KEMENTERIAN PENGAJIAN TINGGI

POLITEKNIK MALAYSIA
SULTAN MIZAN ZAINAL ABIDIN

THE NEXT BIG APP IDEA

MOBILE APPLICATION DEVELOPMENT

FOR MALAYSIAN POLYTECHNIC STUDENTS

by
Melati Sabtu
Syaiful Bachtiar Shahinan
Department of Information Technology & Communication
Polytechnic of Sultan Mizan Zainal Abidin
Dungun, Terengganu

This page intentionally left blank.

THE NEXT BIG APP IDEA

MOBILE APPLICATION DEVELOPMENT

FOR MALAYSIAN POLYTECHNIC STUDENTS

by
Melati Sabtu
Syaiful Bachtiar Shahinan
Department of Information Technology & Communication
Polytechnic of Sultan Mizan Zainal Abidin
Dungun, Terengganu

Melati Sabtu | Syaiful Bachtiar Shahinan
Department of Information Technology & Communication
Polytechnic of Sultan Mizan Zainal Abidin
Dungun, Terengganu

Published by:
Polytechnic of Sultan Mizan Zainal Abidin
Km 08, Jalan Paka, 23000 Dungun, Terengganu

eISBN 978-967-2099-69-7

Perpustakaan Negara Malaysia Cataloguing-in-Publication Data

Melati Sabtu, 1982-
The Next Big App Idea : Mobile Application Development For Malaysian Polytechnic Students / by Melati Sabtu, Syaiful Bachtiar Shahinan.
Mode of access: Internet
eISBN 978-967-2099-69-7
1. Android (Electronic resource).
2. Application software--Development.
3. Government publications--Malaysia.
4. Electronic books.
I. Syaiful Bachtiar Shahinan, 1980-. II. Title.
005.25

ACKNOWLEDGEMENT

Design Thinking: Practical Applications

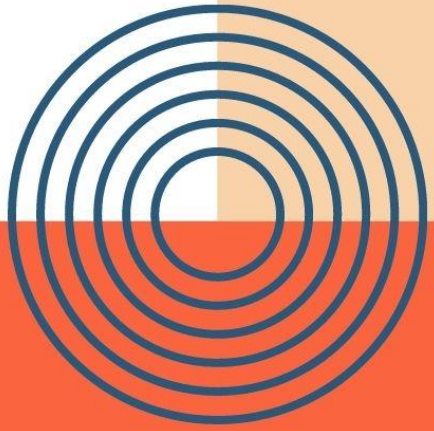
Developing applications can be a core skill in the field of information technology. Nowadays, businesses are increasingly looking for mobile apps to enhance their relationships with customers and improve their internal processes. They need individuals skilled in developing mobile applications that support this initiative.

Therefore, this book is intended to be an introduction to mobile application development. Users will have the basic skills to develop Android applications from app creation through app publishing. Step by step is shown so that users can more easily understand and can do it themselves.

This book is the full property of Polytechnic of Sultan Mizan Zainal Abidin which is used on the online learning platform PSMZA.

Any questions can contact us at the email address:

- melatisabtu@gmail.com
- naimrichdesign@gmail.com



CONTENTS

01

GETTING STARTED WITH ANDROID

1.1

Explain what is Android ?

1.2

Set-up the development environment for Android

1.3

Develop first Android Application

1.1 Explain what is Android ?

Android is an open source operating system based on the Linux kernel, middleware and key applications for mobile devices. It is designed primarily for touch screen mobile devices such as smartphones and tablet computers. **Android** was developed by the Open Handset Alliance, led by Google.

Android offers an integrated approach to app development for mobile devices which means developers only need to develop for **Android**, and the app should be able to run on different devices powered by **Android**. The **Android SDK** provides the tools and APIs necessary to begin developing applications on the **Android** platform using the Java programming language.

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 additional update, with the primary goal of improving the user interface, both in terms of functionality and performance.

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

History of Android

Android, Inc. was founded in Palo Alto, California, in October 2003 by **Andy Rubin** (founder of Danger), **Rich Miner** (founder of Wildfire Communications, Inc.), **Nick Sears** (former VP of T-Mobile), and **Chris White** (Head of Design and Development between WebTV interface) to develop "***smart mobile devices that are more aware of their location and preferences***".

The initial goal of Android development was to develop an operating system aimed at sophisticated digital cameras, but then they realized that the market for the device was not large enough, and then turned to the development of the Android smartphone market to compete with Symbian and Windows Mobile.

Figure 1-1: *Founders and Developers of Android Operating System*

Android Versions

The development of the Android operating system was started in 2003 by Android, Inc. Later, it was purchased by Google in 2005. The version history of the Android operating system began with the launch of Android 1.0 beta in November 2007.

Since April 2009, each version of Android has been developed with a code name based on a dessert item. The first Android version which was released under the numerical order format was Android 10. API level is an integer value that uniquely identifies the API revision framework offered by the Android platform version.

Android Versions, Name and API Level

Code name	Version numbers	API level	Release date
No codename	1.0	1	September 23, 2008
No codename	1.1	2	February 9, 2009
Cupcake	1.5	3	April 27, 2009
Donut	1.6	4	September 15, 2009
Eclair	2.0 - 2.1	5 - 7	October 26, 2009
Froyo	2.2 - 2.2.3	8	May 20, 2010
Gingerbread	2.3 - 2.3.7	9 - 10	December 6, 2010
Honeycomb	3.0 - 3.2.6	11 - 13	February 22, 2011
Ice Cream Sandwich	4.0 - 4.0.4	14 - 15	October 18, 2011
Jelly Bean	4.1 - 4.3.1	16 - 18	July 9, 2012
KitKat	4.4 - 4.4.4	19 - 20	October 31, 2013
Lollipop	5.0 - 5.1.1	21- 22	November 12, 2014
Marshmallow	6.0 - 6.0.1	23	October 5, 2015
Nougat	7.0	24	August 22, 2016
Nougat	7.1.0 - 7.1.2	25	October 4, 2016
Oreo	8.0	26	August 21, 2017
Oreo	8.1	27	December 5, 2017
Pie	9.0	28	August 6, 2018
Android 10	10.0	29	September 3, 2019
Android 11	11	30	September 8, 2020

Table 1-1: Android version, code name and API level provided by Google

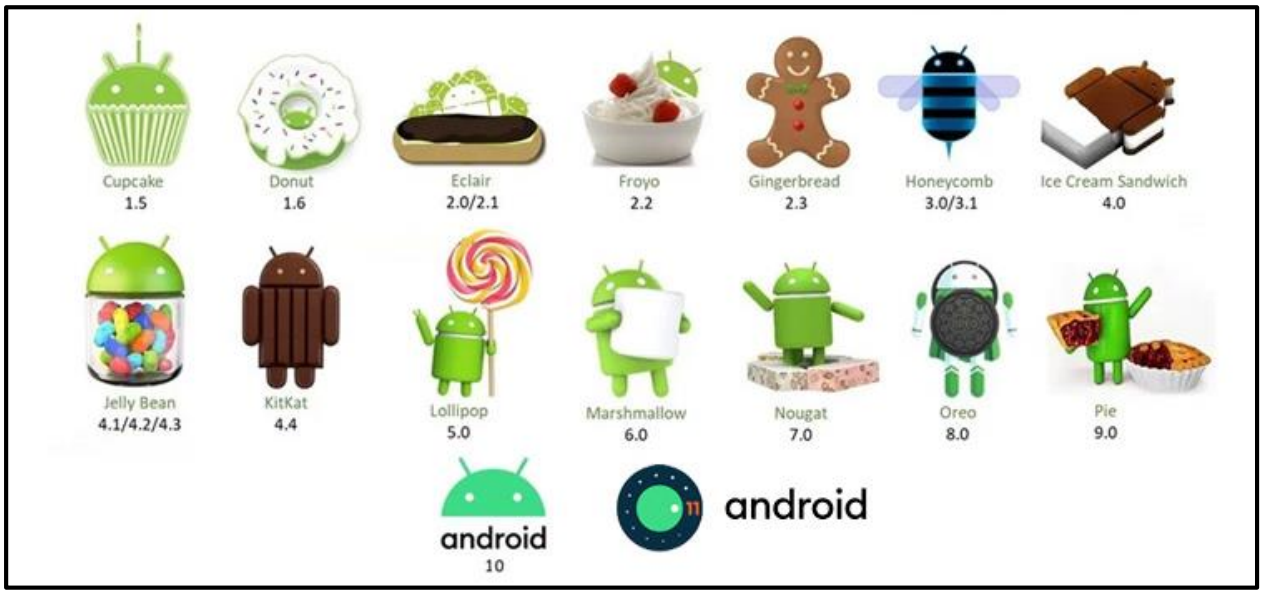


Figure 1-3: Android Version Lists

Figure 1-4: The first commercially available smartphone running Android was the HTC Dream, also known as T-Mobile G1, announced on September 23, 2008

Features of Android

Android Features & Descriptions

Beautiful UI • Android OS basic screen provides a beautiful and intuitive user interface.
Connectivity • GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, NFC and WiMAX.
Storage • SQLite, a lightweight relational database, is used for data storage purposes.
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.
Messaging • SMS and MMS
Web Browser • Based on the open-source WebKit layout engine, coupled with Chrome's V8 JavaScript engine supporting HTML5 and CSS3.
Multi-Touch • Android has native support for multi-touch which was initially made available in handsets such as the HTC Hero.
Multi-Tasking • User can jump from one task to another and same time various application can run simultaneously.
Resizable Widgets • Widgets are resizable, so users can expand them to show more content or shrink them to save space.
Multi-Language • Supports single direction and bi-directional text.
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.
Wi-Fi Direct • A technology that lets apps discover and pair directly, over a high-bandwidth peer-to-peer connection.
Android Beam • A popular NFC-based technology that lets users instantly share, just by touching two NFC-enabled phones together.

Table 1-2: Features of Android

Architecture of Android

Android is an open source, Linux-based software stack made for a wide variety of devices and form factors. The following diagram shows the main components of the Android operating system platform. Each section is described in more detail below.

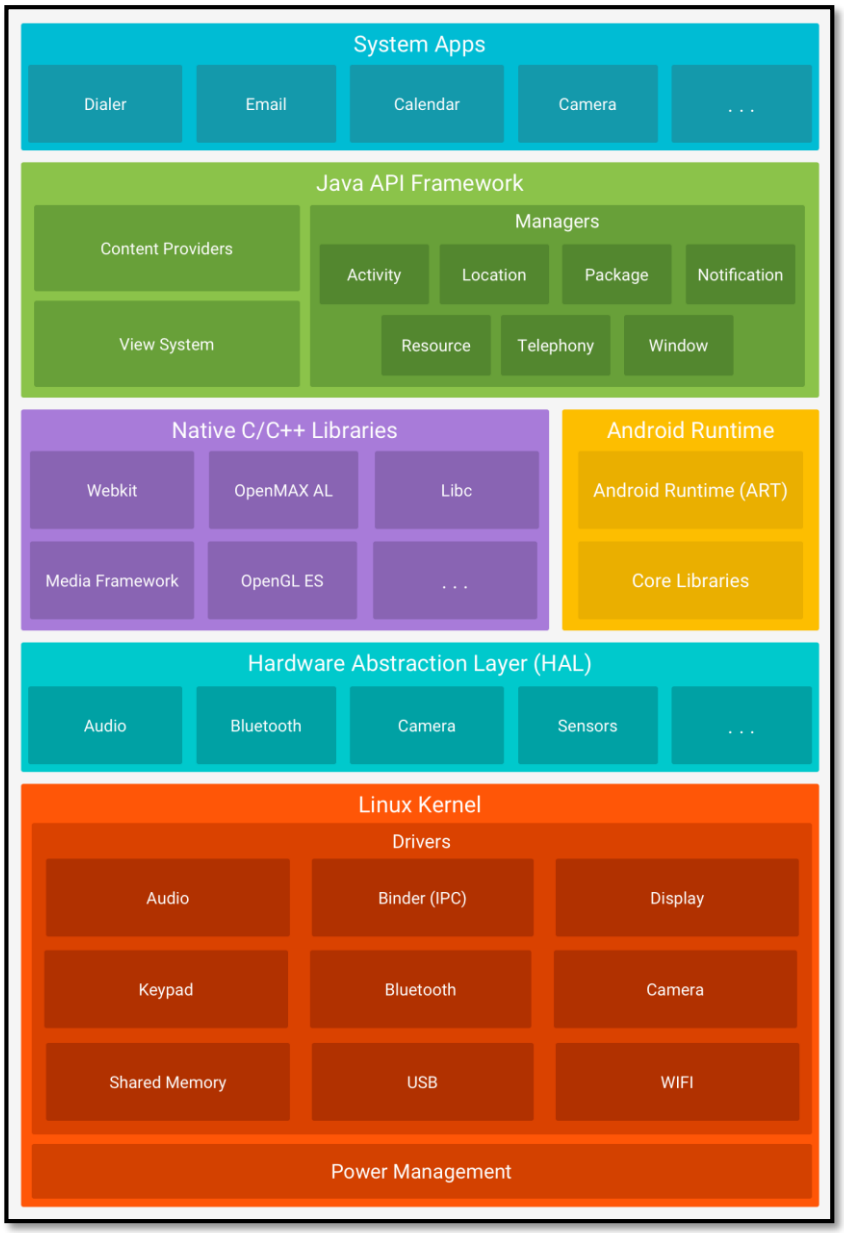


Figure 1-5: The Android Software Stacks

Descriptions of Android Software Stack

Linux Kernel

- The basis of the Android platform for communication layer for underlying hardware.
- Using the Linux kernel allows Android to take advantage of key security features and allows device manufacturers to develop hardware drivers for well-known kernels.

Hardware Abstraction Layer (HAL)

- The hardware abstraction (HAL) layer provides a standard interface that showcases the hardware capabilities of a device to the higher-end Java API framework.
- HAL consists of several library modules, each implementing an interface for a specific type of hardware component, such as a camera or Bluetooth module.
- When the skeleton API makes a call to access the device hardware, the Android system loads the library module for that hardware component.

Android Runtime (ART)

- Android Runtime (ART) is the application process time environment used by the Android operating system. ART performs a code translation of application bytes into native instructions that are then executed by the device's run time environment.
- Some key features of ART include ahead-of-time (AOT) and just-in-time (JIT) compilation, optimized garbage collection (GC) and enhanced debugging support, including custom sampling profilers, detailed diagnostic exceptions and fault reports, and the ability to set monitoring points to monitor specific areas.

Native C/C++ Libraries

- Many core Android system components and services, such as ART and HAL, are built from native code that requires native libraries written in C and C ++.
- The Android platform provides a Java framework API to demonstrate the functionality of some of these native libraries to applications.

- If developers want to develop applications that require C or C ++ code, Android NDK can be used to access some of these native platform libraries directly from native code.

Java API Framework

- The entire suite of Android OS features is available through APIs written in Java which allows high-level interactions with the Android system.
- These APIs form the building blocks needed to create Android applications by facilitating the reuse of modular system components and services, which include the following:
  i. A rich and extensive Display System that can be used to create application UI, including lists, grids, text boxes, buttons, and even embeddable web browsers
  ii. Resource Manager, provides access to non-code resources such as strings, graphics, and localized layout files
  iii. Notification Manager that allows all applications to display special alerts in the status bar
  iv. Activity Manager that manages the application life cycle and provides a common navigation layout
  v. Content Providers that allow applications to access data from other applications, such as the Contacts application, or to share their own data
  vi. Developers have full access to the same framework APIs used by Android system applications.

System Applications

- Android comes with a bunch of core apps like the Browser, Camera, Gallery, Music, Phone, Email, SMS messaging, Calendar and more.
- System applications serve both as applications for users and to provide key capabilities that developers can access from their own applications.
- For example, if the user's application wants to deliver an SMS message, the user does not need to build its own function, instead the user can ask which SMS application is already installed to deliver the message to the specified recipient user.

Table 1-3: Android Software Stacks

Android Devices in the Market

Mobile technology is gaining huge attention in the business and IT worlds. This technology represents a dramatic change in the capacity of the technology that enables potential economic advantages for those who can take advantage of it. Mobile technology is the foundation of innovation in reaching customers, and in redesigning business processes and software products that led to the creation of many small businesses.

Android devices (phones and tablets) have a unique set of hardware and software capabilities that make the way users interact with devices different for each. To fully capture the capabilities of the device and not degrade the user experience, developers must design such unique features.

Android devices originally used four hardware buttons to support the use of the user's device. These buttons are the **Home Button**, the **Menu Button**, the **Search Button**, and the **Back Button**. Users can press one of these buttons at any time during the use of the application, which will affect the functionality of the application. The **Home** and **Back** buttons work without relying on code, while the **Menu** and **Search** buttons only provide functionality if the application is coded specifically to use these buttons.

Figure 1-6: Android hardware buttons

1.2 Set-up the development environment for Android

Before any work can begin on Android application development, the first step is to configure the computer system to function as a development platform. This involves a number of steps consisting of installing the **Java Development Kit (JDK)** and **Android Studio Integrated Development Environment (IDE)** which also includes the **Android Software Development Kit (SDK)**.

Android application development can be done on one of the following platforms. See below to see the minimum specs you need to play with your Android device!

Windows	• 64-bit Microsoft® Windows® 8/10 • x86_64 CPU architecture; 2nd generation Intel Core or newer, or AMD CPU with support for a Windows Hypervisor • 8 GB RAM or more • 8 GB of available disk space minimum (IDE + Android SDK + Android Emulator) • 1280 x 800 minimum screen resolution
Mac	• MacOS® 10.14 (Mojave) or higher • ARM-based chips, or 2nd generation Intel Core or newer with support for Hypervisor.Framework • 8 GB RAM or more • 8 GB of available disk space minimum (IDE + Android SDK + Android Emulator) • 1280 x 800 minimum screen resolution

Linux	• Any 64-bit Linux distribution that supports Gnome, KDE, or Unity DE; GNU C Library (glibc) 2.31 or later • x86_64 CPU architecture; 2nd generation Intel Core or newer, or AMD processor with support for AMD Virtualization (AMD-V) and SSSE3 • 8 GB RAM or more • 8 GB of available disk space minimum (IDE + Android SDK + Android Emulator) • 1280 x 800 minimum screen resolution
Chrome OS	• 8 GB RAM or more recommended • 4 GB of available disk space minimum • Intel i5 or higher (U series or higher) recommended • 1280 x 800 minimum screen resolution

Table 1-4: Android Studio specifications' platforms

Installing the Java Development Kit (JDK) and Android Studio Package

Android Studio is available for computers running **Windows** or **Linux**, and for **Macs** running **macOS**. **OpenJDK** (Java Development Kit) is integrated with Android Studio.

The installation is similar for all platforms.

1. Navigate to the Android Studio download page located at the following URL https://developer.android.com/studio and follow the instructions to download and install Android Studio.

2. Accept the default configuration for all steps, and make sure all components are selected for installation.

3. Once the installation is complete, the setup wizard downloads and installs additional components, including the Android SDK. Be patient, as this process may take some time, depending on your internet speed.

4. When the installation is complete, Android Studio starts, and you're ready to create your first project.

TUTORIAL: Set Up the Development Environment for Android

Learning Outcomes:
By the end of this tutorial, you should be able to set up the development environment for Android.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Download and Set-up Java Development Kit (JDK)

1. Download the latest version of Java JDK from Oracle's Java site
   https://www.oracle.com/java/technologies/javase-downloads.html

2. Follow the given instructions to install and configure the setup.

3. Finally set **PATH** and **JAVA_HOME** environment variables to refer to the directory that contains **java** and **javac.**

   Alternatively, if using on Windows, right-click on My Computer, select Properties, then Advanced, then Environment Variables. Then, update the PATH value and press the OK button.

B. Download and Set-up Android Studio

4. Download the latest version android studio from Android Studio site
   https://developer.android.com/studio

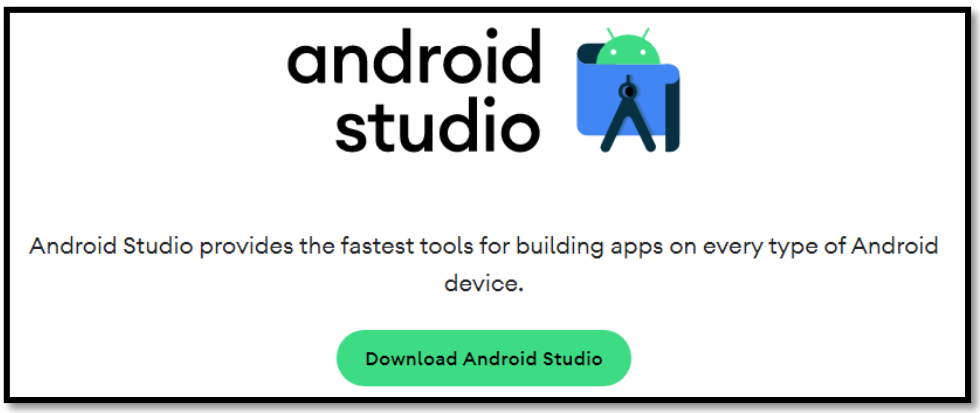


Figure 1-7: Download Android Studio from the Android Studio website

5. Run on windows machine according to android studio wizard guideline.

6. Launch **Android Studio.exe**.

Figure 1-8: Android Studio Setup launcher

7. Initiate JDK path or later version in android studio installer.

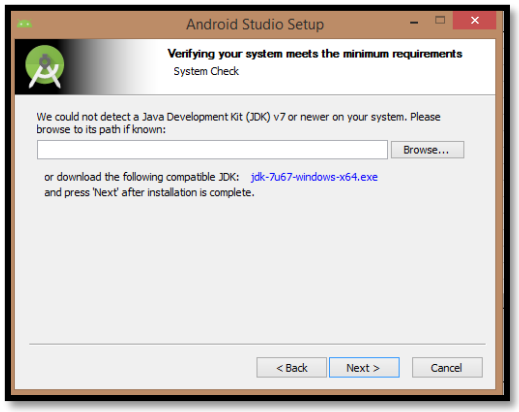


Figure 1-9: Android Studio Setup - JDK initiation

8. Check the components, which are required to create applications.

Figure 1-10: Android Studio Setup – Choose Components

9. Specify the location of local machine path for Android Studio and Android SDK.

10. Specify the ram space for Android emulator by default it would take 512MB of local machine RAM.

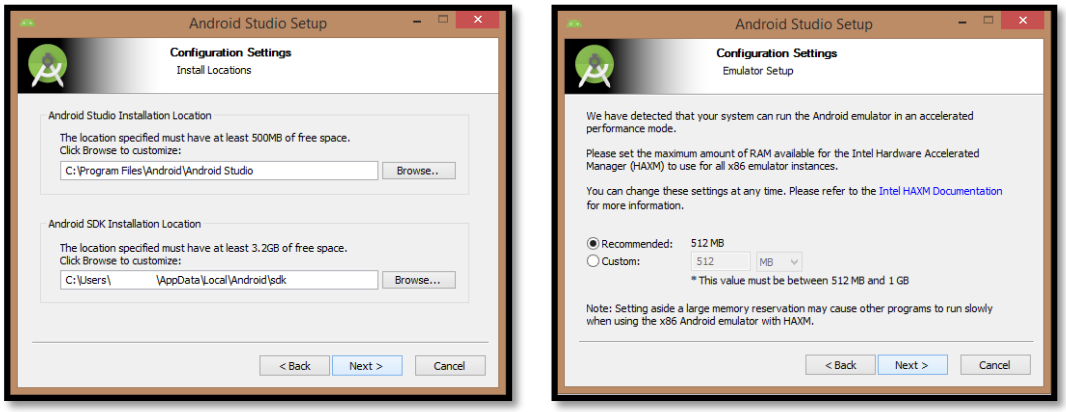


Figure 1-11: Android Studio Setup – Configuration Settings

11. At final stage, it would extract SDK packages into our local machine, it would take a while time to finish the task and would take 2626MB of Hard disk space.

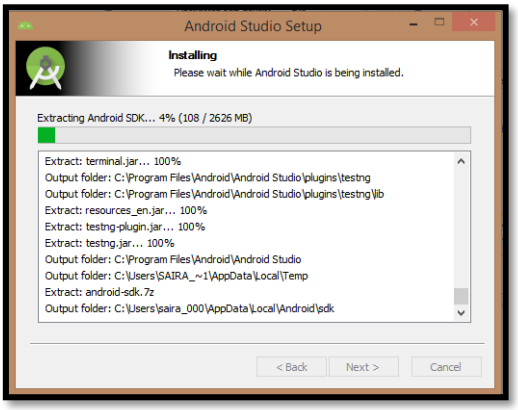


Figure 1-12: Android Studio Setup – Installing

12. After done all above steps perfectly, get finish button and it going to be open Android Studio Project with **Welcome to Android Studio** message.

1.3 Develop first Android Application

Before moving on to a slightly more advanced topic, now is a good time to verify that all the required development packages are installed and working properly. The best way to achieve this is to create an Android app, compile and run it. You will write Android applications in the Java programming language using an IDE called Android Studio. Android Studio is an IDE designed specifically for Android development.

This lesson describes how to create a simple Android application project using Android Studio. You will learn how to create a new Android project and create "Hello, World!" project with Android Studio.

If you don't have a project opened, Android Studio shows the Welcome screen, where you can create a new project by clicking **Start a new Android Studio project**. If you do have a project opened, you start creating a new project by selecting **File > New > New Project** from the main menu. You should then see the **Create New Project** wizard, which lets you choose the type of project you want to create and populates with code and resources to get you started. Once the project is created, you will explore the use of the Android emulator environment to run application tests.

TUTORIAL: Develop First Android Application

Learning Outcomes:
By the end of this tutorial, you should be able to create a "Hello, World!" project with Android Studio and run it.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Create a new project

1. Open Android Studio

2. In the **Welcome to Android Studio** dialog, click **+ Create New Project** link to get started.

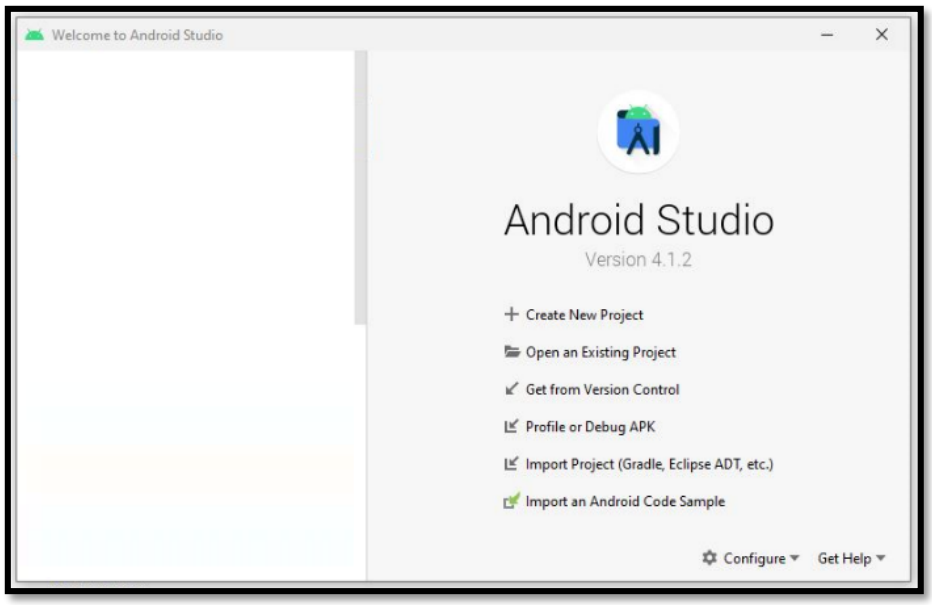


Figure 1-13: Android Studio welcome screen

3. In the **Select a Project Template** Window, select **Empty Activity**. Click **Next**.

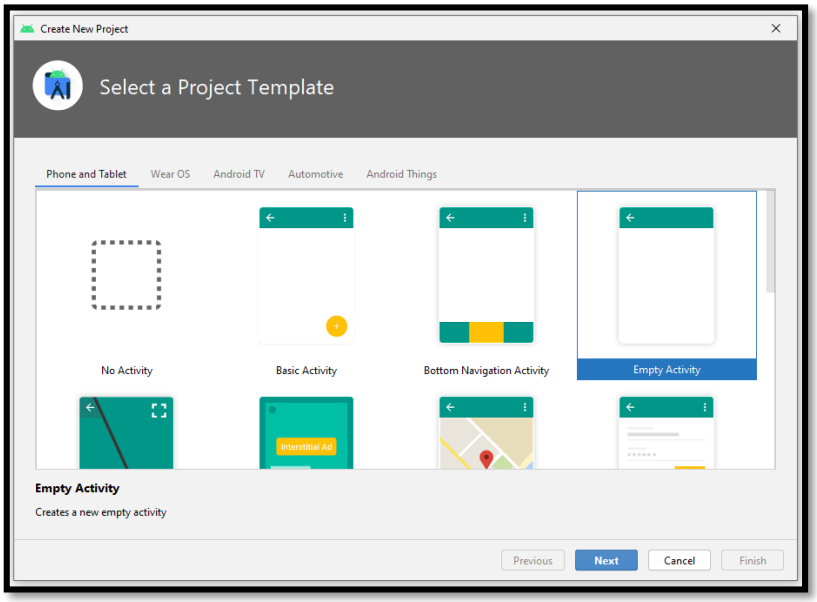


Figure 1-14: Select a Project Template Window

4. In the **Configure Your Project** window, use the following data of input for your project:
   - Give application a name such as **My First Application**.
   - Select **Java** from the **Language** drop-down menu**.**
   - Leave the defaults for the other fields.

5. Click **Finish.**

6. After some processing time, the **Android Studio** main window appears.

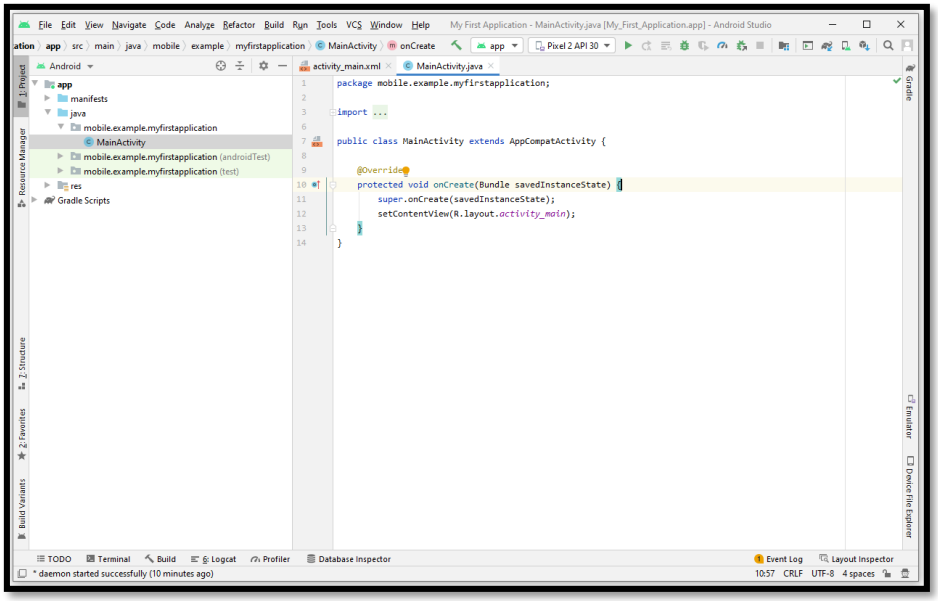


Figure 1-15: Android Studio main window

B. Explore the Project Structure and files

7. First, be sure the **Project** window is open (select **View > Tool Windows > Project**) and the Android view is selected from the drop-down list at the top of that window.

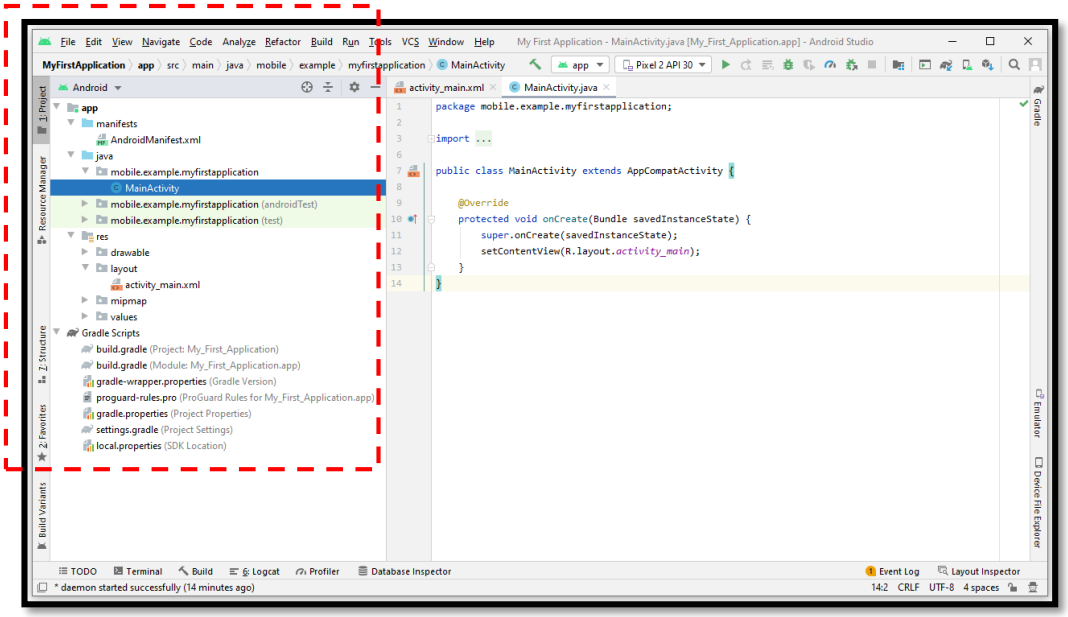


Figure 1-16: Review the generated Project Structure and files

8. In the **Project > Android** view you see four top-level folders below your **app** folder: **manifests, java, res** and **Gradle Scripts**.

9. Expand the **manifests** folder.

app > manifests > AndroidManifest.xml

The manifest file describes all the components of your Android app and is read by the Android runtime system when your app is executed.

10. Expand the **java** folder. All your Java language files are organized here.

app > java > com.example.myfirstapp > MainActivity

This is the main activity that contains the Java source code files for your Android app. It's the entry point for your app. When you build and run your app, the system launches an instance of this Activity and loads its layout.

11. Expand the **res** folder. This folder contains all the resources for your Android app, including images, layout files, strings, icons, and styling.

    app > res > layout > activity_main.xml

    This XML file defines the layout for the activity's user interface (UI). It contains a **TextView** element with the text "**Hello, World!**"

12. Expand the **Gradle Scripts** folder.

    Gradle Scripts > build.gradle

    There are two files with this name: one for the project "**Project: My_First_Application**" and one for the app module "**Module: My_First_Application.app**". Each module has its own **build.gradle** file. Each module's **build.gradle** file use to control how the Gradle plugin builds app.

C. Create Android Virtual Device (AVD)

13. To create a new AVD, open the AVD Manager via the **Tools > AVD Manager** menu tab.

14. Define AVD by clicking the **Create Virtual Device** button, at the bottom of the **AVD Manager** dialog (Figure 1-17).

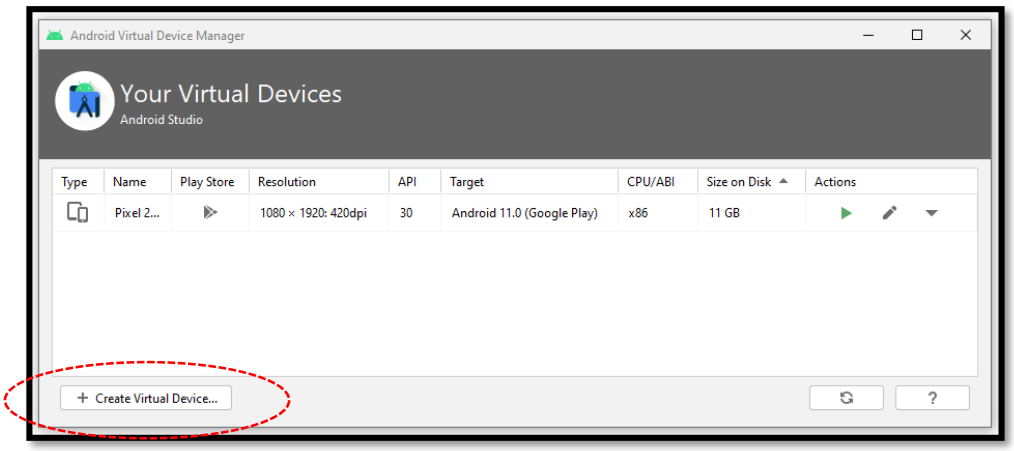


Figure 1-17: Android Virtual Device Manager

15. The **Select Hardware** dialog appears (Figure 1-18). Select values and then click **Next**.

16. Notice that only some hardware profiles are indicated to include **Play Store**. This indicates that these profiles are fully CTS compliant (Compatibility Test Suite) and may use system images that include the Play Store app.

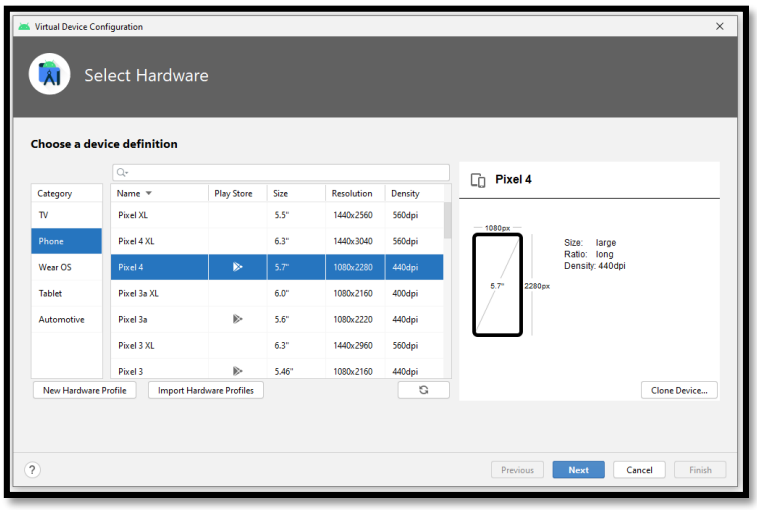


Figure 1-18: Virtual Device Configuration – Device Definition

17. The **System Image** dialog appears (Figure 1-19). Select the latest API level for AVD, and then click **Next**.

- The **Recommended** tab lists recommended system images. The other tabs include a more complete list.
- If you see **Download** next to the system image, you need to click it to download the system image. You must be connected to the internet to download it.

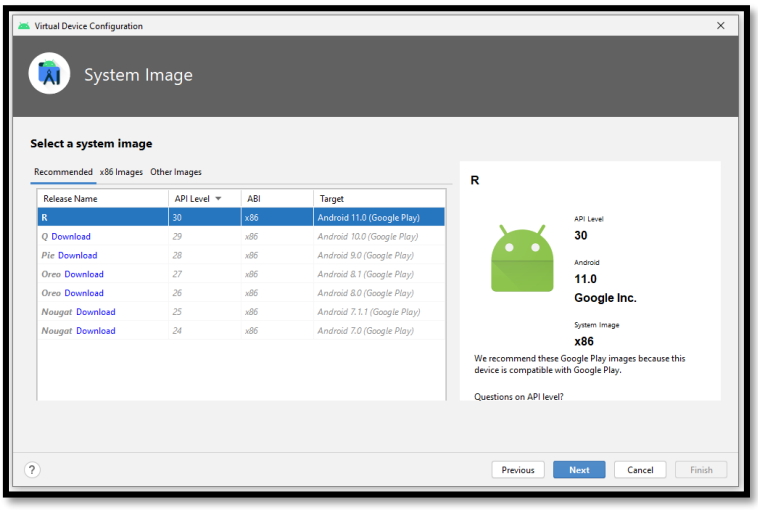


Figure 1-19: Virtual Device Configuration – System Image

18. The **Verify Configuration** dialog appears (Figure 1-20). Afterwards, click **Finish** button. This will create the AVD configuration and display it under the list of available virtual devices.

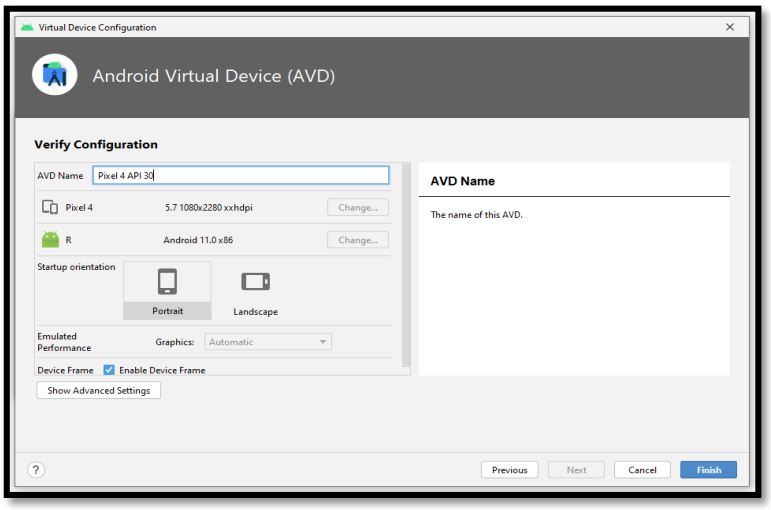


Figure 1-20: Virtual Device Configuration – Verify Configuration

19. The AVD appears in the **Your Virtual Devices** dialog (Figure 1-21). You can now ready to deploy and run your application on this virtual device.

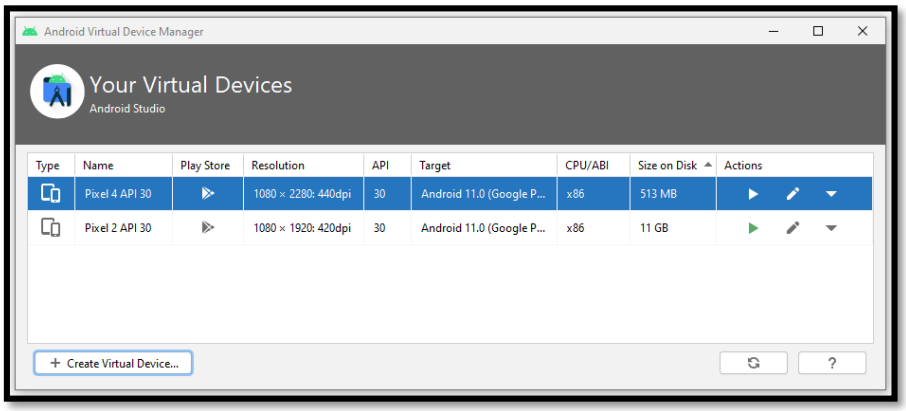


Figure 1-21: Android Virtual Device Manager - Your Virtual Devices page

D. Run your app on an emulator

20. In the toolbar, select your app from the run/debug configurations drop-down menu.

21. From the target device drop-down menu, select the AVD emulator that you want to run your app on.

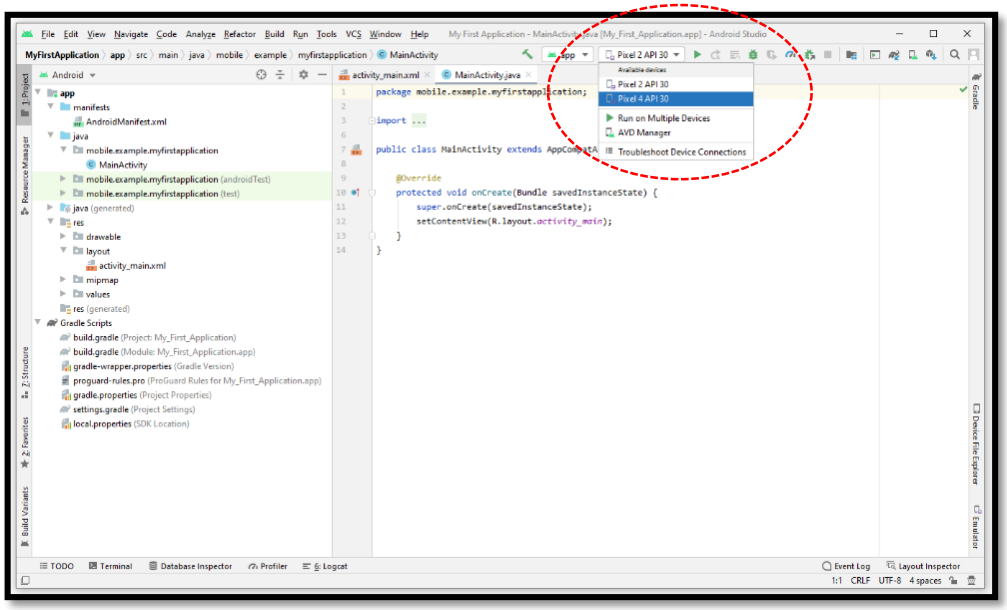


Figure 1-22: AVD emulator selection

22. Click **Run** .

23. The AVD emulator starts and boots just like a physical device. Depending on the speed of your computer, this may take a while. You can look in the small horizontal status bar at the very bottom of Android Studio for messages to see the progress.

Messages that might appear briefly in the status bar

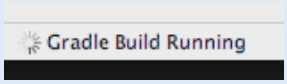
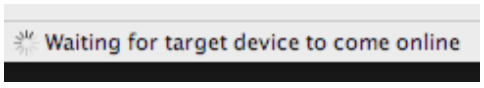
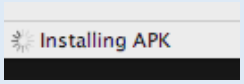
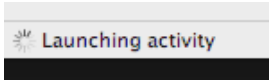
1. Gradle build running	
2. Waiting for target device to come online	
3. Installing APK	
4. Launching activity	

Table 1-5: AVD Emulator activities

24. Once your app builds and the emulator is ready, Android Studio uploads the app to the emulator. You now see **"Hello, World!"** displayed in the app.

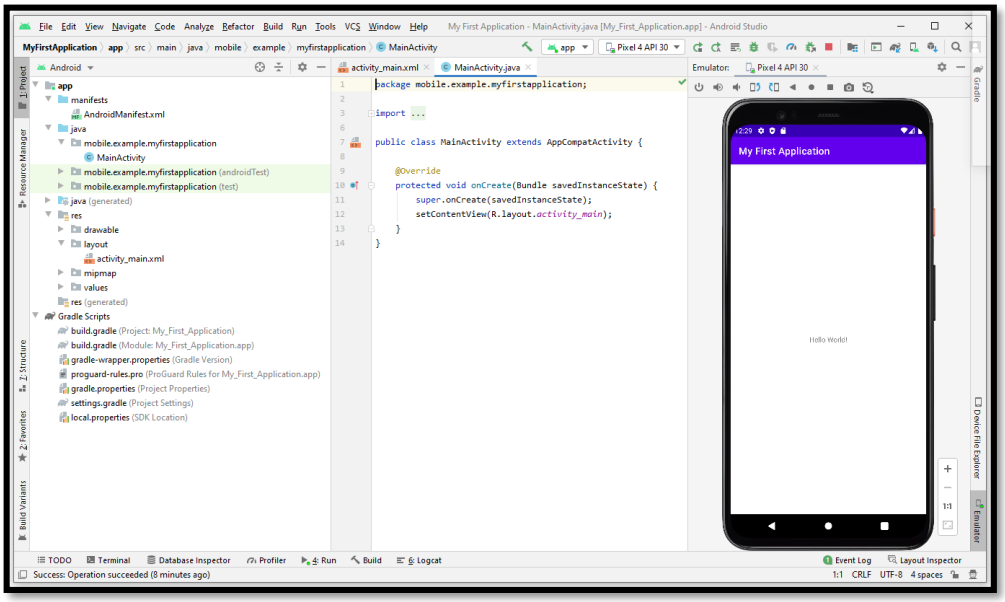


Figure 1-23: Android Studio environment

E. Import an existing project

To import an existing, local project into Android Studio, proceed as follows:

1. Click **File > New > Import Project**.
2. In the window that appears, navigate to the root directory of the project you want to import.
3. Click **OK**.

Android Studio then opens the project in a new IDE window and indexes its contents.

02

SET-UP THE DEVELOPMENT ENVIRONMENT FOR ANDROID

2.1
Design Application User Interface

2.2
Navigating Between Activities

2.3
Utilize Telephony and SMS Services

2.1 Design Application User Interface

Illustrate Activity Life Cycle

Activity in Android is one of the most important components of Android. It is the Activity where we put the UI of our application. Therefore, if we are new in Android development then we have to learn what is Activity on Android and what is the life cycle of Activity.

A. About the Activity

Every time we open the Android app, you will see some UI drawn on our screen. The screen is called **Activity**. An **Activity** represents a single screen with a user interface just like window or frame and provides a visual interface for user interaction. It is the basic component of Android and every time you open an app, then we are opening some activities. Each **Activity** usually supports one thing that the user can do, such as viewing an email message or showing a login screen.

For example:
When we open our Gmail app, then we see our email on the screen. The email is available in Activities. If we open some specific email, then that email will be opened in some other Activity.

In Android, **Activity** is where the Android Application begins its process. An **Activity** is an application user interface screen. There are a series of methods carried out in an **Activity**. Applications often comprise several activities.

In this chapter you learn about the activity lifecycle, the callback events you can implement to perform tasks in each stage of the lifecycle, and how to handle **Activity** instance states throughout the activity lifecycle.

B. About the Activity Lifecycle

The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it's created to when it's destroyed and the system reclaims its resources. As the user interacts with your app and other apps on the device, activities move into different states.

For example (refer Figure 2-24):

1. When you start an app, the app's main activity ("**Activity 1**" in the figure below) is started, comes to the foreground, and receives the user focus.

2. When you start a second activity ("**Activity 2**" in the figure below), a new activity is created and started, and the main activity is stopped.

3. When you're done with the **Activity 2** and navigate back, **Activity 1** resumes. **Activity 2** stops and is no longer needed.

4. If the user doesn't resume **Activity 2**, the system eventually destroys it.

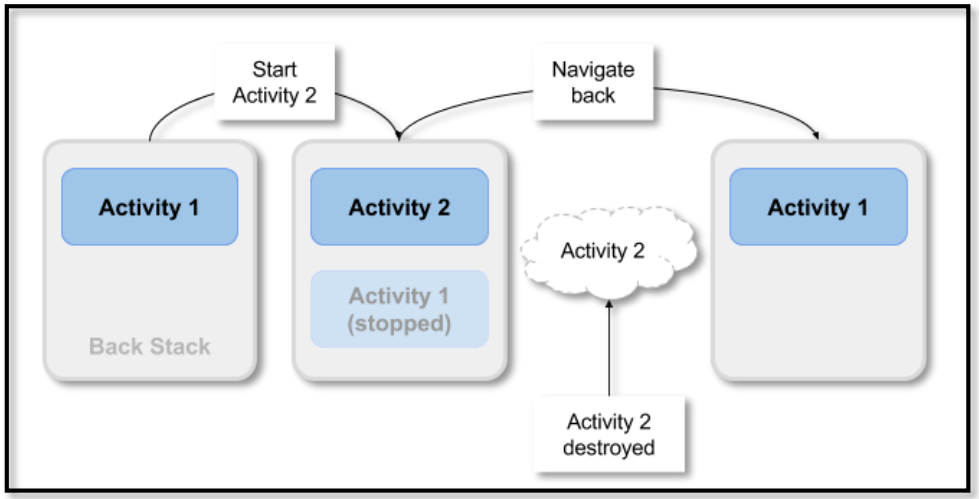


Figure 2-24: The activity lifecycle process

C. Activity States and Lifecycle Callback Methods

In general, activity lifecycle has **SEVEN (7) Callback Methods**. Let's see the seven (7) lifecycle methods of android Activity. The seven (7) lifecycle method of Activity describes how activity will behave at different states.

Figure 2-25 below shows each of the **Activity** states and the callback methods that occur as the **Activity** transitions between different states.

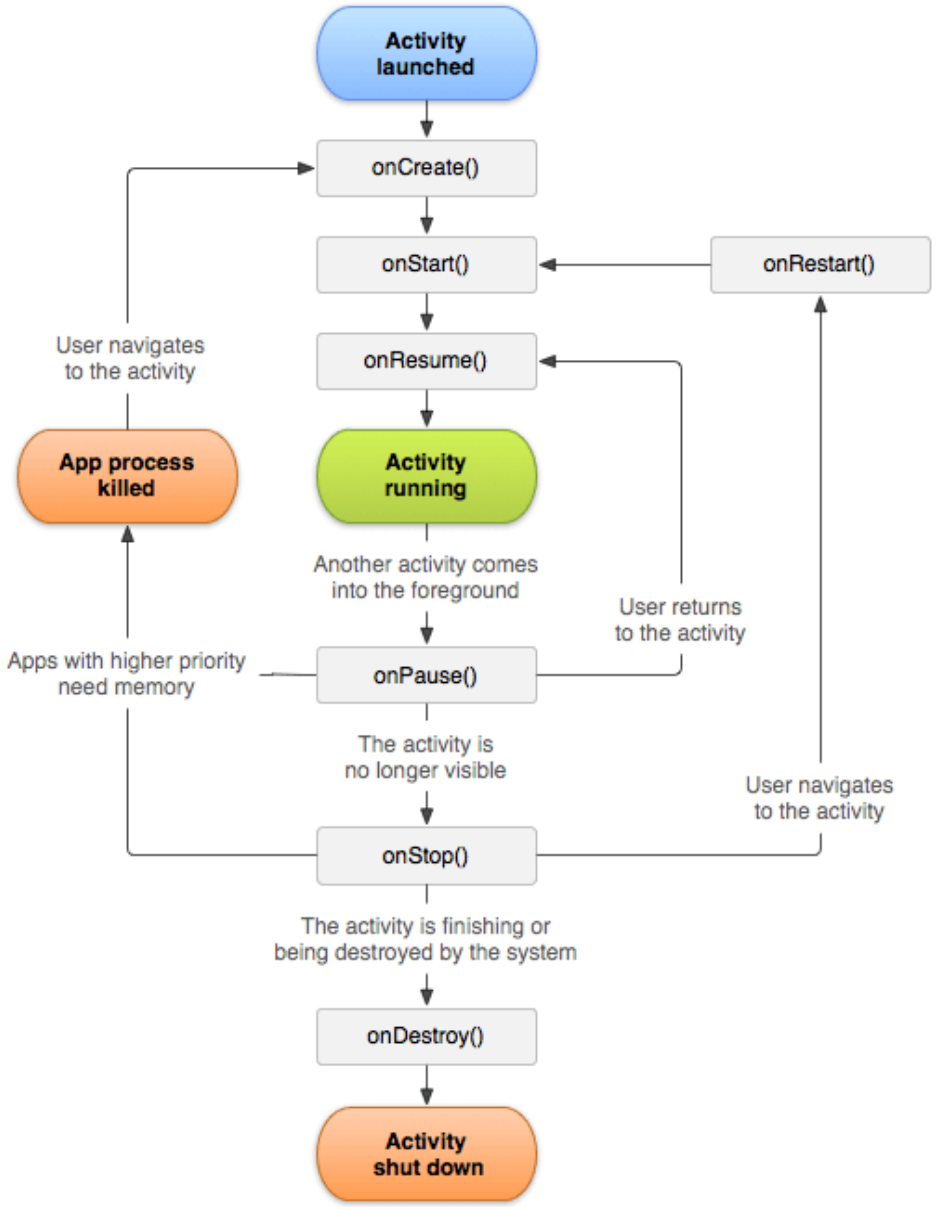


Figure 2-25: The Activity states and lifecycle callback methods

The Activity lifecycle consists of 7 methods:

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

Table 2-6: Activity Lifecycle's methods

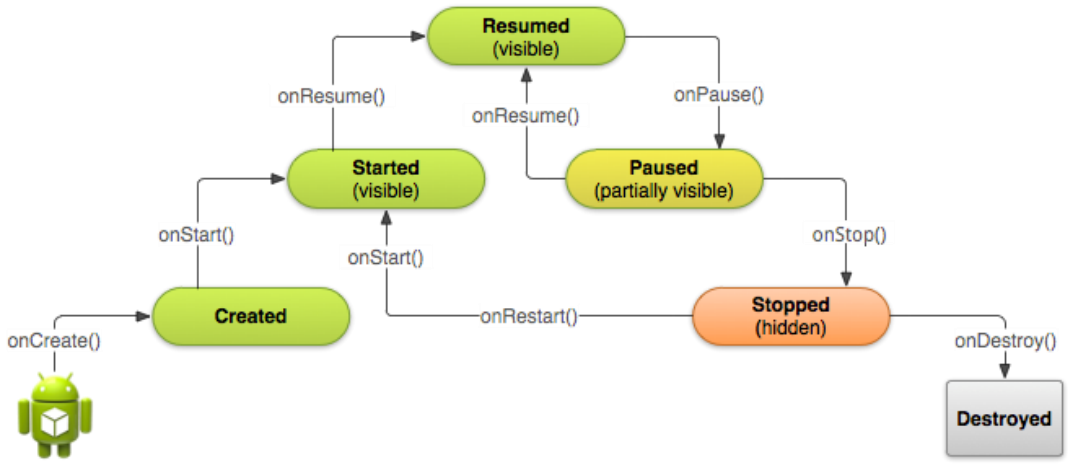


Figure 2-26: The Activity states and lifecycle callback methods

Depending on the complexity of your **Activity**, you probably don't need to implement all the lifecycle callback methods in an **Activity**. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect.

TUTORIAL: Android Activity Lifecycle methods

Learning Outcomes:
By the end of this tutorial, you should be able to create android app that will displaying the content on the logcat which show the activity life cycle.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Create android app to show the activity life cycle on the logcat

1. Open Android Studio and create new project. Name it as **ActivityLifeCycle**. Create your package as well. It should be name as **com.example.activitylifecycle**. Click **Finish**.

2. In the **MainActivity.java**, change the code in the **onCreate()** method as shown below:

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

    Log.d("lifecycle","onCreate invoked");
}
```

3. Then, still in **MainActivity.java**, add all the method in **Activity Life Cycle**, as shown below.

```
@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");
}
```

4. Import the **util.Log** package in your **MainActivity.java.**

```
import android.util.Log;
```

5. Leave your **activity_main.xml** file just the way it is. Now, run your program. You will not see any output on the AVD emulator or device. You need to open **logcat**.

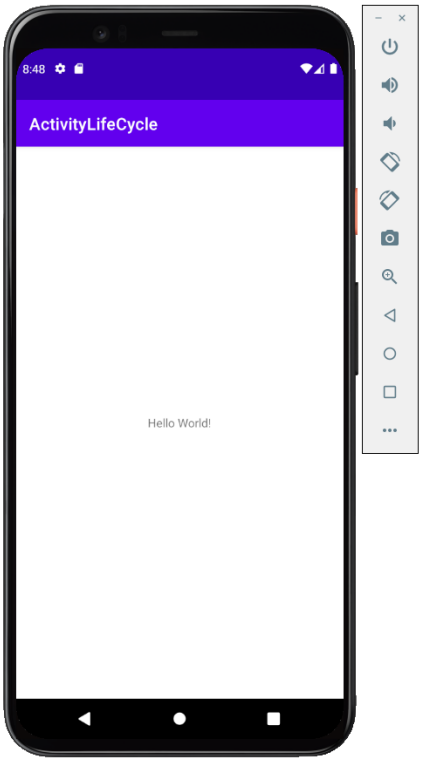


Figure 2-27: AVD emulator

6. See on the logcat: **onCreate**, **onStart** and **onResume** methods are invoked.

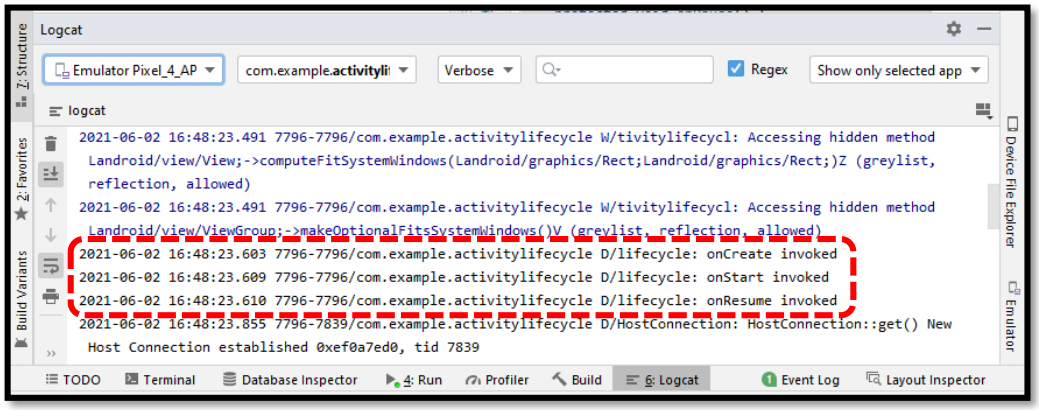


Figure 2-28: onCreate, onStart and onResume methods

7. When you run your app for the first time, in the logcat, it shows that the first method that invoke is **oncreate**, and then **onstart** and **onResume.**

8. Now click on the **HOME** button. You will see in the logcat, method **onPause** and **onStop** are invoked.

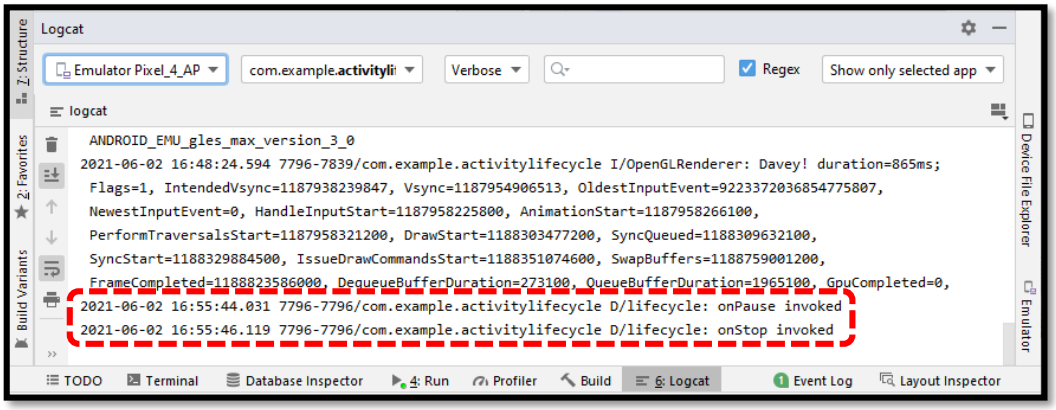


Figure 2-29: onPause and onStop methods

9. Now see on the emulator. It is on the **HOME**. Now click on the center button to launch the app again.

10. Now, click on the **ActivityLifeCycle** icon to launch the app again.

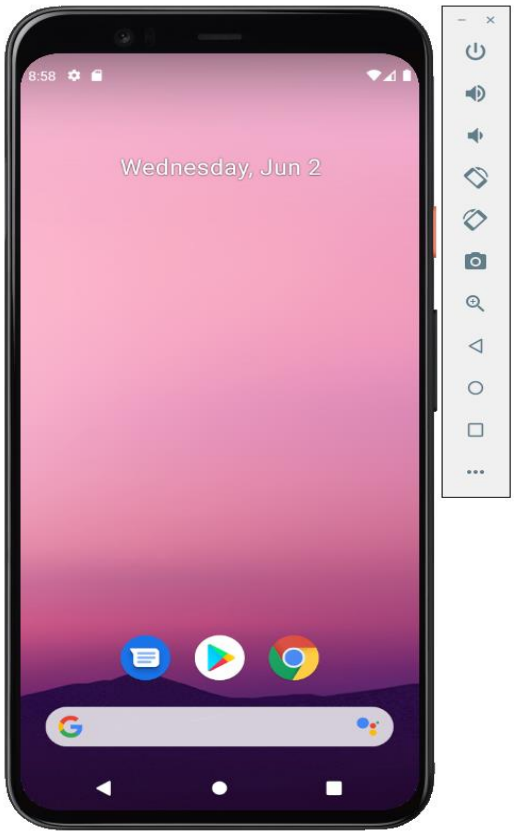


Figure 2-30: HOME

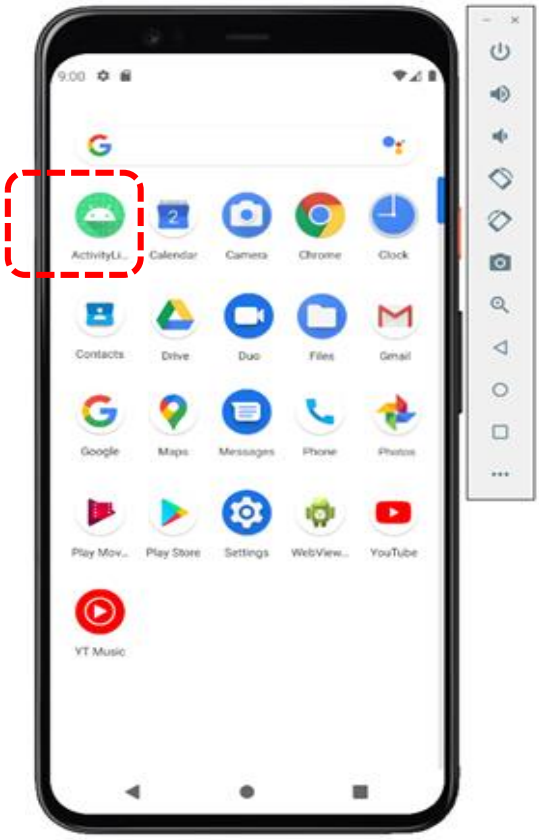


Figure 2-31: ActivityLifeCycle icon

11. Now, see on the logcat: **onRestart**, **onStart** and **onResume** methods are invoked.

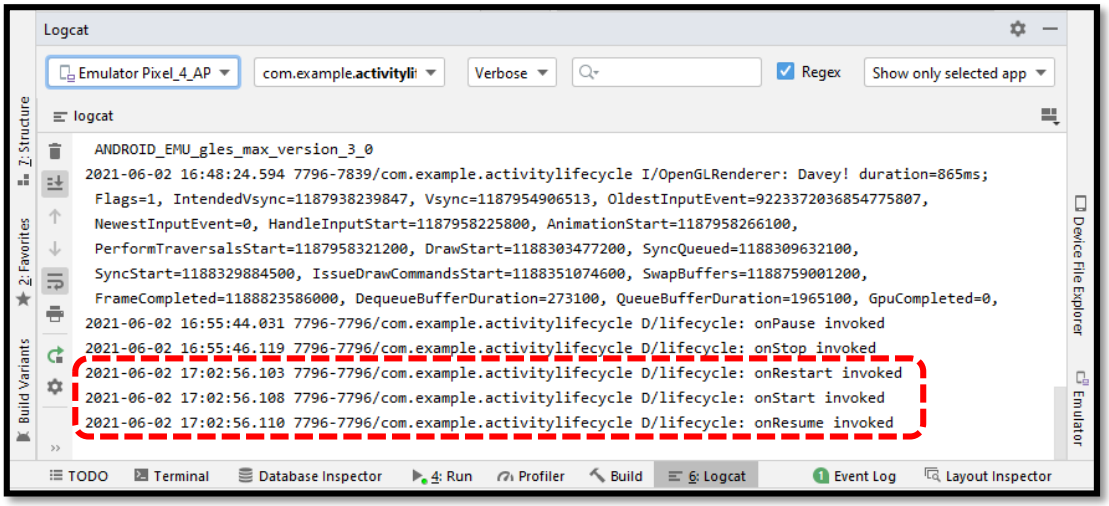


Figure 2-32: onRestart, onStart and onResume methods

12. If you see the emulator, application is started again.

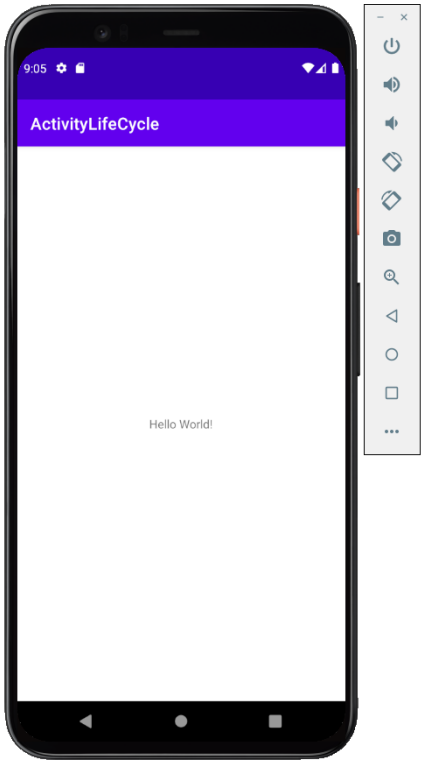


Figure 2-33: ActivityLifeCycle started again

13. You also can do other task such as open another app. As you do that, it also will invoke the **onPause** and **onStop** method.

14. Now click on the **Back** button (closed the app). You will see method **onPause**, **onStop** and **onDestroy** will be invoke. And that's how its explain the Activity Life Cycle.

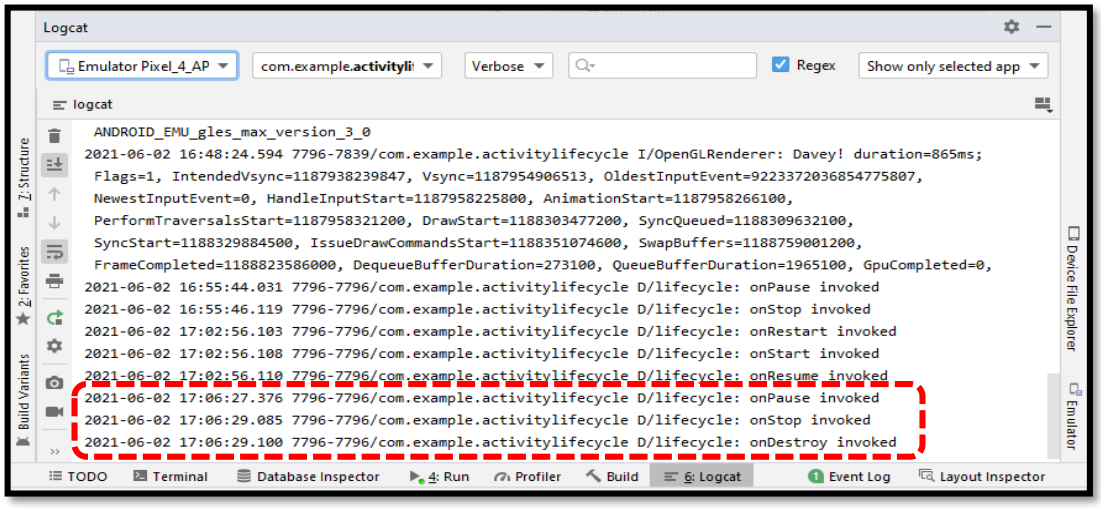


Figure 2-34: onPause, onStop and onDestroy methods

User Interface (UI) of Android Application

A. User Interfaces

The UI is placed on the Activity via the Activity's **setContentView()** method. In Android, the UI composes of **View** and **ViewGroup** objects, organized in a single view-tree structure. Once a view-tree is constructed, you can add the root of the view-tree to the Activity as the content view via Activity's **setContentView()** method.

B. View

The first thing in Android you need to learn is something called **Views**. A **View** is a rectangular area visible on the screen where the user can see and interact with. A **View** is an interactive UI component, widget or control, such as **TextView**, **EditText**, **Button**, **RadioButton**, etc., in package **android.widget**. It has a width and height, and sometimes a background color. A **View** is responsible for drawing itself and handling events such as clicking and entering texts.

The illustration (Figure 2-35) shows **Views** of three different types.

An **ImageView** displays an image such as an icon or photo. A **TextView** displays text. A **Button** is a **TextView** that is sensitive to touch: tap it with your finger and it will respond.

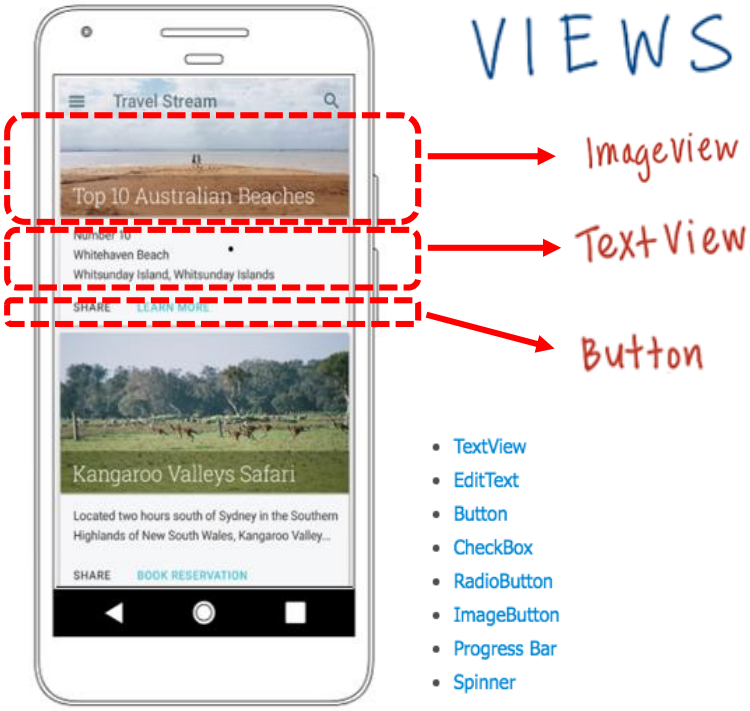


Figure 2-35: Views

C. ViewGroup

ViewGroup is an invisible container that defines the layout structure for **View** components. There are many types of ready-to-use **ViewGroups** in Android. Following are the commonly used **ViewGroup** subclasses in android applications:

Linear Layout, Relative Layout, Table Layout, Frame Layout, Web View, List View and Grid View.

A **ViewGroup** is a big **View** that can contain smaller **Views** inside of it. The smaller **Views** are called the children of the **ViewGroup** and might be **TextViews** or **ImageViews**. The **ViewGroup** is called the parent of its children. The illustration (Figure 2-36) shows one of the most common **ViewGroups**, a vertical **LinearLayout**.

The **ViewGroup** itself might be transparent, serving only to contain and position its children. Its children, however, will almost always be visible. **ViewGroup** has height, width, background color and other attributes, even its also can be transparent bacground.

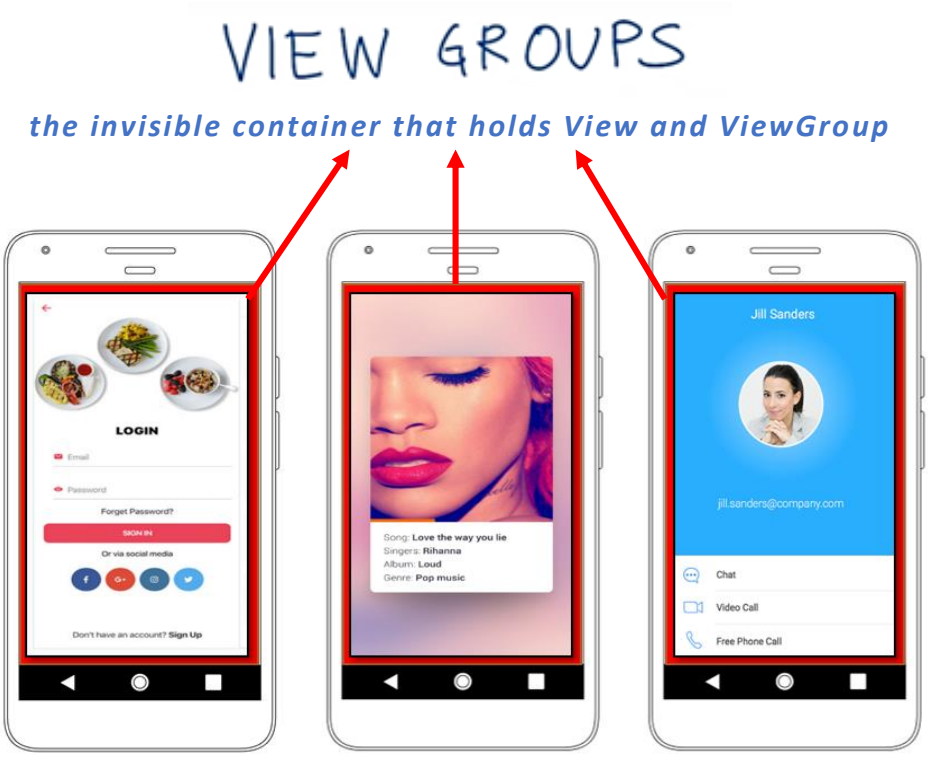


Figure 2-36: ViewGroups

D. XML Tag

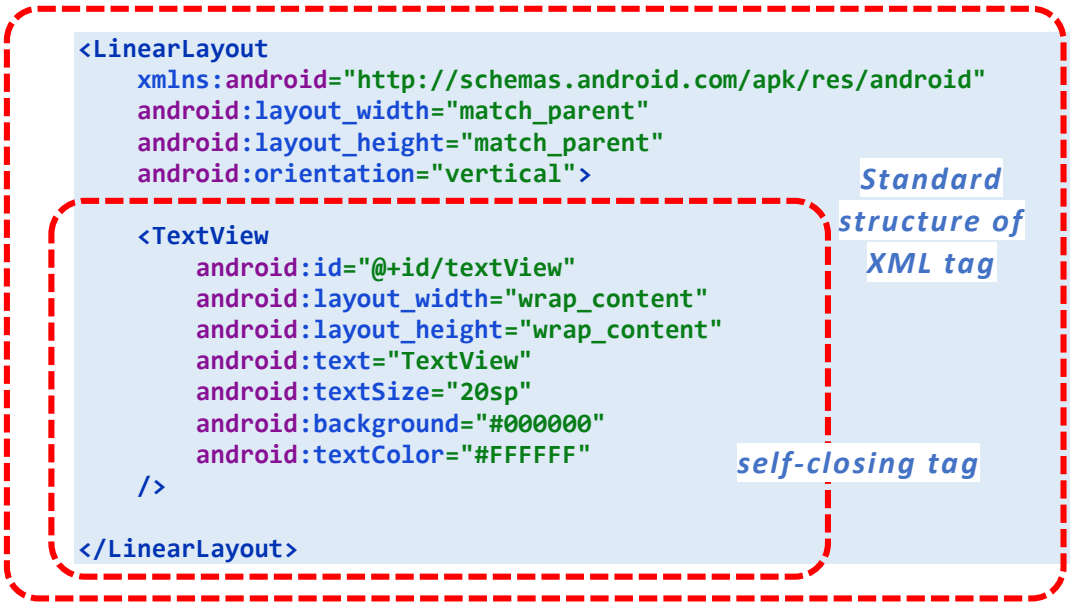
The XML is a notation for writing a file containing pieces of information called **elements**. To indicate where an element begins and ends, we write tags. A tag is easy to recognize because it always begins and ends with the characters **<** and **>**. An element often consists of a pair of tags, plus all the content between them. The standard structure of XML tag looks like this:

```
<tag_name
    attribute 1_name="attribute1_value"
    attribute 2_name="attribute 2_value"
    attributeN_name="attributeN_value" >

    some content
    ........

</tag_name>
```

An element that does not need to enclose any content can consist of a single tag. In this case, the tag ends with the characters **/>** and we say that it is a **self-closing tag**. The standard structure of XML self-closing tag looks like this:

```
<tag_name
    attributes..
    attributes..
/>
```

Sample code:




```
<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/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="TextView"
        android:textSize="20sp"
        android:background="#000000"
        android:textColor="#FFFFFF"
    />

</LinearLayout>
```

E. Android Defining Styles

To create an **Android defining style**, we need to follow the below steps:

- We need to add **<style>** element in the XML file with a **name** attribute to uniquely identify the style.
- To define attributes of **style**, we need to add an **<item>** elements with a name that defines a style attribute and we need to add appropriate value to each **<item>** element.

Following is the example of defining a style in separate XML file using **<style>** element. We created a style "**TextviewStyle**" with all required style attributes.

```
<style name="TextviewStyle">
    <item name="android:id">textView</item>
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:text">TextView</item>
    <item name="android:textSize">20sp</item>
    <item name="android:backround">#000000</item>
    <item name="android:textColor">#FFFFFF</item>
</style>
```

F. Density Independence Pixels

Density Independent Pixel is an abstract unit that is based on the density of a screen. These units help maintain UI elements with the same physical dimensions across different density devices while maintaining the element's sharpness. The denser the screen, the more pixels are needed to maintain the same physical dimensions. In Android Development, we have seen many developers using **device-independent pixels (dp)** and **scale-independent pixels (sp)** as a measurement unit for all the views. Both **dp** and **sp** follow the concept of density and can be used almost identically, albeit with a few differences.

Device-Independent Pixels (dp)	Scale-Independent Pixels (sp)
It used for **defining the sizes** in all widgets, ranging from TextView to LinearLayout.	It is used for **defining text size**, as it scales according to the font size preference on a mobile device.

Sample code:

```
<Button
    android:layout_width="75dp"
    android:layout_height="60dp"
    android:textSize="18sp"
/>
```

G. Hex Color (Hexadecimal Color)

A color is created by mixing together red, green, and blue, in that order. Write a **hash sign (#)** and then specify the amount of each component with a pair of "**hexadecimal digits**" where 00 is the minimum amount, FF is the maximum, and 80 is halfway. You can directly specify the value as HEX color code as we do for CSS files in HTML.

Sample code:

```
android:background="#FFFFCC"
android:textColor="#9C27B0"
```

TUTORIAL: Create User Interface of Android Application

Learning Outcomes:
By the end of this tutorial, you should be able use the Android Studio Layout Editor to create a layout that includes a textbox and a button.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Open the Layout Editor

1. In the Project window, open **app > res > layout > activity_main.xml.**

2. If your editor shows the XML source, click the **Design** tab at the top right of the window.

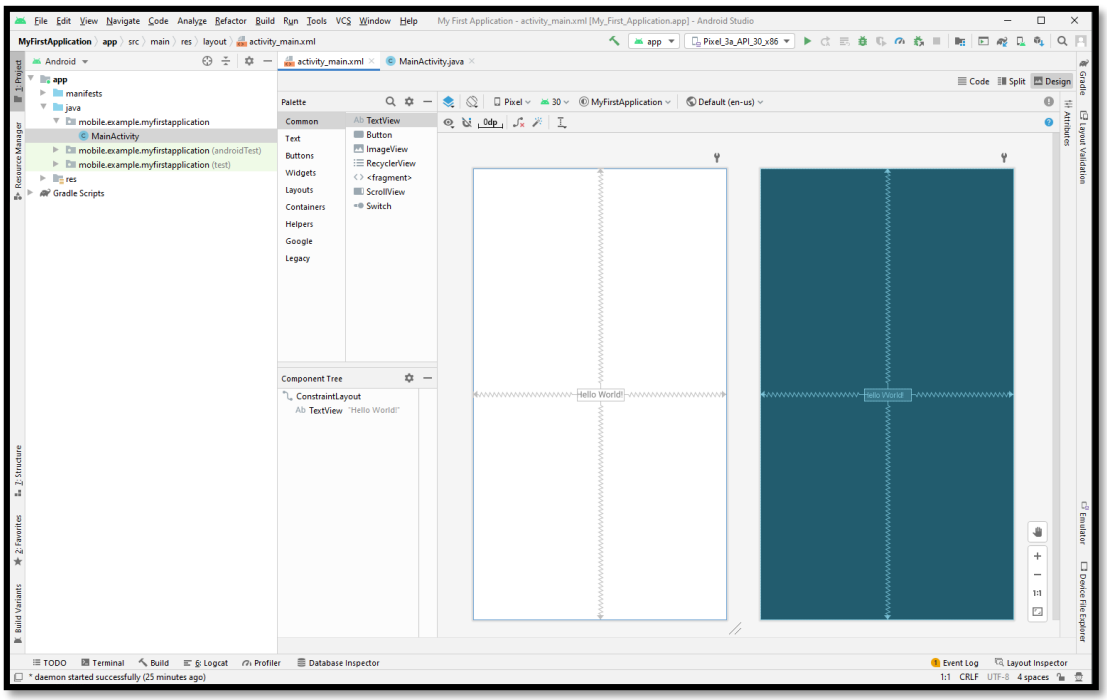


*Figure 2-37: The Layout Editor showing **activity_main.xml***

3. The **Component Tree** panel shows the layout's hierarchy of views. In this case, the root view is a **ConstraintLayout**, which contains just one **TextView** object.

4. Notice the **Palette** at the top left of the layout editor. Move the sides if you need to, so that you can see many of the items in the palette.

5. Click on some of the categories, and scroll the listed items if needed to get an idea of what's available.

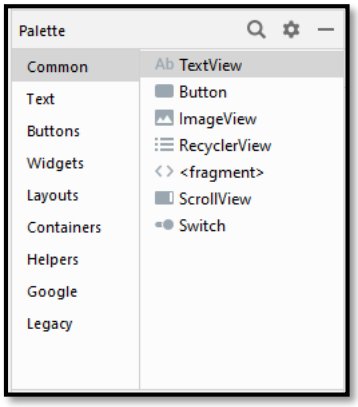


Figure 2-38: Palette

B. TextView with example in Android Studio

6. **TextView** displays text to the user and optionally allows them to edit it programmatically.

TextView code in XML:

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintHorizontal_bias="0.247"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

7. Attributes of **TextView** :
   - **id:** used to uniquely identify a text view.
   - **layout_width :** specifies the basic width of the view.
   - **layout_height:** specifies the basic height of the view.
   - **text:** used to set the text in a text view.
   - **wrap_content:** the view expands only as much as needed to fit its contents.

C. Add Button and constrain the position

8. Let's add **Button** and align the baselines by move the cursor over the circle at the top of the **Button** onto the circle at the top of the **TextView**.

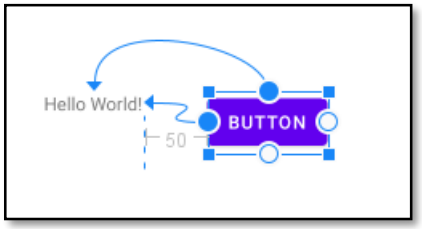


Figure 2-39: The Button is aligned with the TextView

Button code in XML:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="50dp"
    android:layout_marginLeft="50dp"
    android:text="Button"
    app:layout_constraintStart_toEndOf="@+id/textView"
    app:layout_constraintTop_toTopOf="@+id/textView" />
```

9. To delete an individual constraint, hover over the circular handle and click it has it turns thick line and then **Delete**.

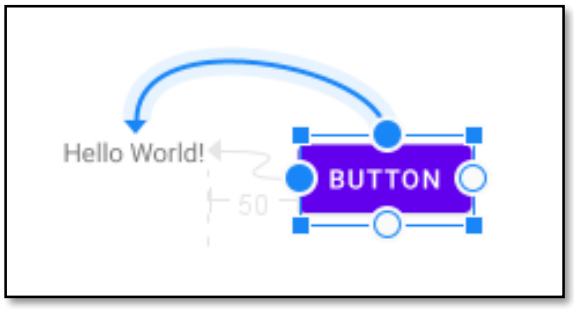


Figure 2-40: Deleting constraints

10. Before adding another **Button**, relabel this **Button** so things are a little clearer about which **Button** is which.
    - Click on the **Button** you just added in the design layout.
    - Look at the **Attributes** panel on the right, and notice the **id** field.
    - Change the **id** from **button** to **second_button**.

11. You can go ahead and try replacing some of your own layouts with a constraint layout.

Organize Layout

A **layout** defines the structure for a user interface in Android application, such as in an activity. 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, as shown in figure below.

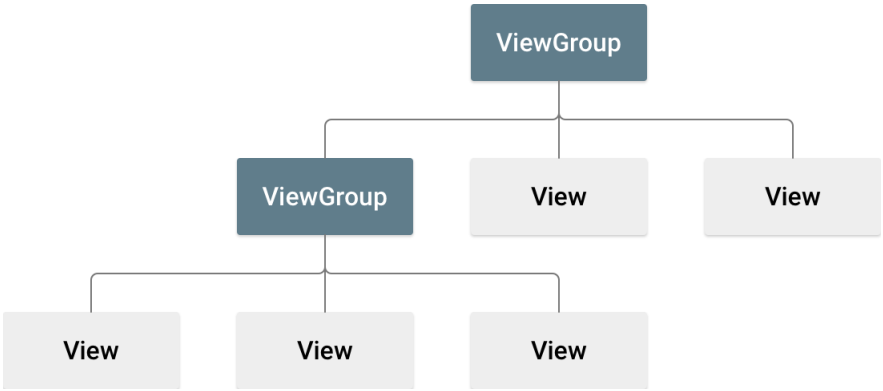


Figure 2-41: Illustration of a view hierarchy, which defines a UI layout.

Types of Layouts

A. Frame Layout - *placeholder on screen that you can use to display a single view*

Framelayout is a **ViewGroup** subclass that is used to specify the position of **View** instances it contains on the top of each other to display only single **View** inside the **FrameLayout**. In simple manner, **FrameLayout** is designed to block out an area on the screen to display a single item.

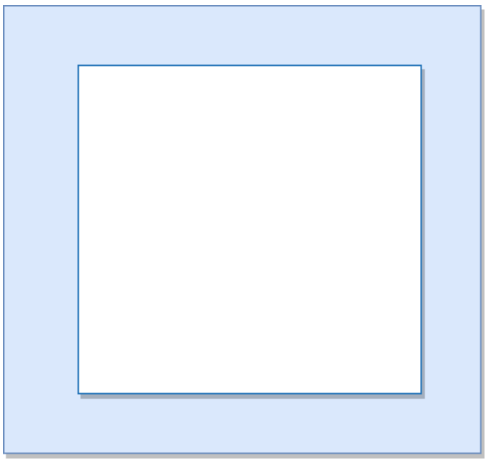


*Figure 2-42: The pictorial representation of **FrameLayout** in android applications*

In android, **FrameLayout** will act as a placeholder on the screen, and it is used to hold a single child view. In **FrameLayout**, the child views are added in a stack and the most recently added child will show on the top. We can add multiple children views to **FrameLayout** and control their position by using gravity attributes in **FrameLayout**.

Sample code:

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="centerCrop"
        app:srcCompat="@drawable/orange" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="40sp"
        android:text="This activity using ....."
        android:textSize="28sp"
        android:background="#4C374A"
```

```
        android:textColor="#FFFFFF" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="right|bottom"
        android:layout_marginBottom="40sp"
        android:text="Frame Layout"
        android:textSize="28sp"
        android:background="#4C374A"
        android:textColor="#FFFFFF" />

</FrameLayout>
```

*Figure 2-43: Output of Android **FrameLayout** example*

B. Linear Layout - ***aligns all children in a single direction, vertically or horizontally***

LinearLayout is a **ViewGroup** subclass which is used to render all child **View** instances one by one either in **Horizontal** direction or **Vertical** direction based on the **orientation** property. **LinearLayout** orientation can be specified using **android:orientation** attribute.

In **LinearLayout**, the child **View** instances arranged one by one, so the horizontal list will have only one row of multiple columns and vertical list will have one column of multiple rows.

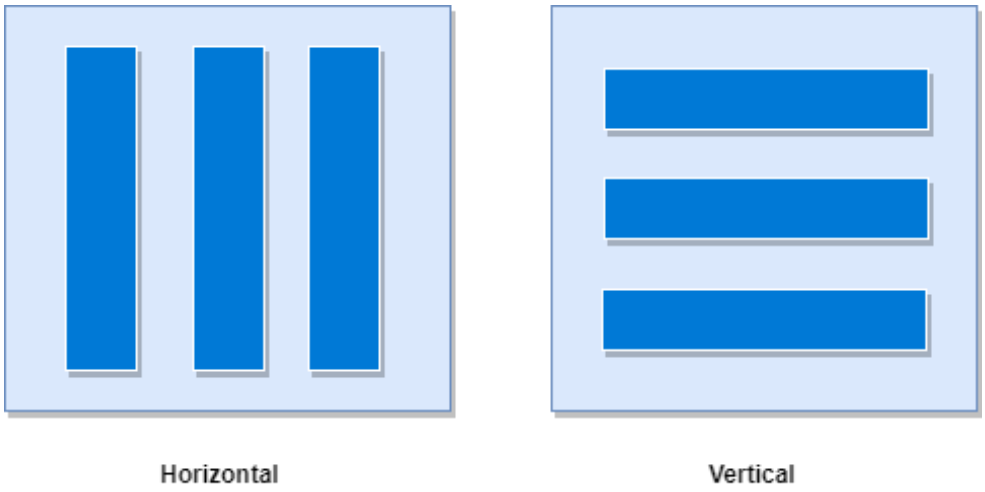


*Figure 2-44: The pictorial representation of **LinearLayout** in android applications*

Sample code:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/txtTo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="To"/>
    <EditText
        android:id="@+id/txtSub"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Subject"/>
    <EditText
        android:id="@+id/txtMsg"
        android:layout_width="match_parent"
```

```
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="Message"/>
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="Send"/>

</LinearLayout>
```

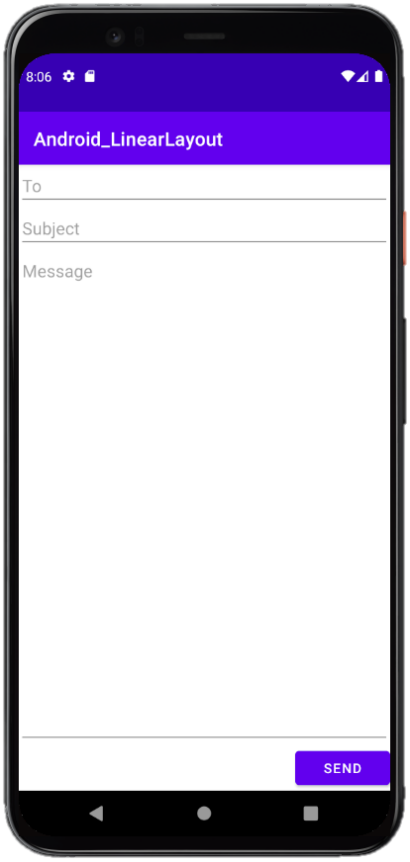


*Figure 2-45: Output of Android **LinearLayout** example*

C. Table Layout - *groups views into rows and columns*

TableLayout is a **ViewGroup** subclass that is used to display the child **View** elements in rows and columns. In android, **TableLayout** will position its children's elements into rows and columns, and it won't display any border lines for rows, columns, or cells. The **TableLayout** in android will work same as the HTML table and the table will have as many columns as the row with the most cells. The **TableLayout** can be explained as **<table>** and **TableRow** is like **<tr>** element.

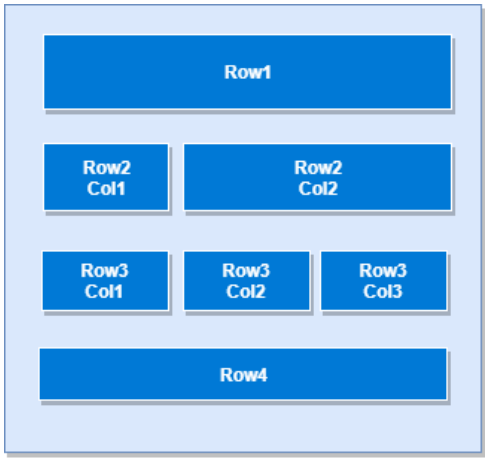


*Figure 2-46: The pictorial representation of **TableLayout** in android applications*

Sample code:

```
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="100dp"
    android:paddingLeft="10dp"
    android:paddingRight="10dp" >


    <TableRow android:background="#33FCFF" android:padding="5dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Student Id" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="User Name" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Location" />
    </TableRow>
```

```
    <TableRow android:background="#DAE8FC" android:padding="5dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="1" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Suzana Dasari" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Dungun" />
    </TableRow>

    <TableRow android:background="#DAE8FC" android:padding="5dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="2" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Rohana Alisha" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Kuala Terengganu" />
    </TableRow>

    <TableRow android:background="#DAE8FC" android:padding="5dp">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="3" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Trisha Divasini" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Besut" />
    </TableRow>

</TableLayout>
```

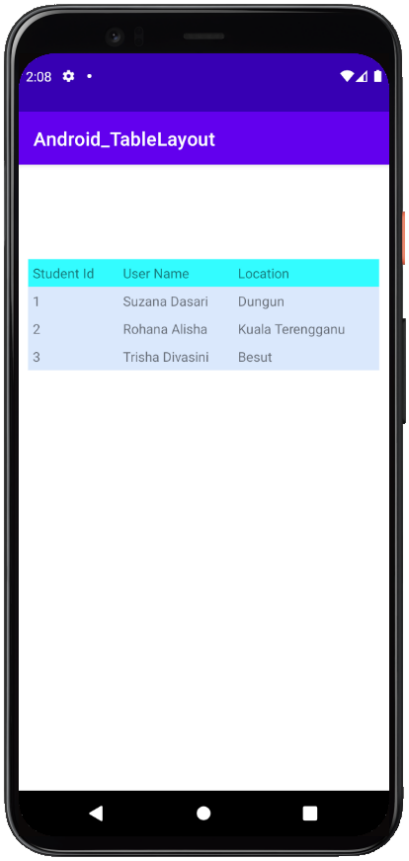


*Figure 2-47: Output of Android **TableLayout** example*

D. Relative Layout - displays child views in relative positions

RelativeLayout is a **ViewGroup** which is used to specify the position of child **View** instances relative to each other (Child A to the left of Child B) or relative to the parent (aligned to the top of parent). In android, **RelativeLayout** is very useful to design user interface because by using **RelativeLayout** we can eliminate the nested view groups and keep our layout hierarchy flat, which improves the performance of application.

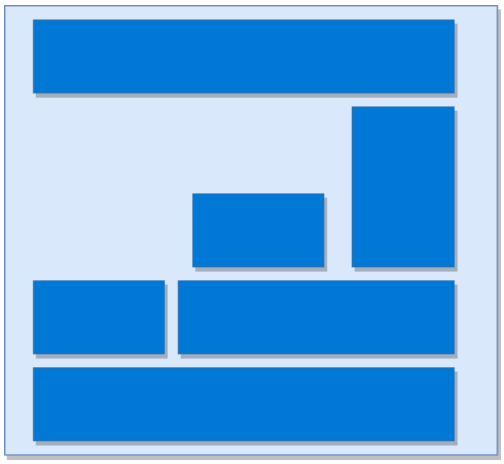


*Figure 2-48: The pictorial representation of **RelativeLayout** in android applications*

In **RelativeLayout** we need to specify the position of child views relative to each other or relative to the parent. In case if we didn't specify the position of child views, by default all child views are positioned to top-left of the layout.

Sample code:

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="10dp"
    android:paddingRight="10dp" >

    <Button
        android:id="@+id/btn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:text="Button1" />

    <Button
        android:id="@+id/btn2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:text="Button2" />
```

```
    <Button
        android:id="@+id/btn3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        android:text="Button3" />

    <Button
        android:id="@+id/btn4"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Button4" />

    <Button
        android:id="@+id/btn5"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBottom="@+id/btn2"
        android:layout_centerHorizontal="true"
        android:text="Button5" />

    <Button
        android:id="@+id/btn6"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@+id/btn4"
        android:layout_centerHorizontal="true"
        android:text="Button6" />

    <Button
        android:id="@+id/btn7"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toEndOf="@+id/btn1"
        android:layout_toRightOf="@+id/btn1"
        android:layout_alignParentRight="true"
        android:text="Button7" />

</RelativeLayout>
```

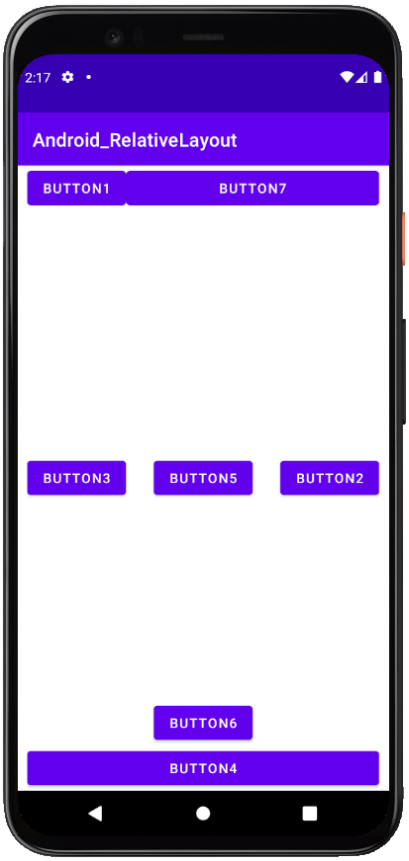


*Figure 2-49: Output of Android **RelativeLayout** example*

E. Grid Layout

In Android **GridLayout**, we can specify the number of columns and rows that the grid will have. We can customize the **GridLayout** according to our requirements, like setting the size, color or the margin for the Layout.

A **GridLayout** basically places its children in a rectangular grid. This grid has a set of a number of thin lines that separate the view area into cells. Suppose you have a grid of **N** columns, then we will have **N+1** grid indices that would be starting from **0.**

The number of rows and columns within the grid can be declared using the **android:rowCount** and **android:columnCount** properties.

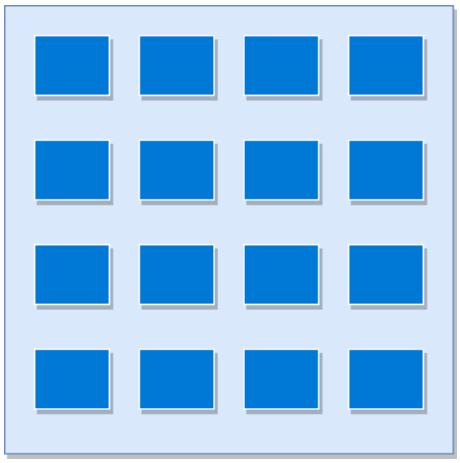


*Figure 2-50: The pictorial representation of **GridLayout** in android applications*

Sample code:

```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/GridLayout1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:columnCount="2"
    android:rowCount="2" >

    <Button
        android:id="@+id/button1"
        android:layout_gravity="left|top"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_gravity="left|top"
        android:text="Button 2" />

    <Button
        android:id="@+id/button3"
        android:layout_gravity="left|top"
        android:text="Button 3" />
```

```
    <Button
        android:id="@+id/button4"
        android:layout_gravity="left|top"
        android:text="Button 4" />

</GridLayout>
```

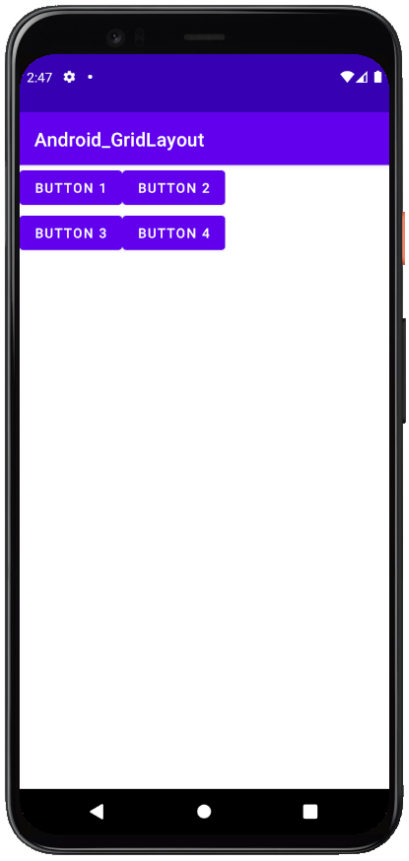


*Figure 2-51: Output of Android **GridLayout** example*

Layout Attributes

Each layout has a set of attributes which define the visual properties of that layout. There are few common attributes among all the layouts and their are other attributes which are specific to that layout.

Following are common attributes and will be applied to all the layouts.

Attribute	Description
android:id	Specifies the ID which uniquely identifies the view.
android:layout_width	Specifies the width of the layout.
android:layout_height	Specifies the height of the layout
android:layout_marginTop	Specifies the extra space on the top side of the layout.
android:layout_marginBottom	Specifies the extra space on the bottom side of the layout.
android:layout_marginLeft	Specifies the extra space on the left side of the layout.
android:layout_marginRight	Specifies the extra space on the right side of the layout.
android:layout_gravity	Specifies how child Views are positioned.
android:layout_weight	Specifies how much of the extra space in the layout should be allocated to the View.
android:layout_x	Specifies the x-coordinate of the layout.
android:layout_y	Specifies the y-coordinate of the layout.
android:layout_width	Specifies the width of the layout.
android:paddingLeft	Specifies the left padding filled for the layout.
android:paddingRight	Specifies the right padding filled for the layout.
android:paddingTop	Specifies the top padding filled for the layout.
android:paddingBottom	Specifies the bottom padding filled for the layout.

Table 2-7: Layout attributes

Adapt to Display Orientation

The **screenOrientation**, also known as **screen rotation** or **display orientation** is the attribute of activity element. The orientation of android activity can be portrait, landscape, sensor, unspecified etc. You need to define it in the **AndroidManifest.xml** file.

Sample code:

```
<activity android:name="package_name.Your_ActivityName"
  android:screenOrientation="orientation_type">
</activity>
```

Example:

```
<activity android:name=".MainActivity"
  android:screenOrientation="portrait">
</activity>
```

```
<activity android:name=".SecondActivity"
  android:screenOrientation="landscape">
</activity>
```

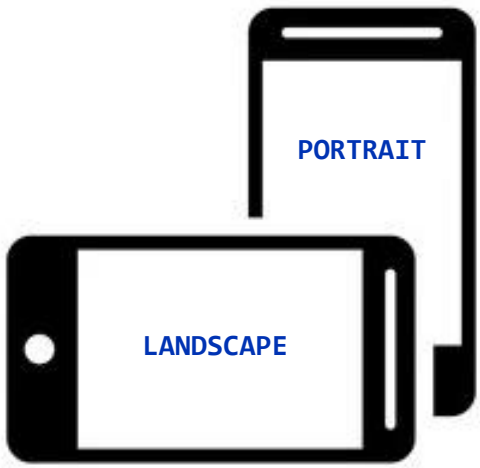


Figure 2-52: Display orietation

TUTORIAL: Change Display Orientation

Learning Outcomes:
By the end of this tutorial, you should be able to change screen orientation for Landscape and Portrait mode.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Creating the Activities

1. Create two (2) activities of different screen orientation. The first activity will be as "**portrait**" orientation and Second activity as "**landscape**" orientation state.

2. **Creating the XML file:**
   - **activity_main.xml**: XML file for first activity consist of constraint layout with **Button** and **TextView** in it. This activity is in **Potrait** state.

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Potrait Orientation" />

<Button
    android:id="@+id/btnNext"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onClick"
    android:text="Next Activity" />
```

   - **activity_next.xml**: XML file for second activity consist of constraint layout with **TextView** in it. This activity is in **Landscape** state.

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Landscape Orientation" />
```

3. **Creating the Java file:**
   - **MainActivity.java**: Java file for Main Activity, in which **setOnClick()** *listener* is attached to the button to launch next activity with different orientation.

```
public class MainActivity extends AppCompatActivity {

    // declare button variable
    Button button;

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

        // initialise button with id
        button = findViewById(R.id.btnNext);
    }

    // onClickListener attached to button
    // to send intent to next activity
    public void onClick(View v){
        // Create instance of intent and
        // startActivity with intent object
        Intent intent = new Intent(MainActivity.this,
                NextActivity.class);
        startActivity(intent);
    }
}
```

   - **NextActivity.java**: Java file for Next Activity, which is in Landscape orientation.

```
public class NextActivity extends AppCompatActivity {

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

4. **Updating the AndroidManifest file:** In **AndroidManifest.xml** file, add the **screenOrientation** state in activity along with its orientation. Here, we provide "**portrait**" orientation for **MainActivity** and "**landscape**" for **NextActivity**. Below is the code for AndroidManifest file.

    - Define potrait orientation for **MainActivity:**

```
<activity android:name=".MainActivity"
    android:screenOrientation="portrait">
```

    - Define landscape orientation for **NextActivity:**

```
<activity android:name=".NextActivity"
    android:screenOrientation="landscape">
```

5. Now, run your program. You will see output on the emulator or device.

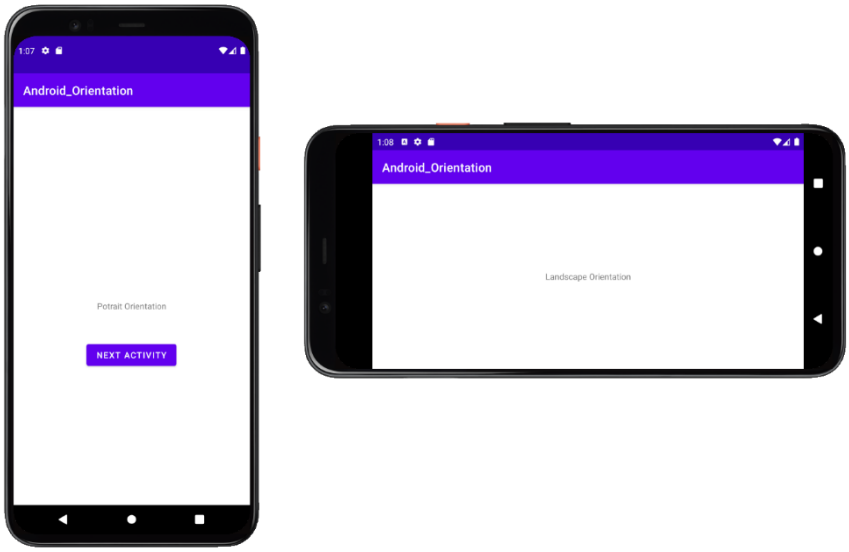


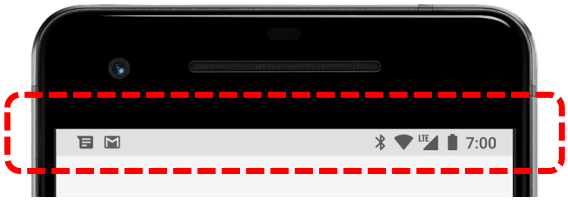
Figure 2-53: Display orientation output

Add Notifications and Actions to the Action Bar

Notifications are messages that Android displays outside of your app's UI to give users alerts, reminders, communications from others, or other real-time information from your app. Users can tap the notification to open your app or take action directly from the notification. **Notifications** appear to users in different locations and formats, such as icons in the status bar, more detailed entries in the notification drawer, as badges on app icons, and on automatically paired devices.

Status Bar and Notification Drawer

1. When you issue a notification, it first appears as an icon in the status bar.



The status bar contains the clock, battery icon, and other notification icons as shown in an image. Most of the time, it is at the top of the screen.

Figure 2-54: Status bar

2. Users can swipe down on the status bar to open the notification drawer, where they can see more details and take action with the notification.

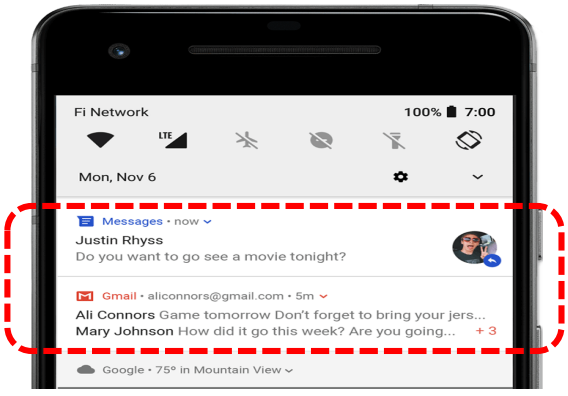


Figure 2-55: Notification drawer

3. Users can drag notifications in the drawer to show an expanded view, which shows additional content and action buttons, if provided.
4. Notifications remain visible in the notification drawer until terminated by an application or user.

Notification Anatomy

The design of a notification is determined by system templates.

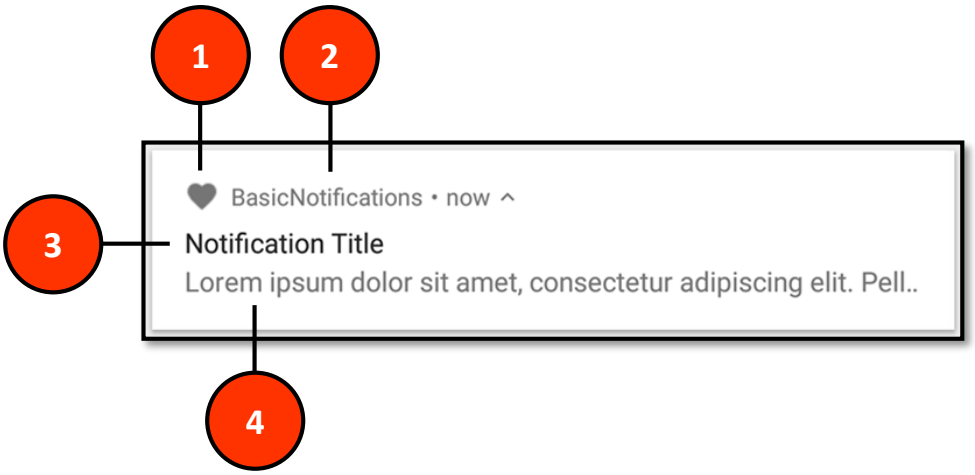


Figure 2-56: A notification with basic details

The most common parts of a notification are indicated in Figure 2-56 as follows:

1. **Small icon**: This is required and set with **setSmallIcon().**
2. **App name:** This is provided by the system.
3. **Title**: This is optional and set with **setContentTitle()**.
4. **Text**: This is optional and set with **setContentText()**.

Action Bar

The **Action Bar**, if it exists for an activity, will be at the top of the activity's content area, typically directly underneath the status bar. It is a menu bar that runs across the top of the activity screen in android. **Android ActionBar** can contain menu items that become visible when the user clicks the "**menu**" button.

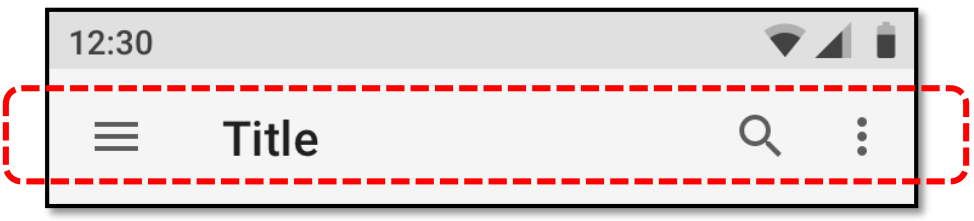


Figure 2-57: Action bar

TUTORIAL: Create a Basic Notification

Learning Outcomes:
By the end of this tutorial, you should be able to create a notification that the user can click on to launch an activity in your app.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Set the notification content and notification's tap action

1. Create two (2) activities. The first activity will be as "**MainActivity**" consist of **Button** and **TextView** in it and second activity as "**NotificationActivity**" consist of **TextView** in it.

2. In the **MainActivity.java**, add **showNotification()** method as shown below.

    - To get started, you need to set the notification's content and channel using a **NotificationCompat.Builder** object.
    - Every notification should respond to a tap, usually to open an activity in your app that corresponds to the notification. To do so, you must specify a content intent defined with a **PendingIntent** object and pass it to **setContentIntent()**.

```
private void showNotification(Intent i) {

    PendingIntent pendingIntent = PendingIntent.getActivity(MainActivity.this,
        0, i, PendingIntent.FLAG_UPDATE_CURRENT);

    //Set the notification content
    NotificationCompat.Builder builder
        = new NotificationCompat.Builder(MainActivity.this, "MYChannel")
        .setSmallIcon(R.drawable.ic_baseline_announcement)
        .setContentTitle("My Notification")
        .setContentText("Hello world!! Let's create notification")
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)

        //Set the intent that will fire when the user taps the notification
        .setContentIntent(pendingIntent)
        .setAutoCancel(true);

    //Show the notification
    NotificationManager notificationManager
        = (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    notificationManager.notify(NOTIFICATION_ID, builder.build());

}
```

B. **Create a channel and set the importance**

3. Still in the **MainActivity.java**, add **createNotificationChannel()** method as shown below.

```
private void createNotificationChannel() {

    // Create the NotificationChannel, but only on API 26+ because
    // the NotificationChannel class is new and not in the support library
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {

        CharSequence name = "MY Channel";
        String description = "Channel for MY notifications";
        int importance = NotificationManager.IMPORTANCE_DEFAULT;

        NotificationChannel channel = new NotificationChannel("MYChannel",
                name, importance);
        channel.setDescription(description);

        // Register the channel with the system; you can't change the importance
        // or other notification behaviors after this
        NotificationManager notificationManager =
                getSystemService(NotificationManager.class);
        notificationManager.createNotificationChannel(channel);
    }
}
```

C. **Modify the onCreate() method**

4. Change the code in the **onCreate ()** method as shown below.

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

    //call method
    createNotificationChannel();

    btnNotify = (Button) findViewById(R.id.buttonNoti);
    getSupportActionBar().setSubtitle("Let's create notification.");

    //function for button notification
    btnNotify.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            //Create an explicit intent for an Activity in app
            Intent i = new Intent(getApplicationContext(),
                    NotificationActivity.class);
            i.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP);
            //call method
            showNotification(i);
        }
    });

}
```

5. Now, run your program. You will get output of the notification app as shown below.

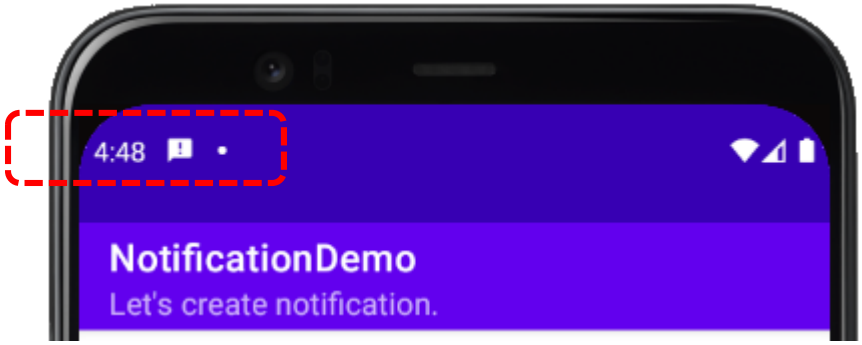


Figure 2-58: Notification icons appear on the left side of the status bar

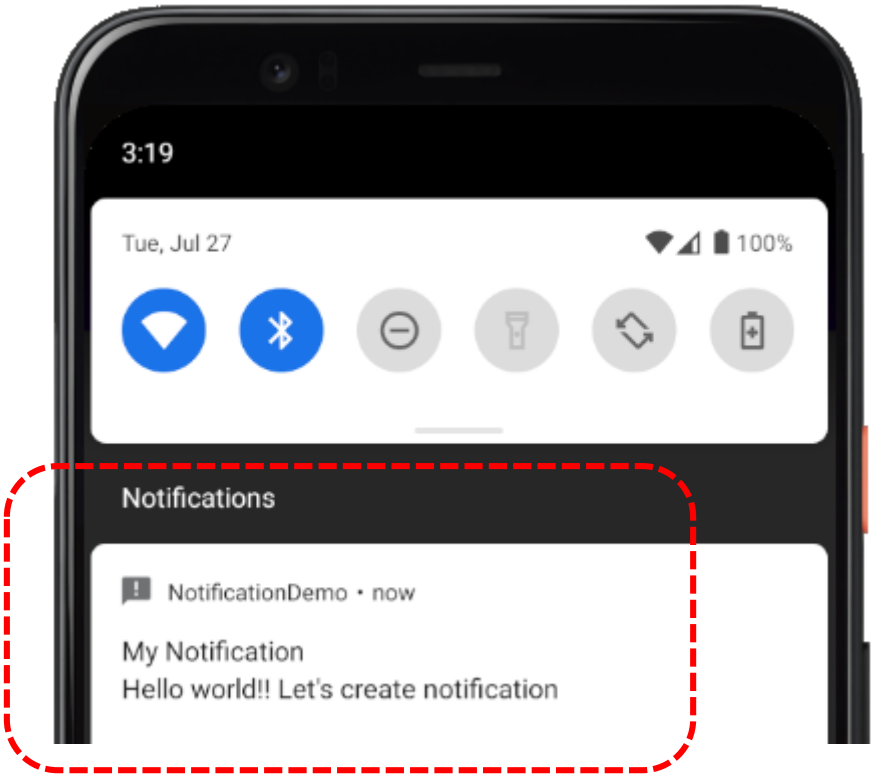


Figure 2-59: A notification with a title and text in the notification drawer

Design User Interface with View

In Android applications, various types of **ViewGroups** are used to design UI. The following are Basic Views in Android applications.

- TextView
- EditText
- Button
- ImageButton
- CheckBox
- ToggleButton
- RadioBtton
- RadioGroup

TUTORIAL: Create a Basic Views

Learning Outcomes:
By the end of this tutorial, you should be able to create basic views in android applications.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Implementation

1. Create a new Android project called **AppView**.

2. By default, it creates **activity_main.xml** file which contains a **TextView** element.

3. Design the layout - use **LinearLayout**.

```
<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"
    android:orientation="vertical"
    tools:context=".MainActivity">

</LinearLayout>
```

4. The **TextView** is used to display text/caption to the user. This is the most basic **View** and very frequently used in an application.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!" />
```

5. The next **View** is a subclass of **TextView** and it is **EditText**. This **View** allows the user to edit the text displayed.

```
<EditText
    android:id="@+id/txtUserName"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

6. **Button** represents a push-button widget.

```
<Button
    android:id="@+id/btnAdd"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="ADD" />
```

7. **ImageButton** is similar to **Button View** except that it displays an image with text.

```
<ImageButton
    android:id="@+id/imgButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    app:srcCompat="@android:drawable/btn_star_big_on" />
```

8. **CheckBox** is a type of button that has two states; i.e., checked or unchecked.

```
<CheckBox
    android:id="@+id/chkStudent"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Student" />

<CheckBox
    android:id="@+id/chkStaff"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Staff" />
```

9. **RadioGroup** and **RadioButton**, both have two states: either checked or unchecked. A **RadioGroup** is used to group together one or more **RadioButton Views**, thereby allowing only one **RadioButton** to be checked within the **RadioGroup**.

```
<RadioGroup
    android:id="@+id/rdGroup"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

<RadioButton
    android:id="@+id/rbMale"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Male" />

<RadioButton
    android:id="@+id/rbFemale"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Female" />
```

```
</RadioGroup>
```

10. **ToggleButton** displays checked/unchecked states using a light indicator.

```
<ToggleButton
    android:id="@+id/toggleButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

11. Now, run your program. You will get output of the basic views as shown below.

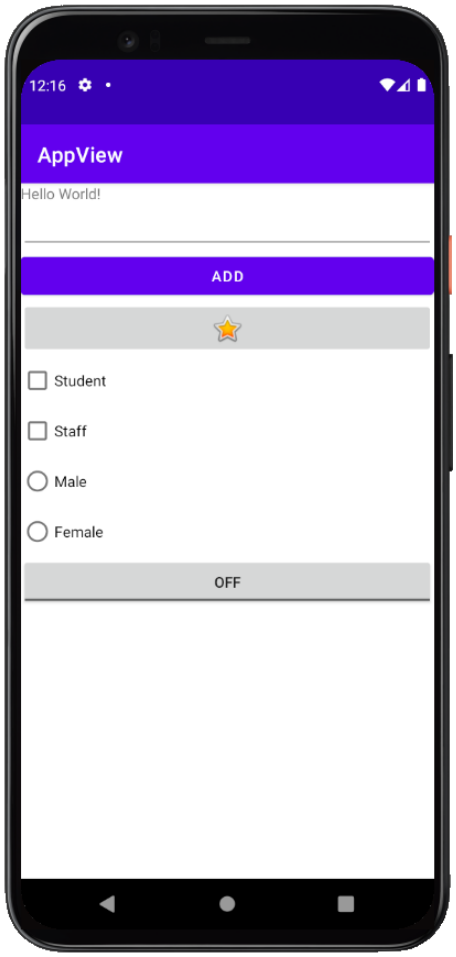


Figure 2-60: Basic views output

12. Add some java code in the **onCreate ()** method as below to handle **View** events for elements, like **Button** and **CheckBox**.

```
Button buttonAdd = (Button) findViewById(R.id.btnAdd);
buttonAdd.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view){
        DisplayMessage("You have clicked the Add button");
    }
});

CheckBox checkBox = (CheckBox) findViewById(R.id.chkStudent);
checkBox.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view){
        if (((CheckBox) view).isChecked())
            DisplayMessage("Student check box is checked");
        else
            DisplayMessage("Student check box is unchecked");
    }
});
```

13. Add a common method to display the text message as below.

```
private void DisplayMessage(String textMessage) {
    Toast.makeText(getBaseContext(), textMessage,
            Toast.LENGTH_SHORT).show();
}
```

14. Run your program again. Try clicked the **ADD Button** and checked the **Student Checkbox**. See the result of the implemented code.

Display Image and Menu with View

Simple ImageView

One type of **View** is an **ImageView**, which displays an image such as an icon or a photograph. An **ImageView** on the screen is drawn by a Java object inside the Android device. In fact, the Java object is the real **ImageView**. But when talking about what the user sees, it's convenient to refer to the rectangular area on the screen as the "**ImageView**".

Sample code:

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:srcCompat="@drawable/house" />
```

First, you need to put an image (house.jpg) into the **drawable** folder under the **res** folder in Android Studio.

Menu

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the **Menu** APIs to present user actions and other options in your activities.
Each menu must have an XML file related to it which defines its layout. These are the tags associated with the menu option:

i. **<menu>** - This is the container element for menu (similar to **LinearLayout**).
ii. **<item>** - This denotes an item and is nested inside of the menu tag. Be aware that an item element can hold a **<menu>** element to represent a submenu.

TUTORIAL: Create an Android Menu

Learning Outcomes:
By the end of this tutorial, you should be able to implement an options menu in any of your Android SDK applications.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Create a Resources Folder

1. Create a new Android project called **Android_Menu**.

2. To create a menu, you need a **menu folder**, so create one inside the "**res**" folder.
   - Right click on **res** in the project view in Android Studio and click "**New**" -> "**Android Resource Directory**". Change the resource type to "**menu**" in the dropdown menu and then click "**OK**".

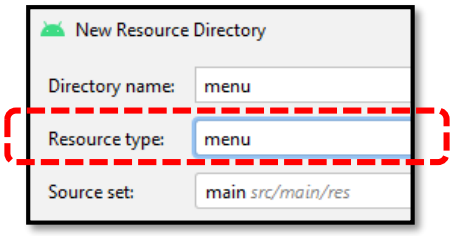


Figure 2-61: Create menu folder

3. You can see a new **Android Resource Directory** "**menu**" gets created.
   - Create **menu file** in menu folder, right click on "**menu**" directory -> "**New**" -> "**Menu resource file**".
   - Give the file name, **menu_main,** then click **Ok**.

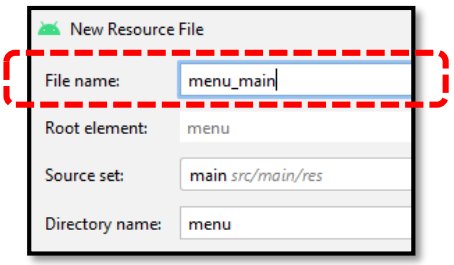


Figure 2-62: Create menu resource file

4. Now, you can see the directory structure, that shows the new file, **menu_main.xml** in **menu** directory.

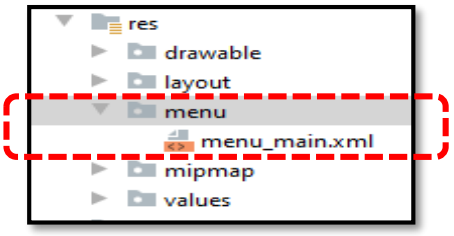


Figure 2-63: menu_main.xml

B. Create a Menu xml File

5. You can add one or more items to your options menu depending on the needs of your own project. Add an item for each menu option using the following syntax inside **menu_main.xml**.

```
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
    <item  android:id="@+id/about"
        android:title="About"/>
    <item  android:id="@+id/help"
        android:title="Help"/>
</menu>
```

C. Inflate your Menu resource

6. Add the following method to Java code, inside the class declaration and after the **onCreate()** method.

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    return true;
}
```

7. Run your program. You will get output of the android menu as shown below.

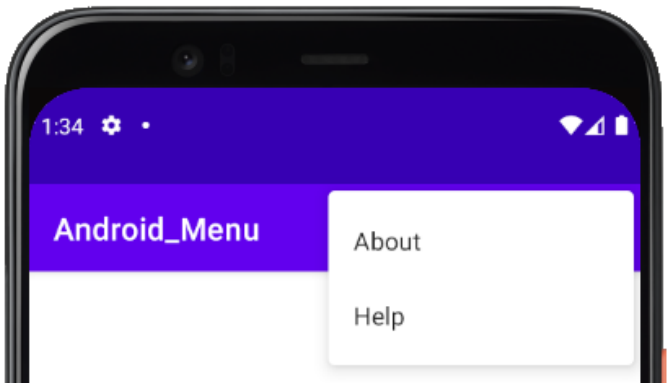


Figure 2-64: Android menu output

2.2 Navigating Between Activities

Link Activities Using Intents

Android applications can contain zero or more activity. When your app has more than one activity, you may need to navigate from one activity to another. In Android, you navigate between activities through what is known as **intent**. An **Intent** is a messaging object you can use to request an action from another app component.

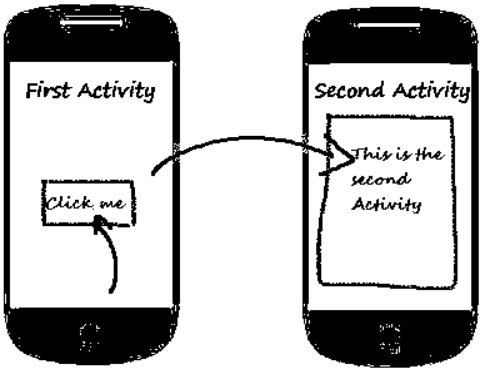


Figure 2-65: Intents

Below is a sample code when you click the button, the second activity will open.

Create a method that finds the **Button View** with the given ID

```
Button btnNext = (Button) findViewById(R.id.btnShow);

btnNext.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view) {
        Intent i = new Intent(getApplicationContext(),SecondActivity.class);
        startActivity(i);
    }
});
```

Set intent so that when user clicked **Button**, it will open the second activity

Set event for **Button**

Type of Android Intents

There are two (2) types of intents in Android:

1. **Implicit Intent**

   - It specifies the only action to be performed and does not directly specify Android Components. They are used for communication across two different applications.
   - The action generally specify that what is to be performed and optionally some data is required for that action. Data is usually expressed as a **URI (Uniform Resource Identifier)** that can be represent as an image in a gallery or a person in a contacts database for instance.
   - **Example:** When you tap the SHARE button in any app you can see the Gmail, Bluetooth, and other sharing app options. Here user sends a request (implicit intent) which can be handle by these Gmail, Bluetooth-like app.
   - Sample code:

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

2. **Explicit Intent**

   - It specifies for communication inside the application. Like changing activities inside the application. The component name is generally specified to which the intent has to be delivered.
   - **Example:** There are two activities (FirstActivity, SecondActivity). When you click on 'GO TO OTHER ACTIVITY' button in the first activity, then you move to second activity. When you click on 'GO TO HOME ACTIVITY' button in the second activity, then you move to the first activity.
   - Sample code:

```
Intent i = new Intent(getApplicationContext(),
        SecondActivity.class);
startActivity(i);
```

Passing Data using Intent Object

Through **Intent** we can move from one activity to another activity within the same application. **Intent** can also be used to pass the data from one activity to another activity.

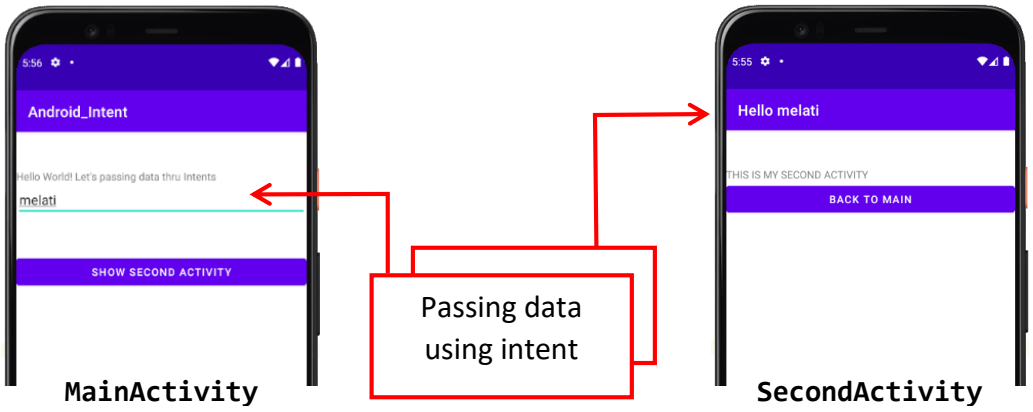


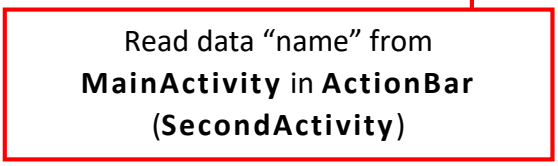
Figure 2-66: Passing data using intents output

Sample code:

```
public class MainActivity extends AppCompatActivity {

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

        //declare component
        Button btnNext = (Button) findViewById(R.id.btnShow);
        EditText txt_username = (EditText) findViewById(R.id.txtusername);

        //function for button Next
        btnNext.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view) {
                String name = txt_username.getText().toString();
                Intent i = new Intent(getApplicationContext(),
                        SecondActivity.class);
                i.putExtra("name", name);
                startActivity(i);
            }
        });
    }

}
```

Method **putExtra()** sends the data to next activity by passing key-value pair

```
public class SecondActivity extends AppCompatActivity {

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

        //declare component
        Button btnBack = (Button) findViewById(R.id.buttonBack);

        Intent i = getIntent();
        String uname = i.getStringExtra("name");
        getSupportActionBar().setTitle("Hello " + uname);

        //button back to MainActivity
        btnBack.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view) {
                Intent i = new Intent(getApplicationContext(),
                        MainActivity.class);
                startActivity(i);
            }
        });
    }
}
```



2.3 Utilize Telephony and SMS Services

Telephony

The Android SDK provides a number of useful utilities to integrate phone features available on the device with applications. The **telephony** system is a software framework to provide mobile phones with **telephony** functionalities, such as voice call, Video call, SMS, MMS ,data service, network management and so on. **Telephony** framework for Android has four (4) layered Architecture.

1. **Communication Processor**
   - It is an input / output processor for transmitting and collecting data from a number of remote terminals. It is a specialized processor designed to communicate with a data communication network

2. **Radio Interface Layer (RIL)**
   - This is the link between the hardware and services of the Android phone frame. This is the protocol stack for Phones.

3. **Android Telephony Services**
   - The Telephony Framework starts and is initiated along with the system. All queries by the Application API are addressed to RIL using this service.

4. **High Level Telephony Applications**
   - This is the UI of a phone-related Application such as Dialer, SMS, MMS, Call tracker, etc. The application is started with the android system boot. This is tied to the telephone frame service.

To uses **telephony** features, set the **<uses-feature>** tag with the **android.hardware.telephony** feature (or one of its sub-features) in manifest file. Adding **telephony** features to an application enables a more integrated user experience and enhances the overall value of the application to the users.

SMS Services

In android, we can send SMS from our android application in two (2) ways either by using **SMSManager** API or **Intents** based on our requirements. If we use **SMSManager** API, it will directly send SMS from our application. In case if we use **Intent** with proper action (**ACTION_VIEW**), it will invoke a built-in SMS app to send SMS from our application.

In android, to send SMS using **SMSManager** API we need to write the code like as shown below.

```
SmsManager smgr = SmsManager.getDefault();
smgr.sendTextMessage(MobileNumber,null,Message,null,null);
```

SMSManager API required **SEND_SMS** permission in our android manifest to send SMS. Following is the code snippet to set **SEND_SMS** permissions in manifest file.

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

TUTORIAL: Create an Android Send SMS

Learning Outcomes:
By the end of this tutorial, you should be able to implement an Android send SMS application.

Hardware/Software:
Computer, Android Studio and latest SDK version. **Android_Menu**.

Procedure:

A. Create an Android Send SMS

1. Create a new android application using android studio.

2. Create an **activity_main.xml** as shown below.

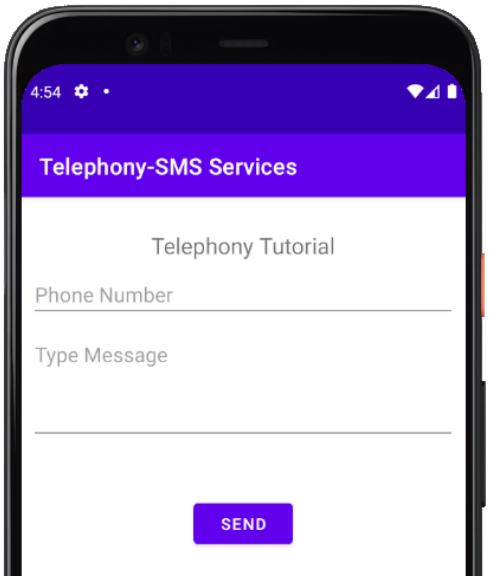


Figure 2-67: Android send SMS layout

3. Open an **MainActivity.java** and modify the **onCreate()** method like as shown below.

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

  //declare component
  txtPhoneNumber = (EditText) findViewById(R.id.editText_PhoneNumber);
  txtMessager = (EditText) findViewById(R.id.editText_Message);
  btnSend = (Button) findViewById(R.id.buttonSend);

  //function for button Send
```

```
  btnSend.setOnClickListener(new View.OnClickListener(){
    @Override
    public void onClick(View view) {
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M){
          if(checkSelfPermission(Manifest.permission.SEND_SMS) ==
                    PackageManager.PERMISSION_GRANTED){
              sendSMS();
          }else{
              requestPermissions(new String[]{Manifest.permission.SEND_SMS},1);
          }
        }
    }
  });
}
```

4. Add **sendSMS()** method after **onCreate()** method like as shown below to send SMS using **SMSManager** API.

```
private void sendSMS(){
    //get data input
    String phoneNO = txtPhoneNumber.getText().toString().trim();
    String SMS = txtMessager.getText().toString().trim();

    try{
        //use SmsManager to send SMS
        SmsManager smsMgr = SmsManager.getDefault();
        smsMgr.sendTextMessage(phoneNO, null, SMS, null, null);
        Toast.makeText(this, "Message Sent !", Toast.LENGTH_SHORT).show();
    }catch(Exception e){
        e.printStackTrace();
        Toast.makeText(this, "Message Failed To Sent !",
                Toast.LENGTH_SHORT).show();
    }
}
```

5. Following is the default content of **AndroidManifest.xml** to set **SEND_SMS** permissions in manifest file.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.telephonyevent">
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <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/Theme.TelephonyEvent">
        <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>
```

6. Let's try to run your application.
    - You can enter a desired mobile number and a text message to be sent on that number. Finally click on **Send** button to send your SMS.
    - Make sure your GSM/CDMA connection is working fine to deliver your SMS to its recipient.

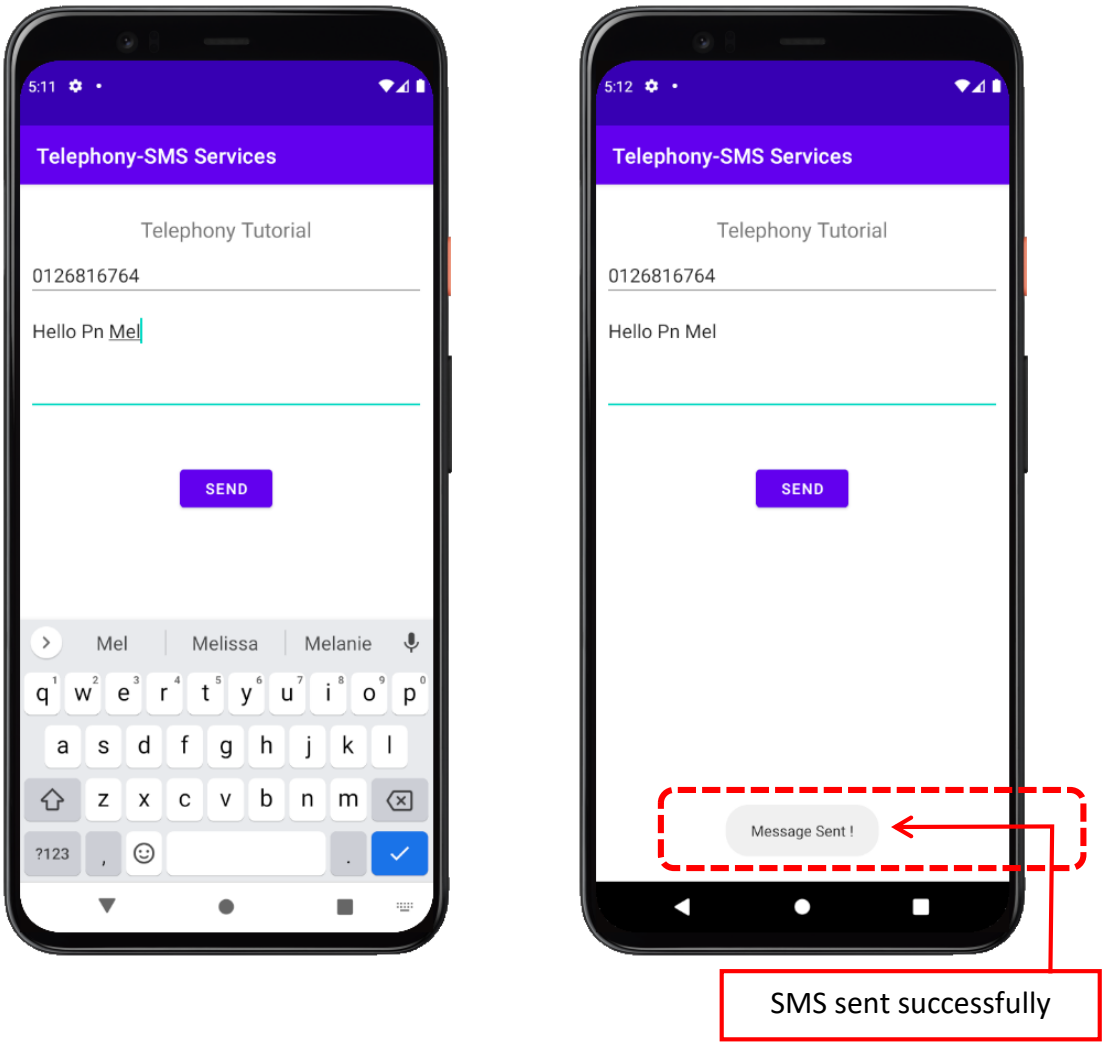


Figure 2-68: Android send SMS output

03

DATA PERSISTENCE AND MULTIMEDIA

3.1

Construct Persistence Data in Android

3.2

Apply Graphics and Animations

3.3

Apply Multimedia Components

3.1 Construct Persistent Data in Android

Various Data Persistence and Access Mechanism in a Mobile Application

Android provides several options for saving data persistent applications. The solution chosen will depend on specific needs, such as whether the data should lose their application or be accessible to other applications (and users) and how much space data takes up. Here are data persistence approaches:

1. **SharedPreferences:** store primitive private data on key-value pairs
2. **FlatFiles:** Save arbitrary files to internal or external device storage
3. **SQLite Databases:** store structured data in a private database

Each of these approaches provides relevant capabilities for different tasks in the application. **SharedPreferences** are often used for a limited set of data that represents users' preferences about how they want the application configured. They can also be used for other data that needs to survive throughout life cycle changes. **FlatFiles** are useful for backing up data and sending it to other users. Finally, **databases** are the workforce for data manipulation, storage, and retrieval. Developing an understanding of where, when, and how to use this data persistence approach is important to effective Android application development.

Implement Data Persistence and Access

A. SharedPreferences

SharedPreferences used for a limited set of data that represent user choices about the way they want the app configured. **SharedPreferences** allows user to store and retrieve key / values pairs of primitive data types. User can use **SharedPreferences** to store primitive data: booleans, floats, ints, longs, and strings.

Preferences are implemented through use of the **SharedPreferences** class. A **SharedPreferences** object can be used to store primitive data (e.g: integers and strings) in a key/value pair. Each value has its own key for storage and retrieval of that data. **SharedPreferences** are stored in private memory to the app and will persist as long as the app remains installed on the device. App upgrades will not impact the values stored with **SharedPreferences**.

There are two (2) main modes for accessing **SharedPreferences**.

1. **getSharedPreferences (String name, int mode)**
    - Used when there is more than one set of preferences for an app identified by name that will be passed in the first parameter or user want the preferences available to any **Activity** in the app.

```
getSharedPreferences("String preference name", int mode);
```

2. **getPreferences (int mode)**
    - If user need a set of preferences only for a single Activity.

```
getPreferences(int mode);
```

With each of these methods, user need to set an access mode. Using 0 (zero) makes the preferences private to the app. Data is stored by using a method appropriate to the value being saved (e.g: **putBoolean** or **putInt**) and supplying a string that will be the key for future access to that value. Likewise, to read **SharedPreferences** values use the methods as **getBoolean()** and **getInt()**.

Sample code: Example of persistence with **SharedPrefereces**

```
public static final String PREFS_NAME = "MyPrefsFile";
private boolean test;

private void store(){
    SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
    SharedPreferences.Editor editor = settings.edit();
    editor.putBoolean("test", test);
    // Commit editings
    editor.commit();
}
private void recover(){
    SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
    test = settings.getBoolean("test", false);
}
```

B. FlatFiles

Files are written and read as a stream of bytes. As to the Android system, a file is one thing. It does not have parts, such as different objects, within it. The advantage of files- data is stored efficiently and doesn't have to worry about what data is stored within the stream. Meanwhile, the disadvantage of files- depends to the developer to code the reading & writing of the file so that the data can be used appropriately when needed. Standard flat file input/ output, useful for backing up data and transmitting to other users.

Files can be written to either **internal** (private to the app, will persist as long as the app is installed on the device) or **external** storage (such as an SD card, accessible (including being able to modify and delete). Files are written and read from storage using the **FileInputStream** and **FileOutputStream.**

1. **FileInputStream**
    - Example of simple persistence with **FileInputStream**:

```
FileInputStream fin = openFileInput("mytextfile.txt");

int c;
String temp="";
while( (c = fin.read()) != -1){
    temp = temp + Character.toString((char)c);
}
fin.close();
```

- The method **openFileInput()** is used to open a file and read it. It returns an instance of **FileInputStream**. After that, method read one character at a time from the file and then print it.

2. **FileOutputStream**
   - Example of simple persistence with **FileOutputStream**:

```
String FILENAME = "hello_file";
String string = "hello world!";

FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

- The method **openFileOutput()** is used to create and save a file, and returns an instance of FileOutputStream:

C. SQLite Databases

A database is very useful for any large or small system, unless system deals only with simple data, without using a bank to store information. The Android uses the **SQLite** database that is open-source and widely used in popular applications.

SQLite provides capabilities for retrieval and manipulation of the stored data through the use of queries written in **Structured Query Language (SQL)**. Data stored in a **SQLite** database is private to the app and will persist as long as the app is installed on the device. An app may create and use multiple databases, and each database can have many tables, making data storage via **SQLite** both extensive and flexible.

Construct and Leverage Relational Database on Devices Using SQLite

1. **Use constants for table names and database creation query**
    - Define constants for database and table names

```
private static final String DATABASE_NAME = "my_sqlite_db";
private static final String TABLE_NAME = "student";
```

2. **Creating a SQLite Database Instance Using the Application Context**
    - use the **openOrCreateDatabase**() method.

```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
. . .
. . .
SQLiteDatabase mDB = openOrCreateDatabase (DATABASE_NAME,
        Context.MODE_PRIVATE,null);
```

3. **Configuring the SQLite Database Properties**
    - Some important database configuration options include version and locale features.

```
import java.util.Locale;
. . .
mdb.setLocale(Locale.getDefault());
mdb.setVersion(1);
```

4. **Creating Tables and Other SQLite Schema Objects**

```
mdb.execSQL("CREATE TABLE IF NOT EXISTS " + TABLE_NAME
         + " (rollno VACHAR, name VARCHAR, marks VARCHAR);");
```

5. **Creating, Updating and Deleting Database Records**

    - Insert Records – used to insert a new row in the database

```
mdb.execSQL("INSERT INTO " + TABLE_NAME + " VALUES ('"
        + Rollno.getText() + "', '"
        + Name.getText() + "', '"
        + Marks.getText() + "');");
```

- Update Records - used to update the fields of an existing row

```
mdb.execSQL("UPDATE " + TABLE_NAME
        + " SET name = '" + Name.getText()
        + "', marks = '" + Marks.getText()
        + "' WHERE rollno = '" + Rollno.getText() + "'");
```

- Delete Records - used to delete the existing rows

```
mdb.execSQL("DELETE FROM student WHERE rollno = '"
        + Rollno.getText() + "'");
```

6. **Closing a SQLite Database**

```
db.close();
```

Sharing Data in Android Using Content Provider

The **Content Providers** are a very important component that serves for the purpose of a relational database to store application data. The role of the **Content Provider** in the android system is like a central repository where application data is stored, and this makes it facilitates other applications to access and modify data securely based on user needs. The Android system allows **Content Provider** to store application data in several ways.

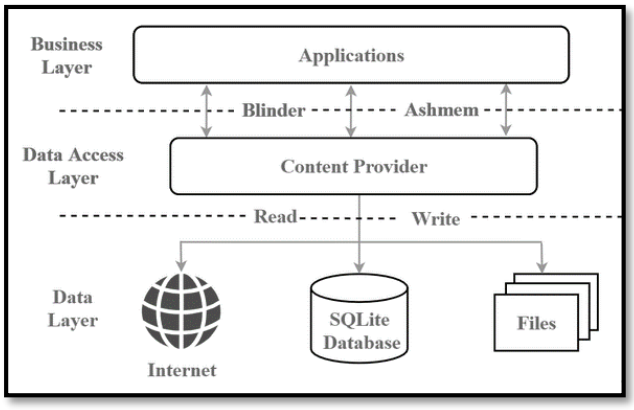


Figure 3-69: Content Provider

Users can manage to store the application data like images, audios, videos, and personal contact information by storing it in a **SQLite Database**, in a file, or even on a network. In oder to share the data, **Content Providers** have certain permissions that are used to grant or restrict other applications' rights to interfere with the data.

Content URI

Content URI (Uniform Resource Identifier) is the key concept of Content Providers. To access the data from a content provider, URI is used as a query string. The **Content URI** is essentially the address of where to find the data within the provider. A content URI always starts with **content://** and then includes the **authority** of a provider which is the provider's symbolic name.

Structure of a **Content URI** consists of four (4) parts:

content://authority/path/ID

- **content://** all the content provider URIs should start with this value
- **authority** represents the domain, and for content providers customarily ends in **.provider**
- **path** is the path to the data
- **ID** uniquely identifies the data set to search

Operations in Content Provider

Each android application can be a **Content Provider**. When **Content Provider** is accessed, the **ContentResolver** object will be used in the application context. The **ContentResolver** communicates with the provider, an instance of the class that implements **Content Provider**. The **ContentResolver** object receives data request from the client and perform the request action on behalf of the client and deliver the results back to the client.

This **ContentResolver** object has methods - **insert()**, **update()**, **query()** and **delete()** that call identically-named methods in the provider object, an instance of one of the concrete subclasses of **Content Provider.** The methods provide the basic "**CRUD**" (namely **Create**, **Read**, **Update**, and **Delete**) functions of persistent storage.

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

Example: Android Content Provider, Content URI and ContentResolver

Android **Content Provider** is mainly used for data sharing between different applications. It provides a complete set of mechanisms to allow one program to access data in another program, and also to ensure the security of the data being accessed. **Content URI** is a unique resource identifier that **Content Provider** app provides for client app to access it's shared data.

To get data from a **Content Provider**, a **ContentResolver** needs to be used in the application. Then the **ContentResolver**'s method can be used to insert, delete, update and query data shared by other content providers. This is something like SQLite database operation.

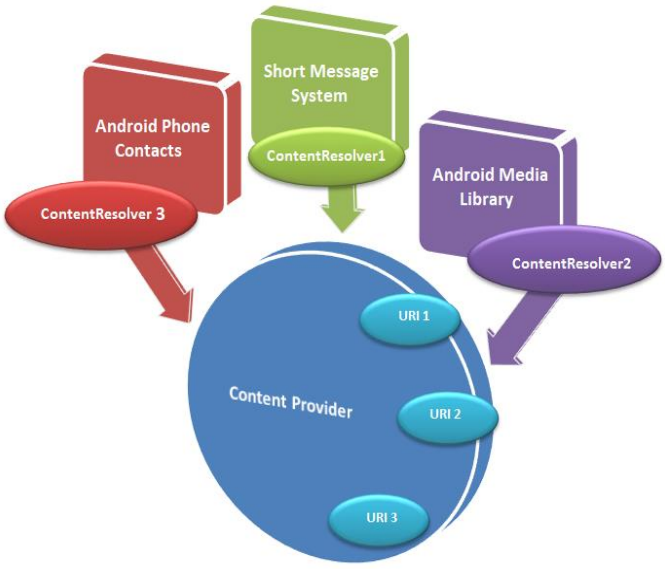


Figure 3-70: Android content provider

3.2 Apply Graphics and Animation Android

Graphics and animations help make Android apps interesting and fun to use; however, it is important to remember that some interactions occur through screen readers, alternative input devices, or with assisted zoom. Also, some interactions may occur without audio capabilities.

Applications are more useful in these situations if they are designed with accessibility in mind: providing hints and navigational assistance in the user interface, and ensuring there is text or description content for the UI pictorial elements.

Multiple Screens Density and Size

For applications, the Android system provides a consistent development environment across devices and handles most of the work to adjust each application's user interface to the screen on which it is displayed. At the same time, the system provides APIs that can control application's UI for specific screen sizes and densities.

A. Terms and Concepts

1. **Screen Size**
   Actual physical size, measured as the screen's diagonal. For simplicity, Android groups all actual screen sizes into four generalized sizes: **small**, **normal**, **large**, and **extra-large**.

2. **Screen Density**
   The quantity of pixels within a physical area of the screen; usually referred to as **dpi** (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.

For simplicity, Android groups all actual screen densities into six generalized densities: **low**, **medium**, **high**, **extra-high**, **extra-extra-high**, and **extra-extra-extra-high**.

3. **Orientation**

   The orientation of the screen from the user's point of view. This is either **landscape** or **portrait**, meaning that the screen's aspect ratio is either wide or tall, respectively. Be aware that not only do different devices operate in different orientations by default, but the orientation can change at runtime when the user rotates the device.

4. **Resolution**

   The total number of **physical pixels** on a screen. When adding support for multiple screens, applications do not work directly with resolution; applications should be concerned only with screen size and density, as specified by the generalized size and density groups.

5. **Density-independent pixel (dp)**

   A **virtual pixel** unit that user should use when defining UI layout, to express layout dimensions or position in a density-independent way. The density-independent pixel is equivalent to one physical pixel on a 160 dpi screen, which is the baseline density assumed by the system for a "medium" density screen. At runtime, the system transparently handles any scaling of the dp units, as necessary, based on the actual density of the screen in use. The conversion of dp units to screen pixels is simple: px= dp* (dpi / 160).

   For example, on a 240 dpi screen, 1 dp equals 1.5 physical pixels. You should always use dp units when defining your application's UI, to ensure proper display of your UI on screens with different densities.

B. Range of Screens Supported

Starting with Android 1.6 (API Level 4), Android provides support for multiple screen sizes and densities, reflecting the many different screen configurations that a device may have. To make it easier to design a user interface for multiple screens, Android divides the actual screen size and density range into:

1. A set of four (4) generalized sizes:

   small, ***normal***, ***large***, *and* ***xlarge***

2. A set of six (6) generalized densities:

   ldpi (low) ~120dpi, ***mdpi (medium) ~160dpi***, ***hdpi high) ~240dpi***, ***xhdpi (extra-high) ~320dpi***, ***xxhdpi (extra-extra-high) ~480dpi*** *and* ***xxxhdpi (extra-extra-extra-high) ~640dpi***

C. How to Support Multiple Screen

1. Explicitly declare in the manifest which screen sizes your application supports.
2. Provide different layouts for different screen sizes.
3. Provide different bitmap drawablesfor different screen densities

D. The Best Practice to Ensure Compatibility Screen Display

1. Use **wrap_content**, **fill_parent** or **dp units** when specifying dimensions in an XML layout file
2. Do not use hard coded pixel values in your application code
3. Do not use AbsoluteLayout (it's deprecated)
4. Supply alternative bitmap drawables for different screen densities

Animation Types and Capabilities

Animation is the process of creating motion and changing shape of a specific view. Animation in android can be done in various ways. Animation in Android is generally used to give the UI a rich look and feel.

Animation basically consists of three (3) types as follows::

A. Property Animation

Introduced in Android 3.0 (API level 11), an extensible and flexible system that can be used to animate the properties of any object, not just **View** objects. This flexibility allows animations to be encapsulated in distinct classes that will make code sharing easier. **Property Animation** can be used to add any animation in the CheckBox, RadioButtons, and widgets other than any view.

The **android.animation** provides classes which handle property animation. The **Property Animation** system lets user define the following characteristics of an animation:

- **Duration**: User can specify the duration of an animation. The default length is 300 ms.
- **Time interpolation**: User can specify how property values are calculated as a function of the elapsed animation time.
- **Repeat count and behavior**: User can specify whether or not to have an animation repeat when reaching the end of the period and how many times to repeat the animation. User can also specify whether user 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**: User can group animations into logical sets that play together or sequentially or after specified delays.
- **Frame refresh delay**: User can specify how often to refresh frames of animation. The default is set to refresh every 10 ms, but the speed in which application can refresh frames is ultimately dependent on how busy the system is overall and how fast the system can service the underlying timer.

B. View Animation

View Animation is an original animation API's in Android, also called as **Tween Animation** and available in all versions of Android. This API is limited in that it will only work with **View** objects and can only perform simple transformations on those Views. **View animations** are typically defined in XML files found in the **/Resources/anim** folder. The **android.view.animation** provides classes which handle view animation.

An example of **View Animation** can be used, such as : if we have a **TextView** object, we can move, rotate, grow, or shrink the text. If it has a background image, the background image will be transformed along with the text.

C. Drawable Animation

Drawable Animation is the simplest animation API, used if user want to animate one image over another. The simple way to understand is to animate drawable is to load a series of drawable one after another to create an animation. A simple example of **Drawable Animation** can be seen in many apps Splash screen on apps logo animation.

The important methods of Animation

- **startAnimation()** - This method will start the animation
- **clearAnimation()** - This method will clear the animation running on a specific view

TUTORIAL: Graphics and Animation Capabilities to an Application

Learning Outcomes:
By the end of this tutorial, you should be able to add animations to ImageView.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Create New Project

1. Open Android Studio and create new project. Name it as **Android_Animation**. Click **Finish**.

B. Working with the activity_main.xml

2. Create **ImageView** along with **Buttons** that will add animation to the view as shown below.

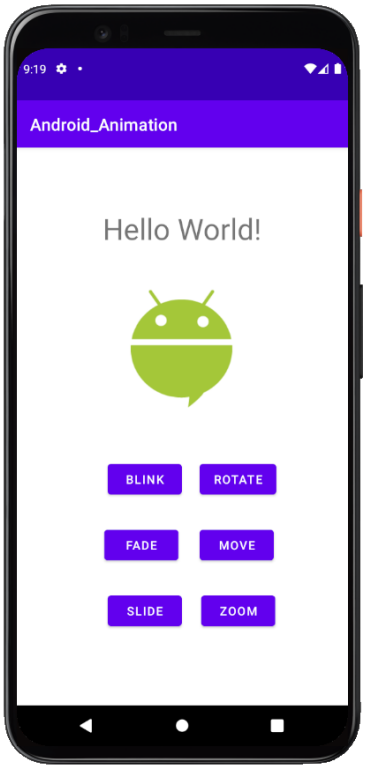


Figure 3-71: Animation interface

C. Create 6 different types of animation for ImageView

3. To create new animations, create a new directory for storing all animations. Navigate to the **app > res > .** Right-click on **res >> New >> Directory >>** Name directory as **"anim"**.

4. Inside this directory, create animations. For creating a new **anim** right click on the **anim directory >> Animation Resource file** and give the name to file.

5. Below is the code snippet for six (6) different animations.

blink.xml

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

    <alpha android:fromAlpha="0.0"
        android:toAlpha="1.0"
        android:interpolator="@android:anim/accelerate_interpolator"
        android:duration="500"
        android:repeatMode="reverse"
        android:repeatCount="infinite"/>

</set>
```

rotate.xml

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

    <rotate
        android:duration="6000"
        android:fromDegrees="0"
        android:pivotX="50%"
        android:pivotY="50%"
        android:toDegrees="360" />

    <rotate
        android:duration="6000"
        android:fromDegrees="360"
        android:pivotX="50%"
        android:pivotY="50%"
        android:startOffset="5000"
        android:toDegrees="0" />

</set>
```

fade.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator" >

    <alpha
```

```
        android:duration="1000"
        android:fromAlpha="0"
        android:toAlpha="1" />

    <alpha
        android:duration="1000"
        android:fromAlpha="1"
        android:startOffset="2000"
        android:toAlpha="0" />

</set>
```

move.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/linear_interpolator"
    android:fillAfter="true" >

    <translate
        android:fromXDelta="0%p"
        android:toXDelta="75%p"
        android:duration="700" />

</set>
```

slide.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true" >

    <scale
        android:duration="500"
        android:fromXScale="1.0"
        android:fromYScale="1.0"
        android:interpolator="@android:anim/linear_interpolator"
        android:toXScale="1.0"
        android:toYScale="0.0" />

</set>
```

zoom.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true" >

<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromXScale="0.5"
    android:toXScale="3.0"
    android:fromYScale="0.5"
    android:toYScale="3.0"
    android:duration="5000"
    android:pivotX="50%"
    android:pivotY="50%" >
</scale>
```

```
<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:startOffset="5000"
    android:fromXScale="3.0"
    android:toXScale="0.5"
    android:fromYScale="3.0"
    android:toYScale="0.5"
    android:duration="5000"
    android:pivotX="50%"
    android:pivotY="50%" >
</scale>


</set>
```

D. Working with the MainActivity.java file

6. Add animation to the **ImageView** by clicking a specific **Button**.

```
public class MainActivity extends AppCompatActivity {

    ImageView imageView;
    Button blinkBTN, rotateBTN, fadeBTN, moveBTN, slideBTN, zoomBTN;

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

        imageView = findViewById(R.id.imgView);
        blinkBTN = findViewById(R.id.btnBlink);
        rotateBTN = findViewById(R.id.btnRotate);
        fadeBTN = findViewById(R.id.btnFade);
        moveBTN = findViewById(R.id.btnMove);
        slideBTN = findViewById(R.id.btnSlide);
        zoomBTN = findViewById(R.id.btnZoom);

        blinkBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add blink animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.blink);
                imageView.startAnimation(animation);
            }
        });

        rotateBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add rotate animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.rotate);
```

```
                imageView.startAnimation(animation);
            }
        });

        fadeBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add fade animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.fade);
                imageView.startAnimation(animation);
            }
        });

        moveBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add move animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.move);
                imageView.startAnimation(animation);
            }
        });

        slideBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add slide animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.slide);
                imageView.startAnimation(animation);
            }
        });

        zoomBTN.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // To add zoom animation
                Animation animation =
                        AnimationUtils.loadAnimation(getApplicationContext(),
                        R.anim.zoom);
                imageView.startAnimation(animation);
            }
        });

    }
}
```

E. Output

7. Now, run your program. You will see animation output on the emulator or device.

3.3 Apply Multimedia Component

Many modern “smart devices” have built-in cameras to capture and display still images, video, and sophisticated music playback abilities. Basic smartphone has at least one camera, sometimes two for the front-facing cameras used for video chat and self-portraits (selfies).

As an application developer, we are free to make use of any media codec that is available on any Android-powered device, including those provided by the Android platform and those that are device-specific. However, it is a best practice to use media encoding profiles that are device-agnostic.

A. Audio Capture

The Android multimedia framework includes support for capturing and encoding a variety of common audio formats, so that we can easily integrate audio into applications. We can record audio using the **MediaRecorder** APIs if supported by the device hardware.

B. Camera

The Android framework supports capturing images and video through the **android.hardware.camera2** API or camera **Intent**.

Media Container and Codecs

The table below describes the media format support built into the Android platform. Note that any given mobile device may provide support for additional formats or file types not listed in the table.

Type	Format / Codec	Supported File Type(s) / Container Formats
Audio	AAC LC	• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a) • ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported) • MPEG-TS (.ts, not seekable, Android 3.0+)
	HE-AACv1 (AAC+)	
	HE-AACv2 (enhanced AAC+)	
	AAC ELD (enhanced low delay AAC)	
	AMR-NB	3GPP (.3gp)
	AMR-WB	3GPP (.3gp)
	FLAC	FLAC (.flac) only
	MP3	MP3 (.mp3)
	MIDI	• Type 0 and 1 (.mid, .xmf, .mxmf) • RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy)
	Vorbis	• Ogg (.ogg) • Matroska (.mkv, Android 4.0+)
	PCM/WAVE	WAVE (.wav)
Image	JPEG	JPEG (.jpg)
	GIF	GIF (.gif)
	PNG	PNG (.png)
	BMP	BMP (.bmp)
	WebP	WebP (.webp)
Video	H.263	• 3GPP (.3gp) • MPEG-4 (.mp4)
	H.264 AVC	• 3GPP (.3gp) • MPEG-4 (.mp4) • MPEG-TS (.ts, AAC audio only, not seekable, Android 3.0+)
	MPEG-4 SP	3GPP (.3gp)
	VP8	• WebM (.webm) • Matroska (.mkv, Android 4.0+)

Table 3-8: Media container and codecs

TUTORIAL: Implement Media

Learning Outcomes:
By the end of this tutorial, you should be able to implement VideoView.

Hardware/Software:
Computer, Android Studio and latest SDK version.

Procedure:

A. Create New Project

1. Open Android Studio and create new project. Name it as **Android_Media**. Click **Finish**.

B. Working with the activity_main.xml

2. Create **VideoView** along with **Button** that will play video to the view as shown below.

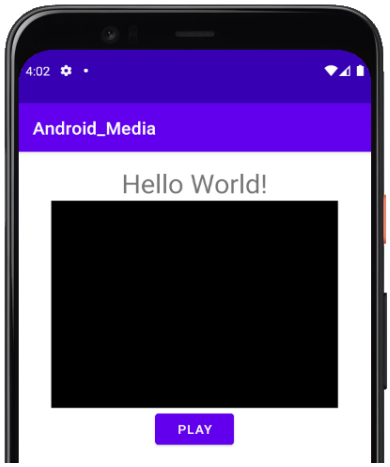


Figure 3-72: Media interface

C. Create raw folder

3. To create new media, create a new directory for storing media. Navigate to the **app > res > .** Right-click on **res >> New >> Folder >> Res Folder >> Configure Component.**
4. Tick checkbox "**Change Folder Location**". Name **New Folder Location** as **"src/main/res/raw"**. Drag mp4 video into **raw** folder.

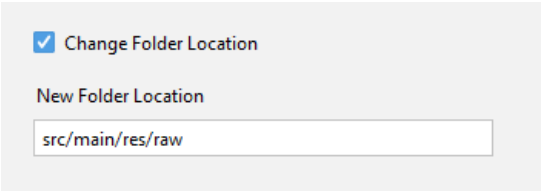


Figure 3-73: Create raw folder

D. Working with the MainActivity.java file

5. Add animation to the **videoview** by clicking a specific **Button**.

```
package com.example.android_media;

import androidx.appcompat.app.AppCompatActivity;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.MediaController;
import android.widget.VideoView;

public class MainActivity extends AppCompatActivity {

    VideoView videoview;
    Button btnPLAY;
    MediaController mediac;

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

        // initiate components
        btnPLAY = (Button) findViewById(R.id.btnPlay);
        videoview = (VideoView) findViewById(R.id.videoView);
        mediac = new MediaController(this);
    }

    public void videoplay(View view){
        String VideoURL = "android.resource://com.example.android_media/"
                + R.raw.tech;
        Uri uri = Uri.parse(VideoURL);
        videoview.setVideoURI(uri);
        videoview.setMediaController(mediac);
        mediac.setAnchorView(videoview);
        videoview.start();
    }
}
```

E. Output

6. Now, run your program. You will see media output on the emulator or device.

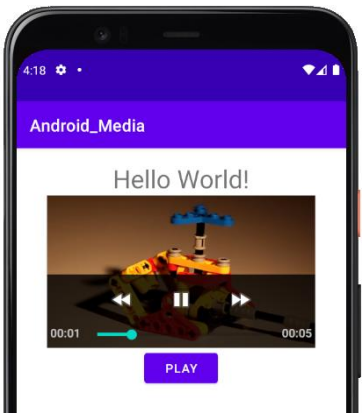


Figure 3-74: Media output

04

PUBLISHING ANDROID APPLICATION

4.1

Test Android Apllication Components

4.2

Publish Application

4.1 Test Android Application Components

Fundamentals of Testing

Users interact with developed mobile app on a variety of levels, from pressing a button to downloading information to their devices. Therefore, as a mobile developer should test a variety of use cases and interactions as iteratively developing app.

As mobile app expands, there are a number of activity features that need to be considered such as fetching data from a server, interacting with the device's sensors, accessing local storage, or rendering complex user interfaces. The versatility of mobile app demands a comprehensive testing strategy.

View app as a series of modules

To make code easier to test, develop code in terms of modules, where each module represents a specific task that users complete within app. This perspective differs the stack-based view of an mobile app that typically contains layers representing the UI, business logic, and data.

It's important to set well-defined boundaries around each module, and to create new modules as app grows in scale and complexity. Each module should have only one area of focus, and the APIs that allow communication between modules must be consistent. To make it easier and faster to test the interactions between these modules, consider creating a fake module implementation. In testing, the real implementation of one module can call the fake implementation of the other module.

However, create a new module, don't be too dogmatic to make it complete right away. It doesn't matter if a particular module doesn't have one or more application stack layers.

Running test on different types of devices

When running a test on a device, you can choose between the following types:

1. **Real device**

   Real devices offer the highest fidelity but also require the most time to run tests.

2. **Virtual device (such as the emulator in Android Studio)**

   Virtual devices offer a balance of fidelity and speed. When using a virtual devices for testing, use snapshots to minimize setup time in between tests.

3. **Simulated device (such as Roboletric)**

   Simulated devices, on the other hand, provide better test speeds with lower fidelity costs. However, platform improvements in binary sources and realistic compilers allow simulation devices to produce more realistic results.

Write tests

Once the test environment is configured, it is time to write a test that evaluates the functionality of the mobile application. This section describes how to write small, medium and large tests.

1. **Small Test**

   Small tests are unit tests that validate mobile app's behavior one class at a time.

2. **Medium Test**

   Medium tests are integration tests that validate either interactions between levels of the stack within a module, or interactions between related modules.

3. **Large Tests**

   Large tests are end-to-end tests that validate journeys spanning multiple modules of mobile app.

While working on the pyramid, from small tests to large tests, each test increases in fidelity but also increases in execution time and effort to maintain and debug. Therefore, should write more unit tests than integration tests, and more integration tests than end-to-end tests. Although the test portion for each category can vary based on the application use case, it generally recommends the following division between categories: 70 percent small, 20 percent medium, and 10 percent large.

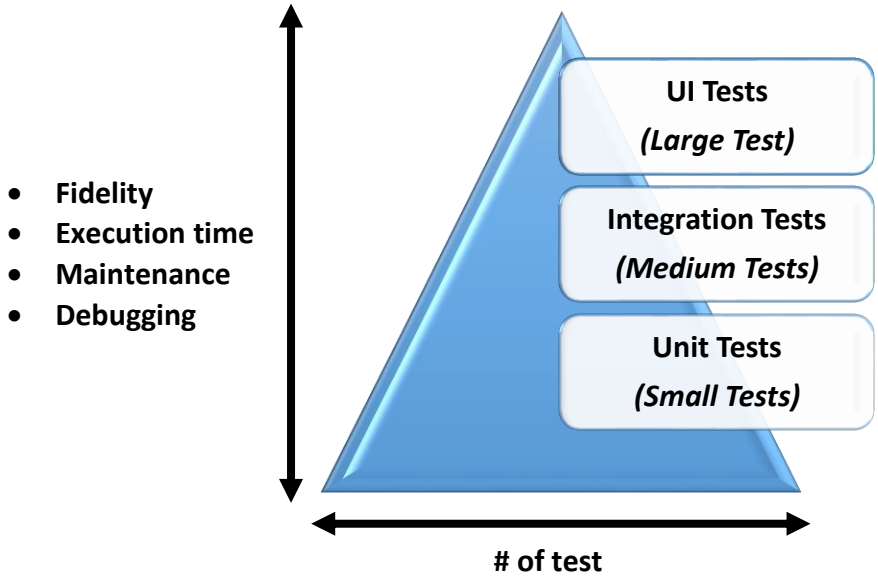


Figure 4-75: The Testing Pyramid, showing the three categories of tests that should include in mobile app's test suite

The **small test** written must be a highly focused unit test that thoroughly validates the functionality and contracts of each class within mobile app. In addition to testing each mobile application unit by running small tests, it must validate app's behavior from the module level. To do so, write a **medium test**, which is an integration test that validates the collaboration and interaction of a group of units.

While it is important to test each class and module in a mobile application separately, it is equally important to validate an end-to-end workflow that guides users through multiple modules and features. This type of testing form unavoidable difficulties in code, but can minimize this effect by validating an app that's as close to the actual, finished product as possible.

If the mobile app is small enough, it may only require only one suite of **large tests** to evaluate the app's functionality as a whole. Otherwise, should divide the large test suite by team ownership, functional vertical, or user goals. Typically, it is better to test an app on an emulated device or a cloud-based service like Firebase Test Lab, rather than on a physical device, as it can test multiple combinations of screen sizes and hardware configurations more easily and quickly.

Android JUnit Framework for Unit Testing

Install the dependencies

To use **JUnit** tests for Android application, add a dependency to Gradle build file.

```
dependencies {

    // Unit testing dependencies
    testImplementation 'junit:junit:4.+'

}
```

JUnit is a Unit Testing Framework for Java Applications. It is an automation framework for Unit as well as UI Testing. It contains annotations such as @Test, @Before, @After etc. Unit tests are generally written before writing the actual application.

Unit Testing is done to ensure that developer would be unable to write low quality / incorrect code. It makes sense to write a Unit Test before writing the actual app as there will be no bias towards test success, write the test first and the actual code should adhere to the design guidelines laid out by the test.

JUnit methods

There are several methods provided by **JUnit Framework**.

1. **assertThat ()** : create custom assertions and not just true and false values. It takes in 3 arguments. A reason/description, input value to be checked, expected actual value.

2. **is ()** : returns a Matcher to match the source object to the one provided as the parameter of method **is()**.

3. **equalTo ()** : checks for equality between the expected and actual value.

4. **when ()** : a very powerful method which takes in a method call as its parameter. It takes in the method call which is to be stubbed/duplicated. Once the method stub is executed, method **then()** is called.

5. **thenReturn ()** : called after the method stub provided in when() method has finished running. It is used to return the result of the method, if it is not void.

Android Unit Test Tutorial

1. Whenever an android project is developed, there are three **java packages** visible.

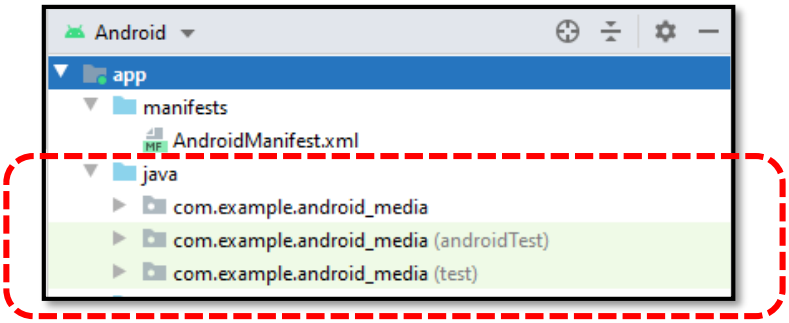


Figure 4-76: Java pakages

There is a main package (the first one), and inside this package there is all application's code. Next, there are two more packages that can be differentiate these packages with shown hint **androidTest** and **test**.

2. Now go to app-level **build.gradle** file and see the dependencies block.

```
dependencies {

    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'com.google.android.material:material:1.4.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'

    // Unit testing dependencies
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'

}
```

Figure 4-77: build.gradle

You can see there are **testImplementation** and **androidTestImplementation. testImplementation** is a library available inside test package. And the same way if you want the library to be available inside android Test package, you need to use **androidTestImplementation**. Now, **JUnit Testing Framework** is added by default here. Each time a project is developed, it will be added by default. **JUnit** will be used for unit test code.

4.2 Publish Application

Prepare an application for publishing

Publishing is the general process that makes Android applications available to users. Various distribution opportunities are available for Android app developers. Many developers choose to sell their apps through mobile marketplaces such as Google Play. Others develop their own distribution mechanisms, for example, they may sell their apps from websites.

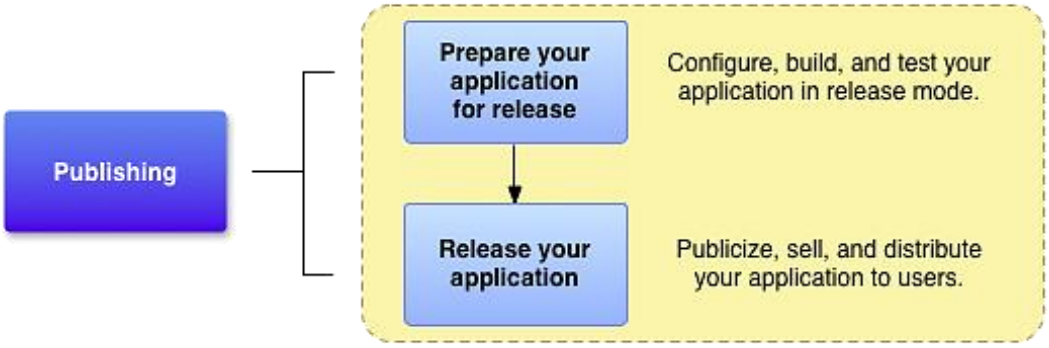


Figure 4-78: Publishing application

There are two (2) main tasks of publishing an Android application:

1. **Prepare the application for release**
   During the preparation step, build a release version of application, which users can download and install on their Android-powered devices.

2. **Release the application to users**
   During the release step, publicize, sell, and distribute the release version of application to users.

04 Publishing Android Application

Preparing an application for publish is a multi-step process that involves the following tasks :

1. **Configure the application to be released**
   Remove call **Logs** and **android:debuggable** attributes from the manifest file. Provide values for the android **attributes:versionCode** and **android:versionName**, located in the **<manifest>** element. Configure some other settings to meet Google Play requirements or accommodate whatever method is used to release the app. Can also use the release build type to set build settings for the published version of the application.

2. **Building and signing a release version of application**
   Use the Gradle build file with the release build type to build and sign the release version of the application.

3. **Testing the release version of application**
   Before distributing the app, test the release version thoroughly on at least one target mobile device and one target tablet device.

4. **Updating application resources for release**
   Ensure that all application resources such as multimedia and graphics files are updated and included with the application or staged on the correct production server.

5. **Preparing remote servers and services that application depends on**
   If the application depends on an external server or service, make sure it is secure and ready for production.

Configure the application version and API requirements

During the publishing preparation steps, developer build a release version of the app, which users can download and install on their Android-powered devices. Versions are an important component of application improvement and maintenance strategies. Version is important because:

1. Users need to have specific information about the version of the application installed on their device and the upgraded version available for installation.

2. Other apps - including other applications that published together as a suite, need to query the system for app's version, to determine compatibility and identify dependencies.

3. The service that will be used to publish the application may also need to request the type of application for that version, so that they can display that version to the user. Publishing services may also need to check application versions to determine compatibility and establish an upgrade / drop relationship.

The Android system uses app version information to protect against downgrades. The Android system enforces system version compatibility as specified by the **minSdkVersion** setting in the build file. This setting allows the application to specify the appropriate minimum system API.

Set application version information

To define the version information for app, set values for the version settings in the Gradle build files. These values are then merged into app's manifest file during the build process. Two settings are available, and you should always define values for both of them: **versionCode** and **versionName**.

Define default values for these settings by including them in the **defaultConfig{}** block, nested inside the **android{}** block of module's **build.gradle** file. Override these default values for different versions of app by defining separate values for individual build types or product flavors.

Set API level requirements

If app requires a specific minimum version of the Android platform, specify that version requirement as API level settings in the app's **build.gradle** file. During the build process, these settings are merged into app's manifest file. Specifying API level requirements ensures that app can only be installed on devices that are running a compatible version of the Android platform.

Package, sign and optimize the application

Android requires that all APKs be digitally signed with a certificate before they are installed on a device or updated. Following is the steps need to sign and publish a new app to Google Play:

1. Generate an upload key and keystore

You can generate one using Android Studio as follows:

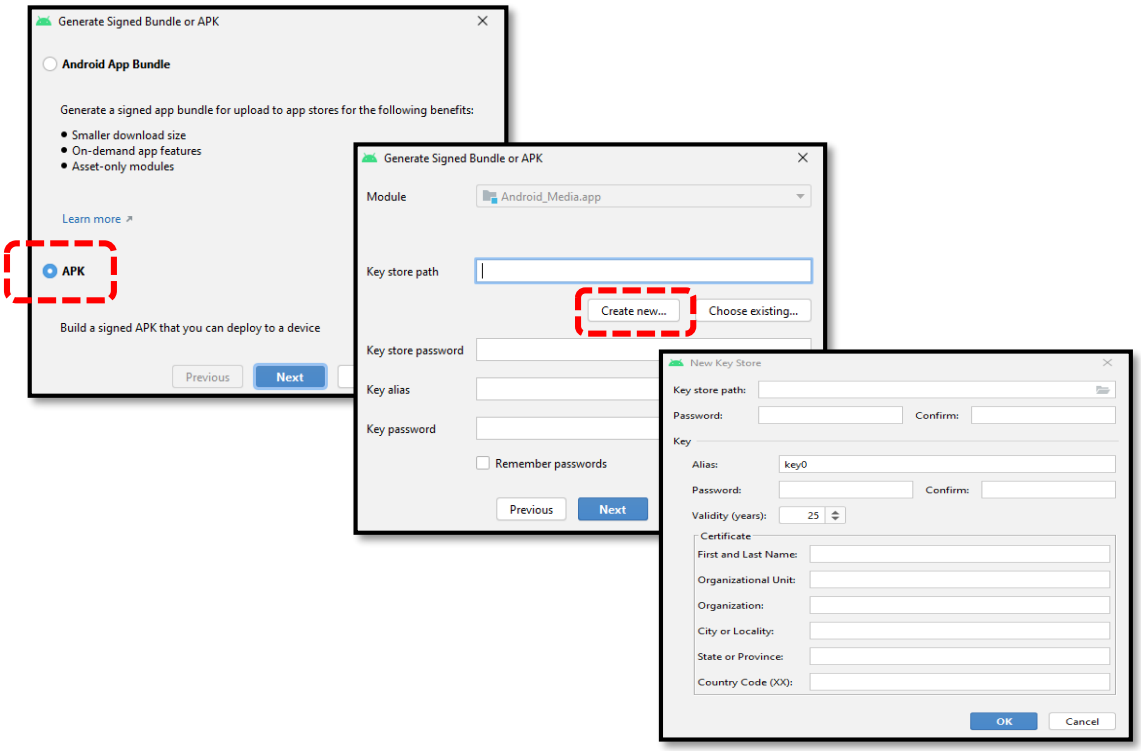


Figure 4-79: Create a new upload key and keystore in Android Studio

- In the menu bar, click **Build > Generate Signed Bundle/APK.**
- In the **Generate Signed Bundle or APK** dialog, **select Android App Bundle** or **APK** and click **Next.**
- Below the field for **Key store path**, click **Create new.**

- On the **New Key Store** window, provide the information for keystore and key.

  Key store

  Key store path: Select the location where keystore should be created.

  Password: Create and confirm a secure password for keystore.

  Key

  Alias: Enter an identifying name for key.

  Password: Create and confirm a secure password for key. This should be different from the password chose for keystore.

  Validity (years): Set the length of time in years that key will be valid. The key should be valid for at least 25 years, so you can sign app updates with the same key through the lifespan of app.

  Certificate: Enter some information about yourself for certificate. This information is not displayed in app, but is included in your certificate as part of the APK.

- Once complete the form, click **OK.**

2. Sign app with upload key

To sign app using Android Studio, and export an existing app signing key, follow these steps:

- Click **Build > Generate Signed Bundle/APK.**
- In the **Generate Signed Bundle or APK** dialog, select either **Android App Bundle** or **APK** and click **Next.**

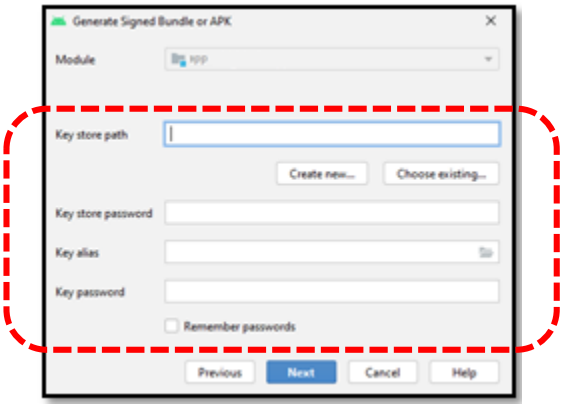


Figure 4-80: Generate signed bundle

- Select a module from the drop down.
- Specify the path to keystore, the alias for key, and enter the passwords for both. Click **Next.**
- Select a destination folder for signed app, select the build type (free/paid), choose the product flavor(s) if applicable. Click **Finish.**

3. Opt in to Play App Signing

To configure signing for an app that has not yet been published to Google Play, proceed as follows:

- Sign in to your Play Console.
- Follow the steps to prepare & roll out release to create a new release.
- After you choose a release track, configure app signing under the **App Integrity** section. The key use to sign first release becomes upload key, and should use it to sign future releases.

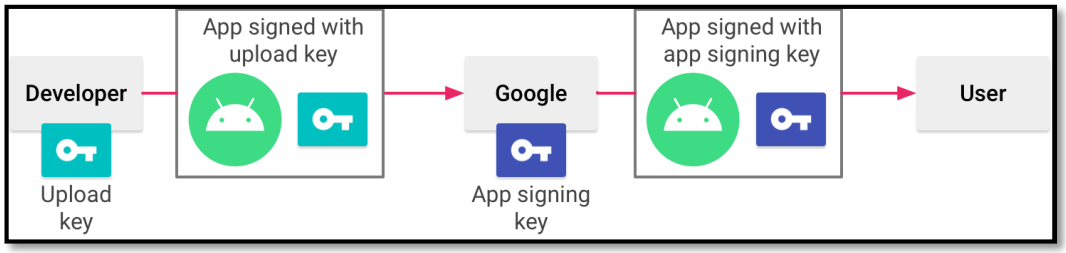


Figure 4-81: Signing an app with Play App Signing

4. Upload your app to Google Play

After build and sign the release version of app, the next step is to upload it to Google Play to inspect, test, and publish app. Google Play supports compressed app downloads of only 150 MB or less.

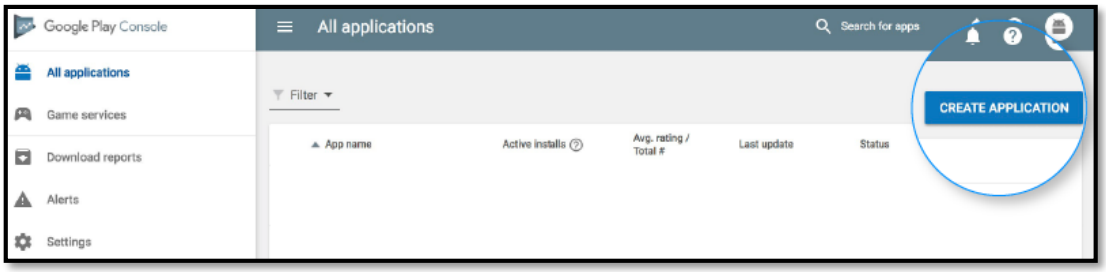


Figure 4-82: Publish an app on Play Store

5. Prepare & roll out release of your app

Android apps can be released in several ways. Usually, apps are released through an application marketplace such as Google Play, but can also release apps on their own websites or by sending apps directly to users.

Figure 4-83: Release app

Distribute the application on online Application Stores

App distribution is the process of releasing an app to a broad set of users in order to promote app engagement and usage. Often, app marketers will seek out app distribution channels and platforms as a way to advertise their app - either organically or paid.

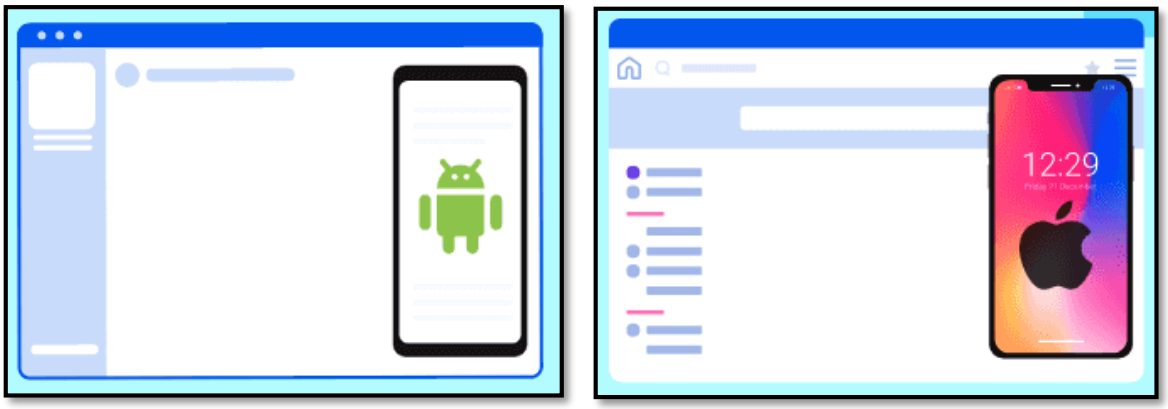


Figure 4-84: Distribute app

1. **Pre-Launch**

   Test app potential with an early release. Introduce a few core features of the app and use popular channels with strong communities, to gain interest and attract people. This helps growing user base and get the attention of investors. With an early release, fascinate a few early adopters before start with creating the brand-new framework.

2. **Optimize App**

   Optimize app for app stores just like optimize a website for a search engine. Find the relevant keywords with the help of effective SEO tools and make sure add those keywords to the app title and description.

3. **Find the Niche**

   To help app stand out, differentiate it with the other apps available in the market. There are two ways of doing this – release such features that no other app in the app store has or find own niche.

4. **Craft App Presence on App Store**

   'The First Impression is the Last Impression'. When users find app in their searches and stumble upon it, get a few seconds to impress them. Make sure app's home page is loaded with well-crafted screenshots and has engaging content. It should highlight the effective features and other factors of app that can help users understand app better. If app is paid, offer users a basic version with a free download and limited features.

5. **Be Exclusive**

   Make sure uniquely promote app. Use the app marketing strategies that help develop people's interest and attract them to download app. There are various effective ways to promote an app for free.

6. **Leverage Social Presence**

   Social Media is one of the most powerful marketing tools available online. Leverage social media presence to the optimum level. Provide users with the social media integrations in the app so that they could share their experiences and achievements among their social circle. Offer reward points or coupon discounts for every referral share.

7. **Develop User Interest**

   Release app for a few users first and have rest use the app through referrals and invites. This makes users curious about app and helps get the target audience by word of mouth marketing. Also, good reviews from these users can help rank higher in App Store optimization.

References

1. Iversen, J., & Eierman, M. (2014). *Learning Mobile App Development: A Hands-On Guide to Building Apps with Ios and Android*. Pearson Education.

2. Smyth, N. (2015). *Android Studio Development Essentials: Android 6 Edition*. eBookFrenzy.

3. https://developer.android.com/guide/platform

4. https://www.javatpoint.com/android-life-cycle-of-activity

5. https://www.tutorialspoint.com/android/android_overview.htm

6. https://www.tutlane.com/tutorial/android/android-framelayout-with-examples

7. https://www.geeksforgeeks.org/screen-orientations-in-android-with-examples/

8. https://www.c-sharpcorner.com/article/designing-user-interface-with-views-in-android-application/

"This book starts with the basics of Android, from emulator to Android Studio and expects you to have some prior programming language knowledge. Simple presentation, only fun, wit and information. This book guides you correctly on important topics from the developmental perspective on each topic."

The Next Big App Idea : Mobile Application Development For Malaysian Polytechnic Students

by Melati Sabtu, Syaiful Bachtiar Shahinan

e ISBN 978-967-2099-69-7

9 7 8 9 6 7 2 0 9 9 6 9 7