Smart Phone, Tablet or TV then separate gradle files must to be created for all.

You can even start your gradle system through command line tool. Following commands are used for it.

- . /gradlew build – (build project)
- . /gradlew clean build – (build project complete scratch)
- . /gradlew clean build – (run the test
- . /gradlew wrapper – (to see all the available tasks)

Basically Gradle: Android Build Tool is

- What devices run your app?
- Compile to executable
- Dependency management
- App signing for Google Play
- Automated Tests

Android Studio Debugger

Debugging is the process of finding and fixing errors (bugs) or unexpected behaviour in your code. All code has bugs, from incorrect behaviour in your app, to behaviour that excessively consumes memory or network resources, to actual app freeing or crashing.

- Bugs can result for many reasons:
- Error in your design or implementation
- Android Framework limitations
- Missing requirements or assumptions for how the app should work
- Device limitations

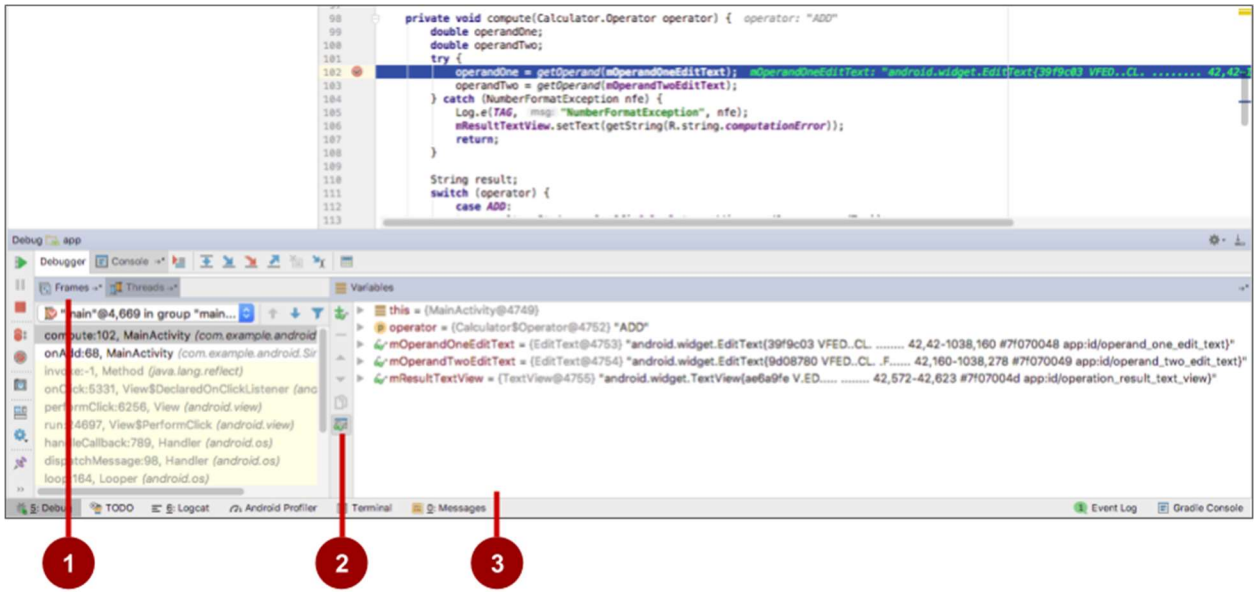
Use the debugging, testing and profiling capabilities in Android Studio to help you reproduce, find, and resolve all of these problems. Those capabilities include:

- The Logcat pane for log messages
- The Debugger pane for viewing frames, threads, and variables
- Debug mode for running apps with breakpoints
- Test Frameworks such as Junit or Espresso
- Dalvik Debug Monitor Server (DDMS), to track resource usage

Run your app in debug mode

Running an app in debug mode is similar to running the app. You can either run an app in debug mode, or attach the debugger to an already-running app.

To start debugging, click Debug in the toolbar. Android Studio builds an APK, signs it with a debug key, installs it on your selected device, then runs it and opens the Debug pane with the Debugger and Console tabs.



Android Threading

Android Thread Class

When an application is launched, Android creates its own Linux process. Beside this system, it creates a thread of execution for that application called the main thread or UI thread.

The main thread is nothing but a handler thread. The main thread is responsible for handling events from all over the app, like callbacks associated with the life-cycle information or callbacks from input events. It can also handle events from other apps.

Any block of code that needs to be run is pushed into a work queue and then serviced by the main thread. As the main thread does so much work, it's better to offer longer works to other threads so as not to disturb UI thread from its rendering duties. It's essential to avoid using the main thread to perform any operation that may end up keeping it blocked resulting in ANRs (application not responding).

Network operations or database calls or the loading of certain components are some examples that may cause the blocking of the main thread when they are being executed on main thread. They're executed synchronously, which means the UI will remain completely unresponsive until the task gets completed. To avoid this situation, they're usually executed in separate threads, which avoids blocking the UI while the tasks are being performed. This means they're executed asynchronously from the UI.

Android provides many ways of creating and managing threads, and there are many third-party libraries that make thread management a lot easier. Each thread class is intended for a specific purpose; however, picking the right one that suits our needs is very important.

The different thread classes available are:

- AsyncTask: Helps get work on/off the UI thread
- HandlerThread: Thread for callbacks
- ThreadPoolExecutor: Running lots of parallel work
- IntentService: Helps get intents off the UI thread

AsyncTask

AsyncTask enables the proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without the use of threads or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and doesn't constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods, it's highly recommended you use the various APIs provided by the java.util.concurrent package, such as Executor, ThreadPoolExecutor, and FutureTask.

When an asynchronous task is executed, the task goes through four steps:

onPreExecute(): Invoked on the UI thread before the task is executed. This step is normally used to do something before the task gets started — for instance, simply showing a progress dialog in the user interface.

doInBackground(Params...): Invoked on the background thread after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step, and it sends the result to the onPostExecute(). This step can also use publishProgress(..) to publish one or more units of progress.

onProgressUpdate(Progress...): Invoked on the UI thread after a call to publishProgress(..). This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.

onPostExecute(Result): Invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

A task can be canceled at any time by invoking cancel(boolean...). We need to do this by checking whether the task is canceled or running.

When to use AsyncTask

AsyncTask is the perfect solution for the short work that ends quickly which requires frequent UI updates.

However, the async task falls short if you need your deferred task to run beyond the lifetime of the activity/fragment. It's worth nothing even something as simple as screen rotation can cause the activity to be destroyed.

Order of Execution

By default, all created AsyncTasks will share the same thread and be executed in a sequential fashion from a single message queue. Synchronous execution affects individual tasks.

It we want tasks to execute parallelly, we can use THREAD_POOL_EXECUTOR.

HandlerThread

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to a Looper.

A Handler thread is a subclass of the normal Java thread class. A handler thread is a long-running thread that grabs work from the queue and operates on it. It's a combination of other Android primitives, namely:

Looper: Keeps the thread alive and holds the message queue

MessageQueue: Class holding the list of messages to be dispatched by a Looper

Handler: Allows us to send and process message objects associated with a thread's MessageQueue.

This means we can keep it running in the background and feed it with more and more packages of work sequentially one after the other until we quit it. Handler threads run outside of your activity's life cycle, so they need to be cleaned up properly or else you'll have thread leaks.

There are two main ways to create handler threads:

- Create a new Handler thread, and get the looper. Now, create a new handler by assigning the looper of the created handler thread and post your tasks on this handler.

- Extend the Handler thread by creating the *CustomHandlerThread* class. Then, create a handler to process the task. You'd take this approach if you knew the task you wanted to perform and just needed to pass in the parameters. An example would be like creating a custom *HandlerThread* class to download images or to perform networking tasks.

```
HandlerThread handlerThread = new HandlerThread("TesHandlerThread");
handlerThread.start();
Looper looper = handlerThread.getLooper();
Handler handler = new Handler(looper);
handler.post(new Runnable(){…});
```

When we create a handler thread, don't forget to set its priority because CPU can handle only a small number of threads in parallel, so setting the priority can help the system know the right way to schedule this work while other threads are fighting for attention.

NOTE: call *handerThread.quit ( )* when you're done with the background thread or on your activity's *onDestroy ( )* method.

We can post updates to the UI thread by using local broadcast or by creating a handler with the main looper.

```
Handler mainHandler = new Handler(context.getMainLooper());
mainHandler.post(myRunnable);
```

When to use handler threads?

Handler threads are the perfect solution for the long-running background work that doesn't need UI updates.

ThreadPoolExecutor

What is a thread pool?

A thread pool is basically a pool of threads waiting to be given a task. The tasks that are assigned to these threads will run in parallel. As the tasks execute in parallel, we may want to ensure our code is thread-safe. A thread pool mainly addresses two problems:

Improved performance when executing a large of asynchronous tasks due to reduce per-task overhead.

A means of bounding and managing resources (including threads) when executing a collection of tasks.

Example:

If we got 40BMP to decode, where each bitmap takes 4ms to decode, and if we do it on a single thread, it takes over 160ms to decode all bitmaps.

However, if we do it with 10 threads, each decodes four bitmaps. Thus, the time taken to decode these 40 bitmaps would be only 16ms.

The problem here is how to pass the work to each thread, how to schedule that work, and how to manage those threads. It's a very big problem. This is the place where **ThreadPoolExecutor** comes into play.

What is ThreadPoolExecutor?

A ThreadPoolExecutor is a class that extends **AbstractExecutorService**.

ThreadPoolExecutor takes care of all the threads.

It assigns tasks to threads

- It keeps them alive
- It terminates the threads

The way it works under the hood is that tasks to be run are maintained in a work queue. From the work queue, a task is assigned to a thread whenever a thread in the pool becomes free or available.

Runnable

It's an interface that's to be implemented by a class whose instances are intended to be executed by a thread. Simply put: it's a command or a task that can be executed. Its frequently used to run code in a different thread.

```
Runnable mRunnable = new Runnable() {
@Override public void run() {
// Do some work
}
};
```

Executor

An executor is an interface used to decouple task submission from task execution. Its an object that executes Runnable.

```
Executor mExecutor = Executors.newSingleThreadExecutor();

mExecutor.execute(mRunnable);
```

ExecutorService

An executor that manages asynchronous tasks.

```
ExecutorService mExecutorService = Executors.newFixedThreadPool(10);
mExecutorService.execute(mRunnable);
```

ThreadPoolExecutor

An **ExecutorService** that assigns tasks to a pool of threads.

More threads aren't always good because the CPU can only execute a certain number of threads in parallel. Once we exceed that number, the CPU has to make some expensive calculations to decide which thread should get assigned based on priority.

While creating the instance of **ThreadPoolExecutor**, we can specify the number of initial threads and the number of maximum threads. As the workload in the thread pool changes, it will scale the number of live threads to match.

It's usually recommended to allocate threads based on the number of available cores. This can be achieved by:

```
int NUMBER_OF_CORES = Runtime.getRuntime().availableProcessors();
```

Note:

This doesn't necessarily return the actual number of physical cores on the device. Its possible CPU may deactivate some cores to save battery, etc.

```
ThreadPoolExecutor(
int corePoolSize, // Initial pool size
int maximumPoolSize, // Max pool size
long keepAliveTime, // Time idle thread waits before terminating
TimeUnit unit // Sets the Time Unit for keepAliveTime
BlockingQueue<Runnable> workQueue) // Work Queue
```

What are these parameters?

corePoolSize

The minimum number of threads to keep in the pool. Initially, there are zero threads in the pool. But as tasks are added to the queue, new threads are created. If fewer than **corePoolSize** threads are running, the **Executor** always prefers adding a new thread rather than queuing.

maximumPoolSize

The maximum number of threads allowed in the pool. If this exceeds the corePoolSize and the current number of threads is > = corePoolSize, then new worker threads will be created only if the queue is full.

keepAliveTime

When the number of threads is greater than the core, the noncore threads (exceeds idle threads) will wait for a new task, and if they don't get one within the time defined by this parameter, they'll terminate.

unit

The unit of time for **keepAliveTime**.

workQueue

The task queue, which will only hold runnable tasks. It will have to be a **BlockingQueue**.

When to use ThreadPoolExecutor?

ThreadPoolExecutor is a powerful task-execution framework we can use when a large number of tasks need to be executed parallelly, as it supports task addition in a queue, task cancellation, and task prioritization.

IntentService

IntentService is subclass inherited from **Service**.

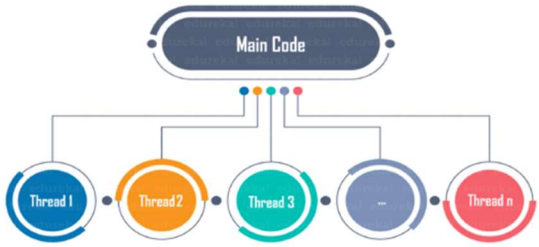
Service is a very important component in Android programming. Sometimes we might have a task to be performed even after the application is closed. This is the scenario in which **Service** will be more helpful. **Service** can be invoked and cancelled by **StartService ()/stopService ()** and runs in the background for a long time. It can also be cancelled by calling **stopSelf ()** inside of it.

When to use IntentService?

IntentService handles asynchronous requests on demand. This is the best option if you don't require that your service handle multiple requests concurrently.

Android Multithreading

Multithreading is a way to do multiple things simultaneously or in parallel to each other. You can either use Java based methods to create thread like 'Thread' class or Android specific ways like handlers, loopers etc.



Examples:

- Games are very good examples of threading. You can use multiple objects in games like cars, motor bikes, animals, people etc. All these objects are nothing but just threads that run your game application.
- Railway ticket reservation system where multiple customers accessing the server.
- Web Browsers: A web browser can download any number of files and web pages (multiple tabs) at the same time and still lets you continue browsing.
- Web servers: A threaded web server handles each request with a new thread. There is a thread pool and every time a new request comes in; it is assigned to a thread from the thread pool.
- Text Editors: When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads.
- IDE: IDEs like Android Studio run multiple threads at the same time. You can open multiple programs at the same time. It is also giving suggestions on the completion of a command which is a separate thread.

Android Debugging

Memory Profiler

The Android Memory Profiler is a tool which helps you understand the memory usage of your application.

Every Android Developer should understand memory management. Memory pitfalls cause many of the crashes and performance issues in Android Apps.

Android Profiler, which replaces Android Monitor tools, is included in Android Studio 3.0 and later. It measures several performance aspects of an app in real-time like:

- Battery
- Network
- CPU
- Memory

Android Memory Management

The Android virtual machine keeps track of each memory allocation in the heap. The heap is a chunk of memory where the system allocates Java/Kotlin objects.

There's a process for reclaiming unused memory known as Garbage Collection. It has the following objectives:

- Find objects that nobody needs.
- Reclaim the memory used by those objects and return it to the heap.

You don't generally request a garbage collection. Instead, the system has a running set of criteria to determine when to perform one.

To enable a multi-task environment, Android puts a limit on the heap size for each app. This size will vary depending on how much available RAM the device has. When the heap capacity is full and system tries to allocate more memory, you could get an OutOfMemoryError.

Logging in Android

The Android system uses a centralized system for all logs. The application programmer can also write custom log messages. The tooling to develop Android applications allows you to define filters for the log statements you are interested in.

To write log statements, you use the *android.util.Log* class with the following methods:

- Log.v ( )
- Log.d ( )
- Log.i ( )
- Log.w ( )
- Log.e ( )
- Log.wtf ( )

Systrace in Android

The Systrace tool helps analyse the performance of your application by capturing and displaying execution times of your application processes and other system processes.

The tool combines the data from the Android kernel such as the CPU scheduler, disk activity and application threads to generate an HTML report that shows an overall picture of an Android device's system processes for a given period of time.

The Systrace tool is particularly useful in diagnosing display problems where an application is slow to draw or stutters while displaying motion or animation.

Exceptions in Android

Exceptions are the unexpected events that come up while running or performing any program. Due to exceptions, the execution is disturbed and the expected flow of the application is not executed.

The different type of exceptions found in Android are:

ActivityNotFoundException

This is a very common exception. It causes your application to stop during the start or execution of your app.

In this case, check if you have declared your activity in the AndroidManifest.xml file.

DexException

This error occurs because the app, when packaging, finds two .dex files that define the same set of methods.

Usually this happens because the app accidentally acquired 2 separate dependencies on the same library.

To resolve, make sure that none of your dependencies could accidentally be added twice in such manner.

NetworkOnMainThreadException

The exception that is thrown when an application attempts to perform a networking operation on its main thread.

This is only thrown for applications targeting the Honeycomb SDK or higher. Applications targeting earlier SDK versions are allowed to do networking on their main event loop threads, but it's heavily discouraged.

OutOfMemoryError

This is a runtime error that happens when you request a large amount of memory on the heap. This is common when loading a Bitmap into an ImageView

UncaughtException

If you want to handle uncaught exceptions try to catch them all in onCreate method of your application class.

ApplicationNotResponding

This is very common dialog is caused by the system killing unresponsive applications that have hung on a long-running operation. This could occur when a network I/O connection to a third-party is unresponsive, perhaps due to unreliable connectivity, and many other such conditions.

NullpointerException

This exception is thrown when an application attempts to use null in a case where an object is required.

These includes: Calling the instance method of a null object. Accessing or modifying the field of a null object.

StackOverflowError

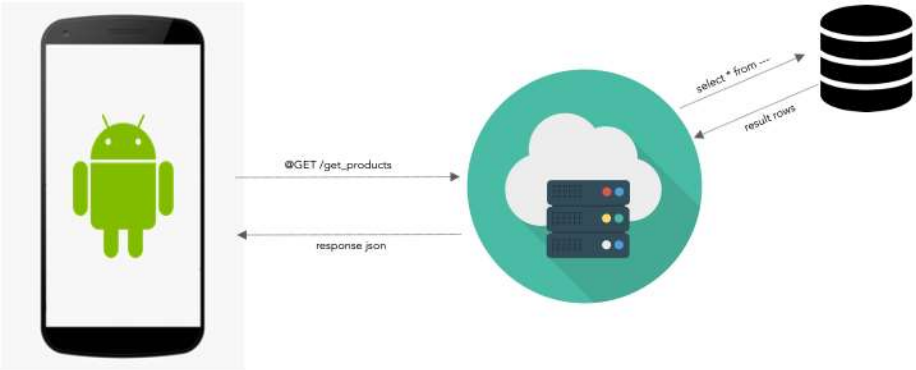
You might run into this one which indicates infinite or deep recursion when creating nested layouts programmatically. Ideally these will be caught during implementation.

Android Exception/Error Handling

Most of the time errors come from the data layer, such as:

- **Device Errors**: Cannot access database, permission denied accessing other app's internal storage.
- **API failures**: No internet connection, Page not found, Server Error…
- **3rd Party Provider errors**: e.g., too many requests at the same time for commute times service, access denied from firebase database.
- Ans many more…

The right way to handle any error is to show a message to the user. Note: that is true almost for any project, but feel free to adjust it to your project requirement.



Memory Leaks in Android

Memory Leaks

A Memory leak happens when your code allocates memory for an object, but never deallocates it. This can happen for many reasons.

No matter the cause, when a memory leak occurs the Garbage Collector thinks an object is still needed because it's still referenced by other objects. But those references should have cleared.

Memory leaks can happen easily on an Android device if not taken care of while building apps, as Android devices are provided with very less memory.

Reasons Memory leak happens:

Most important reason i.e., Memory leaks can happen easily on an Android device if not taken care of while building apps, as Android devices are provided with very less memory.

Holding reference of UI specific object in the background:

Never hold the reference of UI specific object in a background task, as it leads to memory leak.

Using static views:

Do not use static views as it is always available, static views never get killed.

Using static context:

Never use context as static.

Using context:

Be careful while using context, deciding which context is suitable at places is most important. Use application context if possible and use activity context only if required.

Never forget to say goodbye to listeners after being served:

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.

Using inner class:

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.

Using anonymous class:

This is very similar to non-static inner class as it is very helpful but still avoid using an anonymous class.

Using views in collection:

Avoid putting views in collections that do not have clear memory pattern. WeakHashMap store views as values. Since WeakHashMap store views as a hard reference it is better to avoid using it.

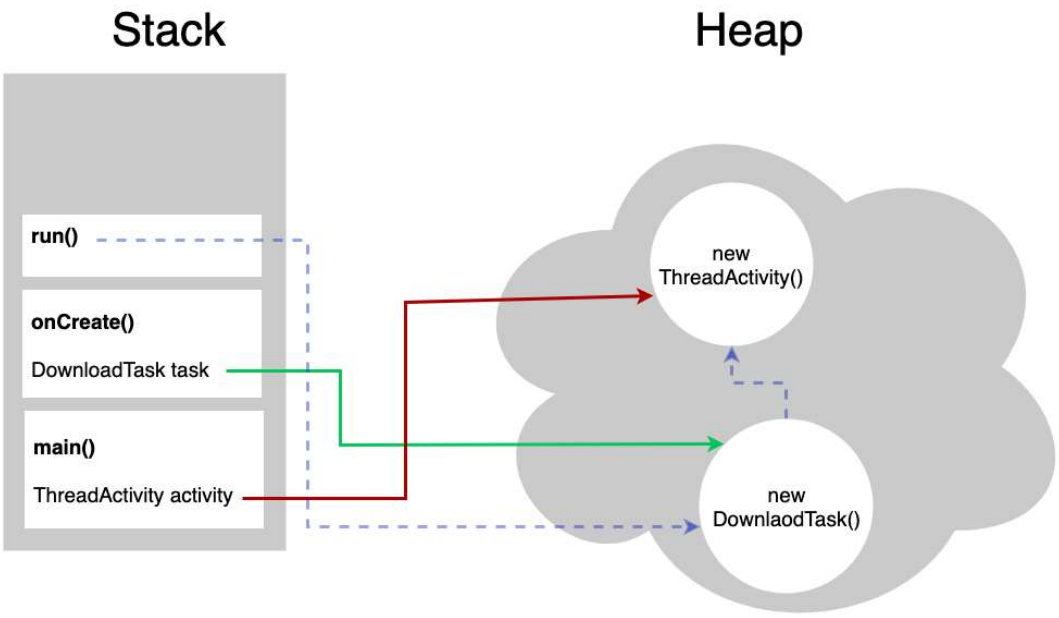
These all are the reasons leads to memory leaks and then ANR, lags in app and OutOfMemoryError that ultimately leads to uninstallation of your app by users.

If memory leak happens due to these specific reasons, as the garbage collector is not intended to handle these, these are your own mistakes try not to commit these.

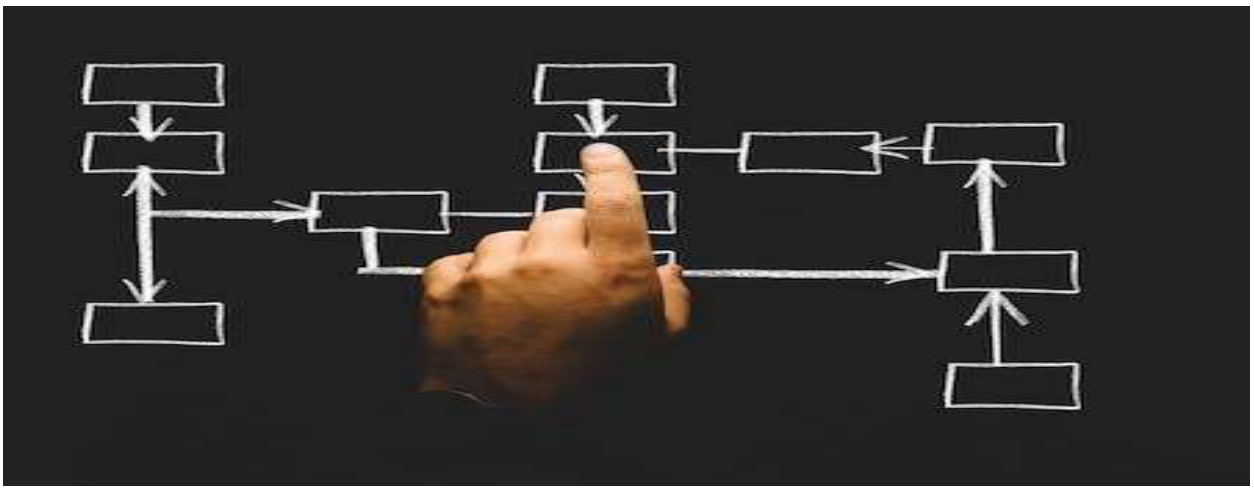
How to detect and solve memory leaks issue?

LeakCanary is a library designed to detect memory leaks.

Even detecting and fixing a single bug by own is difficult, so you can imagine how difficult is to find and fix memory leaks for the whole app. Thanks to 'The saviour' LeakCanary for saving us from memory leaks problem, it runs along with the application and helps in detecting where the application is leaking memory. It even notifies us about where our app is leaking.



Android Context



As the name suggests, it's the context of current state of the application/object. It lets the newly created objects understand what has been going on.

Typically, you call it to get information regarding another part of your program (activity and package/application).

You can get the context by invoking *getApplicationContext ( )*, *getContext ( )*, *getBaseContext ( )* or *this* (when in a class that extends from Context, such as the Application, Activity, Service and intentService classes).

Few important points about the context:

- It is the context of the current state of the application
- It can be used to get information regarding the Activity and Application.
- It can be used to get access to resources, databases and shared preferences etc.
- Both the Activity and Application classes extend the Context class.
- Wrong use of Context can easily lead to memory leaks in an Android application.

Mainly two types of context:

Application context:

- It is the application and we are present in Application.
- For example: MyApplication (which extends Application class). It is an instance of MyApplication only.
- It is an instance that is the singleton and can be accessed in activity via *getApplicationContext ( )*.

Activity context:

- It is the activity and we are present in Activity.
- For example: MainActivity. It is an instance of MainActivity only.

Third – Party Libraries

Image Loading: Glide & Picasso

Image Loading libraries come in handy to avoid high memory consumption caused by loading multiple images at the same time. Images are the greatest source of **Out of Memory errors** in android development. These libraries, therefore, reduce the hassle of loading and caching images together with minimizing memory usage to provide a seamless user experience.

Glide

Glide is an image loading library focused on smooth scrolling. Glide ensures image loading is both as fast and as smooth as possible by applying smart automatic down-sampling and caching to minimize storage overhead and decode times. It also reuses resources like byte arrays and automatically releases application resources where necessary. At the time of writing, Glide's latest version requires a minimum SDK of API 14 (android 4.0) and requires a compile SDK of API 26 (android 8.0) or later.

It provides **animated GIF** support and handles **image loading/caching**.

Using Glide

We first need to make sure we have the Maven and Google repositories in our project `build.gradle` file:

```
repositories {
mavenCentral()
google()
}
```

Then, we add the library dependency in our app-module `build.gradle` file and sync it to make the library available for use:

```
implementation 'com.github.bumptech.glide:glide:4.4.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.4.0'
```

We can then load an image from a remote URL with a single line of code:

```
GlideApp
.with(this)
.load("https://res.cloudinary.com/demo/video/upload/dog.png")
.into(imageView);
```

The `with` method can take a `Context`, `Activity`, `Fragment` or `View` object. The `load` method takes a remote URL or a drawable file (e.g. `R.drawable.image`). The `imageView` instance, passed as an argument to the `into` method, should be of type `ImageView`.

Picasso

Picasso is another great image library for android. It's created and maintained by Square, a company that heavily dependent and contributor to the open-source world, that caters to image loading and processing. By using Picasso, the process of displaying images from external locations is simplified. Picasso supports complex image transformations, automatic caching to disk, ImageView recycling and download cancellation in an adapter.

The library handles every stage of the process. It starts by handling HTTP requests and also handles the caching of the image. Just as Glide does.

Which is better Picasso or Glide?

The way Glide loads an image to memory and do the caching is better than Picasso which let an image loaded far faster. In addition, it also helps preventing an app from popular OutOfMemoryError. GIF animation loading is a killing feature provided by Glide.

Using Picasso

The first thing we need to do is add the Picasso dependency in our app-module `build.gradle` file:

```
implementation 'com.squareup.picasso:picasso:2.5.2'
```

After that, we sync our `gradle` file and load an image resources with a single line of code:

```
Picasso
.with(this)
.load("https://res.cloudinary.com/demo/video/upload/dog.png")
.into(imageView);
```

As we can see, the API provided by Picasso is very similar to the one provided by Glide.

Features	Glide	Picasso
Size and Method Count	Has a bigger .jar size and more methods.	Has smaller .jar size and fewer methods.
Disk Caching	It downloads the image from the given URL, resize it to the size of the image view and stores it to the disk cache.	It downloads the image and stores the full-size image in the cache.
Memory	More memory efficient.	Less memory efficient as compared to Glide.
Image Load time	Loads slower from remote `URLs` but faster from cache.	Vice versa.
Other features	Animated GIF support, Thumbnail support, Configurations and customizations	

Video Streaming

Displaying videos poses to be another difficult task during development. The process and details to take care of be too numerous to handle. In this category, there are a few available options. However, as the most popular and powerful one is ExoPlayer.

ExoPlayer

ExoPlayer is an android media player library developed by Google. It provides an alternative to android's MediaPlayer API for playing audio and video (both locally and over the internet) with some added advantages.

ExoPlayer supports features not currently supported by android's MediaPlayer API, like DASH and SmoothStreaming adaptive playbacks. One of ExoPlayer's biggest advantage is its easy customization.

Using ExoPlayer

The first step is to make sure we have the JCenter and Google repositories included in the project `build.gradle` configuration file:

```
repositories {
jcenter()
google()
}
```

Next, we need to add a Gradle compile dependency to the same file:

```
implementation 'com.google.android.exoplayer:exoplayer:2.6.0'
```

Then, in our layout resources file, we add the `SimpleExoPlayerView` component:

```
<com.google.android.exoplayer2.ui.SimpleExoPlayerView
android:id="@+id/simple_exoplayer_view"
android:layout_width="match_parent"
android:layout_height="wrap_content"/>
```

After that, in the corresponding Activity class, we instantiate ExoPlayer's classes:

```
SimpleExoPlayerView simpleExoPlayerView;
SimpleExoPlayer player;
```

We then initialize our `SimpleExoPlayerView` in the `onCreate` method of our activity:

```
simpleExoPlayerView = findViewById(R.id.simple_exoplayer_view);
```

And, in the `onStart` method, we call the `setupPlayer` method:

```
@Override
protected void onStart() {
super.onStart();
setupPlayer();
}
```

And the `setupPlayer` method:

```
void setupPlayer(){
BandwidthMeter bandwidthMeter = new DefaultBandwidthMeter();
TrackSelection.Factory videoTrackSelectionFactory =
new AdaptiveTrackSelection.Factory(bandwidthMeter);
TrackSelector trackSelector =
new DefaultTrackSelector(videoTrackSelectionFactory);

//initialize the player with default configurations
player = ExoPlayerFactory.newSimpleInstance(this, trackSelector);

//Assign simpleExoPlayerView
simpleExoPlayerView.setPlayer(player);

// Produces DataSource instances through which media data is loaded.
DataSource.Factory dataSourceFactory =
new DefaultDataSourceFactory(this, Util.getUserAgent(this, "CloudinaryExoplayer"));

// Produces Extractor instances for parsing the media data.
ExtractorsFactory extractorsFactory = new DefaultExtractorsFactory();

// This is the MediaSource representing the media to be played.
MediaSource videoSource = new ExtractorMediaSource(videoUri,
dataSourceFactory, extractorsFactory, null, null);

// Prepare the player with the source.
player.prepare(videoSource);
}
```

Here, we initialized the `player` with some default configurations and then assigned it to the `SimpleExoPlayerView` instance. The `videoUri` is of type `Uri`. Every file stored our device has a `Uri` turning it into a unique address to that file. If we intend to display a video from a remote `URL`, we have to parse it this way:

```
Uri videoUri = Uri.parse("any_remote_url");
```

With this, we have basic implementation of ExoPlayer.

Dependency Injection

Dependency injection is a concept where an object does not need to configure its dependencies. Instead, dependencies are passed in by another object. This principle helps us to decouple our classes from their implementation.

It is worth nothing that this is a good software engineering practice because it makes our code loosely-coupled, which makes it easier to maintain and test. There are a number of dependency injection libraries but Dagger2 seems to be the lord of them.

Dagger2

Dagger2 is a fully static, compile-time dependency injection framework for both Java and Android. It is an upgrade to the earlier version Dagger1 created by Square that is now maintained by Google. The recent Dagger version includes android specific helpers for android. Specifically, the auto-generation of subcomponents using a new code generator. Dagger2 is very deep and may require more than the brief sample usage for adequate understanding, but let's take a look at it anyway.

Using Dagger2

As always, first we need to add the dependencies to our app-module `build.gradle` file:

```
implementation 'com.google.dagger:dagger:2.14.1'
annotationProcessor 'com.google.dagger:dagger-compiler:2.14.1'
// we add this so we can use the android support libraries
implementation 'com.google.dagger:dagger-android-support:2.14.1'
annotationProcessor 'com.google.dagger:dagger-android-processor:2.14.1'
```

After that, we can create an activity builder module class to enable Dagger create sub-components for the activity that will need dependencies.

```
@Module
public abstract class ActivityBindingModule {
@ContributesAndroidInjector()
abstract MainActivity mainActivity();
}
```

We can optionally create a module class with specific dependencies to an activity. We then have to add the modules to the constructor of the `@ContributesAndroidInjector` annotation for that activity, for instance:

```
@ContributesAndroidInjector(modules = {MainActivityModule.class} )
abstract MainActivity mainActivity();
```

we can also create another module class to provide dependencies to be used beyond just one activity class.

A `module` class is annotated with `@Module` and is responsible for objects. Objects are provided by creating methods (usually annotated with `@Providers` or `@Binds`) that have the same return type as what is to be provided. The sample module class below provides `String` for our app:

```
@Module
public abstract class AppModule {
@Provides
static String providesString(){
return "I love Auth0";
}
}
```

With `@Module` in place, we then create an abstract class or an interface. It will be named `AppComponent` in our case. The `AppComponent` is annotated with `@Component`. The annotation takes in the module classes created earlier including the `AndroidSupportInjection` class which is from the support library.

Dagger generates a class which then implements this interface. The class provides the injected instances from the modules passed. The `Component` interface looks like this:

```
@Singleton
@Component(modules = {AndroidSupportInjectionModule.class, AppModule.class,
ActivityBindingModule.class})
public interface AppComponent extends AndroidInjector<AppController> {
@Override
void inject(App instance);

@Component.Builder
interface Builder {
@BindsInstance
Builder application(Application application);
AppComponent build();
}
}
```

`AppController` is our application class for the app where objects are initialized once throughout app life-cycle. Our `AppController` looks this way:

```
public class App extends Application implements HasActivityInjector {
@Inject
DispatchingAndroidInjector<Activity> activityDispatchingAndroidInjector;
@Override
public void onCreate() {
super.onCreate();
DaggerAppComponent.builder().application(this)
.build().inject(this);
}
@Override
public DispatchingAndroidInjector<Activity> activityInjector() {
return activityDispatchingAndroidInjector; }}
```

we had to create an instance of `DispatchingAndroidInjector <Activity>` and return it in the implemented method. The reason for this is to perform members-injection on activities. Then, we built our `Component` and injected the `Application` class into it.

Finally, in the activity where we intend to use dependencies, we implement `HasSupportFragmentInjector` and access our dependencies by just using the `@Inject` annotation.

```
public class MainActivity extends AppCompatActivity implements HasSupportFragmentInjector {

@Inject
DispatchingAndroidInjector<Fragment> fragmentDispatchingAndroidInjector;

@Inject
String string;

@Override
protected void onCreate(Bundle savedInstanceState) {
AndroidInjection.inject(this);
super.onCreate(savedInstanceState);
Log.d("TAG",string);
}

@Override
public AndroidInjector<Fragment> supportFragmentInjector() {
return fragmentDispatchingAndroidInjector;
}
}
```

We did something similar to what we did in the `App` class. The only difference is that we are implementing `HasSupportFragmentInjector` so the `DispatchingAndroidInjector <T>` has `Fragments` in its type now. The logic is this: The `Application` will contain `Activities`, and these will house `Fragments`. Thereafter, we called `AndroidInjection.inject(this)` in our `onCreate` method.

From our brief example, if we run the app, we should see `I love Auth0` printed on the log. We were able to inject the dependencies instead of initializing it in the class. This is just a bit of Daagger2 has to offer.

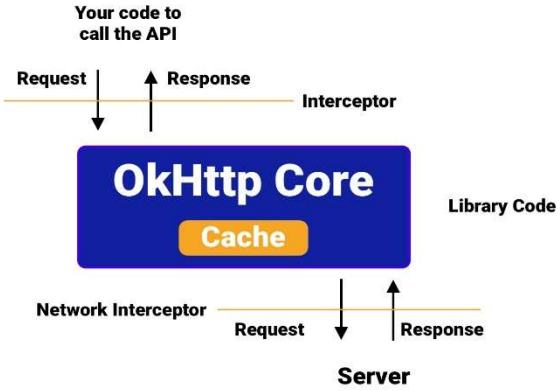
Networking - Retrofit

OkHttp (HTTP + HTTP 2)

Currently, OkHttp is the most used Http client library provided by the square.github.io community for Java and Android developers. Using OkHttp is easy. It supports both synchronous blocking calls and asynchronous blocking calls with callbacks.

OkHttp by default supports network catching under the hood for avoiding the repeat network request.

OkHttp requires a minimum Android 5.0+ (API Level 21 +) and Java 8+.



Retrofit

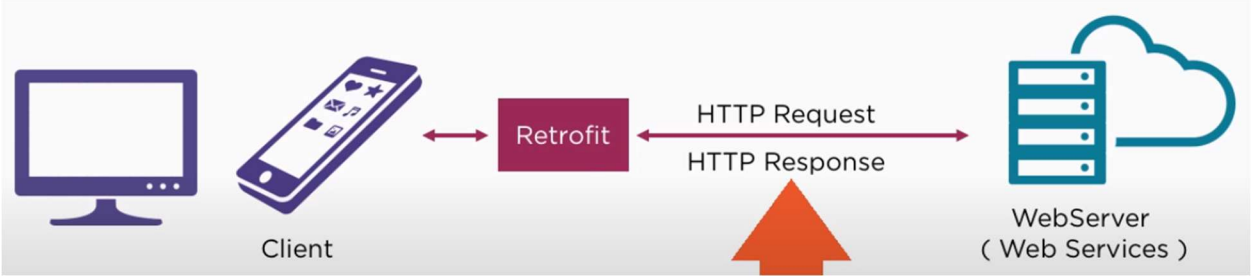
No doubt Retrofit is our favourite library when it comes to networking. Retrofit is an awesome type-safe HTTP client for Android and Java built by Square.

Retrofit is a REST client for Java and Android. It makes it relatively easy to retrieve and upload JSON (or other structured data) via a REST based webservice. In Retrofit you configure which converter is used for the data serialization.

It makes it easy to consume JSON or XML data which is parsed into Plain Old Java Objects (POJO's).

The Retrofit makes it so much easier to create Http request via annotations.

Retrofit is a popular choice for **managing http request as well as response** and **web service integration**.



Example: (Facebook)

Whenever we open our application, we try to refresh our feed. Once you do it your Facebook application sends a http request to the Facebook web server and the server then again sends back the http response, So, that you can see the stories in your feed.

What makes your app professional?

Of course, making your application online is your first step, but after you connect your application to the world you need to architect your application in such a way that it handles User Authentication seamlessly, run request in the background thread, parse JSON to a useable class object and handle images effectively which is another challenge for developers in the real world.

Popular Http Clients

- HTTPUrlConnection
- Volley
- Retrofit
- OkHttp

In case of **OkHttp**, it is another library that handles HTTP communication but at poorly low level means to achieve a task over Http request, you need write too many lines of code, let's assume you need to write 10 lines of code to achieve your task but with time we got a class called **HttpUrlConnection** which is built on top of OkHttp and it handles request with more abstraction, what you achieve here in 10 lines of code, you can fairly achieve that in only 7 lines of code.

But still we are not satisfied, so we got **Retrofit**, which again sits on top of OkHttp client, so now it allows developers to handle request with fairly high-level abstraction, here you can achieve the same task with just 3 to 4 lines of code.

Well **volley** is another high-level library similar to retrofit but it has its own limitations which was solved in Retrofit.

Note: All of these libraries work asynchronously in the background thread, this way they do not use main thread and thus prevent long running operations from blocking the user interface.

Disadvantages of HTTPUrlConnection

The major disadvantages of this class are that the code for this class is bit more difficult to read and write when compared to other options you will need to write a lot of boilerplate code and you will need to be comfortable with byte arrays, stream readers and whatnot parsing responses for this class requires again manual process that is there is no inbuilt mechanism to parse JSON response when you use this class you have to manually manage the background threads to perform multiple asynchronous requests.

- Poor readability and less expressive
- Lots of boilerplate code
- No built-in support for parsing JSON response
- Manage background thread manually
- Poor resource management

Disadvantage of Volley

- Volley does not have REST and authentication friendly features compared to Retrofit, the documentation and community support for volley is meager when compared to retrofit.
- Limited REST specific features
- Poor authentication layer
- Meager Documentation
- Smaller community

Features of Retrofit

- Retrofit has a huge active community that will help you to easily troubleshoot your errors.
- Well, this library allows you to write expressive and concise code. It helps you to manage resources very efficiently. For example, how to manage background thread effectively, manage asynchronous calls and queues.
- It has a build in support to parse JSON using GSON library.
- It has mechanism to handle error effectively.
- It has built-in user authentication support.

HTTP

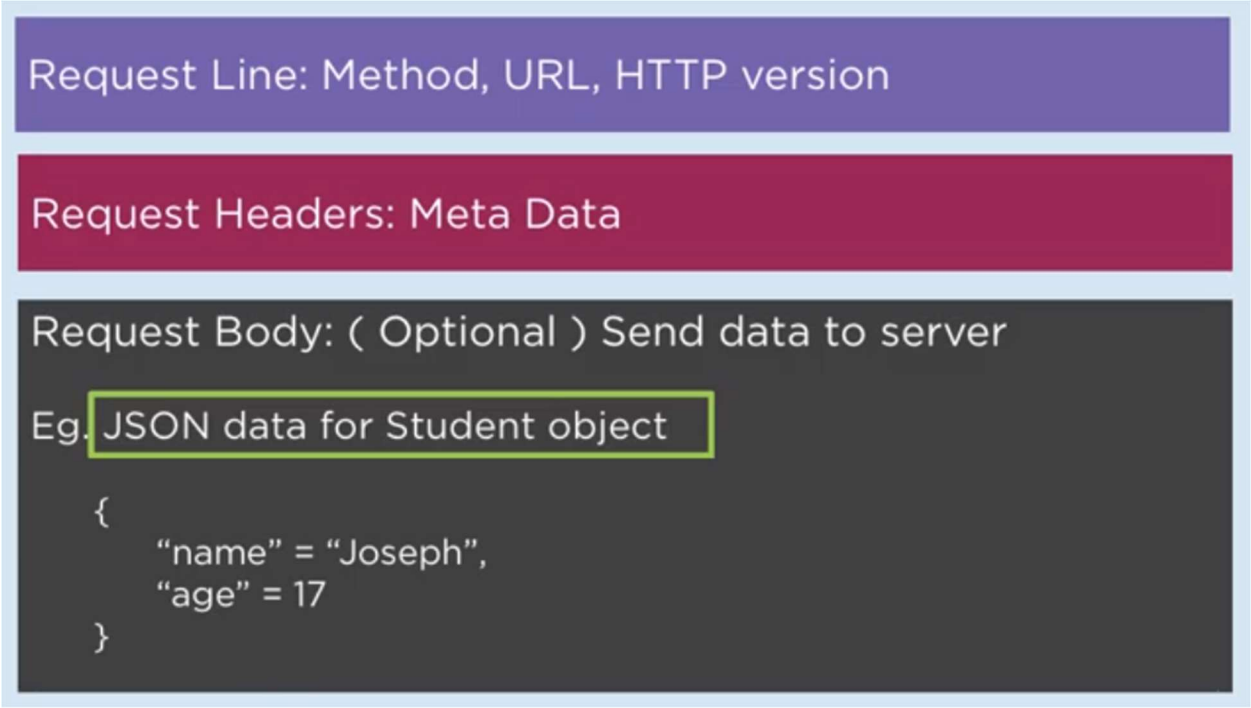
Hyper Text Transfer Protocol

It is an application-level protocol for distributed, collaborative, and hypermedia information systems.

You can call HTTP as a language for communication between Client and Server.

From client we send a request to the server. And this request is formatted as per HTTP standards which is understood by the web server. The server then parses the request, process it and respond back to the client again in a way which is understandable by the client in the form of HTTP response. Thus, completing the communication flow.

HTTP Request Structure



Main HTTP Methods

- GET: Retrieve a resource
- POST: Add a resource
- PUT: Replace a resource
- PATCH: Update parts of Resource
- DELETE: Remove a resource

Example:

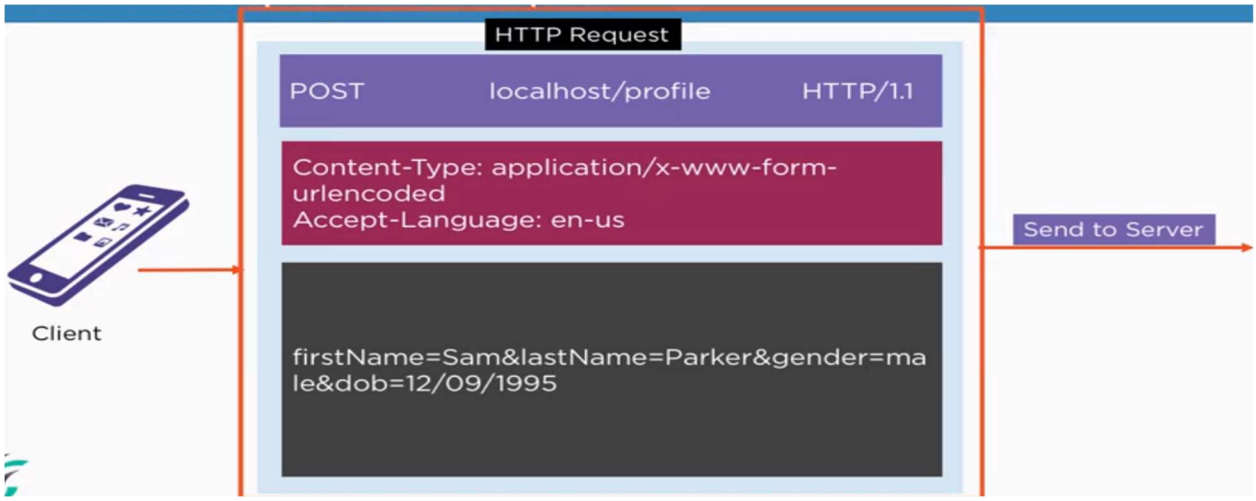
Assuming our application user wants to add his profile to the server. So, in order to send this data to our web service, we need to package it as Http Request.

In Request line we can set the method type as POST because we want to add a new data to the server. Then we can define our URL, where you want to send data and then include the HTTP version.

We can also include the HTTP Header that specifies the data format. Which is in our case is the form data. You can also add header for language such as: English.

Finally, we can package up the form fields with their corresponding values and attach them in the body of Request such as FirstName, LastName, Gender and DOB.

Once our HTTP request formation is complete, we will then send it to the server.



Now the server on the other hand will receive the incoming HTTP Request from the app. Once the server receives our request it creates a new profile object for the user and then add it to the database. Well, it might perform other server-side tasks such as sending out confirmation email to the user.

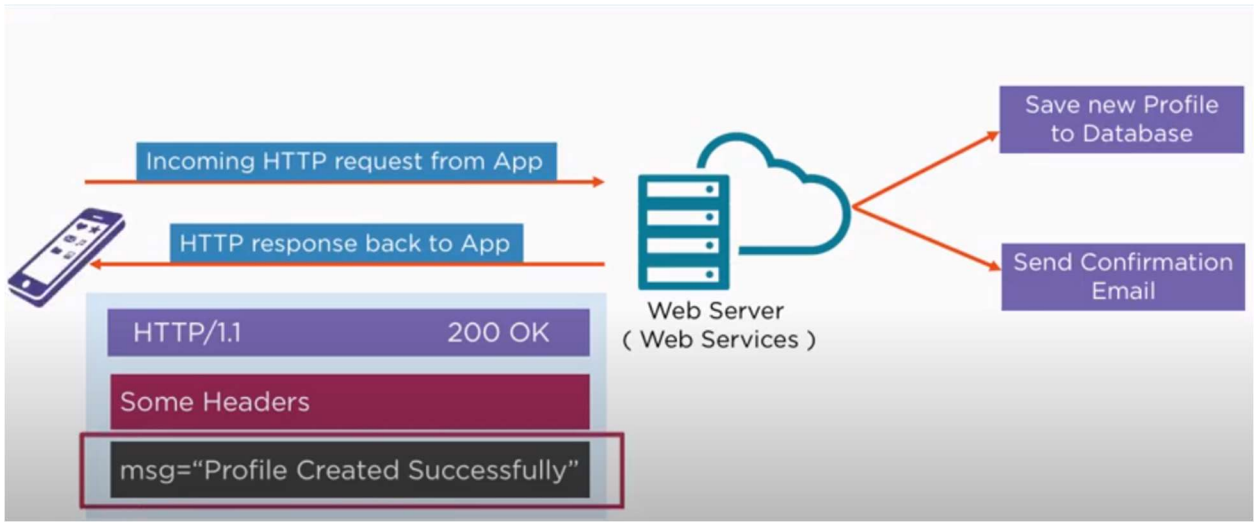
Well, assuming these operations are successful. Then in return the server creates HTTP Response message so as to send it back to the client.

Structure of HTTP Response

Response looks similar to Request, but in case of request line. It simply contains the HTTP version and a Status code detailing how things went.

In this case it is a success, we are sending back 200 OK which indicates everything was successful. Additionally, we can some Headers as well as some body, then we can define our message "Profile Created Successfully".

Now once the Response is received by our client, then in the client we can parse the data and display a message to the user that profile was created successfully.



HTTP Status Code

Status Code Range	Meaning
100's	Informational
200's	Success
300's	Redirects
400's	Client Error
500's	Server Error

- 100: We are sending some informational data.
- 200: The Request was successful.
- 300: Deals with Redirection.
- 400: The client has made some types for error while making the request such as sending some bad data or some authentication error such as unauthorized access.
- 500: Indicates some kind of error happen in the server side.

Web Services

The task is to read the request and process it and send back the response. So, in short, the web service is a normal program running in your server.

What is RESTful Web Service and Why?

In **SOAP** and **WSDL** Services. These web services transmit data using XML. So, they are heavy and not mobile friendly.

In **URL Patterns**, for a user related operation we have to use these URLs (Smartherd.com/getUser || Smartherd.com/addUser || Smartherd.com/updateUser || Smartherd.com/deleteUser. So, for just one operation you have to use 4 different URLs which is not scalable.

Definition: RESTful Web Service

A Web Service is RESTful when it provides stateless operations to manage data using different HTTP methods and structured URLs.

Properties of RESTful web services:

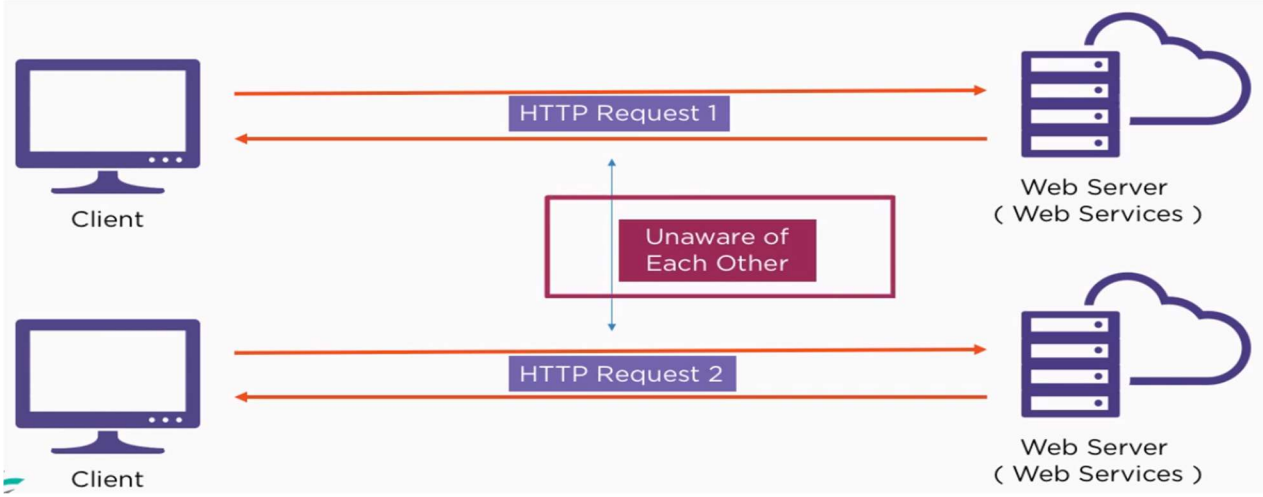
Stateless Client-Server Relationship

The RESTful web services are implemented through client-server relationship and this relationship is stateless.

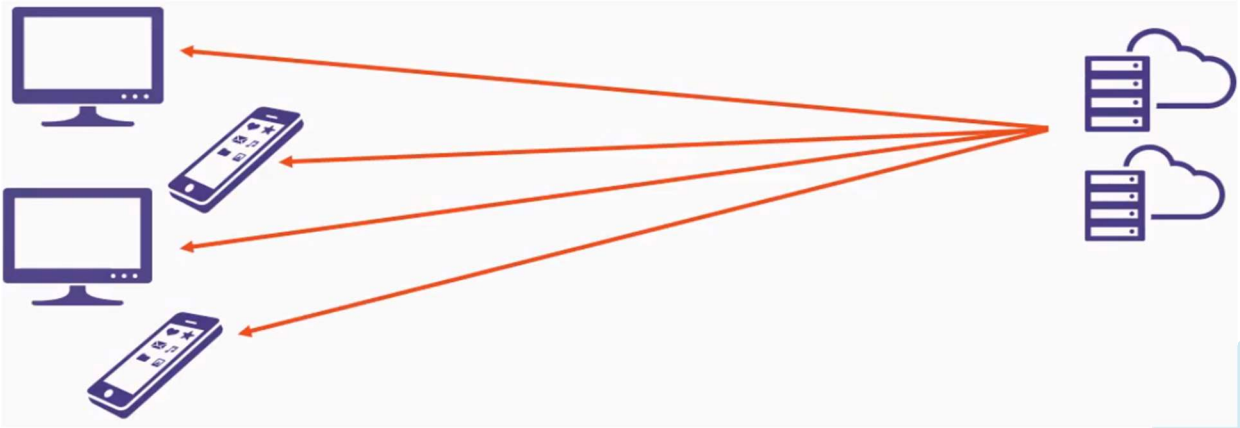
What do you mean by stateless client-server relationship?

Now from a client we send a request to the server. Let's call it Request 1.

Now the same client sends another request to the server. Let's call it Request 2. So, when these two requests are independent of each other then we call this operation as stateless that is the result of these two requests depends only on its own requested data and they are totally unaware of each other and also note that the client-server relationship should be loosely coupled that is you are allowed to change your UI in your client without modifying your web services.



The client-server relationship also allows us to greatly scale up our application to handle increasing demand. For example, if the no. of user increases, we can increase the no. of web servers as well to handle these clients.



Utilize HTTP Methods

Uses HTTP methods like GET, PUT, POST, DELETE and PATCH.

Structured and Consistent URLs

The RESTful communication is generally implemented through structured URLs that consistent.

These URLs work in conjunction of these HTTP methods. For example, to perform operation on the user you can use the same URL but with different methods. This way you can perform GET. ADD and DELETE operation.

HTTP Method	Consistent URL	Web Service Operation
GET	smartherd.com/user	Fetch User
POST	smartherd.com/user	Add User
PUT	smartherd.com/user	Update User
DELETE	smartherd.com/user	Delete User

Now, we can extend these URLs to include more information. For example, we can an identifier in the end of these URLs to get or delete a specific entry

HTTP Method	Consistent URL	Web Service Operation
GET	smartherd.com/user/3	Get user with ID of 3
DELETE	smartherd.com/user/3/comments	Delete comments of user whose ID is 3
GET	smartherd.com/user/3/name	Get name of the user whose ID is 3

Consistent Data type Transfer

These services can use different data types (JSON or XML) for transferring information.

But generally, REST uses JSON.

Web service Technologies

- .NET
- PHP
- Ruby
- Node: Light weight JavaScript library that is pretty easy to get started with. You can accomplish a task with a minimal code. Its syntax is super easy.
- AWS Lambda
- Azure Functions

Advantages of Local Web Server

Using locale web server during development phase has its own advantages:

- Edit / Add URLs anytime
- Easy debugging
- Add / Edit sample data anytime
- Hassle free

Why we need GSON in Retrofit?

Retrofit does not include a built-in **Data Mapper** for sending and receiving data. But it supports a strong integration with GSON library that helps to convert data into JSON and backend forth. Thus, GSON is basically a converter that helps to map data.

How to implement Retrofit?

Step 1:

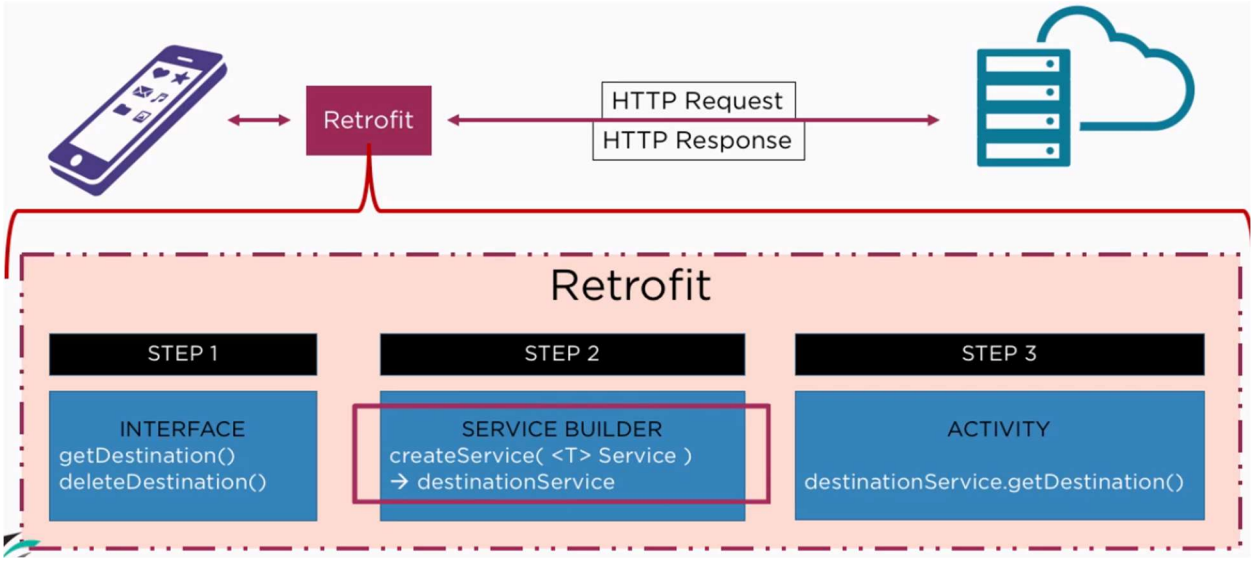
The 1st step you have to take is to create an Interface which is contain various functions which will map to the end point URLs of your web service such as `getDestination ( )` and `deleteDestination ( )`.

Step 2:

In the next step you have to create a Service that will help you to call these functions present within the Interface.

Step 3:

And as a last step within your Activity you have to initialize the `Service Builder` and then call the function of this Interface, which you see in the Activity like `destinationService.getDestination ( )`.



Using Retrofit

To use Retrofit, first, we need to add the dependency to our app `build.gradle` file:

```
implementation 'com.squareup.retrofit2:retrofit:2.3.0'
```

We would also need to insert dependency for converters that we intend to use. Converts handle the mapping of Java objects to the response body.

```
implementation 'com.squareup.retrofit2:converter-gson:2.3.0'
implementation 'com.squareup.retrofit2:converter-scalars:2.3.0'
```

The first converter in the list above (`converter-gson`), as the name states, maps to and from the JSON format. The second is used when we want to deal with primitive data types like `String`. After importing the desired converters, we then create an interface to configure the endpoints that will be accessed:

```
public interface ApiService {
@GET("/data")
Call<ResponseClass> fetchData(@Body JsonObject jsonObject);
}
```

From the snippet, we have an endpoint `/data` that requires `JsonObject` `@Body` for the request. We also have a `ResponseClass` which will be mapped to the expected response body of our request. However, the class is omitted here for brevity. To map JSON objects to POJOs, we can use the **JsonSchema2Pojo** library.

After defining the endpoints available, we then create a Retrofit client:

```
public class RetrofitClient {
static ApiService getService(){
OkHttpClient.Builder httpClient = new OkHttpClient.Builder();
Retrofit.Builder builder = new Retrofit.Builder()
.baseUrl("http://127.0.0.1:5000/")
.addConverterFactory(GsonConverterFactory.create());

Retrofit retrofit = builder
.client(httpClient.build())
.build();

return retrofit.create(ApiService.class);
}
}
```

When building our Retrofit object, we can add as many converters as we want. This provides us a variety of options to parse our data. Thereafter, we make a network request by calling:

```
RetrofitClient.getService().fetchData(jsonObject).enqueue(new Callback<ResponseClass>() {
@Override
public void onResponse(Call<ResponseClass> call, Response<ResponseClass> response) {

}

@Override
public void onFailure(Call<ResponseClass> call, Throwable t) {

}
});
```

Where `JsonObject` contains the request parameters. This request will be made to the `http://127.0.0.1:5000/data` endpoint, as defined in the previous code snippet. As we can see in this last snippet, Retrofit also provides us callback methods to give us the status of the request.

MultiThreading (Reactive Programming)

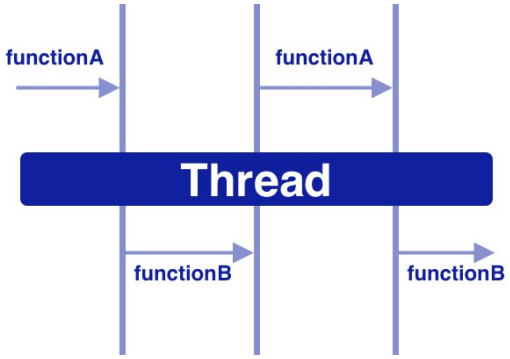
Kotlin Coroutines

This is a framework which is available to handle multithreading leads to callback hells and blocking states with thread-safe execution.

Also, it is a very efficient and complete framework to manage concurrency (It allows performing multiple tasks or processes simultaneously.) in a more performant and simple way.

Coroutines: Coroutines = Co + Routines

Here, Co means cooperation and Routines means functions. It meant that when functions cooperate with each other, we call it as Coroutines.



Suppose we have two function as functionA and functionB.

functionA as below:	functionB as below:

fun functionA (case: Int) {
when (case) {
1 -> {
taskA1()
functionB(1)
}
2 -> {
taskA2()
functionB(2)
}
3 -> {
taskA3()
functionB(3)
}
4 -> {
taskA4()
functionB(4)
}
}
}
	fun functionB (case: Int) {
	when (case) {
	1 -> {
	taskB1()
	functionA(2)
	}
	2 -> {
	taskB2()
	functionA(3)
	}
	3 -> {
	taskB3()
	functionA(4)
	}
	4 -> {
	taskB4()
	}
	}
	}

Then we call the functionA as below:

```
functionA (1)
```

here, functionA will do the taskA1 and give control to the functionB to execute the taskB1.

Then, functionB will do the taskB1 and give the control back to the functionA to execute the taskA2 and so on.

The important thing is that functionA and functionB are cooperating with each other.

With Kotlin Coroutines, the above cooperation can be done very easily which is without the use of when or switch case which I have used in the above example for the sake of understanding.

There are endless possibilities that open up because of the cooperative nature of functions. A few of the possibilities are as follows:

It can execute a few lines of functionA and then execute a few lines of functionB and then again, a few lines of functionA and so on. This will be helpful when a thread is sitting idle not doing anything, in that case, it can execute a few lines of another function. This way, it can take the full advantage of thread. Ultimately the cooperation helps in multitasking.

It will enable writing asynchronous code in a synchronous way.

Overall, the Coroutines make the multitasking very easy.

So, we can say that Coroutines and the threads both are multitasking. But the difference is that threads are managed by the OS and coroutines by the users as it can execute a few lines of function by taking advantage of the cooperation.

It's an optimized framework written over the actual threading by taking advantage of the cooperative nature of functions to make it light and yet powerful. So, we can say that Coroutines are lightweight threads. A lightweight thread means it doesn't map on the native thread, so it doesn't require context switching on the processor, so they are faster.

What does it mean when I say 'it doesn't map on the native thread'?

Coroutines are available in many languages. Basically, there are two types of Coroutines:

- Stackless
- Stackful

Kotlin implements Stackless coroutines – it means that the coroutines don't have their own stack, so they don't map on the native thread.

NOTE:

Coroutines do not replace threads, it's more like a framework to manage it.

Definition of Coroutines: A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

Why there is a need for Kotlin Coroutines?

Let's take very standard use-case of an Android Application which is as follows:

- Fetch User from the server
- Show the User in the UI

```
fun fetchUser(): User {

    // make network call

    // return user

}

fun showUser(user: User) {

    // show user

}

fun fetchAndShowUser() {

    val user = fetchUser()

    showUser(user)

}
```

When we call the fetchAndShowUser function, it will throw the NetworkOnMainThreadException as the network call is not allowed on the main thread.

There are many ways to solve that. A few of them are as follows:

Using Callback: Here, we run the fetchUser in the background thread and we pass the result with the callback.

```
fun fetchAndShowUser() {

    fetchUser { user ->

        showUser(user)

    }

}
```

Using RxJava: Reactive world approach. This way we can get rid of the nested callback.

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

Using Coroutines: Here the code looks like synchronous, but it is asynchronous.

```
suspend fun fetchAndShowUser() {
    val user = fetchUser() // fetch on IO thread
    showUser(user) // back on UI thread
}
```

Here, the above code looks synchronous, but it is asynchronous.

Implementation of Kotlin Coroutines in Android

Add the Kotlin Coroutines dependencies in the Android project as below:

```
dependencies {
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:x.x.x"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:x.x.x"
}
```

Now, our function `fetchUser` will look like below,

```
suspend fun fetchUser(): User {
return GlobalScope.async(Dispatchers.IO) {
// make network call
// return user
```

```
}.await()

}
```

And the `fetchAndShowUser` like below,

```
suspend fun fetchAndShowUser() {

val user = fetchUser() // fetch on IO thread

showUser(user) // back on UI thread

}
```

And the `showUser` function as below which is same as it was earlier:

```
fun showUser(user: User) {

// show user

}
```

We have introduced two things here as follows:

`Dispatcher`: Dispatchers help coroutines in deciding the thread on which the work has to be done. There are majorly three types of Dispatchers which are as,

- `Default`
- `Main`
- `IO`

`IO` dispatcher is used to do the network and disk-related work.

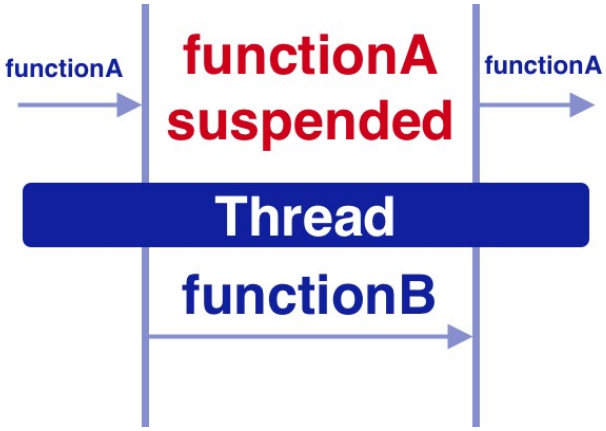
`Default` is used to do the CPU intensive work.

`Main` is the UI thread of Android.

In order to use these, we need to wrap the work under the `async` function. Async function looks like below.

```
suspend fun async() // implementation removed for brevity
```

`Suspend`: Suspend function is a function that could be started, paused and resume.



Suspend functions are only allowed to be called from a coroutines or another suspend function. You can see that the `async` function which includes the keyword `suspend`. So, in order to use that, we need to make our function `suspend` too.

So, the `fetchAndShowUser` can only be called from another suspend function or a coroutines. We can't make the `onCreate` function of an activity `suspend`, so we need to call it from the coroutines like below:

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

    super.onCreate(savedInstanceState)

    GlobalScope.launch(Dispatchers.Main) {

        fetchAndShowUser()

    }

}
```

Which is actually,

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

    super.onCreate(savedInstanceState)

    GlobalScope.launch(Dispatchers.Main) {

        val user = fetchUser() // fetch on IO thread

        showUser(user) // back on UI thread

    }

}
```

`showUser` will run on UI thread because we have used the `Dispatcher.Main` to launch it.

There are two functions in Kotlin to start the coroutines which are as follows:

- `Launch { }`
- `Async { }`

Launch vs Async in Kotlin Coroutines

The difference is that the `launch { }` does not return anything and the `async { }` returns an instance of `Deferred <T>`, which has an `await ( )` function that returns the result of the coroutines like we have future in Java in which we do `future.get ( )` to get the result.

In order words:

- `launch`: fire and forget
- `async`: perform a task and return a result

Example:

We have a function `fetchUserAndSaveInDatabase` like below:

```
suspend fun fetchUserAndSaveInDatabase() {

    // fetch user from network

    // save user in database

    // and do not return anything

}
```

Now, we can use the launch like below:

```
GlobalScope.launch(Dispatchers.Main) {

    fetchUserAndSaveInDatabase() // do on IO thread

}
```

As the `fetchUserAndSaveInDatabase` does not return anything, we can use the `launch` to complete that task and then so something on Main Thread.

But when we need the result back, we need to use the `async`.

We have two functions which return User like below:

```
suspend fun fetchFirstUser(): User {

    // make network call

    // return user

}

suspend fun fetchSecondUser(): User {

    // make network call

    // return user

}
```

Now, we can use the `async` like below:

```
GlobalScope.launch(Dispatchers.Main) {

    val userOne = async(Dispatchers.IO) { fetchFirstUser() }

    val userTwo = async(Dispatchers.IO) { fetchSecondUser() }

    showUsers(userOne.await(), userTwo.await()) // back on UI thread

}
```

Here, it makes both the network call in parallel, wait for the results, and then calls the `showUsers` function.

So, now that, we have understood the difference between the `launch` function and the `async` function.

There is something called `withContext`.

```
suspend fun fetchUser(): User {

    return GlobalScope.async(Dispatchers.IO) {

        // make network call

        // return user

    }.await()}
```

`withContext` is nothing but another way of writing the async where we do not have to write `await {}`.

```
suspend fun fetchUser(): User {
    return withContext(Dispatchers.IO) {
        // make network call
        // return user
    }
}
```

But there are many more things that we should know about the `withContext` and the `await`.

Now, let's use `withContext` in our `async` example of `fetchFirstUser` and `fetchSecondUser` in parallel.

```
GlobalScope.launch(Dispatchers.Main) {
    val userOne = withContext(Dispatchers.IO) { fetchFirstUser() }
    val userTwo = withContext(Dispatchers.IO) { fetchSecondUser() }
    showUsers(userOne, userTwo) // back on UI thread
}
```

When we use `withContext`, it will run in series instead of parallel. That is a major difference.

The thumb-rules:

- Use `withContext` when you do not need the parallel execution.
- Use `async` only when you need the parallel execution.
- Both `withContext` and `async` can be used to get the result which is not possible with the `launch`.
- Use `withContext` to return the result of a single task.
- Use `async` for results from multiple tasks that run in parallel.

Scopes in Kotlin Coroutines

Scopes in Kotlin Coroutines are very useful because we need to cancel the background tasks as soon as the activity is destroyed.

Assuming that our activity is the scope, the background task should get cancelled as soon as the activity is destroyed.

In activity, we need to implement `CoroutineScope`.

```
class MainActivity : AppCompatActivity(), CoroutineScope {

    override val coroutineContext: CoroutineContext

        get() = Dispatchers.Main + job

    private lateinit var job: Job

}
```

In the `onCreate` and `onDestroy` function.

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

    super.onCreate(savedInstanceState)

    job = Job() // create the Job

}

override fun onDestroy() {

    job.cancel() // cancel the Job

    super.onDestroy()

}
```

Now, just use the launch like below:

```
launch {

    val userOne = async(Dispatchers.IO) { fetchFirstUser() }

    val userTwo = async(Dispatchers.IO) { fetchSecondUser() }

    showUsers(userOne.await(), userTwo.await())

}
```

As soon as the activity is destroyed, the task will get canceled if it is running because we have defined the scope.

When we need the global scope, which is our application scope, not the activity scope, we can use the `GlobalScope` as below:

```
GlobalScope.launch(Dispatchers.Main) {

    val userOne = async(Dispatchers.IO) { fetchFirstUser() }

    val userTwo = async(Dispatchers.IO) { fetchSecondUser() }

}
```

Here, even if the activity gets destroyed, the `fetchUser` functions will continue as we have used the `GlobalScope`.

This is how the Scopes in Kotlin Coroutines are very useful.

Exception Handling in Kotlin Coroutines

When using launch

One way is to try-catch block:

```
GlobalScope.launch(Dispatchers.Main) {

    try {

        fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread

    } catch (exception: Exception) {

        Log.d(TAG, "$exception handled !")

    }

}
```

Another way is to use a handler:

For this we need to create an exception handler like below:

```
val handler = CoroutineExceptionHandler { _, exception ->

    Log.d(TAG, "$exception handled !")

}
```

Then, we can attach the handler like below:

```
GlobalScope.launch(Dispatchers.Main + handler) {

    fetchUserAndSaveInDatabase() // do on IO thread and back to UI Thread}
```

If there is an exception in `fetchUserAndSaveInDatabase`, it will be handled by the handler which we have attached.

When using in the activity scope, we can attach the exception in our `coroutineContext` as below:

```
class MainActivity : AppCompatActivity(), CoroutineScope {

    override val coroutineContext: CoroutineContext

        get() = Dispatchers.Main + job + handler

    private lateinit var job: Job

}
```

And use like below:

```
launch {

    fetchUserAndSaveInDatabase()

}
```

When using async

When using async, we need to use the try-catch block to handle the exception like below:

```
val deferredUser = GlobalScope.async {

    fetchUser()

}

try {

    val user = deferredUser.await()

} catch (exception: Exception) {

    Log.d(TAG, "$exception handled !")

}
```

Example:

Suppose, we have two network calls like below:

- `getUser ( )`
- `getMoreUsers ( )`

And, we are making the network calls in series like below:

```
launch {
    try {
        val users = getUsers()
        val moreUsers = getMoreUsers()
    } catch (exception: Exception) {
        Log.d(TAG, "$exception handled !")
    }
}
```

If one of the network calls fails, it will directly go to the `catch` block.

But suppose, we want to return an empty list or the network call which has failed and continue with the response from the other network call. We can add the `try-catch` block to the individual network call like below:

```
launch {
    try {
        val users = try {
            getUsers()
        } catch (e: Exception) {
            emptyList<User>()
        }
        val moreUsers = try {
            getMoreUsers()
        } catch (e: Exception) {
```

```
            emptyList<User>()

        }

    } catch (exception: Exception) {

        Log.d(TAG, "$exception handled !")

    }}
```

This way, if any error comes, it will continue with the empty list.

Now, what if we want to make the network calls in parallel. We can write the code below using `async`.

```
launch {

    try {

        val usersDeferred = async {  getUsers() }

        val moreUsersDeferred = async { getMoreUsers() }

        val users = usersDeferred.await()

        val moreUsers = moreUsersDeferred.await()

    } catch (exception: Exception) {

        Log.d(TAG, "$exception handled !")

    }

}
```

Here, we will face one problem, if any network error comes, the application will `crash!`, it will NOT go to `catch` block.

To solve this, we will have to use the `coroutineScope` like below:

```
launch {

    try {

        coroutineScope {

            val usersDeferred = async {  getUsers() }

            val moreUsersDeferred = async { getMoreUsers() }
```

```
            val users = usersDeferred.await()
            val moreUsers = moreUsersDeferred.await()
        }
    } catch (exception: Exception) {
        Log.d(TAG, "$exception handled !")
    }
}
```

Now, if any network error comes, it will go to the `catch` block.

But suppose again, we want to return empty list for the network call which has failed and continue with the response from the other network call. We will have to use the `supervisoeScope` and the `try-catch` block to the individual network call like below:

```
launch {
    try {
        supervisorScope {
            val usersDeferred = async { getUsers() }
            val moreUsersDeferred = async { getMoreUsers() }
            val users = try {
                usersDeferred.await()
            } catch (e: Exception) {
                emptyList<User>()
            }
            val moreUsers = try {
                moreUsersDeferred.await()
            } catch (e: Exception) {
                emptyList<User>()
            }
```

```
        }

    } catch (exception: Exception) {

        Log.d(TAG, "$exception handled !")

    }

}
```

Again, this way, if any error comes, it will continue with the empty list.

This is how `supervisorScope` helps.

NOTE:

- While NOT using `async`, we can go ahead with the `try-catch` or the `CoroutineExceptionHandler` and achieve anything based on our use-cases.
- While using `async`, in addition to `try-catch`, we have two options: `coroutineScope` and `supervisorScope`.
- With `async`, use `supervisorScope` with the individual `try-catch` for each task in addition to help top-level `try-catch`, when you want to continue with other tasks if one or some of them have failed.
- With `async`, use `coroutineScope` with the top-level `try-catch`, when you do NOT want to continue with other tasks if any of them have failed.
- The major difference is that a `coroutineScope` will cancel whenever any of its children fail. If we want to continue with the other tasks even when one fails, we go with the `supervisorScope`. A `supervisorScope` won't cancel other children when one of them fails.

This is how the exception handling can be done in the Kotlin Coroutines.

RxJava2 & RxAndroid

Reactive Programming is a paradigm where data is emitted from a component (a source) to another component (a subscriber). This helps us to handle asynchronous tasks effectively. Reactive programming libraries are, therefore, libraries that help in passing the data from sources to subscribers. The most popular alternative available for android developers is RxJava2 and RxAndroid.

RxJava

What is RxJava?

RxJava is a JVM library for doing asynchronous and event-based program by using observable sequences. It's 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.

But, RxJava is made for primarily any Java projects. To use RxJava in Android, we will also need **RxAndroid**.

RxJava is a JVM implementation of Reactive Extensions.

So, what is **Reactive Extension**?

Reactive Extension (ReactiveX) as a library for composing asynchronous and event-based programs by using observable sequences.

The definition presents three different terms: **asynchronous**, **event-based** and **observable sequences**. It also says composing programs.

- **Asynchronous**: It implies that the different parts of a program run simultaneously.
- **Event-Based**: The program executes the codes based on the events generated while the program is running. For example, a Button click triggers an event and then the program's event handler receives this event and does some work.

- **Observable Sequences**: Observables are the source that emits data to Observers. We can understand observable as suppliers, they process and supply data to other components. It does some and emits some values.

RxJava free us from the callback hell by providing the composing style of programming. We can plug in various transformations that resembles the functional programming. It extends the observer pattern to support sequences of data and /or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.

RxJava is an implementation of Reactive Stream Specification.

How does this Reactive Stream implement?

- Observables: It emits the values.
- Observer: It gets the values.
- Observer get the emitted values from Observable by subscribing on it.

All components are as follows:

- **Flowable**, **Observable**, **Single** and **Completable** – does some work and emit values.
- **Subscription** – work is going on or completed or is used to cancel.
- **Operators** – Modify Data
- **Schedulers** – Where the work should be done, which thread like main thread etc.
- **Subscriber/Disposable** – where the response will be sent after work has been completed.

Examples

```
// Observable just is a factory method that emmits the values.

   private Observable<String> getObservable() {

      return Observable.just("Cricket", "Football");

   }

   private Observer<String> getObserver() {

      return new Observer<String>() {

         @Override

         public void onSubscribe(Disposable d) {
```

```
        }

        @Override

        public void onNext(String value) {

            System.out.println(value);

        }

        @Override

        public void onError(Throwable e) {

        }

        @Override

        public void onComplete() {

            System.out.println("onComplete");

        }

    };

}
```

Now, we have to connect both **Observables** and **Observer** with a **Subscription**. Only then can it actually do anything:

```
private void doSomeWork() {

    getObservable()

            // Run on a background thread

            .subscribeOn(Schedulers.io())

            // Be notified on the main thread

            .observeOn(AndroidSchedulers.mainThread())

            .subscribe(getObserver());

}
```

This will print the following

```
Cricket

Football

onComplete
```

This means the **Observable** emits two strings, one by one, which are observed by **Observer**.

So, here it goes to **onNext** two times and then after emitting all the values it goes to **onComplete**.

Types of Observables

Observables produce stream to be observed by an Observer when it subscribes. There are various types and ways this stream can be emitted and it depends on the use case. Some of those uses are as listed below:

- Items are to be emitted one by one.
- Items are to be emitted by controlling the producer's emission (with backpressure).
- Only one item needs to be emitted as part of success.
- Item has to be emitted based on conditions.

There are **5** types of observables as follows:

Observables: It emits 0…. N elements, and then completes successfully or with an error.

The **Observer** for the **Simple** type of **Observable** will be:

```
private Observer<String> getObserver() {

      return new Observer<String>() {

         @Override

         public void onSubscribe(Disposable d) {

         }

         @Override

         public void onNext(String value) {

         }

         @Override

         public void onError(Throwable e) {

         }
```

```
        @Override
        public void onComplete() {
        }
    };
}
```

Flowable: Similar to Observable but with a backpressure strategy.

So, the Observer for the Flowable type of Observable will be:

```
private Observer<String> getObserver() {
    return new Observer<String>() {
        @Override
        public void onSubscribe(Disposable d) {
        }
        @Override
        public void onNext(String value) {
        }
        @Override
        public void onError(Throwable e) {
        }
        @Override
        public void onComplete() {
        }
    };
}
```

Single: It completes with a value successfully or an error. (doesn't have onComplete callback, instead onSuccess (val)).

The Observer for the Single type of Observable will be:

```
private SingleObserver<Integer> getSingleObserver() {

      return new SingleObserver<Integer>() {

         @Override

         public void onSubscribe(Disposable d) {

         }

         @Override

         public void onSuccess(Integer value) {

         }

         @Override

         public void onError(Throwable e) {

         }

      };

   }
```

Maybe: It completes with/without a value or completes with an error.

The Observer for the Maybe of Observable will be:

```
private MaybeObserver<Integer> getMaybeObserver() {

      return new MaybeObserver<Integer>() {

         @Override

         public void onSubscribe(Disposable d) {

         }

         @Override

         public void onSuccess(Integer value) {

         }
```

```
        @Override
        public void onError(Throwable e) {
        }
        @Override
        public void onComplete() {
        }
    };
  }
```

Completable: It just signals if it has completed successfully or with an error.

The Observer for the Completable Observable will be:

```
private CompletableObserver getCompletableObserver() {
    return new CompletableObserver() {
        @Override
        public void onSubscribe(Disposable d) {
        }
        @Override
        public void onComplete() {
        }
        @Override
        public void onError(Throwable e) {
        }
    };
  }
```

So, we will have to choose which one to use on basis of our use-case.

Schedulers

RxJava Schedulers is all about managing thread at low level. They provide high-level constructs to manage with concurrency and deal with details by itself. They create workers who are responsible for scheduling and running code. By default, RxJava will not introduce concurrency and will run the operations on the subscription thread.

There are two methods through which we can introduce Schedulers into our chain of operations:

- **subscribeOn**: It specify which Scheduler invokes the code contained in Observable.cteate ().
- **observeOn**: It allows control to which Scheduler executes the code in the downstream operators.

RxJava provides some general use Schedulers:

- **Schedulers.computation ():** Used for CPU intensive tasks.
- **Schedulers.io ():** Used for IO bound tasks.
- **Schedulers.from (Executor):** Use with custom ExecutorService.
- **Schedulers.newThread ():** It always creates a new thread when a worker is needed. Since it's not thread pooled and always creates a new thread instead of reusing one, this scheduler is not very useful.

What are Operators?

Operators are basically a set of functions that can operate on any observable and defines the observable, how and when it should emit the data stream.

Few useful operators are as follows:

Map: It transforms the items emitted by an Observable by applying a function to each item.

Example: Let's say we are getting ApiUser Object from api server then we are converting it into User Object because may be our database support User Not ApiUser Object. Here we are using Map Operator to do that.

Zip: It combines the emissions of multiple Observables together via a specified function, then emits a single item for each combination based on the results of this function.

Example: Let's say we have to make two network call one for getting the list of users who loves cricket, another for users who loves football. And then returns the list of users who loves both. Here we are using Zip operator to do that.

Filter: It emits only those items from an Observable that pass a predicate test.

Example: Let's say our server return the list of my friends, but we need to filter out only who those friends who is following me. Here comes the filter operator to do so.

FlatMap: It transforms the items emitted by an Observable into Observables, then flatters the emissions from those into a single Observable.

Take: It emits only the first n items emitted by an Observable.

Reduce: It applies a function to each item emitted by an Observable, sequentially, and emits the final value.

Skip: It suppresses the first n items emitted by an Observable.

Buffer: It periodically gathers items emitted by an Observable into bundles and emits these bundles rather than emitting the items one at a time.

Concat: It emits the emissions from two or more Observables without interleaving them.

Replay: It ensures that all Observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items.

Merge: It combines multiple Observables into one by merging their emissions.

RxJava Subject

A Subject is a sort of bridge or proxy that is available in some implementations of ReactiveX that acts both as an observer and as an Observable. Because it is an observer, it can subscribe to one or more Observables, and because it is an Observable, it can pass through the items it observes by re-emitting them, and it can also emit new items.

There are 4 types of Subject available in RxJava.

- Publish Subject
- Replay Subject
- Behavior Subject
- Async Subject

Observable: Assume that a professor is an observable. The professor teaches about some topic.

Observer: Assume that a student is an observer. The student observes the topic being taught by the professor.

Publish Subject

It emits all the subsequent items of the source Observable at the time of subscription.

Here, if a student entered late into the classroom, he just wants to listen from that point of time when he entered the classroom. So, publish will be the best for this use case.

```
1.  PublishSubject<Integer> source = PublishSubject.create();
2.
3.    // It will get 1, 2, 3, 4 and onComplete
4.    source.subscribe(getFirstObserver());
5.
6.    source.onNext(1);
7.    source.onNext(2);
8.    source.onNext(3);
9.
10.   // It will get 4 and onComplete for second observer also.
11.   source.subscribe(getSecondObserver());
12.
13.   source.onNext(4);
14.   source.onComplete();
```

Replay Subject

It emits all the items of the source observable, regardless of when the subscriber subscribes.

Here, if a student entered late into classroom, he wants to listen from the beginning. So, here we will use Replay to achieve this.

```
1.  ReplaySubject<Integer> source = ReplaySubject.create();
2.     // It will get 1, 2, 3, 4
3.     source.subscribe(getFirstObserver());
4.     source.onNext(1);
5.     source.onNext(2);
6.     source.onNext(3);
7.     source.onNext(4);
8.     source.onComplete();
9.     // It will also get 1, 2, 3, 4 as we have used replay Subject
10.    source.subscribe(getSecondObserver());
```

Behavior Subject

It emits the most recently emitted item and all the subsequent items of the source Observable when an observer subscribes to it.

Here, if a student entered late into the classroom, he wants to listen the most recent things (not from the beginning) being taught by the professor so that he gets the idea of the context. So, here we will use Behavior.

```
1.  BehaviorSubject<Integer> source = BehaviorSubject.create();
2.     // It will get 1, 2, 3, 4 and onComplete
3.     source.subscribe(getFirstObserver());
4.     source.onNext(1);
5.     source.onNext(2);
6.     source.onNext(3);
7.     // It will get 3(last emitted)and 4(subsequent item) and onComplete
8.     source.subscribe(getSecondObserver());
9.     source.onNext(4);
10.    source.onComplete();
```

Async Subject

It only emits the last value of the source Observable (and only the last value).

Here, if a student entered at any point of time into the classroom, and he wants to listen only about the last thing (and only the last thing) being taught. So, here we will use Async.

```
1.  AsyncSubject<Integer> source = AsyncSubject.create();
2.     // It will get only 4 and onComplete
3.     source.subscribe(getFirstObserver());
4.     source.onNext(1);
5.     source.onNext(2);
6.     source.onNext(3);
7.     // It will also get only get 4 and onComplete
```

```
8.    source.subscribe(getSecondObserver());
9.    source.onNext(4);
10.   source.onComplete();
```

so, whenever you are stuck with these types of cases, the RxJava Subject will be your best friend.

Using Disposable in RxJava

As the android applications do many operations in background, many times it can happen that the activity no longer needs the data after sometime from that operation, in that case, there must have a way to unsubscribe to avoid memory leaks.

So, the RxJava gives us the Disposable to unsubscribe to avoid memory leaks.

Example, Let's say there is an observable with emits some values, but while emitting values, the use presses back button, then it must stop emitting values.

```
1.  public class DisposableExampleActivity extends AppCompatActivity {
2.      private final CompositeDisposable disposables = new CompositeDisposable();
3.
4.      @Override
5.      protected void onStop() {
6.          super.onStop();
7.          disposables.clear(); // do not send event after activity has been stopped
8.      }
9.      /*
10.      * Example to understand how to use disposables.
11.      * disposables is cleared in onDestroy of this activity.
12.      */
13.     void doSomeWork() {
14.         disposables.add(sampleObservable()
15.                 // Run on a background thread
16.                 .subscribeOn(Schedulers.io())
17.                 // Be notified on the main thread
18.                 .observeOn(AndroidSchedulers.mainThread())
19.                 .subscribeWith(new DisposableObserver<String>() {
20.                     @Override
21.                     public void onComplete() {
22.
23.                     }
24.                     @Override
25.                     public void onError(Throwable e) {
26.
27.                     }
28.                     @Override
29.                     public void onNext(String value) {
30.
31.                     }
32.                 }));
33.     }
34.     static Observable<String> sampleObservable() {
35.         return Observable.defer(new Callable<ObservableSource<? extends String>>() {
36.             @Override
37.             public ObservableSource<? extends String> call() throws Exception {
```

```
            // Do some long running operation
            SystemClock.sleep(2000);
            return Observable.just("one", "two", "three", "four", "five");
        }
    });
  }
}
```

This way memory leaks can be avoided by using Disposable.

Another important thing is that there are two methods one is `clear ()` and another is `dispose ()`.

1. `clear ()` will clear all, but can accept new disposable.
2. `Dispose ()`will clear all and set `isDisposed = true`, so it will not accept any new disposable.

Backpressure

In cases where a publisher is emitting items more rapidly than an operator or subscriber can consume them, then the items that are overflowing from the publisher need to be handled. If for example we try to maintain an ever-expanding buffer of items emitted by the faster publisher to eventually combine with items emitted by the slower one. This could result in OutOfMemoryError.

Backpressure relates to a feedback mechanism through which the subscriber can signal to the producer how much data it can consume and so to produce only that amount.

Example:

The Subscriber has an onSubscribe (Subscription) method, Subscription:request(long n), it is through this it can signal upstream that it's ready to receive a number of items and after it processes the items request another batch. There is a default implementation which requests **Long.MAX_VALUE** which basically means "**send all you have**". But we have not seen the code in the producer that takes consideration of the number of items requested by the subscriber.

```
Flowable.create(new FlowableOnSubscribe<String>() {
        @Override
        public void subscribe(@NonNull FlowableEmitter<String> emitter)
                throws Exception {
            int count = 0;
            while (true) {
                count++;
                emitter.onNext(count + "\n");
            }
        }
    }, BackpressureStrategy.DROP)
            .observeOn(Schedulers.newThread(), false, 3)
            .subscribe(new Subscriber<String>() {
                @Override
                public void onSubscribe(Subscription s) {

                }

                @Override
                public void onNext(String value) {
```

```
21.                 try {
22.                     Thread.sleep(1000);
23.                 } catch (InterruptedException e) {
24.                     e.printStackTrace();
25.                 }
26.             }
27.
28.             @Override
29.             public void onError(Throwable t) {
30.
31.             }
32.
33.             @Override
34.             public void onComplete() {
35.
36.             }
37.         });
```

The 2nd parameter **BackpressureStrategy** allows us to specify what to do in the case of overproduction.

- **BackpressureStrategy.BUFFER:** It buffers in memory the events that overflow. If we don't provide some threshold, it might lead to OutOfMemoryError.
- **BackpressureStrategy.DROP**: It simply drop the overflowing events
- **BackpressureStrategy.LATEST**: It keeps only recent event and discards previous unconsumed events.
- **BackpressureStrategy.ERROR**: We get an error in the Subscriber immediately.
- **BackpressureStrategy.MISSING**: We use this when we don't care about backpressure(we let one of the downstream operators onBackpressureXXX handle it)

By default the subscriber requests **Long.MAX_VALUE** since the code flowable.subscribe(onNext, onError, onComplete) uses a default onSubscribe. So unless we override onSubscribe, it would overflow. We can control this through operator **observeOn()** that make its own request to the upstream Publisher(256 by default), but can take a parameter to specify the request size.

In the above example, initially, the Subscriber requests Long.MAX_VALUE, but as the subscription travels upstream through the operators to the source Flowable, the operator observeOn subscribes to the source and requests just 3 items from the source instead. Since we used BackpressureStrategy.DROP, all the items emitted outside the 3, get discarded and thus never reach our subscriber.

There are also specialized operators to handle backpressure the onBackpressureXXX operators:

1. **onBackpressureBuffer**
2. **onBackpressureDrop**
3. **onBackpressureLatest**

```
.onBackpressureDrop(val -> System.out.println("Dropping " + val))
```

These operators request Long.MAX_VALUE(unbounded amount) from upstream and then take it upon themselves to manage the requests from downstream. In the case of *onBackpressureBuffer* it adds in an internal queue and sends downstream the events as requested, *onBackpressureDrop* just discards events that are received from upstream more than requested from downstream, *onBackpressureLatest* also drops emitted events excluding the last emitted event(most recent). The last onBackpressureXXX operator overrides the previous one if they are chained.

What is RxAndroid?

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

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 and 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.

In Android, updating UI from background thread is technically not possible, so using *AndroidSchedulers.mainThread ( )* we can update anything on the main thread. Internally it utilizes the concept of Handler and Looper to perform the action on the main thread.

RxAndroid uses RxJava internally and compiles it. But while using RxAndroid in our project we still add the dependency of RxJava to work with like:

```
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'

implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
```

Use cases

- RxJava has the power of operators and as the saying goes by,
- "RxJava has an operator for almost everything".

Case 1:

Consider an example, where we want to do an API call and save it to some storage/file. It would be a long-running task and doing a long-running task on the main thread might lead to unexpected behavior like App Not Responding.

So, to do the above-mentioned task we might think to use AsyncTask as our goto solution. But with Android R, AsyncTask is going to deprecated, and then libraries like RxJava will be solution for it.

Using RxJava over AsyncTask helps us to write less code. It provides better management of the code as using AsyncTask might make the code lengthy and hard to manage.

Case 2:

- Consider a use-case where we might want to fetch user details from an API and from the user's ID which we got from the previous API we will call another API and fetch the user's friend list.
- Doing it using AsyncTask we might have to do use multiple AsyncTask and manage the results in was way where we want to combine all the AsyncTask to return the result as a single response.
- But using RxJava we can use the power of **zip** operator to combine the results of multiple different API calls and return a single response.

Case 3:

- Consider an example of doing an API call and getting a list of users and from that, we want only the data which matches the given current condition.
- A general approach is to do the API call, and from the collection, we can then filter the content of that specific user based on the condition and them return the data.
- But using RxJava we can directly filter out the data while returning the API responses by using the **filter** operator and we do all of this by doing the thread management.

The Three O's

- **Observables** is the one who pushes the data towards its Observer.
- **Observer** keeps following Observables. So, whenever Observables pushes or emits any item or data, the Observer gets notified along with the data and then he can take any action.
- And, **Operator** here can perform the operation of manipulation, transformation and merge different data together.

Using RxAndroid

RxJava is a library that lets us implement reactive programming and, as such, create reactive applications. RxJava2 is an update to the earlier version of RxJava. In RxJava2, we have **Observables**, **Observers** and **Schedulers**.

Observables are the data sources and they exist in various types: **Observer**, **Single**, **Flowable**, **Maybe** and **Completable**.

Notably, the **Flowable** comes with a backpressure support. Backpressure is when an Observer can signal to the Observable that the latter is emitting values too fast. Observers are the data receivers (or consumers) while Schedulers help to manage threads.

RxAndroid on the other hand, is an extension of Rxjava2. It offers functionalities just peculiar to the Android platform, like the provision of a Scheduler that schedules on the main thread or any given Looper.

To use RxAndroid, we need to add the dependency to our app-module `build.gradle` file:

```
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
// Because RxAndroid releases are few and far between, it is recommended you also
// explicitly depend on RxJava's latest version for bug fixes and new features.
compile 'io.reactivex.rxjava2:rxjava:2.1.7'
```

Then, we create an instance of a `CompositeDisposable` in our class. The `CompositeDisposable` is simply a container that can hold multiple disposables:

```
disposables.add(
Observable.just("Hello world!")
// Run on a background thread
.subscribeOn(Schedulers.io())
// Be notified on the main thread
.observeOn(AndroidSchedulers.mainThread())
.subscribeWith(new DisposableObserver<String>() {
@Override
public void onComplete() {

}
@Override
public void onError(Throwable e) {

}
@Override
public void onNext(String value) {

}
})
);
```

In the code snippet above, what we added to our `disposables` variable consists of the data source which will emit one string, the thread where the process will take place, the thread where our observer will be notified of the result, and our subscriber/observer. The `DisposableObserver <String>` is our observer, which provides us with three implemented methods.

First is `onNext` which is called whenever data is emitted, next is `onError` to show that an error occurred, and finally `onCompleted` to show that the `Observable` has finished emitting data and won't call `onNext` anymore.

We then clear resources when our activity is in background to avoid memory leaks:

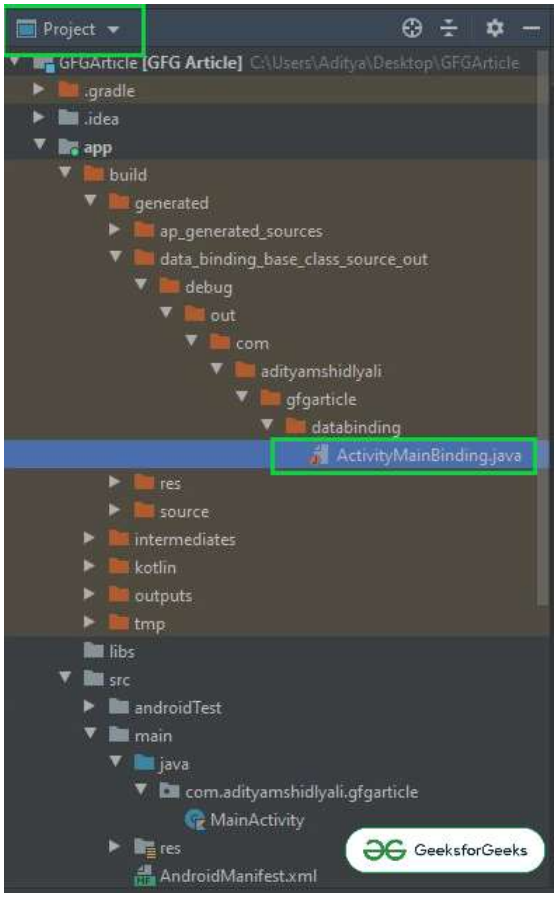
```
@Override
protected void onPause() {
super.onPause();
disposables.clear();
}
```

Android View Binding

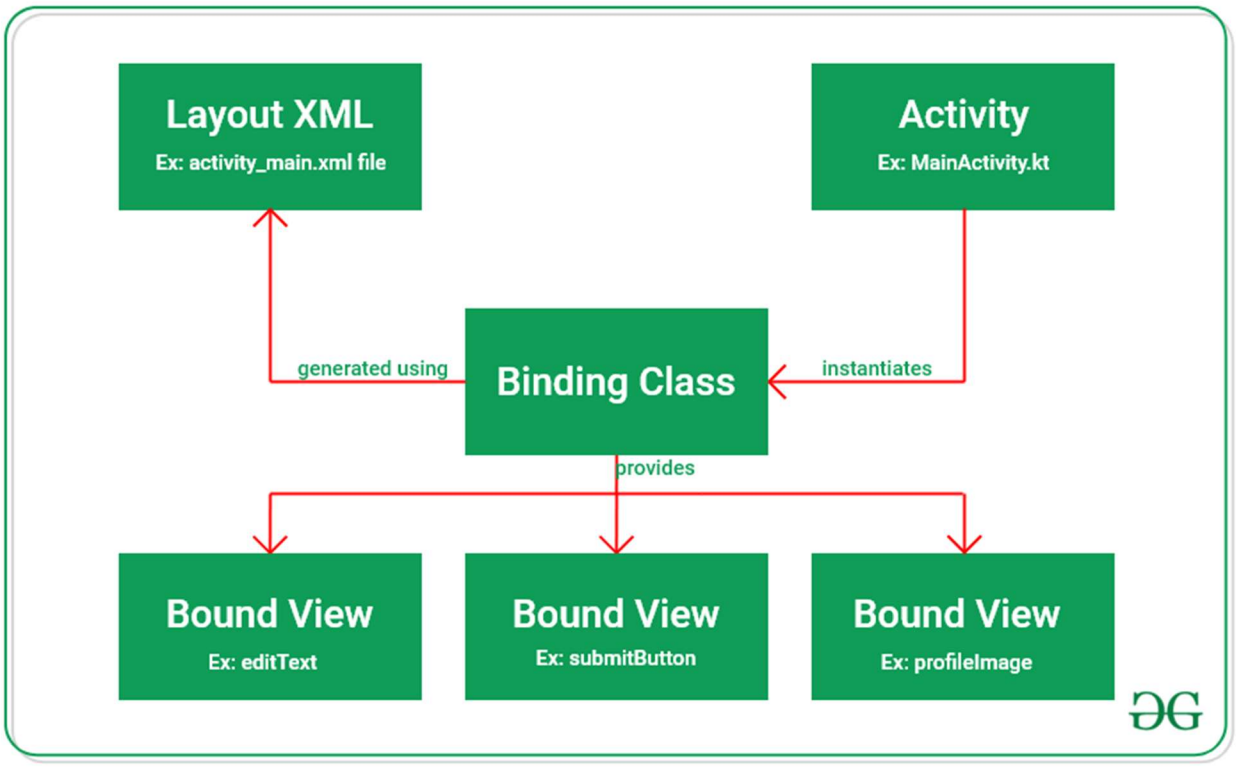
View Binding is one of the best features which provides the views to bind with the activity which is ongoing. Replacing the `findViewById ()` method, hence reducing the boilerplate code, generated the instances of the views of the current layout. And most important feature about View Binding is it's always null safe.

View Binding features in Android

- ViewBinding is always null safe and type-safe, which supports both Java and Kotlin.
- ViewBinding is introduced in the Gradle version 3.6 and above.
- ViewBinding also helps to reduce the boilerplate code, hence reducing the code redundancy.
- While using the ViewBinding proper naming convention are need to be followed because it creates the binding class internally using the name of the same layout file. Naming the layout file in the snake case is preferred. For Example, the ViewBinding creates **activity_main.xml** (snake case) file as **ActivityMainBinding** (Pascal case), which contains all the property and instances of all the views containing in that layout.
- Ans also whatever IDs of all elements are created inside the layout XML file, the ViewBinding converts them to camel case. For example: **android:id=" button_submit" -> buttonSubmit**. Which is much useful in the code readability?
- Using ViewBinding the compilation of the code is a bit faster as compared to the traditional `findViewById ()` method.
- The **ActivityMainBinding** class is generated in the following path under the project hierarchy this can be viewed.



The following is the flow of how the objects of the properties from the layout are generated.



View Binding libraries emerged when there was a need to reduce the boilerplate code when assigning views to variables and having access to them in the activity class. Libraries in this area are limited. Basically, there are two that worth mentioning: **ButterKnife** and **Android Databinding**.

Butterknife

ButterKnife is a view binding library developed by Jake Wharton. Butterknife is a library that helps us assign `ids` to views easily thereby avoiding the excess `findViewById`. According to the documentation, "**ButterKnife is like Dagger only infinitely less sharp**". This means that view binding can be seen as a form of dependency injection. In ButterKnife, annotations are used to generate boilerplate code for us instead.

Using Butterknife

To use Butterknife, we need to add the dependencies in our app-module `build.gradle` file as follows:

```
implementation 'com.jakewharton:butterknife:8.8.1'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.8.1'
```

Then, in our activity, we use the `@BindView` annotation to assign an `id` to its view:

```
class MainActivity extends AppCompatActivity {
   @BindView(R.id.firstname) EditText firstName;

   @Override public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main_activity);
      ButterKnife.bind(this);
      // TODO ...
   }
}
```

Note that, in the code snippet above, we initialized ButterKnife in our `onCreate` method by using its `bind` method. Alternatively, if we want to use `ButterKnife` in a `Fragment`, we initialize it in the `onCreateView` method this way:

```
View view = inflater.inflate(R.layout.sample_fragment, container, false);
ButterKnife.bind(this,view);
```

With ButterKnife, we also avoid creating `OnClickListeners`. For instance, we can use an `@OnClick` annotation together with the view to add a click listener to a view:

```
@OnClick(R.id.button)
void buttonClicked() {
  // TODO ...
}
```

Android Databinding Library

The Android Databinding Library is inbuilt to the Android Support Library. It requires at least Android Studio version 1.3 to work. This library, unlike ButterKnife, does not make use annotations.

Using Databinding Library

Enable data binding in the app-module `build.gradle` file and sync:

```
android {
   ....
   dataBinding {
      enabled = true
   }
}
```

Thereafter, we set the root tag of our layout file to `layout`:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools">

    <android.support.constraint.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/textview"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

    </android.support.constraint.ConstraintLayout>

</layout>
```

An activity binding class is then generated for us (`ActivityMainBinding`) based on the naming of the layout: `activity.main.xml`. we will use an instance of this class to access our views. We also have another class, `DataBindingUtil`, generated to handle other utilities.

Then in our activity class:

```
public class MainActivity extends AppCompatActivity {

    ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
        binding.textview.setText("Hello world!");
    }
}
```

The Android Databinding Library offers a replacement for `onClick` listeners together with a whole lot of other features.

These two libraries, ButterKnife and Android Databinding Library, significantly reduce the amount of code written to access views. However, the Android DataBinding library looks like a winner here because it is easier to setup, it achieves the result with fewer code as compared to ButterKnife and offers more functionalities.

Android Testing

Testing is simply executing a software with the aim of finding bugs. Testing has evolved over time and has gone beyond just being one of the final stages of a software development process. In fact, testing could be adopted as an early part of the coding stage where tests are written first, then the logic of the software is then implemented to only pass what the test expects. This is usually referred to as **Test Driven Development**. It is good practice to write tests for our applications as it helps us to spot bugs quickly and enhance our application.

There are a number of test libraries with different strengths available for Android development. Let's take a look at four of them: Junit, Mockito, Robolectric and Espresso.

Junit

Junit is a framework used for unit testing. Unit testing is a type of testing where individual units of the source code are tested. The framework contains a set of `assert` methods to check an expected result. Junit makes heavy use of annotations. Just to name a few, we have `@Test` (to identify a test method), `@Before` (to declare a method that should be called before a test is called), and `@After` (to declare a method that should be called after a test).

Using Junit

First, we add the dependency in our app-module `build.gradle` file:

```
testImplementation 'junit:junit:4.12'
```

then our sample test class looks like this:

```
public class ExampleUnitTest {
    @Test
    public void additionIsCorrect() throws Exception {
        assertEquals(4, 2 + 2);
    }
}
```

Here, we checked to affirm that the addition of two and two equals four. Junit tests are usually very fast because they run on the JVM and don't require the device or an emulator,

Mockito

Most times, the classes we intend to write tests for depend on other classes. Configuring these classes just for this purpose can be hectic. This is where Mockito comes in. It is a mocking framework that helps us create and configure mock (fake) objects. It is usually used with together with Junit.

Using Mockito

First, we need to make sure the Jcenter repository, `jcenter ()` is in our project `build.gradle` file. Next, we add the dependencies to app-module `build.gradle` file and sync:

```
testCompile 'junit:junit:4.12'
// required if we want to use Mockito for unit tests
testCompile 'org.mockito:mockito-core:2.13.0'
// required if we want to use Mockito for Android tests
androidTestCompile 'org.mockito:mockito-android:2.13.0'
```

Then we use Mockito like that:

```
@Test
public void mockitoTest throws Exception {
   List mockedList = mock(List.class);

   //using mock object
   mockedList.add("one");
   mockedList.clear();

   //verification
   verify(mockedList).add("one");
   verify(mockedList).clear();
}
```

Here, we mocked a list, added data to it, and cleared it. We then verified that these actions were performed.

Espresso

Espresso is a test framework which is part of the Android Testing Support Library. This test framework allows us to create user interface tests for our android apps. This means that, with Espresso, we can write tests that can check if the text of a `TextView` matches another text. Espresso tests run on both actual devices and emulators and behave as if an actual user was using the app.

Using Espresso

First, we add these dependencies to our app-module `build.gradle` file:

```
androidTestCompile 'com.android.support.test.espresso:espresso-core:3.0.1'
androidTestCompile 'com.android.support.test:runner:1.0.1'
```

Then, still in the same `gradle` file, we set the instrumentation runner. Let's not forget to sync our Gradle files after that:

```
defaultConfig {
    applicationId "com.my.awesome.app"
    minSdkVersion 15
    targetSdkVersion 26
    versionCode 1
    versionName "1.0"

    testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
}
```

Then, we can create a test file (e.g., `EspressoSampleTest`) that will look like this:

```
@RunWith(AndroidJUnit4.class)
public class EspressoSampleTest {
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule =
            new ActivityTestRule(MainActivity.class);

    @Test
    public void isHelloWorldDisplayed() {
        onView(withText("Hello world!"))
          .check(matches(isDisplayed()));
    }
}
```

This test class checks to see if "Hello World!" is displayed when `MainActivity` is opened.

From our study of the various test libraries, we deduce the Junit competes with no one as it works together with other libraries. Espresso gives us a good platform for our user interface tests. Both Mockito and Robolectric have similar capabilities, but Robolectric achieves testing with fewer code. Robolectric also offers more functionalities like being able to test views. Robolectric also have advantage of being supported by Google Engineers.

Custom Fonts

Almost every android developer is passionate about the look and feel of their app. Sometimes we might need to go the extra mile into choosing a unique font for the app to give it the same feel across all devices. In situations like this, there are some libraries that can help us to use a custom font for all our texts in the app.

Calligraphy

Calligraphy is one of the most popular custom font libraries available and it is easy to get along with. With this library, we can easily declare a signal font across our whole application or define fonts individually to a text.

Using Calligraphy

As usual, we add the dependency in our app-module `build.gradle` file and sync it:

```
implementation 'uk.co.chrisjenx:calligraphy:2.3.0'
```

then, we create an `assets` folder and insert our custom font there. We can do that by right-clicking the app root folder in our project directory in android studio, select `New`, choose `Folder`, and then `Assets Folder`. This will generate an assets folder. Therefore, we initialize the library and set our default font in the `Application` class:

```
public class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        CalligraphyConfig.initDefault(new CalligraphyConfig.Builder()
                .setDefaultFontPath("red-velvet.ttf")
                .setFontAttrId(R.attr.fontPath)
                .build()
        );
    }
}
```

Finally, we override the `attachBaseContext` method in each of our activities:

```
@Override
protected void attachBaseContext(Context newBase) {
    super.attachBaseContext(CalligraphyContextWrapper.wrap(newBase));
}
```

And we are good to go! This gives us the Red Velvet font as our app's default font. We can also decide to apply a particular font to a single text like this:

```
<TextView
    android:text="@string/hello_world"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    fontPath="fonts/red-velvet.ttf"/>
```

Custom Fonts with Support Library

Thanks to the Android Support Library, from version `26` upward, we can make use of custom fonts without having to increase our app dependencies. This is so because in one way or other way we find the `appcompat` dependency in our `build.gradle` file. It is automatically added when creating a new android project.

Using Custom Fonts with Support Library

We add the dependency in our app-module `build.gradle` file and sync it:

```
implementation 'com.android.support:appcompat-v7:26.1.0'
```

First, we create a `fonts` resource folder. We can do this by right-clicking the `res` folder and then choose "New -> Android resource" directory. After that, we have to choose font as the resource type and select OK. We then add the desired font files in the `font` resource directory (e.g., `redvelvet`).

We can apply the custom fonts directly in our XML layouts:

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:fontFamily="@font/redvelvet"/>
```

additionally, we can create a new font family, a font family is a set of font files along with their style and weight details. Right click on the font source folder, select `New` and then `New Font Resource File`, insert a name and select OK. A sample font family looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<font-family xmlns:app="http://schemas.android.com/apk/res-auto">
   <font
      app:font="@font/redvelvet"
      app:fontStyle="normal"
      app:fontWeight="400" />
</font-family>
```

When we apply a font to our `XML` layout, the system picks the correct font based on th text style we used. Apart from applying the fonts in the `XML` layouts, we can also apply them programmatically and in multiple forms.

Job Scheduling

More often than not, our android applications might need to perform operations out of the user's interaction. This requires handling tasks asynchronously and intelligently to optimize the app's performance and the device in general. This can equally be called handling tasks in the background.

Android has its own API for scheduling background tasks, JobScheduler, which comes with a drawback, it can only be used when supporting API 21 i.e., android 5.0 or later.

Two other common options in this category are **Android job library** by Evernote and **Firebase Jobdispatcher** by Firebase.

Firebase Jobdispatcher comes at another cost which is the need for Google Play Services. However, it is compatible all the way back to API 9 (Android 23). Android job combines the effort of Android's JobScheduler and Firebase Jobdispatcher to provide a firm job scheduling library.

Android-job

Android job is an android library used to handle jobs in the background. Depending on the android version either the `JobScheduler`, `GcmNetworkManager` or `AlarmManager` will be used. This is why this library wins the heart of all. Instead of using separate APIs within one codebase and checking for API versions to know which scheduling API to use, Android job reduces our code size together with the stress and does this for us, this library requires API 14 (Android 4.0) or later.

Using Android job

To use android job, we add the dependency in our app-module `build.gradle` file and sync it:

```
compile 'com.evernote:android-job:1.2.1'
```

Then, we initialize our `JobManager` in the `Application` class:

```
public class App extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        JobManager.create(this).addJobCreator(new SampleJobCreator());
    }
}
```

The `Application` class is used to initialize objects just once throughout the app lifecycle. The `SampleJobCreator` is a class which returns instances of a `Job` based on the Job's unique tag. The `SampleJobCreator` can look like this:

```
public class SampleJobCreator implements JobCreator {
    @Override
    @Nullable
    public Job create(@NonNull String tag) {
        switch (tag) {
            case FirstJob.TAG:
                return new FirstJob();
            default:
                return null;
        }
    }
}
```

Finally, our `Job` class named `FirstJob` looks like this:

```
public class FirstJob extends Job {
    public static final String TAG = "first_job_tag";
    @Override
    @NonNull
    protected Result onRunJob(@NonNull Params params) {
        // run your job here
        return Result.SUCCESS;
    }

    public static void scheduleJob(long timeJobShouldStart) {
        new JobRequest.Builder(ReviewStayJob.TAG)
                .setExact(timeJobShouldStart)
                .build()
                .schedule();
    }
}
```

In the code snippet above, we all the `Job` what do in the `onRunJob` method. Then, we schedule a job by just calling `FirstJob.scheduleJob (timeMills` (where `timeInMillis` is of datatype `long`).

Android Security

Auth0

Securing applications with Auth0 is very easy and brings a lot of great features to the table. With Auth0, we only have to write a few lines of code to get solid identity management solution, single sign-on, support for social identity providers (like Facebook, GitHub, Twitter etc), and support for enterprise identity providers (Active Directory, LDAP, custom etc).

Dependencies

To import this library, we have to include the following dependency in our `build.gradle` file:

```
dependencies {
    compile 'com.auth0.android:auth0:1.12.0'
}
```

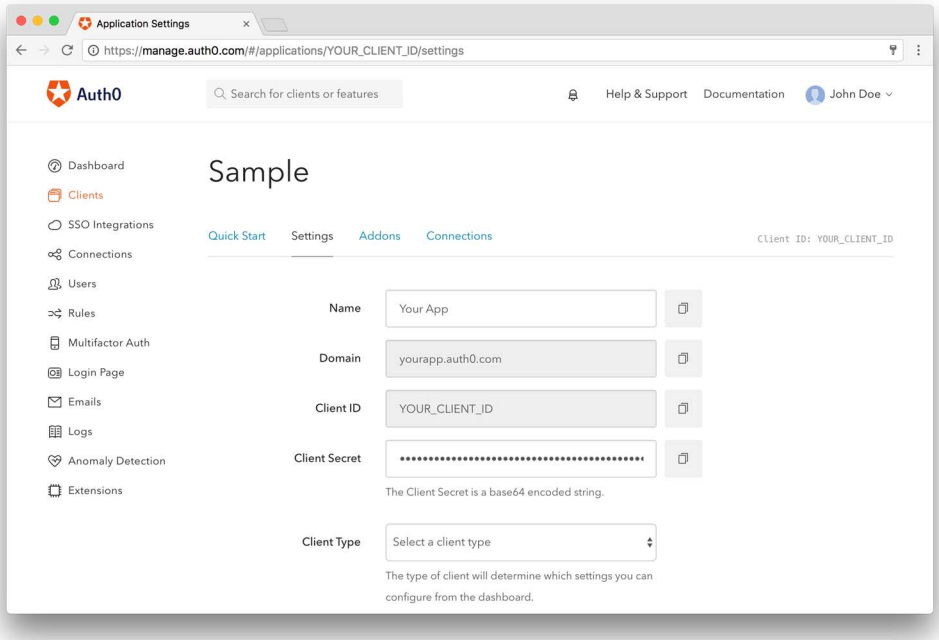
After that, we need to open our app's `AndroidManifest.xml` file and add the following permission:

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

Create an Auth0 Application

After importing the library and adding the permission, we need to register the application in our Auth0 dashboard. By the way, if we don't have an Auth0 account, this is a great time to create a free one.

In the Auth0 dashboard, we have to go to **Applications** and then click on the **Create Application** button. In the form that is shown, we have to define a name for the application and select the **Native** type for it. After that, we can hit the **Create** button. This will lead us to a screen similar to the following one:



On this screen, we have to configure a callback URL. This is a URL in our android app where Auth0 redirects the user after they have authenticated.

We need to whitelist the callback URL for our android app in the **Allowed Callback URLs** filed in the **Setting** page of our Auth0 application. If we do not set any callback URL, our users will see a mismatch error when they log in.

```
demo://bkrebs.auth0.com/android/OUR_APP_PACKAGE_NAME/callback
```

Let's not forget to replace **OURAPPPACKAGEN_NAME** with our android application's package name. we can find this name in the `applicationId` attribute of the `app/build.gradle` file.

Set Credentials

Our android application needs some details from Auth0 to communicate with it. We can get these details from the **Settings** section for our Auth0 application in the Auth0 dashboard.

We need the following information:

- **Client ID**
- **Domain**

It's suggested that we do not hardcore these values as we may need to change them in the future. Instead, let's use String Resources, such as `@string/com_auth0_domain`, to define the values.

Let's edit our `res/values/strings.xml` file as follows:

```
<resources>
    <string name="com_auth0_client_id">2qu4Cxt4h2x9In7Cj0s7Zg5FxhKpjooK</string>
    <string name="com_auth0_domain">bkrebs.auth0.com</string>
</resources>
```

These values have to be replaced by those found in the **Settings** section of our Auth0 application.

Android Login

To implement the login functionality in our android app, we need to add manifest placeholders required by the SDK. These placeholders are used internally to define an `intent-filter` that captures the authentication callback URL configured previously.

To add the manifest placeholders, let's add the next line:

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 25
    buildToolsVersion "25.0.3"
    defaultConfig {
        applicationId "com.auth0.samples"
        minSdkVersion 15
        targetSdkVersion 25
        //...

        //---> Add the next line
```

```
        manifestPlaceholders = [auth0Domain: "@string/com_auth0_domain", auth0Scheme:
"demo"]
        //<---
    }
}
```

After that, we have to run Sync Project with Gradle Files inside Android Studio or execute `./gradlew clean assembleDebug` from the command line.

Start the Authentication Process

The Auth0 login page is the easiest way to set up authentication in our application. It's recommended using the Auth0 login page for the best experience, best security, and the fullest array of features.

Now we have to implement a method to start the authentication process. Let's call this method `login` and add it to our `MainActivity` class.

```
private void login() {
    Auth0 auth0 = new Auth0(this);
    auth0.setOIDCConformant(true);
    WebAuthProvider.init(auth0)
                .withScheme("demo")
                .withAudience(String.format("https://%s/userinfo",
getString(R.string.com_auth0_domain)))
                .start(MainActivity.this, new AuthCallback() {
                    @Override
                    public void onFailure(@NonNull Dialog dialog) {
                      // Show error Dialog to user
                    }

                    @Override
                    public void onFailure(AuthenticationException exception) {
                      // Show error to user
                    }

                    @Override
                    public void onSuccess(@NonNull Credentials credentials) {
                        // Store credentials
                        // Navigate to your main activity
                    }
                });
}
```

As we can see, we had to create a new instance of the Auth0 class to hold user credentials. We can use a constructor that receives an Android Context if we have added the following String resources:

- `R.string.com_auth0_client_id`
- `R.string.com_auth0_domain`

If we prefer to hardcore the resources, we can use the constructor that receives both strings. Then, we can use the `WebAuthProvider` class to authenticate with any connection enabled on our application in the Auth0 dashboard.

After we call the `WebAuthProvider#start` function, the browser launches and shows the Auth0 login page. Once the user authenticates, the callback URL is called. The callback URL contains the final of the authentication process.

Capture the Result

After authentication, the browser redirects the user to our application with the authentication result. The SDK captures the result and parses it.

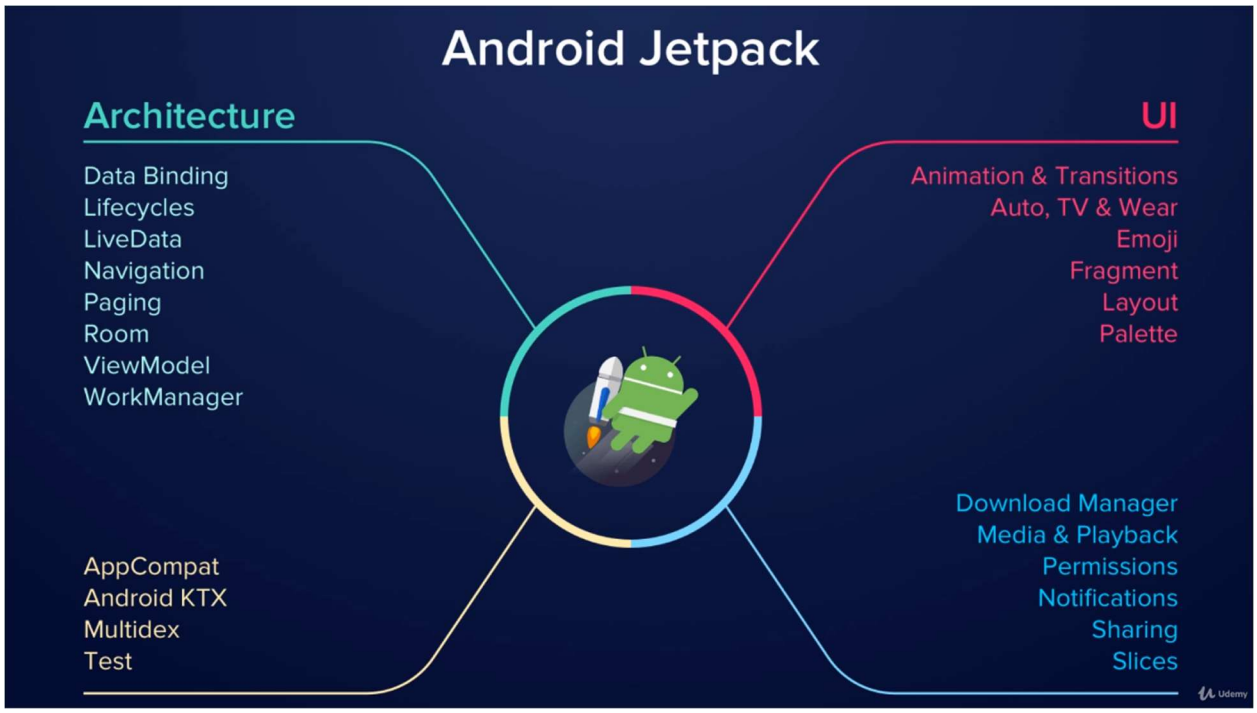
We do not need to declare a specific `intent-filter` for our activity because we have defined the manifest placeholders with, we Auth0 Domain and Scheme values.

The `AndroidManifest.xml` file should look like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.auth0.samples">
    <uses-permission android:name="android.permission.INTERNET" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
        <activity android:name="com.auth0.samples.MainActivity">
          <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
    </application>
</manifest>
```

That's it, we now have an android application secured with Auth0.

Android Jetpack



What is Android Jetpack?

Android Jetpack is a collection of Android software components which helps us in building great Android apps.

These software components help in:

- Following the best practices and writing the boilerplate code (a boilerplate is a unit of writing that can be reused over and over without change or with small change).
- Making complex things very simple.

Earlier there were many challenges which are as follows:

- Managing activity lifecycles.
- Surviving configuration changes.
- Preventing memory leaks.

All the major problems have been solved by the Android Jetpack's software components.

So, the solution for all the problems is Android Jetpack.

Another most important thing about the Jetpack is that it gets updated more frequently than the Android platform so that we always get the latest version.

Jetpack comprises the androidx package libraries, unbundled from the platform APIs. This means that it offers backward compatibility.

Android Jetpack Components

Android Jetpack components are a collection of libraries that are individually adoptable and built to work together while taking advantage of Kotlin language features that make us more productive.

These software components have been arranged in 4 categories which are as follows:

- Foundation Components
- Architecture Components
- Behaviour Components
- UI Components

Ways to include android jetpack libraries in the application

Add google repository in the `build.gradle` file of the application project.

```
allprojects {
repositories {
            google ()
            jcenter ()
            }
}
```

All Jetpack components are available in the Google Maven repository, include them in the `build.gradle` file.

```
allprojects {
repositories {
            jcenter ()
            maven {url 'https://maven.google.com'}
            }
}
```

Foundation Components

The foundation components provide the following:

- Backward compatibility
- Testing
- Kotlin language support

The following are the list of all Foundation Components:

1. AppCompat
2. AndroidKTX
3. Test
4. Multidex

AppCompat

The AppCompat library in Jetpack foundation includes all of the components from the **v7 library**. This includes AppCompat, CardView, GridLayout, MediaRouter, Palette, RecyclerView, Renderscript, Preferences, Leanback, Vector Drawable, Design, Custom tabs etc.

Moreover, this library provides implementation support to the **material design user interface** which makes AppCompat very useful for the developers.

Following are some key areas of an android application that are difficult to build but can be designed effortlessly using the AppCompat library:

- **App bar:** The topmost element in an application is its app bar/action bar which generally contains the name of the application or current activity name. AppCompat library facilitates the developers to design a more customized App bar. It provides a toolbar for designing and painting the background of the app bar with primary colors or gradient background (multiple colors).

- **Navigation Drawer:** It is the left pane menu that provides an easy navigation guide of the application. This is a very common design pattern now a days, and so it is needed to add very often. By using the AppCompat library, it is very straightforward to manage the colors and icons of the navigation drawer. In order to create a navigation drawer, developers just required to add DrawerLayout with the main view and a NavigationView. Further, this library also assists in managing the active state of each menu element with XML resources.

- **Permissions:** In the newer version of android OS, applications ask permission from users in order to access device hardware such as camera, microphone etc or to perform certain actions. The further operations performed by the application depend upon the choice selected by the user i.e., whether permission is granted or not. However, this was not the case before the **Marshmallow** (6.0) version of the Android OS.

  In older versions, developers do not need to write any code in the SDK which asks permission from the users and check whether it is granted or not. This problem has been solved by the **ContextCompat** (a component of AppCompat) which checks the permission status of an operation and request from the user if needed irrespective of the android OS version.

- **Resources**: Previously, more lines of code are needed to design the XML resources like drawable resources which are supported by a specific Android OS version. Using **ContextCompat** (part of AppCompat), the number of lines of code reduce drastically as well as the dependency of drawable on the OS version also removed.

- **Dialog box:** The AppCompat library has an **AppCompatDialog** component which is similar to the toolbar. It helps in making the material design dialog box in an application.

- **Share actions:** Sharing of any document/file from one application to another platform is possible because of the AppCompat library. The list of share actions like Gmail, Facebook, message etc which appears on the screen when a document/file is selected to share is provided by the activity toolbar of the **AppCompat** library.

AndroidKTX

This library is the only one among the foundation components which was introduced for the first time with the release o the Jetpack. AndroidKTX is a collection of Kotlin extensions that are designed to facilitate developers to remove boilerplate code as well as write concise code while developing android applications with Kotlin language. Here **KTX** in the name stands for **Kotlin Extensions**. Below is an example of a piece of code without using and after using the Android KTX library.

Code snippet of SQLite without using KTX library:

```
1.  db.beginTransaction()
2.  try {
3.   // insert data
4.   db.setTransactionSuccessful()
5.  }
6.  finally {
7.   db.endTransaction()
8.  }
```

Above code after using KTX library:

```
1.  db.transaction {
2.    // insert data
3.  }
```

This example clearly depicts how the Andorid KTX library reduced the lines of code and gracefully organized the SQLite transactions using a simple function along with a training lambda. Many of the Jetpack libraries are linked with various Android KTX modules. For example, one can use the below extensions while working with the Navigation component of the Android Jetpack.

- android.arch.navigation:navigation-common-ktx
- android.arch.navigation:navigation-fragment-ktx
- android.arch.navigation:navigation-runtime-ktx
- android.arch.navigation:navigation-ui-ktx

Test

This part of the foundation component includes the **Espresso UI** testing framework for the runtime UI test and **AndroidJUnitRunner** for the purpose of unit testing of Android applications. Espresso is primarily used for testing of UI elements. Further, AndroidJUnitRunner performs small tests on the logic of each and every method. It assures the fast and accurate testing of a piece of logic within the project code.

Multidex

The Multidex property of Android is one of the most important features while developing mobile applications. **Dex** is the format of the executable file which runs on **the Android virtual machine (known as Dalvik)**. To make a .dex file according to the Dalvik Executable specification, it must **not contain more than 65,536** methods considering all libraries of the project.

While making any real-life mobile application, the count of methods in project libraries can easily cross that number. In this situation, developers take the help of Multidex library which performs the system split of the .dex file of the application into multiple .dex files. Further, the Multidex component also provides support to the collective dex files of an application.

Detecting the need for Multidex in the application:

It is impractical for everyone to keep a record of method count while developing an application. Thus, to know if there is a need for Multidex in the project files or not, one should look out for the following errors. If the IDE shows one of these errors while building the application files, then it means there is a need to enable Multidex.

Error Message 1:

```
trouble writing output:

Too many field references: 131000; max is 65536.

You may try using –multi-dex option.
```

Error Message 2:

```
Conversion to Dalvik format failed:

Unable to execute dex: method ID not in [0, 0xffff]: 65536
```

Enable Multidex:

The code to enable Multidex depends upon the minimum SDK version of the android app. For different SDK version, below is the code snippet of the `build.gradle` file:

For minimum SDK version equal to or above 21:

```
1.  android{
2.    // ...
3.    defaultConfig {
4.      // ...
5.      minSdkVersion 21
6.      targetSdkVersion 28
7.      multiDexEnabled true
8.    }
9.    // ...
10. }
```

For minimum SDK version less than 21:

For AndoridX

```
android {
  defaultConfig {
    // ...
    minSdkVersion 18
    targetSdkVersion 28
    multiDexEnabled true
  }
  // ...
}
dependencies {
  // ...
  implementation 'androidx.multidex:multidex:2.0.0'
}
```

For old Support Library

```
android {
  defaultConfig {
    // ...
    minSdkVersion 18
    targetSdkVersion 28
    multiDexEnabled true
  }
  // ...
}
dependencies {
  // ...
  implementation 'com.android.support:multidex:1.0.3'
}
```

Architecture Components

The architecture components help us in building:

- Robust Apps
- Testable Apps
- Maintainable Apps

Further, Architecture Components could be classified as:

1. Room
2. WorkManager
3. Lifecycle
4. ViewModel
5. LiveData
6. Navigation
7. Paging
8. DataBinding

Android Jetpack with its architecture component defines some key principles which one should follow as a secure path to develop good and robust mobile apps. It does not support any particular architecture pattern but suggests clear separation of concerns and controlling of UI from Model. By following these rules, developers can avoid problems related to lifecycle and it will be test and maintain the application.

Following are the architecture component's element and their responsibilities:

1. The **View layer** is represented by the Activity/Fragment. They only deal with user interaction and observes as well as exhibits the **LiveData** element which is taken from the **ViewModel**.

2. **ViewModel** keeps a check on the Lifecycle of **View** and is responsible for maintaining data consistency during configuration changes in the device or other android lifecycle events.

3. The **Repository** is a class with no proper implementation and is responsible for gathering data from all the sources. It handles all the data and transforms them into observables **LiveData** and makes it accessible to **ViewModel**.

4. **Room** is an SQLite mapping library that overcomes the challenges of SQLite database like writing boilerplate codes, and query checking at compile time. It has the stability to return queries directly with observable **LiveData**.

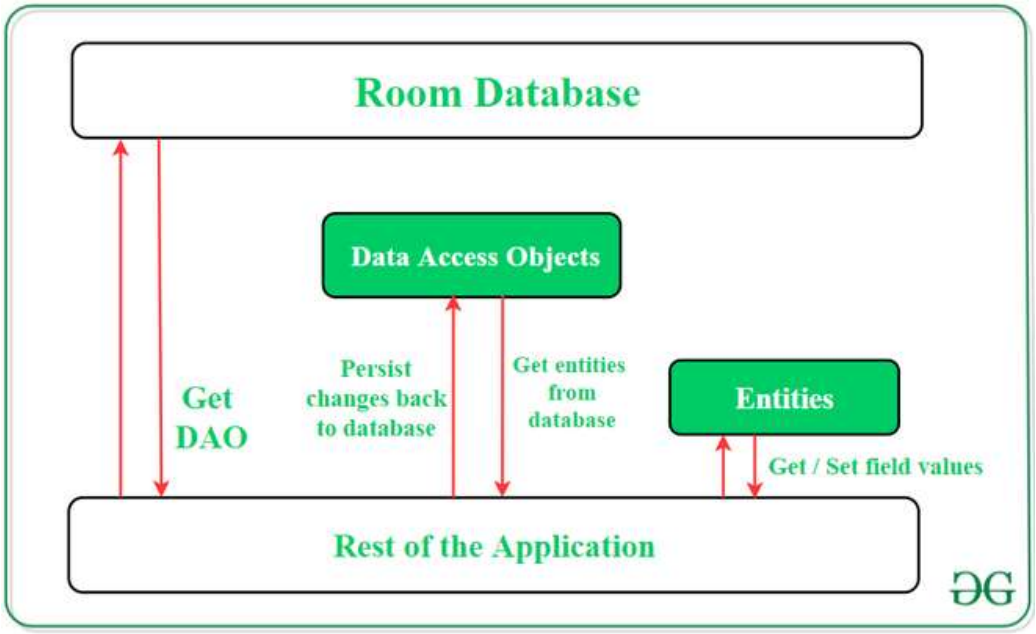
Room Component

The requirement of a database in android is fulfilled by SQLite from the very beginning. However, it comes with some severe drawbacks like not checking the queries at compile time, it does not save plain-old-Java-objects (commonly referred to as POJOs). Developers also need to write a lot of boilerplate code to make the SQLite database work in the android OS environment. The Room component comes into the picture as an **SQLite Object Mapping Library** which overcomes all the mentioned challenges. Room converts queries directly into objects, check errors in queries at the compile time, and is also capable of persisting the Java POJOs.

Moreover, it produces LiveData results/observables from the given query result. Because of this versatile nature of the Room component, Google officially supports and recommends developers to use it.

The Room consists of the following sub-components:

1. **Entity**: It is the annotated class for which the Room creates a table within the database. The field of the class represents columns in the table.
2. **DAO (Data Access Objects)**: It is responsible for defining the methods to access the database and to perform operations.
3. **Database**: It is an abstract class that extends **RoomDatabase** class and it serves as the main access point to the underlying apps relational data.



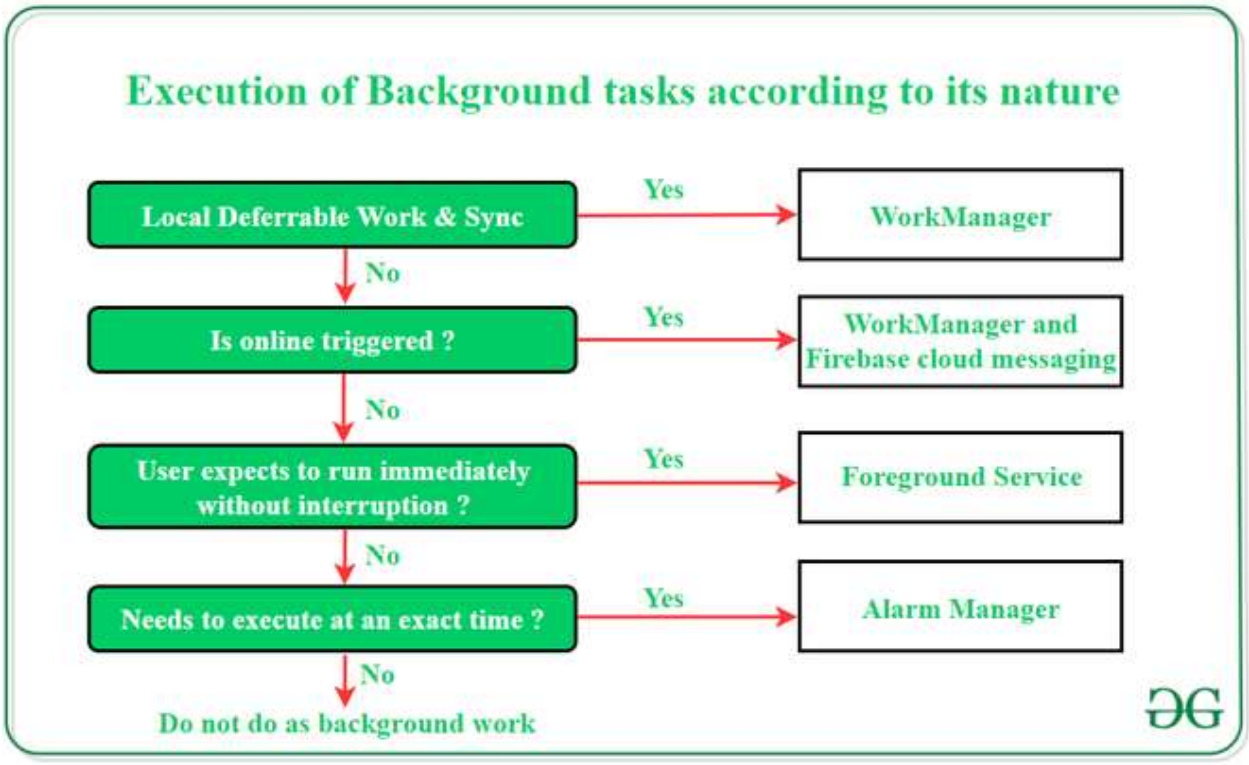
Advantages of Room Component:

- Reduce boilerplate code
- Simplifies database access mechanism
- Easy to implement migrations
- Test ability is high

WorkManager

WorkManager API provides an optimal solution to manage the background tasks in android which are deferrable (can be run later and is still useful) as well as guaranteed (runs even if the device restarts) in nature. Its capability to cover the power saving feature of android and the ability to run with or without Google Play Services are the reasons for its popularity among developers. Further, it is also backward compatible with API level 14.

The ability of android devices to download a file/document in chunks i.e., occasionally (like user resumes download as per the availability of the Wi-Fi Network) and the feature of saving the downloaded state even if the device restarts is possible only because of WorkManager API. Its execution depends upon the order of request made by the user and for each work request, it returns the state which is displayed on the UI of the device.

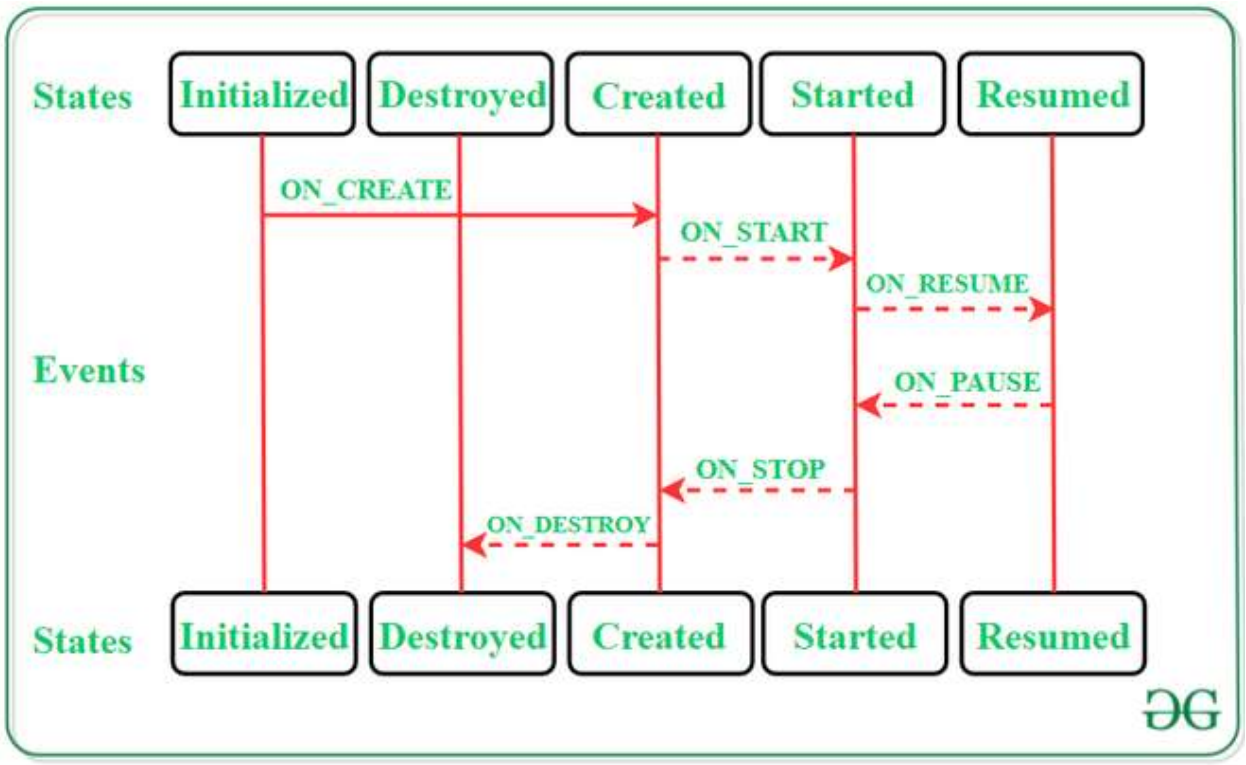


Advantages of WorkManager Component:

- Provides backward compatibility
- Scheduling and chaining of tasks is possible
- Users can keep track of/status of the tasks

Lifecycle-Aware Components

Details of the lifecycle state of an android component are stored in the **Lifecycle** class and it permits other components/objects to observe this state. An android component is lifecycle aware if it has the ability to detect the change in the lifecycle state of other components and respond accordingly. Proper management of application lifecycles is a challenging task for the developers as it can lead to severe issue like memory leaks and app crashing. The **android.arch.lifecycle** package facilitates the developers by managing the lifecycle attached to the components directly by the system. By implementing the **LifecycleObsever** interface within the desired class, it can be configured to the lifecycle aware.



There are 2 ways to create Activity/Fragment which are Lifecycle aware:

Extend a LifecycleActivity or LifecycleFragment:

```
1.  class MainActivity : LifecycleActivity() {
2.    override fun onCreate(savedInstanceState: Bundle?) {
3.       super.onCreate(savedInstanceState)
4.       setContentView(R.layout.activity_main)
5.    }
6.  }
```

Connect LifecycleObserver and Lifecycle events through annotation:

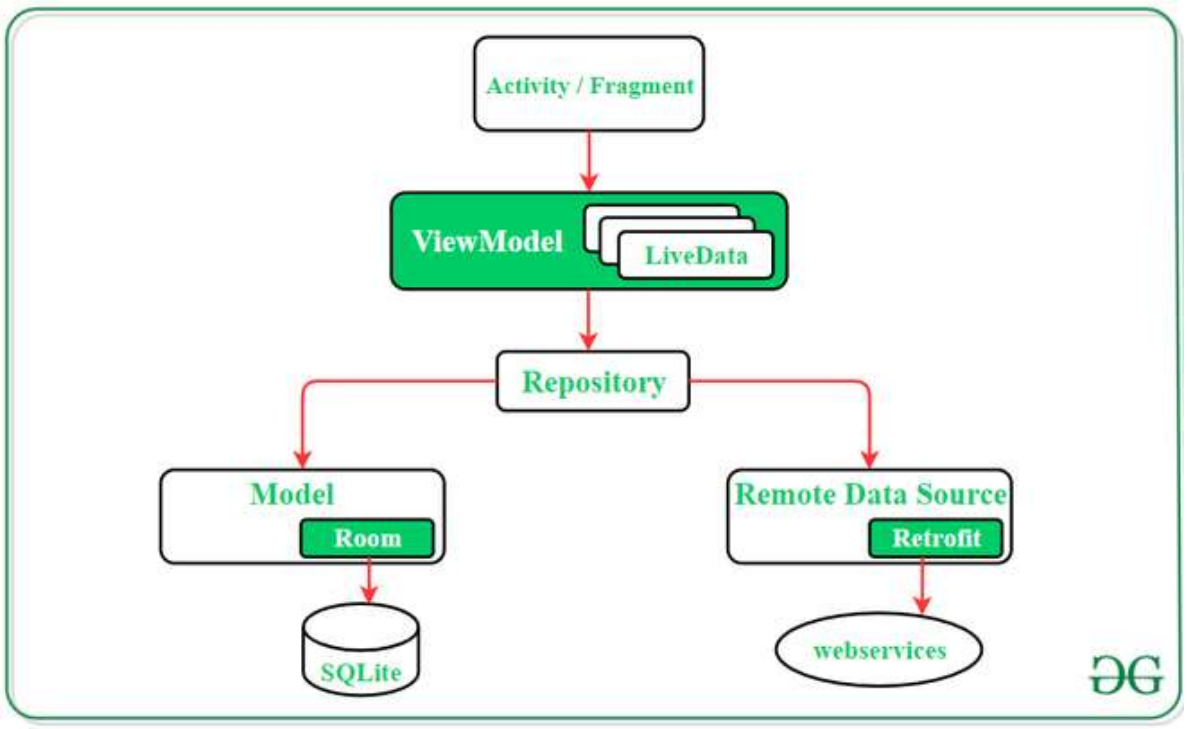
```
1.  class MainActivityObserver : LifecycleObserver, AnkoLogger {
2.    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
3.    fun onResume() {
4.       info("onResume")
5.    }
6.    @OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)
7.    fun onPause() {
8.       info("onPause")
9.    }
10. }
```

Advantages of Lifecycle aware Components:

- Helps in creating organized application components
- Ease in testing and maintenance of components
- Less code requirement to executes tasks

ViewModel

ViewModel is one of the most critical class of the Android Jetpack Architecture Component that supports data for UI components. Its purpose is to hold and manage the UI related data. Moreover, its main function is to maintain integrity and allows data to service during configuration changes like screen rotations. Any kind of configuration change in android devices tends to recreate the whole activity of the application. It means the data will be lost if it has been not saved and restored properly from the activity which was destroyed. To avoid these issues, it is recommended to store all UI data in the ViewModel instead of an activity.



An activity must extend the ViewModel class to create a view model:

```
1.  class MainActivityViewModel : ViewModel() {
2.  ……
3.  ……..
4.  }
```

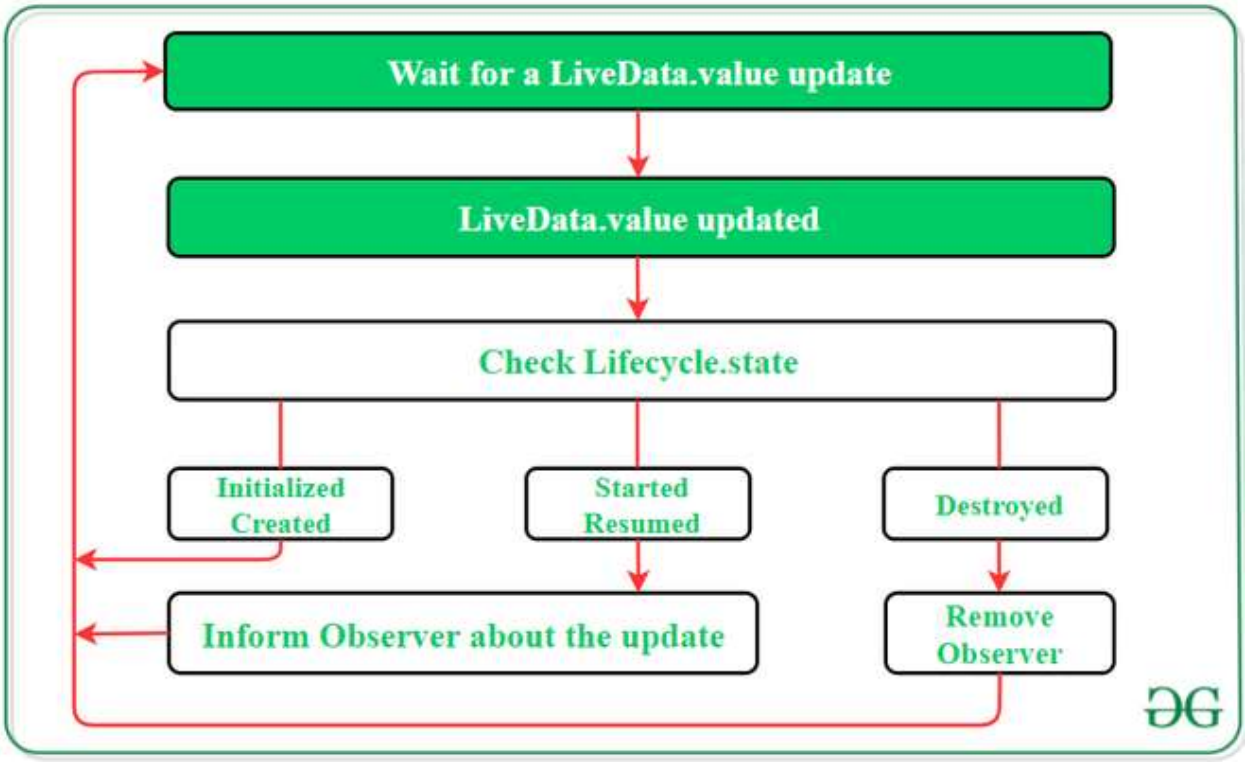
Advantages of ViewModel component:

- Helps in data management during configuration changes
- Reduce UI bugs and crashes
- Best practice for software design

LiveData

This component is an observable data holder class i.e, the contained value can be observed. LiveData is a lifecycle aware component and thus it performs its functions according to the lifecycle state of the other application components.

Further, if the observer's lifecycle state is active i.e, either **STARTED** or **RESUMED**, only then LiveData updates the app component. LiveData always checks the observer's state before making any update to ensure that the observer must be active to receive it. If the observer's lifecycle state is destroyed, LiveData is capable to remove it, and thus it avoids memory leaks. It makes the task of data synchronization easier.



It is necessary to implement **onActive** and **onInactive** methods by LiveData:

```
1.  class LocationLiveData(context: Context)
2.    : LiveData<Location>(), AnkoLogger, LocationListener {
3.    private val locationManager: LocationManager =
4.          context.getSystemService(Context.LOCATION_SERVICE) as LocationManager
5.    override fun onActive() {
6.      info("onActive")
7.      locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0f, this)
8.    }
9.    override fun onInactive() {
10.     info("onInactive")
11.     locationManager.removeUpdates(this)
12.   }
13.   // ....
14. }
```

In order to observe a LiveData Component `observer (Lifecycleowner, Observer <T>` method is called:

```
1.  fun observeLocation() {
2.      val location = LocationLiveData(this)
3.      location.observe(this,
4.            Observer { location ->
5.              info("location: $location")
6.            }}}}
```

Advantages of LiveData component:

- UI is updated as per the appropriate change in the data
- It removes the stopped or destroyed activities which reduce the chance of app crash
- No memory leaks as LiveData are a lifecycle aware component

Navigation Component

The navigation component of Android architecture is a framework for designing the in-app UI. Developers can follow the single-activity app architecture to structure the application UI. Navigation manages the complexity related to the **fragment transactions** in the application. It also facilitates developers to display transactions and **Back Behavior**.

By using Navigation Component, one can get all the benefits of other Architecture components as well like **Lifecycle** and **ViewModel**. It helps in implementing basic navigation styles like simple button clicks to complex navigation patterns like app bar and navigation drawer. Further, it also supports deep links and helpers which enables the connections of this component with the navigation drawer and bottom navigation.

Advantages of Navigation Component:

- Ease the transaction through animated visualization
- Supports deep linking
- Handle fragment transactions
- Support common as well as complex navigation pattern

Paging

While developing an android application, it is very much important to organize the data loading process. Mostly, apps display one activity/fragment at a time and thus require to load and display only a small section of data. However, at the same time, the application is processing a large dataset that is currently of no use. Thus, it is a critical area of application which must be handled, otherwise can lead to waste of the device's battery and the network bandwidth.

The **Paging library** of the android architecture component provides the optimal solution of these issues. This library provides the facility to load the application data slowly and in a cautious manner. Moreover, paging is a very cooperative library for those applications in which the displayed data is updating continuously as it provides a list of **unbounded sites** as well as **large but bounded lists**.

Following are the three common ways of loading data through the paging library:

1. Loading data is stored in the device database
2. Loading data over a network by serving as a cache for the database
3. Load data by serving from the back-end server

Advantages of Paging component:

- Easy to integrate with RecyclerView in order to display a large data set.
- Compatible with LiveData and RxJava for updating the UI data
- Loads data gradually with caution

Data Binding

Data Binding library is a support library that provides the feature of binding UI components in an activity/fragment to the data sources of the application. The library carries out this binding task in a declarative format and not in a programmatical way.

Example: To find a TextView widget and bind it to the `userName` property of the ViewModel variable, the `findViewById ()` method is called:

```
TextView textView = findViewById(R.id.sample_text);

textView.setText(viewModel.getUserName());
```

After using the Data Binding library, the above code changes by using the assignment expression as follows:

```
<TextView

 android:text="@{viewmodel.userName}" />
```

Advantages of Data Binding Component:

- Make code simpler and easy to maintain by removing UI frameworks called in the activity
- Allows classes and methods to observe changes in data
- Allows to make objects and filled which works as collection observable.

Behaviour Components

All the behaviour components are as follows:

- DownloadManager
- Media & Playback
- Permissions
- Notifications
- Sharing
- Slices

DownloadManager

The **DownloadManager** is a system service in android that helps in downloading bulky files in the background thread. The ability of mobile devices to download and store files locally is very important and necessary because it is not very appealing to users to keep their internet on all the time. Moreover, continuous internet usage drains the battery of devices faster and leads to increased cost.

DownloadManager class handle **only HTTP** downloads, and it is responsible for avoiding connectivity problems, for resume downloading if the device reboot and mechanism to retry if the file crashes. Since it is a system service, users can simply start a download and listen for a broadcast event to handle the finished download.

1. **Initializing the DownloadManager**
   Like any other system service, the DownloadManager is initialize using the `getSystemService ()` method. The resultant object is then cast into the DownloadManager class.

```
1.  private void initializeDownloadManager() {
2.      downloadManager= (DownloadManager) getSystemService(DOWNLOAD_SERVICE);
3.   }
```

2. **Creating a download request**
   The HTTP request of the user is defined in the Request class which is an inner class of DownloadManager.

```
1.  DownloadManager.Request request=new
2.  DownloadManager.Request(Uri.parse("https://media.geeksforgeeks.org/wp-content/cdn-
    uploads/gfg_200x200-min.png"));
3.      request.setTitle("GfG_logo")
4.           .setDescription("File is downloading...")
5.           .setDestinationInExternalFilesDir(this,
6.                Environment.DIRECTORY_DOWNLOADS,fileName)
7.           .setNotificationVisibility(DownloadManager.Request.VISIBILITY_VISIBLE_NOTIFY_CO
    MPLETED);
```

3. **Enqueue a Download**
   To enqueue a download (adding a download request to the queue of the download manager), the `enqueue ()` method is used. This queue is processed automatically by the system and it returns a **download ID**.

```
downLoadId=downloadManager.enqueue(request);
```

4. **Remove/Delete a download file**
   In order to remove/delete a download file from the download manager, the `remove ()` method is called and the download ID of the file is passed into it as an argument.

```
private void deleteDownloadedFile(){

    downloadManager.remove(downLoadId);

 }
```

Media & Playback

Jetpack provides a **background-compatible API** for the android multimedia framework. The included Media libraries facilitate developers to integrate audio, video and image files into an application. Further, the **Playback libraries** allow android apps to playback video and audio using sessions and media controllers.

The MediaPlayer APIs can access and play media files from the application's resources (raw resources), from standalone files in the files system, or from a data stream coming through a network connection.

Below mentioned classes are used to play sound and video in the android framework:

1. **MediaPlayer**: Primary API responsible for playing audio and video

2. **AudioManager**: Manage audio sources and audio output (volume control) on the device. The

The android framework provides a variety of options to the developers on the use of the type of media players in an application:

- **Media Player**: This is a clean player having basic functionalities and it supports the most commonly used audio-video formats as well as data sources. The simple and easy to use interface of this player makes it suitable in many use cases; however, it supports minimal customization.

- **ExoPlayer**: An open-source software that supports high-performance features and adaptive streaming technology like **DASH** and **HLS**.

- **YouTube**: For those use cases in which an app play videos precisely from the YouTube platform, developers consider integrating this API.

- **Custom Media Player**: In order to design a fully customized media player that fulfils th exact need for an application, one can use low-level media APIs such as MediaCodec, MediaDRM and AudoTrack.

To design an application using MediaPlayer, some appropriate declaration has to be made in the manifest file to use the associated features.

1. **To stream any network-based content using MediaPlayer, the app must request network access:**

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

2. To prevent the screen from dimming or the processor from sleeping while an application is running a MediaPlayer, the app must request Wake Lock permission:

```
<uses-permission android:name=”android.permission.WAKE_LOCK” />
```

Permission

This area of the Behavior component accommodates a permission system along with a set of predefined permissions for certain tasks in the android apps. Almost every application requests some permissions from the user. For example, if an application requires network access to carry out a task, it will define permission for the user.

Developers declare the permissions for an application in its manifest file and code must handle both the situations i.e, acceptance as well as denial of the requested permission by the user. Moreover, this file can also define additional permissions that will be used to restrict access to particular components.

The concept of asking permission in the android system has changed significantly after API 23. Initially, the applications ask for all the permissions from the user at the time of installation. After the release of **API level 23**, the applications seek permissions during runtime. With this new model of permission to an app for a certain task rather than selecting “**allow always**”. There are different levels present in the system permissions, one of them is **protection levels**.

Following are the two important protection level for any permission:

1. **Normal**: The set of permissions that falls in this category are safe in terms of user’s privacy as well as carrying out tasks that require the involvement of other applications. This type of permission is granted to the application by default. To set a time zone for a device/application is an example of normal permissions.

2. **Dangerous**: This class of permissions targets user’s private information, for example, permission to read user’s contact data. These permissions have optional potential to affect the operations of other applications as well. Generally, the app asks dangerous permissions during runtime.

Permission groups:

Dangerous permissions that seek the user's choice to grant or deny it are classified into 9 groups. The purpose of doing such type of grouping is to facilitate users to grant all the permissions required by a component with a single action instead of selecting one by one.

For example, it is convenient for users to grant all permissions related to the editing of contacts at once rather than granting access separately to view, edit and add contacts.

Following is the table of permission groups:

Permission	Group Description
Calendar	Managing calendars
Contacts	Managing contacts
Location	Current device location
Camera	Taking photos and recording videos
Phone	Dialling and managing phone calls
SMS	Sending and viewing messages
Microphone	Audio recording
Storage	Accessing photos, media and files
Body Sensors	Pulse rate, heart rate and similar kind of data

Notifications

Generating a notification is one of the most beneficial features of today's applications. Its prime purpose is to inform the users regarding event happening inside an app. By using it in the correct way, this utility has the capability to elevate user's motivation on using the application. Android provided the Notification service from the very beginning and it has evolved continuously over the period of time.

Developers can accommodate various kinds of images and buttons in the notification area which makes them more expressive. Not only mobile phones but Android TV and Wearables also us ethe notification feature in order to control their media operations.

Following are the types of notifications that are generally used:

1. Communication from other users
2. Well-timed and informative task reminders

Ways to inform the users:

The device can attract the user's attention and can inform them regarding the arrived notifications. Following are the ways to do the same:

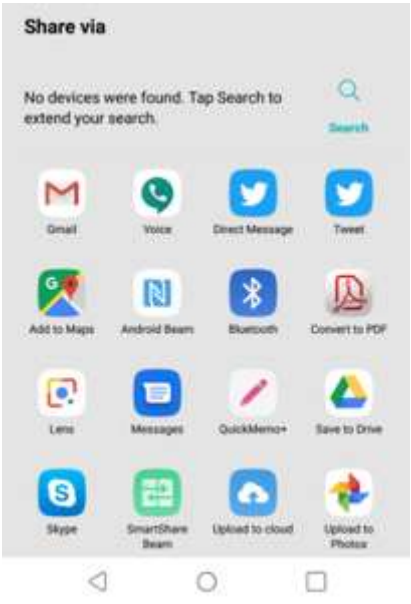
- Play a sound or vibrate
- Show a status bar icon
- Display notification on the lock screen
- Blink the device's LED
- Notification can peek onto the current screen

Sharing

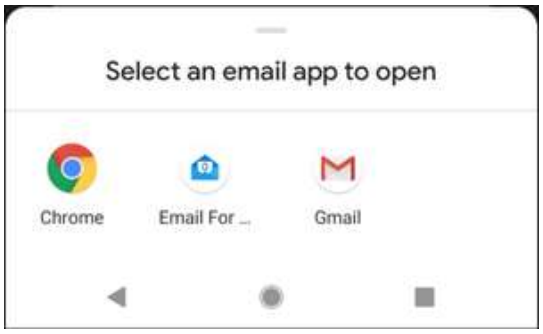
In today's world, humans are very much dependent on mobile applications. Users need to share the day-to-day important information with their colleagues, family or friends. This information can be in the form of text, images or other document files. Thus, it is extremely important for the android apps to have the ability to communicatee and integrate with each other. The Sharing division of Behavior component is responsible for sharing and receiving contents with different applications. The **ShareActionProvider** class is used to carry out the task of the sharing contents and information.

In order to ease the procedure of sharing information across various applications, Android uses Intent and their associated extras. In android, there are two ways for users by which they can share data between apps:

1. Using Android Share Sheet: It is a dialog box that appears on the screen when a user makes a share action request. All the apps available on the device and are compatible with the sharing feature appears on the screen in a sheer-like structure. The main aim of this share sheet is to send information or content outside an app or directly to another user.



2. **Using Android Intent Resolver**: It is mainly used for sharing a file within different applications available on the device. For example, opening an email file on the device and asking the user to choose their preferred main application.



Slices

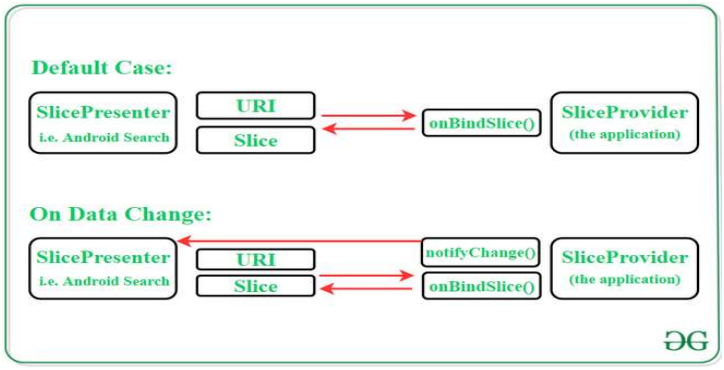
Android provides a new technique to display the remote contents in the form of **Slices**. It is the UI component that displays the contents from an application to the **Google Search app** or in different platforms like **Google Assistant** or **Google Assistant devices**.

Being a pare of Jetpack, it has backward compatibility up to **Android 4.4 (API 19)**. Developers find this module very beneficial because by making app data available using Slices, a user can find information about that app by using Google Search or Assistant. Further, Android Jetpack facilitates in creating flexible UI templates using Slices that capable of displaying app data outside the app.

In the Android project, **SliceProvider** class allows an application to produce content that will be displayed in system spaces. SliceProvider is the extension of ContentProvider and Slices are constructed on the top of the Content Providers.

Developers can host variety of slices from an application as it is based on the **content URIs**. The application will receive a content URI and then select the kind of slice needed to build and present in front of a user.

The `onBindSlice ()` method is called when an application needs to display the slice. This method returns the slice based upon the content URI received as an input. To update the slice `notifyChange ()` method is called. Android search applications like Google Assistant or Google Search are termed as **SlicePresenter**. Slices are fetched by SlicePresenter by calling System API along with the Slice URI.



UI Components

The UI components provide widgets and helpers to make your app not only easy, but delightful use.

All the UI components are as follows:

1. Animation & Transition
2. Auto
3. Emoji
4. Fragment
5. Layout
6. Palette
7. TV
8. Wear

Animation & Transition

Jetpack offers API to set up different kinds of animations available for android apps. This framework impacts the ability to move widgets as well as switching between screens with animation and transition in an application. To improve the user experience, animations are used to animate the changes occurring in an app and the transition framework gives the power to configure the appearance of that change.

Developers can manage the way in which the transition modifies the application appearance while switching from one screen to another. Jetpack Compose is the toolkit used for building native Android UI. It offers a more modular approach for developing the apps by organizing the code in smaller and reusable components. These components are easy to maintain and the code written in it describes the appearance of a UI in a **Declarative Fashion** i.e, based upon the available state. Developers use this declarative nature of Jetpack Compose to showcase complex animations in a beautiful and expressive manner.

A composable named **Transition** is used to create animations in Android. Its flexible nature allows developers to easily pass the information to the user through animating a component's property.

Following are the elements involved in Transition composable that controls the overall animation of a component:

- **TransitionDefinition**: Includes definition of all animations as well as different states of animation required during the transition.
- **initState**: Describe the initial stage of the transition. If it is undefined, it takes the value of the first **toState** available in the transition.
- **toState**: Describe the next state of the transition.
- **clock**: Manage the animation with the change in time. It is an optional parameter.
- **onStateChangeFinished**: An optional listener that notifies the completion of a state change animation.
- **children**: It is composable that will be animated.

```
1.  Method signature of Transition:
2.  @Composable
3.  fun <T> Transition(
4.    definition: TransitionDefinition<T>,
5.    toState: T,
6.    clock: AnimationClockObservable = AnimationClockAmbient.current,
7.    initState: T = toState,
8.    onStateChangeFinished: ((T) -> Unit)? = null,
9.    children: @Composable() (state: TransitionState) -> Unit
10. )
```

Android Jetpack provides some predefined **animation builders** that developers can use directly in their app. The TransitionDefinition includes the code for all these animations.

1. **Tween**: To animate the geometrical changes of an object like moving, rotating, stretching etc.

2. **Physics**: To define spring animations for an object by providing a damping ratio and stiffness.

3. **Keyframe**: To create an animation in which the value of the target object changes over the course of time.

4. **Snap**: To animate the instant switch from one state to another.

5. **Repeatable**: Used to repeat an animation as many times as the developer wants.

Auto

Nowadays people are dependent upon smartphone apps up to such extent that they need them even while driving. The reason for using the mobile phone can be an urgent call or to just enjoy music. Google realized this use case and developed Android Auto with a vision to minimize driver's interaction with the phone as well as to assure safety plus security on the road.

The task is to bring the most practical applications to the user's smartphone or on the compatible car display. Android Auto offers an extensive list of applications to use conveniently on the vehicle's display.

Android Auto is compatible only with phones running on **Android 6.0 (API level 23)** or higher.

The following category of applications can be built, test and distributed on Android Auto:

1. **Navigation apps**: The Google Maps navigation interface enables a user to set destinations, choose various routes, and view live traffic. It assists the driver through voice-guided driving directions at every turn and also estimates the arrival time of the destination. This application continues to run in the background even if the user switches to another screen. Moreover, users can also set other interest in this app like parking and location of gas stations, restaurants etc.

2. **Messaging apps**: Messaging or calling someone is likely the most dangerous thing while driving. To take care of this issue, Android Auto provides messaging apps that receive messages or notifications and read them aloud using the text-to-speech feature. Users can also send replies through voice input in the car. To activate the hands-free voice commands in Android Auto, one can press the **talk** button on the steering wheel or can trigger the device by saying **Ok Google**.

3. **Media apps**: This kind of apps allows users to browse and play any type of audio content in the car. It accepts voice commands to play radio, music or audiobooks.

Jetpack libraries offer two options for developing android apps for cars namely Android Auto and Android Automotive OS. Android Auto apps along with an Android phone are capable of providing a driver-optimized app experience.

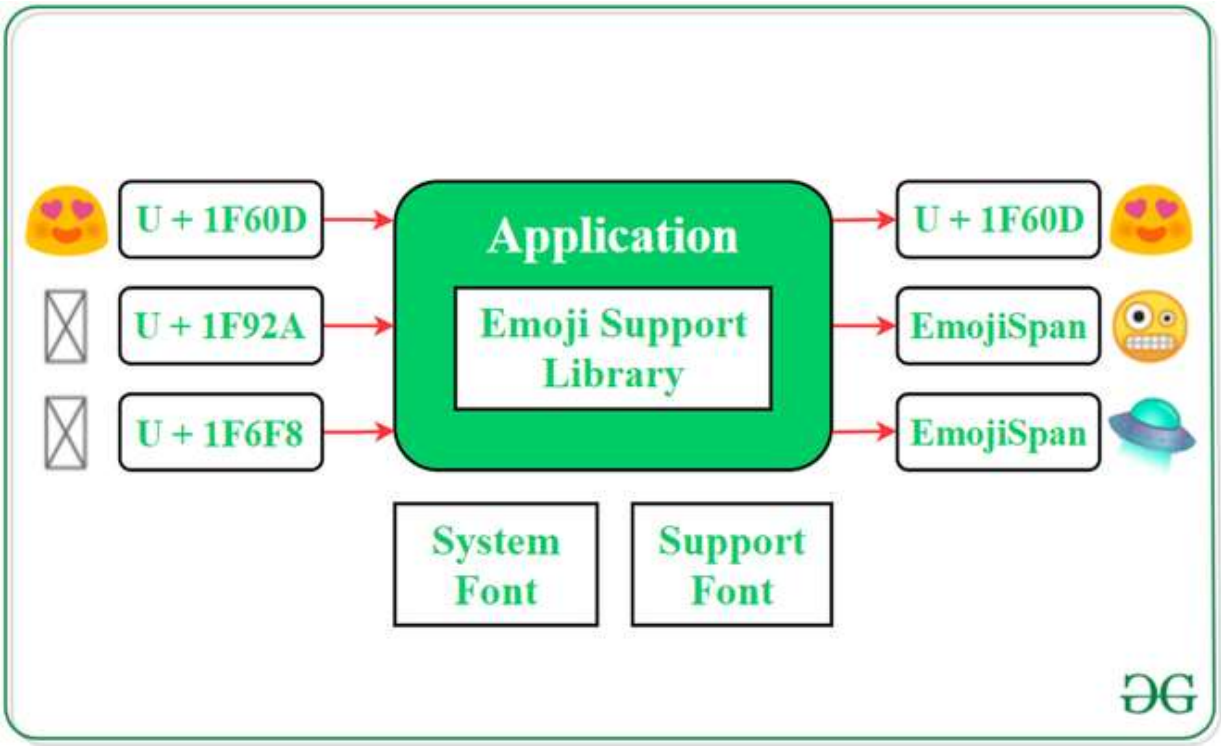
On the other hand, Android Automotive OS is an Android based infotainment system that is embedded into vehicles. With this, the car becomes a self-supporting android device that can run applications on the car's screen. Developers prefer one app architecture while building applications to cover both use cases.

Emoji

If an application is used to communicate between people, emojis are definitely going to be part of that app. The **Unicode standard** is adding new emojis very frequently, thus it becomes important that the user should be able to see the latest Emoji irrespective of the android device version.

Google has released a brand-new library called EmojiCompat in order to handle emoji characters and to use downloadable font support. This library assures that an app is up to date with the latest emojis irrespective of the device OS version.

EmojiCompat identifies an emoji using its `CharSequence` and replaces them with **EmojiSpans** (if required) to assure that the sender and receiver will observe the emoji in the exact same way. Users need to update this library dependency regularly to have the latest emojis. If an application is not using the EmojiCompat library, then the user will see an empty box with the **cross sign** (☒) in place of an emoji. The backward compatibility of this library is up to Android 4.4 (API Level 19). For the Android OS version lower than that, the emoji will be displayed exactly like a regular TextView.



Adding the EmojiCompat support library into the project:

Add the below-mentioned implementation in the app-level build.gradle file:

```
1.    dependencies {
2.        …..
3.        …..
4.        implementation “androidx.emoji:emoji:28.0.0”
5.    }
```

For using Emoji Widgets in AppCompat, add the AppCompat support library to the dependencies section:

```
1.   dependencies {
2.        …..
3.        …..
4.        implementation “androidx.emoji:emoji-appcompat:$version”
5.    }
```

Emoji Views/Widgets:

Widget	Class
EmojiTextView	android.support.text.emoji.widget.EmojiTextView
EmojiEditText	android.support.text.emoji.widget.EmojiEditText
EmojiButton	android.support.text.emoji.widget.EmojiButton
EmojiAppCompatTextView	android.support.text.emoji.widget.EmojiAppCompatTextView
EmojiAppCompatEditText	android.support.text.emoji.widget.EmojiAppCompatEditText
EmojiAppCompatButton	android.support.text.emoji.widget.EmojiAppCompatButton

Fragment

The **Fragment support class** of android has shifted into this segment of Jetpack. Fragments allow separating the UI into discrete pieces that brings **modularity and reusability** into the UI of an activity. A part of the user interface is defined by a fragment which is then embedded into an activity. There is no existence of fragments without an activity. With the release of android Jetpack, Google provided some major improvements and advanced features in using the fragments.

Navigation, **BottomNavigationView** and **ViewPager2** library of the Jetpack are designed to work with fragments in a much more effective way. Moreover, the proper integration of the fragment class with the lifecycle class of the jetpack architecture component is also guaranteed.

Following is the list of newly added features and improvements by Google for android developers:

1. Sharing and communicating among fragments:
   In order to maintain the independent fragments, developers write code in such a way that does not allow fragments to communicate directly with other fragments or with their host activity. The Jetpack fragment library provides two options to establish the communication namely **Fragment Result API** and shared **ViewModel**. The Fragment Rest API is suitable for **one-time results** with data that could be accommodated in a bundle. Further, if there is a requirement to share persistent data along with any custom APIs, ViewModel is preferred. It is also capable of storing and managing UI data. Developers can choose between the two approaches according to the requirement of the app.

2. Constructor can hold the Layout resource ID:

   The AndroidX AppCompat 1.1.0 and Fragment 1.1.0 enable the constructor to take the layout ID as a parameter. With this, a considerable decrease in the number of method overrides is observed in fragments. Now, the inflater can be called manually to inflate the view of a fragment, without overriding the `onCreateView ()` method. This makes the classes more readable.

```
class MyFragmentActivity: FragmentActivity(R.layout.my_fragment_activity)

class MyFragment : Fragment(R.layout.my_fragment)
```

3. FragmentManager and Navigation library:
   All crucial task of a fragment like adding, replacing as well as sending them back to the stack carried out by the **FragmentManager** class. To handle all these navigation related tasks, Jetpack recommends using the **Navigation library**. The framework of this library provides some best practices for developers so that they can work effectively with fragments, fragment manager and the back stack. Every android application that contains fragments in its UI, have to use FragmentManager at some level. However, developers might not interact with the FragmentManager directly while using the Jetpack Navigation library.

4. FragmentFactory:
Android developers had always raised the issue with Fragments that there is no scope of using a constructor with arguments. For instance, developers cannot annotate the fragment constructor with inject and specify the arguments while using Dagger2 for dependency injection. The **AndroidX library** introduced along with Jetpack offers FragmentFactory class that is capable of handling this and similar issues related to fragment creation. The structure of this API is straight forward and generalized which facilitates developers to create the fragment instance in their own customized way. In order to override the default way of instantiating the Fragment, one has to register the FragmentFactory in the FragmentManager of the application.

```
1.  class MyFragmentFactory : FragmentFactory() {
2.   override fun instantiate(classLoader: ClassLoader, className: String): Fragment {
3.    // loadFragmentClass() method is called to acquire the Class object
4.    val fragmentClass = loadFragmentClass(classLoader, className)
5.    // use className or fragmentClass as an argument to define the
6.    // prefered manner of instantiating the Fragment object
7.    return super.instantiate(classLoader, className)
8.   }
9.  }
```

5. Testing fragment:
AndroidX Test has been launched by Google to make testing a necessary part of Jetpack. The existing libraries along with some new APIs and full Kotlin support of the AndroidX Test provides a way to write suitable and concise tests. **FragmentScenario** class of the AndroidX library construct the environment for performing tests on Fragments. It consists of two major methods for launching fragments in a test, the first one is `launchInContainer ()` that is used for testing the user interface of a fragment.
Another method is the `launch ()` that is used for testing without the fragment's user interface. In some cases, fragments have some dependencies. To generate test versions of these dependencies, one has to provide a custom FragmentFactory to the `launchInContainer ()` or `launch ()` methods. Developers can choose one of the methods and can use Espresso UI tests to check out the information regarding the UI elements of the fragment.

For using FragmentScenario class, one needs to define the fragment testing artifact in the app-level `build.gradle` file:

```
1.  dependencies {
2.    def fragment_version = "1.2.5"
3.    debugImplementation "androidx.fragment:fragment-testing:$fragment_version"
4.  }
```

6. FragmentContainerView:
AndroidX Fragment 1.2.0 brings **FragmentContainerView** that extends FrameLayout and provides customized layout design for the fragments in android apps. It is used as a parent to the fragment so that it can coordinate with fragment behavior and introduce flexibility in the tasks like **Fragment Transactions**. It also supports the **<fragment>** attributes and addresses the issues of window insets dispatching. Moreover, this container resolves some animation issues related to the z-ordering of fragments like exiting fragments no longer appear on the top of the view.

Layout

User interface structure like activity of an application is defined by Layout. It defines the **View** and **ViewGroup** objects. View and ViewGroup can be created in two ways: **by declaring UI elements in XML** or **by writing code i.e, programmatically**. This portion of Jetpack covers some of the most common layouts like LinearLayout, RelativeLayout and the brand new ConstraintLayout.

Moreover, the official Jetpack Layout documentation provides some guidance to create a list of items using RecyclerView and the card layout using CardView. A View is visible to the user. EditView, TextView and Button are examples of View. On the other hand, a ViewGroup is a container object that defines layout structure for View and thus it invisible. Examples of ViewGroup are LinearLayout, RelativeLayout and ConstraintLayout.

Palette

Providing the right color combination plays a major role in uplifting the user experience. Thus, it is an important aspect during the app development process. Developers often build applications in which **UI elements** change their color according to the time (day and night). This type of practice gives the user a good kind of feeling and assure an immersive experience during app usage. To carry out these tasks, Android Jetpack provides a new Palette support library. It is capable of extracting a small set of colors from an image. The extracted color style the Ui controls of the app and update the icons based on the background image's color.

The **Material Design** of android app is the reason behind the popularity of dynamic use of color. The extracted color or palette contains vibrant and muted tones of the image. It also includes foreground text colors to ensure maximum readability.

To include Palette API in a project, update the app-level `build.gradle` file in the following manner:

```
1.  dependencies {
2.    …..
3.    …..
4.    implementation 'androidx.palette:palette:1.0.0'
5.  }
```

The Palette gives the choice to developers to select the number of colors they want to be generated from a certain image source. The default value of `numberOfColors` in the resulting palette is set as 16. However, the count can go up to 24-32. The time taken in generating the complete palette is directly proportional to the color count. Once the palette is generated, a **swatch** (a kind of method) is used to access those colors. Each color profile has an associated swatch that returns the color in that palette.

Following are the color profiles generated by the palette API:

Profile	Swatch
Light Vibrant	Palette.getLightVibrantSwatch ()
Vibrant	Palette.getVibrantSwatch ()
Dark Vibrant	Palette.getDarkVibrantSwatch ()
Light Muted	Palette.getLightMutedSwatch ()
Muted	Palette.getMutedSwatch ()
Dark Muted	Palette.getDarkMutedSwatch ()

TV

Jetpack offers several key components to assist developers in building apps for **Android Smart TVs**. The structure of the Android TV application is the same as the mobile/tablet apps, however, there are some obvious differences. The hardware and controllers of a TV are very much different than mobile devices. Further, the navigation system is to be handled through a **d-pad** on a TV remove (up, down, left and right arrow buttons). To address these concerns, Jetpack offers **Leanback library**. This library also resolves the issue of searching as well as recommending content to the user on Android TV.

The dependency of the Leanback library can be added in the app `build.gradle` file:

```
1.  dependencies {
2.    def leanback_version = "1.0.0"
3.    implementation "androidx.leanback:leanback:$leanback_version"
4.  }
```

Wear OS

Android version for wearable devices is called **Wear OS**. Android Jetpack includes the **Wear UI library** that enables developers to create apps that can play and control media from a watch. A standalone watch app or watch face also be created using the UI component of the library. Jetpack assures that the user interface remains optimized and compatible across all apps. Wear OS also supports mobile device features like notifications and 'actions on Google'. Developers can develop three kinds of systems under the wear OS:

1. **Wear apps**: Applications that will run on smartwatches or gears. It supports device features like sensors and the GPU.
2. **Watch faces**: It is specially made for custom drawings, colors, animations and contextual information.
3. **Complication data providers**: Provides custom data such as text, images etc. to watch faces.

Android Architecture

What is Architecture?

If you are building an application in an organized manner with some set of rules, describe proper functionalities and implement it with proper protocols, the it is called an Architecture.

Role of Architecture:

Let us say if we are not using any architecture and we are writing our code in a class/activity/fragment in an unorganized manner then the problems we will face are:

- The number of lines of code will increase that it will become complex to understand.
- It decreases readability and increases the number of bugs. Thus, it is difficult to test and reduces the quality of the product.

So, to provide clear data flow which increases robustness, scalability, bug resistant, increase readability, easy to modify and increase productivity and provide a quality app. Thus, we should use proper architecture, suitable to work in a team.

Some principles for good Architecture in Android:

To get good architecture there are some basic concepts we should follow. They are:

- **Separation of concern**: Component should do what it is required. Shown in the diagram.

  This we can achieve by Architecture pattern.

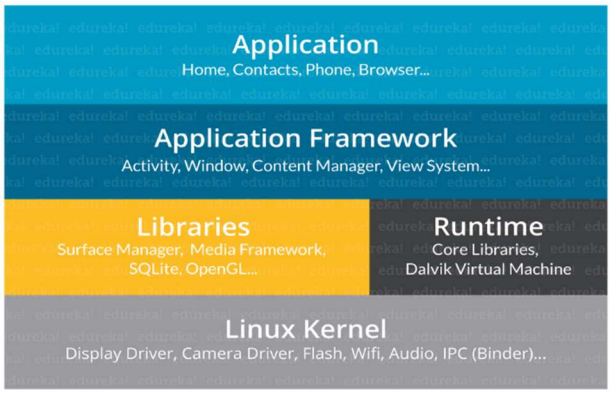
- **No Hard Dependency**: It should be fixed if every component should work on some limited amount of dependency. All dependencies should be provided from outside. Use Dependency Injections.
- **Manage Life Cycle & Data Persistence**: It can be achieved by Architecture Component.

Explain Android Architecture?

Android architecture refers to the different layers in the Android stack. It includes your operating system, middleware and important applications. Each layer in the architecture provides different services to the layer just above it.

The four layers in the Android stack are:

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



Linux Kernel Layer

Linux Kernel is the bottom most layer in the architecture of Android. It never really interacts with the users and developers, but is at the heart of the whole system.

It's important because it provides the functions in the Android system such as hardware abstraction, memory management programs, security settings, power management software, other hardware drivers, network stack etc.

Libraries

The next layer in the Android architecture includes libraries. Libraries carry a set of instructions to guide the device to handle different types of data.

For example – the playback ans recording of various audio and video formats is guided by the Media Framework Library.

Runtime

The third section of the architecture is runtime which provides a key component called Dalvik Virtual Machine (DVM). But Android Runtime has replaced DVM since Android Lollipop.

ART uses Ahead of Time Approach (AOT) instead of JIT. Using AOT, the dex files are compiled before they are needed. Usually, they are done at installation time only and then stored in phone storage.

Application Framework Layer

Our applications directly interact with these blocks of the Android architecture. These programs manage the basic functions of phone like resource management, voice call management etc. It provides many higher-level services to applications in the form of Java classes.

Some important blocks of Application Framework are activity management, content providers, resource manager, notification manager, view system.

Android Framework is an essential part of the Android Architecture. It is a set of APIs that allows developers to write apps.

Application Layer

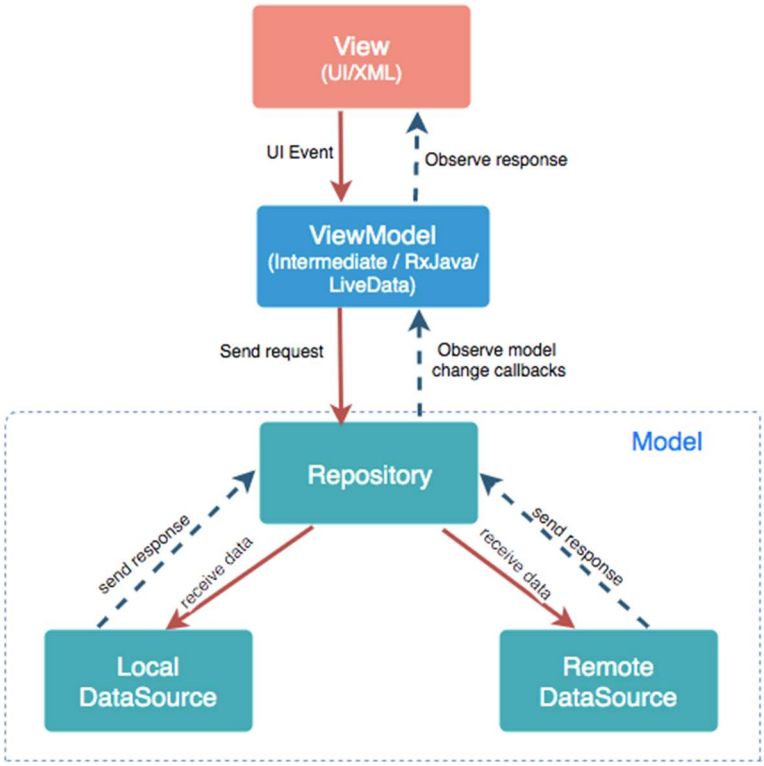
The applications are at the topmost layer of the Android stack. All the applications will be installed in this layer such as address book, games etc.

MVVM (Model – View – ViewModel)

What is MVVM Architecture?

The MVVM architecture is all about guiding you in how to organize and structure your code to write maintainable, testable and extensible applications.

MVVM architecture is a Model-View-ViewModel architecture that removes the coupling between each component. Most importantly, in this architecture, the children don't have reference to the parent, they only have the reference by observables.



Model:

It represents the data and the business logic of the Android Application. It consists of the business logic, local and remote data source, model classes, repository.

View:

It consists of the UI code (Activity, Fragment), XML. It sends the user action to the ViewModel but does not get the response back directly. To get the response, it has to subscribe to the observables which ViewModel exposes to it.

ViewModel:

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

Advantages

- Your code is even more easily testable than with plain MVVM.
- Your code is further decoupled.
- The package structure is even easier to navigate.
- The project is even easier to maintain.
- Your team can add new features even more quickly.

Disadvantages

- It has a slightly steep learning curve. How all the layers work together may take some time to understand, especially if you are coming from patterns like simple MVVM or MVP.
- It adds a lot of extra classes, so it's not ideal for low-complexity projects.

Layers of MVVM with Clean Architecture

The code is divided into three separate layers:

- Presentation Layer
- Domain Layer
- Data Layer

Presentation Layer

This includes our `Activities`, `Fragments`, and `ViewModels`. An `Activity` should be dumb as possible. Never put your business logic in `Activities`.

An `Activity` will talk to a `ViewModel` and a `ViewModel` will talk to the domain layer to perform actions. A `ViewModel` never talks to the data layer directly.

Domain Layer

The domain layer contains all the `Use Cases` of your application.

We should never block the UI when we fetch data from the database or our remote server. This is the place where we decide to execute our `UseCase` on a background thread and receive the response on the main thread.

The purpose of the `UseCases` is to be a mediator between your `ViewModel` and `Repositories`.

Data Layer

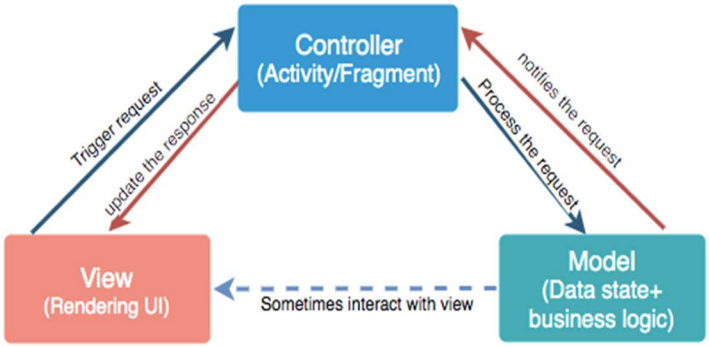
This has all the repositories which the domain layer can use. This layer exposes a data source API to outside classes.

MVC

It is a Model-View-Controller. The most commonly used architecture. These are the three components used in MVC.

- Model:
  It is business Logic and Data State. Getting and manipulating the data, communicates with controller, interacts with the database, sometimes update the views.
- View:
  What we see. User Interface consists of HTML/CSS/XML. It communicates with the controller and sometimes interacts with the model. It passed some dynamic views through the controller.
- Controller:
  It is Activity/Fragment. It communicates with view and model. It takes the user inputs from view/REST services. Process request Get data from the model and passes to the view.

As shown in the figure:



Advantages:

- It keeps business logic separate in the model.
- Support asynchronous techniques.
- The modification does not affect the entire model.
- Faster development process.

Disadvantages:

- Due to large code controller is unmanageable.
- Hinders the Unit testing.
- Increased Complexity.

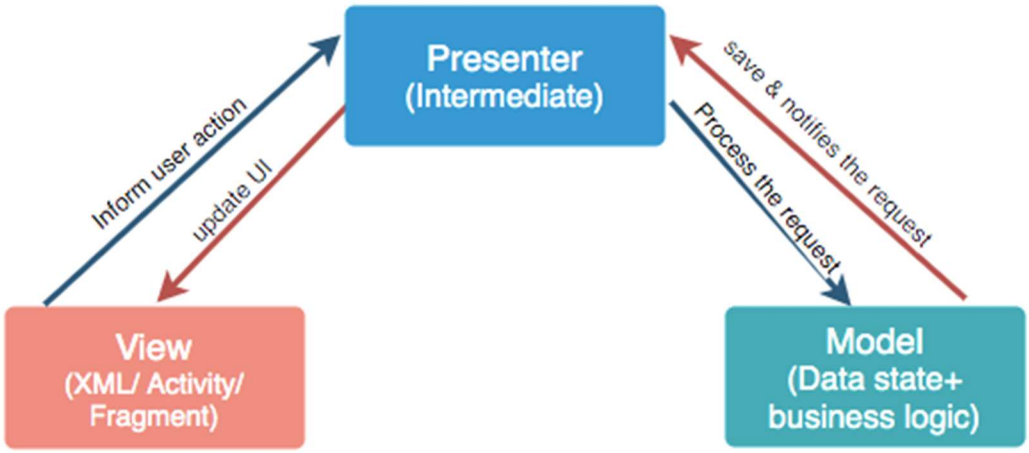
Here are some examples where it is used- RubyOn Rails, Codeigniter, Angular, Django etc.

MVP

It as Model-View-Presenter. For the phase of developing time or for the phase of developers it is vital to divide the architecture into layers. It breaks the dependency on what we have on view.

- Model:
  It is business logic and Data State. Getting and manipulating the data, communicates with the presenter, interacts with the database. It doesn't interact with the view.
- View:
  Consists of UI, activity and fragment. It interacts with the presenter.
- Presenter:
  It presents the data from the model. Control all the behavior that want to display from the app. It drives the view. It tells view what to do. Any interaction between the model and the view is handled by the presenter. Saves the data back to the model.

As shown in the figure:



Advantages:

- It makes view dumb so that you can swap the view easily.
- Reusable of View and Presenter.
- Code is more readable and maintainable.
- Easy testing as business logic separated from UI.

Disadvantages:

- Tight coupling between View and Presenter.
- Huge number of interfaces for interaction between layers.
- The code size is quite excessive.

MVI

MVI stands for **Model-View-Intent**. MVI is one of the newest architecture patterns for android, inspired by the unidirectional and cyclical nature of the **Cycle.js** framework.

MVI works in a very different way compared to its distant relatives, MVC, MVP or MVVM. The role of each MVI components is as follows:

- **Model** represents a state. Models in MVI should be immutable to ensure a unidirectional data flow between them and the other layers in your architecture.
- Like in MVP, Interfaces in MVI represents **Views**, which are then implemented in one or more Activities or Fragments.
- Intent represents an intention or a desire to perform an action, either by the user or the app itself. For every action, a View receives an Intent. The Presenter observes the Intent, and Models translate it into a new state.

Models

Other architecture patterns implement Models as a layer to hold data and act as a bridge to the backend of an app such as databases or APIs. However, in MVI, Models both hold data and represent the state of the app.

What is the state of the app?

In reactive programming, an app reacts to a change, such as the value of a variable or a button click in your UI. When an app reacts to a change, it transitions to a new **state**. The new state may appear as a UI change with a progress bar, a new list of movies or a different screen.

To illustrate how models work in MVI, imagine you want to retrieve a list of the most popular movies from a web service such as the TMDB API. In an app built with the usual MVP pattern, Models are a class representing data like this:

```
1.  data class Movie(
2.    var voteCount: Int? = null,
3.    var id: Int? = null,
4.    var video: Boolean? = null,
5.    var voteAverage: Float? = null,
6.    var title: String? = null,
7.    var popularity: Float? = null,
8.    var posterPath: String? = null,
9.    var originalLanguage: String? = null,
10.   var originalTitle: String? = null,
11.   var genreIds: List<Int>? = null,
12.   var backdropPath: String? = null,
13.   var adult: Boolean? = null,
14.   var overview: String? = null,
15.   var releaseDate: String? = null
16. )
```

In this case, the Presenter is in charge of using the Model above to display a list of movies with code like this:

```
1.  class MainPresenter(private var view: MainView?) {
2.    override fun onViewCreated() {
3.      view.showLoading()
4.      loadMovieList { movieList ->
5.        movieList.let {
6.          this.onQuerySuccess(movieList)
7.        }
8.      }
9.    }
10.
11.   override fun onQuerySuccess(data: List<Movie>) {
12.     view.hideLoading()
13.     view.displayMovieList(data)
14.   }
15. }
```

While this approach is not bad, there are still a couple of issues that MVI attempts to solve:

- **Multiple inputs**: In MVP and MVVM, the Presenter and the ViewModel often end up with a large number of inputs and outputs to manage. This is problematic in big apps with many background tasks.

- **Multiple states**: In MVP and MVVM, the business logic and the Views may have different states at any point. Developers often synchronize the state with Observable and Observer callbacks, but this may lead to conflicting behavior.

To solve this issue, make your Models represent a **state** rather than data.

Using the previous example, this is how you could create a Model that represents a state:

```
1.  sealed class MovieState {
2.    object LoadingState : MovieState()
3.    data class DataState(val data: List<Movie>) : MovieState()
4.    data class ErrorState(val data: String) : MovieState()
5.    data class ConfirmationState(val movie: Movie) : MovieState()
6.    object FinishState : MovieState()
7.  }
```

When you create Models this way, you no longer have to manage the state in multiple places such as in the views, presenters or ViewModel. The Model will indicate when your app should display a progress bar, an error message or a list of items.

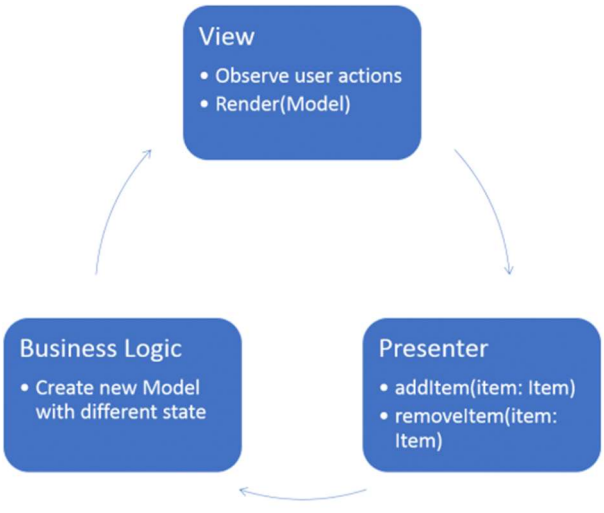
Then, the Presenter for the above example would look like this:

```
1.  class MainPresenter {
2.
3.    private val compositeDisposable = CompositeDisposable()
4.    private lateinit var view: MainView
5.
6.    fun bind(view: MainView) {
7.      this.view = view
8.      compositeDisposable.add(observeMovieDeleteIntent())
9.      compositeDisposable.add(observeMovieDisplay())
10.   }
11.
12.   fun unbind() {
13.     if (!compositeDisposable.isDisposed) {
14.       compositeDisposable.dispose()
15.     }
16.   }
17.
18.   private fun observeMovieDisplay() = loadMovieList()
19.       .observeOn(AndroidSchedulers.mainThread())
20.       .doOnSubscribe { view.render(MovieState.LoadingState) }
21.       .doOnNext { view.render(it) }
22.       .subscribe()
23. }
```

Your Presenter now has one output: the **state** of the View. This is done with the View's `render ()`, which accepts the current state of the app as an argument.

Another distinctive characteristic of models in MVI is that they should be immutable to maintain your business logic as the single source of truth. This way, you are sure that your Models won't be modified in multiple places. They'll maintain a single state during the whole lifecycle of the app.

The following diagram illustrates the interaction between the different layers:



Thanks to the immutability of your Models and the cyclical flow of your layers, you get other benefits:

- **Single State**: Since immutable data structures are very easy to handle and must be managed in one place, you can be sure there will be a single state between all the layers in your app.
- **Thread Safety**: This is especially useful while working with reactive apps that make use of libraries such as **RxJava** or **LiveData**. Since no methods can modify your Models, they will always need to be recreated and kept in a single place. This protects against side effects such as different objects modifying your Models from different threads.

These examples are hypothetical. You could construct your Models and Presenters differently, but the main premise would be the same.

Views and Intents

Like with MVP, MVI defines an interface for the View, acting as a contract generally implemented by a **Fragment** or an **Activity**. Views in MVI tend to have a single `render()` that accepts a state to render to the screen. Views in MVI use Observable `intent()`s to respond to user actions. MVP on the other hand generally uses verbose method names to define different inputs and outputs.

Note: Intents in MVI don't represent the usual `android.content.Intent` class used for things like starting a new class. Intents in MVI represent a future action that changes the app's state.

This is how a View in MVI might look:

```
1.  class MainActivity : MainView {
2.
3.    override fun onCreate(savedInstanceState: Bundle?) {
4.      super.onCreate(savedInstanceState)
5.      setContentView(R.layout.activity_main)
6.    }
7.
8.    //1
9.    override fun displayMoviesIntent() = button.clicks()
10.
11.   //2
12.   override fun render(state: MovieState) {
13.     when(state) {
14.       is MovieState.DataState -> renderDataState(state)
15.       is MovieState.LoadingState -> renderLoadingState()
16.       is MovieState.ErrorState -> renderErrorState(state)
17.     }
18.   }
19.
20.   //4
21.   private fun renderDataState(dataState: MovieState.DataState) {
22.       //Render movie list
23.   }
24.
25.   //3
26.   private fun renderLoadingState() {
27.       //Render progress bar on screen
```

```
28.   }
29.
30.   //5
31.   private fun renderErrorState(errorState: MovieState.ErrorState) {
32.       //Display error message
33.   }
34. }
35.
```

Taking each section in turn:

1. **displayMoviesIntent**: Binds UI actions to the appropriate intents. In this case, it binds a button click Observable as an intent. This would be defined as part of your `MainView`.

   Note: We're using RxBinding to convert button click listeners into RxJava Observables.

2. **render**: Map your ViewState to the correct methods in your View. This would also be defined as part of your `MainView`.
3. **renderDataState**: Render the Model data to the View. This data can be anything such as weather data, a list of movies or an error. This is generally defined as an internal method for updating the display based on the state.
4. **renderLoadingState**: Render a loading screen in your View.
5. **renderErrorState**: Render an error message in your View.

The example above demonstrates how one `render()` receives the state of your app from your Presenter and an Intent triggered by a button click. The result is a UI change such as an error message or a loading screen.

State Reducers

With mutable Models, its easy to change the state of your app. To add, remove or update some underlying data, call a method in your Models such as this:

```
myModel.insert(items)
```

You know that Models are immutable, so you have to recreate them each time the state of your app changes. If you want to display new data, create a new Model. What do you do when you need information from a previous state?

The **answer**: State reducers.

The concept of **State Reducers** derives from **Reducer Functions** in reactive programming. Reducer functions provide steps to merge things into the accumulator component.

Reducer functions are a handy tool for developers, and most standard libraries have similar methods already implemented for their immutable data structures. Kotlin's Lists, for example, include `reduce()`, which accumulates a value starting with the first element of the list, applying the operation passed as an argument:

```
1.  val myList = listOf(1, 2, 3, 4, 5)
2.  var result = myList.reduce { accumulator, currentValue ->
3.    println("accumulator = $accumulator, currentValue = $currentValue")
4.    accumulator + currentValue }
5.  println(result)
```

Running the above code would produce the following output:

```
1.  accumulator = 1, currentValue = 2
2.  accumulator = 3, currentValue = 3
3.  accumulator = 6, currentValue = 4
4.  accumulator = 10, currentValue = 5
5.  15
```

Note: You can run the above snippet of code in an online Kotlin REPL.

The above code iterates over each element of `myList` using `reduce` and adds each element to the current accumulator value.

Reducer functions consist of two main components:

- **Accumulator**: The total value accumulated so far in each iteration of your reducer function. It should be the first argument.
- **Current Value**: The current value passing through each iteration of your reducer function. It should be the second argument.

MVI: Advantages and Disadvantages

Model-View-Intent is a tool to create maintainable and scalable apps.

The main advantages of MVI are:

- A unidirectional and cyclical data flow.
- A consistent state during the lifecycle of Views.
- Immutable Models that provide reliable behavior and thread safety on big apps.

One downside of using MVI rather than other architecture patterns for Android is that the learning curve for this pattern tends to be a bit longer. You need a decent amount of knowledge of other intermediate and advanced topics such as reactive programming, multi-threading and RxJava. Architecture patterns such as MVC or MVP might be easier to grasp for new Android developers.

Android Firebase

Firebase Realtime Database is a cloud-hosted database that supports multiple platforms Android, iOS and Web. All the data is stored in JSON format and any changes in the data, reflects immediately by performing sync across all the platforms & devices. This allows us to build more flexible real time apps easily with minimal effort.

In simpler words, we can say that a Firebase Realtime Database is a type of database that is cloud-hosted i.e., it runs on the cloud and it can be accessed from any platform i.e., it can be accessed from Android, iOS and Web at the same time. So, make the database once and use it on different platforms. The data stored in the database if of the form of NoSQL database and is stored in JSON format.

The best part is that whenever there is a change in the database, it will be immediately reflected in all the devices connected to it.

Advantages of using Firebase Realtime Database

Realtime:

The data stored in the Firebase Realtime Database will be reflected at real time i.e., if there is a change in the values in the database then that change will be reflected back to all the users at that instant only and no delay will be there.

Large Accessibility:

The Firebase Realtime Database can be accessed from various platforms like Android, iOS, Web. So, you need not write the same code for different platforms a number of times.

Offline Mode:

This is the best advantage of using Firebase Realtime Database. If you are not connected with the internet and you changed something on your application then that changes will be updated once you are online i.e., your device is connected to the internet. So, even if there is no internet, the user feels like using the services the same as done when there is the internet.

No Application Server:

There is no need for application server here because the data is directly accessed from the mobile device.

Control Access to Data:

By default, no one is allowed to change the data in the Firebase Realtime Database but you can control the access of data i.e., you can set which user can access the data.

FCM

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably send messages at no cost.

Using FCM, you can notify a client app that new email or other data is available to sync. You can send notification messages to drive user re-engagement and retention. For use cases such as instant messaging, a message can transfer a payload of up to KB to a client app.

The Android push notifications through FCM actually treats the Data Messages as notification messages itself. As the interactions in the data messages are handled by the app itself, FCM's work just to deliver a notification and the message content.

Firebase Crashlytics

Firebase Crashlytics is a lightweight, real time crash reporter that helps you track, prioritize and fix stability issues that erode your app quality. Crashlytics saves you troubleshooting time by intelligently grouping crashes and highlighting the circumstances that led up to them.

Crashlytics provides a software development kit (SDK) that developers can integrate into an Android, iOS or Unity app. It was created by Fabric. The company was acquired by Google in 2017, and Crashlytics is now part of Google's Firebase platform.

Firebase Analytics

Firebase Analytics is a tool which allows you to do exactly that – it helps us to learn how our Android and iOS users are engaging with our application. From setup, it'll automatically begin tracking a defined set of events – meaning we can begin learning from the very first app.

Google Analytics for Firebase provides free, unlimited reporting on up to 500 distinct events. The SDK automatically captures certain key events and user properties, and you can define your own custom events to measure the things that uniquely matter to your business.

Firebase Remote Config

Firebase Remote Config is a cloud service that lets you change the behavior and appearance of your app without requiring users to download an app update. When using Remote Config, you create in-app default values that control the behavior and appearance of your app. Then, you can later use the Firebase console or the Remote Config backend APIs to override in-app default values for all app users or for segments of your user base. Your app controls when updates are applied, and it can frequently check for updates and apply them with a negligible impact on performance.

Firebase App Indexing

Firebase App Indexing gets your app into Google Search. If users have your app installed, they can launch your app and go directly to the content they're searching for. App Indexing reengages your app users by helping them find public content right on their device, even offering query autocompletions to help them more quickly find what they need. If users don't yet have your app, relevant queries trigger an install card for your app in Search results.

To set up your Android app for indexing by Google, use the Android App Links Assistant in Android Studio or follow these steps:

- Create deep links to specific content in your app by adding intent filters in your app manifest.
- Verify ownership of your app content through a website association.

Firebase Dynamic Links

Firebase Dynamic Links are the links that work the way you want, on multiple platforms, and whether or not your app is already installed.

With Dynamic Links, your users get the best available experience for the platform they open your link on. If a user opens a Dynamic Link on iOS or Android, they can be taken directly to the linked content in your native app. If a user opens the same Dynamic Link in a desktop browser, they can be taken to the equivalent content on your website.

In addition, Dynamic Links work across app installs: if a user opens a Dynamic Link on iOS or Android and doesn't have your app installed, the user can be prompted to install it; then, after installation, your app starts and can access the link.

Unit Testing

Unit tests are the fundamental tests in your app testing strategy. A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class or component) in a repeatable way.

You should build unit tests when you need to verify the logic of specific code in your app.

Unit testing is done during the development (coding phase) of an application by the developers.

Local unit testing

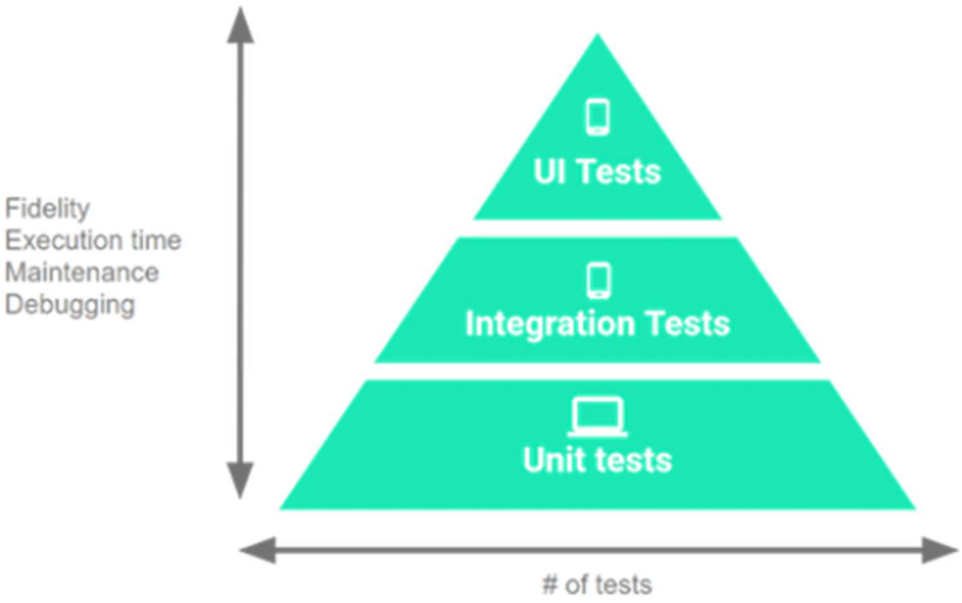
In Android, Local Unit tests refer to those tests that does not require an Android Device (Physical or Emulator) Instrumentation. These tests run on the local JVM (Java Virtual Machine) of your development environment, thus the name Local Unit Test.

Instrumentation Testing

An instrumentation test is a test written by you or your team to specifically to test your app, using the Espresso and UI Automator 2.0 Android test frameworks. When you write an instrumentation test, you create a second APK module that you later upload to Test Lab along with the APK module for your app.

Testing Pyramid

Tests are typically broken into three different kinds:



The Testing Pyramid, showing the three categories of tests that you should include in your app's test suite.

UI Tests:

These tests interact with the UI of your app, they emulate the user behavior and assert UI results. These are the slowest and most expensive tests you can write because they require a device/emulator to run. On Android, the most commonly used tools for UI testing are Espresso and UI Automator.

Integration Tests:

When you need to check how your code interacts with other parts of the Android framework but without the complexity of the UI. These tests don't require a device/emulator to run. On Android, the most common tool for integration testing is Roboelectric.

Unit Tests:

The system under test (SUT) is one class and you focus only on it. All dependencies are considered to be working correctly (and ideally have their own unit tests), so they are mocked or stubbed.

These tests are the fastest and least expensive tests you can write because they don't require a device/emulator to run. On Android, the most commonly used tools for unit testing are Junit and Mockito.

A typical rule of thumb is to have the following split among the categories:

- UI Tests: 10%
- Integration Tests: 20%
- Unit Testing: 70%

Because unit tests are so important in the testing pyramid and also easy to write.

Android Security

Android has built-in security features that significantly reduce the frequency and impact of application security issues. The system is designed so that you can typically build your apps with the default system and file permissions and avoid difficult decisions about security.

The following core security features help you build secure apps:

- The Android Application Sandbox, which isolates your app data and code execution from other apps.
- An application framework with robust implementations of common security functionality such as cryptography, permissions, and secure IPC.
- Technologies like ASLR, NX, ProPolice, safe_iop, OpenBSD dlmalloc, OpenBSD calloc, and Linux mmap_min_addr to mitigate risks associated with common memory management errors.
- An encrypted file system that can be enabled to protect data on lost or stolen devices.
- User-granted permissions to restrict access to system features and user data.
- Application-defined permissions to control application data on a per-app basis.

Android Proguard

Proguard is free Java class file shrinker, optimizer, obfuscator, and reverified. It detects and removes unused classes, fields, methods, and attributes. Mobile app development companies use **Proguard** in **android**, it optimizes bytecode and removes unused instructions.

While using Proguard, the Applications code is converted to Java bytecode by the Java compiler. After the conversion, it is then optimized by Proguard using the rules which we have written. Then dex converts it to optimized Dalvik byte code.

Android R8

R8 is a tool that converts our Java byte code into an optimized dex code. It iterates through the whole application and then it optimizes like removing unused classes, methods etc. It runs on the compile time. It helps us to reduce the size of the build and to make our app to be more secure. R8 uses Proguard rules to modify its default behavior.

Android App Release

Publishing is the general process that makes your Android applications available to users. When you publish an Android application you perform two main tasks:

- You prepare the application for release.

  During the preparation step you build a release version of your application, which users can download and install on their Android-powered devices.

- You release the application to users.

  During the release step you publicize, sell, and distribute the release version of your application to users.

. keystore File

Java **keystore** (. jks or . **keystore**): A binary **file** that serves as a repository of certificates and private keys. Play Encrypt Private Key (PEPK) tool: Use this tool to export private keys from a Java **Keystore** and encrypt them for transfer to Google Play.

App Bundle

An **Android App Bundle** is a publishing format that includes all your **app's** compiled code and resources, and defers APK generation and signing to Google Play. ... You no longer have to build, sign, and manage multiple APKs to optimize support for different devices, and users get smaller, more-optimized downloads.

Android SDK

Android SDK is a comprehensive list of all files required to kickstart the creation of an application by the developer.

The kit includes tools like ADB bridge, build tools, SDK tools, additional codes to create java or kotlin files, and emulator to begin creating the groundwork for an app.

Every time Google comes up with a new version of Android, a new SDK is released. Developers are required to download and use the latest versions to develop apps for incoming phones in the market.

Components:

Platform Tools

Platform tools are specific to the version of Android you want to create applications for. Once downloaded, these tools need to be continuously updated. The platform tools are backward-supportive means they usually support any older version of Android.

Build Tools

Build tools include tools that are necessary for an app to use the minimum amount of battery power while it is in operation.

SDK Tools

The SDK tools are composed of debuggers, libraries with important resources, sample code for creating java profiles, and other tutorials.

Ideally, for any app development, find the required SDK tool and using it is the first step. It is similar to setting up your workshop, when making a new piece of furniture.

Android Emulator

This is a virtual device that can be set to imitate a particular Android device as a target. It fulfils almost all the needs of the required device and can be used to check and execute developing Android applications.

Android Debug Bridge

This tool is considered extremely versatile and lets you debug and run apps, all along running a variety of different commands on the device.

How to install Android SDK?

Luckily for developers, today, the Android SDK and Java Development Kit (JDK) are a part of Android Studio.

As a result, once a developer downloads Android Studio, it can function in the background, and they don't have to take the pain to download two separate applications. However, once downloaded, the Android SDK will require regular updates and the SDK manager is extremely helpful in managing those updates.

Android apk compilation process

The build process for a typical Android app module:

- The compiler converts your source code into DEX (Dalvik Executable) files, which include the bytecode that runs on Android devices, and everything else into compiled resources.
- The APK packager combines the DEX files and compiled resources into a single APK. Before your app can be installed and deployed onto an Android device, however the APK must be signed.
- The APK packager signs your APK using either the debug or release keystore:
    - If you are building a debug version of your app, that is, an app you intend only for testing and profiling, the packager signs your app with the debug keystore. Android Studio automatically configures new projects with a debug keystore.
    - If you are building a release version of your app that you intend to release externally, the packager signs your app with the release keystore.
- Before generating your final APK, the packager uses the **zipalign** tool to optimize your app to use less memory when running on a device.

Android Interview Questions

ART – Android Runtime

The Dalvik Virtual Machine is dead. Yes, Google stopped using it in 2014, although you will find most of the Android tutorials online, still not updated, but please be informed that Dalvik Virtual Machine is not used in Android anymore.

The new runtime is known as ART or Android Runtime which is very well compatible with its predecessor Dalvik, but do comes in with a lot of new features like:

- Ahead-of-Time compilation
- Improved Garbage collection
- Improved Debugging and diagnostics

ADB (Android Debug Bridge)

Android Debug Bridge (ADB) is a command-line tool that lets you communicate with a device. The ADB command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run variety of commands on a device. It is a client-server program that includes three components:

- A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an ADB command.
- A daemon (adbd), which runs commands on a device. The daemon runs as a background process on each device.
- A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

The R Class

When your application is compiled the R class is generated. It creates constants that allow you to dynamically identify the various contents of the res folder, including layouts.

setContentView

it inflates the layout. Essentially what happens is that Android reads your XML file and generates Java objects for each of the tags in your layout file. You can then edit these objects in the Java code by calling methods on the Java objects.

findViewById

findViewById does exactly as the name implies. It gets a view from your XML layout by its ID, and stores it for you in a Java object.

setText vs append

The difference between setText and append is that setText overwrites what was in the TextView, while append simply adds text onto whatever text was already there.

NOTES

- wrap_content
  wrap_content will shrink the view to wrap whatever is displayed inside the view.
  For example, if the view is filled with the text "Hello world" then wrap_content for the width will set the width of the view to be exact width that "Hello world" takes up on the screen.
- match_parent
  match_parent will expand the size of the view to be as large as the parent view which it is nested inside of.
- dp or dip (density independent pixels)
  is an abstract unit that is based on physical density of the screen. These units are relative to a 160-dpi screen, so one dp is one pixel on a 160-dpi screen.
- sp (Scalable Pixels or scale independent pixels)
  this is like the dp unit, but it is also scaled by the user's font size preference. It is recommended you use this unit when specifying font sizes, so they will be adjusted for both the screen density and user's preference.
- px (Pixels) corresponds to actual pixels on the screen.
- Padding and Margin
  Padding and layout_margin are two very similar attributes. Both determines the space around a view. The difference is that padding determines space within the boundaries of the view, and layout_margin determines the space outside the boundaries of the view. For

Accessibility

Accessibility refers to the design of products, devices, services or environments for people who experience disabilities. Android provides accessibility features like,

- TalkBack which is a pre-installed screen reader service provided by Google. It uses spoken feedback to describe to describe the results of actions such as launching an app, and events such as notifications.
- Explore by Touch which is a system feature that works with TalkBack, allowing you to touch your device's screen and hear what's under your finger via spoken feedback. This feature is helpful to users with low vision.
- Accessibility settings that let you modify your device's display and sound options, such as increasing the text size, changing the speed at which text is spoken and more.

Android Instant run

Android Instant run is an Android feature that is implemented in android studio version 2.3 and higher. It reduces considerably the time when you update the code or resources of the project such as Java code, layout, styles.

As well as when you deploy the app, you can just click apply changes to update the running app without building a new android package or apk. (Menu Run → Apply changes)

Android Project Folder

Your Android Project contains:

- Kotlin files for the core logic of the app
- A resources folder for static content such as images and strings
- An Android Manifest file that defines essential app details so the OS can launch your app
- Gradle scripts, for building and running app

AppCompatActivity

AppCompatActivity is a subclass of Android that gives us access to modern android features while preventing backwards compatibility with older versions of Android.

Functionally, what you should always do is, use AppCompatActivity to make your app available to the largest number of devices and users possible.

View Binding

View binding is a feature that allows you to more easily write code that interacts with views. Once view binding is enabled in a module, it generates a binding class for each XML layout file present in that module. An instance of a binding class contains direct reference to all views that have an ID in the corresponding layout.

View binding replaces findViewById. View binding generates a binding object for each XML layout. You use this binding object to reference views, using their resource ids as the name:

```
// Creating a binding object for the main_activity.xml layout
binding = ActivityMainBinding.inflate(layoutInflater)

// Referencing a view with the ID roll_button
binding.rollButton
```

View binding has the following benefits over findViewById:

- **Type safety:** findViewById requires you to specify the type of view you expect to be returned. For example, if you accidentally specify that you expect an ImageButton to be returned when the actual type is a Button, you'll get a ClassCastException. View binding protects you from this type of error because the view is a correctly typed property of the binding.

- **Null Safety:** findViewById expects an integer parameter, which is meant to be the resource ID of a view. It is possible, though to pass in any integer as a parameter, including unrelated integers and invalid view ids. If you supply an integer that doesn't match a view resource id in the layout, findViewById returns null and can cause a NullpointerException. View binding is null safe because you reference view objects directly and don't look them up by integer IDs.

lateinit

This keyword promises the Kotlin compiler that the variable will be initialized before calling any operations on it.

Basically, we're promising that we will not leave it as null and therefore we don't need to set it to null or have it been a nullable variable. We can go ahead and treat it as non-null anywhere we use it.

XML tools Name space

- The tools namespace is used when you want to find dummy content that is only used when you're previewing the app in the preview pane.
- Attributes using the tools data space are removed when you actually compile the app. So, the whole line will be gone by the time that we compile.
- Namespaces in generally help to resolve ambiguity when referring to attributes that have the same name.

What is Android?

Android is a mobile operating system based on a modified version of the Linux kernel and other open source software, designed primarily for touchscreen mobile devices such as smartphones and tablets.

What is the differnece between android : layout_gravity and android : gravity?

android : layout_gravity is the Outside gravity of the View. Specifies the direction in which the View should touch its parent's border.

android : gravity is the Inside gravity of the View. Specifies in which directionits contents should align.

What does AAPT stand for and what is it all about?

AAPT is an abbreviation for Android Asset Packaging Tool. It helps developer in handling zip-compatible archives including the creation, extraction, and sighting of its components.

Cite some limitations of the Android operating system?

Well, Android certainly suffers from some disadvantages despite inherent benfits:

- Since the sizes of devices vary, it becomes difficult to design apps that can adapt to all screen sizes.
- Certain applications do not run on upgraded Android operating systems.
- Some of the Android apps may not be virus-free. Since the app store is open to everyone, there is no system monitoring the same.

What do you understand by ANR?

ANR stands for Application Not Responding. When an application does not respond for a long time, a dialogue box appears on the screen starting that the application has become unresponsive.

How can you use escape characters in the form of attributes?

Escape characters can be used as attributs by using backslashes twice. For instance, in order to create a new line character, you have to use '\\n'.

How will make support the vector drawables in lower level API in Android ?

Enable the use of support library for vector drawables in build.gradle file (app level):

```
vectorDrawables.useSupportLibrary = true
```

What are the features of Android?

Google has changed the lives of everyone by launching a product that improves the mobile experience for everyone. Android helps in understanding your tests and needs by giving various features such as having wallpapers, themes and launchers which completely change the look of your device's interface.

Android has plenty of features. Some of the features are listed below,

- Open-source
- User Interface
- Multi touch
- Customizable operating system
- Variety of apps can be developed
- Reduces overall complexity
- Supports messaging services, web browser, storage (SQLite), connectivity, media and many more.
- Rich development environment

Apart from these, Android is used in various fields. If your open your playstore, there are several categories where Android can be used.

What is APK format?

The APK file or Android application package is the compressed file format that is used to distribute and install software and middleware onto Google's Android operating system. The file has .apk extension and has all the application code, resource files, certificates and other files, compressed in it.

What is the difference between File, Class and Activity in Android?

- **File**: File is a block of arbitrary information or resources for storing information. It can be any file type.
- **Class**: Class is a compiled form of .java file which Android uses to produce an executable apk.
- **Activity**: Activity is the equivalent of a Frame/Window in GUI toolkits. It is not a file or a file type but just a class that can be extended in Android to load UI elements on view.

What is Google Android SDK? What are the tools placed in Android SDK?

The Google Android SDK is a toolset that provides a developer the API libraries and tools required to build, test, and debug apps for Android in Windows, Mac or Linux. The tools placed in Android SDK are:

- **Android Emulator**
- **DDMS** – Dalvik Debug Monitoring Services
- **AAPT** – Android Asset Packaging tool
- **ADB** – Android Debug Bridge

What is a Toast? Write its syntax.

- Toast notification is a message that pops up on the window.
- It only covers the expanse of space required for the message and the user's recent activity remains visible and interactive.
- The notification automatically fades in and out and does not accept interaction events.

Syntax:

```
Toast.makeText(ProjectActivity.this, "Your message here", Toast.LENGTH_LONG).show();
```

How will you pass data to sub-activities?

We can use **Bundles** to pass data to sub-activities.

There are like HashMaps that and take trivial data types. These Bundles transport information from one activity to another.

```
1. Bundle b=new Bundle();
2. b.putString("Email","abc@xyz.com");
3. i.putExtras(b); // where I is intent
```

What is the use of WebView in Android?

WebView is a view that display web pages inside your application. According to Android, "this class is the basis upon which you can roll your own web browser or simply display some online content within your activity".

It uses the **Webkit rendering engine** to display web pages and includes methods to navigate forward and backward through a history, zoom in and out, perform text searches and more. In order to add WebView to your application, you have to add <WebView> element to your XML layout file.

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 memory.
- External Storage – Store public data on the shared external storage
- SQLite Database – Store structured data in private database

Can you deploy executable JARs (Java ARchive) on Android? Which package is supported by it?

No, Android does not support JAR deployments. Applications are packed into Android Package (.apk) using Android Asset Packaging Tool (APT) and then deployed on to the Android platform.

What is a Sticky Intent?

A sticky intent is a broadcast from sendStickyBroadcast() method which floats around even after the broadcast, allowing others to collect data from it.

Explain Folder, File & Description of Android Apps?

- **src** : contains the .java source files for your project.
- **gen** : contains the .R file, a compiler-generated file that references all the resources found in your project.
- **bin** : contains the Android package files .apk built by the ADT during the build process and everything else needed to run an Android Application.
- **res/drawable-hdpi** : this is a directory for drawable objects that are designed for high-density screens.
- **res/layout** : this is a directory for files that define your app's user interface.
- **res/values** : this is a directory for other various XML files that contain a collection of resources, such as strings and colors definitions.
- **AndroidManifest.xml** : this is the manifest file which describes the fundamental characteristics of the app and defines each of its components.

Can you change the name of an application after its deployment?

It is not recommended to change the application name after its deployment. This may break some functionality.

What is the difference between a fragment and an activity?

Activity is typically a single, focused operation that a user can perform such as dial a number, take a picture, send an email, view a map etc.

Fragment is a modular section of an activity, with its own lifecycle and input events, and which can be added or removed at will. Also, a fragment's lifecycle is directly affected by its host activity's lifecycle i.e., when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all of its fragments.

What database is used in Android? How does it differ from client-server database management systems?

SQLite is the open-source relational database. The SQLite engine is serverless, transactional, and self-contained.

Instead of the typical client-server relationship of most database management systems, the SQLite engine is integrally linked with the application. The library can also be called dynamically and makes use of simple function calls that reduce latency in database access.

What is DDMS?

DDMS stands for Dalvik Debug Monitor Server. It gives the following array of debugging features:

- ✓ Port forwarding services
- ✓ Screen capture on the device
- ✓ Thread and heap information
- ✓ Logcat
- ✓ Incoming call and SMS spoofing
- ✓ Network traffic tracking
- ✓ Location data spoofing

What are the containers?

Containers, holds objects and widgets together, depending on which specific items are needed and in what particular arrangement that is wanted.

Containers may hold levels, fields, buttons or even child containers.

What is the importance of an emulator in Android?

- The emulator lets you play around an interface that acts as if it were an actual device.
- It let you write and test codes, and even debug.
- It acts as a safe place for testing codes especially if its in the early design phase.

What is the difference between a regular bitmap and nine-patched image?

A nine-patch image, unlike bitmap, can be resized and used as background or other image sizes for the target device.

The Nine-patch refers to the way you can resize the image: 4 corners that are unscaled, 4 edges that are scaled in 1 axis, and the middle one that can be scaled into both axes. This is what differentiates a nine-patch image from a regular bitmap.

What are the core building blocks of Android?

- Activity : An activity is the screen representation of any application in Android. Each activity has a layout file where you can place your UI.
- Content Provider : Content providers share data between applications.
- Service : It is a component that runs in the background to perform long-running operations without interacting with the user and it even works if application is destroyed.
- Broadcast Receivers : It respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents.

What are the dialog boxes that are supported in Android?

Android supports four dialog boxes:

1. AlertDialog : An alert dialog box supports zero to three button and a list of selectable elements, including checkboxes and radio buttons. Among the other dialog boxes, the most suggested dialog box is the alert dialog box.
2. ProgressDialog : This dialog box displays a progress wheel or a progress bar. It is an extension of AlertDialog and supports adding buttons.
3. DatePickerDialog : This dialog box is used for selecting a date by the user.
4. TimePickerDialog : This dialog box is used for selecting time by the user.

What is a sticky broadcast? Give an example.

A sticky broadcast is used for communicating between applications. These broadcasts happen without the user being notified. The Android OS normally treats each application as if it were a separate user.

Example: 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.

What are the exceptions in Android?

- InflateException : When error conditions occur this exception is thrown.
- Surface.OutOfResourceException : When a surface is not created or resized, this exception is thrown.
- SurfaceHolder.BadSurfaceTypeException : When invoked in a surface 'SURFACE_TYPE_PUSH_BUFFERS', this exception is thrown from lockCanvas ( ) method.
- WindowManager.BadTokenException : This exception is thrown at the time of trying to view an invalid WindowManager.LayoutParamstoken.

How are escape characters used as attribute?

Escape characters are preceded by double backslashes (//). For example: a newline character is created using 'n'.

What is orientation in Android? How is it done?

Orientation helps to represent the layout in a row or column.

```
<activity android name=".activity" android: screenOrientation="portrait">
```

What is the significance of the .dex files?

- Android programs are compiled into '.dex' (Dalvik Executable) files, which are zipped into a single .apk file on the device.
- .dex files can be created by translating compiled applications written in Java.
- .dex is format that is optimized for effective storage and memory-mappable executions.

What is AIDL? What are the datatypes supported by AIDL?

AIDL stands for Android Interface Definition Language.

- It handles the interface requirements between a client and a service to communicate at the same level through interprocess communication.
- The process involves breaking down objects into primitives that are Android understandable.

Data Types:

- String
- List
- Map
- CharSequence
- Native Java Data types (int, long, char and Boolean)

Why do you refer to the image as @mipmap/ic_launcher, instead of @drawable/ic_launcher?

The app icon is in the res/mipmap folder (instead of the res/drawable folder) because Android handles app icons in a special way.

Android wants your app icon to be crisp and clear, regardless of the device, so on some large screen devices, Android use a higher-resolution version of the icon compared to other icons in your app for that device.

What is Android Regression?

As a developer, being able to refactor your code is an important skill to have. This means that the functionality of the app will remain the same, but visually the app will look differently. You can't break any existing functionality (for example, you can't lose the images or audio playback capabilities) when switching over to the new design. When you break something, and the user loses the ability to do something in the app compared to earlier versions, this is called regression.

Android App Configurations

If you are developing apps for the enterprise market, you may need to satisfy particular requirements set by an organization's policies. *Managed configuration* or *App configurations*, previously known as *application restrictions*, allow the organization's IT admin to remotely specify setting for apps. This capability is particularly useful for organization-approved apps deployed to a work profile.

For example, an organization might require that approved apps allow the IT admin to:

- Allow or block URLs for a web browser
- Configure whether an app is allowed to sync content via cellular, or just by Wi-Fi
- Configure the app's email setting

Let's say, there are three activities A, B, C. Each having button to start activity without finishing in order A starts B, B starts C, C starts A. Can you show me what all set of lifecycle methods gets called while navigation from A to B, B to C, C to A?

- When Activity A started, its onCreate ( ), onStart ( ), onResume ( ) gets called in order.
- When Activity B started, A : onPause ( ) gets called then B : onCreate ( ), onStart ( ), onResume ( ) gets called in order.
- When Activity C started, B : onPause gets called then C : onCreate ( ), onStart ( ), onResume ( ) gets called in order then B : onStop ( ).
- When again Activity A started from C, As A was not destroyed So, first C : onPause ( ) gets called then A : onRestart ( ), onStart ( ), onResume ( ) gets called in order then C : onStop ( ).

How many ways app can send Broadcast?

Android provides three ways for apps to send broadcast:

1. The sendOrderedBroadcast (intent, String) method sends broadcasts to one receiver at a time.
2. The sendBroadcast (intent) method sends broadcasts to all receivers in an undefined order.
3. The LocalBroadcastManager sends broadcasts to receivers that are in the same app as the sender. If you don't need to send broadcasts across apps, use local broadcasts.

What are configuration changes and when Do they Happen?

Configuration changes cause Android to restart the activity. They include changes to screen orientation, keyboard availability and multi-window mode.

When a configuration changes occurs, you must issue a call to onDestroy ( ). You follow this with a call to onCreate ( ).

What is Material Design?

Material Design is a design language that Google developed in 2014. It consists of the building blocks for your UI components. This language, and the design principles it embodies, are applicable to Android as well as other applications.

What are Quality Guidelines?

Google also provides an extensive list of quality guidelines. You can use this list to ensure that your app similar to current quality standards. If your app passes these tests, you can be sure it meets your user's expectations for performance, stability and security.

How do you manage Resources for different screen sizes?

Android devices come in a variety of sizes and resolutions. When you're building your app, it's important to decide which of the many models you will support. Once you've made this decision, there are three common approaches you can take to make sure your app supports all screen sizes:

1. Use view dimensions
2. Create several layouts, depending on the screen
3. Provide your images and resources as bitmaps

Of course, you may also use a combination of these approaches in your app.

What are Support Libraries?

Android Support Libraries are packages that contain code. They support a specific range of Android platform versions and sets of features. Most of these support libraries are now part of the androidx namespace. This is the recommended set of libraries to use.

What is Localization? How do you do it in Android?

Localization is the ability of an app to support multiple languages, time zones, currencies, number formats etc. This allows you to distribute your app widely to different countries and populations.

Your app can bundle its localization through using Android's resource directory framework, which specifies items (strings, images, layouts…) for different locales.

What is Data Binding?

The Data Binding library helps you bind UI components in your layouts to data source in your app. It uses declarative format.

What is LiveData?

LiveData is an observable class. The advantage of LiveData over libraries like Rx is that it is lifecycle-aware. It only updates data in the STARTED or RESUME state.

LiveData is closely related to coroutines.

What is Room?

Room is a persistence library built over SQLite to help you create databases more easily. Room uses caching to persist your app's data locally. This provides the app with consistent data, regardless of internet connectivity.

What is the Navigation Component?

The navigation component is a framework that controls navigation within an Android app. It manages the back stack. It also handles functions with different controls, such as the app bar or drawers. The navigation component helps you provide a consistent user experience by enforcing established navigation principles.

Where do you organize your Tests in Android?

When you create an Android Studio project, you create two directories for testing:

1. The test directory is for unit tests that run locally. Usually, the JVM runs these tests.
2. The androidTest directory is for tests that run on devices. You'll use this directory for other kinds of tests, such as integration tests and end-to-end tests.

What are Unit Tests? How do you do them in Android?

Unit Tests run locally. Since they aren't run on a device. They don't have access to any Android framework library. It's possible to use libraries that allow you to call Android framework methods from unit tests, however these libraries substitute only simulate device behavior. The preferred libraries are either Junit or Mockito.

The ability to design and implement unit testing is usually a requirement for mid-senior levels.

JUnit: is the standard Java library for testing, which is usually coupled with AndroidX Test.

Mockito: is another popular open-source testing framework.

Instrumentation Tests

Instrumentation tests are quite similar to unit tests but depend on a device or simulator to run. Since instrumentation tests are run on device, you have access to the Android device libraries. The two libraries, Junit and Mockito are also used for instrumentation tests.

UI Tests

UI tests simulate a user's interactions with your UI. The most popular library for UI testing is Espresso.

What is Application class?

The application class in Android is the base class within an Android app that contains all other components such as activities and services. The application class, or anu subclass of the application class, is instantiated before any other class when the process for your application/package is created.

What is Armv7?

There are 3 CPU architectures in Android. ARMv7 is the most common as it is optimised for battery consumption. ARM64 is an evolved version of that, that supports 64-bit processing for more powerful computing. ARMx86, is the least used for these three, since it is not battery friendly. It is more powerful than the other two.

Why bytecode cannot be run in Android?

Android uses DVM (Dalvik Virtual Machine) rather using JVM (Java Virtual Machine).

What is a BuildType in Gradle? And what can you use it for?

Build types define properties that Gradle uses when building and packaging your Android app.

- A build type defines how a module is built, for example whether ProGuard is run.
- A product flavour defines what is built, such as which resources are included in the build.
- Gradle creates a build variant for every possible combination of your project's product flavour and build types.

Explain the build process in Android?

- First step involves compiling the resources folder (/res) using the AAPT (Android asset packaging tool). These are compiled to a single class file called R.java. This is a class that just contains constants.
- Second step involves the Java source code being compiled to .class files by javac, and then the class files are converted to Dalvik bytecode by the "dx" tool, which is included in the SDK tools. The output is classes.dex.
- The final step involves the android apk builder which takes all the input and builds the apk (android packaging key) file.

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 ( )`.

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.

Why would you do the setContentView ( ) in onCreate ( ) of Activity class?

As onCreate ( ) of an Activity is called only once, this is the point where most initialisation should go. It is inefficient to set the content in onResume ( ) or onStart ( ) (which are called multiple times) as the setContentView ( ) is a heavy operation.

onSavedInstanceState ( ) and onRestoreInstanceState ( ) in activity?

onRestoreInstanceState ( )– When activity is recreated after it was previously destroyed, we can recover the saved state from the Bundle that the system passes to the activity. Both the `onCreate ( )` and `onRestoreInstanceState ( )` callback methods receive the same Bundle that contains the instance state information. But because the `onCreate ( )` method is called whether the system is creating a new instance of your activity or recreating a previous one, you must check whether the state Bundle is null before you attempt to read it. If it is null, then the system is creating a new instance of the activity, instead of restoring a previous one that was destroyed.

`onSavedInstanceState ( )` – is a method used to store data before pausing the activity.

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.

Example: Suppose there is an activity stack 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 in 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.

Example: 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.

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.

Example: 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 modes as 'SingleTask', the new activity stack will be A -> B. Activities C and D will be destroyed.

SingleInstance

Same as 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.

Example: 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

How does the activity respond when the user rotates the screen?

When the screen is rotated, the current instance of 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 the similar flow as they were when the activity was first created.

Mention two ways to clear the back stack of activities when a new Activity is called using intent?

The first approach is to use FLAG_ACTIVITY_CLEAR_TOP flag. The second way is by using FLAG_ACTIVITY_CLEAR_TASK and FLAG_ACTIVITY_NEW_TASK in conjunction.

What's the difference between FLAG_ACTIVITY_CLEAR_TASK and FLAG_ACTIVITY_CLEAR_TOP?

FLAG_ACTIVITY_CLEAR_TASK:

It is used to clear all the activities from the task including any existing instances of the class invoked. 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.

FLAG_ACTIVITY_CLEAR_TOP:

If set, and the activity being launched is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it will be closed and this intent will be delivered to the (now on top) old activity as a new intent.

Access data using content providers?

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.

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 communicating 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/minutes.

What is the onTrimemory ( ) method?

`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 from your app in several ways or kill your 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 your app process, you can implement the `ComponentCallbacks2` interface in your Activity

classes. The provided `onTrimMemory ( )` callback method allows your app to listen for memory related events when your app is in either the foreground or the background, and then release objects in response to app lifecycle or system events that indicates the system needs to reclaim memory.

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 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.

Implementing Android bound service with Android Messenger

- Service based on Messenger can communicate with other components in different processes, known as Inter Process Communication (IPC), without using AIDL.
- A **Service Handler**: This component handles incoming requests from clients that interact with the service itself.
- A **Messenger**: This class is used to create an object implementing `IBinder` interface so that a client can interact with the service.

What is a ThreadPool? And it is more effective than using several separate Threads?

Creating and Destroying threads has a high CPU usage, so when we need to perform lots of small, simple tasks concurrently, the overhead of creating our own threads can take up a significant portion of the CPU cycles and severely affect the final response time.

ThreadPool consists of a task queue and a group of worker threads, which allows it to run multiple parallel instances of a task.

Difference between Serializable and Parcelable?

Serialization is the process of converting an object into a stream of bytes in order to store an object into memory, so that it can be recreated at a later time, while still keeping the object's original state and data.

How to disallow serialization?

We can declare the variable as transient.

Serialization is a standard Java interface.

Parcelable is an Android specific interface where you implement the serialization yourself. It was created to be far more efficient than Serializable (The problem with this approach is that reflection is used and it is a slow process. This mechanism also tends to create a lot of temporary objects and cause quite a bit of garbage collection.).

What is a Pending Intent?

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

What are intent filters?

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

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).

Difference between adding/replacing fragment in backstack?

- **Replace** removes the existing fragment and adds a new fragment. This means when you press 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 that means existing fragment will be active and they won't be in 'paused' state hence when a back button is pressed `onCreateView` is called for existing fragment (the fragment which was there before new fragment was added).
- In terms of 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.

Why is it recommended to use only the default constructor to create a Fragment?

The reason why you should be passing parameters through bundle is because when the system restores fragment (e.g., on config change), it will automatically restore your bundle. This way you are guaranteed to restore the state of the fragment correctly to the same state the fragment was initialized with.

You're replacing one Fragment with another – how do you ensure that the user can return to the previous Fragment, by pressing the Back button?

We need to save each Fragment transaction to the backstack, by calling `addToBackStack ( )` before you `commit ( )` that transaction.

What are retained fragments?

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

Difference between FragmentPagerAdapter vs FragmentStatePagerAdapter?

`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. 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.

What is the difference between Dialog and DialogFragment?

A fragment that displays a dialog window, floating on top of its activity's window. This fragment contains a Dialog object, which it displays as appropriate based on the fragment's state.

Dialogs are entirely dependent on Activities. If the screen is rotated, the dialog is dismissed. Dialog fragments take care o orientation, configuration changes as well.

When might you use a 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.

What are Adapters?

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

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 optimise the UI design for certain screen sizes. For example, different UI for tablets.
- **Use the smallest width qualifier**: for example, you can create a layout named `main_activity` 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 your layout.

Outline the process of creating custom views?

- Create a class that Subclass a view
- Create a res/values/attrs.xml file and declare the attributes you want to use with your custom view.
- In your View class, add a constructor method, instantiate the Paint object, and retrieve your custom attributes.
- Override either `onSizeChanged ( )` or `onMeasure ( )`.

- Draw your View by overriding `onDraw ( )`.

Briefly describe some ways that you can optimize View usage.

- Checking for excessive overdraw: install your app on an Android device, and then enable the "Debug GPU Overview" option.
- Flattening your view hierarchy: inspect your view hierarchy using Android Studio's 'Hierarchy Viewer' tool.
- Measuring how long it takes each View to complete the measure, layout, and draw phases. You can also use Hierarchy Viewer to identify any parts of the rendering pipeline that you need to optimise.

Bitmap pooling in Android?

Bitmap pooling is a simple technique, that aims to reuse bitmaps instead of creating new ones every time. When you need a bitmap, you check a bitmap stack to see if there are any bitmaps available. If there are not bitmaps available you create a new bitmap otherwise you pop a bitmap from the stack and reuse it. Then when you are done with the bitmap, you can put it on a stack.

What is the permission protection levels in Android?

Normal – A lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, without asking for the user's explicit approval.

Dangerous – A higher-risk permission. Any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.

Signature – A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.

SignatureOrSystem – A permission that the system grants only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission.

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.

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 of time, only 10 or so items could fir

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.

- **But what happens when you start scrolling and need to start showing next views?** Again, a simple approach would be creating 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 you're 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 happen 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 they held with new data.

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 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 "ViewHolder" 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

How would you implement swipe animation in Android?

```
<set xmlns:android="http://schemas.android.com/apk/res/android" android:shareInterpolator="false">
<translate android:fromXDelta="-100%"
android:toXDelta="0%"
android:fromYDelta="0%"
android:toYDelta="0%"
android:duration="700"/>
</set>
```

Launch vs Async in Kotlin Coroutines

The difference is that the `launch { }` does not return anything and the `async { }` returns an instance of `Deferred <T>`, which has an `await ( )` function that returns the result of the coroutines like we have future in Java in which we do `future.get ( )` to get the result.

In other words:

- `Launch`: fire and forget
- `Async`: perform a task and return a result

Example: Suppose we have a function like this,

```
fun fetchUserAndSaveInDatabase (){

        // fetch user from network

        // save user in database

        // and do not return anything

}
```

Now, we can use the launch like below:

```
GlobalScope.launch(Dispatchers.Main) {

        fetchUserAndSaveInDatabase () // do on IO thread

}
```

As the function doesn't return anything, we can use launch to complete that task and then do something on main thread.

Now, suppose we need the result back, we need to use the async.

For example: we have two functions like below which return user, the first one is fetch first user and we have another function is fetch second user. So, both will be making the network call and returning the user.

Here I have not made the function suspend, we can make it based on the requirement.

```
fun fetchFirstUser (): User {

        // make network call

        // return user}

fun fetchSecondUser (): User {

        // make network call

        // return user}
```

Now, we can use the async like below:

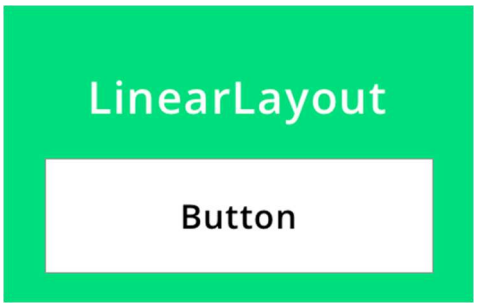
```
GlobalScope.launch(Dispatchers.Main) {

        val userOne = async(Dispatchers.IO){fetchFirstUser()}

        val userTwo = async(Dispatchers.IO){fetchSeconduser()}
```

```
        showUsers (userOne.await(), userTwo.await()) // back on UI thread

}
```

We can call both the function and user 1 and user 2 are here the deferred one so it means, it will have the function of await, we can call the await on those. So, we can call like this and then we can show into the UI back on the UI thread. So, here it will make both the network call in parallel and waits for the results and then we will call the show user function.

What is Touch Control and Events in Android?

In this case, we are going to talk about a case where we have a LinearLayout containing a button like,

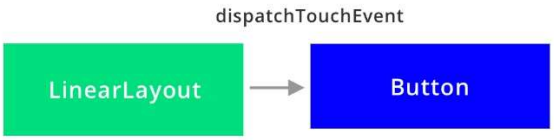


What happens when we touch the screen?

So, when we touch the screen the activity's view gets the touch event notification first also known as `DecorView` in Android. Now, we generally don't work with the touch of `DecorView`. So, the touch gets transferred to the ViewGroup and subsequently to its children in the XML file.

But how can we transfer the touch event trigger?

In Android, the ViewGroup transfers the touch events from top to bottom in the hierarchy of ViewGroup to its children using `dispatchTouchEvent ( )`.



How do we intercept touch events?

First when we perform a touch action, then ViewGroup gets the touch event, and then it is intercepted in the ViewGroup itself using `onInterceptTouchEvent ( )`.

If on intercepting if we return `true` then the touch event is not passed to its children and if we pass `false`, the Android eco-system gets notified that the ViewGroup wants to dispatch the event to its children, in our case it is a button.

In general, if returning true, it means we have handled the event in the ViewGroup itself, no need to dispatch to its children.

Now, as the button we have, is the last view in our view tree. So, it won't be able to pass the touch event to its children anymore as it has none. So, in button, we would have our last `onInterceptTouchEvent` being called.

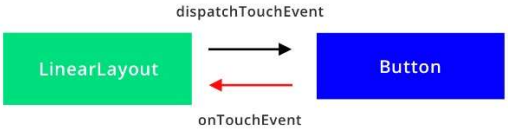
Intercepting of the event can only happen in ViewGroups and not Views.

How touch is handled in the view?

When dispatching the touch event from top to bottom in the view hierarchy, we need to see at which position of the tree we need to handle the touch on the view.

When handling the dispatching event, the top position of the hierarchy takes the lead, but when it comes to handling the touch the child views using `onTouchEvent` are always the first, and then it keeps moving towards the ViewGroups.

Touch event works just like the dispatching of the events but in the reverse order from child to parent.



Let's say if we dispatch the event from ViewGroup and intercept the event there, it depends on the return value (true/false) that shall the touch of the view be handled on the ViewGroup or the children.

So, in our case, if the `onTouchEvent` of `Button` returns true, then it means that it has been handled and then, it will not go to the `LinearLayout`.

What are touch events we majorly work within Android for handling the touch control?

When we get the touch event, it gets handled by `onTouchEvent` which also has a parameter of type `MotionEvent`.

```
fun onTouchEvent (event: MotionEvent)
```

All the task performed regarding the touch has its reference in the event parameter. We can have the coordinates like X and Y points on the screen of the point on the touch.

It even has the actions in it, like let's see if we tap on the screen then `MotionEvent.ACTION_DOWN` is called and when we lift the touch `MotionEvent.ACTION_UP` is called.

Even dragging a finger on the screen, the action is `MotionEvent.ACTION_MOVE`.

So, the flow on the view happens is when we want to tap the button,

`Activity` -> `dispatchTouchEvent (LinearLayout)` -> `dispatchTouchEvent (Button)` -> `onTouchEvent (Button)`

And when don't want to tap the button but want to handle the click on LinearLayout, the flow would be:

`Activity` -> `dispatchTouchEvent (LinearLayout)` -> `dispatchTouchEvent (Button)` -> `onTouchEvent (Button) (will return false)` -> `onTouchEvent (LinearLayout)`.

Integrating Work Manager in Android?

What is Work Manager?

Work manager is a library part of Android Jetpack which makes it easy to schedule deferrable, asynchronous tasks that are expected to run even if the app exists or device restarts i.e., even your app restarts due to any issue Work Manager makes sure the scheduled task executes again. Isn't that great?

To integrate work manager in your project,

```
dependencies {

  def work_version = "2.2.0"

    implementation "androidx.work:work-runtime:$work_version"

  }
```

Now, as a next step, we will create a **Worker** class. Worker class is responsible to perform work synchronously on a background thread provided by the work manager.

```
class YourWorkerClass (appContext: Context, workerParams: WorkerParameters):
Worker(appContext, workerParams) {

    override fun doWork(): Result {

       // Your work here.

      // Your task result

        return Result.success()

    }

}
```

In the above class,

- `doWork ( )` method is responsible to execute your task on the background thread. Whatever task you want to perform has to be written here.
- Result returns the status of the work done in `doWork ( )` method. If it returns `Result.success ( )` it means the task was successful if the status is `Result.failure ( )`, the task was not-successful and lastly, if it returns `Result.retry ( )` it means the task will execute again after some time.

Difference between invisible and gone for the View visibility status?

Invisible:

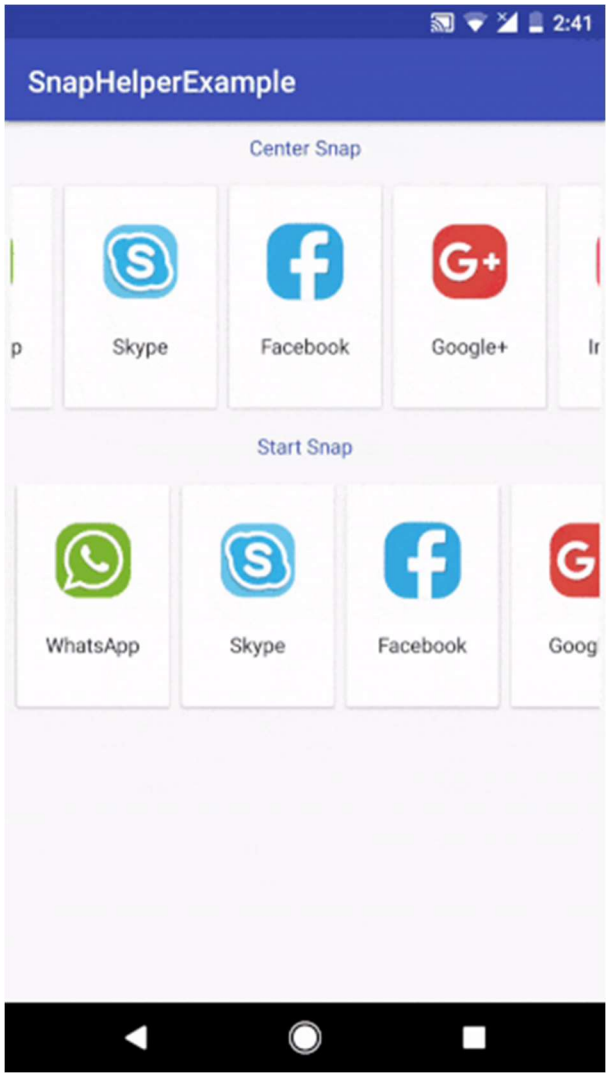
This view is invisible, but it still takes up space for layout purposes.

Gone:

This view is invisible, and it doesn't take any space for layout purposes.

What is SnapHelper in Android?

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.



I have an application and I need to be able to share data with other applications, how would I at first initially approach that?

So, sharing data between apps, I mean if the data is going to be unique to your app and that your app is generating some kind of data, that only that has or only that app would store then I would use a **content provider**.

Content providers it, pretty much interacts with some sort of database, some sort of data persistence on the back end of may be shared preferences or something like that or SQL database, SQLite.

But then if you have a content provider that then becomes accessible to any app on your Android device via URI. So, you could call your content resolver to find where this URI is and communicate with the content provider to manipulate that database, either pull data or perhaps push data or update data.

And so, what would be a good example where a content provider would make sense, so something provides by the system?

So, I think a good example is just what comes standard on an Android phone and that would be your **contact list**.

Say you want to upload a contact from some new way, may be a website or take a picture of their contact information. And you build up that data, and then using the content resolver you access the address at address book and you push that contact into your contacts.

Explain to me some of the differences between implicit and explicit intents?

So, for **Explicit** intents, you are very specific about the activity you want to launch. So that's probably what you use most often is, I'm saying, I need to launch my login activity or some other activity in my application. And that's usually done with explicit intent.

With **Implicit** intents, you can kind of say, I need to be able to send an email and here's a message. I don't know about the client that's going to receive it. You can also build them within your own application, so if you care about switching out that implementation later, you could do that using implicit intents.

If you wanted to throw your login activity up, you don't necessarily have to specify the explicit one. You could instead specify an implicit intent to launch that activity.

So, if I had an application that was an image editor. And someone opens an image, it could launch an intent to my app if I enabled that.

So, a problem that often confuses new Android developers is usually around rotation, right? So, a common issue you'll see is like if I have a progress bar or I have supposed I 'm building a game and I have a score or something. But when I rotate, I lose that information I lose like maybe the progress either set it out or the score that I calculated at the time. Can you explain to me some more detail why that is occurring?

So, this has to do with one fundamental concepts of Android development and the Android operating system, which is the activity lifecycle or fragment lifecycle as well.

So, when you create a fragment, it's attached with your activity. When you rotate a device, your activity is destroyed. Your fragment will be destroyed. And so, if you don't have any sort of check to see if your application has been running, the activity will reset as if it's the first time it opened.

Everything will, any sort of data or something that you change throughout the lifecycle of the application will then start anew.

So, what you're saying is that, this is all information that was either set by the user, before the activity started? So, these are the things where they mutated state while the activity was running and then we need to save those off.

Right, and even if we do save them somewhere and we turn the device again if we don't reference that, nothing's going to happen.

Answer the below questions for the code?

```
int a = 1000;

Integer b = 1000;

Integer c = new Integer (1000);
```

1. a == b (**true**)
2. b == c (**false**)
3. b. equals (a) (**true**)
4. c. equals (b) (**true**)

This (==) equals compare the memory references or address references, where A and B are stored in the heap and (.) equals method is used for the content comparisons.

Also dot equals is a method declared in the object class of Java. Any class that we create inherit the object class of Java. So, by default this method appears in any class that you create and the default implementation of dot equals is that it also compares the memory address that is exactly equal to == by default.

But integer class is the class provided by the Java and this class overrides this dot equals method and here it declares that we have to check the value or content stored at A and B that is the integer values 1000 here and 1000 here will be compared by the implementation of integer class.

Now, the first is **a == b**, so this int a is a primitive type and Integer b is the object type. Whenever this comparison occurs, the integer class provides auto boxing and unboxing. So, here when since a is integer b will unbox itself in the value of 1000. So, 1000 primitive value is equals to 1000 primitive value. So, a ==b is **true**.

Now. in **b == c**,

When I declare Integer b = 1000; Java creates a new object and assign it to b.

Also, in other case (Integer c = new Integer (1000)), when I comparing b equals to c, then these two are different objects. And this will then compare the memory addresses. So, it will be **false**.

Now, in **b. equals (a),** it will compare the values of 1000 by the default implementation of integer where it overrides the equal, so this will be **true.**

Similarly, **c. equals (b),** is also **true**.

Answer the following questions?

```
String a = "abc";

String b = "abc";

String c = new String ("abc")
```

1. a == b -> true
2. b == c -> false
3. a. equals (b) -> true
4. c. equals (b) -> true

In Java, when we create string literals, it creates the object and stores in the heap and is shared by any other declaration for the same string literal.

So, a and b in fact will point to the object 'abc', which is the same memory reference, so **Q. 1** will be **true**.

And for **Q. 2**, this will be **false**. Because c is especially you have mentioned new keyword. So, it will be allocated separately in the heap and in the string pool. So, they have the different memory references.

In **Q. 3**, if I write a dot equals b, the dot equals implementation of string compares the values, so this will be **true**.

Similarly, **Q. 4** is also **true**.

Answer the below questions from the mentioned code?

```
1.  class Int {
2.     public int val;
3.
4.     public Int (int val){
5.                 this.val = val;
6.     }
7.  }
8.  Int a = new Int(1000);
9.  Int b = new Int(1000);
```

1. a == b -> false
2. a. equals (b) ->false

If you think that since it is int values there, then this will compare the int value. Then this is not true because they are two different objects a and b, so this objects a and b will have different memory references in the heap and both will have a value of 1000 in that heap address. So, Q. 1 is false.

Also, Q. 2 is false, because dot equal by default as defined in the object class compares the memory addresses so, a and b have different memory addresses. So, the result will be false.

```
1.  class Int {
2.     public int val;
```

```
3.
4.      public Int(int val){
5.                  this.val = val;
6.      }
7.      public boolean equals(Int b){
8.                  return this.val = b.val;
9.      }
10. }
11.  Int a = new Int(1000);
12.  Int b = new Int(1000);
```

1. a == b -> false
2. a. equals(b) -> true

In the same class, I add a method equals and it returns by comparing the value if this.val and past variable b. equals is compared and then returned then this dot equals method will be true and == will be false.

NOTE:

== compares the memory references and dot equals by default compare the memory addresses but overridden value will compare based on the class definition.

AsyncTask Questions:

AsyncTask is very fundamental to how Android works with threading. So, in the current scenario the developer today tends to use RxJava and other frameworks for doing background operations. So, AsyncTask has become less in experiences but this very important for three things (means AsyncTask has been chosen),

1. It will test the framework level exposure of the developer that if the developer has an understanding how the Android framework works.
2. It will test the knowledge of asynchronous app behavior means how the developer should program in asynchronous environment.
3. The familiarity with the life cycle of the activity

So, all these things are tested with the user AsyncTask.

What is a background thread?

Any thread other than the UI thread or main thread which we execute to get some result is called a background thread.

How does AsyncTask help to achieve background execution?

AsyncTask creates a worker thread and executes the task through that thread.

How to update the UI through AsyncTask?

Suppose you have an application; the application fetches some data from the server and after that it updates that UI. So, for this model you need to use AsyncTask?

Q. after AsyncTask gets the result from the server, how can it update the UI.

Basically, the catch here is that, UI in Android has been designed to be updated by the main thread or the thread which has created it.

So, the question will be; how does this UI get updated through the main thread when the execution was being done by the worker thread.

AsyncTask provides us with two methods specifically for these scenarios, and first method is **doing background**, so doing background will do the operation on the worker thread and then we have **on post execute**. So, on post execute will post the result that will be on the main thread. So, we can update the UI from the main thread.

NOTE

Synchronous: waits until the task has completed.

Asynchronous: completes a task in the background and can notify you when you complete.

Suppose, you will be given 3 async task A, B and C. and behavior is like below,

AsyncTask A → completes in 3 sec.

AsyncTask B → completes in 1 sec.

AsyncTask C → completes in 2 sec.

After that *A. execute ( )*, *B. execute ( )* and *c. execute ( )* is called. So, which is finish first?

If you don't know about AsyncTask design then you would reply that A, B, C anyone could be finished before other, because this is asynchronous operations. But this is not the case.

AsyncTask, by default creates a single worker thread universally for the application. And if we execute 3 tasks, all these tasks will be executed serially. So, A will execute first and B then and then C.

The next Question will be:

Suppose, you want to make A, B and C execute parallelly, then what should you do?

Here, you should know the concept of **thread pools**. So, there are other methods for that and one method is *execute on executors*.

In execute on executors, you have to pass the thread pool executor and after that A, B and C will run in parallel.

If ABC was running and the activity was back pressed then what will happen to the task ABC?

You may think that ABC would be stopped but that is not the case.

A, B and C will execute itself without any interference with the activity's existence and after ABC has been finished, on post execute will be called.

When onPostExecute was called and we were updating the UI. So, what problem that can arise in the scenario.

So, you have to think that this activity is destroyed so view might have been destroyed and in on post execute you are updating the UI. So, you will be referencing the memory of those view which may not exist. So that may lead to exception.

How to resolve the above issue?

Means we want to solve the issue where we only call method to update the UI, when the view actually exists. So, there can be lots of solutions for this. So, one of the solutions could be that you create your AsyncTask, extend an AsyncTask you take a call back object. You keep the call back instance in the weak reference because call back may be implemented by the activity. So, you would not want to hold the reference to the activity and then check if the callback is not null before sending result in the onPostExecute. So, if I do this then the problem, when a view was updated, when it was before destroyed would be solved and this actually can be solved in a number of ways.

You can solve this by using **Loaders** or **ViewModel**.

Java or Kotlin? Which is better for Android?

Kotlin Application Deployment is faster to compile, lightweight and prevent application from increasing size. Any chunk of code written in **Kotlin** is much smaller compared to Java, as it is less verbose and less code means fewer bugs. Kotlin compiles the code to a bytecode which can be executed in the JVM.

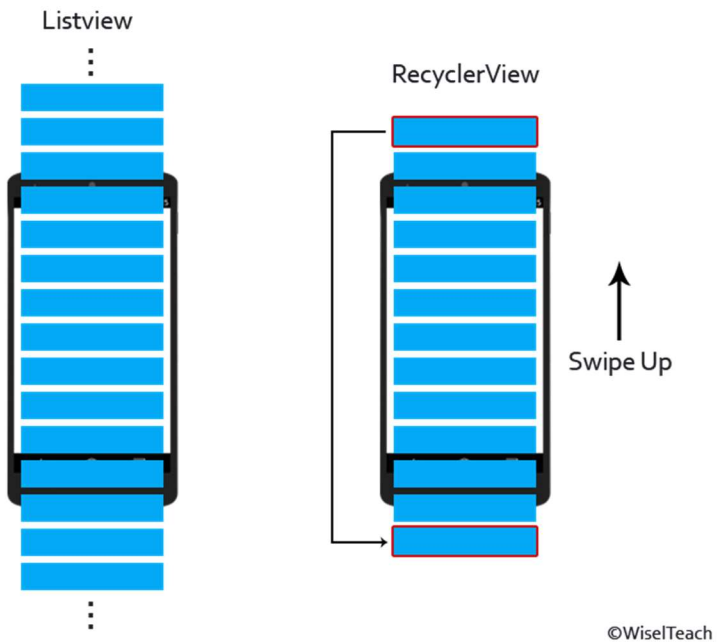
Difference between `onStart ( )` and `onResume ( )`.

`onStart ( )` called when the activity is becoming visible to the user. But might not be in the foreground (e.g. an AlertFragment is on top or any other possible use case).

`onResume ( )` called when the activity will start interacting with the user. Or called when the activity is in the foreground, or the user can interact with the Activity.

Difference between RecyclerView and ListView

With Android Lollipop, the **RecyclerView** made its way officially. The RecyclerView is much more powerful, flexible and a major enhancement over **ListView**.



1. ViewHolder Pattern.
   In a ListView, it was recommended to use the ViewHolder pattern but it was never a compulsion. In case of RecyclerView, this is mandatory using the RecyclerView.ViewHolder class. This is one of the major differences between the ListView and the RecyclerView.

2. LayoutManager
   This is another massive enhancement brought to the RecyclerView. In a ListView, the only type of view available is the vertical ListView. There is no official way to even implement horizontal ListView.

   Now using a RecyclerView, we can have a
   - LinearLayoutManager which supports both vertical and horizontal lists.
   - StaggeredLayoutManagerdf which supports Pinterest like staggered lists.
   - GridLayoutManager which supports displaying grids as seen in Gallery apps.

3. Item Animator
   ListViews are lacking in support of good animations, but the RecyclerView brings a whole new dimension to it. Using the RecylerView.ItemAnimator class, animating the views becomes so much easy and intuitive.

4. Item Decoration
   In case of ListViews, dynamically decorating items like adding borders or dividers was never easy. But in case of RecyclerView, the RecyclerView.ItemDecorator class gives huge control to the developers but makes things a bit more time consuming and complex.

5. OnItemTouchListener
   Intercepting item clicks on a ListView was simple, thanks to its AdapterView.OnItemClickListener interface. But the RecyclerView gives much more power and control to its developers by the RecyclerView.OnItemTouchListener but it complicates things a bit for the developer.
   In simple words, the RecyclerView is much more customizable than the ListView and gives a lot of control and power to its developers.

6. Performance on Loading
   RecyclerView prepares view just ahead and behind the visible entries, which is great if you are fetching bitmaps in background. Performance is dramatically faster, especially if you use RecyclerView.setHasFixedSize. The old ListView is based on the premise that there's no way to precalculate or cache the size of entries in the list, which causes insane complications when scrolling and performing layout. Takes a while to get used to it, but once you do, you'll never go back.

How can two distinct Android apps interact?

At the simplest level there are two different ways for apps to interact on Android:

- Via Intents, passing data from one application to another
- Through services, where one application provides functionality for others to use.

How would you communicate between two Fragments?

All **Fragment-to-Fragment communication** is done either through a shared **ViewModel** or through the associated **Activity**.

Two fragments should never communicate directly.

The recommended way to communicate between fragments is to create a **Shard ViewModel** object. Both fragments can access the ViewModel through their containing Activity. The fragments can update data within the ViewModel and if the data is exposed using LiveData the new state will be paused to the other fragment as long as it is observing the LiveData from the ViewModel.

Another way is to define an **interface** in your fragment.

What is Android Data Binding?

The **Data Binding Library** is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

Layouts are often defined in activities with code that calls UI framework methods. For example, the code below calls `findViewById ( )` to find a TextView widget and bind it to the `userName` property of the `viewModel` variable:

```
TextView textView = findViewById(R.id.sample_text);

textView.setText(viewModel.getUserName());
```

The following example shows how to use the Data Binding Library to assign text to the widget directly in the layout file. This removes the need to call any of the Java code shown above.

```
<TextView

   android:text="@{viewmodel.userName}" />
```

The pros of using Android Data Binding:

- Reduces boilerplate code which in turns bring
- Less coupling
- Stronger readability
- Powerful, easy to implement custom attribute and custom view
- Even faster than `findViewById` – The binding does a single pass on the View hierarchy, extracting the Views with IDs. This mechanism can be faster than calling `findViewById` for several Views.

What is a JobScheduler?

The **JobScheduler** API performs an operation for your application when a set of predefined conditions are met (such as when a device is plugged into a power source or connected to a Wi-Fi network). This allows your app to perform the given task while being considerate of the device's battery at the cost of timing control.

Here is the example when you would use this job scheduler:

- Tasks that should be done once the device is connect to a power supply

- Tasks that require network access or a Wi-Fi connection
- Tasks that are not critical or user facing
- Tasks that should be running on a regular basis as batch where the timing is not critical.

What is the ViewHolder pattern? Why should we use it?

Every time when the adapter calls `getView ( )` method, the `findViewById ( )` method is also called. This is a very intensive work for the mobile CPU and so affects the performance of the application and the battery consumption increases. `ViewHolder` is a design pattern which can be applied as a way around repeated use of `findViewById`.

A `ViewHolder` holds the reference to the id of the view resource and calls to the resource will not be required after you "find" them: Thus, performance of the application increases.

`View.setTag (Object)` allows you to tell the View to hold an arbitrary object. If we use it to hold an instance of our ViewHolder after we do our `findViewById (int)` calls, then we can use `View.getTag ( )` on recycled views to avoid having to make the calls again and again.

What is the difference between Handler vs AsyncTask vs Thread?

The **Handler** class can be used to register to a thread and provides a simple channel to send data to this thread. A handler allows you communicate back with the UI thread from other background thread.

The **AsyncTask** class encapsulates the creation of a background process and the synchronization with the main thread. It also supports reporting progress of the running tasks.

And a Thread is basically the core element of multithreading which a developer can use with the following disadvantage:

- Handle synchronization with the main thread if you post back results to the user interface
- No default for canceling the thread
- No default thread pooling
- No default for handling configuration changes in Android

What is difference between compileSdkVersion and targetSdkVersion?

The `compileSdkVersion` is the version of the API the app is compiled against. This means you can use Android API features included in that version of the API (as well as all previous versions). If you try and use API 16 features but set `compileSdkVersion` to 15, you will get a compilation error. If you set `compileSdkVersion` to 16 you can still run the app on an API 15 device as long as your app's execution paths do not attempt to invoke any APIs specific to API 16.

The `targetSdkVersion` has nothing to do with how your app is compiled or what APIs you can utilize. The `targetSdkVersion` is supposed to indicate that you have tested your app on (presumably up to and including) the version you specify. This is more like a certification or sign off you are giving the Android OS as a hint to how it should handle your app in terms of OS features.

What is the difference between a Bundle and an Intent?

A **Bundle** is a collection of key-value pairs.

However, an **intent** is much more. It contains information about an operation that should be performed. This new operation is defined by the action it can be used for, and the data it should show/edit/add. The system uses this information for finding a suitable app component (activity/broadcast/service) for the requested action.

What is Android Gyroscope?

Most Android-powered devices have an **Accelerometer**, and many now include a **Gyroscope**. For example, during a single sensor event the accelerometer returns acceleration force data for the three coordinate axes, and the Gyroscope returns rate of rotation data for the three coordinate axes.

An **acceleration** sensor measures the acceleration applied to the device, including the force of gravity. Accelerometer use the standard sensor coordinate system.

The **gyroscope** measures the rate of rotation in rad/s around a device's x, y and z axis.

Standard gyroscope provides raw rotational data without any filtering or correction for noise and drift (bias). In practice, gyroscope noise and drift will introduce errors that need to be compressed for.

The following code shows you how to get an instance of the default gyroscope:

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val sensor: Sensor? = sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE)
```

Fragment Lifecycle

Fragments are hosted by an Activity and it contains the same set of activity lifecycle callbacks `onCreate`, `onStart`, `onResume`, `onPause`, `onStop` and `onDestroy`.

When activity transitions between the states it calls the same six life cycle callbacks for all the fragments inside the current activity.

So, what's the difference between the Fragment Lifecycle and Activity Lifecycle?

The main difference is that the Activity creates only **one view** for the entire life cycle but Fragment **views** can be recreated and even dynamically changed during the life cycle. Because of this difference we need treat fragment life cycle a little bit differently and we will start with `onCreate`.

This callback is called immediately after on creating the activity. At this time oncreating an activity may not be finished yet, so work with the views here can lead to a crash. therefore, don't use this callback for anything related to the views

The **main goal** of fragment is to represent the view and that's why on view created callback is heavily used.

What is difference between FragmentPagerAdapter vs FragmentStatePagerAdapter?

FragmentPagerAdapter: Each fragment visited by the user will be stored in the memory but the view will be destroyed. When the page I revisited, then the view will be created not the instance of the fragment.

FragmentStatePagerAdapter: Here, the fragment instance will be destroyed when it is not visible to the user, except the saved state of the fragment.

What is the difference between adding/replacing fragment in backstack?

The important difference is:

replace: removes the existing fragment and adds a new fragment.

but add retains the existing fragments and adds a new fragment that means existing fragment will be active and they won't 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).

Why it is recommended to use only default constructor to create a Fragment?

Whenever the android fragment decides to recreate our fragment for example in case of Orientation changes, Andorid calls the no-argument constructor of our fragment.

The reason behind why can't it called the constructor with argument is that Android fragment has no idea what constructor we have created so it can't.

Example:

In case of orientation changes the android framework recreates the new fragment using the no-argument constructor and attach the bundle to the fragment as it has stored the bundle earlier and later again, we can access that data in our onCreate () method by using getArgument like this.

So, it means that when the system restores a fragment it will automatically restore our bundle and we will be able to restore the state of the fragment to the same state, the fragment was initialized with.

What is retained Fragment?

By default, Fragments are destroyed and recreated along with their parent Activity's 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.

What is the purpose of addToBackStack () while committing fragment transaction?

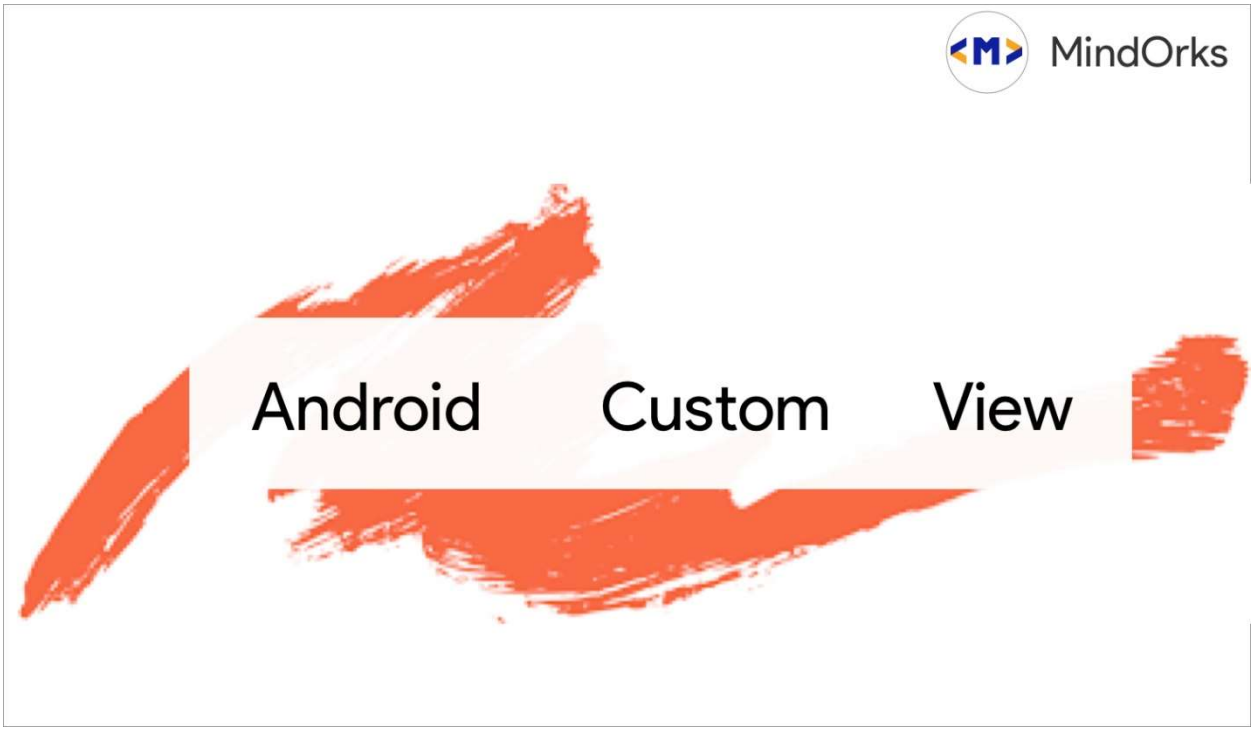
By calling addToBackStack (), the replace transaction is saved to the back stack so the user can reverse the transaction and bring back the previous fragment by pressing the Back button.

Difference between View.GONE and View.INVISIBLE?

`View.INVISIBLE`: This view is invisible, but it still takes up space for layout purposes.

`View.GONE`: This view is invisible, and it doesn't take any space for layout purposes.

Can you create a custom view? How?



When you need to create some custom and reuse the views when it is not provided by the Android Ecosystem. Custom views can be used as widgets like TextView, EditText etc.

Views can be of few types:

`Custom View`: Where we draw everything

`Custom View Groups`: Where we use existing widgets to form an inflatable xml file.

Why should we use Custom Views?

1. To make the UI component reusable.
2. Add new interaction which is not provided by the Android Ecosystem.

How does Android Draw the UI?

`onMeasure ()` -> `onLayout ()` -> `onDraw ()`

here,

1. At first `onMeasure ()` gets called where Android measures the UI from top to bottom. First, it takes the parents container, their children and so on. In `onMeasure ()` children get the constraints provided by their parents.

2. Then, `onLayout ()` is called to plot the positions of the Widgets
3. And finally, in `onDraw ()` the UI gets rendered.

Steps to create a Custom View

1. By Extending an Existing Widget Class (eg. Button, TextView)
2. By Extending a custom view, we will create a class

To start with a custom view, we will create a class

```
Class MyCustomView: View
```

And to start with drawing your views we need to override at least one **constructor**.

```
Class MyCustomView (context: Context): View (context)
```

The constructor needs the basic activity context to draw

```
Class MyCustomView (context: Context, attrs: AttributeSet): View (context, attrs)
```

The constructor creates a new view instance from XML

```
Class MyCustomView (context: Context, attrs: AttributeSet, defstyleAttr: Int): View (context, attrs, defstyleAttr)
```

The constructor also takes the design attribute

```
Class MyCustomView (context: Context, attrs: AttributeSet, defstyleAttr: Int, defStyleRes: Int): View (context, attrs, defstyleAttr, defStyleRes)
```

This constructor takes theme attribute as well and we generally use,

```
Class MyCustomView (context: Context, attrs: AttributeSet): View (context, attrs)
```

To start with and to use the view we have to add in the XML,

```
<com.yourpackagename.MyCustomView
  android:id="@+id/customview"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:layout_centerInParent="true"/>
```

We have successfully created a custom view without doing anything on the UI. Now, to draw something on the UI, we need to override the `onDraw ()` method.

```
override fun onDraw (canvas: Canvas) {
  // call the super method to keep any drawing from the parent side.
  super.onDraw(canvas)
}
```

And all drawing of UI goes here.

1. As we need paint to draw in real life in Custom View Drawing, we need paint as well (Paint Object).
2. A mobile screen in android should be considered as a big canvas where things are drawn based on coordinates and point system.

What are ViewGroups and how they are different from the Views?

View: View objects are the basic building blocks of User Interface elements in Android. View is a simple rectangle box which responds to 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.

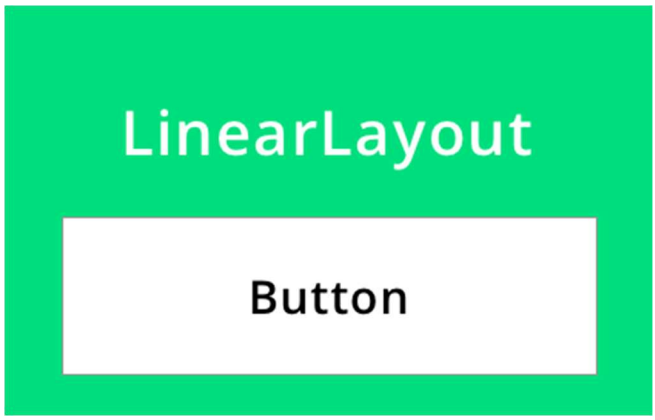
ViewGroup: ViewGroup is the invisible container. It holds View and ViewGroup. For Example. LinearLayout is the ViewGroup that contains Button (View), and other Layouts also. ViewGroup is the base class for Layouts.

How does the Touch Control and Events work in Android?

Almost all phones nowadays running Android are touch-controlled. There are very few phones which are not touch-based.

So, how do the input events actually work and what exactly happens when we touch our screen when have a ViewGroup having different views inside it?

In this case, we are going to talk a case where we have a LinearLayout containing a button like,

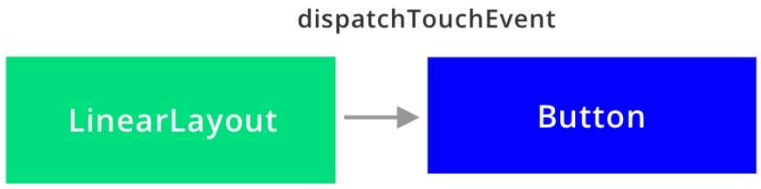


What happens when we touch the screen?

So, when we touch the screen the activity's view gets the touch event notification first also known as `DecorView` in android. Now, we generally don't work with the touch of `DecorView`. So, the touch gets transferred to the ViewGroup and subsequently to its children in the XML file.

But how can we transfer the touch event trigger?

In Android, the ViewGroup transfers the touch event from top to bottom in the hierarchy of ViewGroup to its children using `dispatchTouchEvent ()`.



How do we intercept touch events?

First when we perform a touch action,

The ViewGroup gets the touch event, and then it is intercepted in the ViewGroup itself using `onInterceptTouchEvent ()`.

If on intercepting if we return `true` then the touch event is **not** passed to its children and if we pass `false`, the android Ecosystem gets notified that the ViewGroup wants to dispatch the event to its children, in our case it is a button.

In general, if we returning true, it means we have handled the event in the ViewGroup itself, no need to dispatch to its children.

Now, as the button we have, is the last view in our tree. So, it won't be able to pass the touch event to its children anymore as it has none. So, in button, we would have our last `onInterceptTouchEvent` being called.

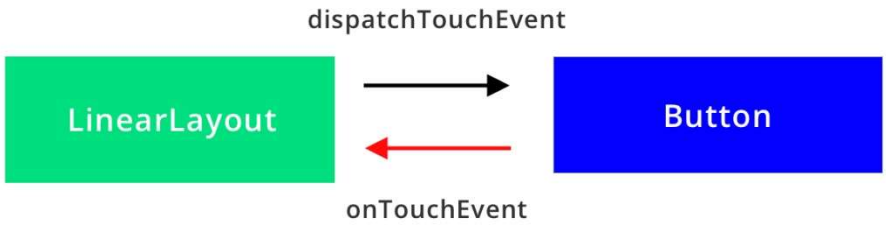
Intercepting of the event can only happen in ViewGroups and not Views.

How touch is handled in the view?

When dispatching the touch event from top to bottom in the view hierarchy, we need to see at which position of the tree we need to handle the touch on the view.

When handling the dispatching the event, the top position of the hierarchy takes the lead, but when it comes to handling the touch the child views using `onTouchEvent` are always the first, and it keeps moving towards the ViewGroups.

Touch event works just like the dispatching of the events but in the **reverse** order from child to parent.



Let's say if we dispatch the event from ViewGroup and intercept the event there, it depends on the return value (true/false) that shall the touch of the view be handled on the ViewGroup or the children.

So, in our case, if `onTouchEvent` of `Button` returns **true**, then it means that it has been handled and then, it will **not** go to the `LinearLayout`.

[What are the touch events we majorly work within Android for handling the touch control?]

When we get the touch event, it gets handled by `onTouchEvent` which also has a parameter of type `MotionEvent`.

```
Fun onTouchEvent (event: MotionEvent)
```

All the task perform regarding the touch has its reference in the event parameter. We can have the coordinates like X and Y points on the screen of the point on the touch.

It even has the actions in it, like let's see if we tap on the screen then `MotionEvent.ACTION_DOWN` is called and when we lift the touch `MotionEvent.ACTION_UP` is called.

Even dragging a finger on the screen, the actions is `MotionEvent.ACTION_MOVE`.

So, the flow on the view happens is when we want to tap the button,

```
Activity -> dispatchTouchEvent (LinearLayout) -> dispatchTouchEvent (Button) -> onTouchEvent (Button)
```

And when we don't want to tap the button but want to handle the click on `LinearLayout`, the flow would be,

```
Activity -> dispatchTouchEvent (LinearLayout) -> dispatchTouchEvent (Button) -> onTouchEvent (Button) (will return false) -> onTouchEvent (LinearLayout)
```

What is the difference between ListView and RecyclerView?

`RecyclerView` was created as a `ListView` improvement, so yes, you can create an attached list with `ListView` control, but using `RecyclerView` is easier as it:

1. Reuses cells while scrolling up/down: This is possible with implementing View Holder in the `ListView` adapter, but it was an optional thing, while in the `RecyclerView` it's the default way of writing adapter.
2. Decouples list from its container: So, you can put list items easily at run time in the different containers (LinearLayout, GridLayout) with setting `LayoutManager`.
3. Animates common list actions: Animations are decoupled and delegated to `ItemAnimator`.

So, `RecyclerView` is a more flexible control for handling "list data" that follows patterns of delegation of concerns and leaves for itself only one task – recycling items.

How does RecyclerView work internally?

RecyclerView is a ViewGroup, which populates a list on a collection of data provided with the help of ViewHolder and draws it to the user on-screen.

Building components of RecyclerView

The major components of RecyclerView are,

- Adapter
- ViewHolder
- LayoutManager

Adapter

It is a subtype of `RecyclerView.Adapter` class. It takes the data set which has to be displayed to the user in RecyclerView. It is like the main responsible class to bind the views and display it.

Most of the tasks happen inside the adapter class of the RecyclerView.

ViewHolder

ViewHolder is a type of helper class that helps us to draw the UI for individual items that we want to draw on the screen.

All the binding of Views of the individual items happens in this class. It is a subclass of `RecyclerView.ViewHolder` class.

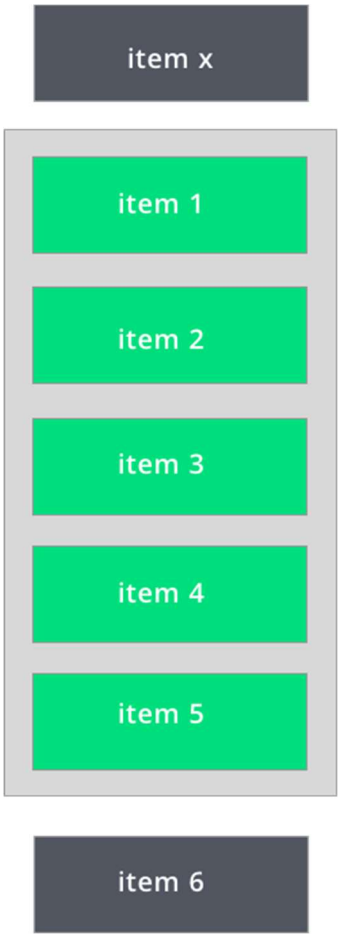
LayoutManager

LayoutManager in RecyclerView helps us to figure out how we need to display the items on the screen. It can be linearly or in a grid. RecyclerView provides by default a few implementations of LayoutManager out of the box.

It is like the governing body of RecyclerView which tells the RecyclerView's adapter when to create a new view.

How it actually works?

When we are scrolling the list, which has 50 items in the collection and we show only 5 items at once in the list like,



Here. **Item 1** to **item 5** is the ones that are visible on the screen. **Item x** is the one that will be loaded next on the screen when we scroll up.

All the items here have their own instance of ViewHolder and the ViewHolder here is helpful for caching the specific item's view.

So, how recycling works here?

Step 01

First **item x** to **item 4** must be shown to screen at the initial launch. So, they are the five items that are in the visible view mode. Let's call them a **visible view**.

And, **item 5** is the new item to be loaded when the list is scrolled up.

Step 02

When we scroll one item above, **item x** moves up and a new item, item 5 comes in the visible view category.

Now, **item 6** is in the waiting view.

Here **item x** has moved out from the visible view on top of the screen. This is called a **scrapped view (**ScrapView is the view in RecyclerView which was once visible and now are not visible on the phone's screen to the user**).**

Step 03

Now, let's say we scrolled one more step up. This will move the **item 1** out of the screen and the **item 6** will move in.

Here, **item 1** also becomes a scrapped view. Now, we have two scrapped views **item x** and **item 1**. They will be now stored in a collection of **scrapped views**.

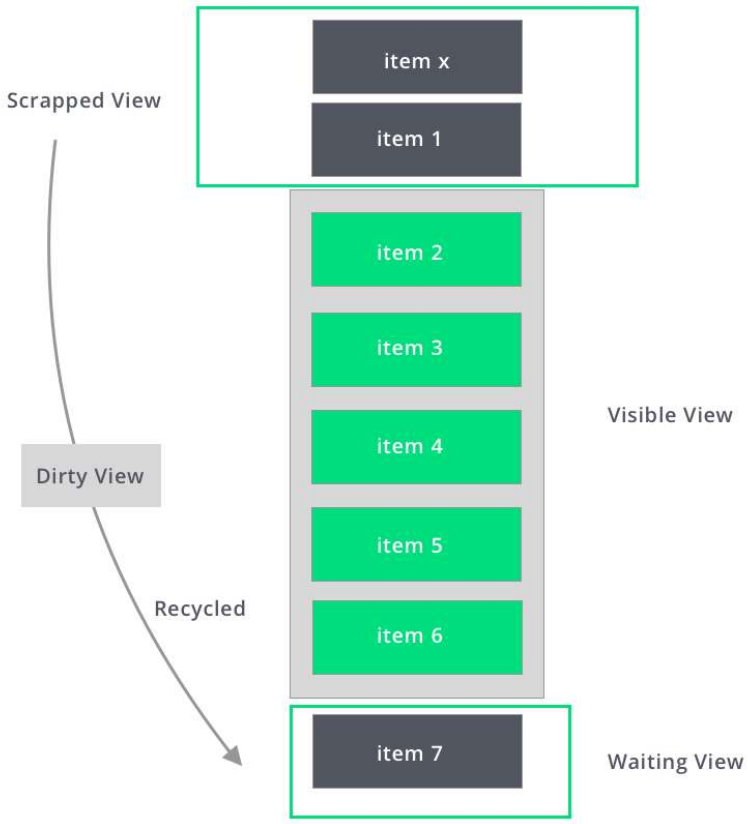
So, now when we have to load a new view in the visible view group, let's consider **item 7**, then the view from the collection from the scrapped view is used.

Step 04

Now, when we are loading **item 7**, we take a view from the collection of scrapped views. The view which we loaded from the scrapped view is called a **dirty view**.

Now, the dirty view gets recycled and is relocated to the new item in the queue which has to be displayed on the screen i.e, **item 7**.

The view which we take from scrap view collection and then after rebound happens by the RecyclerView adapter before it is drawn to the screen are called dirty views.



This is how recycling of views happens in RecyclerView which is one of the main reasons for the better improvement of the RecyclerView. In this process, the views of the item are reused to draw new items on the screen.

Usage of ViewHolder

The other way the views are optimized is because of ViewHolders in RecyclerView.

So, let us say we have 100 items be displayed on the list and we want to show 5 items on the screen.

Each item here has 1 TextView and 1 ImageView in the item.

So, to map each view using **findViewById** is always an expensive task and jut imagine the number of **findViewByIds** we might need to map all the TextViews and ImageViews of the 100 items i.e, 200 findViewByIds.

So, when we are using RecyclerView, then only 6 items are created initially with 5 loaded at once to be displayed on screen and one is the one to be loaded.

Now, if we scroll the list then we have 7 ViewHolders. One each for scrapped view and to be loaded view and 5 for the ones which are displayed.

So, at a time, maximum **findViewByIds** that we are using are only 14 as we have a maximum of 7 ViewHolders.

RecyclerView Optimization: Scrolling Performance Improvement

When we implement RecyclerView in our Android application. Sometimes, we face problems like: The RecyclerView items are not scrolling smoothly. It leads to bad user experience as it seems that our Android app is laggy.

RecyclerView Optimization Techniques

Use Image Loading Library

As the Garbage Collection (GC) runs on the main thread, one of the reasons for unresponsive UI is **the continuous allocation and deallocation of memory**, which leads to the very frequent GC run. By using the bitmap pool concept, we can avoid it.

The best part is that Image Loading libraries like Glide, Fresco uses this bitmap pool concept. So, always use Image Loading libraries. Delegate all image related tasks to these libraries.

Set Image Width and Height

If our image width and height are dynamic (not fixed), and we are getting the `imageUrl` from the server.

If we do not set the correct image width and height prior, the UI is **flicker** during the transition of loading (downloading of the image) and setting of the image into the ImageView (actually making it visible when downloading completes).

So, we should ask our backend developer to send the image size or the aspect ratio, accordingly, we can calculate the required width and height of the ImageView.

Then, we will be able to set the width and height prior only. Hence no flickering. Problem Solved!

Do less in onBindViewHolder method

Our **onBindViewHolder** method should do very less work. We should check our onBindViewHolder method and optimize it. We can see the improvement in our RecyclerView by doing so.

Use Notify Item RecyclerView API

Whenever we have the use-case of the removal, the updation, the addition of the item, use the Notify Item API.

```
adapter.notifyItemRemoved(position)
adapter.notifyItemChanged(position)
adapter.notifyItemInserted(position)
adapter.notifyItemRangeInserted(start, end)
```

In case, we are forced to use `notifyDataSetChanged ()` based on our use-case, we can try the `setHasStableIds (true)`.

```
adapter.setHasStableIds(true)
```

It indicates whether each item in the data set can be represented with a unique identifier of type **Long**.

Even if we call the `notifyDataSetChanged ()`, it does not have to handle the complete reordering of the whole adapter because it can find if the item at a position is the same as before and do less work.

Avoid a nested view

If possible, we should avoid a nested view and try to create a flat view whenever possible. Nesting reduces RecyclerView performance. Flat view improves performance.

Use setHasFixedSize

We should use this method if the height of all the items is equal. Add the below and check the performance.

```
recyclerView.setHasFixedSize(true)
```

Use setRecycledViewPool for Optimizing Nested RecyclerView

As we know that the RecyclerView works on the principle of "Reuse View Using Pool" wherever possible.

If we have the use-case of Nested RecyclerView.

```
- OuterRecyclerView
   - InnerRecyclerViewOne
   - InnerRecyclerViewTwo
```

But by default, the optimization works for that particular RecyclerView because that particular RecyclerView has its own View Pool.

The pool does not get shared between two RecyclerViews having the same types of views.

So, what we can do is that we can create a single ViewPool and pass it to all inner RecyclerViews so that it gets shared like below:

```
class OuterRecyclerViewAdapter() : RecyclerView.Adapter<RecyclerView.ViewHolder>() {

   // code removed for brevity

   private val viewPool = RecyclerView.RecycledViewPool()

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {

      // code removed for brevity

      viewHolderOne.innerRecyclerViewOne.setRecycledViewPool(viewPool)

      // code removed for brevity

      viewHolderTwo.innerRecyclerViewTwo.setRecycledViewPool(viewPool)

   }

   // code removed for brevity}
```

This will improve scrolling performance as it will start reusing the view from the shared ViewPool.

Use setItemViewCacheSize

We can **experiment** by setting the ItemView Cache Size.

```
recyclerView.setItemViewCacheSize(cacheSize)
```

As per the official documentation: It sets the number of offscreen views to retain before adding them to the potentially shared recycled viewpool. The offscreen view cache stays aware of changes in the attached adapter, allowing a LayoutManager to reuse those views unmodified without needing to return to the adapter to re-bind them.

It means that when we scroll the RecyclerView such that there's a view that is just barely completely off screen, the RecyclerView will keep it around o that we can scroll it back into view without having to call the `onBindViewHolder ()` again.

Generally, we do not change the size of the View Cache, but experiment with it, if it works for you, implement it.

These are the things which we can do to improve the performance of RecyclerView.

What is the difference between Dialog and Dialog Fragment?

A **dialog** is a UI that is displayed in a window of its own. The only difference between a Fragment and a dialog is that the **Fragment Dialog** displays its UI in the View hierarchy of the Activity, i.e, in the Activity's window.

What is a LocalBroadcastManager?

In android, we use Broadcast Receiver to send a particular message to every application that is associated with that specific broadcast. It is the same as that of YouTube channel subscription. If you have subscribed to a YouTube channel, then whenever the creator of the channel will upload some video, you will be notified about the same. So, in the same way, Android applications can subscribe or register to certain system events like battery low and in in return, these applications will get notified whenever the subscribed event will occur. But these are Global Broadcasts and should not be used in every case because it has some demerits.

How to use Broadcast?

The whole process of using Broadcast can be divided into two parts:

1. Register Broadcast
2. Receive Broadcast

Register Broadcast

Firstly, you have to register for a broadcast in your application. For example, if you want to receive the Battery low event, then you have to register this in your application. So, to register a particular broadcast in your application, you have two options, either you can register the event in the **AndroidManifest.xml** file of your application or you can register the broadcast via the `Context.registerReceiver ()` method.

Receiver Broadcast

By extending the BroadcastReceiver abstract class, you can receive the broadcast in your application. After extending, all you need to do is override the `onReceive ()` method and perform the required action in the `onReceive ()` method because this method will be called when a particular broadcast is received.

LocalBroadcastManager

If the communication is not between different applications on the Android device, then is it suggested not to use the Global BroadcastManager because there can be some security holes while using Global BroadcastManager and you don't have to worry about this if you are using LocalBroadcastManager. LocalBroadcastManager is used to register and send a broadcast of intents to local objects in your process. It has lots of advantages:

1. Your broadcasting data will not leave your app. So, if there is some leakage in your app then you need worry about that.
2. Another thing that can be noted here is that other applications can't send any kind of broadcasts to your app. So, you need not worry about security holes.

How to use LocalBroadcastManager?

To use LocalBroadcastReceiver, all you need to do is create an instance of LocalBroadcastManager, then send the broadcast and finally receive the broadcast. So, firstly, create an instance of the LocalbroadcastManager:

```
var localBroadcastManager = LocalBroadcastManager.getInstance(context)
```

now, by using the `sendBroadcast ()` method, you can send the broadcast as below:

```
val localIntent = Intent("YOUR_ACTION")
    .putExtra ("DATA", "MindOrks")
localBroadcastManager.sendBroadcast(localIntent)
```

our final task is to receive the broadcast using the `onReceive ()` method on **MyBroadCastReceiver**. You can perform the desired action after receiving the broadcast in the `onReceive ()` method as below:

```
private val listener = MyBroadcastReceiver()

inner class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent){
        when (intent.action) {
            "YOUR_ACTION" -> {
                val data = intent.getStringExtra("DATA")
                Log.d("Your Received data : ", data)
            }
            else -> Toast.makeText(context, "Action Not Found", Toast.LENGTH_LONG).show()
        }

    }
}
```

Now, this will print an output in Logcat,

```
Your Received data: MindOrks
```

So, these are the three basic steps that can be done to use LocalBroadcastManager in our Android application.

Here, you can perform multiple action in `onReceive ()` of the broadcast receiver based on different actions.

If you want some kind of broadcasting in your application, then you should use the concept of **LocalBroadcastManager,** and we should avoid using the **Global Broadcast** because for using Global Broadcast you have to ensure that there are no security holes that can leak your data to other applications.

What is a PendingIntent?

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

Is it possible to run an Android app in multiple processes? How?

You can specify `android: process=": remote"` in your Manifest to have an activity/service run in a separate process.

The "**remote**" is just the name of the remote process, and you can call it whatever you want. If you want several activities/services to run in the same process, just give it the same name.

```
<activity android:name=".RemoteActivity" android:label="@string/app_name"
android:process=":RemoteActivityProcess"/>
```

What is the difference between a regular Bitmap and nine-patch image?

In general, a Nine-patch image allows resizing that can be used as background or other image size requirements for the target device. The Nine-patch refers to the way you can resize the image: 4 corners that are unscaled, 4 edges that are scaled in 1 axis, and the middle one that can be scaled into both axes.

How to optimize android apps for better performance?

Tools and Tips for Optimizing android apps:

1. Use StringBuidler instead of String
2. Picking the Correct Data type
3. Network Requests
4. Location updates
5. Reflection
6. Autoboxing
7. onDraw
8. ViewHolders
9. Resizing Images
10. Strict Mode

How to generate Signed apk in android studio?

Signed apk generates a key and this key can be used to release versions of the app, so it is important to save this key which will be used when the next version of the app is released.

The android system needs that all installed applications be digitally signed with a certificate whose private key is owned by the application's developer. The android system applies the certificate as a means of recognizing the author of an application and establishing trust relations between applications. The essentials point to understand about signing android applications are:

- When developers are ready to publish the android application for end users, they are required to sign it with a suitable private key. They can't publish an application that is signed with the debug key generated by the SDK tools.
- Applications can only be installed when they are signed. Android does not allow unsigned applications to get installed.
- Developers can apply self-signed certificates to sign the application. No certificate authority is required.
- To test and debug the application, the build tools sign the application with a special debug key that is created by the android SDK build tools.
- The system will test a signer certificate's expiration date only at install time. If an application's signer license expires after the application is installed, the application will continue to operate normally.

Importance of Signed apk

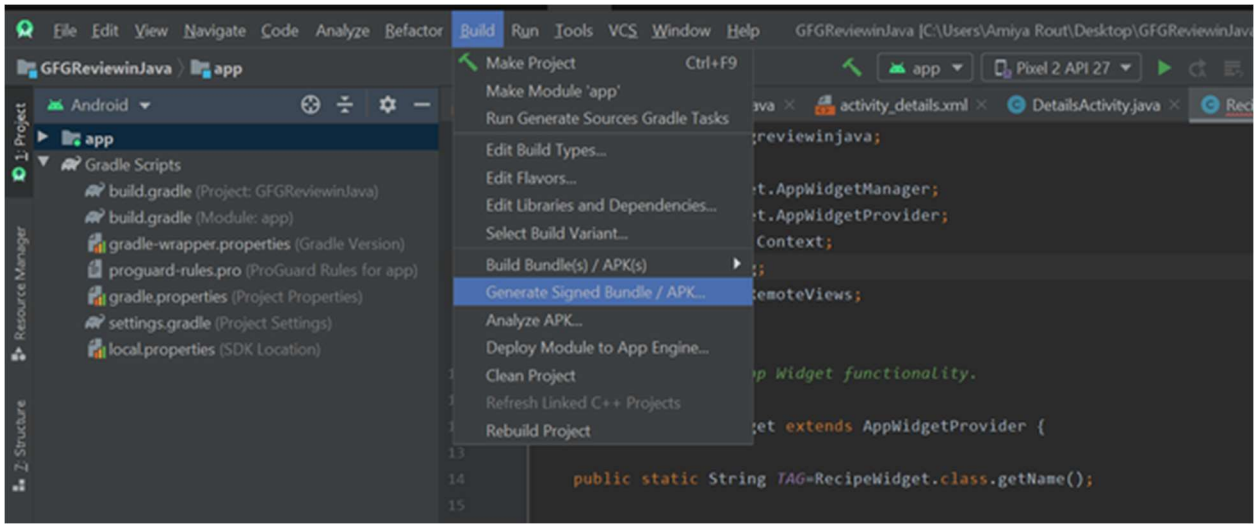
- Application Modularity: The applications signed by the same process are recognized as a single application and are allowed to run on the same process. This allows the developers to make the application in modules and so user can update each module independently.

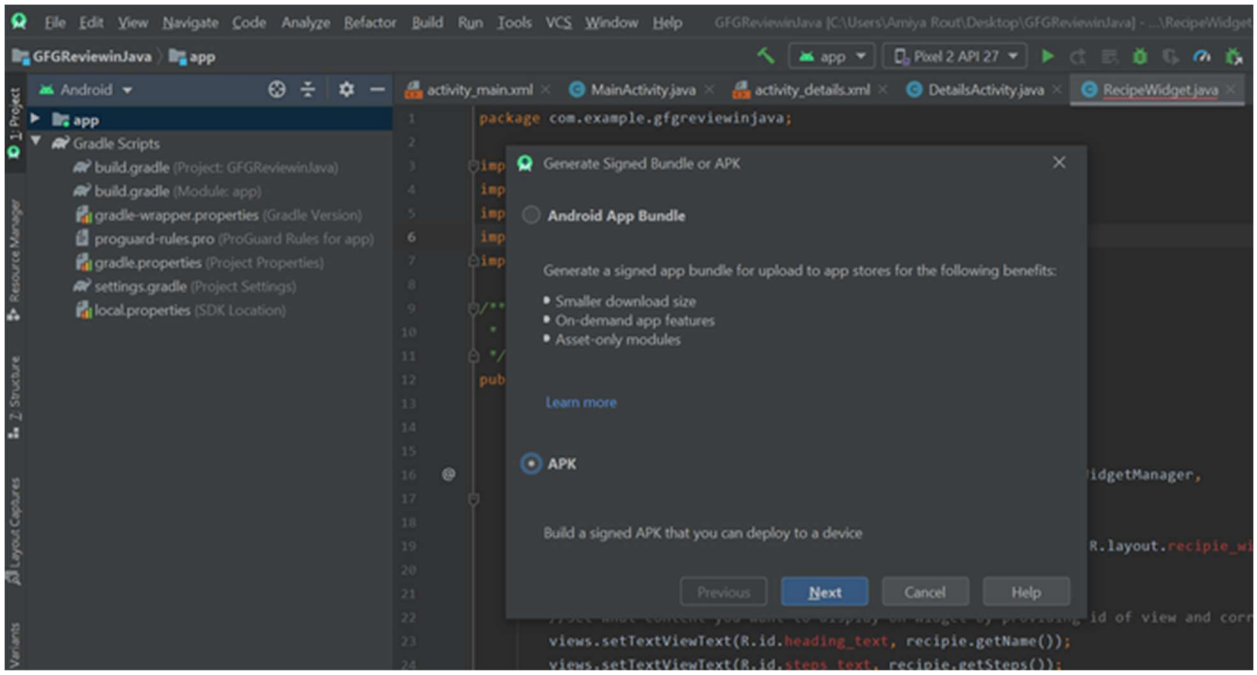
- Application Upgrade: To update the application, the updates must be signed with the same certificates. When the system is installing an update to an application, it relates the certificate in the new version with those in the actual version. If the certificates match correctly, including both the certificate data and order, then the system releases the update. If we sign the new version without using matching certificates, we need to attach a different package name to the application and in this case, the user installs the new version as a completely new application.

- Code/Data Sharing Through Permissions: To allow the applications to use different resources, android system executes **signature-based permissions** enforcement so that the application can exhibit functionality to another application can exhibit functionality to another application which is signed with a specified certificate. By signing various applications with the same certificates and working with signature-based permission checks, your applications can yield code and data in secure manner.

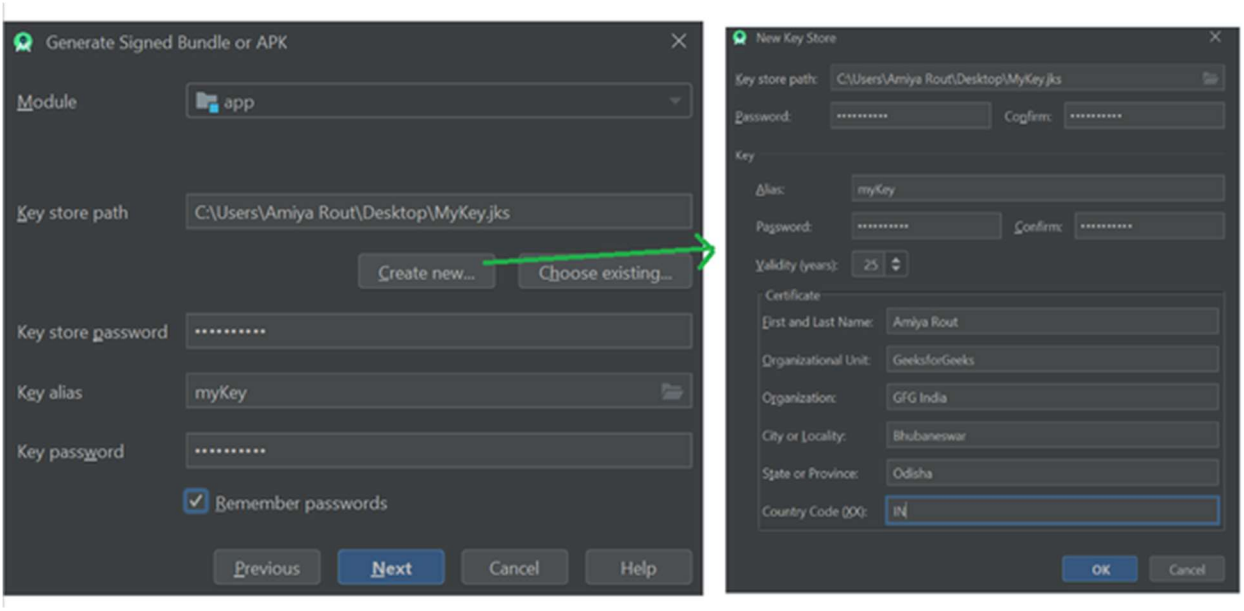
Generating Signed apk in android studio

Step 1: Go to **Build -> Generate Signed Bundle or APK**, a pop up will arise. Choose **APK** in the pop up and click on **Next**.

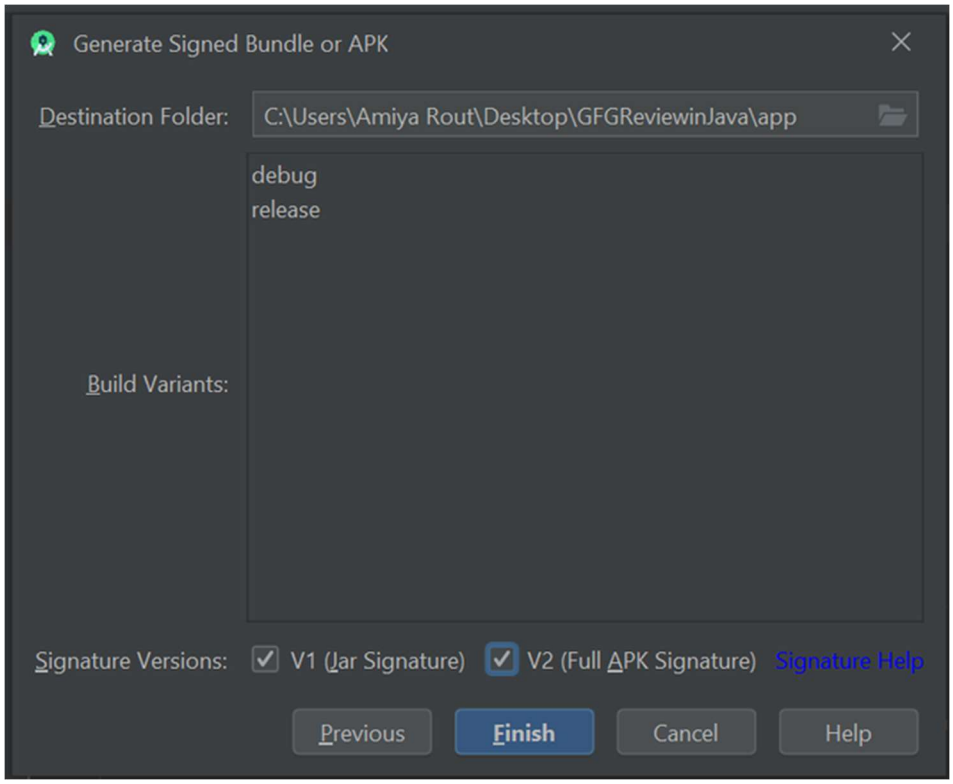




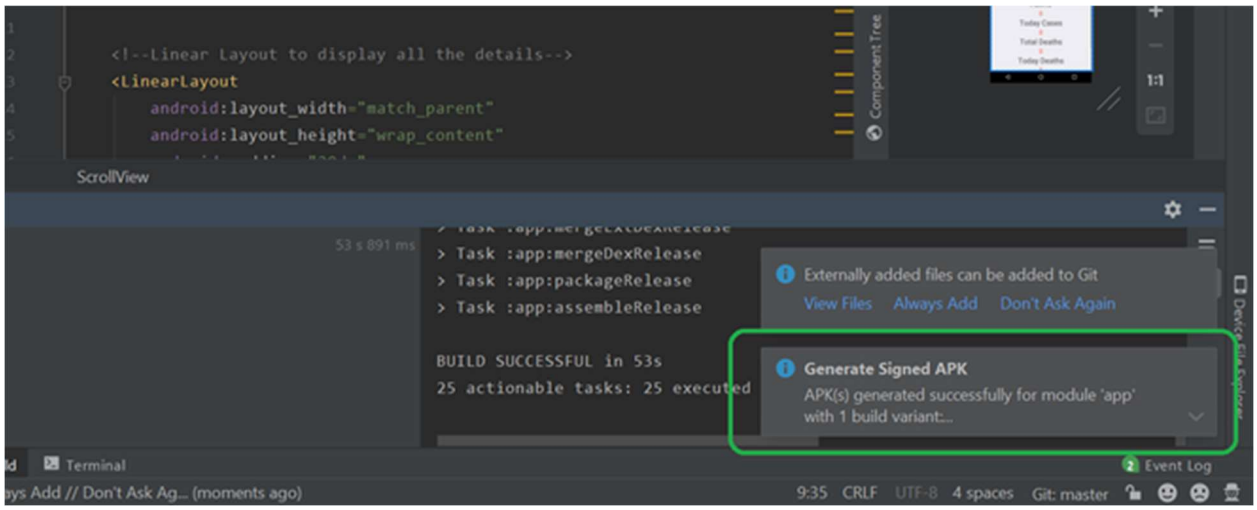
Step 2: After completing step 1, if you already have a key, make sure you just refer to that and you can release the version of that app but in case if it is the first time it is recommended to create a new key. After creating a new key click on **OK** and then click on **Next.**



Step 3: After clicking Next in the next pop up make sure to choose **release** as **Build Variants** and check the two **Signature Versions** as well. Then click on Finish.



Step 4: After successfully completed these steps you can locate your signed apk at **app** -> **release** -> **app-release** as your apk file.



JSON

JSON stands for JavaScript Object Notation.

A JSON file type is .json

JSON objects are used for transfering data between server and client, XML serves the same same purpose. However JSON objects have several advantages over XML.

Let's have a look at the piece of a JSON data: It basically has key-value pairs.

```
var chaitanya = {
   "firstName" : "Chaitanya",
   "lastName" : "Singh",
   "age" :  "28"
};
```

Features of JSON:

- It is light-weight
- It is language independent
- Easy to read and write
- Text based, human readable data exchange format

Why use JSON?

Standard Structure: As we have seen so far that JSON objects are having a standard structure that makes developers job easy to read and write code, because they know what to expect from JSON.

Light weight: When working with AJAX, it is important to load the data quickly and asynchronously without requesting the page re-load. Since JSON is light weighted, it becomes easier to get and load the requested data quickly.

Scalable: JSON is language independent, which means it can work well with most of the modern programming language. Let's say if we need to change the server-side language, in that case it would be easier for us to go ahead with that change as JSON structure is same for all the languages.

JSON vs. XML

Let's see how JSON and XML look when we store the records of 4 students in a text-based format so that we can retrieve it later when required.

JSON style:

```
{"students":[
   {"name":"John", "age":"23", "city":"Agra"},
   {"name":"Steve", "age":"28", "city":"Delhi"},
   {"name":"Peter", "age":"32", "city":"Chennai"},
   {"name":"Chaitanya", "age":"28", "city":"Bangalore"}
]}
```

XML style:

```
<students>
 <student>
  <name>John</name> <age>23</age> <city>Agra</city>
 </student>
 <student>
  <name>Steve</name> <age>28</age> <city>Delhi</city>
 </student>
 <student>
  <name>Peter</name> <age>32</age> <city>Chennai</city>
 </student>
 <student>
  <name>Chaitanya</name> <age>28</age> <city>Bangalore</city>
 </student>
</students>
```

As you can clearly see JSON is much more light-weight compared to XML. Also, in JSON we take advantage of arrays that is not available in XML.

JSON data structure types and how to read them:

1. JSON objects
2. JSON objects in array
3. Nesting of JSON objects

JSON objects

```
var chaitanya = {
  "name" : "Chaitanya Singh",
  "age" : "28",
  "website" : "beginnersbook"
};
```

The above text creates an object that we can access using the variable chaitanya. Inside an object we can have anu number of key-value pairs like we have above. We can access the information out of a JSON object like this:

```
document.writeln("The name is:  " +chaitanya.name);
document.writeln("his age is: " + chaitanya.age);
document.writeln("his website is: "+ chaitanya.website);
```

JSON objects in array

In the above example we have stored the information of one person in a JSON object suppose we want to store the information of more than one person; in that case we can have an array of objects.

```
var students = [{
   "name" : "Steve",
   "age" :  "29",
   "gender" : "male"

},
{
   "name" : "Peter",
   "age" : "32",
   "gender" : "male"

},
{
   "name" : "Sophie",
   "age" : "27",
   "gender" : "female"
}];
```

To access the information out of this array, we do write the code like this:

```
document.writeln(students[0].age); //output would be: 29
document.writeln(students[2].name); //output: Sophie
```

Nesting of JSON objects:

Another way of doing the same thing that we have done above.

```
var students = {
  "steve" : {
  "name" : "Steve",
  "age" :  "29",
  "gender" : "male"
},

"pete" : {
  "name" : "Peter",
  "age" : "32",
  "gender" : "male"
},

"sop" : {
  "name" : "Sophie",
  "age" : "27",
  "gender" : "female"
}
}
```

Do like this to access the info above nested JSON objects:

```
document.writln(students.steve.age); //output: 29
document.writeln(students.sop.gender); //output: female
```

JSON & JavaScript

JSON is considered as a subset of JavaScript but that does not mean that JSON cannot be used with other languages. In fact, it works well with PHP, Perl, Python and many more.

Just to demonstrate how JSON can be used along with JavaScript, here is an example:

How to read data from json file and convert it into a JavaScript object?

We have two ways to do this.

1. Using eval function, but this is not suggested due to security reasons (malicious data can be sent from the server to the client and then eval in the client script with harmful effects).
2. Using JSON parser: No security issues plus it is faster than eval. Here is how to use it:

```
var chaitanya = {
"name" : "Chaitanya Singh",
"age" : "28",
"website" : "beginnersbook"
};
```

We are converting the above JSON object to JavaScript object using JSON parser:

```
var myJSObject = JSON.parse(chaitanya);
```

How to convert JavaScript object to JSON text?

By using method stringify

```
var jsonText= JSON.stringify(myJSObject);
```

Different Versions of Android

Android 1.0

Android 1.0 is the first iteration of Android with an application programming interface 1 or API 1. It was released on September 23, 2008 and officially there was no code name for this version but according to some report, it was known as Android Alpha.

User Features:

- Download and updates via Android Market
- Web Browser
- Camera support
- Gmail, contacts and Google Agenda Synchronization
- Google Maps
- YouTube Application

Android 1.1

Android 1.1 also didn't get any official code name and as per reports, it was known as Android Beta, released on February 9, 2009, with API 2. Android 1.1 introduced features like support for third-party keyboards, video recording, video playback and copy-paste features for the web browsers.

User Features:

- Third-party keyboards support
- Video recording
- Video playback
- Copy paste features for the web browsers
- Show & Hide numeric keyboard. In caller application
- Ability to save MMS attachments

Android 1.5 Cupcake

Android 1.5 was released on April 27, 2009 and it is the first Android version to officially get a dessert nickname. Android 1.5 was known as Android Cupcake and it was developed by Google with API 3.

User Features:

- Bluetooth A2DP, AVRCP support
- Soft-keyboard with text-prediction
- Record/watch videos
- Support for Widgets
- User pictures shown for Favorites in contacts
- Animated screen transitions
- Ability to upload videos to YouTube
- Ability to upload photos to Picasa

Android 1.6 Donut

Android 1.6 Donut was released on September 15, 2009, just a few months after the launch of the Android 1.5 Cupcake and it came with the API 4. This was also called the fourth version of the Android and it introduced some distinctive features.

User Features:

- Gesture Framework
- Turn-by-turn navigation
- Support for CDMA technology
- Support for different sizes
- Battery usage indicator

Android 2.0 Eclair

Android 2.0 Éclair came with an API level of 5 to 7 and it was released on October 26, 2009, again, a fourth major Android update to get released in 2009. The fifth version of Android aka Android 2.0 Éclair offered many features.

User Features:

- Improved UI with a Google search bar on the top
- Supported HTML5
- Digital zoom
- Microsoft Exchange support
- Bluetooth 2.1
- Live wallpapers
- Updated UI

Next major OS updates:

- Android 2.0.1 Éclair with API 6
- Android 2.1 Éclair with API 7

Android 2.2 Froyo

Android 2.2 Froyo was made available to the public on May 20, 2010, and it came with API 8. This version was unveiled during Google I/O 2010 conference and it offered several optimizations when compared to Android 2.0 Eclairs.

User Features:

- Speed improvements
- JIT implementation
- USB Tethering
- Applications installation to the expandable memory
- Upload file support in the browser
- Animated GIFs

Android 2.3 Gingerbread

Android 2.3 Gingerbread was also launched in 2010, to be exact on December 6, 2010, with API 9 to 10. This version introduced features like NFC support and VoIP calls. The Nexus S smartphone from Samsung was the first device to launch with this OS, which was also the first Nexus smartphone from Google that represented stock Android OS.

User Features:

- Updated UI
- Improved keyboard ease of use
- Improved copy/paste
- Improved power management
- Social networking features
- Near Field Communication support
- Native VoIP/SIP support
- Video call support

Developer Features:

- Performance – concurrent garbage collection, faster event distribution, updated video drivers
- NDK – Native Asset Manager, Native Activities + event handling
- Audio effects API
- VP8, WebM, AAC wideband
- Multiple camera sensor support
- Strictmode Debugging

Next major OS updates:

Android 2.3.3 Gingerbread API 10

Developer Features:

- NFC API improvements (peer to peer communication)
- Added unsecure Bluetooth sockets

Android 2.3.4 Gingerbread API 10

User Features:

- Voice or video chat using Google Talk

Developer Features:

- Open Accessory API

Android 2.3.5 Gingerbread API 10

User Features:

- Improved network performance for the Nexus S 4G
- Fixed Bluetooth issues on the Samsung Galaxy S
- Gmail app improvements

Android 2.3.6 Gingerbread API 10

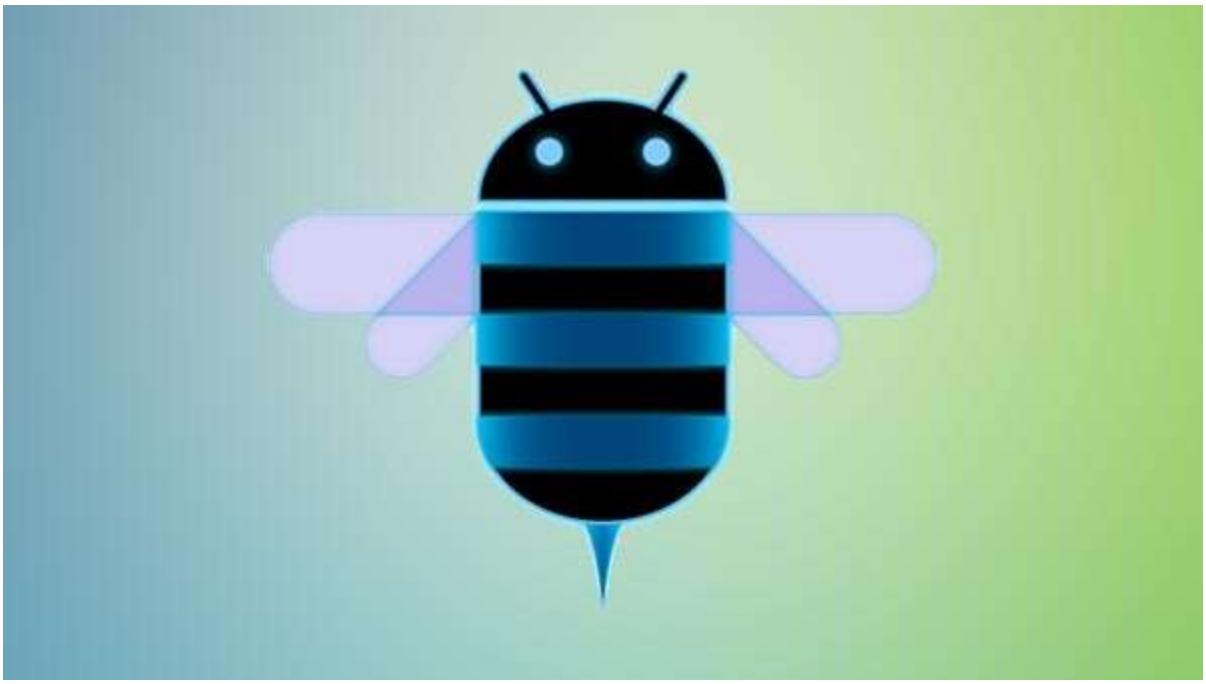
User Features:

- Voice search issue fixed

Android 2.3.7 Gingerbread API 10

User Features:

- Google Wallet support for the Nexus S 4G



Android 3.0 Honeycomb

Android 3.0 Honeycomb was released on February 22, 2011 and it came with API 11 to 13. This version of Android was designed for large screen devices like tablets and the Motorola Xoom was the first product to ship with Android 3.0 Honeycomb.

User Features:

- Multi core support
- Better tablet support
- Updated 3D UI (customizable home screens, recent application viewing)
- Media/Picture transport protocol
- Google eBooks
- Private browsing
- HTTP Live streaming
- System wide Clipboard

Developer Features:

- Contextual action bar
- Fragments first introduce
- Device administration
- High performance Animation Framework
- Forced rendering of layers
- LRU cache
- High performance WIFI lock
- RTP streaming API

Next major OS updates:

Android 3.1 Honeycomb API 12

User Features:

- UI improvements
- Open Accessory API
- USB host API
- Mice, joysticks, gamepads support
- Resizable Home screen widgets
- MTP notifications
- RTP API for audio

Android 3.2 Honeycomb API 13

User Features:

- Optimization for a wider range of tablets
- Compatibility display mode
- Media sync from SD card

Developer Features:

- Extended API for managing screens support

Android 3.2.1 Honeycomb API 13

User Features:

- Android market updates including easier automatic updates
- Google Books updates
- Wi-Fi improvements
- Chinese handwriting prediction improved

Android 3.2.2 Honeycomb API 13

- Minor Fixes

Android 3.2.4 Honeycomb API 13

- Added “Pay as you go” for tablets

Android 3.2.6 Honeycomb API 13

- Minor Fixes

Android 4.0 Ice Cream Sandwich

Android 4.0 Ice Cream Sandwich was released on October 18, 2011, and it came with an API version from 14 to 15. This version was designed to offer a unified experience for both smartphones and tablets. It was also the first Android version to support Face Unlock on select devices.

User Features:

- New lock screen actions
- Improved text input and spell-checking
- Control over network data
- Email app supports EAS v14
- Wi-Fi direct
- Bluetooth Health Device Profile

Developer Features:

- Low-level streaming multimedia
- Grid Layout
- Spell checking service
- Address Space Layout Randomization
- VPN client API
- Remote device camera enable/disable
- Flags to help control system UI elements like system bar from apps
- ZSL exposure, continuous focus, and image zoom

Next major OS updates:

Android 4.0.1 (API 14)

- Facial recognition (Face Unlock)
- UI use hardware acceleration
- Better voice recognition
- Web browser allows up to 16 tabs
- Updated launcher
- Android Beam app to exchange data through NFC

Android 4.0.2 (API 14)

- Minor Fixes

Android 4.0.3 (API 15)

- Social stream API in contacts provider to show updates associated to your contacts
- Video stabilization and QVGA video resolution API access
- Accessibility API refinements for screen readers
- Calendar provider updates

Android 4.0.4 (API 15)

- Stability improvements
- Better camera performance
- Smoother screen rotation

Android 4.1 Jelly Bean

Android 4.1 Jelly Bean got official on July 9, 2012, and it came with an API of 16 to 18. Android Jelly Bean is also officially the 10th iteration of Android and it was developed to offer performance improvements along with smooth user experience when compared to Android 4.0.

User Features:

- Google Now
- Voice Search
- Speed Enhancement
- Camera app improvements
- Accessibility: gesture mode, enable braille external keyboards

Developer Features:

- App stack navigation to define a parent activity in manifest for deep navigation
- MediaActionSound class to make sounds like when a camera takes a photo
- Large, Detailed, multi-action notifications
- Input manager allows you to query input devices
- WIFI/WIFI Direct service discovery
- NFC support large payloads over Bluetooth

Next major OS updates:

Android 4.1.1 (API 16)

- Fix a bug on screen orientation

Android 4.1.2 (API 16)

- Enable Home screen rotation
- Fix bugs and enhance performances

Android 4.2 (API 17)

User Features:

- Lockscreen widgets
- 360-degree images with photo sphere
- Gesture typing, for faster typing
- Wireless display with Miracast
- Daydream to display information when idle or docked
- Multi-user for tablets

Developer Features:

- vsync timing
- triple buffering
- Reduced touch latency
- CPU input boost

Android 4.2.1 (API 17)

- Fix missing December bug in the people app
- Add support for Bluetooth gamepads and joysticks HID devices

Android 4.2.2 (API 17)

- Shows the percentage and estimated time remaining in the active download notifications
- Wireless charging and low battery sounds changed
- Gallery app updated for faster loading with new image transition

Android 4.3 (API 18)

- Dial pad auto-complete
- 4k resolution support
- Camera app UI updated
- Security and performance enhancements

Android 4.4 KitKat

Android 4.4 KitKat got official on October 31, 2013 and it came with an API of 19 to 20. This version of Android was developed to offer an improved user experience on devices with limited hardware capabilities.

Features:

- Screen recording
- New Translucent system UI
- Enhanced notification access
- System-wide settings for closed captioning
- Performance improvements
- Enhance the camera on the Nexus 5
- Security Enhancements
- Several new APIs

Next major OS updates:

- Android 4.4.1 – API 19
- Android 4.4.2 – API 19
- Android 4.4.3 – API 19
- Android 4.4.4 – API 20

Android 5.0 Lollipop

Android 5.0 Lollipop offered a major design overhaul when compared to Android 4.4 KitKat and it was released on November 12, 2014 and it came with an API version 21 to 22. This version of Android offered a redesigned UI and it also replaced Dalvik with ART or Android Runtime to improve application performance and battery optimization.

Features:

- New Design (Material Design)
- Speed Improvement
- Battery consumption improvement
- Multiple SIM cards support
- Quick setting shortcuts to join Wi-Fi networks or control Bluetooth devices
- Lock protection if lost or stolen
- High-definition voice call

Next major OS updates:

- Android 5.0.1 – API 21
- Android 5.0.2 – API 21
- Android 5.1 – API 22
- Android 5.1.1 – API 22

Android 6.0 Marshmallow

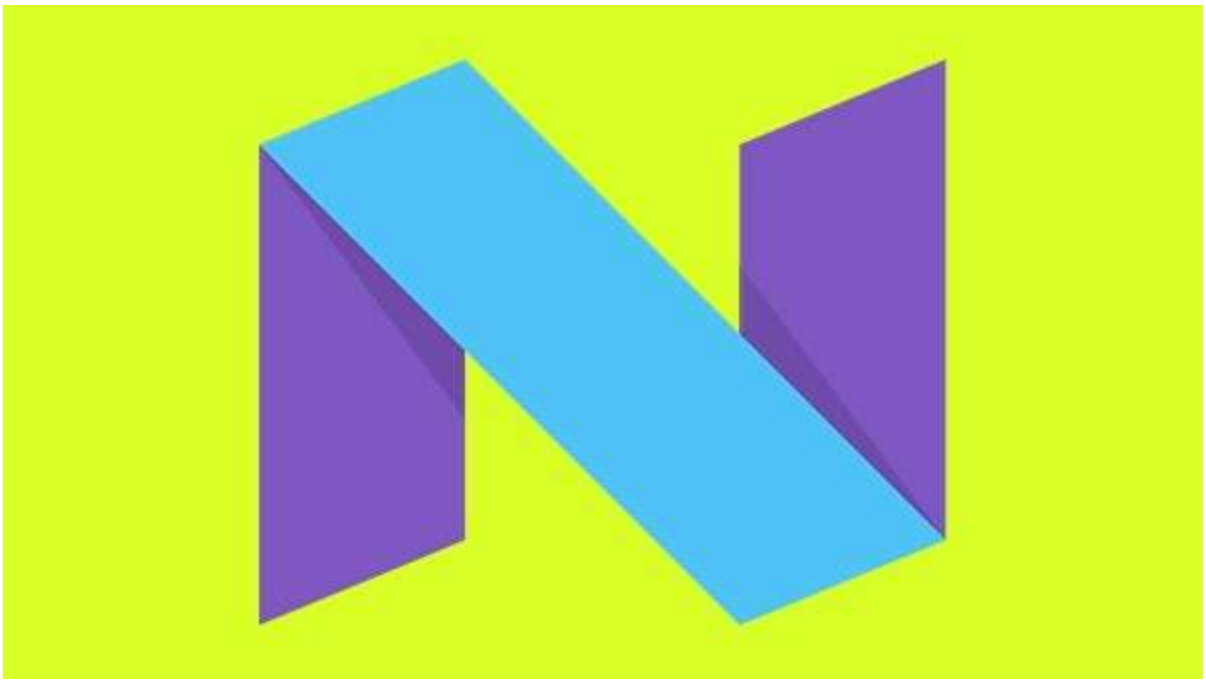
Android 6.0 Marshmallow was released on October 5, 2015, based on API 23. It offered a new permission architecture to improve user-privacy and it also natively supported USB Type-C port and a physical fingerprint sensor.

Features:

- USB Type C support
- Fingerprint Authentication support
- Better battery life with "deep sleep"
- Permission dashboard
- Android Pay
- MIDI support
- Google Now improvements
- New emojis

Next major OS updates:

- Android 6.0.1 – API 23



Android 7.0 Nougat

Android 7.0 Nougat with an API version of 24 to 25 was released on August 22, 2016. Nougat offered support for Vulkan API for better graphics rendering along with a new app notification format.

Features:

- Unicode 9.0 emoji
- Better multitasking
- Multi-window mode (PIP, Freeform window)
- Daydream Virtual Reality mode
- Night Light
- Storage manager improvements
- Option to enable fingerprint swipe down gestures
- Lockscreen redesign
- Battery usage alerts

Next major OS updates:

- Android 7.1 – API 25
- Android 7.1.1 – API 25
- Android 7.1.2 – API 25

Android 8.0 Oreo

Android 8.0 Oreo got launched on August 21, 2017, and this version was based API 26 to 27. Oreo was the first Android version to support Bluetooth 5.0 and wide color gamut.

Features:

- PIP – Picture in Picture with resizable windows
- Android Instant apps
- Improved notification system
- Improved system setting
- Lock screen redesign
- Automatic light and dark themes
- UI updates to 'Power Off' and 'Restart'
- Toast messages are now white in color with same existing transparency

Next major OS updates:

- Android 8.1 – API 27

Android 9 Pie

Android 9 Pie got official on August 6, 2018, with API 28 and it offered a refreshed material design with a new style of navigation buttons. Till today, a lot of smartphones are still based on Android 9 Pie.

Features:

- User interface updates:
    - Round corners across the UI
    - Quick setting menu change
    - Notification bar, the clock has moved to the left
    - New transitions when switching between apps or within apps
    - Volume slider updated
- Richer messaging notifications: with full conversation, large images, smart replies
- The power options now have a 'screenshot' button
- Biometric authentication can now be disabled only once

Android 10

Android 10 was released on September 3, 2019, based on API 29. This version was known as Android Q at the time of development and this is the first modern Android OS that doesn't have a dessert code name. It offered a complete full-screen user interface with a redesigned navigation system, which is a bit similar to the modern iPhones.

Features:

- Battery Privacy support
- Access system setting directly from apps using floating panel
- New audio/video codec support: AV1, HDR10+, Opus
- WPA3 Wi-Fi support
- Foldable phones support
- Dark theme
- Gesture navigation

Android 11

Android 11 was released on September 8, 2020 and this version was based on API 30. It comes with features like conversation notifications and this is also the first official Android version to offer built-in screen recorder.

Features:

- Screen recording
- Better permission settings
- Messaging bubbles
- 5G app support
- Wireless debugging
- Captive portal API
- Shared datasets

Android 12 (Snow Cone)

Android 12 was released on September, 2021 and this version was based on API 31.

It comes with features like:

- More customizable UI (could adjust the whole UI depending on the selected wallpaper)
- Security and privacy enhancements like a timeline to know what data your applications access
- “Material You” design language
- Android Private Compute Core
- AV1 Image File Format (AVIF) support