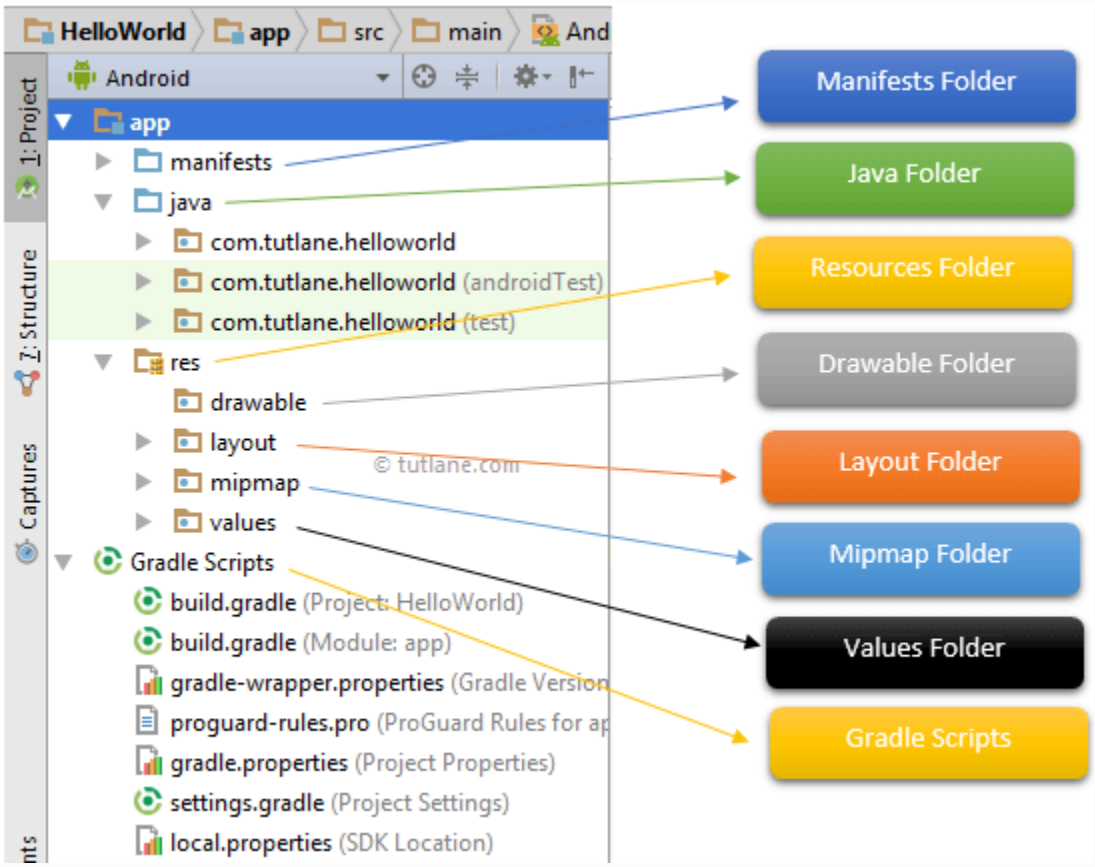
UNIT-II

❖ Anatomy of an Android

To implement android apps, Android Studio is the official IDE (Integrated Development Environment) which is freely provided by Google for android app development.

Once we setup android development environment using android studio and if we create a sample application using android studio, our project folder structure will be like as shown below. In case if you are not aware of creating an application using an android studio please check this Android Hello World App.

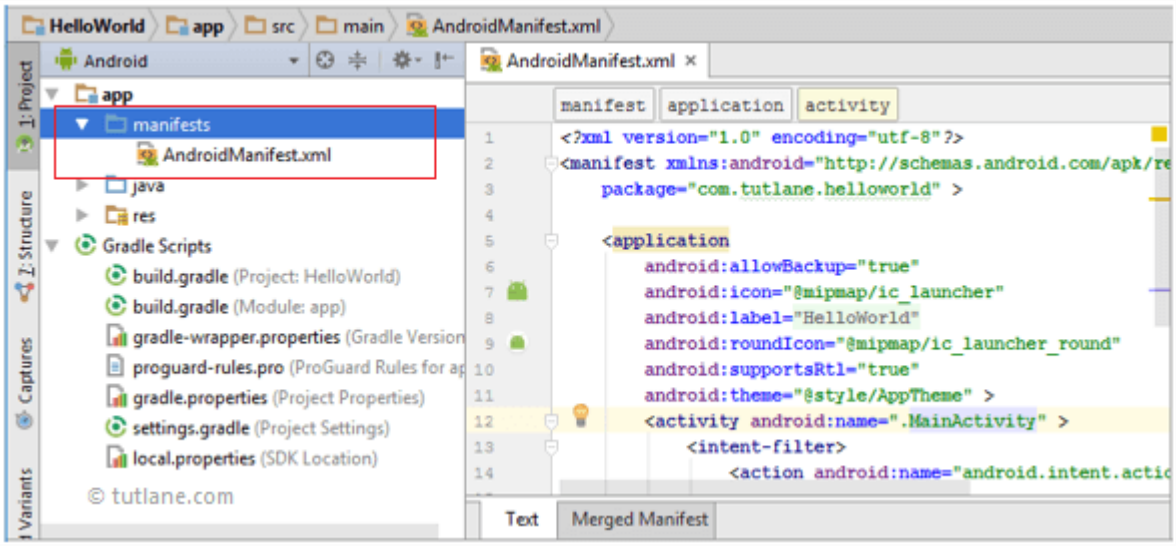


The Android project structure on the disk might be differs from the above representation. To see the actual file structure of the project, select Project from the Project dropdown instead of Android.

The android app project will contain different types of app modules, source code files, and resource files. We will explore all the folders and files in the android app.

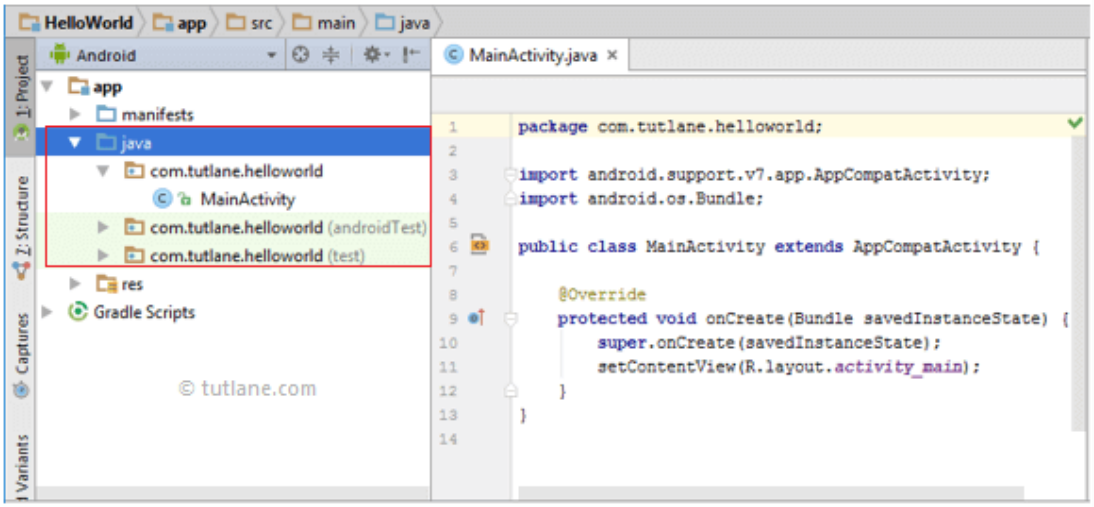
1. Manifests Folder

This folder will contain a manifest file (AndroidManifest.xml) for our android application. This manifest file will contain information about our application such as android version, access permissions, metadata, etc. of our application and its components. The manifest file will act as an intermediate between android OS and our application.Following is the structure of the manifests folder in the android application.



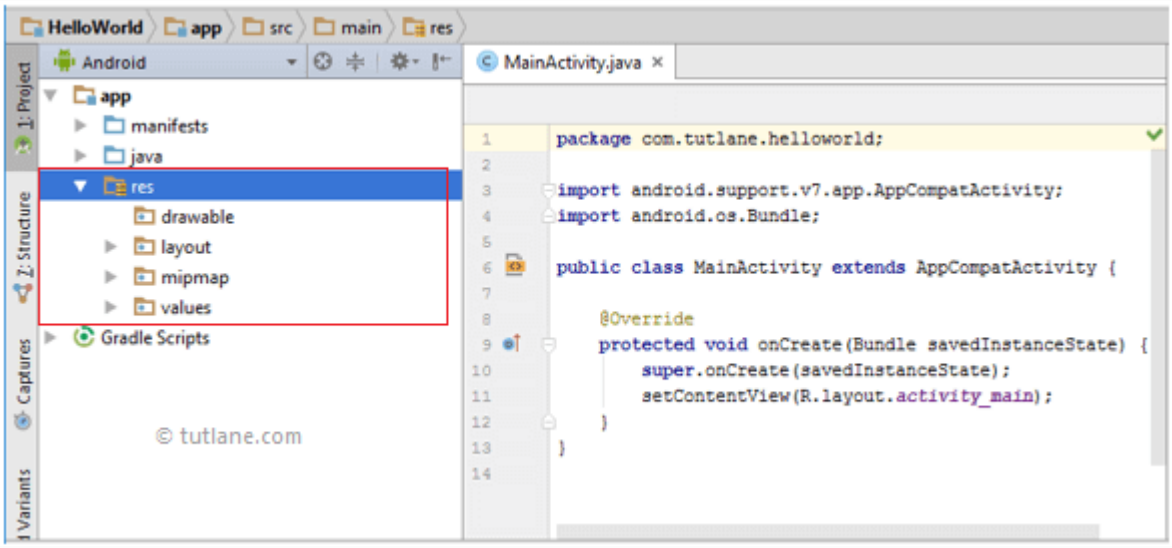
2. Java Folder

This folder will contain all the java source code (.java) files which we'll create during the application development, including JUnit test code. Whenever we create any new project/application, by default the class file MainActivity.java will create automatically under the package name "com.tutlane.helloworld" like as shown below.



3. res (Resources) Folder

It's an important folder that will contain all non-code resources, such as bitmap images, UI strings, XML layouts like as shown below.



Alt Text

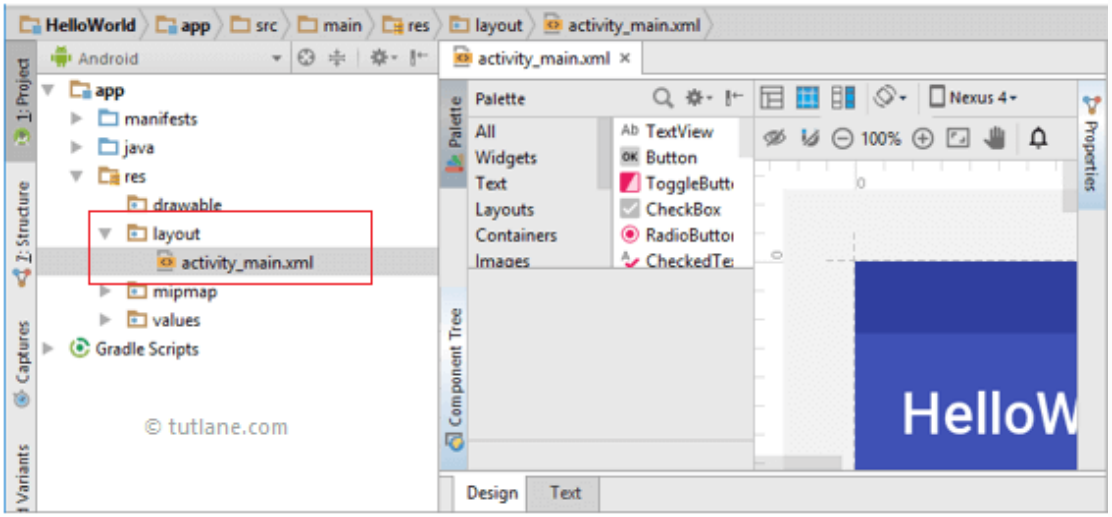
The res (Resources) will contain a different type of folders that are

3.1. Drawable Folder (res/drawable)

It will contain the different types of images as per the requirement of application. It's a best practice to add all the images in a drawable folder other than app/launcher icons for the application development.

3.2. Layout Folder (res/layout)

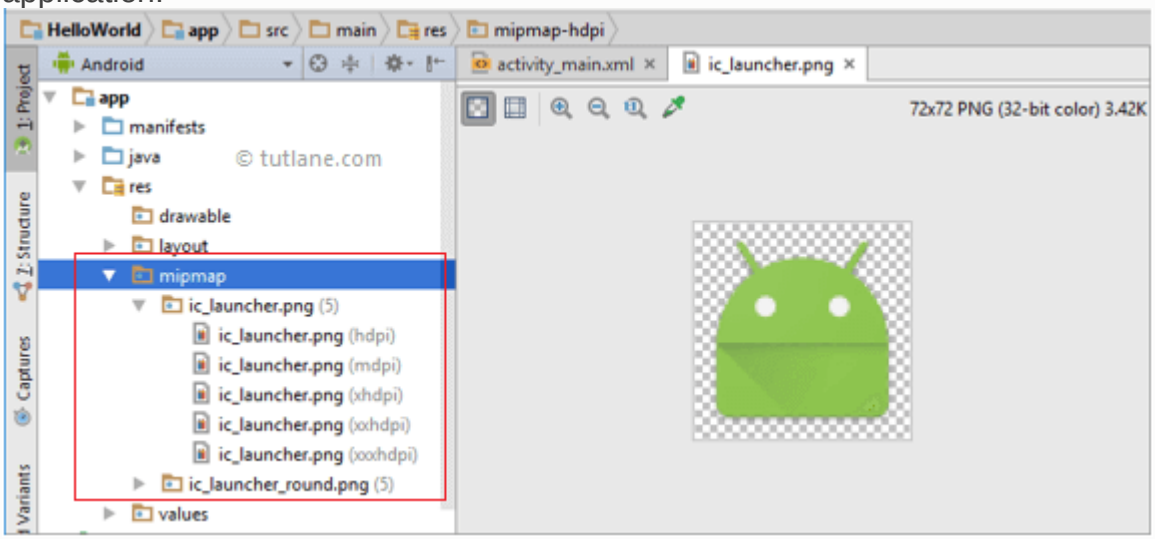
This folder will contain all XML layout files which we used to define the user interface of our application. Following is the structure of the layout folder in the android application.



3.3. Mipmap Folder (res/mipmap)

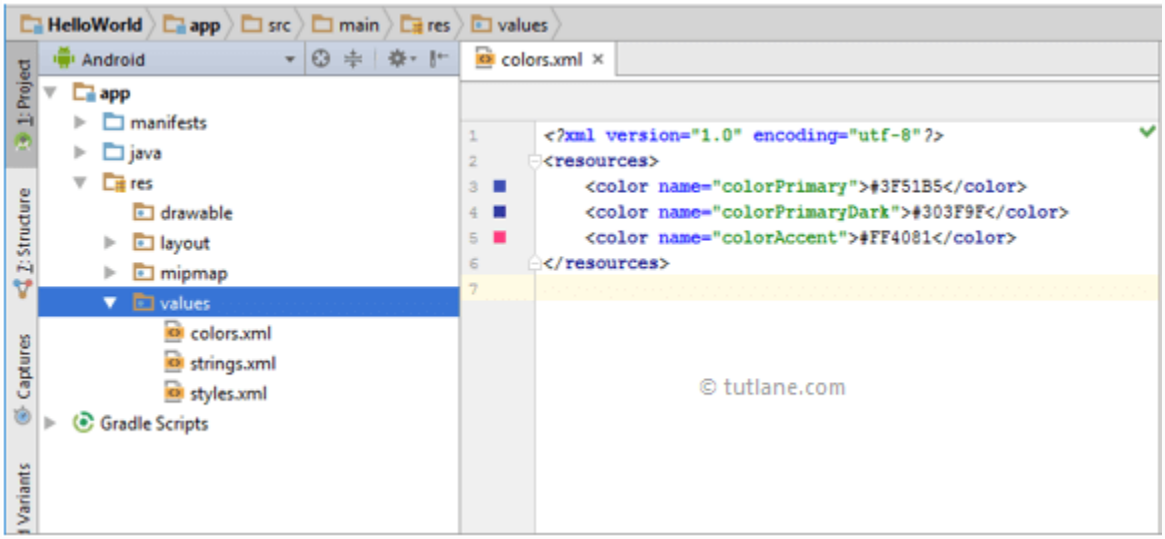
This folder will contain app / launcher icons that are used to show on the home screen. It will contain different density type of icons such as hdpi, mdpi, xhdpi, xxhdpi, xxxhdpi, to use different icons based on the size of the device.

Following is the structure of the mipmap folder in the android application.



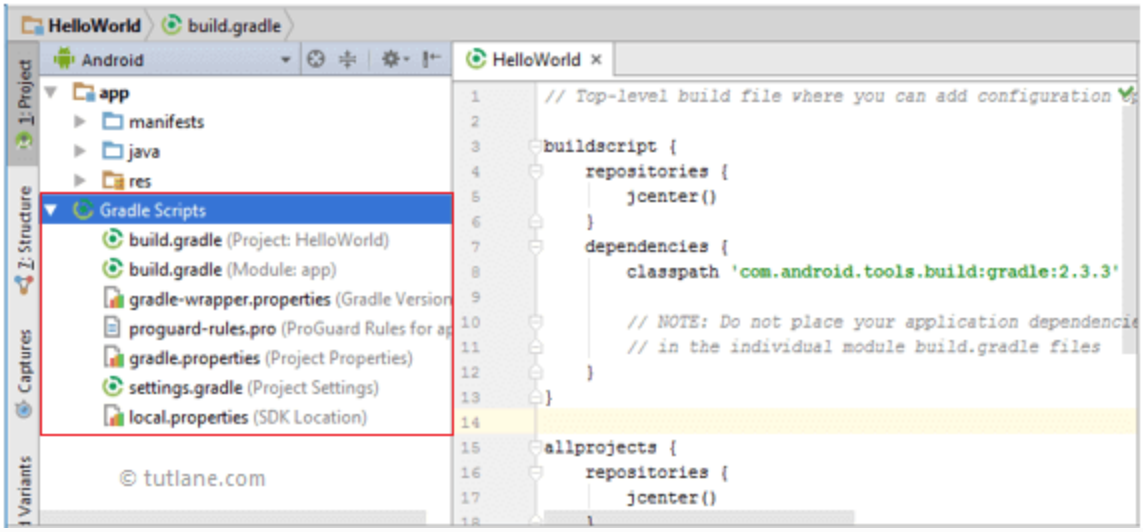
3.4. Values Folder (res/values)

This folder will contain various XML files, such as strings, colors, style definitions and a static array of strings or integers. Following is the structure of the values folder in android application.



4. Gradle Scripts

In android, Gradle means automated build system and by using this we can define a build configuration that applies to all modules in our application. In Gradle build.gradle (Project), and build.gradle (Module) files are useful to build configurations that apply to all our app modules or specific to one app module.Following is the structure of Gradle Scripts in the android application.

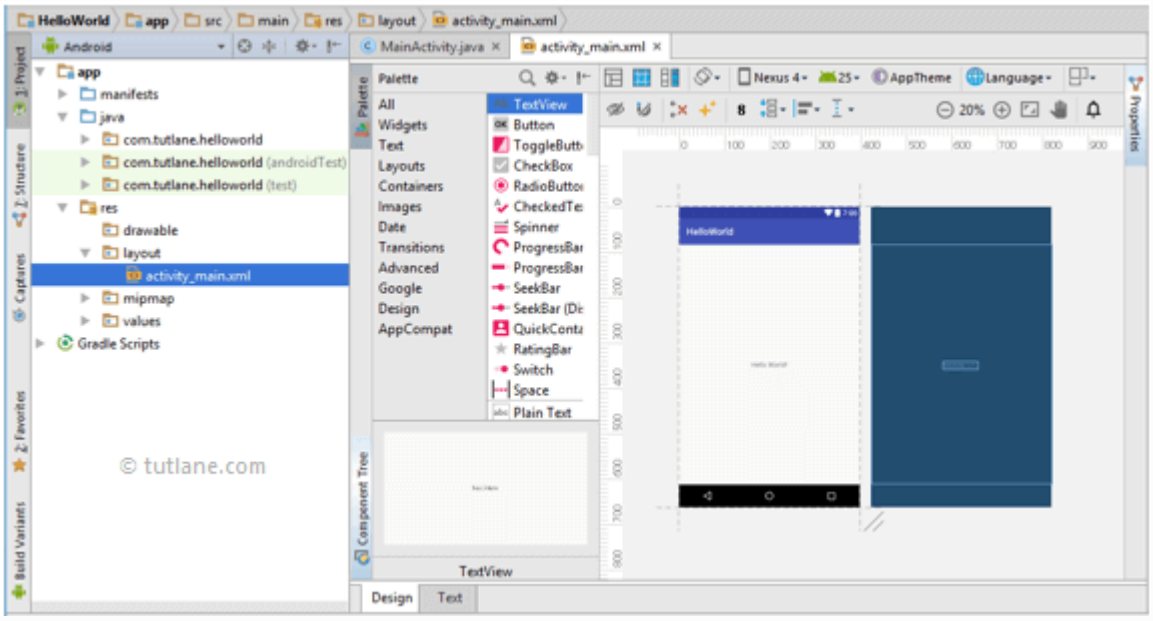


Alt Text

Following are the important files which we need to implement an app in android studio.

5. Android Layout File (activity_main.xml)

The UI of our application will be designed in this file and it will contain Design and Text modes. It will exists in the layouts folder and the structure of activity_main.xml file in Design mode like as shown below.



We can make required design modifications in activity_main.xml file either using Design or Text modes. If we switch to Text mode activity_main.xml file will contain a code like as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context="com.tutlane.helloworld.MainActivity">
  <TextView
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Hello World!"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:layout_constraintTop_toTopOf="parent" />
    </android.support.constraint.ConstraintLayout>
```

6. Android Main Activity File (MainActivity.java)

The main activity file in the android application is MainActivity.java and it will exist in the java folder. The MainActivity.java file will contain the java code to handle all the activities related to our app.

Following is the default code of MainActivity.java file which is generated by our HelloWorld application.

```
package com.tutlane.helloworld;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

7. Android Manifest File (AndroidManifest.xml)

Generally, our application will contain multiple activities and we need to define all those activities in the AndroidManifest.xml file. In our manifest file, we need to mention the main activity for our app using the MAIN action and LAUNCHER category attributes in intent filters (). In case if we didn't mention MAIN action or LAUNCHER category for the main activity, our app icon will not appear in the home screen's list of apps.

Following is the default code of AndroidManifest.xml file which is generated by our HelloWorld application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.tutlane.helloworld" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity android:name=".MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

These are the main folders and files required to implement an application in android studio. If you want to see the actual file structure of the project, select Project from the Project dropdown instead of Android.

❖ Terminologies Correlated to Android

XML file

The preeminent file is used for the structure of an android project. It has complete information about all the components and packages. It initializes the API that is further used by an application.

View

It is the component of the User Interface that occupies the rectangular area on the screen.

Layout

It properly aligned the views on the screen.

Activity

Activity is a User interface screen through which the user interacts. Users have a right to place the UI elements in any way according to the Users choice.

Emulator

The emulator is the virtual device smartphone provided with an android studio. You can run your created application on the emulator and test its UI and function according to the needs.

Intent

It acts as a communicating object. You can establish a communication between two or more than two components as services, broadcast receivers. It is used to start and end the activity and services components.

Services

It is used to run the process even in the background. There is no defined UI for service. Any component can start the service and end the services. You can easily switch between the applications even if the services are running the background.

Content Provider

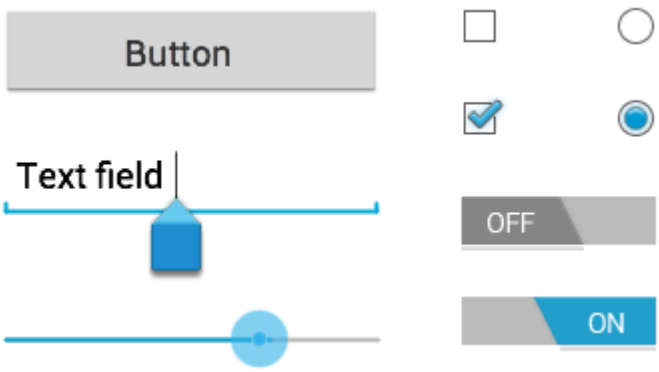
It implemented in two ways:

You can use implement the existing content provider in your application. However, you can also create a new content provider that will provide or share the data with other applications.

❖ Android UI Controls (Textview, EditText, Radio Button, Checkbox)

In android **UI** or **input** controls are the interactive or View components that are used to design the user interface of an application. In android we have a wide variety of UI or input controls available, those are TextView, EditText, Buttons, Checkbox, Progressbar, Spinners, etc.

Following is the pictorial representation of user interface (UI) or input controls in android application.



Generally, in android the user interface of an app is made with a collection of **View** and **ViewGroup** objects.

The **View** is a base class for all UI components in android and it is used to create interactive UI components such as TextView, EditText, Checkbox, Radio Button, etc. and it is responsible for event handling and drawing.

The **ViewGroup** is a subclass of **View** and it will act as a base class for layouts and layout parameters. The ViewGroup will provide invisible containers to hold other Views or ViewGroups and to define the layout properties.

To know more about View and ViewGroup in android applications, check this Android View and ViewGroup.

In android, we can define a UI or input controls in two ways, those are

- Declare UI elements in XML
- Create UI elements at runtime

The android framework will allow us to use either or both of these methods to define our application's UI.

Declare UI Elements in XML

In android, we can create layouts same as web pages in HTML by using default **Views** and **ViewGroups** in the XML file. The layout file must contain only one root element, which must be a **View** or **ViewGroup** object. Once we define the root element, then we can add additional layout objects or widgets as a child elements to build View hierarchy that defines our layout.

Following is the example of defining UI controls (TextView, EditText, Button) in the XML file (**activity_main.xml**) using LinearLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/fstTxt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter Name" />
    <EditText
        android:id="@+id/name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"/>
    <Button
        android:id="@+id/getName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Get Name" />
</LinearLayout>
```

In android, each input control is having a specific set of events and these events will be raised when the user performs particular action like, entering the text or touches the button.

Load XML Layout File from an Activity

Once we are done with the creation of layout with UI controls, we need to load the XML layout resource from our activity **onCreate()** callback method like as shown below.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

If you observe above code we are calling our layout using **setContentView** method in the form of **R.layout.layout_file_name**. Here our xml file name is **activity_main.xml** so we used file name **activity_main**.

Generally, during the launch of our activity, **onCreate()** callback method will be called by android framework to get the required layout for an activity.

Create UI Element at Runtime

If we want to create UI elements at runtime, we need to create our own custom **View** and **ViewGroup** objects programmatically with required layouts.

Following is the example of creating UI elements (TextView, EditText, Button) in LinearLayout using custom **View** and **ViewGroup** objects in an activity programmatically.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView textView1 = new TextView(this);
        textView1.setText("Name:");
        EditText editText1 = new EditText(this);
        editText1.setText("Enter Name");
        Button button1 = new Button(this);
        button1.setText("Add Name");
        LinearLayout linearLayout = new LinearLayout(this);
        linearLayout.addView(textView1);
        linearLayout.addView(editText1);
        linearLayout.addView(button1);
        setContentView(linearLayout);
    }
}
```

By creating a custom **View** and **ViewGroups** programmatically, we can define UI controls in layouts based on our requirements in android applications.

Width and Height

When we define a UI controls in a layout using an XML file, we need to set width and height for every **View** and **ViewGroup** elements using **layout_width** and **layout_height** attributes.

Following is the example of setting width and height for **View** and **ViewGroup** elements in the XML layout file.

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

```
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/fstTxt"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Enter Name" />
</LinearLayout>
```

If you observe above example, we used different values to set layout_width and layout_height, those are

- match_parent
- wrap_content

If we set value **match_parent**, then the **View** or **ViewGroup** will try to match with parent width or height.

If we set value **wrap_content**, then the **View** or **ViewGroup** will try to adjust its width or height based on the content.

Android Different Types of UI Controls

We have a different type of UI controls available in android to implement the user interface for our android applications.
Following are the commonly used UI or input controls in android applications.

- TextView
- EditText
- AutoCompleteTextView
- Button
- ImageButton
- ToggleButton
- CheckBox
- RadioButton

Android TextView

In android, **TextView** is a user interface control that is used to display the text to the user.

Android EditText

In android, **EditText** is a user interface control which is used to allow the user to enter or modify the text.

Android AutoCompleteTextView

In android, **AutoCompleteTextView** is an editable text view which is used to show the list of suggestions based on the user typing text. The list of suggestions will be shown as a dropdown menu from which the user can choose an item to replace the content of the textbox.

Android Button

In android, **Button** is a user interface control that is used to perform an action when the user clicks or tap on it.

Android Image Button

In android, **Image Button** is a user interface control that is used to display a button with an image to perform an action when the user clicks or tap on it.
Generally, the Image button in android looks similar as regular Button and perform the actions same as regular button but only difference is for image button we will add an image instead of text.

Android Toggle Button

In android, **Toggle Button** is a user interface control that is used to display ON (Checked) or OFF (Unchecked) states as a button with a light indicator.

Android CheckBox

In android, **Checkbox** is a two-states button that can be either checked or unchecked.

Android Radio Button

In android, **Radio Button** is a two-states button that can be either checked or unchecked and it cannot be unchecked once it is checked.

❖ Application Context in Android

The **Application Context** is a context tied to the **lifecycle of the application** and is used across the app's components to access application-level resources and services. It remains available throughout the app's lifetime and is not tied to the lifecycle of a specific activity or fragment.

I) Key Features of Application Context

1. **Global Scope**
   The Application Context is not linked to a specific activity or view; it persists as long as the application is running.
2. **Resource Access**
   It provides access to resources like:
   - **Shared Preferences:** Storing small key-value pairs persistently.
   - **File System:** Accessing app-specific files in internal or external storage.
   - **Databases:** Interacting with SQLite or Room databases.
3. **System Services**
   Application Context enables access to Android system-level services, such as:
   - **ConnectivityManager:** Checking or managing network connectivity.
   - **LocationManager:** Accessing location-related functionality.
   - **AlarmManager:** Setting alarms.
   - **NotificationManager:** Managing notifications.

II) When to Use Application Context

1. **Long-lived Objects:**
   Use Application Context when the context needs to outlive the activity or fragment, such as:
   - Background tasks.
   - Application-wide singleton instances.

2. **Avoiding Memory Leaks:**
   It helps prevent memory leaks since the Application Context is tied to the application's lifecycle rather than the activity's.
3. **Accessing Application Resources:**
   When working with components that need resources independent of a specific UI component, such as:
   - Custom adapters.
   - Database helpers.

III) Example:

Accessing Shared Preferences

Here's how you can use **SharedPreferences** with the Application Context:

```
SharedPreferences sharedPreferences =

    getApplicationContext().getSharedPreferences("MyPrefs", Context.MODE_PRIVATE);



SharedPreferences.Editor editor = sharedPreferences.edit();

editor.putString("username", "Ram");

editor.putInt("age", 30);

editor.apply(); // or editor.commit();
```

- **Mode:**
  - Context.MODE_PRIVATE: Only the app can access the file (default).
  - Other modes like MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE are deprecated.
- **apply() vs. commit():**
  - apply(): Asynchronous, does not return a result, faster for UI thread.
  - commit(): Synchronous, returns a boolean indicating success or failure.

2. Retrieving Data

```
SharedPreferences sharedPreferences =

    getApplicationContext().getSharedPreferences("MyPrefs", Context.MODE_PRIVATE);



String username = sharedPreferences.getString("username", "DefaultName");

int age = sharedPreferences.getInt("age", 0);



System.out.println("Username: " + username);

System.out.println("Age: " + age);
```

- The second parameter in getString() or getInt() is the **default value** returned if the key is not found.

❖ Activities in Android

Activities are part of the basic component of the Android application. Users cannot interact with an application on their mobile device without an activity. Activities form the bedrock upon which our mobile application is built. It consists of the layout, text fields, images, and different UI elements which the user can interact with.

Every mobile application you open, the first screen you see is an activity. In Android Studio activities are created using the Java or Kotlin programming language.

Activities are linked through intents, that are used for communication and navigation between different activities. Intents can be used to start a new activity, pass data between activities, or even trigger actions in other components of the app.

This article will provide a detailed explanation of activities, their importance, and activity life cycle along with its methods.

An activity

Activity is more like a container for one or more multiple screens in your app. It's not just seen as a screen but also as a unit that users interact with in your application.

Activities contain information about whether a user is currently interacting with the screen or if the activity is in the background.

It also serves as an entry point for your app. We can have multiple screens in an application that are bundled together into an activity.

For example, A user profile page activity can contain other screens like information details of the user.

However, Jetpack Compose has made everything easier. Using the jetpack compose UI we can have only one main activity with multiple screens while staying in the single main activity.

The main activity in Jetpack Compose serves as the entry point of our application. The main characteristic of activities in Android is the **LIFECYCLE** which means at some point our activity is created, destroyed, paused, etc.

Basic components of an activity

1. **Layout XML:** the layout XML file represents the user interface of an activity. These files contain view-groups (linear layout, constraint layout) and views such as buttons, images, app bars, etc.

   The XML file defines the structure and positioning of these components which the user will interact with like clicking a button to open a new activity.

   However, in Jetpack Compose you can define and structure the features of your app inside the main.activity.kt file.
2. **Activity class:** An activity class comprising a single screen with a user interface is represented by an instance(object) of the Activity class in Android.

   To create a custom activity you must create a subclass of the Activity class and override its lifecycle methods and other crucial methods to the application's needs.
3. **Life cycle methods:** these methods perform various behaviors in activity and can be overridden in different stages. The methods are:`onCreate()`, onStart(), onResume(), onPause(), onStop(), onDestroy(), etc.

❖ Services

A Service is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

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, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

❖ Intents in Android

Intents are messaging objects used to communicate between app components, such as Activities, Services, or Broadcast Receivers. They enable actions like launching activities, starting services, or delivering broadcasts.

Types of Intents

1. Explicit Intent

- **Definition**: Specifies the target component (e.g., an Activity or Service) by name.
- **Use Cases**:
  - Navigating between activities within the same app.
  - Starting a service.
- **Key Features**:
  - Target is explicitly defined using the component name or class.
  - Ideal for intra-app communication.
- **Example**: Navigating to another activity.

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

- **Example**: Starting a service.

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

2. Implicit Intent

- **Definition**: Describes a general action to be performed, allowing the Android system to determine which component (app or service) can handle it.
- **Use Cases**:
  - Sharing data.
  - Opening a web page or sending an email.
  - Choosing apps to handle user actions (e.g., opening a file).
- **Key Features**:
  - Does not explicitly name the target component.
  - Relies on **intent filters** in the manifest to match actions and categories.
- **Example**: Opening a web page.

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

- **Example**: Sharing data.

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.setType("text/plain");
intent.putExtra(Intent.EXTRA_TEXT, "Hello, world!");
startActivity(Intent.createChooser(intent, "Share via"));
```

Components of an Intent

1. **Action**: The operation to be performed (e.g., ACTION_VIEW, ACTION_SEND).
2. **Data**: The data to be acted upon, often specified as a URI.
3. **Category**: Provides additional information about the action (e.g., CATEGORY_DEFAULT).
4. **Extras**: Key-value pairs for passing additional data.
5. **Flags**: Instructions on how to launch the activity (e.g., FLAG_ACTIVITY_NEW_TASK).

Passing Data with Intents

You can use the putExtra() method to pass data and retrieve it in the target component using getExtras().

- **Sending Data**:

```
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("KEY_NAME", "Ram");
intent.putExtra("KEY_AGE", 25);
startActivity(intent);
```

- **Receiving Data**:

```
Intent intent = getIntent();
```

```
String name = intent.getStringExtra("KEY_NAME");
int age = intent.getIntExtra("KEY_AGE", 0);
```

Intent Filters

For an **Implicit Intent** to be handled, the target component must define an **intent filter** in its manifest file.

- **Example**:

```
<activity android:name=".SecondActivity">
 <intent-filter>
 <action android:name="android.intent.action.VIEW"/>
 <category android:name="android.intent.category.DEFAULT"/>
 <data android:scheme="https" android:host="www.example.com"/>
</intent-filter>
 </activity>
```

❖ Passing Data Between Activities using Extras

We can pass data between different activities when their started and when they end.
We start activities by creating `intents`. We can "put extra" information in an intent that starts a new activity.
Gmail has at least two Activities. Let's imagine there is a `ViewInbox` activity and a `ReadEmail` activity. When someone clicks on an email in the inbox the `ViewInbox` activity, the activity needs to create a new intent to start a `ReadEmail` activity. Simply starting the `ReadEmail` activity isn't enough.
The `ReadEmail` activity needs to know what email it's supposed to display.
The `ViewInbox` activity puts extra information in the intent that specifies which email should be displayed.
Here's some simple examples of how to put extra information in an intent, and how to get it out.

When creating new intents, you can also give it *extra* data. Here's an example:

```
    Intent intent = new Intent(EmailListActivity.this, ReadEmailActivity.class);
    intent.putExtra("ID", 123);
    intent.putExtra("SENDER", "RAM");
```

The Intent class has a handful of helper methods you can call to get and store extra data. The main one is `putExtra()`, which takes two parameters: a String that gives the data a name, and the data itself.
With `Intent.putExtra()`, you can put data inside the intent (including Strings, numbers, booleans, certain objects).
Once you start a new activity, you can retrieve the Intent and get the sent data, as follows:

```
    // get the intent that started this activity
    Intent intent = getIntent();

    // get the data from the intent
    int id = intent.getIntExtra("ID", 0);
    String sender = intent.getStringExtra("SENDER");
```

Again, the Intent class has a handful of getters for extra data, usually formatted like *get_Extra*. Examples, `getIntExtra()`, `getStringExtra()`, `getBooleanExtra()`, etc.

❖ *Broadcast Receivers*

Broadcast Receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometime called events or intents. 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.

There are following two important steps to make BroadcastReceiver works for the system broadcasted intents −

- Creating the Broadcast Receiver.
- Registering Broadcast Receiver

There is one additional steps in case you are going to implement your custom intents then you will have to create and broadcast those intents.

Creating the Broadcast Receiver

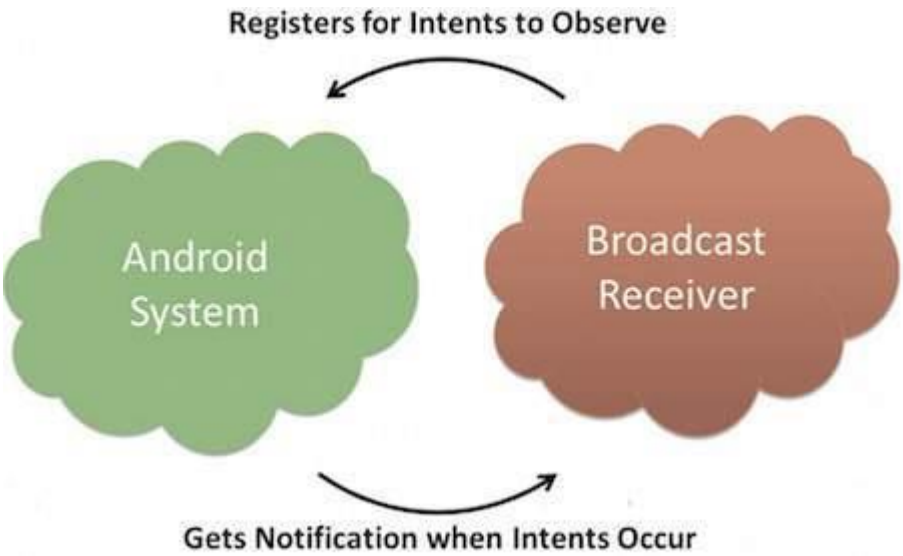
A broadcast receiver is implemented as a subclass of **BroadcastReceiver** class and overriding the onReceive() method where each message is received as a **Intent** object parameter.

```
public class MyReceiver extends BroadcastReceiver {
  @Override
```

```
  public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG).show();
  }
}
```

Registering Broadcast Receiver

An application listens for specific broadcast intents by registering a broadcast receiver in *AndroidManifest.xml* file. Consider we are going to register *MyReceiver* for system generated event ACTION_BOOT_COMPLETED which is fired by the system once the Android system has completed the boot process.



Broadcast-Receiver

```
<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <receiver android:name="MyReceiver">

    <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED">
      </action>
    </intent-filter>

  </receiver>
```

```
</application>
```

Now whenever your Android device gets booted, it will be intercepted by BroadcastReceiver *MyReceiver* and implemented logic inside *onReceive()* will be executed.

There are several system generated events defined as final static fields in the **Intent** class. The following table lists a few important system events.

Sr.No	Event Constant & Description
1	**android.intent.action.BATTERY_CHANGED** Sticky broadcast containing the charging state, level, and other information about the battery.
2	**android.intent.action.BATTERY_LOW** Indicates low battery condition on the device.
3	**android.intent.action.BATTERY_OKAY** Indicates the battery is now okay after being low.
4	**android.intent.action.BOOT_COMPLETED** This is broadcast once, after the system has finished booting.
5	**android.intent.action.BUG_REPORT** Show activity for reporting a bug.
6	**android.intent.action.CALL** Perform a call to someone specified by the data.
7	**android.intent.action.CALL_BUTTON** The user pressed the "call" button to go to the dialer or other appropriate UI for placing a call.
8	**android.intent.action.DATE_CHANGED** The date has changed.
9	**android.intent.action.REBOOT** Have the device reboot.

❖ The Android Manifest File

The **Android Manifest File** (AndroidManifest.xml) is an essential configuration file in every Android application. It provides information about the app's structure and requirements

to the Android operating system. Here's an overview of its purpose and common settings:

Purpose of the Android Manifest File

1. **Declare Application Components**:
   - Registers app components such as activities, services, broadcast receivers, and content providers.
   - Ensures the Android system is aware of the components and their configurations.
2. **Request Permissions**:
   - Specifies permissions the app needs to access system features (e.g., camera, location, internet).
3. **Define Hardware and Software Features**:
   - Declares the hardware or software features required for the app (e.g., camera, GPS).
4. **Set Application Metadata**:
   - Provides metadata like app name, version, icons, and themes.
5. **Filter Device Compatibility**:
   - Restricts app installation on incompatible devices based on hardware, features, or API levels.
6. **Control App Behavior**:
   - Specifies application-level configurations such as launch mode, process behavior, and backup options.

Example AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">

    <!-- Versioning Information -->
    <uses-sdk
        android:minSdkVersion="21"
        android:targetSdkVersion="33" />

    <!-- Permissions -->
    <uses-permission android:name="android.permission.INTERNET" />
```

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.CAMERA" />

<!-- Hardware Features -->
<uses-feature android:name="android.hardware.camera" android:required="true" />

<!-- Application Block -->
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">

    <!-- Main Activity -->
    <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <!-- Secondary Activity -->
    <activity
        android:name=".SecondActivity"
        android:label="@string/second_activity"
        android:exported="true">
        <intent-filter>
            <action android:name="com.example.myapp.ACTION_VIEW" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
    </activity>

    <!-- Service -->
    <service
        android:name=".MyBackgroundService"
```

```
        android:enabled="true"
        android:exported="false" />

    <!-- Broadcast Receiver -->
    <receiver
        android:name=".MyBroadcastReceiver"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>

    <!-- Content Provider -->
    <provider
        android:name=".MyContentProvider"
        android:authorities="com.example.myapp.provider"
        android:exported="false" />

    <!-- Metadata -->
    <meta-data
        android:name="com.google.android.maps.v2.API_KEY"
        android:value="your_api_key" />

  </application>
</manifest>
```

Explanation of the Program

1. **<manifest>**:
   - The root tag includes the app's package name and XML namespace declaration.
2. **Versioning**:
   - <uses-sdk> defines the minimum and target SDK levels.
3. **Permissions**:
   - <uses-permission> declares permissions like internet access, location, and camera.
4. **Hardware Features**:

- <uses-feature> ensures that the app requires a camera and will not run on devices without one.

5. **Application Settings**:
   - <application> defines global app settings, including backup, theme, and the app's icon.
6. **Activity**:
   - <activity> registers MainActivity as the launcher activity with an intent filter for MAIN and LAUNCHER.
   - SecondActivity demonstrates handling a custom action.
7. **Service**:
   - <service> registers a background service (MyBackgroundService).
8. **Broadcast Receiver**:
   - <receiver> registers MyBroadcastReceiver to handle system events like BOOT_COMPLETED.
9. **Content Provider**:
   - <provider> declares a custom content provider (MyContentProvider) with a unique authority.
10. **Metadata**:
    - <meta-data> is used to store custom data like API keys for Google Maps.

❖ Android Emulator

The Android Emulator simulates Android devices on your computer so that you can test your application on a variety of devices and Android API levels without needing to have each physical device. The emulator offers these advantages:

- **Flexibility**: In addition to being able to simulate a variety of devices and Android API levels, the emulator comes with predefined configurations for various Android phone, tablet, Wear OS, Android Automotive OS, and Android TV devices.
- **High fidelity**: The emulator provides almost all the capabilities of a real Android device. You can simulate incoming phone calls and text messages, specify the location of the device, simulate different network speeds, simulate rotation and other hardware sensors, access the Google Play Store, and much more.
- **Speed**: Testing your app on the emulator is in some ways faster and easier than doing so on a physical device. For example, you can transfer data faster to the emulator than to a device connected over USB.

In most cases, the emulator is the best option for your testing needs. This page covers the core emulator functionalities and how to get started with it.

Alternatively, you can deploy your app to a physical device. For more information, see Run apps on a hardware device.

Get started with the emulator

The Android Emulator lets you test your app on many different devices virtually. The emulator comes with Android Studio, so you don't need to install it separately. To use the emulator, follow these basic steps, which are described in more detail in the sections that follow:

1. Verify that you have the system requirements.
2. Create an Android Virtual Device (AVD).
3. Run your app on the emulator.
4. Navigate the emulator.

This page covers the steps to set up and explore your virtual testing environment in more detail. If you already have your app running on the emulator and are ready to use more advanced features, see Advanced emulator usage.

If you're experiencing issues with the emulator, see Troubleshoot known issues with Android Emulator. Depending on your needs and resources, it might be worth delving into system requirements and technical configurations, or it might be better to use a physical device.

Emulator system requirements

For the best experience, you should use the emulator in Android Studio on a computer with at least the following specs:

- 16 GB RAM
- 64-bit Windows 10 or higher, MacOS 12 or higher, Linux, or ChromeOS operating system
- 16 GB disk space

❖ Create an Android Virtual Device

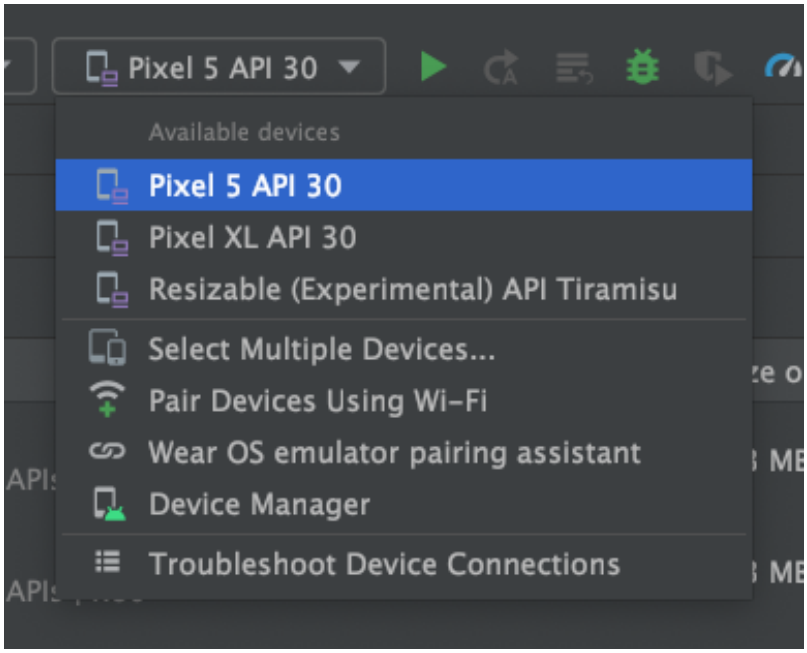
Each instance of the Android Emulator uses an *Android virtual device (AVD)* to specify the Android version and hardware characteristics of the simulated device. To create an AVD, see Create and manage virtual devices.

Each AVD functions as an independent device with its own private storage for user data, SD card, and so on. By default, the emulator stores the user data, SD card data, and cache in a directory specific to that AVD. When you launch the emulator, it loads the user data and SD card data from the AVD directory.

Run your app on the emulator

After you have created an AVD, you can start the Android Emulator and run an app in your project:

1. In the toolbar, select the AVD that you want to run your app on from the target device menu.



The target device menu.

2. Click **Run**. The emulator might take a minute or so to launch for the first time, but subsequent launches use a snapshot and should launch faster. If you experience issues, see the troubleshooting guide.

Once your app is installed on your AVD, you can run it from the device as you would run any app on a device. Any time you want to deploy new changes, you need to click **Run** or **Apply Changes** again.

Wear OS pairing assistant

If you want to test your app with Wear OS devices, the Wear OS pairing assistant guides you step-by-step through pairing Wear OS emulators with physical or virtual phones directly in Android Studio. To learn more, see Use the Wear OS emulator pairing assistant.

Navigate the emulator

While the emulator is running, you can use your computer mouse pointer to mimic your finger on the touch screen and use the emulator panel to perform common actions.

Navigate the emulator screen

Use your computer mouse pointer to mimic your finger on the touchscreen, select menu items and input fields, and click buttons and controls. Use your computer keyboard to type characters and enter emulator shortcuts.