Unit- 1

Introduction to Android?

→ Android is an open source and Linux-based Operating System for mobile devices such as smart phones and tablet computers. Android was developed by the *Open Handset Alliance*, led by Google, and other companies.

→ Android offers a unified approach to application development for mobile devices which means developers need only develop for Android, and their applications should be able to run on different devices powered by Android.

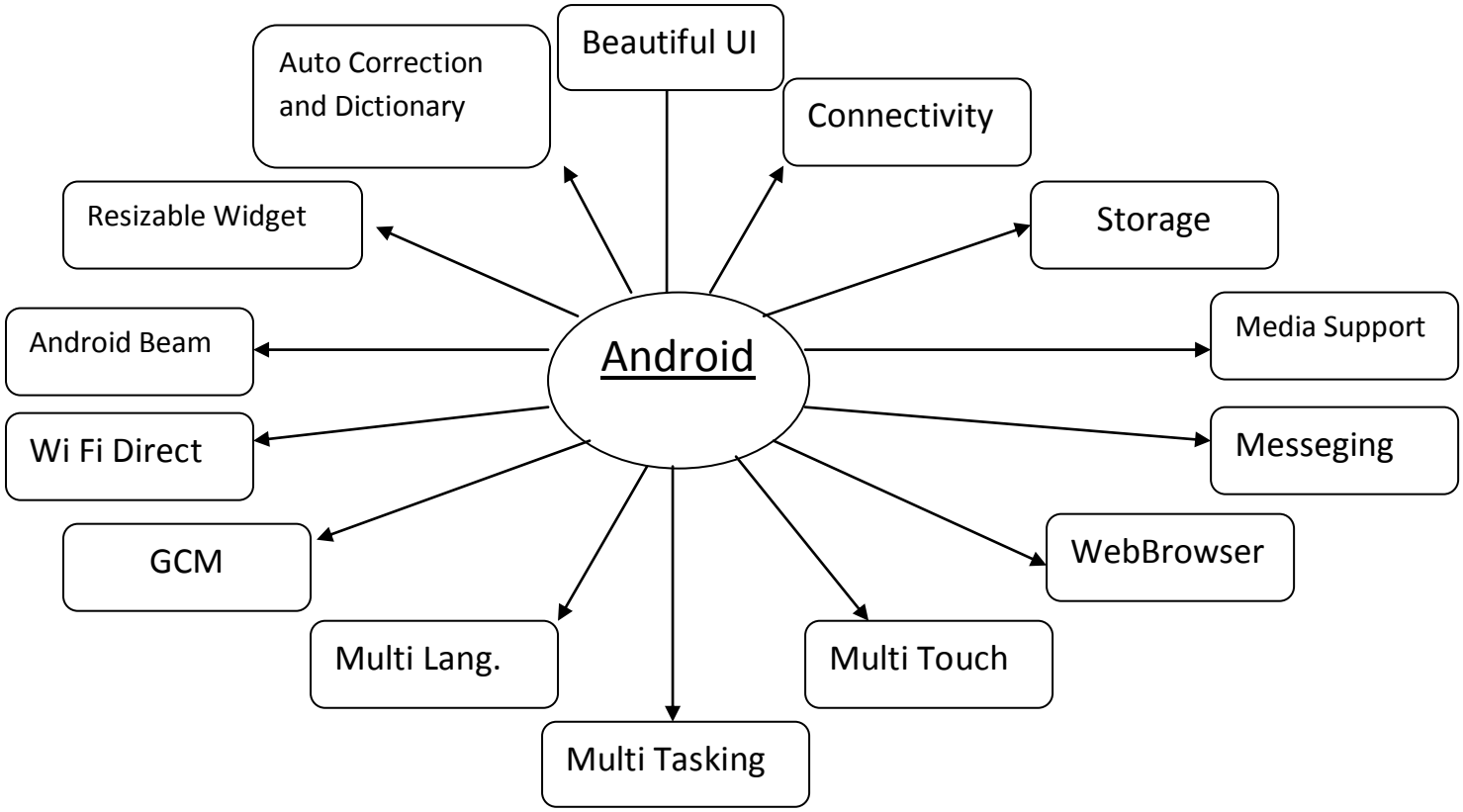
→ The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008.

→ On June 27, 2012, at the Google I/O conference, Google announced the next Android version, 4.1 Jelly Bean. Jelly Bean is an incremental update, with the primary aim of improving the user interface, both in terms of functionality and performance.

→ The source code for Android is available under free and open source software licenses. Google publishes most of the code under the Apache License version 2.0 and the rest, Linux kernel changes, under the GNU General Public License version 2.

Features of Android

→ Android is a powerful operating system competing with Apple 4GS and supports great features. Few of them are listed below –



Sr.No.	Feature & Description
1	Beautiful UI Android OS basic screen provides a beautiful and intuitive user interface.
2	Connectivity GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX.
3	Storage SQLite, a lightweight relational database, is used for data storage purposes.
4	Media support H.263, H.264, MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, AAC 5.1, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP.
5	Messaging SMS and MMS
6	Web browser Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3.
7	Multi-touch Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero.
8	Multi-tasking User can jump from one task to another and same time various application can run simultaneously.
9	Resizable widgets Widgets are resizable, so users can expand them to show more content or shrink them to save space.
10	Multi-Language Supports single direction and bi-directional text.
11	GCM Google Cloud Messaging (GCM) is a service that lets developers send short message data to their users on Android devices, without needing a proprietary sync solution.
12	Wi-Fi Direct A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection.
13	Android Beam A popular NFC-based technology that lets users instantly share, just by touching two NFC-enabled phones together.
14	Auto Correction and Dictionary When any word is misspelled, then Android recommends the meaningful and correct words matching the words that are available in Dictionary. Users can add, edit and remove words from Dictionary as per their wish.

History of Android

→ Initially, Andy Rubin founded Android Incorporation in Palo Alto, California, United States in October, 2003.

→ In 17th August 2005, Google acquired android Incorporation. Since then, it is in the subsidiary of Google Incorporation.

→ The key employees of Android Incorporation are Andy Rubin, Rich Miner, Chris White and Nick Sears.

→ Originally intended for camera but shifted to smart phones later because of low market for camera only.

→ Android is the nick name of Andy Rubin given by coworkers because of his love to robots.

→ In 2007, Google announces the development of android OS.

→ Android made its official public debut in 2008 with Android 1.0 — a release so ancient it didn't even have a cute codename.

→ The first Android mobile was publicly released with Android 1.0 of the T-Mobile G1 (aka HTC Dream) in October 2008.

→ The code names of android ranges from A to P currently, such as Aestro, Blender, Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwitch, Jelly Bean, KitKat, Lollipop Marshmallow, Nougat, Oreo, Pie.

→ Google announced in August 2019 that they were ending the confectionery scheme, and they use numerical ordering for future Android versions.
→ The first Android version which was released under the numerical order format was Android 10.
→ The latest version of Android is Android 11 released on September 8, 2020.

Android Applications

→ Android applications are usually developed in the Java language using the Android Software Development Kit.
→ Once developed, Android applications can be packaged easily and sold out either through a store such as Google Play, SlideME, Opera Mobile Store, Mobango, F-droid and the Amazon Appstore.
→ Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base of any mobile platform and growing fast. Every day more than 1 million new Android devices are activated worldwide.
→ This tutorial has been written with an aim to teach we how to develop and package Android application. We will start from environment setup for Android application programming and then drill down to look into various aspects of Android applications.

⇒ **Categories of Android applications**
→ There are many android applications in the market. The top categories are –
→ Music, News, Multimedia, Sports, Lifestyle, Food & Drink, Travel, Weather, Books, Business, Reference, Navigation, Social Media, Utilities, Finance etc…

Android IDEs

→ There are so many sophisticated Technologies are available to develop android applications, the familiar technologies, which are predominantly using tools as follows
- Android Studio
- Eclipse IDE(Deprecated)

Android Architecture

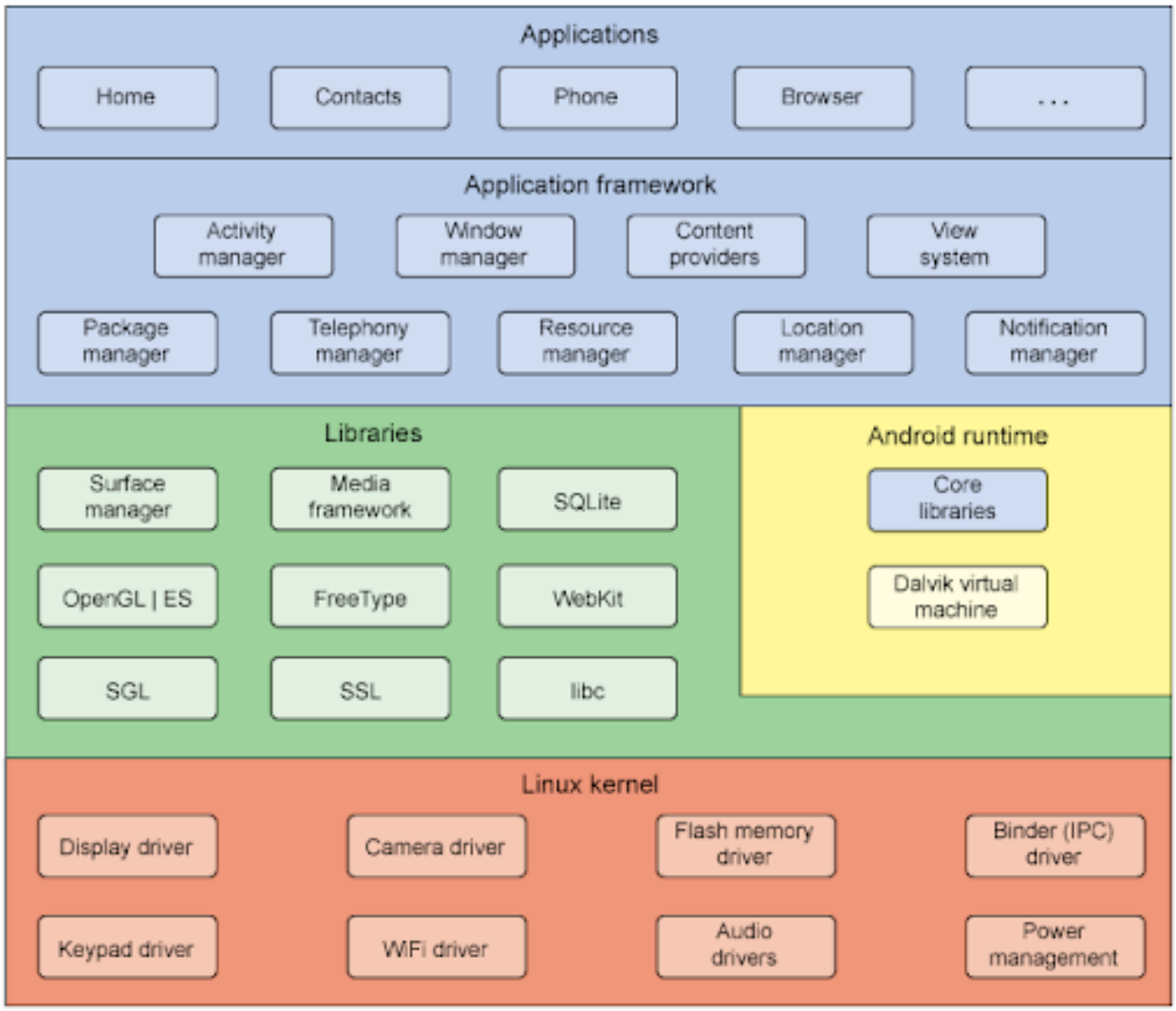
→ android architecture or Android software stack is categorized into five parts:
1. Linux kernel
2. Native libraries (middleware),
3. Android Runtime
4. Application Framework
5. Applications

1. Linux kernel

→ It is the heart of android architecture that exists at the root of android architecture. Linux kernel is responsible for device drivers, power management, memory management, device management and resource access.

2. Libraries(Native Libraries)

→ On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, SSL libraries responsible for Internet security, OpenGL, FreeType for font support, Media for playing and recording audio and video formats, C runtime library (libc) etc.



3. Android Runtime

→ This is the third section of the architecture and available on the second layer from the bottom.

→ In android runtime, there are core libraries and DVM (Dalvik Virtual Machine)

- Core Libraries

This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user interface building, graphics drawing and database access. A summary of some key core Android libraries available to the Android developer is as follows −

- ✓ android.app − Provides access to the application model and is the cornerstone of all Android applications.
- ✓ android.content − Facilitates content access, publishing and messaging between applications and application components.
- ✓ android.database − Used to access data published by content providers and includes SQLite database management classes.
- ✓ android.opengl − A Java interface to the OpenGL ES 3D graphics rendering API.
- ✓ android.os − Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- ✓ android.text − Used to render and manipulate text on a device display.
- ✓ android.view − The fundamental building blocks of application user interfaces.
- ✓ android.widget − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- ✓ android.webkit − A set of classes intended to allow web-browsing capabilities to be built into applications.

- DVM

This section provides a key component called Dalvik Virtual Machine which is responsible to run android application. DVM is like JVM but it is optimized for mobile devices. It consumes less memory and provides fast performance.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language. The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.
The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

4. Application Framework

→ The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.
→ The Android framework includes the following key services −

- Activity Manager − Controls all aspects of the application lifecycle and activity stack.
- Content Providers − Allows applications to publish and share data with other applications.
- Resource Manager − Provides access to non-code embedded resources such as strings, color settings and user interface layouts.
- Notifications Manager − Allows applications to display alerts and notifications to the user.
- View System − An extensible set of views used to create application user interfaces.

5. Applications

→ We will find all the Android application at the top layer. We will write our application to be installed on this layer only. Examples of such applications are Contacts Books, Browser, Games etc.

Android Core Building Blocks

→ Application components are the essential building blocks of an Android application. These components are loosely coupled by the application manifest file *AndroidManifest.xml* that describes each component of the application and how they interact.

→ There are following four main components that can be used within an Android application –

1. Activity
2. Services
3. Broadcast Receivers
4. Content Providers

1. Activities

→ An activity represents a single screen with a user interface, in-short Activity performs actions on the screen. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. If an application has more than one activity, then one of them should be marked as the activity that is presented when the application is launched.

→ An activity is implemented as a subclass of Activity class as follows –

```
public class MainActivity extends Activity {
}
```

2. Services

→ A service is a component that runs in the background to perform long-running operations. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity.

→ A service is implemented as a subclass of Service class as follows −

```
public class MyService extends Service {
}
```

3. Broadcast Receivers

→ Broadcast Receivers simply respond to broadcast messages from other applications or from the system. For example, applications can also initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use, so this is broadcast receiver who will intercept this communication and will initiate appropriate action.

→ A broadcast receiver is implemented as a subclass of BroadcastReceiver class and each message is broadcaster as an Intent object.

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

4. Content Providers

→ A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the *ContentResolver* class. The data may be stored in the file system, the database or somewhere else entirely.

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

```
public class MyContentProvider extends  ContentProvider {
  public void onCreate(){}
}
```

→ We will go through these tags in detail while covering application components in individual chapters.

Additional Components

→ There are additional components which will be used in the construction of above mentioned entities, their logic, and wiring between them. These components are −

1. Fragment

Fragments are like parts of activity. An activity can display one or more fragments on the screen at the same time.

2. Views

A view is the UI element such as button, label, text field etc. Anything that we see is a view.

3. Layouts
View hierarchies that control screen format and appearance of the views.

4. Intents
Intent is used to invoke components. It is mainly used to:

- Start the service
- Launch an activity
- Display a web page
- Display a list of contacts
- Broadcast a message
- Dial a phone call etc.

For example, we may write the following code to view the webpage.

```
Intent intent=new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
startActivity(intent);
```

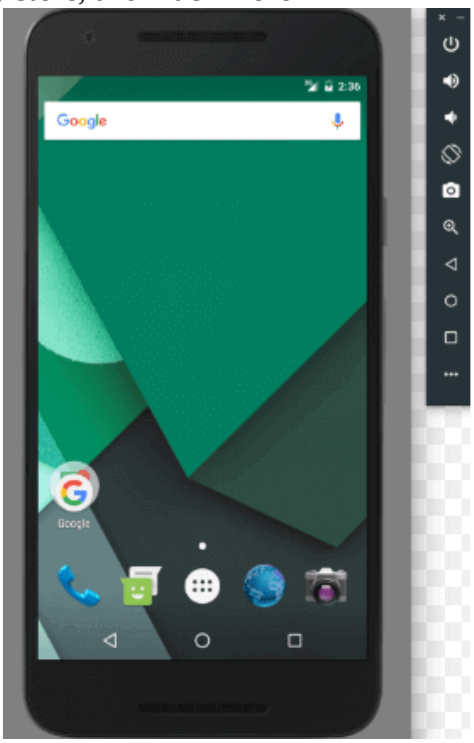
5. Resources
External elements, such as strings, constants and drawable pictures.

6. Manifest
It contains information's about activities, content providers, permissions etc. It is like the web.xml file in Java EE.

7. Android Virtual Device (AVD)
It is used to test the android application without the need for mobile or tablet etc. It can be created in different configurations to emulate different types of real devices.

Android Emulator

→ The Android emulator is an Android Virtual Device (AVD), which represents a specific Android device. We can use the Android emulator as a target device to execute and test our Android application on our PC. The Android emulator provides almost all the functionality of a real device. We can get the incoming phone calls and text messages. It also gives the location of the device and simulates different network speeds. Android emulator simulates rotation and other hardware sensors. It accesses the Google Play store, and much more



→ Testing Android applications on emulator are sometimes faster and easier than doing on a real device. For example, we can transfer data faster to the emulator than to a real device connected through USB.
→ The Android emulator comes with predefined configurations for several Android phones, Wear OS, tablet, Android TV devices.

Requirement and recommendations

→ The Android emulator takes additional requirements beyond the basic system requirement for Android Studio. These requirements are given below:

- SDK Tools 26.1.1 or higher
- 64-bit processor
- Windows: CPU with UG (unrestricted guest) support
- HAXM 6.2.1 or later (recommended HAXM 7.2.0 or later)

Install the emulator

→ The Android emulator is installed while installing the Android Studio. However some components of emulator may or may not be installed while installing Android Studio. To install the emulator component, select the Android Emulator component in the SDK Tools tab of the SDK Manager.

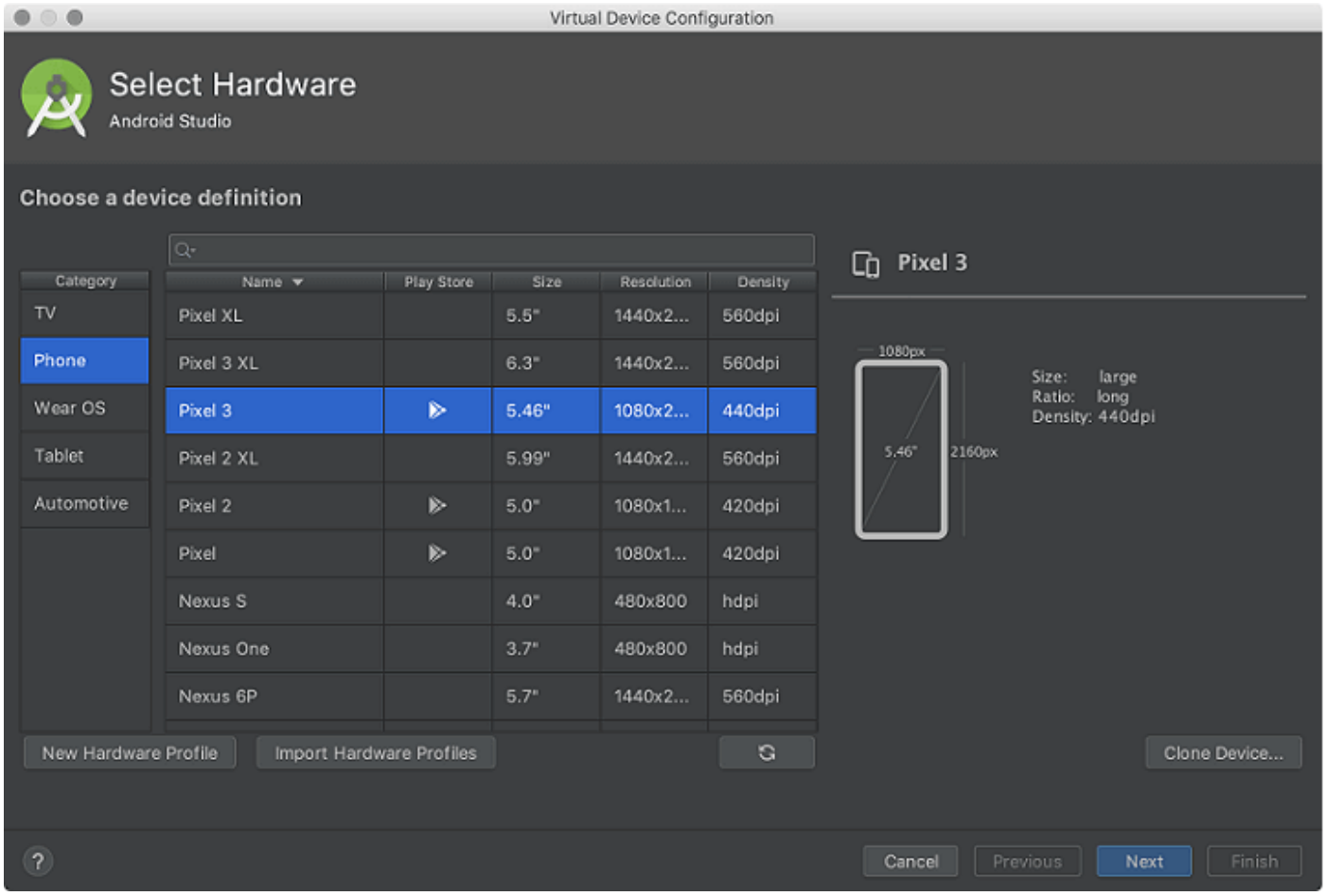
Run an Android app on the Emulator

→ We can run an Android app form the Android Studio project, or we can run an app which is installed on the Android Emulator as we run any app on a device.
→ To start the Android Emulator and run an application in our project:

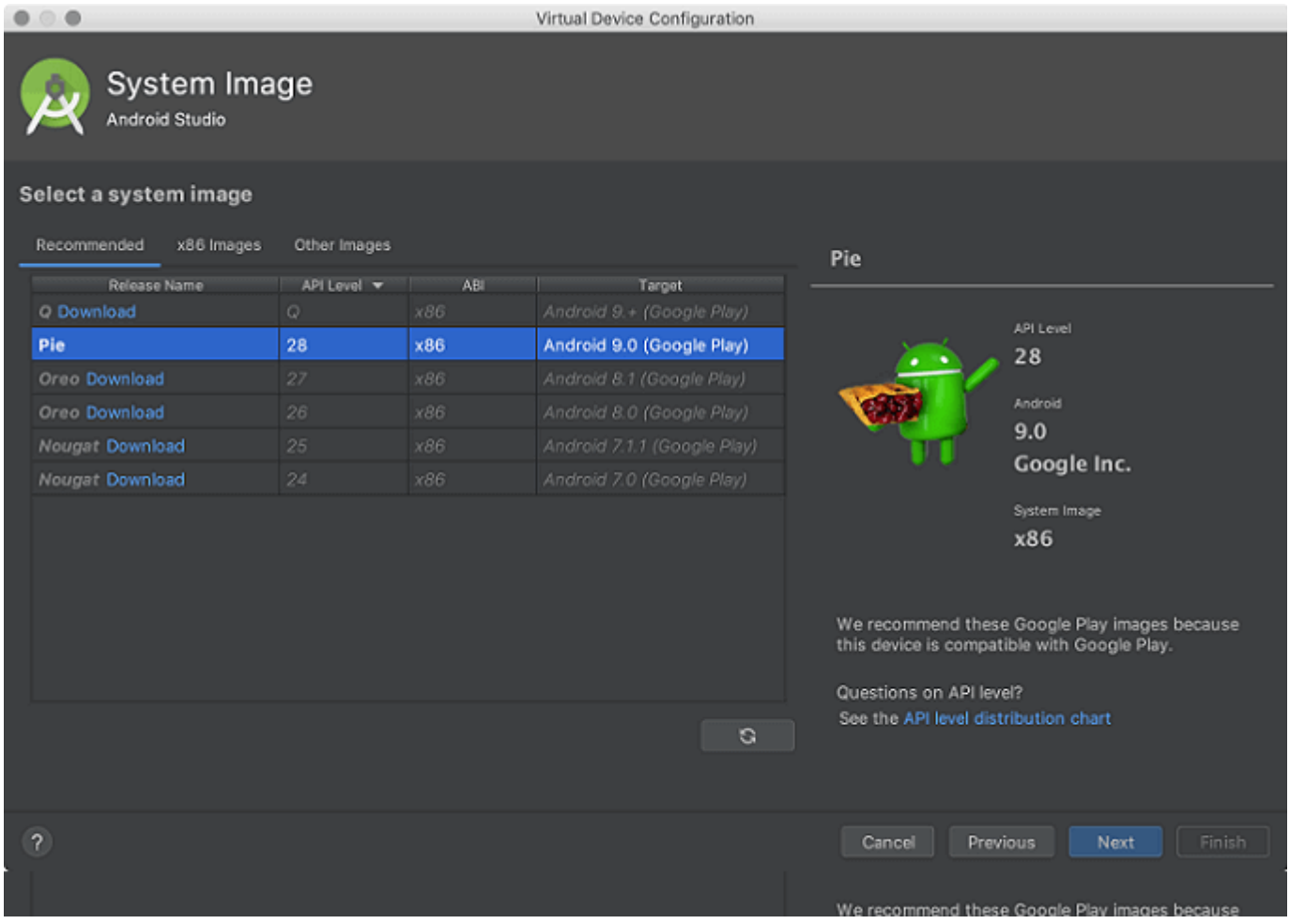
1. In Android Studio, we need to create an Android Virtual Device (AVD) that the emulator can use to install and run our app. To create a new AVD:-

1.1 Open the AVD Manager by clicking Tools > AVD Manager.

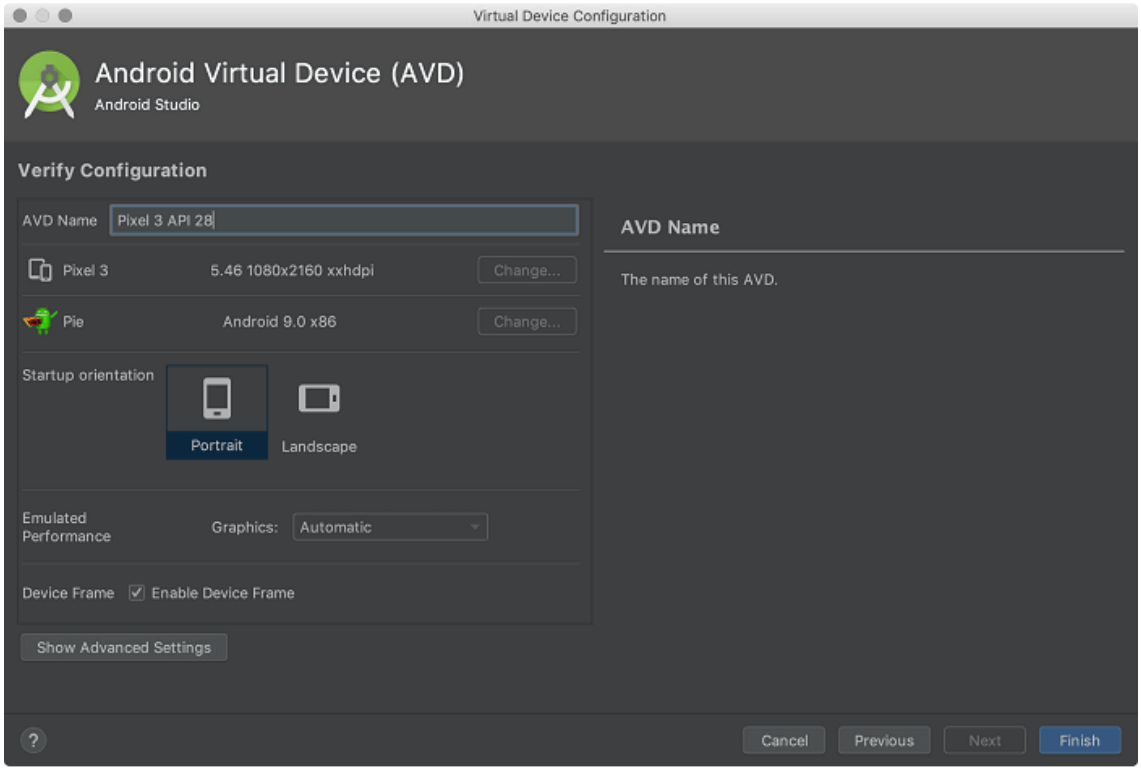
1.2 Click on Create Virtual Device, at the bottom of the AVD Manager dialog. Then Select Hardware page appears.

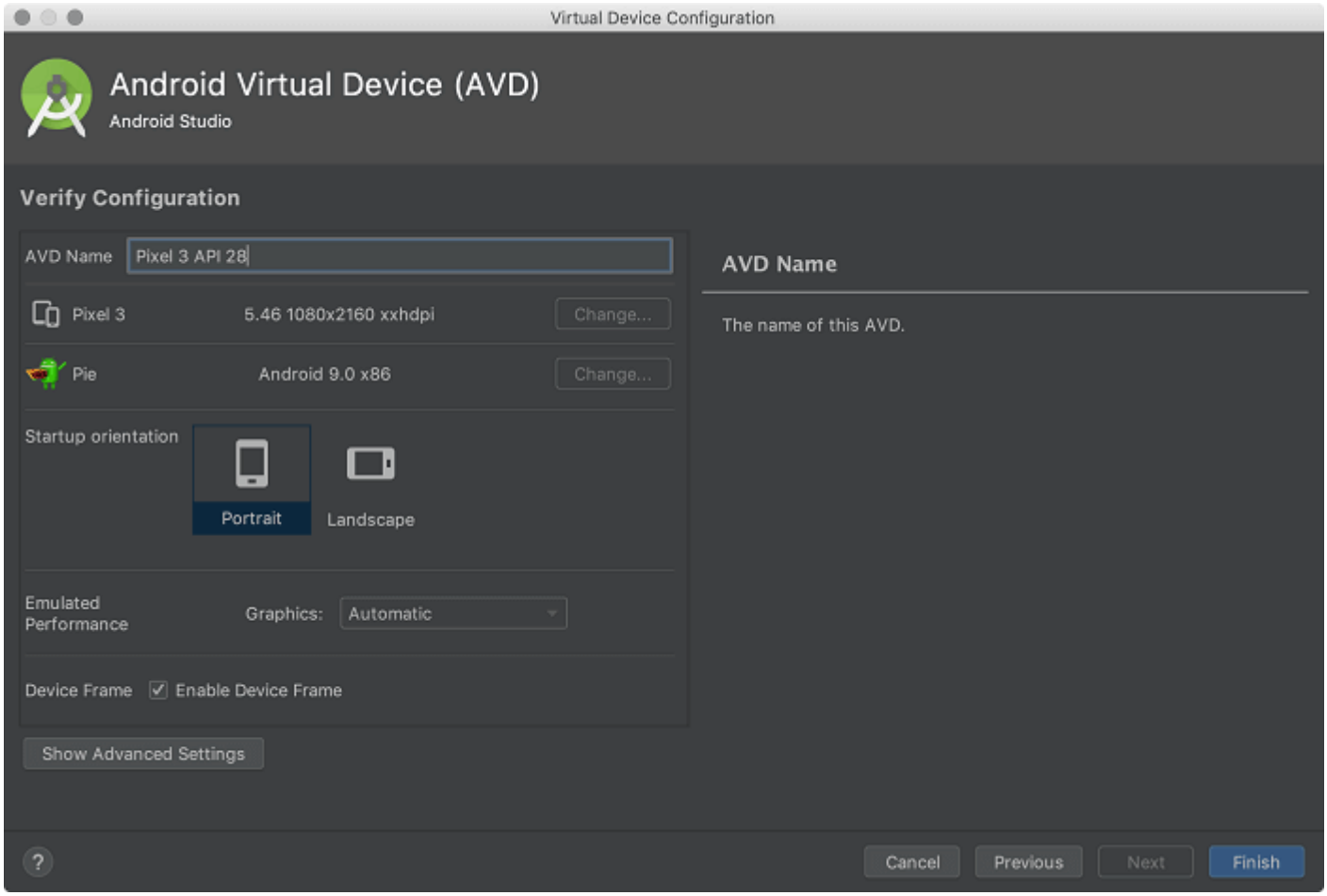


1.3 Select a hardware profile and then click Next. If we don?t see the hardware profile we want, then we can create or import a hardware profile. The System Image page appears.



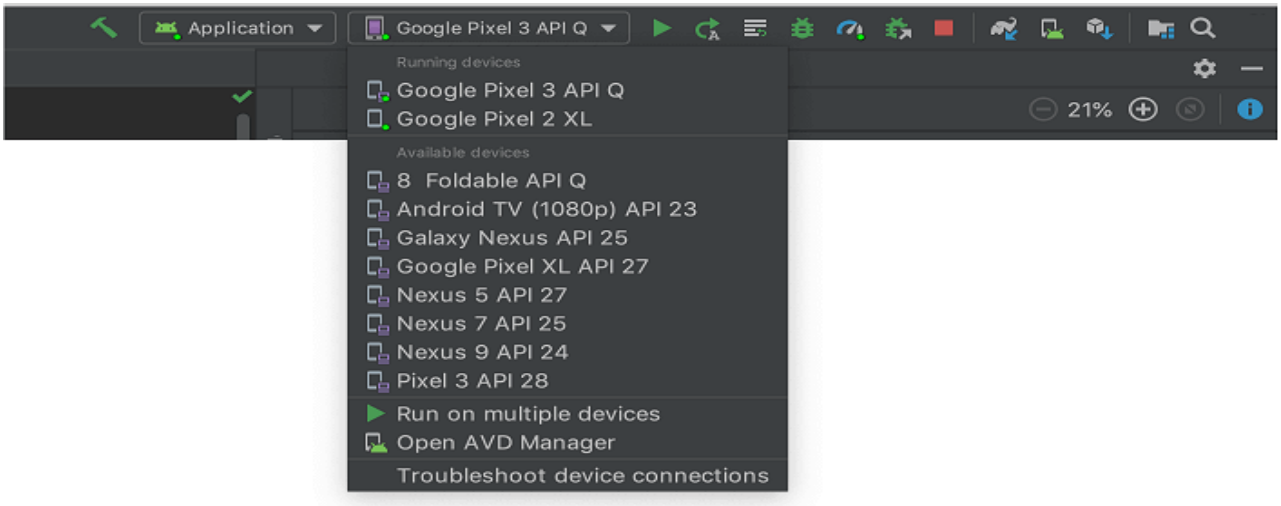
1.4 Select the system image for the particular API level and click Next. This leads to open a Verify Configuration page.





1.5 Change AVD properties if needed, and then click Finish.

2. In the toolbar, choose the AVD, which we want to run our app from the target device from the drop-down menu.



3. Click Run.

Launch the Emulator without first running an app

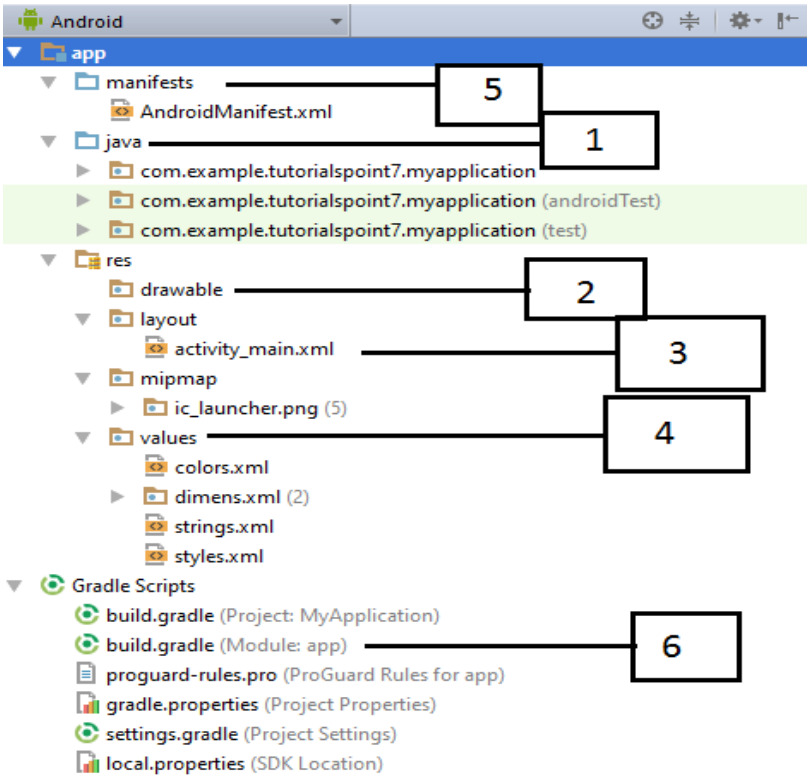
To start the emulator:

1. Open the AVD Manager.
2. Double-click an AVD, or click Run

While the emulator is running, we can run the Android Studio project and select the emulator as the target device. We can also drag an APKs file to install on an emulator, and then run them.

Anatomy of Android Application

→ Android application contains different components such as java source code, string resources, images, manifest file, apk file etc. Let's understand the project structure of android application.



- **AndroidManifeast.xml**

→ The AndroidManifest.xml file *contains information of application package*, including components of the application such as activities, services, broadcast receivers, content providers etc.

→ All the android application must have AndroidManifest.xml file in the root directory.

- **Java**

→ This contains the .java source files for project. By default, it includes a *MainActivity.java* source file having an activity class that runs when our app is launched using the app icon.

- **Generated Java**

→ This contains auto generated files like R.java which contains references to certain resources of the project or Application.

- **res(Resource)**

→ Android support resources like images and certain xml configuration files. These can be keeping separate from the source code. All these resources should be placed inside the "res" folder. The res folder will be having sub folders to keep the resources based on its type.

  - **res/drawable**

→ Drawable folder is a resource directory in an application that provides different bitmap drawables for medium,high,extrahigh,density screens.

- ✓ drawable-ldpi : Bitmap for Lower Density.
- ✓ drawable-mdpi : Bitmap for Medium Density.
- ✓ drawable-hdpi : Bitmap for High Density.
- ✓ drawable-xhdpi : Bitmap for Extra High Density.
- ✓ drawable-xxhdpi : Bitmap for X Extra High Density.
- ✓ drawable-xxxhdpi : Bitmap for X X Extra HighDensity.

  - **res/layout**

→ This folder contains the layout to be used in the application. The layout resources defines the Architecture for the UI(User Interface) in an activity or a component of UI.

  - **res/menu**

→ This folder containes menu resources to be used in application like options menu, context menu, submenu etc.

  - **res/mipmap**

→ This folder is for placing app launcher icon shown on home screen.

- ✓ mipmap-ldpi
- ✓ mipmap-mdpi
- ✓ mipmap-hdpi
- ✓ mipmap-xhdpi
- ✓ mipmap-xxhdpi
- ✓ mipmap-xxxhdpi

  - **res/value**

→ This folder is used to define strings, colors, dimensions, styles, static array of string and integer etc. By convation each type is stored in a separate file like res/values/strings.xml

- **Build.Gradle**

→ This is an auto generated file which contains compile sdk version, build tools version, application, Target SDK version, version code and version name.

Open Handset Alliance (OHA)

→ The Open Handset Alliance (OHA) is a business alliance that was created for the purpose of developing open mobile device standards. It's a consortium of 84 companies such as google, samsung, AKM, synaptics, KDDI, Garmin, Teleca, Ebay, Intel etc.

→ OHA is a group of Mobile and technology leaders who share the vision for changing the mobile experience.

→ The OHA was established on 5th November 2007, led by Google with 34 members, including mobile makers, application developer, some mobile carriers and chip makers. It is committed to advance open standards, provide services and deploy handsets using the Android Platform.

→ Now OHA is a consortium of 84 firms to develop open firms to develop open standards for mobile devices. Member firms include HTC, Sony, Dell, Intel, Motorola, Qualcomm, Texas Instruments, Google, Samsung Electronics, LG Electronics, T-Mobile, Sprint Corporation (now merged with T-Mobile US), Nvidia, and Wind River Systems.

Application, Context, Activities, Services, Intents

⇒ Context

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

⇒ Application

→ Base class for maintaining global application state. The application object is created whenever one of our Android Components is started. It is started in a new process with a unique ID under a single user.

→ Even if we do not specify one in our AndroidManifest.xml file, the Android System creates a default object for us. This object provides the following life cycle methods:

1) **onCreate()** : Called when the application is starting, before any activity, service, or receiver objects (excluding content providers) have been created.
2) **onLowMemory()** : This is called when the overall system is running low on memory, and actively running processes should trim their memory usage.
3) **onTerminate()** : This method is for use in emulated process environments. It will never be called on a production Android device, where processes are removed by simply killing them; no user code (including this callback) is executed when doing so.
4) **onConfigurationChanged()** : Called by the system when the device configuration changes while our component is running.

→ We can provide our own implementation by creating a subclass and specifying the fully-qualified name of this subclass as the "android:name" attribute in our AndroidManifest.xml's <application> tag. The Application class, or our subclass of the Application class, is instantiated before any other class when the process for our application/package is created.

→ The application object starts before any component and runs at least as long as another component of the application runs.

→ If the Android system needs to terminate processes it follows the following priority system.

→ **Priorities:**

Process	Detail	Priority
foreground	An application in which the user is interacting with an activity, or which has service which is bound to such an activity. Also if a service is executing one of its lifecycle methods or a broadcast receiver which runs its onReceive() method.	1
visible	User is not interacting with the activity, but the activity is still(partially) visible or the application has a service which is used by a inactive but visible activity	2

service	Application with a running service which does not qualify for 1 or 2.	3
background	Application with only stopped activities and without a service or executing receiver. Android keeps them in a least recent used (LRU) list and if requires terminates the one which was least used.	4
empty	Application without any active components.	5

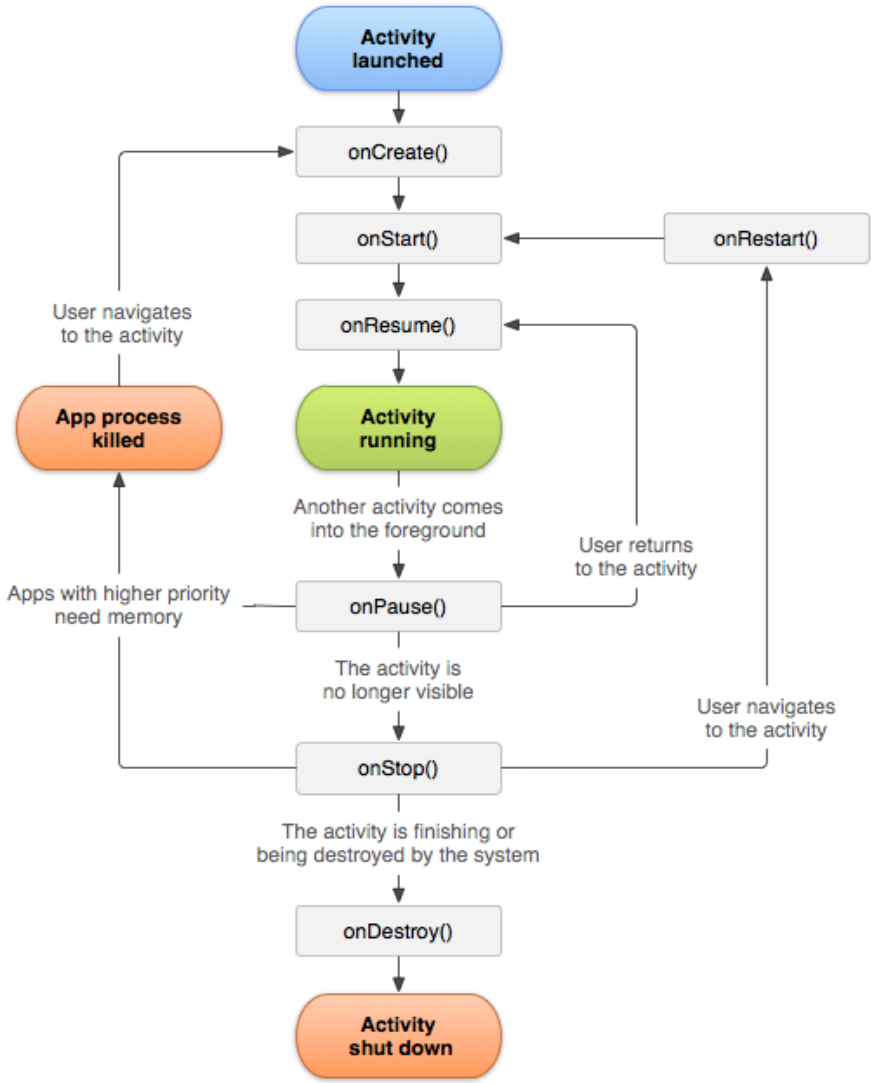
⇒ Activities

→ An activity represents a single screen with a user interface just like window or frame of Java.Android activity is the subclass of ContextThemeWrapper class.

→ Activity State :

State	Description
Running	Activity is visible and interacts with the user.
Paused	Activity is still visible but partially obscured, instance is running but might be killed by the system
Stopped	Activity is not visible, instance is running but might be killed by the system.
Killed	Activity has been terminated by the system of by a call to its finish() method.

→ The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.



→ **Methods:**

Method	Detail
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 our activity is stopped, prior to start.
onDestroy()	called before the activity is destroyed.

→ Example:

```
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      Log.d("lifecycle","onCreate invoked");
   }
   @Override
   protected void onStart() {
      super.onStart();
      Log.d("lifecycle","onStart invoked");
   }
   @Override
   protected void onResume() {
      super.onResume();
      Log.d("lifecycle","onResume invoked");
   }
   @Override
   protected void onPause() {
      super.onPause();
      Log.d("lifecycle","onPause invoked");
   }
   @Override
   protected void onStop() {
      super.onStop();
      Log.d("lifecycle","onStop invoked");
   }
   @Override
   protected void onRestart() {
      super.onRestart();
      Log.d("lifecycle","onRestart invoked");
   }
   @Override
   protected void onDestroy() {
      super.onDestroy();
      Log.d("lifecycle","onDestroy invoked");
   }
}
```

⇒ Services

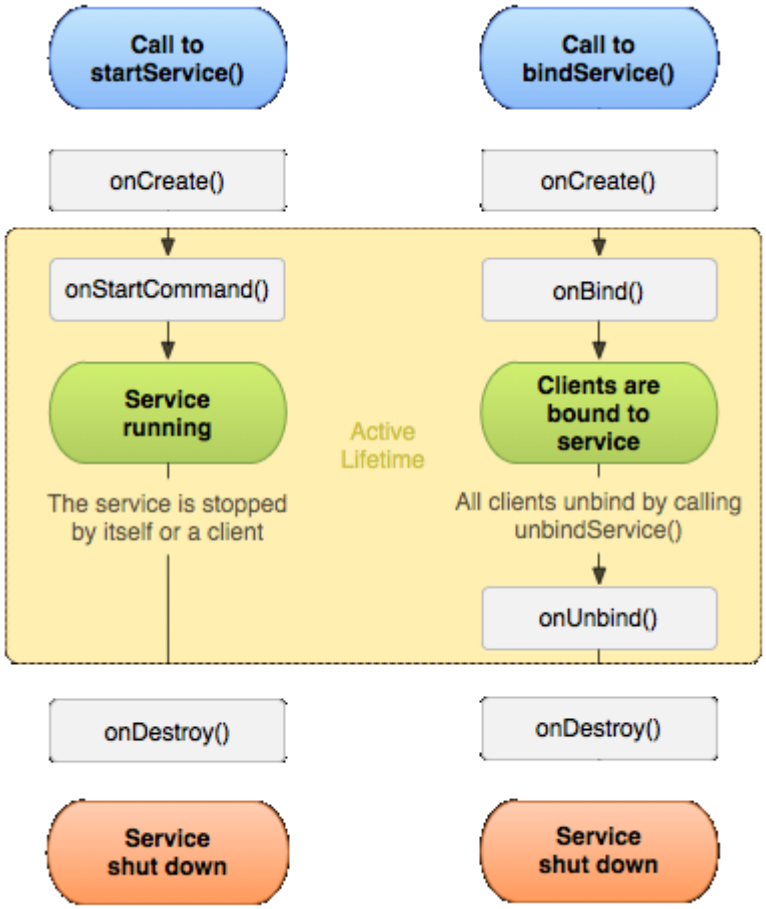
→ A service is component that runs in the background to perform long-running operations without needing to interact with the user. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity. A service can essentially take two states:

State	**Description**
Started	A service is **started** when an application component, such as an activity, starts is by calling

	startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
Bound	A service is **bound** when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service,send requests,get results and even do so across processes with interprocess communication(IPC).

→ A service has life cycle callback methods that we can implement to monitor changes in the service's state and we can perform work at the appropriate stage. The following diagram on the left shows the life cycle when the service is created with startService() and the diagram on the right shows the life cycle when the service is created with bindService()

→ To create an service, we create a Java class that extends the Service base class or one of its existing subclasses. The Service base class defines various callback methods and the most important are given below. We don't need to implement all the callbacks methods. However, it's important that we understand each one and implement those that ensure our app behaves the way users expect.



→ **Methods:**

Method	Detail
onStartCommand()	The system calls this method when another component, such as an activity, requests that the service be started, by calling startService(). If we implement this method, it is our responsibility to stop the service when its work is done, by calling stopSelf() or stopService() methods.
onBind()	The system calls this method when another component wants to bind with the service by calling bindService(). If we implement this method, we must provide an interface that clients use to communicate with the service, by returning an IBinder object. We must always implement this method, but if we don't want to allow binding, then we should return null.
onUnbind()	The system calls this method when all clients have disconnected from a particular interface published by the service.
onRebind()	The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its onUnbind(Intent).

onCreate()	The system calls this method when the service is first created using onStartCommand() or onBind(). This call is required to perform one-time set-up.
onDestroy()	The system calls this method when the service is no longer used and is being destroyed. Our service should implement this to clean up any resources such as threads, registered listeners, receivers, etc.

→ **Example :**

```
import android.app.Service;
import android.os.IBinder;
import android.content.Intent;
import android.os.Bundle;
public class HelloService extends Service {
  /** indicates how to behave if the service is killed */
  int mStartMode;

  /** interface for clients that bind */
  IBinder mBinder;

  /** indicates whether onRebind should be used */
  boolean mAllowRebind;

  /** Called when the service is being created. */
  @Override
  public void onCreate() {

  }

  /** The service is starting, due to a call to startService() */
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    return mStartMode;
  }

  /** A client is binding to the service with bindService() */
  @Override
  public IBinder onBind(Intent intent) {
    return mBinder;
  }

  /** Called when all clients have unbound with unbindService() */
  @Override
  public boolean onUnbind(Intent intent) {
    return mAllowRebind;
  }

  /** Called when a client is binding to the service with bindService()*/
  @Override
  public void onRebind(Intent intent) {

  }

  /** Called when The service is no longer used and is being destroyed */
  @Override
  public void onDestroy() {

  }
}
```

⇒Intent and Intent Filter

❖ Intent

→ The intent itself, an Intent object, is a passive data structure holding an abstract description of an operation to be performed.

→ An Intent is a messaging object we can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use cases:

Starting an activity

- An Activity represents a single screen in an app. We can start a new instance of an Activity by passing an Intent to startActivity().
- The Intent describes the activity to start and carries any necessary data.
- If we want to receive a result from the activity when it finishes, call startActivityForResult(). Activity receives the result as a separate Intent object in our activity's onActivityResult() callback.

Starting a service

- A Service is a component that performs operations in the background without a user interface. With Android 5.0 (API level 21) and later, we can start a service with JobScheduler.
- For versions earlier than Android 5.0 (API level 21), we can start a service by using methods of the Service class. we can start a service to perform a one-time operation (such as downloading a file) by passing an Intent to startService(). The Intent describes the service to start and carries any necessary data.
- If the service is designed with a client-server interface, we can bind to the service from another component by passing an Intent to bindService().

Delivering a broadcast

- A broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging. We can deliver a broadcast to other apps by passing an Intent to sendBroadcast() or sendOrderedBroadcast().

❖ Intent types

→ There are two types of intents:

- **Explicit intents:** specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. We'll typically use an explicit intent to start a component in our own app, because we know the class name of the activity or service we want to start. For example, we might start a new activity within our app in response to a user action, or start a service to download a file in the background.

→ **Example:**

```
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

- **Implicit intents:** do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if we want to show the user a location on a map, we can use an implicit intent to request that another capable app show a specified location on a map.

→ **Example:**

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

String title = getResources().getString(R.string.chooser_title);
// Create intent to show the chooser dialog
Intent chooser = Intent.createChooser(sendIntent, title);

// Verify the original intent will resolve to at least one activity
```

```
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

❖ Intent Object:

→ An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), plus information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon).

→ The primary information contained in an Intent is the following:

- **Component name**

→ The name of the component to start.

→ This is optional, but it's the critical piece of information that makes an intent *explicit*, meaning that the intent should be delivered only to the app component defined by the component name. Without a component name, the intent is *implicit* and the system decides which component should receive the intent based on the other intent information (such as the action, data, and category—described below). If we need to start a specific component in our app, we should specify the component name.

→ This field of the Intent is a ComponentName object, which we can specify using a fully qualified class name of the target component, including the package name of the app, for example, com.example.ExampleActivity. We can set the component name with setComponent(), setClass(), setClassName(), or with the Intent constructor.

- **Action**

→ A string that specifies the generic action to perform (such as *view* or *pick*).

→ In the case of a broadcast intent, this is the action that took place and is being reported. The action largely determines how the rest of the intent is structured—particularly the information that is contained in the data and extras.

→ We can specify our own actions for use by intents within our app (or for use by other apps to invoke components in our app), but we usually specify action constants defined by the Intent class or other framework classes. We can specify the action for an intent with setAction() or with an Intent constructor and read by getAction(). Here are some common actions for starting an activity:

- ACTION_VIEW
- ACTION_SEND
- ACTION_ANSWER
- ACTION_BATTERY_LOW
- ACTION_CALLED

- **Data**

→ The URI (a Uri object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied is generally dictated by the intent's action. For example, if the action is ACTION_EDIT, the data should contain the URI of the document to edit.

→ When creating an intent, it's often important to specify the type of data (its MIME type) in addition to its URI. For example, an activity that's able to display images probably won't be able to play an audio file, even though the URI formats could be similar. Specifying the MIME type of our data helps the Android system find the best component to receive our intent. However, the MIME type can sometimes be inferred from the URI—particularly when the data is a content: URI. A content: URI indicates the data is located on the device and controlled by a ContentProvider, which makes the data MIME type visible to the system.

→ To set only the data URI, call setData(). To set only the MIME type, call setType(). If necessary, We can set both explicitly with setDataAndType().

- **Category**

→ A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an intent, but most intents do not require a category. Here are some common categories:
  - CATEGORY_APP_BROWSABLE
  - CATEGORY_LAUNCHER
  - CATEGORY_APP_CALENDAR
  - CATEGORY_APP_CONTACT
  - CATEGORY_APP_GALLARY
  - CATEGORY_APP_CALCULATOR

→ We can specify a category with addCategory(). To remove the category that previously added removeCategory() method is used. To get category getCategories() method is used.

- **Extras**

→ Key-value pairs that carry additional information required to accomplish the requested action. Just as some actions use particular kinds of data URIs, some actions also use particular extras.

→ We can add extra data with various putExtra() methods, each accepting two parameters: the key name and the value. We can also create a Bundle object with all the extra data, then insert the Bundle in the Intent with putExtras().

→ For example, when creating an intent to send an email with ACTION_SEND, we can specify the *to* recipient with the EXTRA_EMAIL key, and specify the *subject* with the EXTRA_SUBJECT key.

- **Flags**

→ Flags are defined in the Intent class that function as metadata for the intent. The flags may instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). There are some flag constants:
  - FLAG_ACTIVITY_CLEAR_TASK
  - FLAG_ACTIVITY_CLEAR_TOP
  - FLAG_ACTIVITY_NEW_TASK

❖ Intent Filter

→ An intent filter is an expression in an app's manifest file that specifies the type of intents that the component would like to receive. For instance, by declaring an intent filter for an activity, we make it possible for other apps to directly start our activity with a certain kind of intent. Likewise, if we do *not* declare any intent filters for an activity, then it can be started only with an explicit intent.

→ Specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component.

→ Most of the contents of the filter are described by its <action>, <category>, and <data> subelements.

→ **Syntax:**

```
<intent-filter android:icon="drawable resource"
        android:label="string resource"
        android:priority="integer" >
    . . .
</intent-filter>
```

→ **Contained in:**

<activity>
<activity-alias>
<service>
<receiver>
<provider>

→ **attributes:**

android:icon

An icon that represents the parent activity, service, or broadcast receiver when that component is presented to the user as having the capability described by the filter.

android:label

A user-readable label for the parent component. This label, rather than the one set by the parent component, is used when the component is presented to the user as having the capability described by the filter.

android:priority

The priority that should be given to the parent component with regard to handling intents of the type described by the filter. This attribute has meaning for both activities and broadcast receivers:

The value must be an integer, such as "100". Higher numbers have a higher priority. The default value is 0.

android:order

The order in which the filter should be processed when multiple filters match.

order differs from priority in that priority applies across apps, while order disambiguates multiple matching filters in a single app.

When multiple filters could match, use a directed intent instead.

The value must be an integer, such as "100". Higher numbers are matched first. The default value is 0.This attribute was introduced in API Level 28.

→ **Elements:**

<action>

Adds an action to an intent filter. An <intent-filter> element must contain one or more <action> elements. If there are no <action> elements in an intent filter, the filter doesn't accept any Intent objects. See Intents and Intent Filters for details on intent filters and the role of action specifications within a filter.

attributes:

android:name : The name of the action.

<category>

Adds a category name to an intent filter. See Intents and Intent Filters for details on intent filters and the role of category specifications within a filter.

attributes:

android:name : The name of the category.

<data>

Adds a data specification to an intent filter.

attributes:

android:scheme : The scheme part of a URI. This is the minimal essential attribute for specifying a URI; at least one scheme attribute must be set for the filter, or none of the other URI attributes are meaningful.

A scheme is specified without the trailing colon (for example, http, rather than http:).

android:host : The host part of a URI authority. This attribute is meaningless unless a scheme attribute is also specified for the filter. To match multiple subdomains, use an asterisk (*) to match zero or more characters in the host. For example, the host *.google.com matches www.google.com, .google.com, and developer.google.com.

→ **Example:**

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

Android R.java file

→ Android R.java is *an auto-generated file by aapt* (Android Asset Packaging Tool) that contains resource IDs for all the resources of res/ directory.

→ If we create any component in the activity_main.xml file, id for the corresponding component is automatically created in this file. This id can be used in the activity source file to perform any action on the component.

→ It includes a lot of static nested classes such as menu, id, layout, attr, drawable, string etc.

```
public final class R {
  public static final class attr {
  }
  public static final class drawable {
    public static final int ic_launcher=0x7f020000;
  }
  public static final class id {
    public static final int menu_settings=0x7f070000;
  }
  public static final class layout {
    public static final int activity_main=0x7f030000;
  }
  public static final class menu {
    public static final int activity_main=0x7f060000;
  }
  public static final class string {
    public static final int app_name=0x7f040000;
    public static final int hello_world=0x7f040001;
    public static final int menu_settings=0x7f040002;
  }
  public static final class style {
    public static final int AppBaseTheme=0x7f050000;
    public static final int AppTheme=0x7f050001;
  }
}
```

AndroidManifest.xml file in android

→ The AndroidManifest.xml file *contains information of our package*, including components of the application such as activities, services, broadcast receivers, content providers etc.

→ It performs some other tasks also:

- It is responsible to protect the application to access any protected parts by providing the permissions.
- It also declares the android api that the application is going to use.
- It lists the instrumentation classes. The instrumentation classes provides profiling and other informations. These informations are removed just before the application is published etc.

→ This is the required xml file for all the android application and located inside the root directory.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.javatpoint.hello"
  android:versionCode="1"
```

```
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/title_activity_main" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

→ **Elements of the AndroidManifest.xml file**

The elements used in the above xml file are described below.

- ***<manifest>***

manifest is the root element of the AndroidManifest.xml file. It has package attribute that describes the package name of the activity class.

- ***<application>***

application is the subelement of the manifest. It includes the namespace declaration. This element contains several subelements that declares the application component such as activity etc. The commonly used attributes are of this element are icon, label, theme etc. android:icon represents the icon for all the android application components. android:label works as the default label for all the application components. android:theme represents a common theme for all the android activities.

- ***<activity>***

activity is the subelement of application and represents an activity that must be defined in the AndroidManifest.xml file. It has many attributes such as label, name, theme, launchMode etc. android:label represents a label i.e. displayed on the screen. android:name represents a name for the activity class. It is required attribute.

- ***<intent-filter>***

intent-filter is the sub-element of activity that describes the type of intent to which activity, service or broadcast receiver can respond to.

- ***<action>***

It adds an action for the intent-filter. The intent-filter must have at least one action element.

- ***<category>***

It adds a category name to an intent-filter.

- ***<uses-permission>***

Specifies a system permission that the user must grant in order for the app to operate correctly.

- ***<uses-sdk>***

Lets we express an application's compatibility with one or more versions of the Android platform, by means of an API level integer.

- ***<uses-library>***

Specifies a shared library that the application must be linked against.

- ***<permission>***

Declares a security permission that can be used to limit access to specific components or features of this or other applications.

- ***<meta-data>***

A name-value pair for an item of additional, arbitrary data that can be supplied to the parent component.

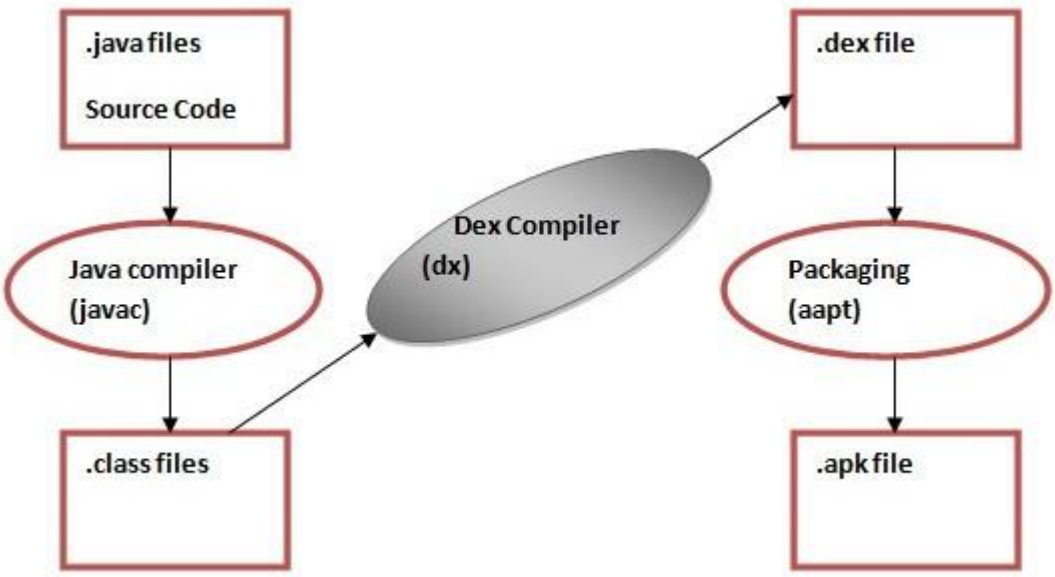
Dalvik Virtual Machine | DVM

→ As we know the modern JVM is high performance and provides excellent memory management. But it needs to be optimized for low-powered handheld devices as well.

→ The Dalvik Virtual Machine (DVM) is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for *memory*, *battery life* and *performance*.

→ Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein.

→ The Dex compiler converts the class files into the .dex file that run on the Dalvik VM. Multiple class files are converted into one dex file.



→ The javac tool compiles the java source file into the class file.

→ The dx tool takes all the class files of our application and generates a single .dex file. It is a platform-specific tool.

→ The Android Assets Packaging Tool (aapt) handles the packaging process.

Android Resources Organizing & Accessing

→ There are many more items which we use to build a good Android application. Apart from coding for the application, we take care of various other resources like static content that our code uses, such as bitmaps, colors, layout definitions, user interface strings, animation instructions, and more. These resources are always maintained separately in various sub-directories under res/ directory of the project.

❖ Default Resources:

→ The resources that we save in the subdirectories defined in table are our "default" resources.

Sr.No.	Directory & Resource Type
1	anim/ XML files that define property animations. They are saved in res/anim/ folder and accessed from the R.anim class.
2	color/ XML files that define a state list of colors. They are saved in res/color/ and accessed from the R.color class.
3	drawable/ Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the R.drawable class.
4	layout/ XML files that define a user interface layout. They are saved in res/layout/ and accessed from the R.layout class.
5	menu/ XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the R.menu class.
6	raw/ Arbitrary files to save in their raw form. We need to call *Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files.
7	values/ XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources we can create in this directory − • arrays.xml for resource arrays, and accessed from the R.array class. • integers.xml for resource integers, and accessed from the R.integer class. • bools.xml for resource boolean, and accessed from the R.bool class. • colors.xml for color values, and accessed from the R.color class. • dimens.xml for dimension values, and accessed from the R.dimen class. • strings.xml for string values, and accessed from the R.string class. • styles.xml for styles, and accessed from the R.style class.
8	xml/ Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. We can save various configuration files here which will be used at run time.

❖ Alternative Resources

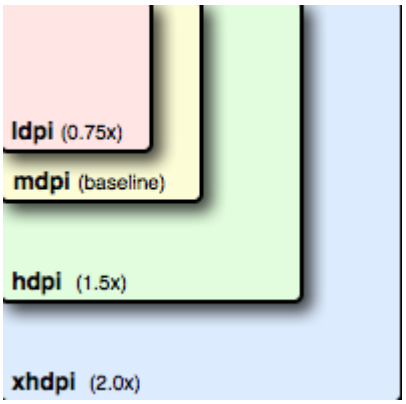
→ One of the most powerful tools available to developers is the option to provide "alternate resources" based on specific qualifiers such as phone size, language, density, and others.

→ Common uses for alternate resources include:

- Alternative layout files for different form factors (i.e phone vs tablet)

- Alternative string resources for different languages (i.e English vs Italian)
- Alternative drawable resources for different screen densities (shown below)
- Alternate style resources for different platform versions (Holo vs Material)
- Alternate layout files for different screen orientations (i.e portrait vs landscape)

→ To specify configuration-specific alternatives for a set of resources, we create a new directory in res in the form of [resource]-[qualifiers]. For example, one best practice is to ensure that all images are provided for multiple screen densities.



→ This is achieved by having res/drawable-hdpi, res/drawable-xhdpi, and res/drawable-xxhdpi folders with different versions of the same image. The correct resource is then selected automatically by the system based on the device density. The directory listing might look like the following:

```
res/
  drawable/
    icon.png
    background.png
  drawable-hdpi/
    icon.png
    background.png
```

Configuration	Qualifier Values	Description
MCC and MNC	Examples: mcc310 mcc310-mnc004 mcc208-mnc00 etc.	The mobile country code (MCC), optionally followed by mobile network code (MNC) from the SIM card in the device. For example, mcc310 is U.S. on any carrier, mcc310-mnc004 is U.S. on Verizon, and mcc208-mnc00 is France on Orange. If the device uses a radio connection (GSM phone), the MCC and MNC values come from the SIM card. We can also use the MCC alone (for example, to include country-specific legal resources in our app). If we need to specify based on the language only, then use the *language and region* qualifier instead (discussed next). If we decide to use the MCC and MNC qualifier, we should do so with care and test that it works as expected. Also see the configuration fields mcc, and mnc, which indicate the current mobile country code and mobile network code, respectively.
Language and region	Examples: en fr en-rUS fr-rFR fr-rCA b+en b+en+US	The language is defined by a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase r). The codes are *not* case-sensitive; the r prefix is used to distinguish the region portion. We cannot specify a region alone. Android 7.0 (API level 24) introduced support for BCP 47 language tags, which we can use to qualify language- and region-specific resources. A

	b+es+419	language tag is composed from a sequence of one or more subtags, each of which refines or narrows the range of language identified by the overall tag. For more information about language tags, see Tags for Identifying Languages. To use a BCP 47 language tag, concatenate b+ and a two-letter ISO 639-1 language code, optionally followed by additional subtags separated by +. The language tag can change during the life of our app if the users change their language in the system settings. See Handling Runtime Changes for information about how this can affect our app during runtime. See Localization for a complete guide to localizing our app for other languages. Also see the getLocales() method, which provides the defined list of locales. This list includes the primary locale.
Layout Direction	ldrtl ldltr	The layout direction of our app. ldrtl means "layout-direction-right-to-left". ldltr means "layout-direction-left-to-right" and is the default implicit value. This can apply to any resource such as layouts, drawables, or values. For example, if we want to provide some specific layout for the Arabic language and some generic layout for any other "right-to-left" language (like Persian or Hebrew) then we would have the following: res/ layout/ main.xml (Default layout) layout-ar/ main.xml (Specific layout for Arabic) layout-ldrtl/ main.xml (Any "right-to-left" language, except for Arabic, because the "ar" language qualifier has a higher precedence) Note: To enable right-to-left layout features for our app, we must set supportsRtl to "true" and set targetSdkVersion to 17 or higher. *Added in API level 17.*
smallestWidth	sw<N>dp Examples: sw320dp sw600dp sw720dp etc.	The fundamental size of a screen, as indicated by the shortest dimension of the available screen area. Specifically, the device's smallestWidth is the shortest of the screen's available height and width (we may also think of it as the "smallest possible width" for the screen). We can use this qualifier to ensure that, regardless of the screen's current orientation, our app's has at least <N> dps of width available for its UI. For example, if our layout requires that its smallest dimension of screen area be at least 600 dp at all times, then we can use this qualifier to create the layout resources, res/layout-sw600dp/. The system uses these resources only when the smallest dimension of available screen is at least 600dp, regardless of whether the 600dp side is the user-perceived height or width. The smallest width is a fixed screen size characteristic of the device; the device's smallest width doesn't change when the screen's orientation changes. Using smallest width to determine the general screen size is useful because width is often the driving factor in designing a layout. A UI will often scroll vertically, but have fairly hard constraints on the minimum space it needs horizontally. The available width is also the key factor in determining whether to use a one-pane layout for handsets or multi-pane layout for tablets. Thus, we likely care most about what the smallest possible width

		will be on each device.
		
		The smallest width of a device takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the screen that account for space along the axis of the smallest width, the system declares the smallest width to be smaller than the actual screen size, because those are screen pixels not available for our UI.
		
		Some values we might use here for common screen sizes:
		
		• 320, for devices with screen configurations such as:
		  • 240x320 ldpi (QVGA handset)
		  • 320x480 mdpi (handset)
		  • 480x800 hdpi (high-density handset)
		480, for screens such as 480x800 mdpi (tablet/handset).
		600, for screens such as 600x1024 mdpi (7" tablet).
		720, for screens such as 720x1280 mdpi (10" tablet).
		
		When our app provides multiple resource directories with different values for the smallestWidth qualifier, the system uses the one closest to (without exceeding) the device's smallestWidth.
		
		Added in API level 13.
		
		Also see the android:requiresSmallestWidthDp attribute, which declares the minimum smallestWidth with which our app is compatible, and the smallestScreenWidthDp configuration field, which holds the device's smallestWidth value.
		
		For more information about designing for different screens and using this qualifier, see the Supporting Multiple Screens developer guide.

Available width	w<N>dp Examples: w720dp w1024dp etc.	Specifies a minimum available screen width, in dp units at which the resource should be used—defined by the <N> value. This configuration value changes when the orientation changes between landscape and portrait to match the current actual width. This is often useful to determine whether to use a multi-pane layout, because even on a tablet device, we often won't want the same multi-pane layout for portrait orientation as we do for landscape. Thus, we can use this to specify the minimum width required for the layout, instead of using both the screen size and orientation qualifiers together. When our app provides multiple resource directories with different values for this configuration, the system uses the one closest to (without exceeding) the device's current screen width. The value here takes into account screen decorations, so if the device has some persistent UI elements on the left or right edge of the display, it uses a value for the width that is smaller than the real screen size, accounting for these UI elements and reducing the app's available space. *Added in API level 13.* Also see the screenWidthDp configuration field, which holds the current screen width. For more information about designing for different screens and using this qualifier, see the Supporting Multiple Screens developer guide.
Available height	h<N>dp	Specifies a minimum available screen height, in "dp" units at which the resource should be used—defined by the <N> value. This configuration

	Examples: h720dp h1024dp etc.	value changes when the orientation changes between landscape and portrait to match the current actual height. Using this to define the height required by our layout is useful in the same way as w<N>dp is for defining the required width, instead of using both the screen size and orientation qualifiers. However, most apps won't need this qualifier, considering that UIs often scroll vertically and are thus more flexible with how much height is available, whereas the width is more rigid. When our app provides multiple resource directories with different values for this configuration, the system uses the one closest to (without exceeding) the device's current screen height. The value here takes into account screen decorations, so if the device has some persistent UI elements on the top or bottom edge of the display, it uses a value for the height that is smaller than the real screen size, accounting for these UI elements and reducing the app's available space. Screen decorations that aren't fixed (such as a phone status bar that can be hidden when full screen) are *not* accounted for here, nor are window decorations like the title bar or action bar, so apps must be prepared to deal with a somewhat smaller space than they specify. *Added in API level 13.* Also see the screenHeightDp configuration field, which holds the current screen height. For more information about designing for different screens and using this qualifier, see the Supporting Multiple Screens developer guide.
Screen size	small normal large xlarge	• small: Screens that are of similar size to a low-density QVGA screen. The minimum layout size for a small screen is approximately 320x426 dp units. Examples are QVGA low-density and VGA high density. • normal: Screens that are of similar size to a medium-density HVGA screen. The minimum layout size for a normal screen is approximately 320x470 dp units. Examples of such screens a WQVGA low-density, HVGA medium-density, WVGA high-density. • large: Screens that are of similar size to a medium-density VGA screen. The minimum layout size for a large screen is approximately 480x640 dp units. Examples are VGA and WVGA medium-density screens. • xlarge: Screens that are considerably larger than the traditional medium-density HVGA screen. The minimum layout size for an xlarge screen is approximately 720x960 dp units. In most cases, devices with extra-large screens would be too large to carry in a pocket and would most likely be tablet-style devices. *Added in API level 9.* Note: Using a size qualifier does not imply that the resources are *only* for screens of that size. If we do not provide alternative resources with qualifiers that better match the current device configuration, the system may use whichever resources are the best match. Caution: If all our resources use a size qualifier that is *larger* than the current screen, the system will not use them and our app will crash at runtime (for example, if all layout resources are tagged with the xlarge qualifier, but the device is a normal-size screen). *Added in API level 4.*

		See Supporting Multiple Screens for more information. Also see the screenLayout configuration field, which indicates whether the screen is small, normal, or large.
Screen aspect	long notlong	• long: Long screens, such as WQVGA, WVGA, FWVGA • notlong: Not long screens, such as QVGA, HVGA, and VGA *Added in API level 4.* This is based purely on the aspect ratio of the screen (a "long" screen is wider). This isn't related to the screen orientation. Also see the screenLayout configuration field, which indicates whether the screen is long.
Round screen	round notround	• round: Round screens, such as a round wearable device • notround: Rectangular screens, such as phones or tablets *Added in API level 23.* Also see the isScreenRound() configuration method, which indicates whether the screen is round.
Wide Color Gamut	widecg nowidecg	• widecg: Displays with a wide color gamut such as Display P3 or AdobeRGB • nowidecg: Displays with a narrow color gamut such as sRGB *Added in API level 26.* Also see the isScreenWideColorGamut() configuration method, which indicates whether the screen has a wide color gamut.
High Dynamic Range (HDR)	highdr lowdr	• highdr: Displays with a high-dynamic range • lowdr: Displays with a low/standard dynamic range *Added in API level 26.* Also see the isScreenHdr() configuration method, which indicates whether the screen has a HDR capabilities.
Screen orientation	port land	• port: Device is in portrait orientation (vertical) • land: Device is in landscape orientation (horizontal) This can change during the life of our app if the user rotates the screen. See Handling Runtime Changes for information about how this affects our app during runtime. Also see the orientation configuration field, which indicates the current device orientation.
UI mode	car desk television appliance watch vrheadset	• car: Device is displaying in a car dock • desk: Device is displaying in a desk dock • television: Device is displaying on a television, providing a "ten foot" experience where its UI is on a large screen that the user is far away from, primarily oriented around DPAD or other non-pointer interaction • appliance: Device is serving as an appliance, with no display • watch: Device has a display and is worn on the wrist • vrheadset: Device is displaying in a virtual reality headset

		Added in API level 8, television added in API 13, watch added in API 20. For information about how our app can respond when the device is inserted into or removed from a dock, read Determining and Monitoring the Docking State and Type. This can change during the life of our app if the user places the device in a dock. We can enable or disable some of these modes using UiModeManager. See Handling Runtime Changes for information about how this affects our app during runtime.
Night mode	night notnight	• night: Night time • notnight: Day time *Added in API level 8.* This can change during the life of our app if night mode is left in auto mode (default), in which case the mode changes based on the time of day. We can enable or disable this mode using UiModeManager. See Handling Runtime Changes for information about how this affects our app during runtime.
Screen pixel density (dpi)	ldpi mdpi hdpi xhdpi xxhdpi xxxhdpi nodpi tvdpi anydpi *nnn*dpi	• ldpi: Low-density screens; approximately 120dpi. • mdpi: Medium-density (on traditional HVGA) screens; approximately 160dpi. • hdpi: High-density screens; approximately 240dpi. • xhdpi: Extra-high-density screens; approximately 320dpi. *Added in API Level 8* • xxhdpi: Extra-extra-high-density screens; approximately 480dpi. *Added in API Level 16* • xxxhdpi: Extra-extra-extra-high-density uses (launcher icon only, see the note in *Supporting Multiple Screens*); approximately 640dpi. *Added in API Level 18* • nodpi: This can be used for bitmap resources that we don't want to be scaled to match the device density. • tvdpi: Screens somewhere between mdpi and hdpi; approximately 213dpi. This isn't considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system scales them as appropriate. *Added in API Level 13* • anydpi: This qualifier matches all screen densities and takes precedence over other qualifiers. This is useful for vector drawables. *Added in API Level 21* • *nnn*dpi: Used to represent non-standard densities, where *nnn* is a positive integer screen density. This shouldn't be used in most cases. Use standard density buckets, which greatly reduces the overhead of supporting the various device screen densities on the market. There is a 3:4:6:8:12:16 scaling ratio between the six primary densities (ignoring the tvdpi density). So, a 9x9 bitmap in ldpi is 12x12 in mdpi, 18x18 in hdpi, 24x24 in xhdpi and so on. If we decide that our image resources don't look good enough on a television or other certain devices and want to try tvdpi resources, the scaling factor is 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi. Note: Using a density qualifier doesn't imply that the resources are *only* for screens of that density. If we don't provide alternative resources with qualifiers that better match the current device configuration, the system may

		use whichever resources are the best match. See Supporting Multiple Screens for more information about how to handle different screen densities and how Android might scale our bitmaps to fit the current density.
Touchscreen type	notouch finger	• notouch: Device doesn't have a touchscreen. • finger: Device has a touchscreen that is intended to be used through direction interaction of the user's finger. Also see the touchscreen configuration field, which indicates the type of touchscreen on the device.
Keyboard availability	keysexposed keyshidden keyssoft	• keysexposed: Device has a keyboard available. If the device has a software keyboard enabled (which is likely), this may be used even when the hardware keyboard *isn't* exposed to the user, even if the device has no hardware keyboard. If no software keyboard is provided or it's disabled, then this is only used when a hardware keyboard is exposed. • keyshidden: Device has a hardware keyboard available but it is hidden *and* the device does *not* have a software keyboard enabled. • keyssoft: Device has a software keyboard enabled, whether it's visible or not. If we provide keysexposed resources, but not keyssoft resources, the system uses the keysexposed resources regardless of whether a keyboard is visible, as long as the system has a software keyboard enabled. This can change during the life of our app if the user opens a hardware keyboard. See Handling Runtime Changes for information about how this affects our app during runtime. Also see the configuration fields hardKeyboardHidden and keyboardHidden, which indicate the visibility of a hardware keyboard and the visibility of any kind of keyboard (including software), respectively.
Primary text input method	nokeys qwerty 12key	• nokeys: Device has no hardware keys for text input. • qwerty: Device has a hardware qwerty keyboard, whether it's visible to the user or not. • 12key: Device has a hardware 12-key keyboard, whether it's visible to the user or not. Also see the keyboard configuration field, which indicates the primary text input method available.
Navigation key availability	navexposed navhidden	• navexposed: Navigation keys are available to the user. • navhidden: Navigation keys aren't available (such as behind a closed lid). This can change during the life of our app if the user reveals the navigation keys. See Handling Runtime Changes for information about how this affects our app during runtime. Also see the navigationHidden configuration field, which indicates whether navigation keys are hidden.
Primary non-touch navigation method	nonav dpad trackball	• nonav: Device has no navigation facility other than using the touchscreen. • dpad: Device has a directional-pad (d-pad) for navigation. • trackball: Device has a trackball for navigation.

	wheel	• wheel: Device has a directional wheel(s) for navigation (uncommon). Also see the navigation configuration field, which indicates the type of navigation method available.
Platform Version (API level)	Examples: v3 v4 v7 etc.	The API level supported by the device. For example, v1 for API level 1 (devices with Android 1.0 or higher) and v4 for API level 4 (devices with Android 1.6 or higher). See the Android API levels document for more information about these values.

❖ Qualifier name rules

→ We can specify multiple qualifiers for a single set of resources, separated by dashes. For example, drawable-en-rUS-land applies to US-English devices in landscape orientation.

→ The qualifiers must be in the order listed in table 2. For example:
- Wrong: drawable-hdpi-port/
- Correct: drawable-port-hdpi/

→ Alternative resource directories cannot be nested. For example, we cannot have res/drawable/drawable-en/.

→ Values are case-insensitive. The resource compiler converts directory names to lower case before processing to avoid problems on case-insensitive file systems. Any capitalization in the names is only to benefit readability.

→ Only one value for each qualifier type is supported. For example, if we want to use the same drawable files for Spain and France, we cannot have a directory named drawable-es-fr/. Instead we need two resource directories, such as drawable-es/ and drawable-fr/, which contain the appropriate files. However, we aren't required to actually duplicate the same files in both locations. Instead, we can create an alias to a resource. See Creating alias resources below.

→ After we save alternative resources into directories named with these qualifiers, Android automatically applies the resources in our app based on the current device configuration. Each time a resource is requested, Android checks for alternative resource directories that contain the requested resource file, then finds the best-matching resource (discussed below). If there are no alternative resources that match a particular device configuration, then Android uses the corresponding default resources (the set of resources for a particular resource type that doesn't include a configuration qualifier).

Creating alias resources

→ When we have a resource that we'd like to use for more than one device configuration (but don't want to provide as a default resource), we don't need to put the same resource in more than one alternative resource directory. Instead, we can (in some cases) create an alternative resource that acts as an alias for a resource saved in our default resource directory.

→ **Drawable**
- To create an alias to an existing drawable, use the <drawable> element.
- For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <drawable name="icon">@drawable/icon_ca</drawable>
</resources>
```

→ **Layout**
- To create an alias to an existing layout, use the <include> element, wrapped in a <merge>.

- For example:

```
<?xml version="1.0" encoding="utf-8"?>
<merge>
   <include layout="@layout/main_ltr"/>
</merge>
```

→ **Strings and other simple values**

- To create an alias to an existing string, simply use the resource ID of the desired string as the value for the new string.
- For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <string name="hello">Hello</string>
   <string name="hi">@string/hello</string>
</resources>
```

→ **Other simple values work the same way.**

- For example, a color:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="red">#f00</color>
   <color name="highlight">@color/red</color>
</resources>
```

Accessing Resources:

→ Once we provide a resource in our application, we can apply it by referencing its resource ID. All resource IDs are defined in our project's R class, which the aapt tool automatically generates.

→ When our application is compiled, aapt generates the R class, which contains resource IDs for all the resources in our res/ directory. For each type of resource, there is an R subclass (for example, R.drawable for all drawable resources), and for each resource of that type, there is a static integer (for example, R.drawable.icon). This integer is the resource ID that we can use to retrieve our resource.

→ Although the R class is where resource IDs are specified, we should never need to look there to discover a resource ID. A resource ID is always composed of:

- The *resource type*: Each resource is grouped into a "type," such as string, drawable, and layout. For more about the different types, see Resource Types.
- The *resource name*, which is either: the filename, excluding the extension; or the value in the XML android:name attribute, if the resource is a simple value (such as a string).

→ There are two ways we can access a resource:

- **In code:** Using a static integer from a sub-class of our R class
- **In XML:** Using a special XML syntax that also corresponds to the resource ID defined in our R class.

❖ Accessing Resource in Code:

→ We can use a resource in code by passing the resource ID as a method parameter.

→ Syntax

[<*package_name*>.]R.<*resource_type*>.<*resource_name*>

- <package_name> is the name of the package in which the resource is located (not required when referencing resources from our own package).
- <resource_type> is the R subclass for the resource type.
- <resource_name> is either the resource filename without the extension or the android:name attribute value in the XML element (for simple values).

→ For example, we can set an ImageView to use the res/drawable/myimage.png resource using setImageResource():

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);
imageView.setImageResource(R.drawable.myimage);
```

❖ Accessing Resource in XML:

→ We can define values for some XML attributes and elements using a reference to an existing resource. We will often do this when creating layout files, to supply strings and images for our widgets.

→ Syntax

@[<package_name>:]<resource_type>/<resource_name>

- <package_name> is the name of the package in which the resource is located (not required when referencing resources from the same package)
- <resource_type> is the R subclass for the resource type
- <resource_name> is either the resource filename without the extension or the android:name attribute value in the XML element (for simple values).

→ For example, if we add a Button to our layout, we should use a string resource for the button text:

```
<Button
 android:layout_width="fill_parent"
 android:layout_height="wrap_content"
 android:text="@string/submit" />
```

❖ Referencing style attributes

→ A style attribute resource allows we to reference the value of an attribute in the currently-applied theme. Referencing a style attribute allows we to customize the look of UI elements by styling them to match standard variations supplied by the current theme, instead of supplying a hard-coded value. Referencing a style attribute essentially says, "use the style that is defined by this attribute, in the current theme."

→ To reference a style attribute, the name syntax is almost identical to the normal resource format, but instead of the at-symbol (@), use a question-mark (?), and the resource type portion is optional. For instance:

?[<*package_name*>:][<*resource_type*>/]<*resource_name*>

→ For example, here's how we can reference an attribute to set the text color to match the "secondary" text color of the system theme:

```
<EditText
id="text"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:textColor="?android:textColorSecondary"
android:text="@string/hello_world" />
```

❖ Accessing original files

→ While uncommon, we might need access our original files and directories. If we do, then saving our files in res/ won't work for we, because the only way to read a resource from res/ is with the resource ID. Instead, we can save our resources in the assets/ directory.

→ Files saved in the assets/ directory are *not* given a resource ID, so we can't reference them through the R class or from XML resources. Instead, we can query files in the assets/ directory like a normal file system and read raw data using AssetManager.

→ However, if all we require is the ability to read raw data (such as a video or audio file), then save the file in the res/raw/ directory and read a stream of bytes using openRawResource().

❖ Accessing platform resources

→ Android contains a number of standard resources, such as styles, themes, and layouts. To access these resource, qualify our resource reference with the android package name.

→ For example, Android provides a layout resource we can use for list items in a ListAdapter:

```
setListAdapter(new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, myarray))
```

→ In this example, simple_list_item_1 is a layout resource defined by the platform for items in a ListView. We can use this instead of creating our own layout for list items.

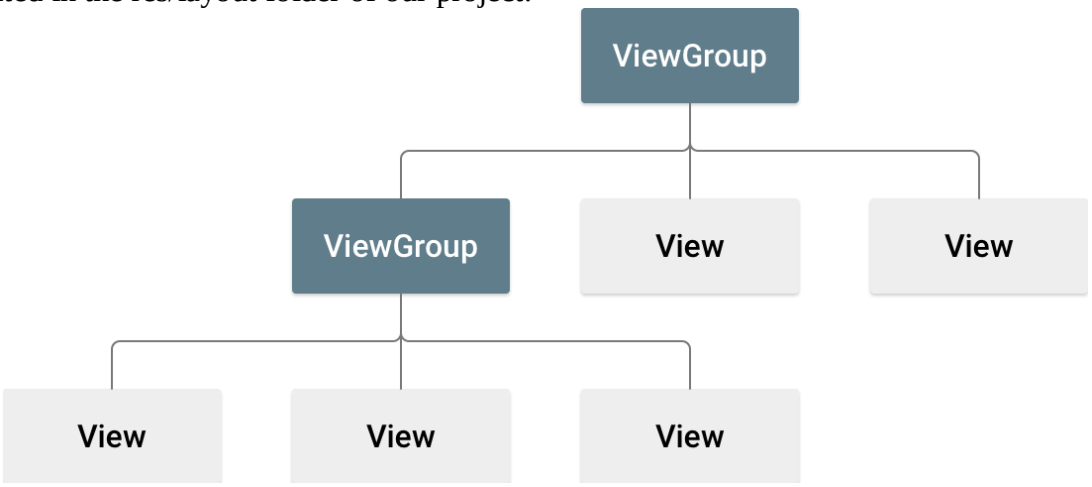
Unit- 2

UI Overview:

→ The basic building block for user interface is a View object which is created from the View class and occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components like buttons, text fields, etc.

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

→ At third level we have different layouts which are subclasses of ViewGroup class and a typical layout defines the visual structure for an Android user interface and can be created either at run time using View/ViewGroup objects or we can declare our layout using simple XML file main_layout.xml which is located in the res/layout folder of our project.



→ All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with. Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects.

→

→ The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView. The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout .

→ We can declare a layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
- **Instantiate layout elements at runtime**. Our app can create View and ViewGroup objects (and manipulate their properties) programmatically.

→ Declaring our UI in XML allows we to separate the presentation of our app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (discussed further in Supporting Different Screen Sizes).

→ The Android framework gives we the flexibility to use either or both of these methods to build our app's UI. For example, we can declare our app's default layouts in XML, and then modify the layout at runtime.

❖ Write the XML

→ Using Android's XML vocabulary, we can quickly design UI layouts and the screen elements they contain, in the same way we create web pages in HTML — with a series of nested elements.

→ Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once we've defined the root element, we can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines our layout.

→ For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
```

```
        android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

→ After we've declared our layout in XML, save the file with the .xml extension, in our Android project's res/layout/ directory, so it will properly compile.

→ More information about the syntax for a layout XML file is available in the Layout Resources document.

❖ Load the XML Resource

→ When we compile our app, each XML layout file is compiled into a View resource. We should load the layout resource from our app code, in our Activity.onCreate() callback implementation. Do so by calling setContentView(), passing it the reference to our layout resource in the form of: R.layout.*layout_file_name*.

→ For example, if our XML layout is saved as main_layout.xml, we would load it for our Activity like so:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

→ The onCreate() callback method in our Activity is called by the Android framework when our Activity is launched.

UI screen elements:

- TextView
- EditText
- AutoCompleteTextView
- Button
- ImageButton
- ImageButton
- ToggleButton
- RadioButton
- RadioGroup
- ProgressDialog
- Spinner
- Time Picker
- Date Picker
- Fragment

Fragment

→ A Fragment represents a reusable portion of our app's UI. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. Fragments cannot live on their own--they must be *hosted* by an activity or another fragment. The fragment's view hierarchy becomes part of, or *attaches to*, the host's view hierarchy.

→ Following are important points about fragment −

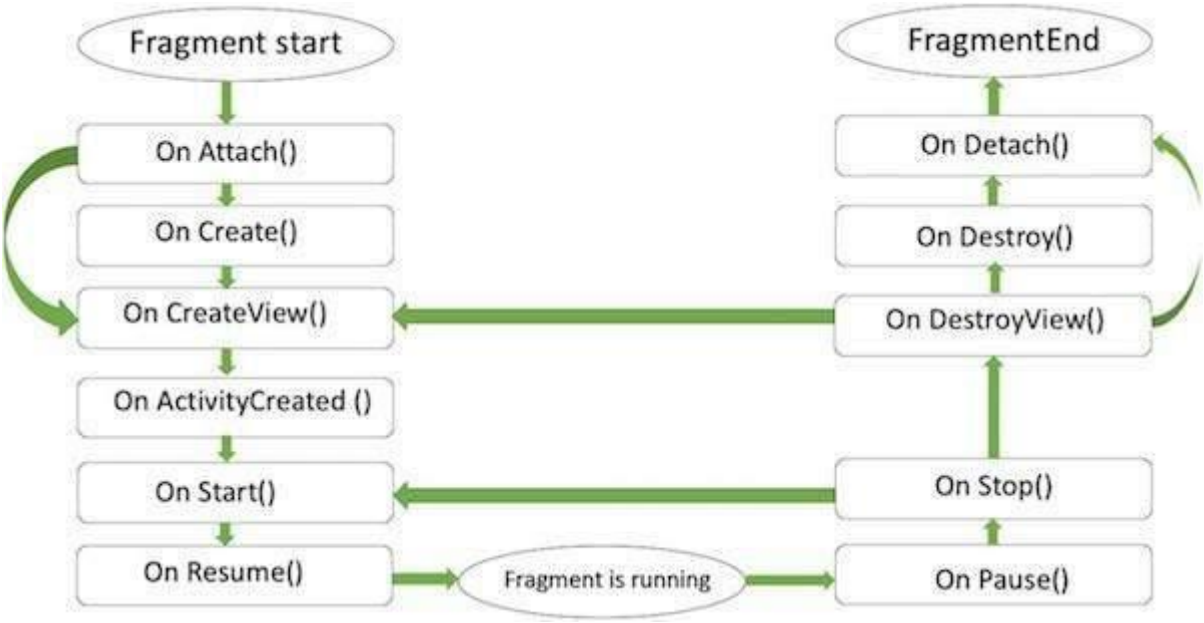
- A fragment has its own layout and its own behaviour with its own life cycle callbacks.
- We can add or remove fragments in an activity while the activity is running.
- We can combine multiple fragments in a single activity to build a multi-pane UI.
- A fragment can be used in multiple activities.
- Fragment life cycle is closely related to the life cycle of its host activity which means when the activity is paused, all the fragments available in the activity will also be stopped.
- A fragment can implement a behaviour that has no user interface component.

- Fragments were added to the Android API in Honeycomb version of Android which API version 11.

→ We create fragments by extending Fragment class and We can insert a fragment into our activity layout by declaring the fragment in the activity's layout file, as a <fragment> element.

❖ Fragment Life Cycle

→ Android fragments have their own life cycle very similar to an android activity. This section briefs different stages of its life cycle.



→ Here is the list of methods which we can to override in our fragment class −

- **onAttach():** The fragment instance is associated with an activity instance.The fragment and the activity is not fully initialized. Typically we 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. We should initialize essential components of the fragment that we 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 our fragment, we must return a View component from this method that is the root of our fragment's layout. We can return null if the fragment does not provide a UI.
- **onActivityCreated():** The onActivityCreated() is called after the onCreateView() method when the host activity is created. Activity and fragment instance have been created as well as the view hierarchy of the activity. At this point, view can be accessed with the findViewById() method. example. In this method we 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 we 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.

❖ Types of Fragments

→ Basically fragments are divided as three stages as shown below.

- **Single frame fragments** − Single frame fragments are for hand hold devices like mobiles, here we can show only one fragment as a view.
- **List fragments** − fragments having special list view is called as list fragment
- **Fragments transaction** − Using with fragment transaction. we can move one fragment to another fragment.

❖ Single Frame Example:

→ LM_Fragment.java

```
public class LM_Fragement extends Fragment {
    public LM_Fragement() {
        // Required empty public constructor
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.lm__fragement, container, false);
    }
}
```

→ lm_fragment.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".LM_Fragement">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:textColor="#000000"
        android:textSize="20px"
        android:text="@string/hello_blank_fragment" />

</FrameLayout>
```

→ PM_Fragment.java

```
public class PM_Fragement extends Fragment {
public PM_Fragement() {
        // Required empty public constructor
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.pm__fragement, container, false);
    }
}
```

→ pm_fragment.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".PM_Fragement">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/hello_blank_fragment"
```

```
        android:textColor="#000000"
        android:textSize="20px"/>

    </FrameLayout>
```

→ **activity_main.xml**

```
    <LinearLayout 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"
      android:orientation="horizontal">

      <fragment
        android:id="@+id/fragment2"
        android:name="com.example.sarvodaya.fragmentexample.PM_Fragement"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_weight="1" />

      <fragment
        android:id="@+id/fragment"
        android:name="com.example.sarvodaya.fragmentexample.LM_Fragement"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_weight="1" />
    </LinearLayout>
```

→ **MainActivity.java**

```
    import android.app.Activity;
    import android.app.FragmentManager;
    import android.app.FragmentTransaction;
    import android.content.res.Configuration;
    import android.os.Bundle;

    public class MainActivity extends Activity {

      @Override
      protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        Configuration config = getResources().getConfiguration();
        /**
         * Check the device orientation and act accordingly
         */

        if (config.orientation == Configuration.ORIENTATION_LANDSCAPE) {
          /**
           * Landscape mode of the device
           */
          LM_Fragement ls_fragment = new LM_Fragement();
          fragmentTransaction.replace(android.R.id.content, ls_fragment);
        }else{
          /**
           * Portrait mode of the device
```

```
            */
            PM_Fragement pm_fragment = new PM_Fragement();
            fragmentTransaction.replace(android.R.id.content, pm_fragment);
        }
        fragmentTransaction.commit();
    }
}
```

Layouts:

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

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

→ Layout Managers (or simply layouts) are said to be extensions of the ViewGroup class. They are used to set the position of child Views within the UI we are building. We can nest the layouts, and therefore we can create arbitrarily complex UIs using a combination of layouts.

→ There is a number of layout classes in the Android SDK. They can be used, modified or can create our own to make the UI for our Views, Fragments and Activities. We can display our contents effectively by using the right combination of layouts.

❖ Types of Android Layout

1) LinearLayout

→ A LinearLayout aligns each of the child View in either a vertical or a horizontal line. A vertical layout has a column of Views, whereas in a horizontal layout there is a row of Views. It supports a weight attribute for each child View that can control the relative size of each child View within the available space.

→ **LinearLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:baselineAligned	This must be a boolean value, either "true" or "false" and prevents the layout from aligning its children's baselines.
android:baselineAlignedChildIndex	When a linear layout is part of another layout that is baseline aligned, it can specify which of its children to baseline align.
android:divider	This is drawable to use as a vertical divider between buttons. We use a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:gravity	This specifies how an object should position its content, on both the

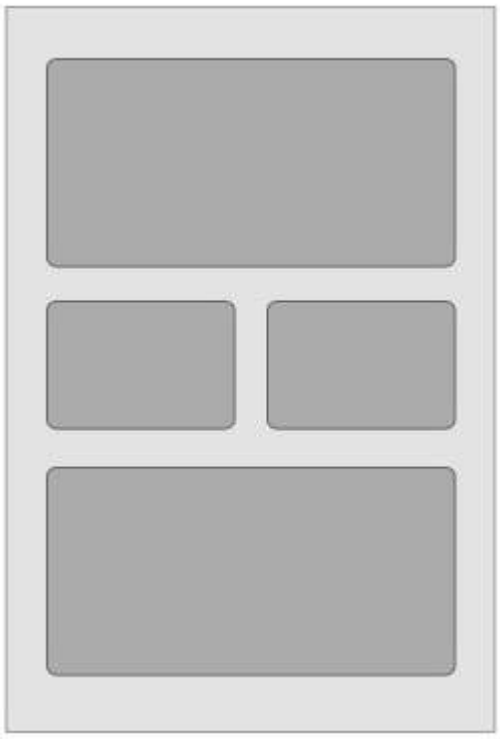
	X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:orientation	This specifies the direction of arrangement and we will use "horizontal" for a row, "vertical" for a column. The default is horizontal.
android:weightSum	Sum up of child weight

→ Example:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical" >

  <Button android:id="@+id/btnStartService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="start_service"/>

  <Button android:id="@+id/btnPauseService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="pause_service"/>

  <Button android:id="@+id/btnStopService"
    android:layout_width="270dp"
    android:layout_height="wrap_content"
    android:text="stop_service"/>

</LinearLayout>
```

2) RelativeLayout

→ It is flexible than other native layouts as it lets us to define the position of each child View relative to the other views and the dimensions of the screen.



→ **RelativeLayout Attributes**

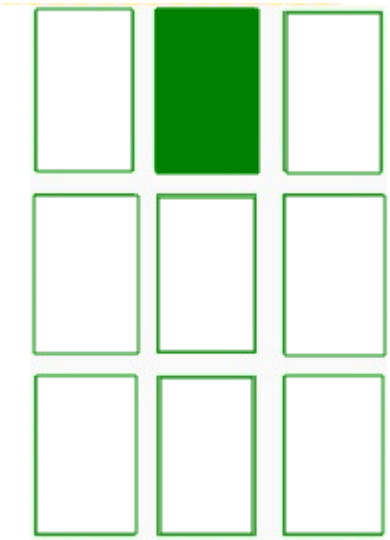
Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:gravity	This specifies how an object should position its content, on both the X and Y axes. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:ignoreGravity	This indicates what view should not be affected by gravity.

Example:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:paddingLeft="16dp"
  android:paddingRight="16dp" >

  <EditText
    android:id="@+id/name"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/reminder" />

  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_alignParentStart="true"
    android:layout_below="@+id/name">

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="New Button"
      android:id="@+id/button" />

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="New Button"
      android:id="@+id/button2" />

  </LinearLayout>

</RelativeLayout>
```

3) FrameLayout

→ It is the simplest of the Layout Managers that pins each child view within its frame. By default the position is the top-left corner, though the gravity attribute can be used to alter its locations. We can add multiple children stacks each new child on top of the one before, with each new View potentially obscuring the previous ones.



→ **FrameLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:foreground	This defines the drawable to draw over the content and possible values may be a color value, in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb".
android:foregroundGravity	Defines the gravity to apply to the foreground drawable. The gravity defaults to fill. Possible values are top, bottom, left, right, center, center_vertical, center_horizontal etc.
android:measureAllChildren	Determines whether to measure all children or just those in the VISIBLE or INVISIBLE state when measuring. Defaults to false.

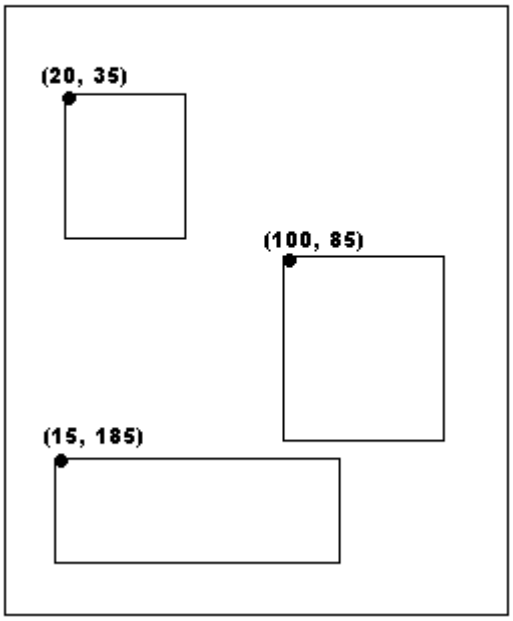
→ **Example:**

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <ImageView
    android:src="@drawable/ic_launcher"
    android:scaleType="fitCenter"
    android:layout_height="250px"
    android:layout_width="250px"/>

  <TextView
    android:text="Frame Demo"
    android:textSize="30px"
    android:textStyle="bold"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:gravity="center"/>
</FrameLayout>
```

4) Absolute Layout

→ An Absolute Layout lets we specify exact locations (x/y coordinates) of its children. Absolute layouts are less flexible and harder to maintain than other types of layouts without absolute positioning.

Absolute Layout



→ **AbsoluteLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:layout_x	This specifies the x-coordinate of the view.
android:layout_y	This specifies the y-coordinate of the view.

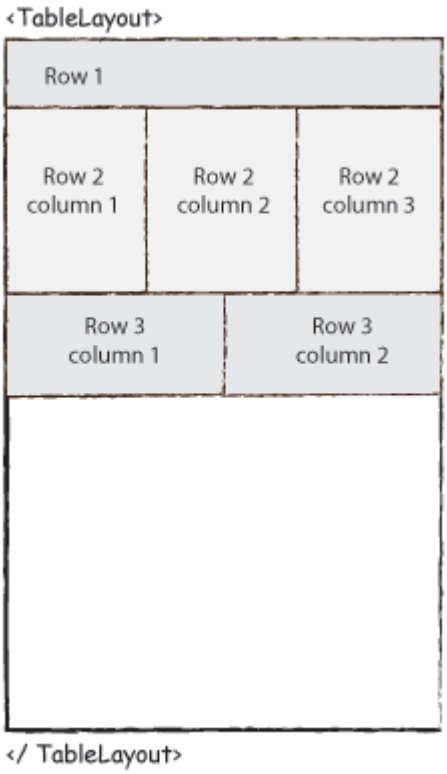
→ Example:

```
<AbsoluteLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <Button
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:text="OK"
    android:layout_x="50px"
    android:layout_y="361px" />
  <Button
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:text="Cancel"
    android:layout_x="225px"
    android:layout_y="361px" />

</AbsoluteLayout>
```

5) Table Layout

→ Android TableLayout going to be arranged groups of views into rows and columns. We will use the <TableRow> element to build a row in the table. Each row has zero or more cells; each cell can hold one View object.

→ TableLayout containers do not display border lines for their rows, columns, or cells.



→ **TableLayout Attributes**

Attribute	Description
android:id	This is the ID which uniquely identifies the layout.
android:collapseColumns	This specifies the zero-based index of the columns to collapse. The column indices must be separated by a comma: 1, 2, 5.
android:shrinkColumns	The zero-based index of the columns to shrink. The column indices must be separated by a comma: 1, 2, 5.
android:stretchColumns	The zero-based index of the columns to stretch. The column indices must be separated by a comma: 1, 2, 5.

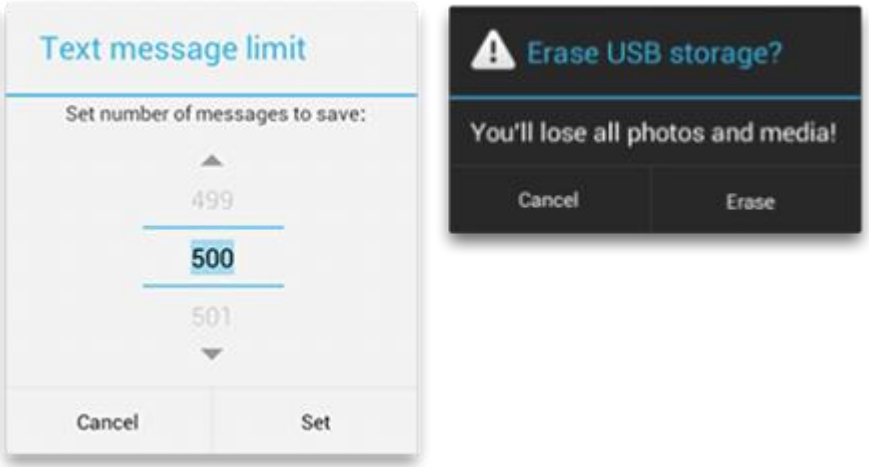
→ Example:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
      android:text="Time"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <TextClock
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/textClock"
      android:layout_column="2" />

  </TableRow>

  <TableRow>
```

```
    <TextView
      android:text="First Name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <EditText
      android:width="200px"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
  </TableRow>

  <TableRow>

    <TextView
      android:text="Last Name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_column="1" />

    <EditText
      android:width="100px"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
  </TableRow>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <RatingBar
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:id="@+id/ratingBar"
      android:layout_column="2" />
  </TableRow>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"/>

  <TableRow
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Submit"
      android:id="@+id/button"
      android:layout_column="2" />
  </TableRow>

</TableLayout>
```

Dialogs:

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



→ The Dialog class is the base class for dialogs, but we should avoid instantiating Dialog directly. Instead, use one of the following subclasses:

❖ AlertDialog

→ Android AlertDialog can be used to display the dialog message with OK and Cancel buttons. It can be used to interrupt and ask the user about his/her choice to continue or discontinue.

→ The AlertDialog class allows we to build a variety of dialog designs and is often the only dialog class we'll need. there are three regions of an alert dialog: Android AlertDialog is the subclass of Dialog class.

i. Title

→ This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If we need to state a simple message or question (such as the dialog in figure 1), we don't need a title.

ii. Content area

→ This can display a message, a list, or other custom layout.

iii. Action buttons

→ There should be no more than three action buttons in a dialog.

→ The AlertDialog.Builder class provides APIs that allow we to create an AlertDialog with these kinds of content, including a custom layout.

Method	Description
public AlertDialog.Builder setTitle(CharSequence)	This method is used to set the title of AlertDialog.
public AlertDialog.Builder setMessage(CharSequence)	This method is used to set the message for AlertDialog.
public AlertDialog.Builder setIcon(int)	This method is used to set the icon over AlertDialog.

Adding buttons

→ To add action buttons we have to call the setPositiveButton() and setNegativeButton() methods:

→ The set...Button() methods require a title for the button (supplied by a string resource) and a DialogInterface.OnClickListener that defines the action to take when the user presses the button.

→ There are three different action buttons we can add:

- **Positive**

        - We should use this to accept and continue with the action (the "OK" action).
    - **Negative**
        - We should use this to cancel the action.
    - **Neutral**
        - We should use this when the user may not want to proceed with the action, but doesn't necessarily want to cancel. It appears between the positive and negative buttons. For example, the action might be "Remind me later."

Adding a list

→ There are three kinds of lists available with the AlertDialog APIs:
- A traditional single-choice list
- A persistent single-choice list (radio buttons)
- A persistent multiple-choice list (checkboxes)

❖ DatePickerDialog or TimePickerDialog

→ A dialog with a pre-defined UI that allows the user to select a date or time.
→ DatePickerDialog and TimePickerDialog classes have onDateSetListener() and onTimeSetListener() callback methods respectively.
→ These callback methods are invoked when the user is done with filling the date and time respectively.
→ The DatePickerDialog class consists of a 5 argument constructor with the parameters listed below.
- **Context**: It requires the application context
- **CallBack Function**: onDateSet() is invoked when the user sets the date with the following parameters:
    - int year : It will be store the current selected year from the dialog
    - int monthOfYear : It will be store the current selected month from the dialog
    - int dayOfMonth : It will be store the current selected day from the dialog
- **int mYear** : It shows the the current year that's visible when the dialog pops up
- **int mMonth** : It shows the the current month that's visible when the dialog pops up
- **int mDay** : It shows the the current day that's visible when the dialog pops up

→ The TimePickerDialog class consists of a 5 argument constructor with the parameters listed below.
- **Context**: It requires the application context
- **CallBack Function**: is invoked when the user sets the time with the following parameters:
    - int hourOfDay : It will be store the current selected hour of the day from the dialog
    - int minute : It will be store the current selected minute from the dialog
- **int mHours** : It shows the the current Hour that's visible when the dialog pops up
- **int mMinute** : It shows the the current minute that's visible when the dialog pops up
- **boolean false** : If its set to false it will show the time in 24 hour format else not

→ **Example:**
→ **AlertDialog Example**
→ **activity_main.xml**
→

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="10dp"
   android:paddingLeft="10dp"
   android:paddingRight="10dp"
   android:paddingTop="10dp"
   tools:context=".MainActivity" >

   <Button
      android:id="@+id/btn"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentTop="true"
      android:layout_centerHorizontal="true"
```

```
        android:layout_marginTop="70dp"
        android:onClick="setDate"
        android:text="See Dialog"
        />
</RelativeLayout>
```

→ **Main_Activity.java**

```
public class MainActivity extends Activity {
Button btn;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
btn=(Button)findViewById(R.id.btn);
AlertDialog.Builder builder=new AlertDialog.Builder(getApplicationContext());
btn.setOnClickListener(new View.OnClickListener() {
   @Override
   public void onClick(View v) {
      builder.setMessage("Do we want to Close this app?")
            .setCancelable(false)
            .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int id) {
                  finish();
                  Toast.makeText(getApplicationContext(),"We choose yes action for
alert",Toast.LENGTH_LONG).show();
               }
            })
            .setNegativeButton("No", new DialogInterface.OnClickListener() {
               @Override
               public void onClick(DialogInterface dialog, int which) {
                  dialog.cancel();
                  Toast.makeText(getApplicationContext(),"We choose no action for
alert",Toast.LENGTH_LONG).show();
               }
            });
      AlertDialog alert=builder.create();
      alert.setTitle("AlertDialogExample");
      alert.show();
   }
});
}
}
```

→ **DatePickerDialog Example:**

➢ **activity_main.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:paddingBottom="10dp"
   android:paddingLeft="10dp"
   android:paddingRight="10dp"
   android:paddingTop="10dp"
   tools:context=".MainActivity" >
```

```
    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="70dp"
        android:onClick="setDate"
        android:text="Set Date"
         />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="24dp"
        android:text="Press the button to set the date"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/button1"
        android:layout_marginTop="66dp"
        android:layout_toLeftOf="@+id/button1"
        android:text="The Date is:"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/button1"
        android:layout_below="@+id/textView2"
        android:layout_marginTop="72dp"
        android:text=""
        android:textAppearance="?android:attr/textAppearanceMedium" />

</RelativeLayout>
```

- **Main_Activity.java**

```
import android.app.Activity;
import android.app.DatePickerDialog;
import android.app.Dialog;
import android.os.Bundle;
import android.view.View;
import android.widget.DatePicker;
import android.widget.TextView;
import android.widget.TimePicker;
import android.widget.Toast;
import java.util.Calendar;

public class MainActivity extends Activity {
    private DatePicker datePicker;
```

```
    private TimePicker timePicker;
    private Calendar calendar;
    private TextView dateView,timeView;
    private int year, month, day;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        dateView = (TextView) findViewById(R.id.textView3);
        calendar = Calendar.getInstance();
        year = calendar.get(Calendar.YEAR);

        month = calendar.get(Calendar.MONTH);
        day = calendar.get(Calendar.DAY_OF_MONTH);
        showDate(year, month + 1, day);
    }
    @Override
    protected Dialog onCreateDialog(int id) {
        // TODO Auto-generated method stub
        if (id == 999) {
            return new DatePickerDialog(this,
                    myDateListener, year, month, day);
        }
        return null;
    }
    public void setDate(View view) {
        showDialog(999);
        Toast.makeText(getApplicationContext(), "ca",
                Toast.LENGTH_SHORT)
                .show();
    }
    private DatePickerDialog.OnDateSetListener myDateListener = new
            DatePickerDialog.OnDateSetListener() {
                @Override
                public void onDateSet(DatePicker arg0,
                                      int arg1, int arg2, int arg3) {
                    // TODO Auto-generated method stub
                    // arg1 = year
                    // arg2 = month
                    // arg3 = day
                    showDate(arg1, arg2 + 1, arg3);
                }
            };

    private void showDate(int year, int month, int day) {
        dateView.setText(new StringBuilder().append(day).append("/")
                .append(month).append("/").append(year));
    }
}
```

Animation:

→ Animations can add visual cues that notify users about what's going on in our app. They are especially useful when the UI changes state, such as when new content loads or new actions become available. Animations also add a polished look to our app, which gives it a higher quality look and feel.

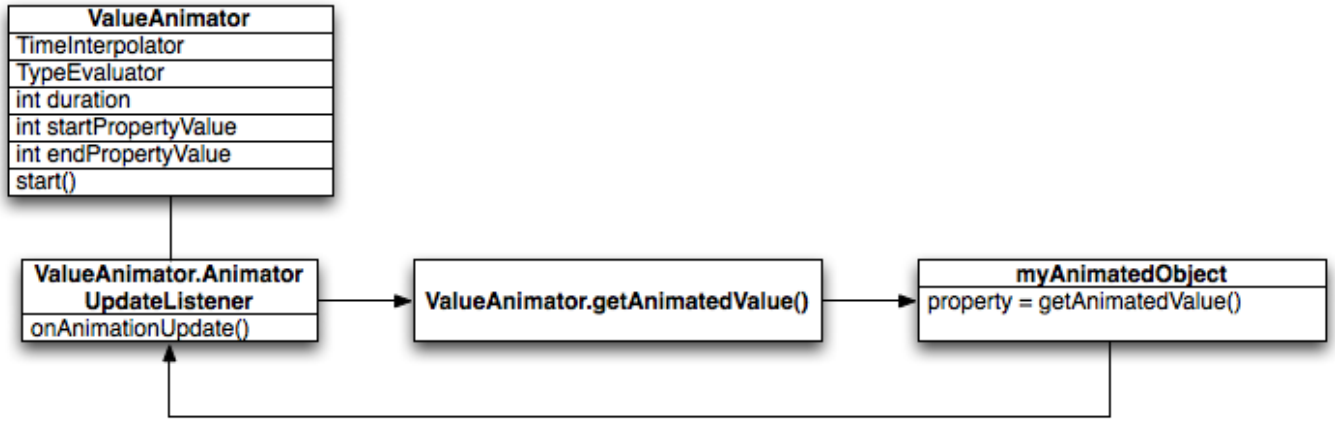
→ Android includes different animation APIs depending on what type of animation we want, so this page provides an overview of the different ways we can add motion to our UI.

1) Property Animation

→ The property animation system is a robust framework that allows us to animate almost anything. We can define an animation to change any object property over time, regardless of whether it draws to the screen or not. A property animation changes a property's (a field in an object) value over a specified length of time. To animate something, we specify the object property that we want to animate, such as an object's position on the screen, how long we want to animate it for, and what values we want to animate between.

→ Characteristics of an animation:

- **Duration:** We can specify the duration of an animation. The default length is 300 ms.
- **Time interpolation:** We can specify how the values for the property are calculated as a function of the animation's current elapsed time.
- **Repeat count and behavior:** We can specify whether or not to have an animation repeat when it reaches the end of a duration and how many times to repeat the animation. We can also specify whether we want the animation to play back in reverse. Setting it to reverse plays the animation forwards then backwards repeatedly, until the number of repeats is reached.
- **Animator sets:** We can group animations into logical sets that play together or sequentially or after specified delays.
- **Frame refresh delay:** We can specify how often to refresh frames of our animation. The default is set to refresh every 10 ms, but the speed in which our application can refresh frames is ultimately dependent on how busy the system is overall and how fast the system can service the underlying timer.



→ file location:
res/animator/*filename*.xml
The filename will be used as the resource ID.

→ compiled resource datatype:
Resource pointer to a ValueAnimator, ObjectAnimator, or AnimatorSet.

→ resource reference:
In Java: R.animator.*filename*
In XML: @[*package*:]animator/*filename*

→ syntax:

```
<set
  android:ordering=["together" | "sequentially"]>

    <objectAnimator
       android:propertyName="string"
       android:duration="int"
       android:valueFrom="float | int | color"
       android:valueTo="float | int | color"
       android:startOffset="int"
       android:repeatCount="int"
```

```
    android:repeatMode=["repeat" | "reverse"]
    android:valueType=["intType" | "floatType"]/>

  <animator
    android:duration="int"
    android:valueFrom="float | int | color"
    android:valueTo="float | int | color"
    android:startOffset="int"
    android:repeatCount="int"
    android:repeatMode=["repeat" | "reverse"]
    android:valueType=["intType" | "floatType"]/>

  <set>
    ...
  </set>
</set>
```

→ The file must have a single root element: either <set>, <objectAnimator>, or <valueAnimator>. We can group animation elements together inside the <set> element, including other <set> elements.

→ **Elements:**

1. <set>

⇒ A container that holds other animation elements (<objectAnimator>, <valueAnimator>, or other <set> elements). Represents an AnimatorSet.

⇒ We can specify nested <set> tags to further group animations together. Each <set> can define its own ordering attribute.

⇒ **Attributes:**

- **android:ordering**
  Keyword. Specifies the play ordering of animations in this set.

Value	Description
sequentially	Play animations in this set sequentially
together (default)	Play animations in this set at the same time.

2. <objectAnimator>

⇒ Animates a specific property of an object over a specific amount of time. Represents an ObjectAnimator.

⇒ **Attributes:**

- **android:propertyName**
  String. ***Required***. *The object's property to animate, referenced by its name. For example we can specify "alpha" or "backgroundColor" for a View object.*
  The objectAnimator element does not expose a target attribute, however, so we cannot set the object to animate in the XML declaration. We have to inflate our animation XML resource by calling loadAnimator() and call setTarget() to set the target object that contains this property.
- **android:valueTo**
  float, int, or color. ***Required***. *The value where the animated property ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:valueFrom**
  float, int, or color. *The value where the animated property starts. If not specified, the animation starts at the value obtained by the property's get method. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:duration**
  int. *The time in milliseconds of the animation. 300 milliseconds is the default.*
- **android:startOffset**
  int. *The amount of milliseconds the animation delays after start() is called.*
- **android:repeatCount**
  int. *How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once*

after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.

- **android:repeatMode**
  int. *How an animation behaves when it reaches the end of the animation. android:repeatCount must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.*
- **android:valueType**
  Keyword. *Do not specify this attribute if the value is a color. The animation framework automatically handles color values*

Value	Description
intType	Specifies that the animated values are integers
floatType (default)	Specifies that the animated values are floats

3. <animator>

⇒ Animates an over a specified amount of time. Represents a ValueAnimator.

⇒ **Attributes:**

- **android:valueTo**
  float, int, or color. ***Required***. *The value where the animation ends. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:valueFrom**
  float, int, or color. ***Required***. *The value where the animation starts. Colors are represented as six digit hexadecimal numbers (for example, #333333).*
- **android:duration**
  int. *The time in milliseconds of the animation. 300ms is the default.*
- **android:startOffset**
  int. *The amount of milliseconds the animation delays after start() is called.*
- **android:repeatCount**
  int. *How many times to repeat an animation. Set to "-1" to infinitely repeat or to a positive integer. For example, a value of "1" means that the animation is repeated once after the initial run of the animation, so the animation plays a total of two times. The default value is "0", which means no repetition.*
- **android:repeatMode**
  int. *How an animation behaves when it reaches the end of the animation. android:repeatCount must be set to a positive integer or "-1" for this attribute to have an effect. Set to "reverse" to have the animation reverse direction with each iteration or "repeat" to have the animation loop from the beginning each time.*
- **android:valueType**
  Keyword. *Do not specify this attribute if the value is a color. The animation framework automatically handles color values.*

Value	Description
intType	Specifies that the animated values are integers
floatType (default)	Specifies that the animated values are floats

→ The Animator class provides the basic structure for creating animations. We normally do not use this class directly as it only provides minimal functionality that must be extended to fully support animating values. The following subclasses extend Animator:

❖ ValueAnimator

→ This class provides a simple timing engine for running animations which calculate animated values and set them on target objects.

→ There is a single timing pulse that all animations use. It runs in a custom handler to ensure that property changes happen on the UI thread.

→ By default, ValueAnimator uses non-linear time interpolation, via the AccelerateDecelerateInterpolator class, which accelerates into and decelerates out of an animation. This behavior can be changed by calling ValueAnimator#setInterpolator(TimeInterpolator).

→ Animators can be created from either code or resource files. Here is an example of a ValueAnimator resource file:

```
<animator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueFrom="1"
    android:valueTo="0"
    android:valueType="floatType"
    android:repeatCount="1"
    android:repeatMode="reverse"/>
```

→ **Methods:**

public Object getAnimatedValue (String propertyName)	The most recent value calculated by this ValueAnimator for propertyName.
public long getCurrentPlayTime()	Gets the current position of the animation in time, which is equal to the current time minus the time that the animation started.
public long getDuration ()	Gets the length of the animation.
public int getRepeatCount ()	Defines how many times the animation should repeat.
public int getRepeatMode ()	Defines what this animation should do when it reaches the end.
public long getStartDelay ()	The amount of time, in milliseconds, to delay starting the animation after start() is called.
public long getTotalDuration ()	Gets the total duration of the animation, accounting for animation sequences, start delay, and repeating.
public PropertyValuesHolder[] getValues ()	Returns the values that this ValueAnimator animates between.
public boolean isRunning ()	Returns whether this Animator is currently running (having been started and gone past any initial startDelay period and not yet ended).
public boolean isStarted ()	Returns whether this Animator has been started and not yet ended.
public void start ()	Starts this animation.
public void pause ()	Pauses a running animation.
public void resume ()	Resumes a paused animation, causing the animator to pick up where it left off when it was paused.
public void reverse ()	Plays the ValueAnimator in reverse.
public ValueAnimator setDuration (long duration)	Sets the length of the animation.
public void setRepeatCount (int value)	Sets how many times the animation should be repeated.
public void setRepeatMode (int value)	Defines what this animation should do when it reaches the end.

❖ ObjectAnimator

→ This subclass of ValueAnimator provides support for animating properties on target objects. The constructors of this class take parameters to define the target object that will be animated as well as the name of the property that will be animated. Appropriate set/get functions are then determined internally and the animation will call these functions as necessary to animate the property.

→ Animators can be created from either code or resource files, as shown here:

```
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueTo="200"
    android:valueType="floatType"
    android:propertyName="y"
    android:repeatCount="1"
    android:repeatMode="reverse"/>
```

→ **Methods of ObjectAnimator Class:**

public String getPropertyName ()	Gets the name of the property that will be animated.

public Object getTarget ()	The target object whose property will be animated by this animation
public ObjectAnimator setDuration (long duration)	Sets the length of the animation.
public void setFloatValues (float... values)	Sets float values that will be animated between.
public void setIntValues (int... values)	Sets int values that will be animated between.
public void setProperty (Property property)	Sets the property that will be animated.
public void setPropertyName (String propertyName)	Sets the name of the property that will be animated.
public void start ()	Starts this animation.

❖ AnimatorSet

→ This class plays a set of Animator objects in the specified order. Animations can be set up to play together, in sequence, or after a specified delay.

→ There are two different approaches to adding animations to a AnimatorSet: either the playTogether() or playSequentially() methods can be called to add a set of animations all at once, or the AnimatorSet#play(Animator) can be used in conjunction with methods in the Builder class to add animations one by one.

→ It is possible to set up a AnimatorSet with circular dependencies between its animations. For example, an animation a1 could be set up to start before animation a2, a2 before a3, and a3 before a1. The results of this configuration are undefined, but will typically result in none of the affected animations being played. Because of this (and because circular dependencies do not make logical sense anyway), circular dependencies should be avoided, and the dependency flow of animations should only be in one direction.

→ **Example:**

```
AnimatorSet bouncer = new AnimatorSet();
bouncer.play(bounceAnim).before(squashAnim1);
bouncer.play(squashAnim1).with(squashAnim2);
bouncer.play(squashAnim1).with(stretchAnim1);
bouncer.play(squashAnim1).with(stretchAnim2);
bouncer.play(bounceBackAnim).after(stretchAnim2);
ValueAnimator fadeAnim = ObjectAnimator.ofFloat(newBall, "alpha", 1f, 0f);
fadeAnim.setDuration(250);
AnimatorSet animatorSet = new AnimatorSet();
animatorSet.play(bouncer).before(fadeAnim);
animatorSet.start();
```

→ **Methods**

public long getDuration ()	Gets the length of each of the child animations of this AnimatorSet.
public AnimatorSet setDuration (long duration)	Sets the length of each of the current child animations of this AnimatorSet.
public long getCurrentPlayTime ()	Returns the milliseconds elapsed since the start of the animation.
public void setCurrentPlayTime (long playTime)	Sets the position of the animation to the specified point in time.
public long getStartDelay ()	The amount of time, in milliseconds, to delay starting the animation after start() is called.
public void setStartDelay (long startDelay)	The amount of time, in milliseconds, to delay starting the animation after start() is called.
public long getTotalDuration ()	Gets the total duration of the animation, accounting for animation sequences, start delay, and repeating. Return DURATION_INFINITE if the duration is infinite.
public boolean isRunning ()	Returns true if any of the child animations of this AnimatorSet have been started and have not yet ended.

public boolean isStarted ()	Returns whether this Animator has been started and not yet ended.
public void pause ()	Pauses a running animation.
public AnimatorSet.Builder play (Animator anim)	This method creates a Builder object, which is used to set up playing constraints.
public void playSequentially (Animator... items)	Sets up this AnimatorSet to play each of the supplied animations when the previous animation ends.
public void playTogether (Animator... items)	Sets up this AnimatorSet to play all of the supplied animations at the same time.
public void resume ()	Resumes a paused animation, causing the animator to pick up where it left off when it was paused.
public void reverse ()	Plays the AnimatorSet in reverse.

2) View Animation

→ We can use the view animation system to perform tweened animation on Views. Tween animation calculates the animation with information such as the start point, end point, size, rotation, and other common aspects of an animation.

→ The view animation framework supports both tween and frame by frame animations, which can both be declared in XML. The following sections describe how to use both methods.

1) Tween animation: Creates an animation by performing a series of transformations on a single image with an Animation
2) Frame animation: or creates an animation by showing a sequence of images in order with an AnimationDrawable.

❖ Tween animation

→ An animation defined in XML that performs transitions such as rotating, fading, moving, and stretching on a graphic.

→ file location:

res/anim/*filename*.xml

The filename will be used as the resource ID.

→ compiled resource datatype:

Resource pointer to an Animation.

→ resource reference:

In Java: R.anim.*filename*

In XML: @[*package*:]anim/*filename*

→ syntax:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:interpolator="@[package:]anim/interpolator_resource"
  android:shareInterpolator=["true" | "false"] >
  <alpha
    android:fromAlpha="float"
    android:toAlpha="float" />
  <scale
    android:fromXScale="float"
    android:toXScale="float"
    android:fromYScale="float"
    android:toYScale="float"
    android:pivotX="float"
    android:pivotY="float" />
  <translate
    android:fromXDelta="float"
    android:toXDelta="float"
```

```
        android:fromYDelta="float"
        android:toYDelta="float" />
    <rotate
        android:fromDegrees="float"
        android:toDegrees="float"
        android:pivotX="float"
        android:pivotY="float" />
    <set>
        ...
    </set>
</set>
```

→ The file must have a single root element: either an <alpha>, <scale>, <translate>, <rotate>, or <set> element that holds a group (or groups) of other animation elements (even nested <set> elements).

→ **Elements:**

1. **<set>**
   - ⇒ A container that holds other animation elements (<alpha>, <scale>, <translate>, <rotate>) or other <set> elements. Represents an AnimationSet.
   - ⇒ **Attributes:**
     - **android:interpolator**
       Interpolator resource. *An Interpolator to apply on the animation. The value must be a reference to a resource that specifies an interpolator (not an interpolator class name). There are default interpolator resources available from the platform or we can create our own interpolator resource.*
     - **android:shareInterpolator**
       Boolean. *"true" if we want to share the same interpolator among all child elements.*

2. **<alpha>**
   - ⇒ A fade-in or fade-out animation. Represents an AlphaAnimation.
   - ⇒ **Attributes:**
     - **android:fromAlpha**
       Float. *Starting opacity offset, where 0.0 is transparent and 1.0 is opaque.*
     - **android:toAlpha**
       Float. *Ending opacity offset, where 0.0 is transparent and 1.0 is opaque.*

3. **<scale>**
   - ⇒ A resizing animation. We can specify the center point of the image from which it grows outward (or inward) by specifying pivotX and pivotY. For example, if these values are 0, 0 (top-left corner), all growth will be down and to the right. Represents a ScaleAnimation.
   - ⇒ **Attributes:**
     - **android:fromXScale**
       Float. *Starting X size offset, where 1.0 is no change.*
     - **android:toXScale**
       Float. *Ending X size offset, where 1.0 is no change.*
     - **android:fromYScale**
       Float. *Starting Y size offset, where 1.0 is no change.*
     - **android:toYScale**
       Float. *Ending Y size offset, where 1.0 is no change.*
     - **android:pivotX**
       Float. *The X coordinate to remain fixed when the object is scaled.*
     - **android:pivotY**
       Float. *The Y coordinate to remain fixed when the object is scaled.*

4. **<translate>**
   - ⇒ A vertical and/or horizontal motion. Supports the following attributes in any of the following three formats: values from -100 to 100 ending with "%", indicating a percentage relative to

itself; values from -100 to 100 ending in "%p", indicating a percentage relative to its parent; a float value with no suffix, indicating an absolute value. Represents a TranslateAnimation.

⇒ **Attributes:**

- **android:fromXDelta**
  Float or percentage. Starting X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").
- **android:toXDelta**
  Float or percentage. *Ending X offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element width (such as "5%"), or in percentage relative to the parent width (such as "5%p").*
- **android:fromYDelta**
  Float or percentage. *Starting Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").*
- **android:toYDelta**
  Float or percentage. *Ending Y offset. Expressed either: in pixels relative to the normal position (such as "5"), in percentage relative to the element height (such as "5%"), or in percentage relative to the parent height (such as "5%p").*

5. <rotate>

⇒ A rotation animation. Represents a RotateAnimation.

⇒ **Attributes:**

- **android:fromDegrees**
  Float. *Starting angular position, in degrees.*
- **android:toDegrees**
  Float. *Ending angular position, in degrees.*
- **android:pivotX**
  Float or percentage. *The X coordinate of the center of rotation. Expressed either: in pixels relative to the object's left edge (such as "5"), in percentage relative to the object's left edge (such as "5%"), or in percentage relative to the parent container's left edge (such as "5%p").*
- **android:pivotY**
  Float or percentage. *The Y coordinate of the center of rotation. Expressed either: in pixels relative to the object's top edge (such as "5"), in percentage relative to the object's top edge (such as "5%"), or in percentage relative to the parent container's top edge (such as "5%p").*

→ *Example*:

XML file saved at res/anim/hyperspace_jump.xml:

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:shareInterpolator="false">
  <scale
    android:interpolator="@android:anim/accelerate_decelerate_interpolator"
    android:fromXScale="1.0"
    android:toXScale="1.4"
    android:fromYScale="1.0"
    android:toYScale="0.6"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fillAfter="false"
    android:duration="700" />
  <set
    android:interpolator="@android:anim/accelerate_interpolator"
    android:startOffset="700">
    <scale
      android:fromXScale="1.4"
      android:toXScale="0.0"
      android:fromYScale="0.6"
```

```
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:duration="400" />
        <rotate
            android:fromDegrees="0"
            android:toDegrees="-45"
            android:toYScale="0.0"
            android:pivotX="50%"
            android:pivotY="50%"
            android:duration="400" />
    </set>
</set>
```

→ This application code will apply the animation to an ImageView and start the animation:

```
ImageView image = (ImageView) findViewById(R.id.image);
Animation hyperspaceJump = AnimationUtils.loadAnimation(this, R.anim.hyperspace_jump);
image.startAnimation(hyperspaceJump);
```

→ Methods of Animation Class:

public void start ()	This method starts the animation.
public void setDuration (long durationMillis)	This method sets the duration of an animation.
public long getDuration ()	This method gets the duration which is set by above method
public void reset ()	Reset the initialization state of this animation.
public void setRepeatCount (int repeatCount)	Sets how many times the animation should be repeated.
public int getRepeatCount ()	Defines how many times the animation should repeat. The default value is 0.
public void setRepeatMode (int repeatMode)	Defines what this animation should do when it reaches the end.
public int getRepeatMode ()	Defines what this animation should do when it reaches the end.

❖ Frame animation

→ An animation defined in XML that shows a sequence of images in order (like a film).

→ file location:

res/drawable/*filename*.xml

The filename will be used as the resource ID.

→ compiled resource datatype:

Resource pointer to an AnimationDrawable.

→ resource reference:

In Java: R.drawable.*filename*

In XML: @[*package*:]drawable.*filename*

→ **Syntax:**

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot=["true" | "false"] >
    <item
        android:drawable="@[package:]drawable/drawable_resource_name"
        android:duration="integer" />
</animation-list>
```

→ **Elements:**

1. <animation-list>

⇒ **Required**. This must be the root element. Contains one or more <item> elements.

⇒ **Attributes:**

- **android:oneshot**
  Boolean. "true" if we want to perform the animation once; "false" to loop the animation.

2. <item>

⇒ A single frame of animation. Must be a child of a <animation-list> element.

⇒ **Attributes:**

- **android:drawable**
  Drawable resource. The drawable to use for this frame.
- **android:duration**
  Integer. The duration to show this frame, in milliseconds.

→ **Example:**

XML file saved at res/anim/rocket.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
   android:oneshot="false">
   <item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
   <item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
   <item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>
```

This application code will set the animation as the background for a View, then play the animation:

```
ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
rocketImage.setBackgroundResource(R.drawable.rocket_thrust);

rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
rocketAnimation.start();
```

Unit- 3

Data and Storage API

→ Android uses a file system that's similar to disk-based file systems on other platforms. Android provides several options for we to save persistent application data.

→ The solution we choose depends on our specific needs, such as whether the data should be private to our application or accessible to other applications (and the user) and how much space our data requires. The system provides several options for we to save our app data:

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

❖ SharedPreferences:

→ Android provides many ways of storing data of an application. One of this way is called Shared Preferences. Shared Preferences allow us to save and retrieve data in the form of key,value pair.

→ In order to use shared preferences, we have to call a method getSharedPreferences() that returns a SharedPreference instance pointing to the file that contains the values of preferences.

→ SharedPreferences sharedpreferences = getSharedPreferences(MyPREFERENCES, Context.MODE_PRIVATE);

→ The first parameter is the key and the second parameter is the MODE. Apart from private there are other modes available that are listed below –

MODE_APPEND	This will append the new preferences with the already existing preferences
MODE_ENABLE_WRITE_AHEAD_LOGGING	Database open flag. When it is set , it would enable write ahead logging by default
MODE_MULTI_PROCESS	This method will check for modification of preferences even if the sharedpreference instance has already been loaded
MODE_PRIVATE	By setting this mode, the file can only be accessed using calling application
MODE_WORLD_READABLE	This mode allow other application to read the preferences
MODE_WORLD_WRITEABLE	This mode allow other application to write the preferences

→ To get a SharedPreferences object for our application, use one of two methods:

- getSharedPreferences() - Use this if we need multiple preferences files identified by name, which we specify with the first parameter.
- getPreferences() - Use this if we need only one preferences file for our Activity. Because this will be the only preferences file for our Activity, we don't supply a name.

→ **Methods of SharedPreferences class:**

contains(String key)	This method is used to check whether the preferences contain a preference.
edit()	This method is used to create a new Editor for these preferences, through which we can make modifications to the data in the preferences and atomically commit those changes back to the

	SharedPreferences object.
getAll()	This method is used to retrieve all values from the preferences.
getBoolean(String key, boolean defValue)	This method is used to retrieve a boolean value from the preferences.
getFloat(String key, float defValue)	This method is used to retrieve a float value from the preferences.
getInt(String key, int defValue)	This method is used to retrieve an int value from the preferences.
getLong(String key, long defValue)	This method is used to retrieve a long value from the preferences.
getString(String key, String defValue)	This method is used to retrieve a String value from the preferences.
getStringSet(String key, Set defValues)	This method is used to retrieve a set of String values from the preferences.
registerOnSharedPreferencechangeListener(SharedPreferences.OnSharedPreferencechangeListener listener)	This method is used to registers a callback to be invoked when a change happens to a preference.
unregisterOnSharedPreferencechangeListener(SharedPreferences.OnSharedPreferencechangeListener listener)	This method is used to unregisters a previous callback.

→ Nested classes of Shared Preferences
- SharedPreferences.Editor: Interface used to write(edit) data in the SP file. Once editing has been done, one must commit() or apply() the changes made to the file.
- SharedPreferences.OnSharedPreferenceChangeListener(): Called when a shared preference is changed, added, or removed. This may be called even if a preference is set to its existing value. This callback will be run on our main thread.

→ We can save something in the sharedpreferences by using SharedPreferences.Editor class. We will call the edit method of SharedPreference instance and will receive it in an editor object. Its syntax is –

```
Editor editor = sharedpreferences.edit();
editor.putString("key", "value");
editor.commit();
```

→ **Methods of Editor class:**

apply()	It is an abstract method. It will commit our changes back from editor to the sharedPreference object we are calling
clear()	It will remove all values from the editor
remove(String key)	It will remove the value whose key has been passed as a parameter
putLong(String key, long value)	It will save a long value in a preference editor
putInt(String key, int value)	It will save a integer value in a preference editor
putFloat(String key, float value)	It will save a float value in a preference editor
putString(String key, String value)	It will save a string value in a preference editor

→ **Example:**
res/layout/activiy_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
  android:layout_height="match_parent
 tools:context=".MainActivity">
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Shared Preference "
        android:id="@+id/textView"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:textSize="35dp" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/editText"
        android:layout_below="@+id/textView"
        android:layout_marginTop="67dp"
        android:hint="Name"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/editText2"
        android:layout_below="@+id/editText"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:hint="Pass" />

    <EditText
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/editText3"
        android:layout_below="@+id/editText2"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:hint="Email" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Save"
        android:id="@+id/button"
        android:layout_below="@+id/editText3"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="50dp" />

</RelativeLayout>
```

MainActivity.java

```
import android.app.Activity;
import android.content.Context;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;
```

```
public class MainActivity extends Activity {
  EditText ed1,ed2,ed3;
  Button b1;

  public static final String MyPREFERENCES = "MyPrefs" ;
  public static final String Name = "nameKey";
  public static final String Phone = "phoneKey";
  public static final String Email = "emailKey";

  SharedPreferences sharedpreferences;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    ed1=(EditText)findViewById(R.id.editText);
    ed2=(EditText)findViewById(R.id.editText2);
    ed3=(EditText)findViewById(R.id.editText3);

    b1=(Button)findViewById(R.id.button);
    sharedpreferences = getSharedPreferences(MyPREFERENCES, Context.MODE_PRIVATE);

    b1.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        String n  = ed1.getText().toString();
        String ph  = ed2.getText().toString();
        String e  = ed3.getText().toString();

        SharedPreferences.Editor editor = sharedpreferences.edit();

        editor.putString(Name, n);
        editor.putString(Phone, ph);
        editor.putString(Email, e);
        editor.commit();
        Toast.makeText(MainActivity.this,"Thanks",Toast.LENGTH_LONG).show();
      }
    });
  }

}
```

res/layout/activiy_second.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Read Shared Preference "
        android:id="@+id/textView"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:textSize="35dp" />

    <TextView
```

```
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/txtnm"
        android:layout_below="@+id/textView"
        android:layout_marginTop="67dp"
        android:hint="Name"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/txtPass"
        android:layout_below="@+id/txtnm"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:hint="Pass" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/txtemail"
        android:layout_below="@+id/txtPass"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentRight="true"
        android:layout_alignParentEnd="true"
        android:hint="Email" />

</RelativeLayout>
```

SecondActivity.java

```
public class SecondActivity extends Activity {

    TextView t1,t2,t3;

    public static final String MyPREFERENCES = "MyPrefs" ;
    public static final String Name = "nameKey";
    public static final String Phone = "phoneKey";
    public static final String Email = "emailKey";

    SharedPreferences sharedpreferences;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        t1 = (TextView)findViewById(R.id.txtnm);
        t2 = (TextView)findViewById(R.id.txtPass);
        t3 = (TextView)findViewById(R.id.txtemail);
        SharedPreferences sh = getSharedPreferences(MyPREFERENCES, MODE_PRIVATE);
        String s1 = sh.getString(Name, "");
        String s2=sh.getString(Phone,"");
        String s3=sh.getString(Email,"");

        // Setting the fetched data
        // in the EditTexts
        t1.setText(s1);
        t2.setText(s2);
```

```
        t3.setText(s3);

    }
}
```

❖ Internal Storage

→ Android provides many kinds of storage for applications to store their data. These storage places are shared preferences, internal and external storage, SQLite storage, and storage via network connection.

→ Internal storage is the storage of the private data on the device memory.

→ By default these files are private and are accessed by only our application and get deleted , when user delete our application.

→ Advantages:

1. It is always available.
2. Files saved here are accessible by only our app bydefault.
3. When the user uninstall the app, System removes all apps files from internal storage.

➢ Writing file

→ In order to use internal storage to write some data in the file, call the openFileOutput() method with the name of the file and the mode. The mode could be private , public e.t.c.

→ Syntax:

FileOutputStream fOut = openFileOutput("filename",MODE_WORLD_READABLE);

→ The method openFileOutput() returns an instance of FileOutputStream. So we receive it in the object of FileInputStream. After that we can call write method to write data on the file.

→ Syntax:

```
String str = "data";
fOut.write(str.getBytes());
fOut.close();
```

→ Methods of FileOutputStream class:

Method	Description
getChannel()	This method returns a write-only FileChannel that shares its position with this stream
getFD()	This method returns the underlying file descriptor
write(byte[] buffer, int byteOffset, int byteCount)	This method Writes count bytes from the byte array buffer starting at position offset to this stream
close()	This method is used to close file.

➢ Reading file

→ In order to read from the file we just created , call the openFileInput() method with the name of the file. It returns an instance of FileInputStream.

→ Syntax:

FileInputStream fin = openFileInput(file);

→ After that, we can call read method to read one character at a time from the file and then we can print it. Its syntax is given below −

```
int c;
String temp="";
while( (c = fin.read()) != -1){
  temp = temp + Character.toString((char)c);
}
//string temp contains all the data of the file.
fin.close();
```

→ **Methods of FileInputStream class**

Method	Description
available()	This method returns an estimated number of bytes that can be read or skipped without blocking for more input
getChannel()	This method returns a read-only FileChannel that shares its position with this stream
getFD()	This method returns the underlying file descriptor
read(byte[] buffer, int byteOffset, int byteCount)	This method reads at most length bytes from this stream and stores them in the byte array b starting at offset
close()	This method is used to close file.

❖ External Storage

→ As like internal storage we are able to save and read data from the device external memory such as SD card. The data is stored in a file specified by the user itself and user can access this file. This files are only accessible till the application exists on we have SD card mounted on our device.

→ External storage is not always available because user can mount external storage and USB storage and in some cases remove it from the device.When User uninstall the app, the System removes files from here only we have them in the Directory from getExternalFilesDir().

→ External storage is the best place for file that do not require access restrictions and for files that we want to share with other apps or allow the user to access with a computer.

→ It is necessary to add external storage read and write permission. For that we need to add permission in Android|Manifest.xml file.

-For writing:
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
-For reading:
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />

External Storage Availability

→ In order to avoid crashing the app first, we need to check storage SD Card is available for reading and write operations. The method getExternalStorageState() is used to determine the state of mounted storage media such as SD Card is missing, read-only or readable, and writable. Below is the code snippet which we will use to check the availability of external storage.

→ **Example:**

```
boolean mExternalStorageAvailable = false;
boolean mExternalStorageWriteable = false;
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
   // We can read and write the media
   mExternalStorageAvailable = mExternalStorageWriteable = true;
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
   // We can only read the media
   mExternalStorageAvailable = true;
   mExternalStorageWriteable = false;
} else {
   // Something else is wrong. It may be one of many other states, but all we need
   //  to know is we can neither read nor write
   mExternalStorageAvailable = mExternalStorageWriteable = false;
}
```

→ **Methods to Store data in External Storage**

- **getExternalStoragePublicDirectory():** This is the present recommended method to keep files public and these files are not deleted even when the app is uninstalled from the system. For eg: Images clicked by the camera are still available even after we uninstall the camera.

- **getExternalFilesDir(String type):** This method is used to store private data that are specific to the app only. And data are removed as we uninstall the app.

- **getExternalStorageDirectory():** This method is not recommended. It is now absolute and it is used to access external storage in older versions, API Level less than 7.

→ The data files saved over external storage devices are publicly accessible on shared external storage using USB mass storage transfer. Data files stored over external storage using a **FileOutputStream** object and can be read using a **FileInputStream** object.(Detail available in Internal Storage)

❖ SQLite Database

→ SQLite is a opensource SQL database that stores data to a text file on a device. Android comes in with built in SQLite database implementation.

→ SQLite supports all the relational database features. In order to access this database, we don't need to establish any kind of connections for it like JDBC,ODBC e.t.c

→ It is embedded in android bydefault. So, there is no need to perform any database setup or administration task.

→ To use a SQLite database in an Android application, it is necessary to create a class that inherits from the SQLiteOpenHelper class, a standard Android class that arranges to open the database file. This class is available in "android.database.SQLite" package.
public class SqlOpenHelper extends SQLiteOpenHelper {}

➢ SQLiteOpenHelper:

→ It checks for the existence of the database file and, if it exists, it opens it; otherwise, it creates one

→ There are two constructors of SQLiteOpenHelper class:

Constructor	Detail
SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)	creates an object for creating, opening and managing the database.
SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version, DatabaseErrorHandler errorHandler)	creates an object for creating, opening and managing the database. It specifies the error handler.

→ **Methods:**

Method	Detail
onCreate()	Called when the database is created for the first time. This is where the creation of tables and the initial population of the tables should happen. **public abstract void onCreate (SQLiteDatabase db)**
onUpgrade()	Called when the database needs to be upgraded. The implementation should use this method to drop tables, add tables, or do anything else it needs to upgrade to the new schema version. **public abstract void onUpgrade (SQLiteDatabase db, int oldVersion, int newVersion)**
onDowngrade()	Called when the database needs to be downgraded. This is strictly similar to onUpgrade(SQLiteDatabase, int, int) method, but is called whenever current version is newer than requested one. **public void onDowngrade (SQLiteDatabase db, int oldVersion, int newVersion)**
onOpen()	Called when the database has been opened. The implementation should check SQLiteDatabase#isReadOnly before updating the database. **public void onOpen (SQLiteDatabase db)**
close()	Close any open database object.

	public void close ()
onConfigure()	Called when the database connection is being configured, to enable features such as write-ahead logging or foreign key support. **public void onConfigure (SQLiteDatabase db)**
getDatabaseName()	Return the name of the SQLite database being opened, as given to the constructor. **public String getDatabaseName ()**
getWritableDatabase()	Create and/or open a database that will be used for reading and writing. The first time this is called, the database will be opened and onCreate(SQLiteDatabase), onUpgrade(SQLiteDatabase, int, int) and/or onOpen(SQLiteDatabase) will be called. **public SQLiteDatabase getWritableDatabase ()**
getReadableDatabase()	Create and/or open a database. This will be the same object returned by getWritableDatabase() unless some problem, such as a full disk, requires the database to be opened read-only. In that case, a read-only database object will be returned. **public SQLiteDatabase getReadableDatabase ()**
setIdleConnectionTimeout()	Sets the maximum number of milliseconds that SQLite connection is allowed to be idle before it is closed and removed from the pool. This method should be called from the constructor of the subclass, before opening the database **public void setIdleConnectionTimeout (long idleConnectionTimeoutMs)**
setWriteAheadLoggingEnabled()	Enables or disables the use of write-ahead logging for the database. Write-ahead logging cannot be used with read-only databases so the value of this flag is ignored if the database is opened read-only. **public void setWriteAheadLoggingEnabled (boolean enabled)**

→ To create a table in SQL, we use the CREATE TABLE statement. Note that Android's API for SQLite assumes that our primary key will be a long integer (long in Java). The primary key column can have any name for now, but when we wrap the data in a ContentProvider this column is required to be named _id, so we'll start on the right foot by using that name now:
CREATE TABLE some_table_name (_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, name TEXT);

→ Example:

```
public class DBHelper extends SQLiteOpenHelper {

  public static final String DATABASE_NAME = "MyDBName.db";
  public static final String CONTACTS_TABLE_NAME = "contacts";
  public static final String CONTACTS_COLUMN_ID = "id";
  public static final String CONTACTS_COLUMN_NAME = "name";
```

```
public static final String CONTACTS_COLUMN_EMAIL = "email";
public static final String CONTACTS_COLUMN_STREET = "street";
public static final String CONTACTS_COLUMN_CITY = "place";
public static final String CONTACTS_COLUMN_PHONE = "phone";
private HashMap hp;

public DBHelper(Context context) {
  super(context, DATABASE_NAME , null, 1);
}

@Override
public void onCreate(SQLiteDatabase db) {
      db.execSQL("create table "+CONTACTS_TABLE_NAME +" " +"(id integer primary key, name
text,phone text,email text, street text,place text)"
  );
}

@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
  db.execSQL("DROP TABLE IF EXISTS contacts");
  onCreate(db);
}
```

➢ SQLiteDatabase:

→ SQLiteDatabase has methods to create, delete, execute SQL commands, and perform other common database management tasks.

→ **Methods:**

Method	Detail
void execSQL(String sql)	executes the sql query not select query. **Parameters** **sql:** the SQL statement to be executed. Multiple statements separated by semicolons are not supported.
long insert(String table, String nullColumnHack, ContentValues values)	inserts a record on the database. The table specifies the table name, nullColumnHack doesn't allow completely null values. If second argument is null, android will store null values if values are empty. The third argument specifies the values to be stored. **Parameters** **table :** Name of the Table **nullColumnHack:** optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If our provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where our values is empty. **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values.
int update(String table, ContentValues values, String whereClause, String[] whereArgs)	updates a row. **Parameters** **table:** Name of the Table **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values. **whereClause:** the optional WHERE clause to apply when updating. Passing null will update all rows. **whereArgs:** We may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.
int delete(String table,	Used to delete table.

String whereClause, String[] whereArgs)	**Parameters** **table:** Name of the Table **whereClause:** the optional WHERE clause to apply when updating. Passing null will delete all rows. **whereArgs:** We may include ?s in the where clause, which will be replaced by the values from whereArgs. The values will be bound as Strings.
Cursor query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy)	returns a cursor over the resultset. **Parameters** **table:** Table Name **columns:** A list of which columns to return. Passing null will return all columns, which is discouraged to prevent reading data from storage that isn't going to be used. **selection:** A filter declaring which rows to return, formatted as an SQL WHERE clause (excluding the WHERE itself). Passing null will return all rows for the given table. **selectionArgs:** We may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings. **groupBy:** A filter declaring how to group rows, formatted as an SQL GROUP BY clause (excluding the GROUP BY itself). Passing null will cause the rows to not be grouped. **having:** A filter declare which row groups to include in the cursor, if row grouping is being used, formatted as an SQL HAVING clause (excluding the HAVING itself). Passing null will cause all row groups to be included, and is required when row grouping is not being used. **orderBy:** How to order the rows, formatted as an SQL ORDER BY clause (excluding the ORDER BY itself). Passing null will use the default sort order, which may be unordered.
public Cursor query (String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)	returns a cursor over the resultset. **Parameters** **limit:** Limits the number of rows returned by the query, formatted as LIMIT clause. Passing null denotes no LIMIT clause. Other parameters are describe above.
public Cursor query (boolean distinct, String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy, String limit)	returns a cursor over the resultset. **Parameters** **distinct:** true if we want each row to be unique, false otherwise. Other parameters are describe above.
public Cursor rawQuery (String sql, String[] selectionArgs)	Runs the provided SQL and returns a Cursor over the result set. **Parameters** **sql:** the SQL statement to be executed. Multiple statements separated by semicolons are not supported. **selectionArgs:** We may include ?s in where clause in the query, which will be replaced by the values from selectionArgs. The values will be bound as Strings.
public static int releaseMemory()	Attempts to release memory that SQLite holds but does not require to operate properly. Typically this memory will come from the page cache.
public long replace (	Convenience method for replacing a row in the database. Inserts a new row

String table, String nullColumnHack, ContentValues initialValues)	if a row does not already exist. **Parameters** **table :** Name of the Table **nullColumnHack:** optional; may be null. SQL doesn't allow inserting a completely empty row without naming at least one column name. If our provided values is empty, no column names are known and an empty row can't be inserted. If not set to null, the nullColumnHack parameter provides the name of nullable column name to explicitly insert a NULL into in the case where our values is empty. **values:** map contains the initial column values for the row. The keys should be the column names and the values the column values.
public boolean isOpen ()	Returns true if the database is currently open.
public boolean isReadOnly ()	Returns true if the database is opened as read only.
public boolean isWriteAheadLoggingEnabled ()	Returns true if write-ahead logging has been enabled for this database.
public static boolean deleteDatabase (File file)	Deletes a database including its journal file and other auxiliary files that may have been created by the database engine. **Parameters** file: The database file path. This value cannot be null.
public long getMaximumSize ()	Returns the maximum size the database may grow to.
public long getPageSize ()	Returns the current database page size, in bytes.
public String getPath()	Gets the path to the database file.
public int getVersion()	Gets the database version.

➢ Cursor interface:

→ This interface provides random read-write access to the result set returned by a database query.

→ Cursor implementations are not required to be synchronized so code using a Cursor from multiple threads should perform its own synchronization when using the Cursor.

→ **Methods:**

Method	Detail
public abstract void close()	Closes the Cursor, releasing all of its resources and making it completely invalid. Unlike deactivate() a call to requery() will not make the Cursor valid again.
public abstract int getColumnCount()	Return total number of columns
public abstract int getColumnIndex(String columnName)	Returns the zero-based index for the given column name, or -1 if the column doesn't exist. If you expect the column to exist use getColumnIndexOrThrow(java.lang.String) instead, which will make the error more clear. **Parameters** **columnName:** the name of the target column.
public abstract String getColumnName (int columnIndex)	Returns the column name at the given zero-based column index. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract String[]	Returns a string array holding the names of all of the columns in the result set in

getColumnNames()	the order in which they were listed in the result.
public abstract int getCount()	Returns the numbers of rows in the cursor.
public abstract double getDouble (int columnIndex)	Returns the value of the requested column as a double. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract float getFloat (int columnIndex)	Returns the value of the requested column as a float. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract int getInt (int columnIndex)	Returns the value of the requested column as an int. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract long getLong (int columnIndex)	Returns the value of the requested column as a long. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract short getShort (int columnIndex)	Returns the value of the requested column as a short. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract String getString (int columnIndex)	Returns the value of the requested column as a String. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract int getType (int columnIndex)	Returns data type of the given column's value. The preferred type of the column is returned but the data may be converted to other types as documented in the get-type methods such as getInt(int), getFloat(int) etc. **Parameters** **columnIndex:** the zero-based index of the target column. Value is 0 or greater
public abstract Bundle getExtras ()	Returns a bundle of extra values. This is an optional way for cursors to provide out-of-band metadata to their users. One use of this is for reporting on the progress of network requests that are required to fetch data for the cursor.
public abstract int getPosition()	Returns the current position of the cursor in the row set. The value is zero-based. When the row set is first returned the cursor will be at positon -1, which is before the first row. After the last row is returned another call to next() will leave the cursor past the last entry, at a position of count().
public abstract boolean move (int offset)	Move the cursor by a relative amount, forward or backward, from the current position. Positive offsets move forwards, negative offsets move backwards. If the final position is outside of the bounds of the result set then the resultant position will be pinned to -1 or count() depending on whether the value is off the front or end of the set, respectively. **Parameter** **offset:** the offset to be applied from the current position.
public abstract boolean moveToFirst ()	Move the cursor to the first row.
public abstract boolean moveToLast ()	Move the cursor to the last row.
public abstract boolean moveToNext ()	Move the cursor to the next row.
public abstract boolean moveToPrevious ()	Move the cursor to the previous row.
public abstract boolean moveToPosition (int position)	Move the cursor to an absolute position. The valid range of values is -1 <= position <= count. This method will return true if the request destination was reachable, otherwise, it returns false. **Parameter** **position:** The zero-based position to move to. Value is -1 or greater
public abstract boolean isAfterLast ()	Returns whether the cursor is pointing to the position after the last row.

public abstract boolean isBeforeFirst ()	Returns whether the cursor is pointing to the position before the first row.
public abstract boolean isClosed ()	return true if the cursor is closed.
public abstract boolean isFirst ()	Returns whether the cursor is pointing to the first row.
public abstract boolean isLast ()	Returns whether the cursor is pointing to the last row.
public abstract boolean isNull (int columnIndex)	Returns true if the value in the indicated column is null.