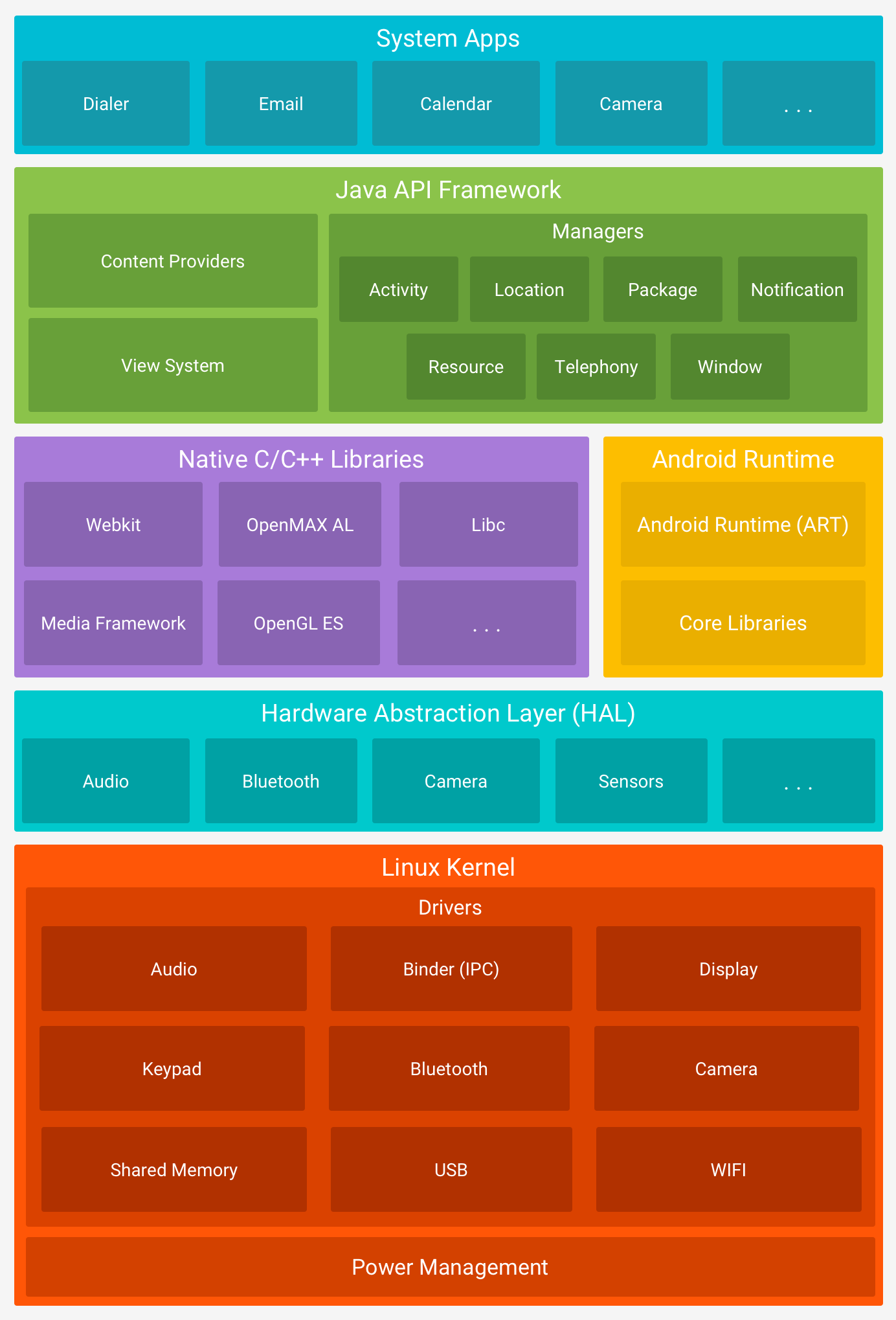
**Android Architecture**

****

**What is an Application class:** An Application class is a base class in your Application starts before all other classes like Activities or services are called.

**Activities**

Activity is like a frame or window in java that represents GUI. It represents single screen of android.

**Activity Lifecycle**



The 7 lifecycle method of Activity describes how activity will behave at different states.

|  |  |
| --- | --- |
| **onCreate** | called when activity is first created. |
| **onStart** | called when activity is becoming visible to the user. |
| **onResume** | called when activity will start interacting with the user. |
| **onPause** | called when activity is not visible to the user. |
| **onStop** | called when activity is no longer visible to the user. |
| **onRestart** | called after your activity is stopped, prior to start. |
| **onDestroy** | called before the activity is destroyed. |

**onPause**: When user goes from one Activity to another Activity previous activity goes in onPause state and when user press back button it goes to main page it called onStart and onResume

**onStop**: if user pressed **Home button** while app is launchedthen app moves to the Stopped state as still running in the background.

**Situations of activity life cycle states:**

* **When open the app**
  + onCreate() --> onStart() --> onResume()
* **When back button pressed and exit the app**
  + onPaused() -- > onStop() --> onDestory()
* **When home button pressed**
  + onPaused() --> onStop()
* **After pressed home button when again open app from recent task list or clicked on icon**
  + onRestart() --> onStart() --> onResume()
* **When open app another app from notification bar or open settings**
  + onPaused() --> onStop()
* **Back button pressed from another app or settings then used can see our app**
  + onRestart() --> onStart() --> onResume()
* **When any dialog open on screen**
  + onPause()
* **After dismiss the dialog or back button from dialog**
  + onResume()
* **Any phone is ringing and user in the app**
  + onPause() --> onResume()
* **When user pressed phone's answer button**
  + onPause()
* **After call end**
  + onResume()
* **When phone screen off**
  + onPaused() --> onStop()
* **When screen is turned back on**
  + onRestart() --> onStart() --> onResume()

**Example Realtime use of activity lifecycle**

* Suppose you are using a calculator app. Three methods are called in succession to start the app.
  + onCreate() - - - > onStart() - - - > onResume()
* When I am using the calculator app, suddenly a call comes the. The calculator activity goes to the background and another activity say. Dealing with the call comes to the foreground, and now two methods are called in succession.
  + onPause() - - - > onStop()
* Now say I finish the conversation on the phone, the calculator activity comes to foreground from the background, so three methods are called in succession.
  + onRestart() - - - > onStart() - - - > onResume()
* Finally, say I have finished all the tasks in calculator app, and I want to exit the app. Futher two methods are called in succession.
  + onStop() - - - > onDestroy()

**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.

**Content Resolver:**

* The Content Resolver is the single, global instance in your application that provides access to your (and other applications') content providers.
* There is only **one single instance** of the ContentResolver class in any given application. it accepts requests from clients, and resolves these requests by directing them to the content provider with a distinct authority. To do this, the Content Resolver stores a mapping from authorities to Content Providers.
* The Content Resolver includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, query, update, delete) in the Content Provider class.
* A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentResolver class.

**Intent:**  Use to pass data between activities and It is used to launch an activity, display a web page, send SMS, send email, etc. Start broadcast receiver, start background service

* **Implicit Intent:** The Implicit intent is used to invoke the system components.

e.g Dial call,

Intent i = new Intent(ACTION\_VIEW,Uri.parse("http://www.interview bit.com"));

startActivity(i);

* **Explicit Intent** : An explicit intent is used to invoke the activity class.

e.g. Intent i = **new** Intent(getApplicationContext(), ActivityTwo.**class**);

startActivity(i);

**Bundle**: Bundles are used to pass the required data between various Android activities. These are like HashMap that can take trivial data types. Below code shows how to transfer a piece of data by using bundle:

e.g. Use bundle in android with intent

FirstActivity:

Intent intent = new Intent(this, SecondActivity.class);

Bundle b=new Bundle();

b.putString("Email","abc@xyz.com");

intent.putExtras(b); // where i is intent

startActivity(intent);

SecondActivity:

Bundle bundle = getIntent().getExtras();

String title = bundle.getString("key1", "Default");

**startActivity**

It will start a new activity and not care when where and how that activity finishes.

e.g for instance: you have an app with a home-screen and a user-info screen: if you press the user-info button, you start the user-info activity with this.

**startActivityForResult**()

It starts another activity from your activity and it expect to get some data from newly started child activity by startAcitvityForResult() and return that to parent activity.

e.g. Start an activity and expect something in return. For instance, on your user-info screen, you can upload a profile picture. You start the gallery-activity with the explicit goal to get a URI back with the preferred picture. You start this activity literally to obtain a result

Intent i = new Intent(this, SecondActivity.class);

startActivityForResult(i, 100);

In your secondActivity class write following code for onClick Event For ex: In secondActivity if you want to send back data:

Intent intent= new Intent();

intent.putExtra("result",result);

setResult(RESULT\_OK,intent);

finish();

Now in your FirstActivity class write following code for the onActivityResult() method.

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data)

{

if (requestCode == 100 && resultCode == Activity.RESULT\_OK){

String result=data.getStringExtra("result");

Log.e("Result",result);

}

}

**Pending Intent**

A pending intent is a wrapper around regular intent that is designed to be used by another application.

PendingIntent is basically an object that wraps another Intent object. Then it can be passed to a foreign application where you’re granting that app the right to perform the operation, i.e., execute the intent as if it were executed from your own app’s process

* PendingIntent.getActivity() : Retrieve a PendingIntent to start an Activity
* PendingIntent.getBroadcast() : Retrieve a PendingIntent to perform a Broadcast
* PendingIntent.getService() : Retrieve a PendingIntent to start a Service

Example:

Intent intent = new Intent(this, SomeActivity.class);

// Creating a pending intent and wrapping our intent

PendingIntent pendingIntent = PendingIntent.getActivity(this, 1, intent,

PendingIntent.FLAG\_UPDATE\_CURRENT);

try {

// Perform the operation associated with our pendingIntent

pendingIntent.send();

} catch (PendingIntent.CanceledException e) {

e.printStackTrace();

}

**flag** : One of the PendingIntent flag that we’ve used in the above example is **FLAG\_UPDATE\_CURRENT**. This one state that if a previous PendingIntent already exists, then the current one will update it with the latest intent. There are many other flags like **FLAG\_CANCEL\_CURRENT** etc.

**Launch Modes:**

There are four launch modes for activity.

1. Standard: This is the default launch mode of activity. If you don’t set any launch mode to your activity, it will use the standard mode by default.

**Example:**

Suppose our current stack is A -> B -> C

and we launch the Activity B again with the launch mode “standard”

then the new stack will be A -> B -> C -> B

1. SingleTop:If an instance of activity already exists at the top of the current task, a new instance will not be created and the Android system will route the intent information through onNewIntent().

**Example:**

Suppose our current stack is A -> B -> C

and we launch the Activity B again with the launch mode “singleTop”

then the new stack will be A -> B -> C -> B.

1. SingleTask:An activity declared with launch mode as singleTask can have only one instance in the system (singleton). At a time only one instance of activity will exist.

Example ->

Suppose our current stack is A -> B -> C -> D

and we launch the Activity B again with the launch mode “singleTask” then the new stack will be A -> B.

old instance and C and D activities are destroyed.

1. SingleInstance:

It is similar to singleTask except that no other activities will be created in the same task. If another Activity is called from this kind of Activity, a new Task would be automatically created to place that new Activity.

Example ->

Suppose our current stack is A -> B -> C

and we launch the Activity D with the launch mode “singleInstance”

then there will be two stacks

A -> B -> C

D

If you call activity E then it will be added to the 1st stack.

1. A -> B -> C -> E
2. D

**Fragment**

The fragment is a part of Activity by which we can display multiple screens on one activity.

**Fragment Life Cycle**



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

----------------------**Fragment A**---------------------------

Intent intent = new Intent(getContext(), TutorialActivity.class);

intent.putExtra(“key”,msg);

startActivity(intent);

----------------------**Activity B** ---------------------------

Intent intent = getIntent();

String mesg = intent.getStringExtra(“key”);

* **Send data from Activity to Fragment:**

----------------------- **Activity A** ----------------------------

Bundle bundle = new Bundle();

bundle.putString("message", "From Activity");

// Set Fragmentclass Arguments

FragmentB fragobj = new FragmentB();

fragobj.setArguments(bundle);

getSupportFragmentManager().beginTransaction().replace(R.id.fragBlayoutID, fragobj).commit(); -- //now replace the argument fragment

----------------------**Fragment B**----------------------------

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

//Get Argument that passed from activity in "data" key value

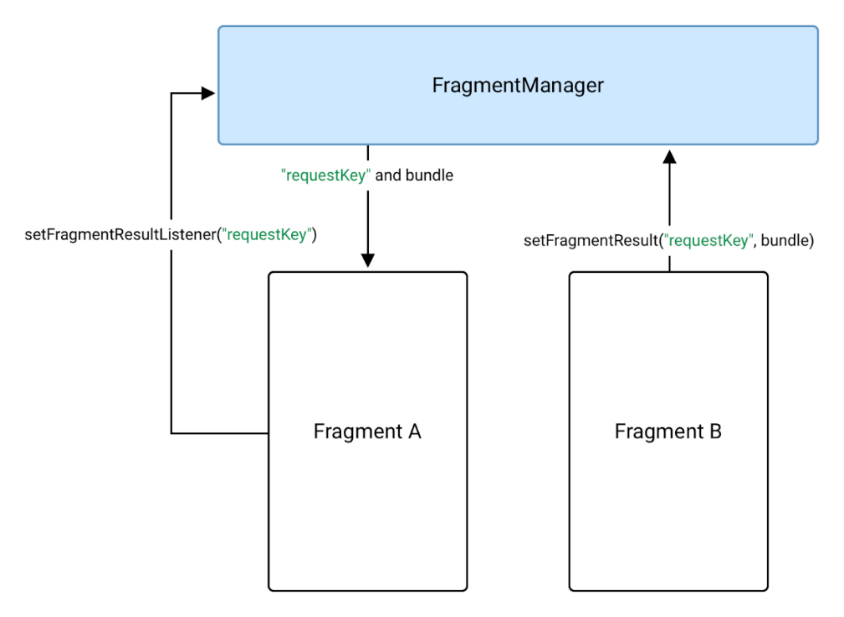
String strtext = getArguments().getString("message");

return view;

}

**Send Data from Fragment to Fragment:**

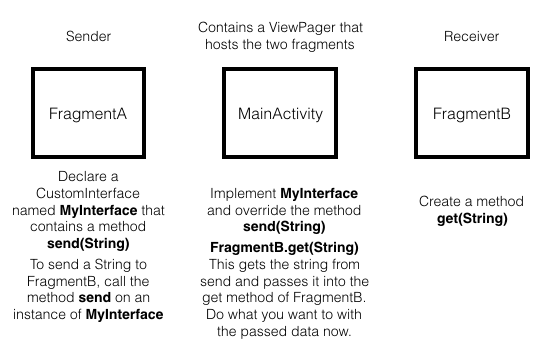
To pass data between fragments we need to create our own interfaces.



**Four ways to communication between Fragments:**

* **Interface**
* **ViewModel**
* **RxJava**
* **Event Bus**

The flow to send a String data from one Fragment to another is shown below.

****

**Add the following dependency in app level build.gradle**

implementation "androidx.fragment:fragment:1.4.0-rc01"

**-----------------Fragment A----------------------------------**

**public void onCreateView(View view, @Nullable Bundle savedInstanceState) {**

**Bundle bundle = new Bundle();**

**bundle.putString(“key”, “value”);**

**getParentFragmentManager().setFragmentResult(“dataFrom”, bundle);**

**}**

**------------Fragment B----------------------------------------**

**public void onCreateView(View view, @Nullable Bundle savedInstanceState) {**

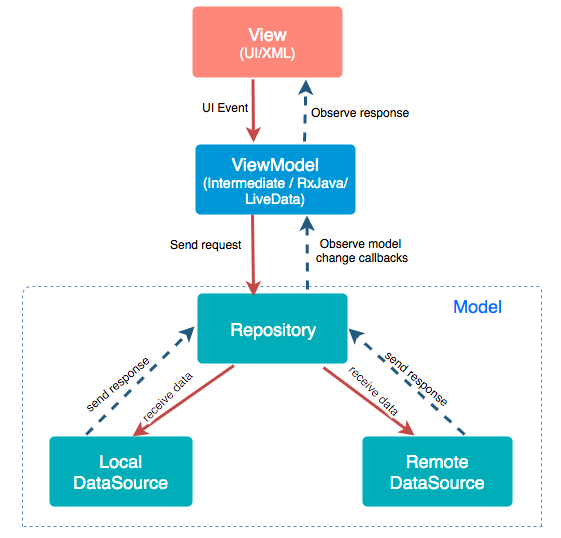
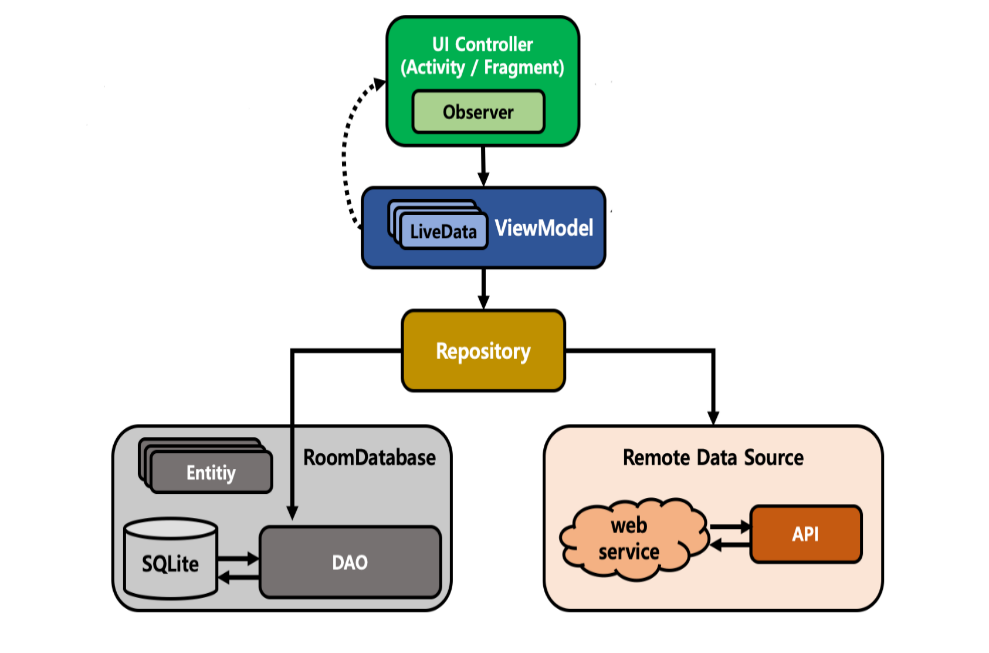
**getParentFragmentManager().setFragmentResultListener(“dataFrom”, this, new FragmentResultListener(){**

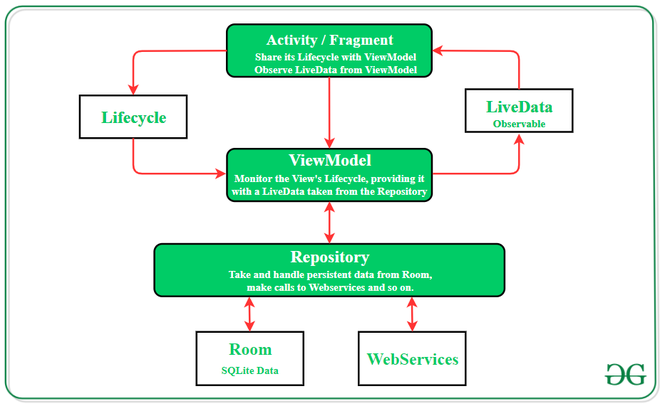
**public void onFragmentResult(String reqKey, Bundle bundle){**

**String result = bundle.getString(“key”);**

**});}**

**Android Architecture Used in Project:**

**MVVM**: MVVM architecture is a Model-View-View Model architecture that removes the tight coupling between each component. Most importantly, in this architecture, the children don't have the direct 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, and repository.

View: It consists of the UI Code (Activity, Fragment), XML. It sends the user action to the View Model 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. It interacts with the Model and exposes the observable that can be observed by the View. The ViewModel's role is to provide data to the UI and survive configuration changes. We can also use a ViewModel to share data between fragments. The ViewModel is part of the lifecycle library.

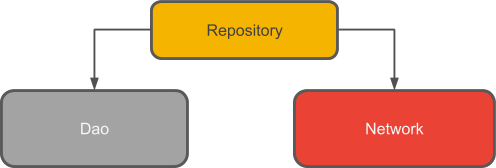
**Note**: If you need the application context, use AndroidViewModel

**What is a Repository?**

A Repository is a class that abstracts access to multiple data sources.A Repository class handles data operations.

**Why use a Repository?**

**Answer:** A Repository manages query threads and allows you to use multiple backends. In the most common example, the Repository implements the logic for deciding whether to fetch data from a network or use results cached in a local database.



### Advantages of MVVM Architecture

* Enhance the reusability of code.
* All modules are independent which improves the testability of each layer.
* Makes easier for unit testing
* Removes tightly coupling between each component.
* Bindings make UI updates easier to handle
* Review at the compilation stage.
* No interfaces between view and model.
* Clean code structure of application
* Developement flexibility
* Seperation of logic

### Disadvantages of MVVM Architecture

* This design pattern is not ideal for small projects.
* If the data binding logic is too complex, the application debug will be a little harder.

**Model View Presenter (MVP):**

MVP (Model View Presenter) design pattern is derived from MVC pattern most of the things is same, where is the controller is replaced by presenter.

It provides modularity, testability, and a more clean and maintainable codebase.

the presenter knows about the view and view knows about the presenter. They interact with each other through an interface.



It is composed of the following three components:

* **Model:** Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers. This handles the data part of our application.
* **View:** UI(User Interface) layer. It provides the visualization of the data and keep a track of the user’s action in order to notify the Presenter.
* **Presenter:** It is the mediator between Model and View. Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user’s input notification from the View.The presenter will contain much of the business code and replaces the controller from MVC.

1. Activity, Fragment and a CustomView act as the View part of the application.
2. The Presenter is responsible for listening to user interactions (on the View) and model updates (database, APIs) as well as updating the Model and the View.
3. Generally, a View and Presenter are in a one to one relationship. One Presenter class manages one View at a time.
4. [Interfaces](https://www.journaldev.com/1601/interface-in-java) need to be defined and implemented to communicate between View-Presenter and Presenter-Model.
5. The Presenter is responsible for handling all the background tasks. Android SDK classes must be avoided in the presenter classes.
6. The View and Model classes can’t have a reference of one another

**Note**: The View never communicates with Model directly.

**Key Points of MVP Architecture**

1. Communication between View-Presenter and Presenter-Model happens via an **interface(also called Contract)**.
2. One Presenter class manages one View at a time i.e., there is a one-to-one relationship between Presenter and View.
3. Model and View class doesn’t have knowledge about each other’s existence.

**Advantages of MVP Architecture**

* No conceptual relationship in android components
* Easy code maintenance and testing as the application’s model, view, and presenter layer are separated.

**Model View Controller (MVC):**



MVC Architecture pattern is a way how the information or data is been presented to the user & how the user interacts/deals with the data view.

**Model:** It contains application data and business logicand rules of the application.  
A Model is responsible for managing data of an app.

It has no knowledge about the interface. The model is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

**VIEW:** Itis theUI(User Interface) layer that holds components that are visible on the screen. A view means presentation of the data in a particular format.

**CONTROLLER:** This component establishes the relationship between the **View** and the **Model.**It contains the core application logic and gets informed of the user’s behavior and updates the Model as per the need. It control all the task/event that a user perform, Such as event handling, navigation, Communication between model & view happens in controller in MVC.  
A Controller receive the input, validate it, & pass the validated input to Model.

**Advantages of MVC architecture pattern**

* MVC pattern increases the code testability and makes it easier to implement new features as it highly supports the separation of concerns.
* Unit testing of Model and Controller is possible as they do not extend or use any Android class.
* Functionalities of the View can be checked through UI tests if the View respect the single responsibility principle

**Disadvantages of MVC architecture pattern**

* Code layers depend on each other even if MVC is applied correctly.
* No parameter to handle UI logic i.e., how to display the data.

**MVP vs MVVM:**

In MVP, the presenter knows about the view and view knows about the presenter. They interact with each other through an interface. In MVVM, only the view knows about the view-model. The view-model has no idea about the view.

**MVP Disadvantages:**

* It takes more time to code the entire interface.
* Tight coupling between View and Presenter
* Huge amount of interfaces for interaction between layers.
* The code size is quite excessive.

MVVM is a further improvement over the MVP architecture pattern. In this pattern, the Presenter is replaced by the ViewModel which handles the entire data processing.

**JAVA Classes:**

In both the architecture patterns, there is a huge number of JAVA classes. However, the code length of the classes in MVVM is comparatively low.

**Data Input:**

In the case of MVP, the data input happens at the View. Similarly, the MVVM receives data input from View too.

**MVC Disadvantages:**

* Due to large code controller is unmanageable.
* Hinders the Unit testing
* Increased Complexity

**Ways to Implement MVVM in the Project**

There are 2 ways to implement MVVM design pattern in Android projects:

* Using the DataBinding library released by Google(Two Way Data Binding)
  + Two-way Data Binding is a technique of binding your objects to your XML layouts such that the Object and the layout can both send data to each other.
* Using any tool like RxJava for DataBinding.

**DataBinding in android?**

* 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.
* It replaces findViewByID
* Allows you to more easily write code that interacts with use.
* for each XML layout file present in that module and instance of a binding class contains direct references
* It helps in minimizing the code of core application logic that binds with View.

First set databinding to true in app/build.gradle

databinding {

enabled **true**

}

Add below line of code in layout file

<layout xmlns:bind="http://schemas.android.com/apk/res/android">

<data>

<variable

name="user"

type="file name which inside package" />

</data>

<LinearLayout ...>

<!—YOUR ACTUAL LAYOUT HERE -->

<EditText android:id="@+id/inPassword" android:layout\_width="match\_parent" android:layout\_height="wrap\_content" android:hint="Password" android:inputType="textPassword" android:padding="8dp" android:text="@={viewModel.userPassword}" />

<Button

android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Button"

android:onClick=" @{()-> user.onFabClicked()}"/>

</LinearLayout>

</layout>

public class Model {

    @Nullable

    String email,password;

    // constructor to initialize

    // the variables

    public Model(String email, String password){

        this.email = email;

        this.password = password;

    }

    // getter and setter methods

    // for email variable

    @Nullable

    public String getEmail() {

        return email;

    }

    public void setEmail(@Nullable String email) {

        this.email = email;

    }

    // getter and setter methods

    // for password variable

    @Nullable

    public String getPassword() {

        return password;

    }

    public void setPassword(@Nullable String password) {

        this.password = password;

    }

}

public class AppViewModel extends BaseObservable {

    // creating object of Model class

    private Model model;

    // getter and setter methods

    // for email variable

    @Bindable

    public String getUserEmail() {

        return model.getEmail();

    }

    public void setUserEmail(String email) {

        model.setEmail(email);

        notifyPropertyChanged(BR.userEmail);

    }

    // getter and setter methods

    // for password variable

    @Bindable

    public String getUserPassword() {

        return model.getPassword();

    }

    public void setUserPassword(String password) {

        model.setPassword(password);

        notifyPropertyChanged(BR.userPassword);

    }

    // constructor of ViewModel class

    public AppViewModel() {

        // instantiating object of

        // model class

        model = new Model("","");

    }

}

Create object of databinding and setContentView in class by using below code

The View class is responsible for updating the UI of the application.

ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity\_main);

**ViewBinding:**

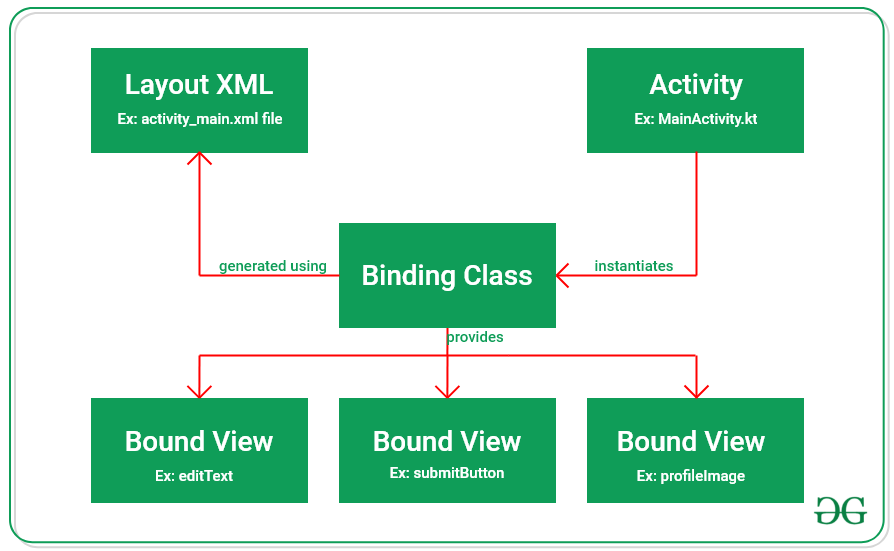
**View Binding** is one of the best features which provides the views to bind with the activity which is ongoing. Replacing the **findViewById()**method, hence reducing the boilerplate code, generated the instances of the views of the current layout.

With view binding, the layouts do not need a layout tag.

* You can't use viewbinding to bind layouts with data in xml (No binding expressions, no BindingAdapters nor two-way binding with viewbinding)
* when view binding is enabled in an android project module. It automatically generates a binding class for the layout file which contains direct reference to those views and by the id that is given to them in the XML layout.
* While using the ViewBinding proper naming conventions are need to be followed because it creates the binding class internally using the name of the same layout file.

**Advantages:**

* The main advantages of viewbinding are speed and efficiency. It has a shorter build time because it avoids the overhead and performance issues associated with databinding due to annotation processors affecting databinding's build time.
* ViewBinding is always null safe and type-safe, which supports both Java and Kotlin.
* Speed Increase -> No more findViewById code and no need to worry if you remove a view errors will start showing right away
* Code Readability -> Code readability is highly increased if you give sensible ids to the views
* Reduces Crashes -> This is achieved by type safety if you are never going to define what a view is to be cast as you can not make a mistake.
* Using ViewBinding the compilation of the code is a bit faster as compared to the traditional findViewById() method.



**Steps to use viewbinding in code**

1. **Enabling the ViewBinding Feature in build.gradle(app)**

**viewBinding {**

**enabled = true**

**}**

1. **Working with the activity\_main.xml file.**

**Just remove <layout> tag as compared to data binding layout.**

**ViewBinding which automatically generates all layout to Binding class.**

**<?xml version="1.0" encoding="utf-8"?>**

**<androidx.constraintlayout.widget.ConstraintLayout**

**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:layout\_width="match\_parent"**

**android:layout\_height="match\_parent"**

**tools:context=".MainActivity"**

**tools:ignore="HardcodedText">**

**<Button**

**android:id="@+id/button"**

**android:layout\_width="wrap\_content"**

**android:layout\_height="wrap\_content"**

**android:layout\_marginTop="16dp"**

**android:text="Button"/>**

**</androidx.constraintlayout.widget.ConstraintLayout>**

1. **Working with the MainActivity file**

ActivityMainBinding activityMainBinding = ActivityMainBinding. inflate( layoutInflater);

**Accessing the properties of the layout goes:** activityMainBinding.buttonSubmit.setOnClickListener

**Android Jetpack?**

Android Jetpack is a collection of Android software components which helps us in building great Android apps.

**Jetpack Libraries/Components:**

* Data Binding.
* LiveData
* ViewModel
* Navigation Component.
* WorkManager
* Room
* Paging

**Following are the architecture component’s element and their responsibilities**

* 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.
* **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.
* 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 observable LiveData and makes it accessible to ViewModel.
* **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 ability to return queries directly with observable LiveData.

Managing activity lifecycles.

Surviving configuration changes.

Preventing memory leaks.

All these major problems have been solved by the Android Jetpack's software components.

These software components have been arranged in 4 categories which are as follows:

1. Foundation Components: Backward compatibility, Testing, Kotlin language support.

App Compat: Degrade gracefully on older versions of Android with material design user interface implementation support.

**Android KTX**: Set of Kotlin extensions to write more concise, idiomatic Kotlin code.

Multidex: Provide support for multiple dex files for apps.

Test: A testing framework for unit and runtime UI tests in Android.

2.Architecture Components: Robust Apps, Testable Apps, Maintainable Apps

LiveData: Notify views of any database changes.

Navigation: Handle everything needed for in-app navigation.

#### Mobile\_navigation.Xml

Add navigation’s in navigation folder so that we can easily navigate in between fragments it’s again a optional process.

If you are not familiar with this declarations you may follow the traditional way of navigation using fragment managers.

1. *Destinations* are the different content areas in your app.
2. *Actions* are logical connections between your destinations that represent paths that users can take.

Paging: Gradually load information on demand from your data source.

Room: Fluent SQLite database access.

ViewModel: Manage UI-related data in a lifecycle-conscious way.

WorkManager: Manage every background jobs in Android with the circumstances we choose.

Data Binding: Declaratively bind UI elements to in our layout to data sources of our app.

Lifecycles: Manages activity and fragment lifecycles of our app.

3.Behavior Components: Notifications, Permissions , Sharing , Assistant

The behavior components help in the integration with standard Android services like

Download Manager: Schedule and manage large downloads in background with auto retry support.

Media & playback: Backwards compatible APIs for media playback and routing (including Google Cast).

Notifications: Provides a backwards-compatible notification API with Wear and Auto support.

Perms=ions: Compatibility APIs for checking and requesting permissions in app.

Preferences: Create interactive settings screens for users to configure.

Sharing: Provides a share action suitable for an app’s action bar.

**4.UI Components**

The UI components provide widgets and helpers to make your app not only easy, but delightful to use.

All the UI components are as follows:

Animation and transitions: Move widgets and transition between screens.

TextView, EditText, Button, RadioButton, CheckBox, ProgressBar, Spinner, SeekBar, AlertDialog

Auto: Components to develop Android Auto apps.

Fragment: A basic unit of composable UI.

Layout: Lay out widgets with different algorithms.

Palette: Pull useful information from color palettes.

**UI Layout**

 ViewGroup is a subclass of View and provides invisible container that hold other Views or other ViewGroups and define their layout properties.

**1. Linear Layout**

LinearLayout is a view group that aligns all children in a single direction, vertically or horizontally.

**2.Relative Layout**

RelativeLayout is a view group that displays child views in relative positions.

3.**ConstraintLayout:** ConstraintLayout is a ViewGroup which allows you to position and size widgets in a adaptable and flexible way to create, which is now the default layout in Android Studio.( starting with API level 9 (Gingerbread))

This allows you to create large, complex, dynamic and responsive views in a flat hierarchy. e.g(**layout\_constraintLeft\_toLeftOf**, **layout\_constraintLeft\_toRightOf**, **layout\_constraintRight\_toLeftOf, layout\_constraintBottom\_toBottomOf**, **layout\_constraintTop\_toTopOf**)

Advantages: ConstraintLayout provides you the ability to completely design your UI with the drag and drop feature provided by the Android Studio design editor

we can control the group of widgets through a single line of code.

we can easily add animations to the UI components which we used in our app.

improve the UI performance over other layouts.

**4.Table Layout**

TableLayout is a view that groups views into rows and columns.

**5.Absolute Layout**

AbsoluteLayout enables you to specify the exact location of its children.

**6.Frame Layout**

The FrameLayout is a placeholder on screen that you can use to display a single view.2+58

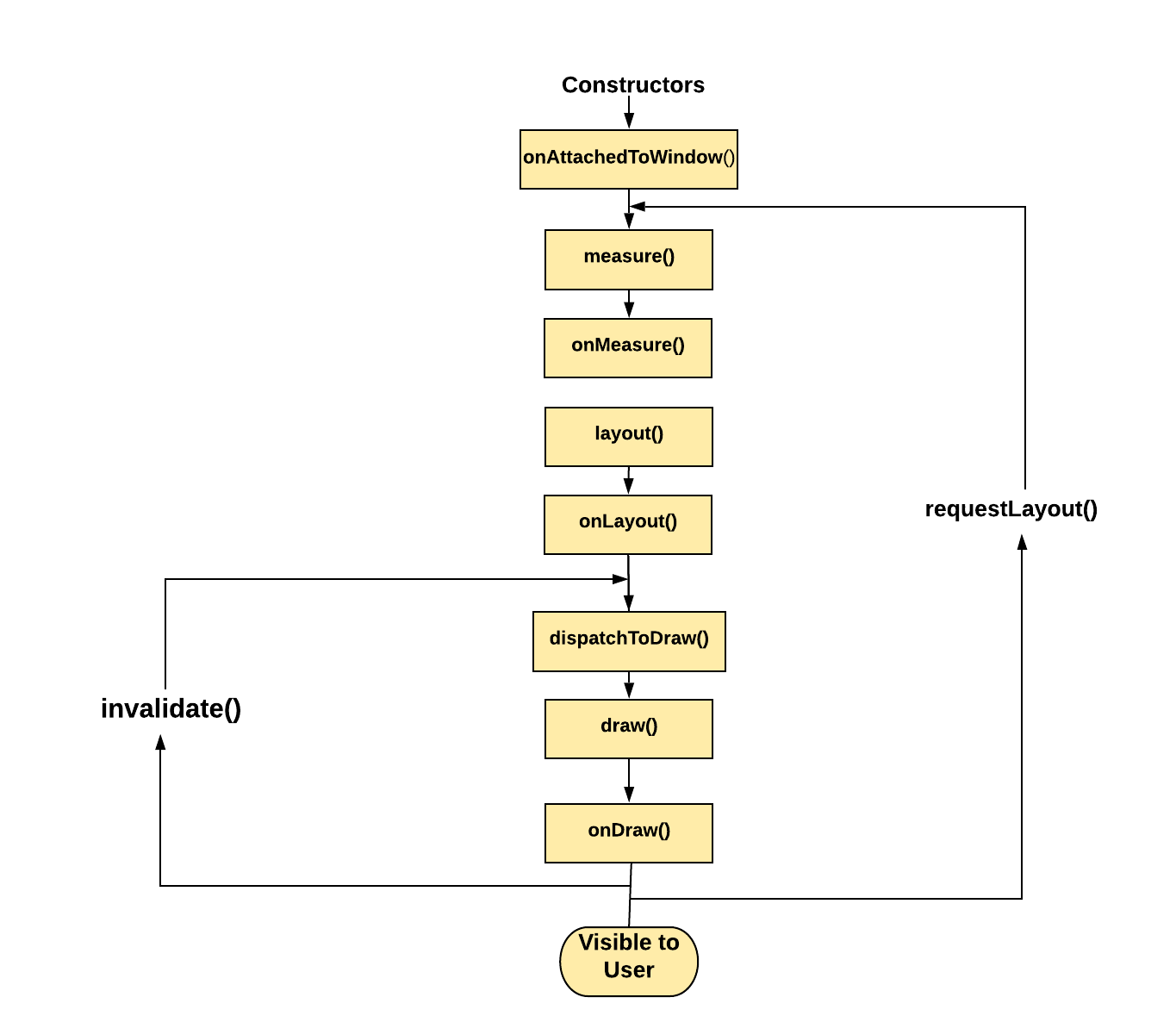
**7. List View**

ListView is a view group that displays a list of scrollable items.

**8.Grid View**

GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

**Custom View:**



## **onAttachedToWindow()**

Called when the view is attached to a window. This is the phase where the view knows it can be active and has a surface for drawing.

## **onMeasure()**

This is called to find out how big a view should be. In the case of ViewGroup, it will go ahead and call measure on each of their child views and the results can help to decide its own size.

## **onLayout()**

This is called after measuring the views to position them on the screen.

**onDraw()**

Sizes and positions are calculated in previous steps, so the view can draw itself based on them. In onDraw(Canvas canvas) Canvas object generated (or updates) has a list of **OpenGL-ES** commands (displayList) to send to the **GPU.**

## **invalidate()**

invalidate() is a method that insists on force reDrawing of a particular view that we wish to show changes.

## **requestLayout()**

At some point, there is a state change in the view. requestLayout() is the signal to the view system that it needs to recalculate the Measure and Layout phase of the views (measure → layout → draw).

**Adapter**:

**The Adapter provides access to the data items**.

Adapter is **a bridge between UI component and data source that helps us to fill data in UI component**. It holds the data and send the data to an Adapter view then view can takes the data from the adapter view and shows the data on different views like as ListView, GridView, Spinner etc.

**ListView:**

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

The main purpose of the adapter is to fetch data from an array or database.

 ListView should only be used when the number of items is always the same and is limited to the screen size.

ListView l = findViewById(R.id.list);

String tutorials[]

        = { "Algorithms", "Data Structures",

            "Languages", "Interview Corner",

            "GATE", "ISRO CS",

            "UGC NET CS", "CS Subjects",

            "Web Technologies" };

        ArrayAdapter<String> arr;

        arr

            = new ArrayAdapter<String>(

                this,

                R.layout.support\_simple\_spinner\_dropdown\_item,

                tutorials);

        l.setAdapter(arr);

**RecyclerView:**

**Need for View Recycling in Android**

It is a practice to use as little memory as possible by recycling unused views to display new content instead of creating new views for the same. Suppose, we are scrolling down through a list of one thousand words. If we create a TextView for each word, we would need one thousand TextViews for this. This would waste a lot of memory since our device’s screen displays only 7-8 TextViews at a time and we need to scroll down if we want to see the rest of them.

**It is more flexible and advance version of list view and grid view.**

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

RecyclerView makes it **easy to efficiently display large sets of data.**

RecyclerView provides **an ability to implement the horizontal, vertical and Expandable List**.

It is mainly used when we have data collections whose elements can change at run time based on user action or any network events.

The RecyclerView is much more powerful, flexible and a **major enhancement over ListView**.

Android [RecyclerView](https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html" \t "_blank) is more advanced version of ListView with improved performance and other benefits. Using RecyclerView and [CardView](https://developer.android.com/reference/android/support/v7/widget/CardView.html" \t "_blank) together, both lists and grids can be created very easily

### LayoutManager

The RecyclerView uses a LayoutManager to determine how the Views will be placed within the RecyclerView. The different LayoutManagers supported by RecyclerView include:

* LinearLayoutManager, a vertical or horizontal list of items
* GridLayoutManager, a grid of items that are all the same size
* StaggeredGridLayoutManager, a grid of items that supports different sizes

**public** **class** MyListAdapter **extends** RecyclerView.Adapter<MyListAdapter.ViewHolder>{

**private** MyListData[] listdata;

   // RecyclerView recyclerView;

**public** MyListAdapter(MyListData[] listdata) {

**this**.listdata = listdata;

    }

    @Override

**public** ViewHolder onCreateViewHolder(ViewGroup parent, **int** viewType) {

        LayoutInflater layoutInflater = LayoutInflater.from(parent.getContext());

        View listItem= layoutInflater.inflate(R.layout.list\_item, parent, **false**);

        ViewHolder viewHolder = **new** ViewHolder(listItem);

**return** viewHolder;

    }

    @Override

**public** **void** onBindViewHolder(ViewHolder holder, **int** position) {

**final** MyListData myListData = listdata[position];

        holder.textView.setText(listdata[position].getDescription());

        holder.imageView.setImageResource(listdata[position].getImgId());

        holder.relativeLayout.setOnClickListener(**new** View.OnClickListener() {

            @Override

**public** **void** onClick(View view) {

                Toast.makeText(view.getContext(),"click on item: "+myListData.getescription(),Toast.LENGTH\_LONG).show();

            }

        });

    }

    @Override

**public** **int** getItemCount() {

**return** listdata.length;

    }

**public** **static** **class** ViewHolder **extends** RecyclerView.ViewHolder {

**public** ImageView imageView;

**public** TextView textView;

**public** RelativeLayout relativeLayout;

**public** ViewHolder(View itemView) {

**super**(itemView);

**this**.imageView = (ImageView) itemView.findViewById(R.id.imageView)

**this**.textView = (TextView) itemView.findViewById(R.id.textView);

            relativeLayout = (RelativeLayout)itemView.findViewById(R.id.relativeLayout);

        }

    }

}

RecyclerView recyclerView = (RecyclerView) findViewById(R.id.recyclerView);

MyListAdapter  adapter = new MyListAdapter(collageFileList);

recyclerView.setAdapter(adapter);

### What are the core building blocks of android?

The core building blocks of Android are:

* Activity
* View
* Intent
* Service
* Content Provider
* Fragment etc.

**Services:**

[Service](https://developer.android.com/reference/android/app/Service) is an [application component](https://developer.android.com/guide/components/fundamentals#Components) that can perform long-running operations in the background. It does not provide a user interface

Android Services are the application components that run in the background. It doesn’t need any direct user interaction. It is used to perform operations on the background such as playing music, handle network transactions and it doesn’t have any UI.

The service runs in the background indefinitely even if application is destroyed**.**

Types of Android Services

These are the three different types of services:

1. Foreground Services

Foreground services are those services that are visible to the users. The users can interact with them at ease and track what’s happening. These services continue to run even when users are using other applications.

The perfect example of this is Music Player and Downloading.

2. Background Services

These services run in the background, such that the user can’t see or access them. These are the tasks that don’t need the user to know them.

Syncing and Storing data can be the best example.

3. Bound Services

Bound service runs as long as some other application component is bound to it. Many components can bind to one service at a time, but once they all unbind, the service will destroy.

**Lifecycle of Android Services**

Android services life-cycle can have two forms of services and they follow two paths, that are: 1.Started Service 2.Bounded Service

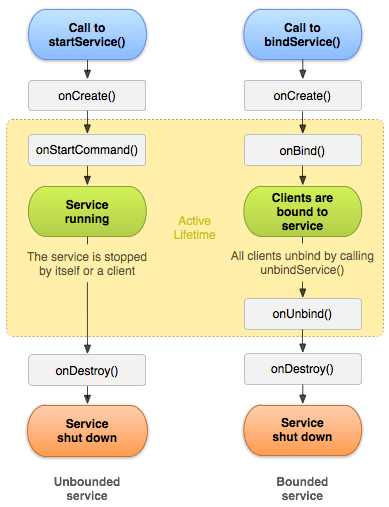
1. Started

A service is started when an application component, such as an activity, starts it by calling startService().

2. Bound

A service is bound when an application component binds to it by calling bindService().

The service cannot be stopped until all clients unbind the service.



**Life Cycle of Service**

[onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int))

The system invokes this method by calling [startService()](https://developer.android.com/reference/android/content/Context" \l "startService(android.content.Intent)) when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling [stopSelf()](https://developer.android.com/reference/android/app/Service" \l "stopSelf()) or [stopService()](https://developer.android.com/reference/android/content/Context" \l "stopService(android.content.Intent)). If you only want to provide binding, you don't need to implement this method.

[onBind()](https://developer.android.com/reference/android/app/Service#onBind(android.content.Intent))

The system invokes this method by calling [bindService()](https://developer.android.com/reference/android/content/Context" \l "bindService(android.content.Intent,%20android.content.ServiceConnection,%20int)) when another component wants to bind with the service (such as to perform RPC).

[onCreate()](https://developer.android.com/reference/android/app/Service#onCreate())

The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either [onStartCommand()](https://developer.android.com/reference/android/app/Service" \l "onStartCommand(android.content.Intent,%20int,%20int)) or [onBind()](https://developer.android.com/reference/android/app/Service" \l "onBind(android.content.Intent))). If the service is already running, this method is not called.

[onDestroy()](https://developer.android.com/reference/android/app/Service#onDestroy())

The system invokes this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

**Example**

the Music app of our mobile device or any other Music application runs in the background and while using the Music app, you can use any other application normally. So, this feature is implemented using the **Service** or **IntentService**.

 If you some limited amount of tasks to be performed in the background, then you can use Service, otherwise, you can use IntentService.

Service class uses the application’s main thread, while IntentService creates a worker thread and uses that thread to run the service.

Intent Service:

It is a base class for services to handle asynchronous requests. It enables running an operation on a single background. It executes long-running programs without affecting any user’s interface interaction. Intent services run and execute in the background and terminate themself as soon as they are executed completely.

* It queues up the upcoming request and executes them one by one.
* Once the queue is empty it stops itself, without the user’s intervention in its lifecycle.

A service can run continuously in the background even if the application is closed or the user switches to another application.

There is a major difference between android services and threads, one must not be confused between the two. Thread is a feature provided by the Operating system to allow the user to perform operations in the background. While service is an [android component](https://www.geeksforgeeks.org/components-android-application/) that performs a long-running operation about which the user might not be aware of as it does not have UI.

**How to use service in code.**

Unbounded service

onCreate

onStartCommand

onDestroy

Bounded service

onCreate

onBind------onRebind-|

onUnBind ------------|

onDestroy

class MediaPlayerDemo extends service

 private MediaPlayer player;

public int onStartCommand(Intent intent, int flags, int startId) {

        player = MediaPlayer.create( this, Settings.System.DEFAULT\_RINGTONE\_URI );

        // providing the boolean

        // value as true to play

        // the audio on loop

        player.setLooping( true );

        // starting the process

        player.start();

}

// execution of the service will

// stop on calling this method

public void onDestroy() {

    super.onDestroy();

    // stopping the process

    player.stop();

}

public IBinder onBind(Intent intent){

        return null;

}

Declare service in manifest file.

 <service android:name=".NewService"/>

**MainActivity.java**

Class MainActivity extends Activity{

 public void onClick(View view) {

        // process to be performed

        // if start button is clicked

        if(view == start){

            // starting the service

Intent intent = new Intent(this, MediaPlayerDemo.class);  
 startService(intent);

        }

        // if stop button is clicked

        else if (view == stop){

            // stopping the service

Intent intent = new Intent(this, MediaPlayerDemo.class);  
 stopService(intent);

        }

    }

}

### Stopping a service

A started service must manage its own lifecycle. That is, the system doesn't stop or destroy the service unless it must recover system memory and the service continues to run after [onStartCommand()](https://developer.android.com/reference/android/app/Service" \l "onStartCommand(android.content.Intent,%20int,%20int)) returns. The service must stop itself by calling [stopSelf()](https://developer.android.com/reference/android/app/Service" \l "stopSelf()), or another component can stop it by calling [stopService()](https://developer.android.com/reference/android/content/Context" \l "stopService(android.content.Intent)).

**States of Services:**

* Destroyed
* Start
* Running

**Differentiate Activities from Services.**

Activities can be closed, or terminated anytime the user wishes. On the other hand, services are designed to run behind the scenes, and can act independently.

| **Activities** | **Services** |
| --- | --- |
| They are designed to run in the foreground. | These are mainly designed to run in the background. Foreground services are also available. |
| Used when the user interface is necessary. | Used when the user interface is not necessary. |
| They are dependent. | They act independently. |

Service Thread

|  |  |
| --- | --- |
| When an application is killed, service is not killed. | When an application is killed, the thread is killed. |

**How to update Ui from Service in Android:**

* **Using Intent Service:**
  + Register broadcast receiver using intent filter in Activity.
  + Update the UI data in text in **onReceive** method.
  + Start the service in onCreate method.
  + Create the intent service class and call **sendBroadcast** data using intent object with key value pair.
* **Using Service:**
  + Register broadcast receiver using intent filter in Activity.
  + Update the UI data in text in **onReceive** method
  + Create the service class and call **sendBroadcast** data from Runnable class with handler using intent object with key value pair.

**Broadcast Receivers**

* A broadcast receiver is a mechanism used for listening to system-level events like listening for incoming calls, SMS, etc. by the host application. It is implemented as a subclass of BroadcastReceiver class and each message is broadcasted as an intent object.
* It is component of android that listens to system-wide broadcast events or intents.
* It doesn’t contain any user interface.
* Broadcast Receivers are used to respond to these system-wide events.
* They handle communicatMion between Android OS and applications.
* Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents.
* android.intent.action.BATTERY\_LOW, android.intent.action.CALL
* when a message is received on the device or when incoming calls are received, or when a device goes to airplane mode, etc. Broadcast Receivers are used to respond to these system-wide events. Broadcast Receivers allow us to register for the system and application events, and when that event happens, then the register receivers 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
* Two important steps to make BroadcastReceiver works for the system broadcasted intents −
  + Creating the Broadcast Receiver.
  + Registering Broadcast Receiver

## **Creating the Broadcast Receiver**

* public class MyReceiver extends BroadcastReceiver {
* @Override
* public void onReceive(Context context, Intent intent) {
* Toast.makeText(context, "Intent Detected.", Toast.LENGTH\_LONG).show();
* }
* }

## **Registering Broadcast Receiver**

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file

receiver android:name="MyReceiver">

<intent-filter>

<action android:name="android.intent.action.BOOT\_COMPLETED">

</action>

</intent-filter>

</receiver>

* **Register defining it programmatically**
* IntentFilter filter = new IntentFilter(); intentFilter.addAction(getPackageName() + "android.net.conn.CONNECTIVITY\_CHANGE"); MyReceiver myReceiver = new MyReceiver(); registerReceiver(myReceiver, filter);

## **Broadcasting Custom Intents**

If you want your application itself should generate and send custom intents then you will have to create and send those intents by using the *sendBroadcast()* method inside your activity class. If you use the *sendStickyBroadcast(Intent)* method, the Intent is **sticky**, meaning the *Intent* you are sending stays around after the broadcast is complete.

**Sticky Intent?**

Sticky Intents allows communication between a function and a service. sendStickyBroadcast() performs a sendBroadcast(Intent) known as sticky.

public void broadcastIntent(View view) {

Intent intent = new Intent();

intent.setAction("com.tutorialspoint.CUSTOM\_INTENT");

sendBroadcast(intent);

}

**Since Android Oreo, implicit broadcast receivers won’t work when registered in the AndroidManifest.xml.**

**How to handle Implicit Receivers in Android Oreo?**: To use Implicit Receivers in your application, you need to define them programmatically in your code, using registerReceiver()

**LocalBroadcastManager:**

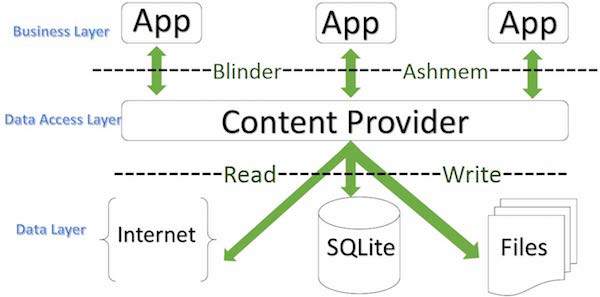
to register for and send broadcasts of Intents to local objects within your process. This has a number of advantages over sending global broadcasts with [Context.sendBroadcast(Intent)](https://developer.android.com/reference/android/content/Context.html" \l "sendBroadcast(android.content.Intent)):

* You know that the data you are broadcasting won't leave your app, so don't need to worry about leaking private data.
* It is not possible for other applications to send these broadcasts to your app, so you don't need to worry about having security holes they can exploit.
* It is more efficient than sending a global broadcast through the system.

**Content Providers:**

It will share the data between applications.

* Content provider is one of the primary building blocks of Android applications, which manages access to a central repository of data. It acts as a standard interface that connects data in one process with code running in another process. So it can be used to share the data between different applications.
* A content provider manages access to a central repository of data. A provider is part of an Android application.
* They handle data and database management issues.
* A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access.



public class My Application extends ContentProvider {

}

A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content using insert(), update(), delete(), and query() methods. In most cases this data is stored in an **SQlite** database.

***Structure of a Content URI:****content://authority/optionalPath/optionalID*

* **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.

**List the various storages that are provided by Android.**

The various storage provided by android are:

* **Shared Preferences**: to save primitive data in key-value pairs
* **Internal Storage**: Internal storage is the storage of the private data on the device memory. By default, these files are private and are accessed by only your application and get deleted, when user delete your application.
* **External Storage**: Data on external storage cannot be private as it is public
* **SQLite Databases**:
  1. SQLite Database is an open-source and light weight database provided in Android which is used to store data inside the user's device in the form of a Text file. such as storing, manipulating or retrieving persistent data from the database.
  2. SQLite does not require a server to run. Hence, it is serverless.
  3. It does not supports XML format.
  4. Android comes in with built in SQLite database implementation.
  5. SQLite supports all the relational database features. In order to access this database.
* Network Connection

### Name the dialog box which is supported by Android?

* Alert Dialog
* Progress Dialog
* Date Picker Dialog
* Time picker Dialog

##### **What is the significance of the .dex files?**

##### Android programs are compiled into .dex (Dalvik Executable) files, which are in turn zipped into a single .apk file on the device. .dex files can be created by automatically, translating compiled applications written in the Java programming language.

**Android SDK:**

It is a collection of software development tools and libraries required to develop **Android** applications.

Latest SDK version: 31.0.0

**JSON:**

It is used to interchange data from the server to the desired place. XML parsing is more complex as compared to JSON parsing. Apart from this, JSON is light weighted, structured and is an independent data exchange format that is used to parse data.

* JSONObject
* JSONArray
* JSONStringer
* JSONTokenizer

[**Support different screen sizes**](https://developer.android.com/training/multiscreen/screensizes)**:**

Android devices come in all shapes and sizes, so your app's layout needs to be flexible. That is, instead of defining your layout with rigid dimensions that assume a certain screen size and aspect ratio, your layout should gracefully respond to different screen sizes and orientations.

* [Multi-window support](https://developer.android.com/guide/topics/ui/multi-window) describes how to support your app in split-screen and picture-in-picture modes.
* 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.

### Avoid hard-coded layout sizes

To ensure that your layout is flexible and adapts to different screen sizes, you should use "w`rap\_content" and "match\_parent" for the width and height of most view components, instead of hard-coded sizes.

"wrap\_content" tells the view to set its size to whatever is necessary to fit the content within that view."match\_parent" makes the view expand to as much as possible within the parent view.

**Different Types of Ad Formats**

* [Banner Ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-banner-ads-in-app)
* [Interstitial Ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-interstitial-ads-in-app)
* [Rewarded Video Ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-rewarded-video-ads-in-app)

Sigup admob account

Integrate ads sdk in build.gradle file

implementation 'com.google.android.gms:play-services-ads:20.3.0'

Add your AdMob app ID ([identified in the AdMob UI](https://support.google.com/admob/answer/7356431)) to your app's AndroidManifest.xml file

**<application>**

**<meta-data  
            android:name="com.google.android.gms.ads.APPLICATION\_ID"  
            android:value="ca-app-pub-xxxxxxxxxxxxxxxx~yyyyyyyyyy"/>**

**</application>**

**Initialize mobile ad sdk in activity.**

[MobileAds.initialize()](https://developers.google.com/android/reference/com/google/android/gms/ads/MobileAds" \l "initialize(android.content.Context,%20com.google.android.gms.ads.initialization.OnInitializationCompleteListener)) which initializes the SDK and calls back a completion listener once initialization is complete .

MoibileAds.initialize(this, R.string.admob\_app\_id);

## **Banner Ads**

[Banner ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-banner-ads-in-app) are a rectangular image or text ads that occupy a spot within an app's layout. They stay on screen while users are interacting with the app, and can refresh automatically after a certain period of time. If you're new to mobile advertising, they're a great place to start.

## **Interstitial Ads**

[Interstitial Ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-interstitial-ads-in-app) are full-screen ads that cover the interface of an app until closed by the user. They're best used at natural pauses in the flow of an app's execution, such as in between levels of a game or just after completing a task.

## **Rewarded Video Ads**

[Rewarded video ads](https://www.tutlane.com/tutorial/android/android-integrate-admob-rewarded-video-ads-in-app) are full-screen video ads and it rewards the users for watching the ads.

**Push Notification:**

A notification is a message you can display to the user outside of your application's normal UI. You can create your own notifications in android very easily.

Push notifications display content such as text and images, and can alert users by playing a sound.

**Firebase Cloud Messaging:**

It is a real-time solution for sending notifications to client app.Though FCM also allows sending out notifications using an app server, here Firebase admin SDK is used

To use Iterable to send push notifications to an Android app, the following things must happen:

* Your users must install your mobile app.
* Your mobile app must store a unique Firebase token on the user's Iterable profile.
* Iterable must contact Firebase, telling it what content to send and which devices to send it to.

**Approach:**

**Step 1: Add Firebase to the project and the required internet permissions.**

**In the app-level Gradle file, add a dependency on the FCM direct boot support library:**

**implementation 'com.google.firebase:firebase-messaging-directboot:20.2.0'**

**Step 2: Create the message receiving class.**

A class which extends the **FirebaseMessagingService**

Add the following code to the **AndroidManifest.xml** file to recognise the FirebaseMessagingService as a service in the app.

**<service android:name=".FirebaseMessageReceiver">**

**<intent-filter>**

**<action android:name="com.google.firebase.MESSAGING\_EVENT" />**

**</intent-filter>**

**</service>**

**Step 3: Working with FirebaseMessageReceiver.java class**

FirebaseMessageReceiver.java class overrides the onMessageReceived() method to handle 2 events:

* If Notification contains any data payload, i.e it is received from the app server.
* If Notification contains any notification payload, i.e. it is sent via the Firebase Admin SDK.

Code snippet:

PendingIntent pendingIntent

                = PendingIntent.getActivity(

**this**, 0, intent,

                PendingIntent.FLAG\_ONE\_SHOT);

        // Create a Builder object using NotificationCompat

        // class. This will allow control over all the flags

        NotificationCompat.Builder builder

                = **new** NotificationCompat

                .Builder(getApplicationContext(),

                channel\_id)

                .setSmallIcon(R.drawable.gfg)

                .setAutoCancel(**true**)

                .setVibrate(**new** **long**[]{1000, 1000, 1000,

                        1000, 1000})

                .setOnlyAlertOnce(**true**)

                .setContentIntent(pendingIntent);

  // Create an object of NotificationManager class to

        // notify the

        // user of events that happen in the background.

        NotificationManager notificationManager

                = (NotificationManager) getSystemService(

                Context.NOTIFICATION\_SERVICE);

        notificationManager.notify(0, builder.build());

**Step 4: Send the notification using FCM**

* Go to Firebase console and choose the appropriate project.
* Choose **Cloud Messaging**.
* Choose **Send your First Message**.
* Click on **Review** and then **Publish**.

**Activity Hierarchy Diagram**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * [java.lang.Object](https://developer.android.com/reference/java/lang/Object) | | | | |
| ↳ | [android.content.Context](https://developer.android.com/reference/android/content/Context) | | | |
|  | ↳ | [android.content.ContextWrapper](https://developer.android.com/reference/android/content/ContextWrapper) | | |
|  |  | ↳ | [android.view.ContextThemeWrapper](https://developer.android.com/reference/android/view/ContextThemeWrapper) | |
|  |  |  | ↳ | android.app.Activity |
| Known direct subclasses  [AccountAuthenticatorActivity](https://developer.android.com/reference/android/accounts/AccountAuthenticatorActivity), [ActivityGroup](https://developer.android.com/reference/android/app/ActivityGroup), [AliasActivity](https://developer.android.com/reference/android/app/AliasActivity), [ExpandableListActivity](https://developer.android.com/reference/android/app/ExpandableListActivity), [ListActivity](https://developer.android.com/reference/android/app/ListActivity), [NativeActivity](https://developer.android.com/reference/android/app/NativeActivity) | | | | |

|  |
| --- |
| Known indirect subclasses  [LauncherActivity](https://developer.android.com/reference/android/app/LauncherActivity), [PreferenceActivity](https://developer.android.com/reference/android/preference/PreferenceActivity), [TabActivity](https://developer.android.com/reference/android/app/TabActivity)  **Service Hierarchy Diagram**  android.app.Service is subclass of ContextWrapper class.  **android.view.ContextWrapper**  **^ |**  **android.app.Service** |
| **indirect Direct subclasses of Services:**  InputMethodService |

**Margin** specifies the space left on four sides in the layout and **padding** specifies the exact position where the element going to be taking place in the layout.

**Logging Mechanism in Android**

**Note:**

* **Debug**

d(String tag, String msg)

Send a DEBUG log message.

e.g. Log.d("MainActivity:","debug");

* **Error**

e(String tag, String msg)

Send an ERROR log message**.**

**e.g.** Log.e("MainActivity:","error");

* **Info**

i(String tag, String msg)

Send an INFO log message.

e.g. Log.i("MainActivity:","info");

* **Verbose**

v(String tag, String msg)

Send a VERBOSE log message.

e.g. Log.v("MainActivity:","verbose");

* **Warn**

w(String tag, String msg)

Send a WARN log message.

e.g. Log.w("MainActivity:","warn");

* **What a Terrible Failure**

wtf(String tag, String msg)

What a Terrible Failure: Report a condition that should never happen

e.g. Log.wtf("MainActivity:","wtf");

**Difference between handler and thread.**

**Handler**: 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 (Update UI).

Handler handler = new Handler();

new Thread(new Runnable() {

public void run() {

handler.post(new Runnable() {

public void run() {

}

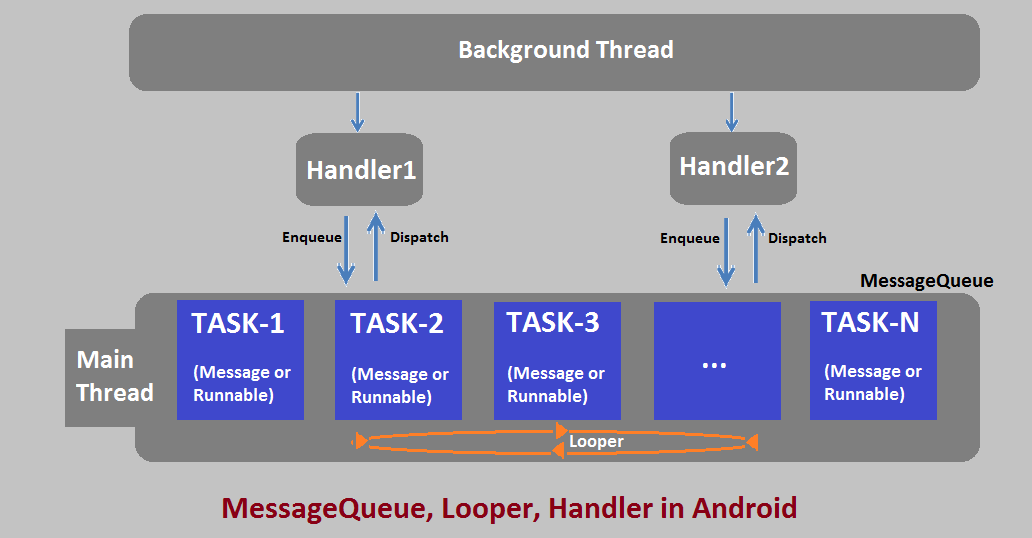
}

}

**Why Handler**

If you need to update the UI from another main Thread, you need to synchronize with the main thread.

**How Handler works**



use **post()** when you want to execute some code on the UI Thread without having to know anything about your Handler object

* **Post()** − it going to post message from background thread to main thread using looper.
* **sendmessage()** − if you want to organize what you have sent to ui (message from background thread) or ui functions. you should use sendMessage().
* **Looper**: A Looper is a class used to loop through the Message Queue attached to the Thread. By default, a thread halts when the execution completes.

**Main Thread:**

Android handles all the UI operations and input events from one single thread which is known as called the Main or UI thread.

**Thread**: Thread is the parent of both AsyncTask and Handler. They both internally use thread, which means you can also create your own thread model like AsyncTask and Handler.

class PrimeThread extends Thread {

long minPrime;

public void run() {

// compute primes larger than minPrime

 . . .

}

}

new Thread(new Runnable() {

@Override

public void run() {

}

}).start();

**runOnUiThread()**

use runOnUiThread() when you want to update your UI from a Non-UI Thread. For eg- If you want to update your UI from a background Thread. You can also use Handler for the same thing.

Sometimes Main thread performs some heavy operations. if user wants to add some extra operations on UI, it will get load and provides ANR. Using runOnUiThread going to do back ground operations on worker thread and update the result on main thread.

new Thread() {

               public void run() {

                  while (i++ < 1000) {

                     try {

                        runOnUiThread(new Runnable() {

                           @Override

                           public void run() {

                              textView.setText("#" + i);

                           }

                        });

                        Thread.sleep(300);

                     } catch (InterruptedException e) {

                        e.printStackTrace();

                     }

                  }

               }

            }.start();

**AsyncTask:** AsyncTask going to do background operation on background thread and update on main thread. It also supports reporting progress of the running tasks.

Android AsyncTask is an abstract class provided by Android which gives us the liberty to perform heavy tasks in the background and keep the UI thread light thus making the application more responsive.

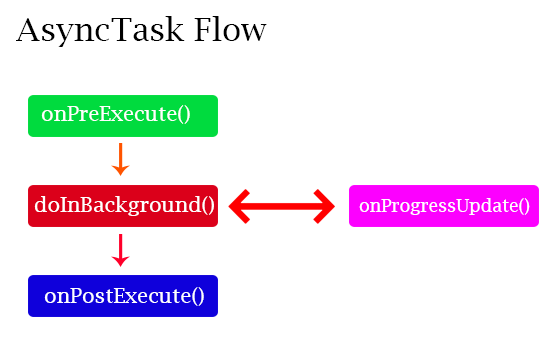
use of AsyncTask in android application keeps the UI thread responsive at all times.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

asynctask help us to make communication between background thread to main thread.

Note:

* The AsyncTask instance must be created and invoked in the UI thread.
* The methods overridden in the AsyncTask class should never be called. They’re called automatically
* AsyncTask can be called only once. Executing it again will throw an exception



## Methods of AsyncTask

* **onPreExecute()** − Before doing background operation we should show something on screen like progressbar or any animation to user. we can directly comminicate background operation using on doInBackground() but for the best 0practice, we should call all asyncTask methods .
* **doInBackground(Params)** − In this method we have to do background operation on background thread. Operations in this method should not touch on any mainthread activities or fragments.
* **onProgressUpdate(Progress…)** − While doing background operation, if you want to update some information on UI, we can use this method.
* **onPostExecute(Result)** − In this method we can update ui of background operation result.

By default, our application code runs in our main thread and every statement is therefore execute in a sequence. If we need to perform long tasks/operations then our main thread is blocked until the corresponding operation has finished. For providing a good user experience in our application we need to use AsyncTasks class that runs in a separate thread.

To use AsyncTask you must subclass it. The parameters are the following

**AsyncTask** <Params, ProgressValue, ResultValue>

1. Params: Params is the type of the parameters sent to the task upon execution.

2. ProgressValue: Progress is the type of the progress units published during the background computation.

3. ResultValue: ResultValue is the type of the result of the background computation.

private class AsyncTaskExample extends AsyncTask<String, Integer, List<String>> {

protected void **onPreExecute**(){

setProgressBarIndeterminateVisibility(true);

progressDialog = new ProgressDialog(MediaActivity.this);

progressDialog.setCancelable(true);

progressDialog.setMax(100);

progressDialog.setMessage("Uploading... media to cloud");

progressDialog.show();

}

/\* Do the task in background/non UI thread

\*/

protected List<String> **doInBackground**(String... params) {

int count = params.length;

List<String> taskList = new ArrayList<>(count);

for (int i = 0; i < count; i++) {

String currentTask = params[i];

taskList.add(currentTask);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

publishProgress((int) (((i + 1) / (float) count) \* 100));

try { StorageUtils.uploadFile(StorageConstants.BUCKET\_NAME, uri, mContext);

} catch (Exception e) {

Log.d("Failure", "Exception: " + e.getMessage());

e.printStackTrace();

}

}

return taskList;

}

// After each task done

protected void **onProgressUpdate**(Integer... values) {

progressDialog.setProgress(values[0]);

}

@Override

protected void **onPostExecute**(List<String> result){

super.onPostExecute(result);

Utils.createToast("Upload complete!", MediaActivity.this);

progressDialog.dismiss();

}

**What is the variance between a regular Bitmap and a Nine-Patch image?**

A Nine-patch image allows resizing. 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.

**Difference between Service, Thread and AsyncTask:**



**What are the exemptions in Android?**

InflateException : When an 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 on 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.

**Which tool used to build encrypted and unencrypted apk expansion files in OBB format**

The jobb tool allows you to build encrypted and unencrypted APK expansion files in Opaque Binary Blob (OBB) format

**What are the four essential states of an activity?**

Active – If the Activity is in the foreground.

Paused – If the Activity is in the background and is still visible.

Stopped – If the Activity is not visible, therefore is hidden or concealed by another Activity.

Destroyed – When the Activity process is completed terminated.

**What are the advantages of native apps over hybrid apps?**

* They work efficiently as they are built for that specific platform.
* Native apps are responsive on all the platform-specific devices.
* They are very fast and the best in the app performance
* Native apps better integrate with mobile hardware
* They have interactive and intuitive User Interface (UI) and User Experience (UX) as per the user expectations based on specific platforms.
* Some of the Native mobile apps work even without the Internet connection
* Native apps are secured and reliable
* They can easily access or utilize the other device-specific capabilities like GPS, Camera, Contacts, etc.

**Context**

* The context in Android is the context of the current state of the application or object
* Context is the Base Object
* it provides services like resolving resources, obtaining access to databases and preferences, and so on.

There are two types of context.

**Application context:**

* This application context is attached to the lifecycle of an application.
* The application context should be used where you need a context whose lifecycle is separate from the current context or when you are passing a context beyond the scope of activity.
* An application context lasts, as long as your app is alive.

**Activity context**

* This activity context is attached to the lifecycle of an activity.
* The activity context can be used when you are passing the context in the scope of an activity or you need the context whose lifecycle is attached to the context of the activity.
* the Activity context dies with your Activity (it is not valid after onDestroy() of that Activity).

**Uses of third-party library in android**

* [Retrofit2](https://github.com/square/retrofit) - REST APIs.

Retrofit library is different from other network libraries because it gives us an easy to use platform through which we don’t need to parse JSON responses as they are done by library itself. It used GSON library in the background to parser the response data. What we need to do is define a POJO (Plain Old Java Object) to parse the response.

We can see the difference in performance that for one discussion means for one network request and response they will take how much time.

1. **AsyncTask**:

one(1) discussion: 941 ms

Seven(7) discussions: 4539 ms

Twenty Five(25) discussions: 13957 ms

2. **Volley**:

one(1) discussion: 560 ms

Seven(7) discussions: 2202 ms

Twenty Five(25) discussions: 4275 ms

3. **Retrofit**:

one(1) discussion: 312 ms

Seven(7) discussions: 889 ms

Twenty Five(25) discussions: 1059 ms

Retrofit automatically serialises the JSON response using a POJO(Plain Old Java Object) which must be defined in advanced for the JSON Structure. To serialise JSON we need a converter to convert it into Gson first.

Retrofit is a type-safe REST client for Android. The library provides a powerful framework for authenticating and interacting with APIs and sending network requests with OkHttp.

**1.Add dependency file in gradle:**

Dependency in gradle : implementation 'com.squareup.retrofit2:retrofit:2.1.0'

**2. Create POJO/Model Class:**

**3.** **Create** **API Interface :**

**4.** **Create** **RestAdapter:**

* [OkHttp3](https://github.com/square/okhttp) - implementing interceptor, logging web server.

is a third-party library developed by Square for sending and receive HTTP-based network requests.

Dependency in gradle : implementation 'com.squareup.okhttp3:okhttp:3.14.6'

* [Glide](https://github.com/bumptech/glide) – Image loader library i.e for loading images. Glide is a fast and efficient open-source media management and image loading framework for Android that wraps media decoding.
  + focused on smooth scrolling.

Dependency in gradle : implementation 'com.github.bumptech.glide:glide:4.12.0'

* [Lottie](https://github.com/airbnb/lottie-android) - implementing animations.
* Picasso: Android Picasso is a powerful image downloading and caching library. It is an image loading/processing library.

Dependency in gradle : implementation 'com.squareup.picasso:picasso:2.5.2'

**Code snippet:**

Picasso

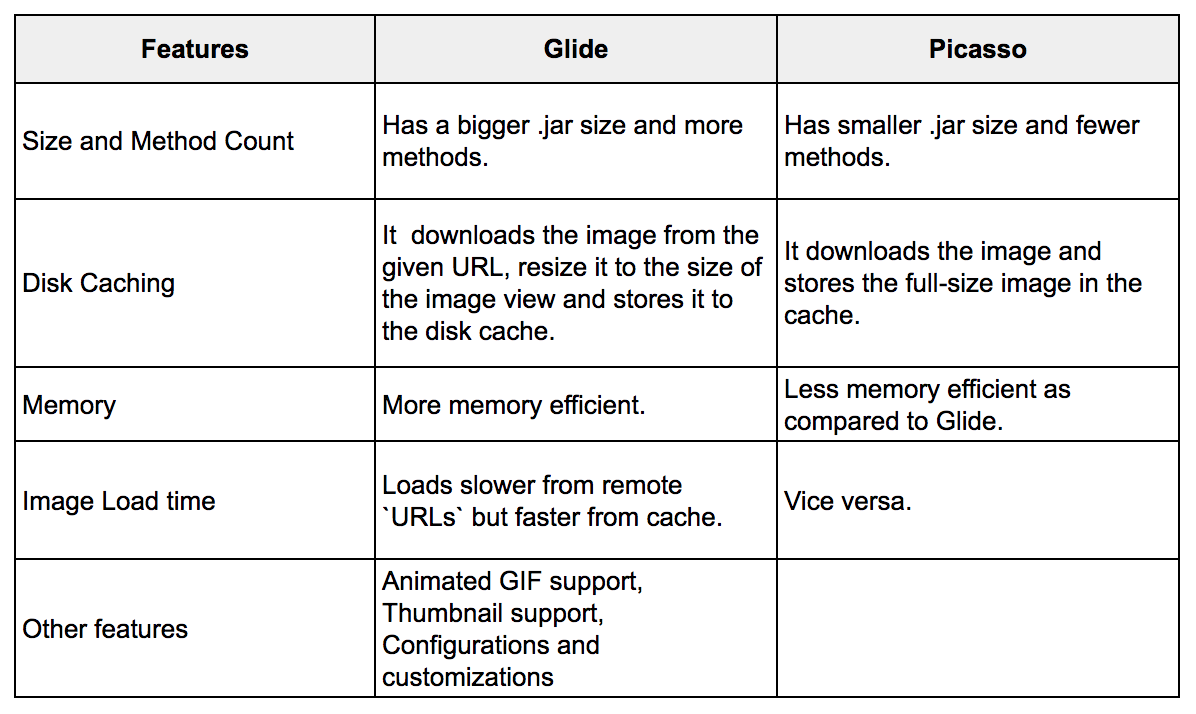
.with(this)

.load("https://res.cloudinary.com/demo/video/upload/dog.png")

.into(imageView);

* Volley: it is an HTTP library that makes networking very easy and fast

Dependency in gradle : implementation 'com.android.volley:volley:1.0.0'



**Memory Management:**

1.Glide is memory efficient as compared to picasso becuase glide loading same image according to size of view but picsso loads full size image.

2.And if you want to prevent from OutOfMemoryError excecption we should go for glide.

**Conclusion:**

If you want small use of image you should go for Picasso but if you have lot of images use and memory management you should go for glide.

### Butterknife: [ButterKnife](http://jakewharton.github.io/butterknife/) is a view binding library.

### that helps us assign ids to views easily thereby avoiding the excess findViewById.

**Firebase:**

* It is serverless and is a Backend-as-a-Service, and it is a real-time database which is basically designed for mobile applications.
* Firebase manages real-time data in the database. So, it easily and quickly exchanges the data to and from the database.
* Firebase provides integration to Google Advertising, AdMob, Data Studio, BigQuery DoubleClick, Play Store.
* Everything from databases, analytics to crash reports are included in Firebase. So, the app development team can stay focused on improving the user experience.
* Firebase applications can be deployed over a secured connection to the firebase server.
* It has JSON storage, which means no barrier between data and objects.
* It is highly secure.

**Install APK on Specific Device if more that one emulator is connected**

adb -s <DEVICE ID> install <PATH TO APK>

e.g. adb -s 5555 install demo.apk.

Android Directory Structure

Src ---

* androidTest
* main
  + assets
  + java
    - class
  + res
    - anim
    - drawable
    - layout
    - menu
    - values
    - color
    - font
    - navigation
    - xml
  + AndroidMainifest.xml
* Test

build.gradle

proguard-rules.pro

**attrs.xml:**

We can define attributes in the top <resources> element or inside of a <declare-styleable> element. Note, all attributes share the same global namespace. That means that even if you create a new attribute inside of a <declare-styleable> element it can be used outside of it and you cannot create another attribute with the same name of a different type.

An <attr> element has two xml attributes name and format. name lets you call it something and this is how you end up referring to it in code, e.g., R.attr.my\_attribute. The format attribute can have different values depending on the 'type' of attribute you want.

* reference - if it references another resource id (e.g, "@color/my\_color", "@layout/my\_layout")
* color
* boolean
* dimension
* float
* integer
* string

**enum** attributes can be defined as follows:

<attr name="my\_enum\_attr"> <enum name="value1" value="1" /><enum name="value2" value="2" /> </attr>

**JobScheduler:**

The JobSchedular API is used for scheduling different types of jobs against the framework that will be executed in your app’s own process.

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

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

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

**Android 11.0 (API 30):** Android developers are continuously working to provide more advanced applications as per the user requirements. Important features included are:

* Native screen recording.
* Auto revokes app permission.
* Mute notifications during the video.
* Increase in touch sensitivity.

**Lint:**

Lint is a code scanning tool provided by the Android Studio to identify, suggest and correct the wrong or the risky code present in the project.

**Dark-Theme/ Dark Mode in android:**

if (isChecked) {

AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE\_NIGHT\_YES); recreate()

} else {

AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE\_NIGHT\_NO)

}

AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE\_NIGHT\_YES);

setTheme(R.style.darkTheme);

**Android Studio IDE Feature and Use:**

**How to resolve layout overlapping issue:**

[Layout Inspector](https://developer.android.com/studio/debug/layout-inspector#layout-inspector) allows you to compare your app layout with design mockups, display a magnified or 3D view of your app.

[Layout Validation](https://developer.android.com/studio/debug/layout-inspector#layout-validation) allows you to simultaneously preview layouts on different devices and display configurations, including variable font sizes or user languages,

**Database Inspector** allows you to inspect, query, and modify your app's databases while your app is running useful for database debugging.

**Android Profiler in Android Studio Usage:**

 It gives you real-time information on how your app consumes CPU, memory, network, and battery resource.

* **CPU profiler**: This tool helps track down runtime performance issues.
* **Memory profiler**: This tool helps track memory allocations.
* **Network profiler**: This tool monitors network traffic usage.
* **Energy profiler**: This tool tracks energy usage, which can contribute to battery drain.

**Android Studio components:**

**Flavors:** Lets you create multiple build flavors, where each flavor specifies a set of configuration settings, such as the module's minimum and target SDK version, and the [version code and version name](https://developer.android.com/tools/publishing/versioning).

**Build Types:** Lets you create and modify build configurations, as described in [Configuring Gradle Builds](https://developer.android.com/tools/building/configuring-gradle). By default, every module has debug and release build types, but you can define more as needed.They are for building/compiling the **SAME CODE** to generate different APKs.

**Dependencies:** Lists the library, file, and module dependencies for this module. You can add, modify, and delete dependencies from this pane.

**.gitignore:** file specifies which files/directories to be ignored by Git.

**build.gradle:** Allows you to customise properties for build system such as setting location of your keystore used for signing the app-release.apk.

**How to share the data to from one application other application**

intent.createChooser(share, "Share Image");

**How to keep backup if database connection lost :**

Create replicas of the database for backup.

**RxJava (ReactiveJava):**

It is an implementation and enhancement of the observer pattern.

RxJava is a JVM library for doing asynchronous and executing event-based programs 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.

Handles threading, synchronization, thread-safety and concurrent data structures internally.

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

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

**Observable**: Observable is a data stream that do some work and emits data.

Observable provides data once subscriber starts listening.

**Observer**: It subscribes to Observable's data of sequence and reacts per item of the observables. Observers are notified whenever Observable emits a data. An Observer handles data one by one. It receives the data emitted by Observable.’

The **map** operator transforms items emitted by an Observable by applying a function to each item.

Observable.from(letters) .map(String::toUpperCase) .subscribe(letter -> result += letter);

The **flatMap** can be used to flatten Observables whenever we end up with nested Observables.

Observable.just("book1", "book2") .flatMap(s -> getTitle()) .subscribe(l -> result += l);

The operator **filter** emits only those items from an observable that pass a predicate test.

let's filter in an integer array for the odd numbers:

Observable.from(numbers)

.filter(i -> (i % 2 == 1))

.subscribe(i -> result += i);

## **Subscriber**

* Observable can have multiple subscribers.
* When an Observable emits an item, each subscriber **onNext**() method gets invoked.
* When an Observable finished emitting items, each subscriber **onComplete**() method gets invoked.
* If an Observable emits error, each subscriber **onError**() method gets invoked

Following are the base classes to create observables.

* **Flowable** − 0..N flows, Emits 0 or n items. Supports Reactive-Streams and back-pressure.
* **Observable** − 0..N flows ,but no back-pressure.
* **Single** − 1 item or error. Can be treated as a reactive version of method call.

**The Basic Steps**

1. Create an Observable that emits data. Below we have created an Observable that emits list of animal names. Here just() operator is used to emit few animal names.

Observable<String> animalsObservable = Observable.just("Ant", "Bee", "Cat", "Dog", "Fox");

2. Create an Observer that listen to Observable. Observer provides the below interface methods to know the the state of Observable.

onSubscribe(): Method will be called when an Observer subscribes to Observable.

onNext(): This method will be called when Observable starts emitting the data.

onError(): In case of any error, onError() method will be called.

onComplete(): When an Observable completes the emission of all the items, onComplete() will be called.

Observer<String> animalsObserver = getAnimalsObserver();

private Observer<String> getAnimalsObserver() {

return new Observer<String>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "onSubscribe");

}

@Override

public void onNext(String s) {

Log.d(TAG, "Name: " + s);

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: " + e.getMessage());

}

@Override

public void onComplete() {

Log.d(TAG, "All items are emitted!");

}

};

}

3. Make Observer subscribe to Observable so that it can start receiving the data. Here, you can notice two more methods, observeOn() and subscribeOn().

subscribeOn(Schedulers.io()): This tell the Observable to run the task on a background thread.

observeOn(AndroidSchedulers.mainThread()): This tells the Observer to receive the data on android UI thread so that you can take any UI related actions.

animalsObservable

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(animalsObserver);

**Dagger 2**:

It is a static compile-time android dependency injection framework that uses Java and kotlin.

It should actually be Dagger, Dagger2 simply implies the second version which was a complete re-write of the DI framework.

With Dagger, you don't have to write tedious and error-prone boilerplate code.

Use constructor injection with @Inject to add types to the Dagger graph whenever it's possible. When it's not:

Use @Binds to tell Dagger which implementation an interface should have.

Use @Provides to tell Dagger how to provide classes that your project doesn't own.

You should only declare modules once in a component.

Name the scope annotations depending on the lifetime where the annotation is used. Examples include @ApplicationScope, @LoggedUserScope, and @ActivityScope.

Specification Request 330 and Annotations. Some of the basic annotations that are used in dagger 2 are:

@Module This annotation is used over the class which is used to construct objects and provide the dependencies.

@Provides This is used over the method in the module class that will return the object.

@Inject This is used over the fields, constructor, or method and indicate that dependencies are requested.

@Component This is used over a component interface which acts as a bridge between @Module and @Inject. (Module class doesn’t provide dependency directly to requesting class, it uses component interface)

@Singleton This is used to indicate only a single instance of dependency object is created

In Android, you usually create a Dagger graph that lives in your application class because you want an instance of the graph to be in memory as long as the app is running

**Dagger 2 Annotations:**

**@Inject Annotation**

First and the most important for DI is the@Inject annotation.

* Constructor Injection — is used with class constructor
* Field Injection — is used with the fields in the class
* Method Injection — is used with the functions/methods

the@Inject annotation will tell the Dagger what all the dependencies needed to be transferred to the dependant.

public class Starks{

/\*\*

\* Explaining different usage

\* of Inject annotations of dagger

\*\*/

//Feild injection

@Inject

Allies allies;

//Constructor injection

@Inject

public Starks(){

//do something..

}

//Method injection

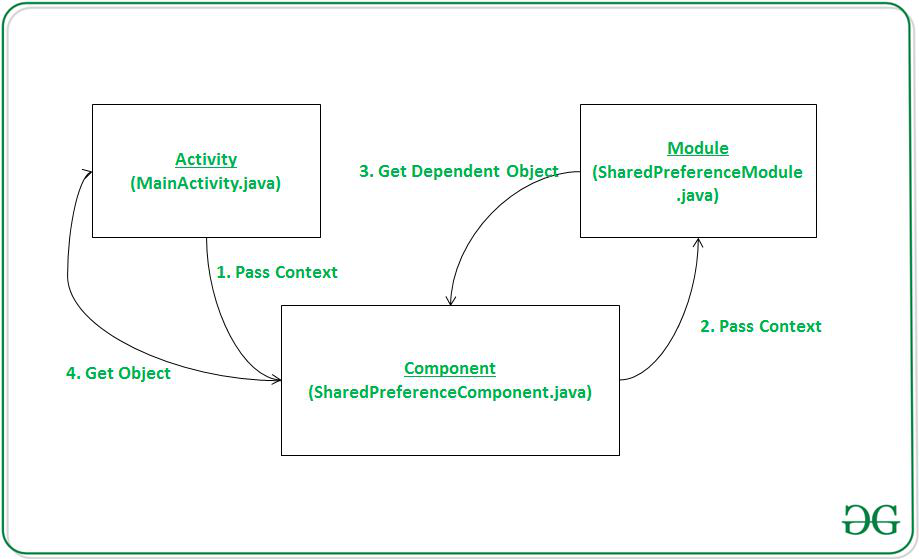
@Inject

private void prepareForWar(){

//do something..

}

}



**Room Database:**

Room is **a database layer on top of an SQLite database**. Room takes care of mundane tasks that you used to handle with an SQLiteOpenHelper . Room uses the DAO to issue queries to its database.

To add a dependency on Room, you must add the Google Maven repository to your project Build.gradle

**roomVersion** = "2.3.0"

We can see database created in android studio:

Go to **Device File Explorer** > **data** > **data** > **app package name** > **databases**

**Also export the room database using the App Inspection > Database Inspector > click on DB > Export as db file in android studio.**

**Advantages of Room Component:**

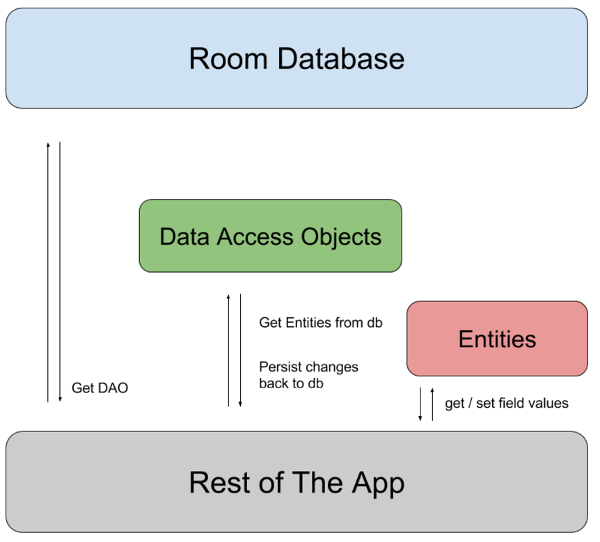
* Reduce boilerplate code
* Simplifies database access mechanism
* Easy to implement migrations
* Test-ability is high

Room provides the following benefits:

* Compile-time verification of SQL queries.
* Convenience annotations that minimize repetitive and error-prone boilerplate code.
* Streamlined database migration paths.

There are three major components in Room:

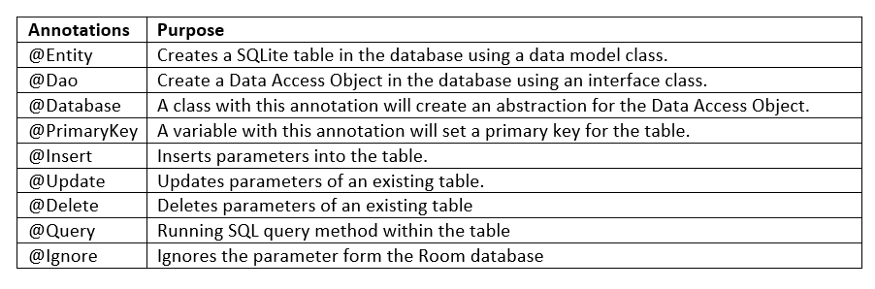
* The [database class](https://developer.android.com/reference/kotlin/androidx/room/Database) that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
* [Data entities](https://developer.android.com/training/data-storage/room/defining-data) that represent tables in your app's database.
* [Data access objects (DAOs)](https://developer.android.com/training/data-storage/room/accessing-data) that provide methods that your app can use to query, update, insert, and delete data in the database.



**Steps to implement Room Database in android**

* Add depedency room+coroutines+livedata+viewmodel in build.gradle
* Create room user model with Entity
* Create a new interface for DAO (known as Data Access Object in Room):
* Create a new class for database by extending it with RoomDatabase and define an INSTANCE object for it:
* Create repository to fetch the data from the database.
* Create a ViewModel Class:( Do action create (insert data), read (show data), delete (delete data) and update (update data)
* Setup DataBinding listener:
* Call Model class in MainActivity.java:

**Important Annotation:**



@Entity  
public class User {  
    @PrimaryKey  
    public int uid;  
  
    @ColumnInfo(name = "first\_name")  
    public String firstName;  
  
    @ColumnInfo(name = "last\_name")  
    public String lastName;  
}

### Data access object (DAO)

The following code defines a DAO called UserDao. UserDao provides the methods that the rest of the app uses to interact with data in the user table.

@Dao  
public interface UserDao {  
    @Query("SELECT \* FROM user")  
    List<User> getAll();  
  
    @Query("SELECT \* FROM user WHERE uid IN (:userIds)")  
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT \* FROM user WHERE first\_name LIKE :first AND " +  
           "last\_name LIKE :last LIMIT 1")  
    User findByName(String first, String last);  
  
    @Insert  
    void insertAll(User... users);  
  
    @Delete  
    void delete(User user);  
}

**Database**

The following code defines an AppDatabase class to hold the database. AppDatabase defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

* The class must be annotated with a [@Database](https://developer.android.com/reference/kotlin/androidx/room/Database) annotation that includes an [entities](https://developer.android.com/reference/kotlin/androidx/room/Database#entities) array that lists all of the data entities associated with the database.
* The class must be an abstract class that extends [RoomDatabase](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase).
* For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.
* @Database(entities = {User.class}, version = 1)  
  public abstract class AppDatabase extends RoomDatabase {  
      public abstract UserDao userDao();  
  }

Purpose of **proguard-rules.pro:**

ProGuard is a java tool to help shrink(Remove unused code in the project), obfuscate,( Rename the names of class, fields, etc) and optimize your code. It is not only especially useful for reducing the overall size of your Android application as well as removing unused classes and methods that contribute towards the intrinsic 64k method limit of Android application.

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

It reduces the size of the application.

It removes the unused classes and methods that contribute to the 64K method counts limit of an Android application.

It makes the application difficult to reverse engineer by obfuscating the code.

ProGuard can be enabled by using the minifyEnabled option for any build type.

buildTypes **{**

dev **{**

minifyEnabled **true** *// enables ProGuard*

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

**}**

“**shrinkResources**” attribute will remove all the resources, those are not used anywhere in the project. Enable this in your build.gradle file by adding below line:

{ shrinkResources true }

**Purpose of debuggable in build.gradle file:**

Android allows the attribute android:debuggable to be set to true in the manifest, so that the app can be debugged. ... With the attribute set to true , users can debug the app even without access to its source code.

buildTypes {

debug {

debuggable true

}

}

**How to add dependency in build.gradle file:**

Using following attribute

**Dependencies {**

Implementation **‘lib-name:version’**

**}**

**What is compileSdkVersion ?**

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, obviously.

**What is targetSdkVersion?**

**targetSdkVersion** is a property that tells the system for which Android version the app was designed and tested on.

If the user runs your app on a device with an android version that is higher than the targetSdkVersion defined in your app, for new android features, the system may introduce some backwards-compatibility behaviour to ensure your app still looks and works in a way that you designed it.

targetSdkVersion cannot be higher than the compileSdkVersion simply because we cannot target things that we know nothing about during compilation.

targetSdkVersion <= compileSdkVersion.

Ideally, the **compileSdkVersion** and **targetSdkVersion** should be equal and both point to the latest SDK. But of course, only after you test that every change introduced in that version works smoothly with your app!

**LiveData:**

LiveData is an observable data holder class that is used to observe the changes of a ViewModel and update those changes.

LiveData is lifecycle-aware, which means that whenever data is updated or changed, the changes are only applied to the specific app components that are in an active state.

**Uses of LiveData:**

LiveData works well in resolving mainly two issues:

1.It removes the leaks caused by the interfaces/callbacks that send results to the UI thread. This is a core feature of an MVVM model where callbacks are sent from ViewModel to activity/fragment.

2.It de-couples tight integration between data, mediator, and the UI layers.

**In order to use LiveData in your Android project, follow these steps:**

* Create a LiveData instance in your ViewModel class to hold the data.
* Set the data in LiveData using setValue().
* Return the LiveData using postValue(). Only this data will be observable by the Observer which is in Views i.e., activity or fragment.
* Finally, observe the data with the help of the Observer() function. Inside Observer(), you can define all your necessary changes in the UI that you want to perform when there is a change or update in the data.
* Once you have completed the above steps, any change in the data stored in LiveData will cause all the Observers associated with the LiveData to be notified. This applies only when the app components are live.

### How to create instance of live data using code.

### private MutableLiveData<Integer> mutableLiveData=new MutableLiveData<>();

### Observer<Integer> liveDataObserver=new Observer<Integer>() {

### @Override

### public void onChanged(@Nullable Integer integer) {

### timerTV.setText("Elapsed : "+integer+ " s");

### }

### };

### Types in LiveData

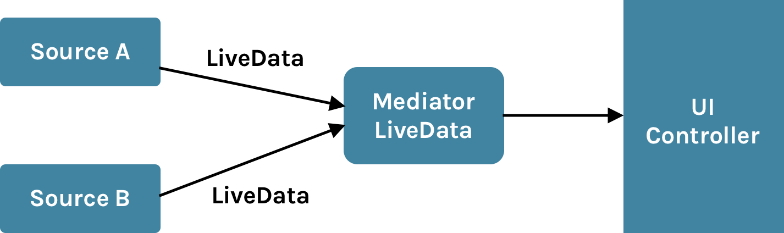
There are subclasses in LiveData that are useful for their properties when updating the UI.

1. LiveData
2. MutableLiveData
3. MediatorLiveData

**LiveData** is immutable by default. By using LiveData we can only observe the data and cannot set the data.

**MutableLiveData** is mutable and is a subclass of LiveData. In MutableLiveData we can observe and set the values using postValue() and setValue() methods (the former being thread-safe) so that we can dispatch values to any live or active observers.

**MediatorLiveData** can observe other LiveData objects such as sources and react to their onChange() events. MediatorLiveData will give us control over when we want to perform an action in particular or when we want to propagate an event.



**Advantages of using LiveData:**

### UI matches with the data state.

### LiveData follows the Observer pattern. In an active state, whenever a component’s data gets changed or updated, the particular component UI matches the state without fail.

### Increased stability of code.

### There are no crashes when activities are stopped. If the app components are inactive those changes are not affected.

* + Memory leaks will be reduced because unwanted events/allocations are automatically handled

### No manual lifecycle handling

### Always up to date with the latest data

### Proper configuration changes/screen orientation

### Sharing resources

**Conclusion:**

Incorporating LiveData into a project significantly reduces boilerplate code for UI updates in the application. The LiveData concept completely mitigates the major problems like orientation change issues.

LiveData also ensures that the UI is always up to date even when the app’s configuration is changed.

**Observable:**

An observable object can have one or more observers. An observer may be any object that implements interface Observer

Observable Data Object refers to the capability of an object to notify others(listeners) about the changes in its data.

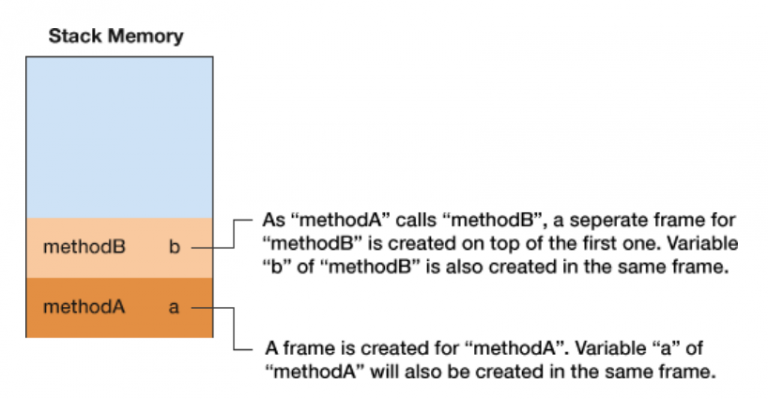
**Overview of memory management**

The Android Runtime (ART) and Dalvik virtual machine use paging and memory-mapping (mmapping) to manage memory.

Java has automatic memory management. It performs routine garbage collection to clean up unused objects and free up the memory. However, it is very important for us to know how the garbage collector works in order to manage the application’s memory effectively. Thus avoiding OutOfMemoryError and/or StackOverflowError exceptions.

**Stack Memory**

Java Stack memory is used for the execution of the thread. They contain method-specific values which that are short-lived and references to the other objects in the heap that are getting referred from the method.



**Heap Memory**

Java heap space is used to allocate memory to the objects. Fp

Garbage collection process runs in the heap memory

**Tools to detect memory leaks**

As discussed, we need to avoid memory leaks to run applications effectively. There are tools to monitor memory usage and detect memory leaks. A few examples include, Android Profiler and Leak Canary.

Android profiler tool provides real-time data to help understand the CPU, Memory, Network and Battery usage of the app. Android profiler is compatible with Android 5.0 and above.

Leak Canary is a memory leak detection library in Android. This library runs along with the app, dumps memory when needed, looks for potential memory leaks and gives a notification with a clean and useful stack trace to find the root cause of the leak.

**Use Android Profiler in Android Studio to Detect Memory Leaks in the Project.**

Step1: Change build type variant to Debug

Step2: Click on Android Profiler

Step3: Selecting Process we want collect data.

Step4: Double-tap the memory line to see recording screen

Step5: Click Capture Heap Dump and then click Record but and wait for recording end

Step6: See if any yellow signs as leaks.

* **Allocations**: Number of objects allocated in this period of time.
* **Deallocations**: Quantity of deallocations in this period of time.
* **Total Count**: Number of objects still allocated of this class.
* **Shallow Size**: Total bytes of memory used for the objects of this class.

**Use LeakCanary lib in Android Studio**

**Step1: Add following dependency in app level build.gradle**

debugImplementation 'com.squareup.leakcanary:leakcanary-android:1.5.1'

**Step2: Add the following code in application class which extends Application in project.**

public void onCreate() {

if (LeakCanary.isInAnalyzerProcess(this)) {

return;

}

LeakCanary.install(this);

}

**Step3: Run/Install the app on device.**

**Step4: Check the dump file generated in Download location from your device. (fileName.hprof)**

**Step5: Open Leak app installed on device and check the leaks if any.**

**How to Avoid Memory Leaks**

Here are some familiar potential leakage that developer needs to keep in mind.

* Remember to close the cursor after querying the database. If you want to keep the cursor open long-term, you must use it carefully and close it as soon as the database task finished.
* Remember to call unregisterReceiver() after calling registerReceiver().
* Avoid Context leakage. If you declare a static member variable “Drawable” in your Activity, and then call view.setBackground(drawable) in onCreate(), after screen rotate, a new Activity instance will be created and the old Activity instance can never be de-allocated because drawable has set the view as callback and view has a reference to Activity (Context
  + There are two ways to avoid this kind of leakage:
    - Do not keep long-lived references to a context-activity. A reference to an activity should have the same life cycle as the activity itself.
    - Try using the context-application instead of a context-activity.
* Be careful about using Threads. Threads in Java are garbage collection roots; that is, the Dalvik Virtual Machine (DVM) keeps hard references to all active threads in the runtime system, and as a result, threads that are left running will never be eligible for garbage collection:
  + Use Loader instead of a thread for performing short-lived asynchronous background queries in conjunction with the Activity lifecycle.
  + Use Service and report the results back to the Activity using a BroadcastReceiver.
  + Use AsyncTask for short-lived operations.

**S.O.L.I.D principles in software development?**

Understanding SOLID Principles will help us write clean and elegant code. It helps us write the code with SOC (Separation of Concerns).

**S - The Single Responsibility Principle (SRP)**

* one class should only have one responsibility.
* Let’s take the OnBindViewHolder method in RecyclerView.Adapter class. The role of the OnBindViewHolder is to map an list item to a view. There should be no logic in this method.

**O - The Open-Closed Principle (OCP)**

* Software entities such as classes, functions, modules should be open for extension but closed for modification.
* if we are required to add a new feature to the project, it is good practice to not modify the existing code but rather write new code that will be used by the existing code.

**L - The Liskov Substitution Principle (LSP)**

* Child classes should never break the parent class’ type definitions.
* This means that a sub class should override the methods from a parent class that does not break the functionality of the parent class.

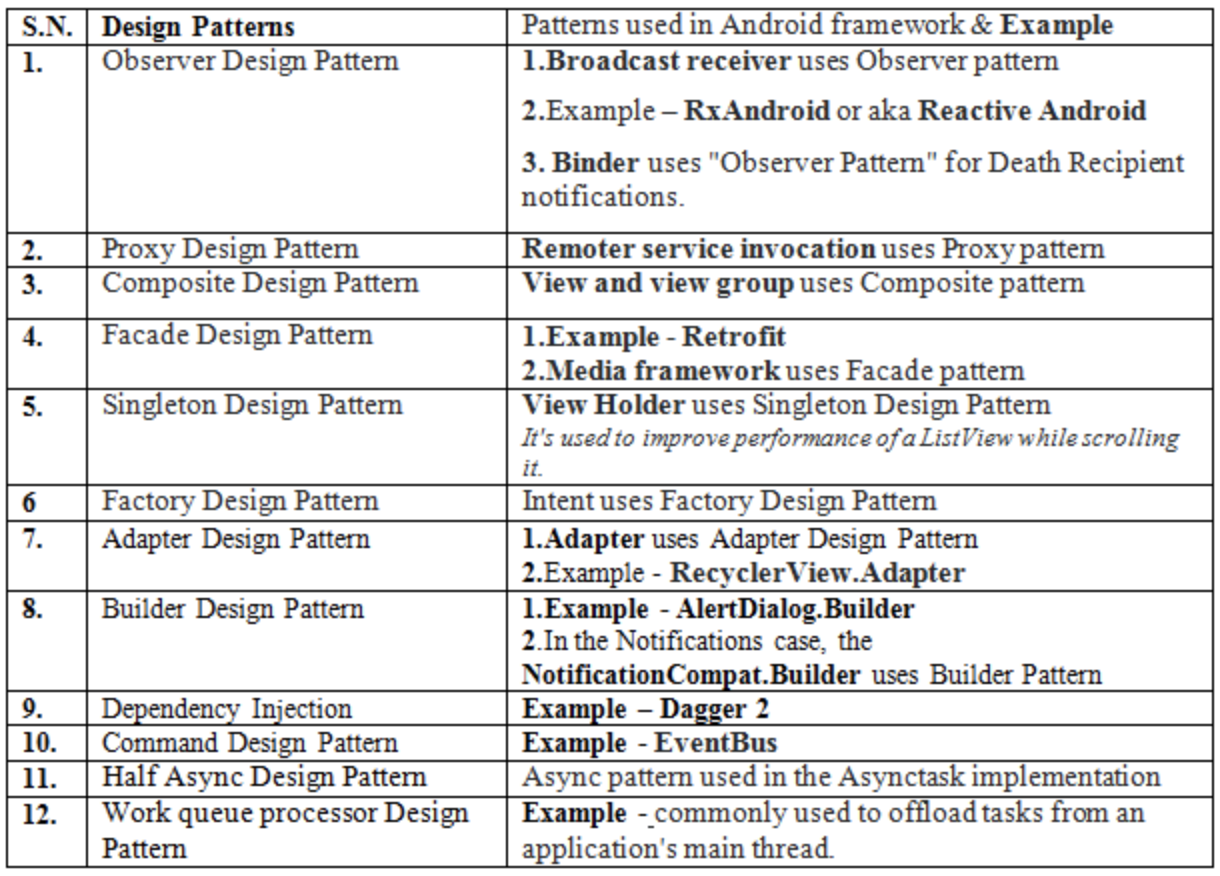
**I - The Interface Segregation Principle (ISP)**

* The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.
* This means that if an interface becomes too fat, then it should be split into smaller interfaces so that the client implementing the interface does not implement methods that are of no use to it.

**D - The Dependency Inversion Principle (DIP)**

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend upon details. Details should depend upon abstractions.
* If you use a class insider another class, this class will be dependent of the class injected.

**Design Pattern in Android:**



**Different ways to debug android application in android studio.**

**Start debugging in android studio**

* Connect device
* Change build variant to debug mode.
* Attach Debugger to Android Process
* Set some breakpoints in the app code.
* Inspect variables
* Add watchpoints
* Use conditional breakpoints
* In the toolbar, click Debug.
* Click the **Step Over** icon  Step Over icon , select **Run > Step Over,** or press **F8**. **Step Over** executes the next line of the code in the current class and method, executing all of the method calls.
* Click the **Step Into**icon  Step Into icon  , select **Run > Step Into,** or press **F7**. **Step Into** jumps into the execution of a method call on the current line.

**Use the system log:**

* **Write log messages in your code. (Log.d(TAG, “Msg”))**

**View the system log:**

**How to share the application to another application?**

The "**exported**" attribute in manifest.xml describes whether or not someone else can be allowed to use it. So if you have "exported=false" on an Activity, no other app, or even the Android system itself, can launch it. **Only you can do** that, from inside your own application.

If you set **exported=true** in the manifest.xmlthen other application can access your application.

**The AsyncTask API is deprecated in Android 11 API Level 30:**

* **Why Android AsyncTask is Deprecated?**

Here is the official reason it is deprecated.

AsyncTask was intended to enable proper and easy use of the UI thread. However, the most common use case was for integrating into UI, and that would cause Context leaks, missed callbacks, or crashes on configuration changes. It also has inconsistent behavior on different versions of the platform, swallows exceptions from doInBackground, and does not provide much utility over using Executors directly.

* **Alternative of AsyncTask**
  + **Kotlin Coroutines**
  + **Using a combination of an Executor and a Handler**

**Using Executer with Handler:**

**ExecutorService mExecutor =** Executors.newSingleThreadExecutor()

executor.execute(new Runnable() {

@Override

public void run() {

// Do something in background (back-end process)

}

});

Handler handler = new Handler(Looper.getMainLooper());

handler.post(new Runnable() {

@Override

public void run() {

// Do something in UI (front-end process)

}

});

Here the executor will execute the task and as soon as the desired results are available, UI changes are made using a handler.

**Note**: If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

AsyncTasks should ideally be used for short operations (a few seconds at the most.)

**Background Execution Limits in Android 8 Oreo:**

Starting from Android Oreo( API 26) and onwards the background applications(when an application is not foreground ) cannot use the started service. When you call startService() method from the background applications simply through the IllegalStateException. In other words, you can say if you call startService() when your application is not in the foreground.

Apps are restricted in two ways:

* [Background Service Limitations](https://developer.android.com/about/versions/oreo/background#services): While an app is idle, there are limits to its use of background services. This does not apply to foreground services, which are more noticeable to the user.
* [Broadcast Limitations](https://developer.android.com/about/versions/oreo/background#broadcasts): With limited exceptions, apps cannot use their manifest to register for implicit broadcasts. any broadcast receivers that we have defined statically within our application manifest that are listening for implicit broadcasts will no longer receive those broadcasts. They can still register for these broadcasts at runtime, and they can use the manifest to register for explicit broadcasts targeted specifically at their app.
  + To avoid background service running into any issues, The possible solution for dealing with Service in Oreo and onward:
    - Job Scheduler
    - Start service in Foreground using **startForegroundService.**
    - JobIntentService (internally used Job Scheduler API )
      * Intent Service gets killed if app is not in foreground, So we can use JobIntentService which is subclass of service
      * Add this **android.permission.WAKE\_LOCK** permission in manifest.xml.
      * Start the service using (class) MyJobIntentService.enquework(this, serviceIntent) (static method); instead of startService(serviceIntent)
      * This JobIntentService has to be stop itself after completing the job by calling stopSelf().
      * This JobIntentService cannot be stopped by stopService.
  + To avoid Broadcast limitation into any issues, there are a number of different approaches,
    - Broadcast whitelist
    - Scheduling jobs
    - Dynamic broadcasts using IntentFilter in activity.

**JobScheduler:**

**E.g.** Want to sync file after every 15 mins but want to do it only if device is connected to internet via Wifi and service to be enabled as soon as device gets booted.

* Using JobIntentService we cannot set the condition for stopping the running service and also cannot set the condition for starting the killed service and can’t stop it explicitly and can’t set the periodicity
* **Solution** : Using JobScehduler we can solve the all problems occurred in JobIntentService.
* JobService is subclass of Service.
  + onStartJob : Boolean for long running task (duration)
  + onStopJob : Boolean : Restart if killed.
  + onDestroy
* Add permission android.permission.RECEIVE\_BOOT\_COMPLETED.