Unit- 2 Introduction to Android and Android Application Design

Introduction to Android?

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

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

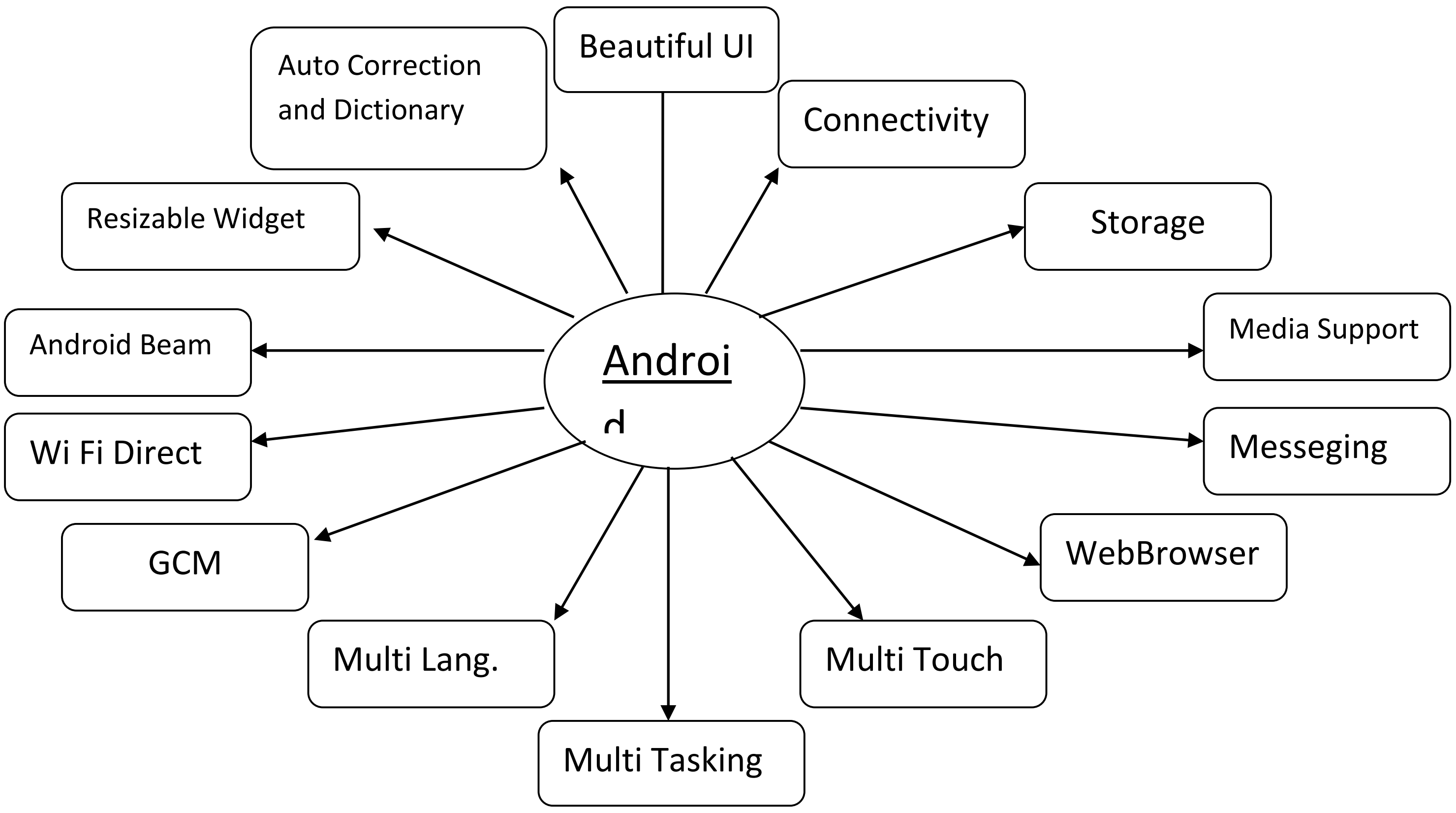
→ The first beta version of the Android Software Development Kit (SDK) was released by Google in 2007 where as the first commercial version, Android 1.0, was released in September 2008.

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

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

Features of Android

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



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

History of Android

→ Initially, Andy Rubin founded Android Incorporation in Palo Alto, California, United States in October, 2003.
→ In 17th August 2005, Google acquired android Incorporation. Since then, it is in the subsidiary of Google Incorporation.
→ The key employees of Android Incorporation are Andy Rubin, Rich Miner, Chris White and Nick Sears.
→ Originally intended for camera but shifted to smart phones later because of low market for camera only.
→ Android is the nick name of Andy Rubin given by coworkers because of his love to robots.
→ In 2007, Google announces the development of android OS.

→ Android made its official public debut in 2008 with Android 1.0 — a release so ancient it didn't even have a cute codename.
→ The first Android mobile was publicly released with Android 1.0 of the T-Mobile G1 (aka HTC Dream) in October 2008.
→ The code names of android ranges from A to P currently, such as Aestro, Blender, Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwitch, Jelly Bean, KitKat, Lollipop Marshmallow, Nougat, Oreo, Pie.
→ Google announced in August 2019 that they were ending the confectionery scheme, and they use numerical ordering for future Android versions.
→ The first Android version which was released under the numerical order format was Android 10.
→ The latest version of Android is Android 11 released on September 8, 2020.

Android Applications

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

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

Android IDEs

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

Android Architecture

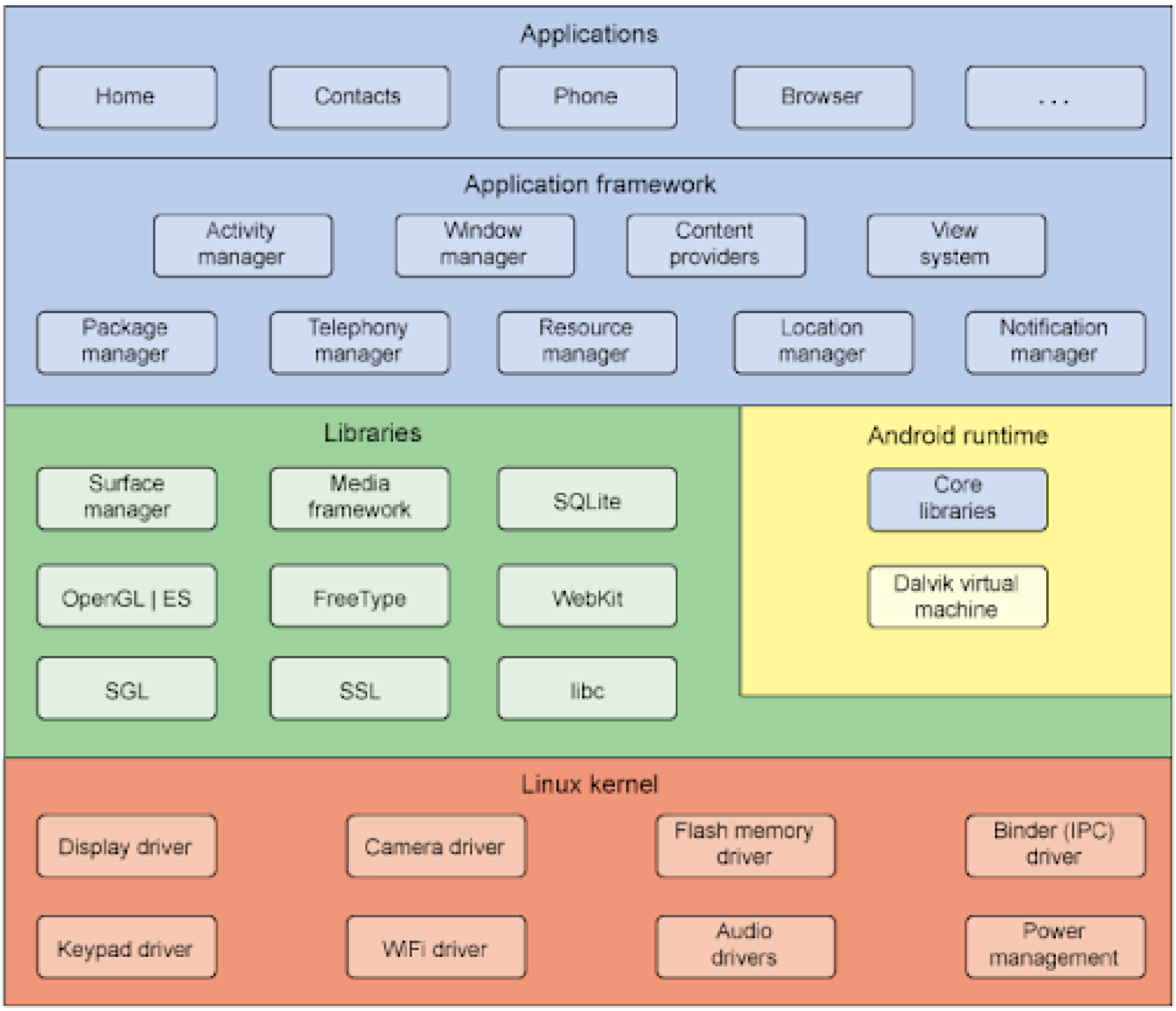
→ android architecture or Android software stack is categorized into five parts:
1. Linux kernel
2. Native libraries (middleware),
3. Android Runtime
4. Application Framework
5. Applications

1. Linux kernel

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

2. Libraries(Native Libraries)

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



3. Android Runtime

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

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

- Core Libraries

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

✓ android.app − Provides access to the application model and is the cornerstone of all Android applications.

✓ android.content − Facilitates content access, publishing and messaging between applications and application components.

✓ android.database − Used to access data published by content providers and includes SQLite database management classes.

✓ android.opengl − A Java interface to the OpenGL ES 3D graphics rendering API.

✓ android.os − Provides applications with access to standard operating system services including messages, system services and inter-process communication.

✓ android.text − Used to render and manipulate text on a device display.

✓ android.view − The fundamental building blocks of application user interfaces.

✓ android.widget − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.

- ✓ android.webkit − A set of classes intended to allow web-browsing capabilities to be built into applications.

- DVM

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

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

4. Application Framework

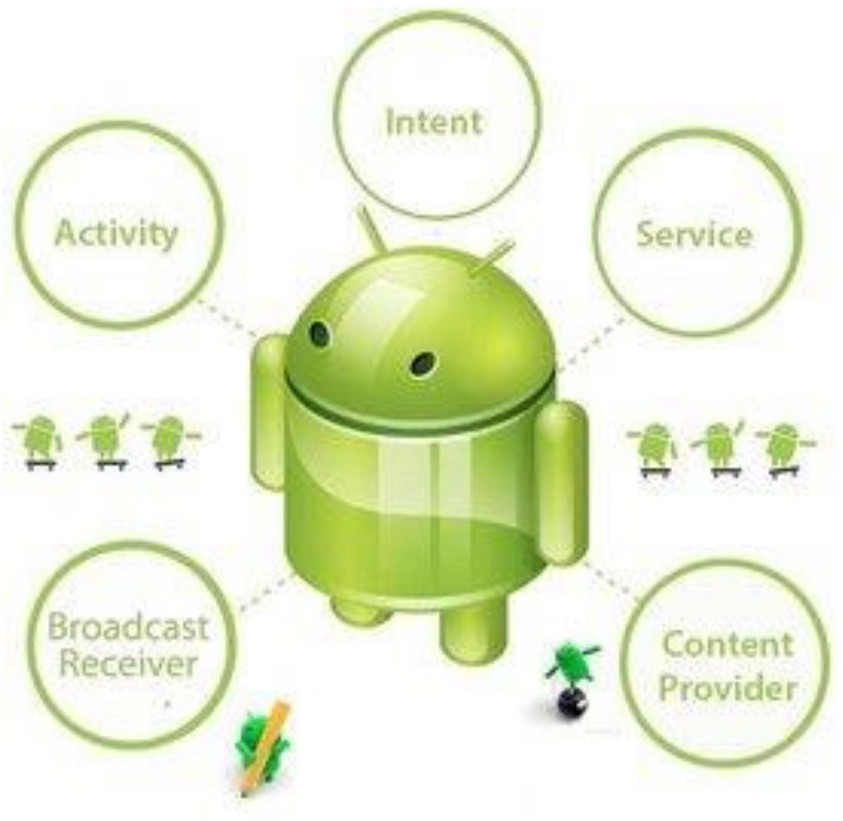
→ The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.
→ The Android framework includes the following key services −

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

5. Applications

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

Android Core Building Blocks



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

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

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

1. Activities

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

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

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

2. Services

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

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

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

3. Broadcast Receivers

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

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

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

4. Content Providers

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

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

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

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

Additional Components

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

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

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

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

4. **Intents**
   Intent is used to invoke components. It is mainly used to:
   - Start the service
   - Launch an activity
   - Display a web page
   - Display a list of contacts
   - Broadcast a message
   - Dial a phone call etc.

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

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

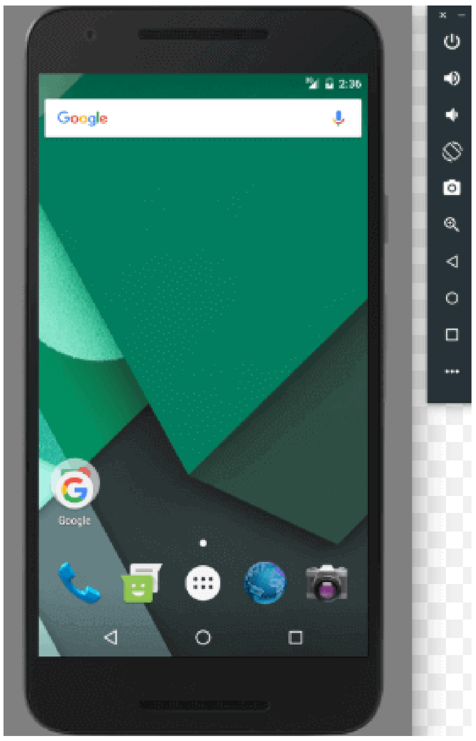
5. **Resources**
   External elements, such as strings, constants and drawable pictures.

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

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

Android Emulator

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



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

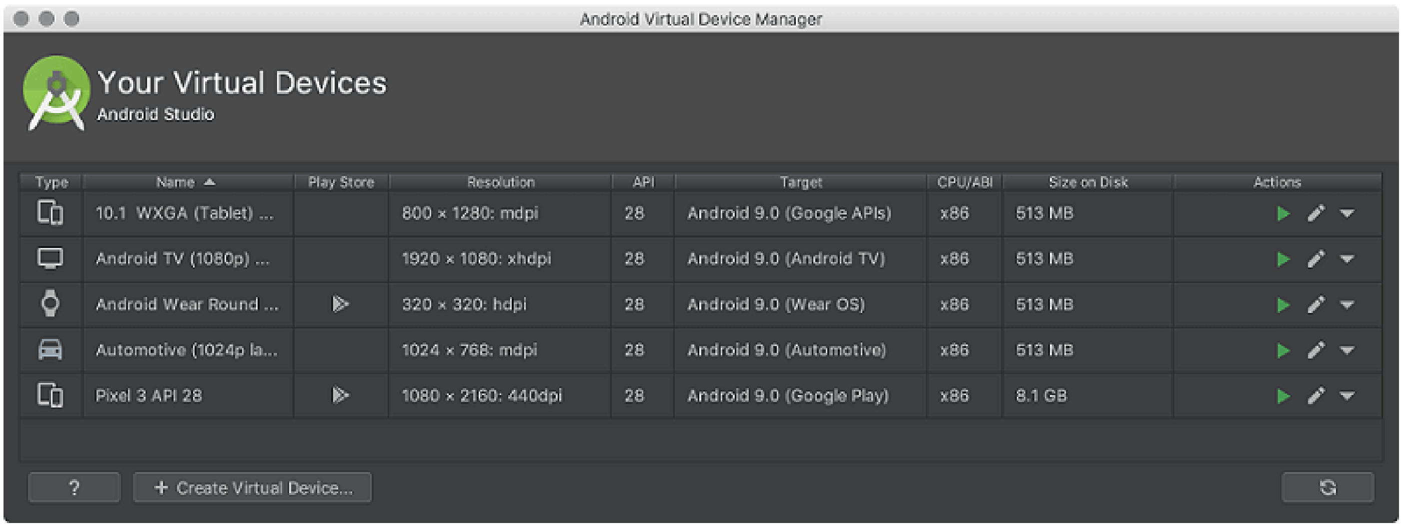
<u>Requirement and recommendations</u>
→ The Android emulator takes additional requirements beyond the basic system requirement for Android Studio. These requirements are given below:
- SDK Tools 26.1.1 or higher
- 64-bit processor
- Windows: CPU with UG (unrestricted guest) support
- HAXM 6.2.1 or later (recommended HAXM 7.2.0 or later)

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

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

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

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

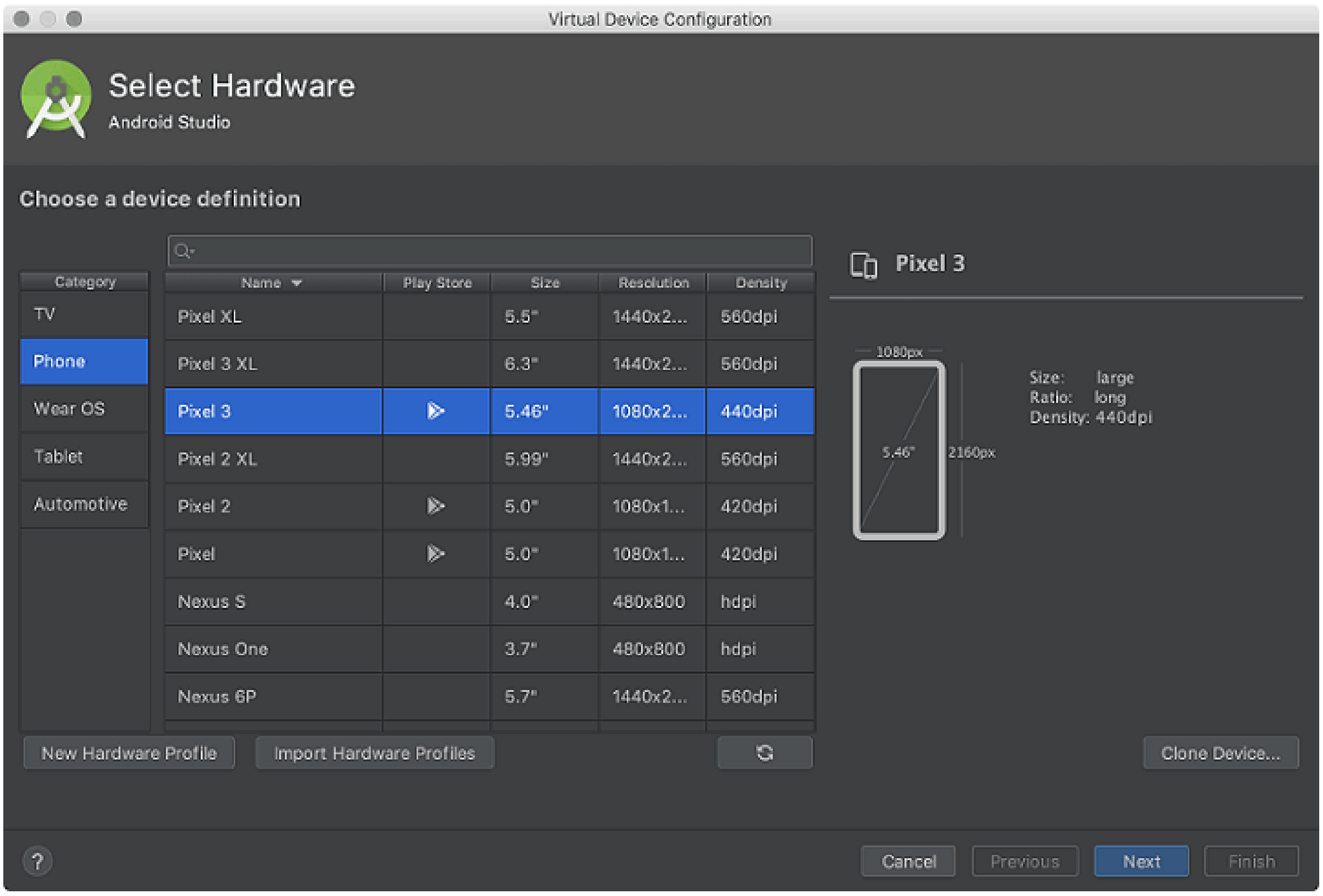


Android Virtual Device Manager

Your Virtual Devices
Android Studio

Type	Name	Play Store	Resolution	API	Target	CPU/ABI	Size on Disk	Actions
	10.1 WXGA (Tablet) ...		800 × 1280: mdpi	28	Android 9.0 (Google APIs)	x86	513 MB	
	Android TV (1080p) ...		1920 × 1080: xhdpi	28	Android 9.0 (Android TV)	x86	513 MB	
	Android Wear Round ...	▶	320 × 320: hdpi	28	Android 9.0 (Wear OS)	x86	513 MB	
	Automotive (1024p la...		1024 × 768: mdpi	28	Android 9.0 (Automotive)	x86	513 MB	
	Pixel 3 API 28	▶	1080 × 2160: 440dpi	28	Android 9.0 (Google Play)	x86	8.1 GB	

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



Virtual Device Configuration

Select Hardware
Android Studio

Choose a device definition

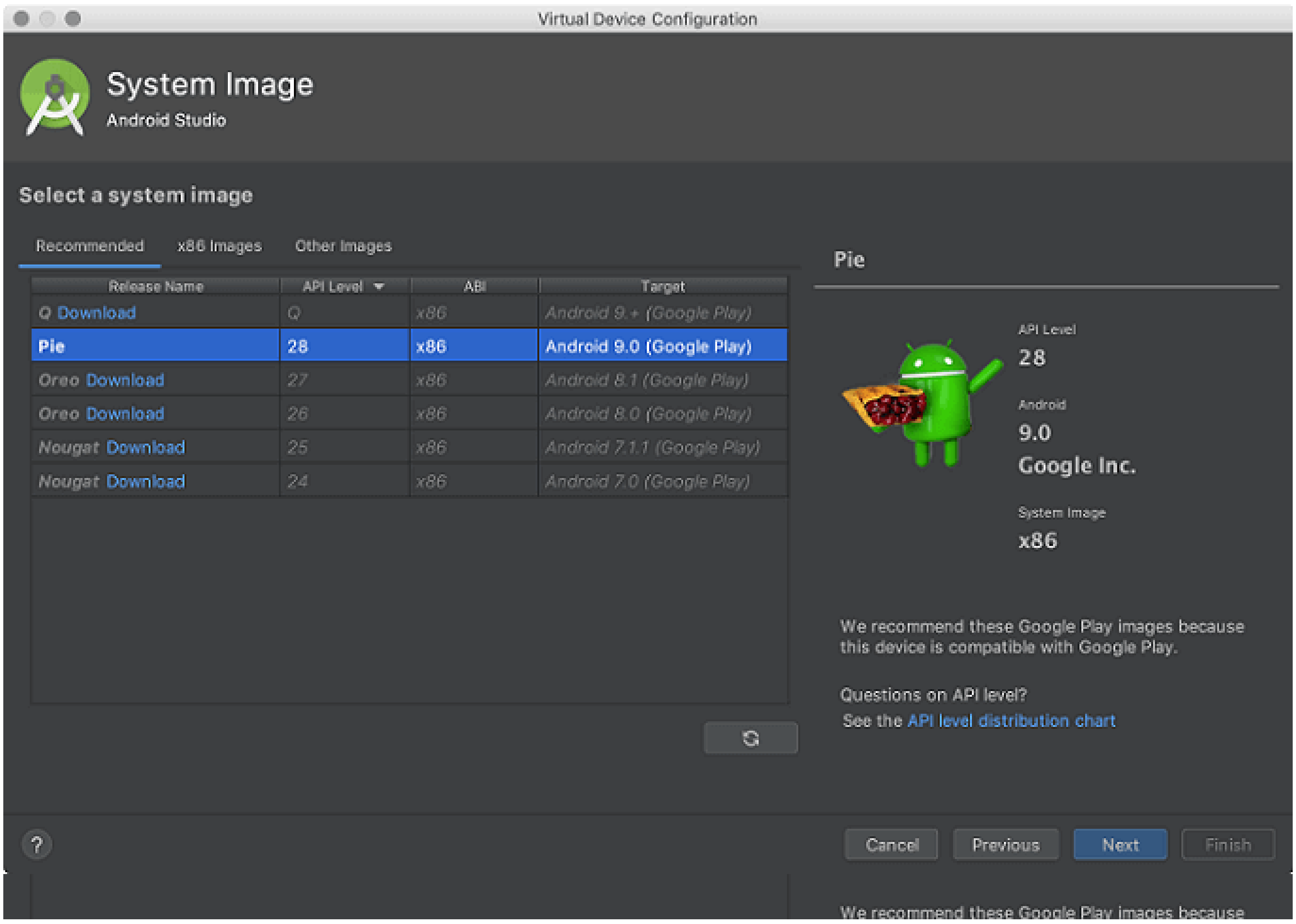
Category
TV
Phone
Wear OS
Tablet
Automotive

Name	Play Store	Size	Resolution	Density
Pixel XL		5.5"	1440x2...	560dpi
Pixel 3 XL		6.3"	1440x2...	560dpi
Pixel 3	▶	5.46"	1080x2...	440dpi
Pixel 2 XL		5.99"	1440x2...	560dpi
Pixel 2	▶	5.0"	1080x1...	420dpi
Pixel	▶	5.0"	1080x1...	420dpi
Nexus S		4.0"	480x800	hdpi
Nexus One		3.7"	480x800	hdpi
Nexus 6P		5.7"	1440x2...	560dpi

Pixel 3

Size: large
Ratio: long
Density: 440dpi

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


Virtual Device Configuration

System Image
Android Studio

Select a system image

Recommended | x86 Images | Other Images

Release Name	API Level	ABI	Target
Q Download	Q	x86	Android 9.+ (Google Play)
Pie	28	x86	Android 9.0 (Google Play)
Oreo Download	27	x86	Android 8.1 (Google Play)
Oreo Download	26	x86	Android 8.0 (Google Play)
Nougat Download	25	x86	Android 7.1.1 (Google Play)
Nougat Download	24	x86	Android 7.0 (Google Play)

Pie

API Level
28

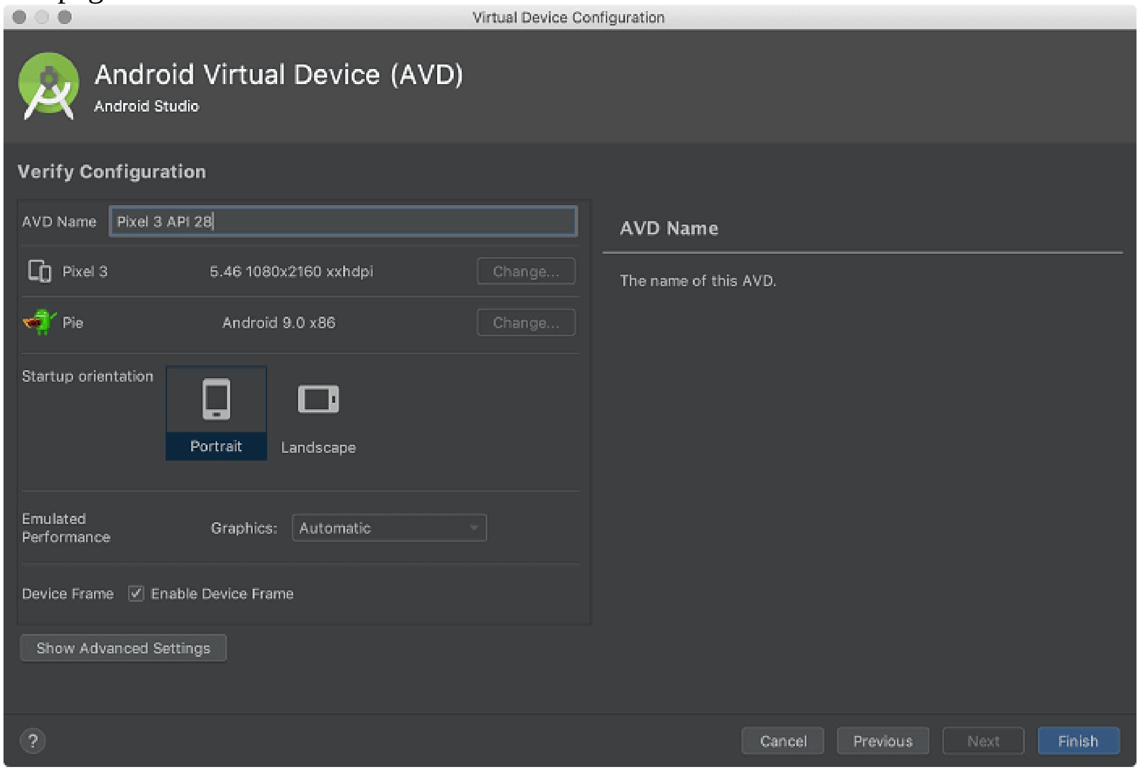
Android
9.0
Google Inc.

System Image
x86

We recommend these Google Play images because this device is compatible with Google Play.

Questions on API level?
See the API level distribution chart

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


Virtual Device Configuration

Android Virtual Device (AVD)
Android Studio

Verify Configuration

AVD Name: Pixel 3 API 28

Pixel 3 — 5.46 1080x2160 xxhdpi

Pie — Android 9.0 x86

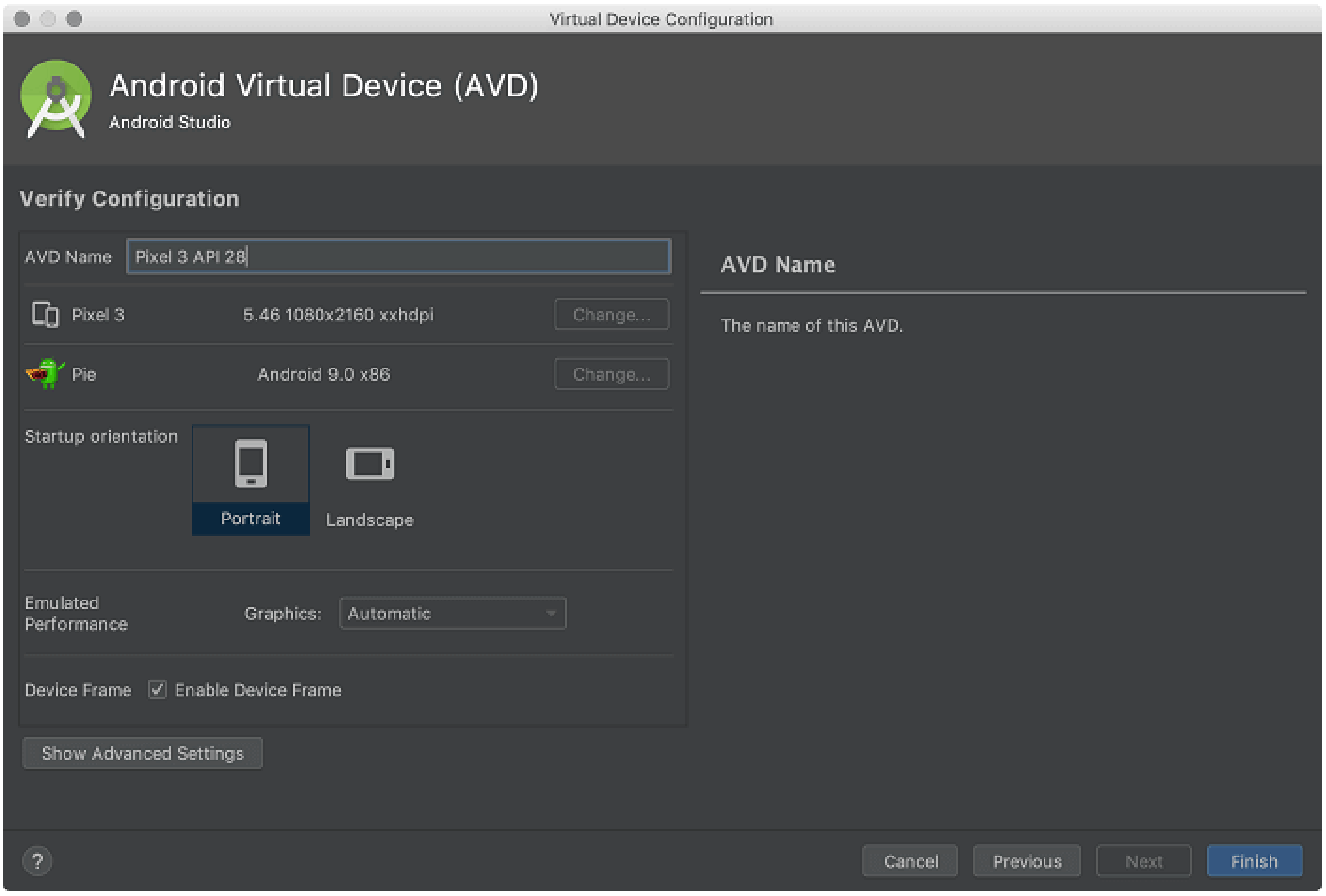
Startup orientation: Portrait | Landscape

Emulated Performance — Graphics: Automatic

Device Frame — ☑ Enable Device Frame

AVD Name

The name of this AVD.


Virtual Device Configuration

Android Virtual Device (AVD)
Android Studio

Verify Configuration

AVD Name: Pixel 3 API 28

Pixel 3 — 5.46 1080x2160 xxhdpi — Change...

Pie — Android 9.0 x86 — Change...

Startup orientation: [✓ Portrait] Landscape

Emulated Performance — Graphics: Automatic

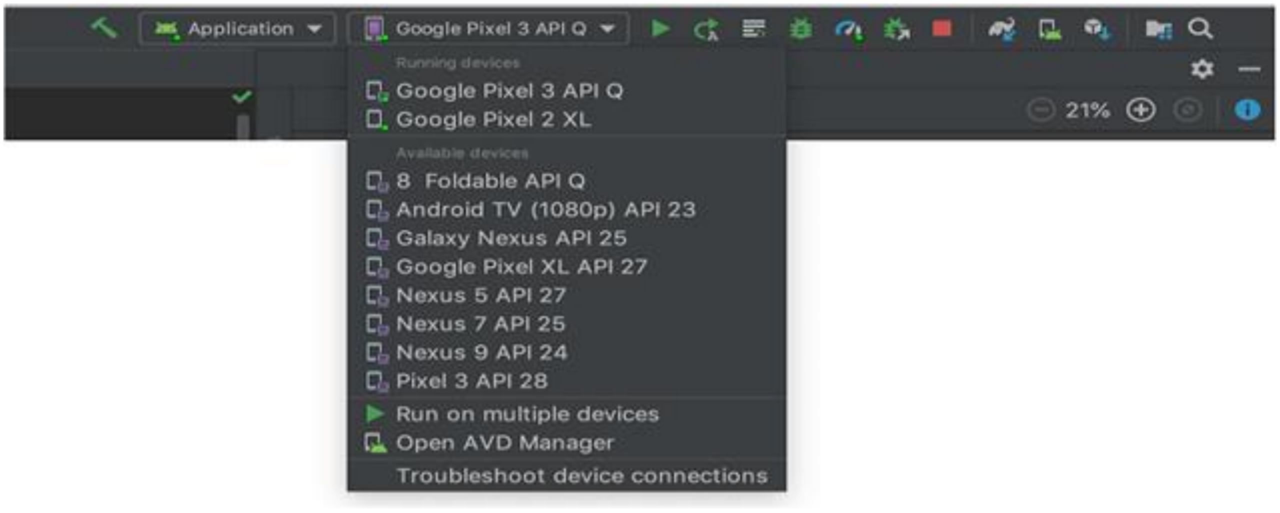
Device Frame — ✓ Enable Device Frame

Show Advanced Settings

AVD Name

The name of this AVD.

1.5 Change AVD properties if needed, and then click Finish.
2. In the toolbar, choose the AVD, which we want to run our app from the target device from the drop-down menu.



Running devices
- Google Pixel 3 API Q
- Google Pixel 2 XL

Available devices
- 8 Foldable API Q
- Android TV (1080p) API 23
- Galaxy Nexus API 25
- Google Pixel XL API 27
- Nexus 5 API 27
- Nexus 7 API 25
- Nexus 9 API 24
- Pixel 3 API 28
- Run on multiple devices
- Open AVD Manager
- Troubleshoot device connections

3. Click Run.
<u>Launch the Emulator without first running an app</u>
To start the emulator:

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

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

Building Sample Android Application

➢ Apps provide multiple entry points

→ Android apps are built as a combination of components that can be invoked individually. For example, an *activity* is a type of app component that provides a user interface (UI).
→ The "main" activity starts when the user taps our app's icon.Wou can also direct the user to an activity from elsewhere, such as from a notification or even from a different app.
→ Other components, such as *WorkManager*, allow our app to perform background tasks without a UI.

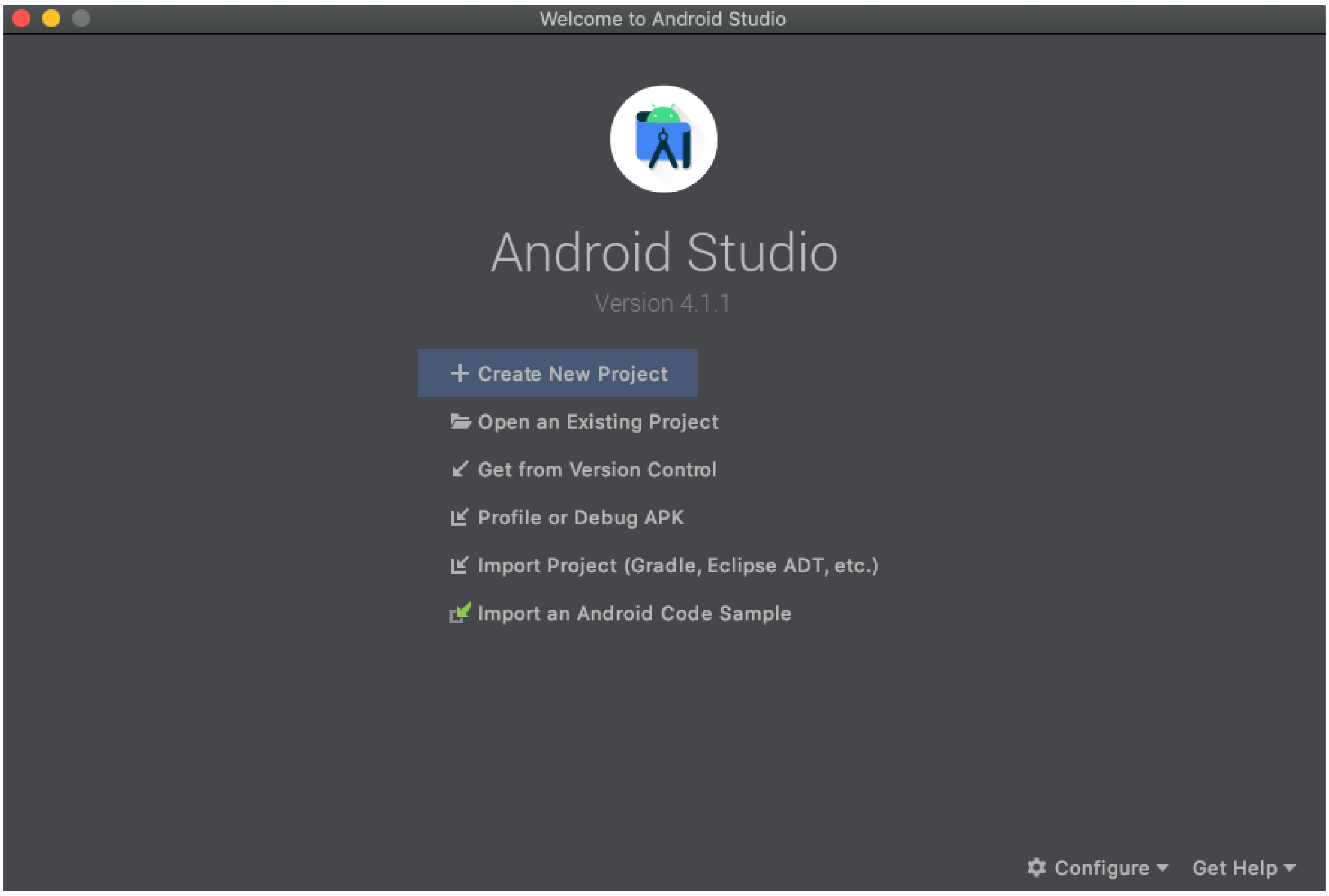
➢ Apps adapt to different devices

→ Android allows we to provide different resources for different devices. For example, we can create different layouts for different screen sizes. The system determines which layout to use based on the screen size of the current device.
→ If any of our app's features need specific hardware, such as a camera, we can query at runtime whether the device has that hardware or not, and then disable the corresponding features if it doesn't. We can specify that our app requires certain hardware so that Google Play won't allow the app to be installed on devices without them.

➢ Create an Android project

→ To create our new Android project, follow these steps:

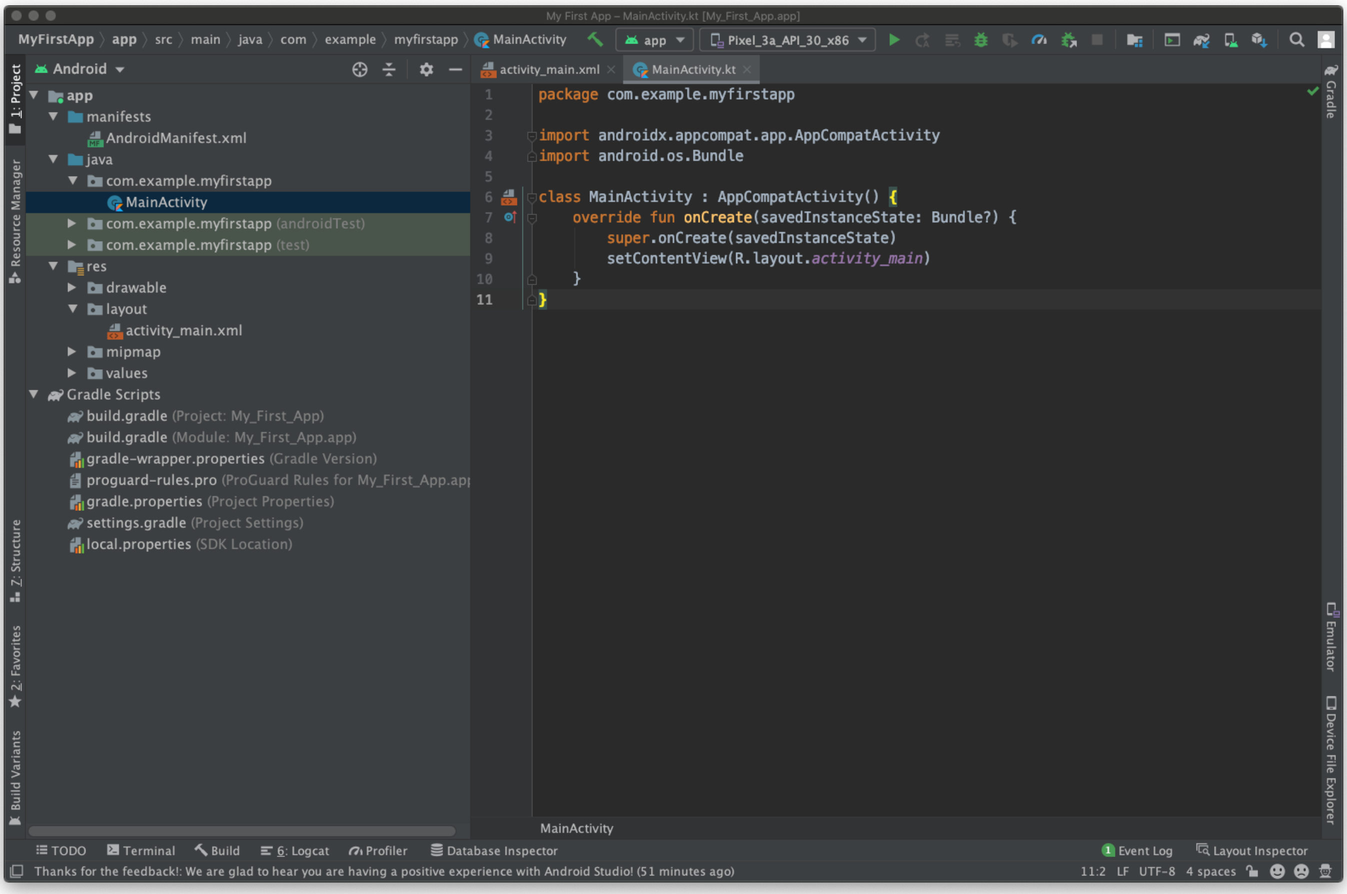
1. **Install the latest version of Android Studio.**
2. **In the** Welcome to Android Studio **window, click** Create New Project**.**



3. **If we have a project already opened, select File > New > New Project.**

4. **In the Select a Project Template window, select Empty Activity and click Next.**
5. **In the Configure our project window, complete the following:**
   1) **Enter "My First App" in the Name field.**
   2) **Enter "com.example.myfirstapp" in the Package name field.**
   3) **If you'd like to place the project in a different folder, change its Save location.**
   4) **Select either Java or Kotlin from the Language drop-down menu.**
   5) **Select the lowest version of Android we want our app to support in the Minimum SDK field.**
   6) **If our app will require legacy library support, mark the Use legacy android.support libraries checkbox.**
   7) **Leave the other options as they are.**
6. **Click Finish.**

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


My First App – MainActivity.kt [My_First_App.app]
MyFirstApp › app › src › main › java › com › example › myfirstapp › MainActivity
app
Pixel_3a_API_30_x86
Android
app
manifests
AndroidManifest.xml
java
com.example.myfirstapp
MainActivity
com.example.myfirstapp (androidTest)
com.example.myfirstapp (test)
res
drawable
layout
activity_main.xml
mipmap
values
Gradle Scripts
build.gradle (Project: My_First_App)
build.gradle (Module: My_First_App.app)
gradle-wrapper.properties (Gradle Version)
proguard-rules.pro (ProGuard Rules for My_First_App.ap
gradle.properties (Project Properties)
settings.gradle (Project Settings)
local.properties (SDK Location)
activity_main.xml
MainActivity.kt
MainActivity
TODO
Terminal
Build
6: Logcat
Profiler
Database Inspector
Event Log
Layout Inspector
11:2 LF UTF-8 4 spaces

```
package com.example.myfirstapp

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Thanks for the feedback!: We are glad to hear you are having a positive experience with Android Studio! (51 minutes ago)

→ Now take a moment to review the most important files.

→ 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. We can then see the following files:

app > java > com.example.myfirstapp > MainActivity

This is the main activity. It's the entry point for our app. When we build and run our app, the system launches an instance of this Activity and loads its layout.

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!"

app > manifests > AndroidManifest.xml

The manifest file describes the fundamental characteristics of the app and defines each of its components.

Gradle Scripts > build.gradle

There are two files with this name: one for the project, "Project: My_First_App," and one for the app module, "Module: My_First_App.app." Each module has its own build.gradle file, but this project currently has just one module. Use each module's build.gradle file to control how the Gradle plugin builds our app. For more information about this file, see Configure our build.

➢ Run app

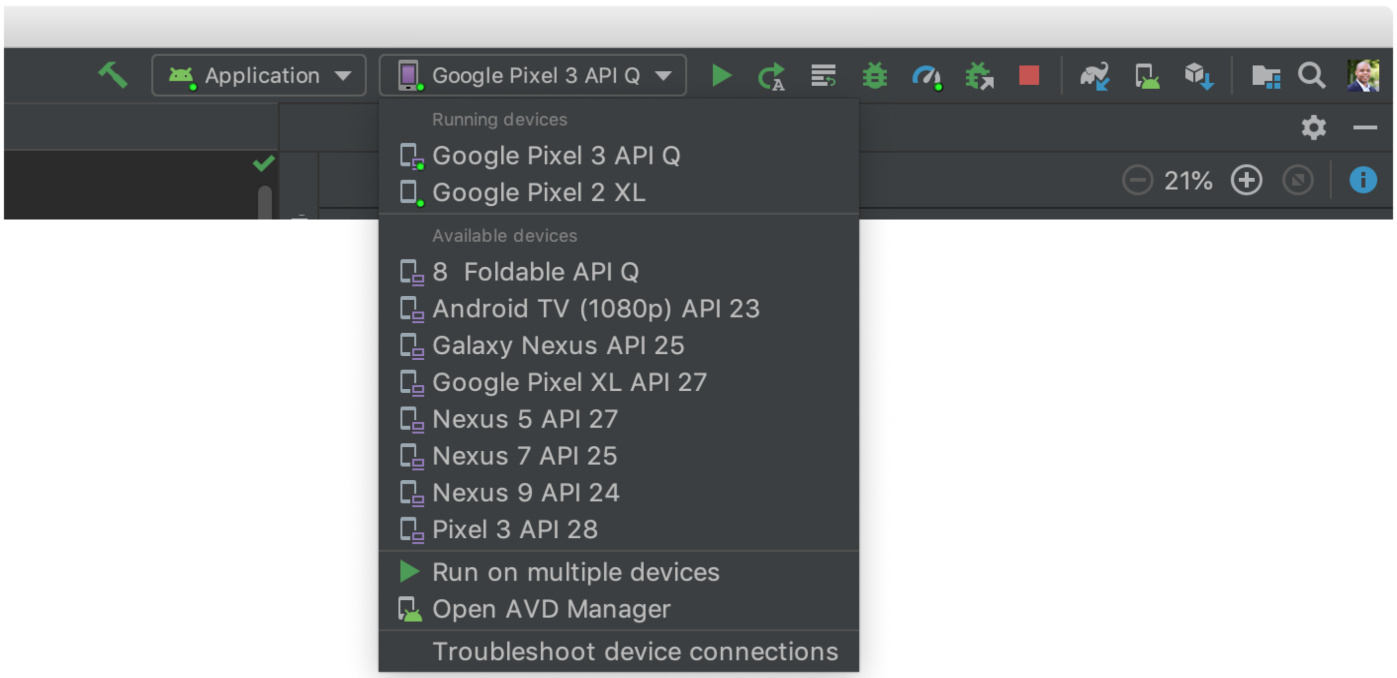
I. <u>Run on a real device</u>

→ Set up device as follows:

1. **Connect device to development machine with a USB cable. If we developed on Windows, we might need to install the appropriate USB driver for device.**
2. **Perform the following steps to enable** USB debugging **in the** Developer options **window:**
   1) Open the Settings app.
   2) If device uses Android v8.0 or higher, select System. Otherwise, proceed to the next step.
   3) Scroll to the bottom and select About phone.
   4) Scroll to the bottom and tap Build number seven times.
   5) Return to the previous screen, scroll to the bottom, and tap Developer options.
   6) In the Developer options window, scroll down to find and enable USB debugging.

→ Run the app on device as follows:

1. **In Android Studio, select our app from the run/debug configurations drop-down menu in the toolbar.**
2. **In the toolbar, select the device that we want to run app on from the target device drop-down menu.**



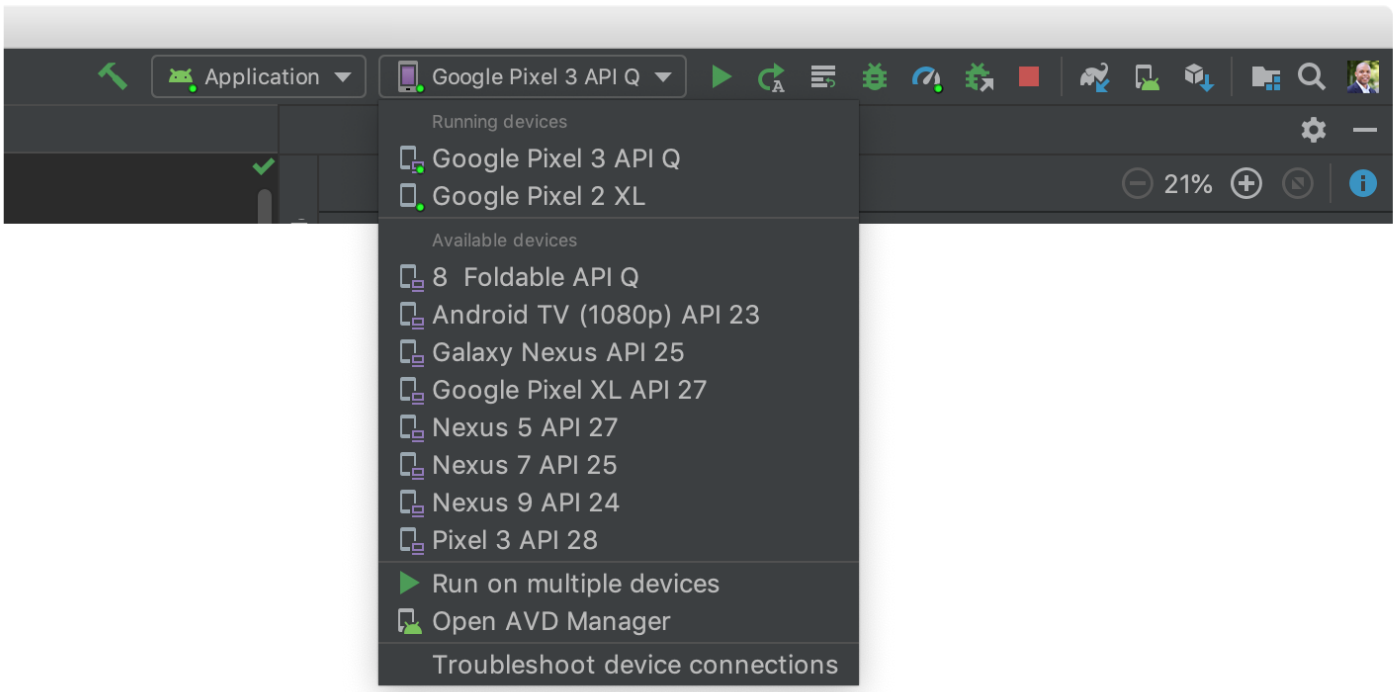
3. **Click Run ▶.**

→ Android Studio installs our app on our connected device and starts it. We now see "Hello, World!" displayed in the app on our device.

II. <u>Run on an emulator</u>

→ Run the app on an emulator as follows:

1. **In Android Studio, create an Android Virtual Device (AVD) that the emulator can use to install and run our app.**
2. **In the toolbar, select our app from the run/debug configurations drop-down menu.**
3. **From the target device drop-down menu, select the AVD that we want to run our app on.**

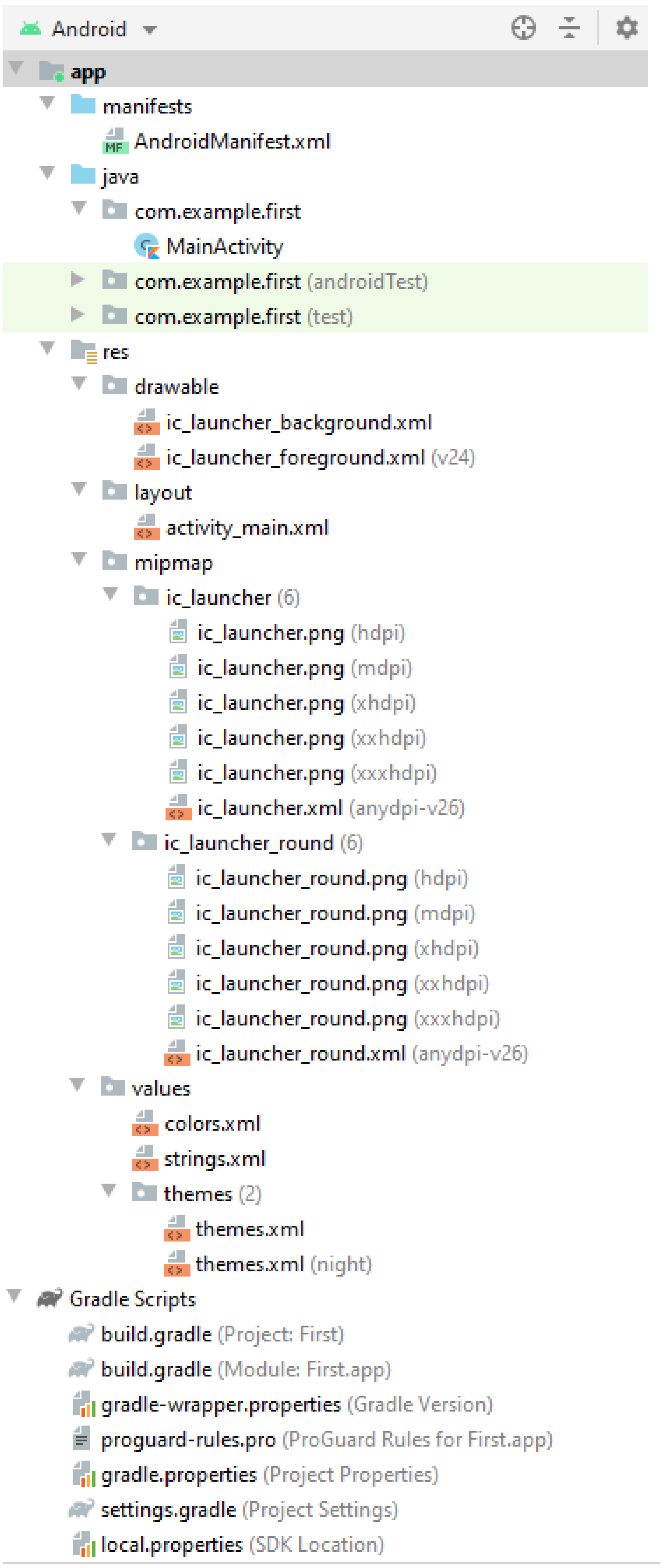


4. **Click Run ▶.**

→ Android Studio installs the app on the AVD and starts the emulator. We now see "Hello, World!" displayed in the app.

Anatomy of Android Application

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



```
Android
app
  manifests
    AndroidManifest.xml
  java
    com.example.first
      MainActivity
    com.example.first (androidTest)
    com.example.first (test)
  res
    drawable
      ic_launcher_background.xml
      ic_launcher_foreground.xml (v24)
    layout
      activity_main.xml
    mipmap
      ic_launcher (6)
        ic_launcher.png (hdpi)
        ic_launcher.png (mdpi)
        ic_launcher.png (xhdpi)
        ic_launcher.png (xxhdpi)
        ic_launcher.png (xxxhdpi)
        ic_launcher.xml (anydpi-v26)
      ic_launcher_round (6)
        ic_launcher_round.png (hdpi)
        ic_launcher_round.png (mdpi)
        ic_launcher_round.png (xhdpi)
        ic_launcher_round.png (xxhdpi)
        ic_launcher_round.png (xxxhdpi)
        ic_launcher_round.xml (anydpi-v26)
    values
      colors.xml
      strings.xml
      themes (2)
        themes.xml
        themes.xml (night)
Gradle Scripts
  build.gradle (Project: First)
  build.gradle (Module: First.app)
  gradle-wrapper.properties (Gradle Version)
  proguard-rules.pro (ProGuard Rules for First.app)
  gradle.properties (Project Properties)
  settings.gradle (Project Settings)
  local.properties (SDK Location)
```

- **AndroidManifeast.xml**

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

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

- **Java**

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

- **res(Resource)**

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

  - **res/drawable**

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

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

  - **res/layout**

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

  - **res/menu**

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

  - **res/mipmap**

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

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

  - **res/value**

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

- **Build.Gradle**

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

Open Handset Alliance (OHA)

→ The Open Handset Alliance (OHA) is a business alliance that was created for the purpose of developing open mobile device standards. It's a consortium of 84 companies such as google, samsung, AKM, synaptics, KDDI, Garmin, Teleca, Ebay, Intel etc.
→ OHA is a group of Mobile and technology leaders who share the vision for changing the mobile experience.
→ The OHA was established on 5th November 2007, led by Google with 34 members, including mobile makers, application developer, some mobile carriers and chip makers. It is committed to advance open standards, provide services and deploy handsets using the Android Platform.
→ Now OHA is a consortium of 84 firms to develop open firms to develop open standards for mobile devices. Member firms include HTC, Sony, Dell, Intel, Motorola, Qualcomm, Texas Instruments, Google, Samsung Electronics, LG Electronics, T-Mobile, Sprint Corporation (now merged with T-Mobile US), Nvidia, and Wind River Systems.

Application, Context, Activities, Services, Intents

⇒ Context

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

⇒ Application

→ Base class for maintaining global application state. The application object is created whenever one of our Android Components is started. It is started in a new process with a unique ID under a single user.
→ Even if we do not specify one in our AndroidManifest.xml file, the Android System creates a default object for us. This object provides the following life cycle methods:

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

→ We can provide our own implementation by creating a subclass and specifying the fully-qualified name of this subclass as the "android:name" attribute in our AndroidManifest.xml's <application> tag. The Application class, or our subclass of the Application class, is instantiated before any other class when the process for our application/package is created.
→ The application object starts before any component and runs at least as long as another component of the application runs.
→ If the Android system needs to terminate processes it follows the following priority system.

→ **Priorities:**

Process	**Detail**	**Priority**
foreground	An application in which the user is interacting with an activity, or which has service which is bound to such an activity. Also if a service is executing	1

	one of its lifecycle methods or a broadcast receiver which runs its onReceive() method.	
visible	User is not interacting with the activity, but the activity is still(partially) visible or the application has a service which is used by a inactive but visible activity	2
service	Application with a running service which does not qualify for 1 or 2.	3
background	Application with only stopped activities and without a service or executing receiver. Android keeps them in a least recent used (LRU) list and if requires terminates the one which was least used.	4
empty	Application without any active components.	5

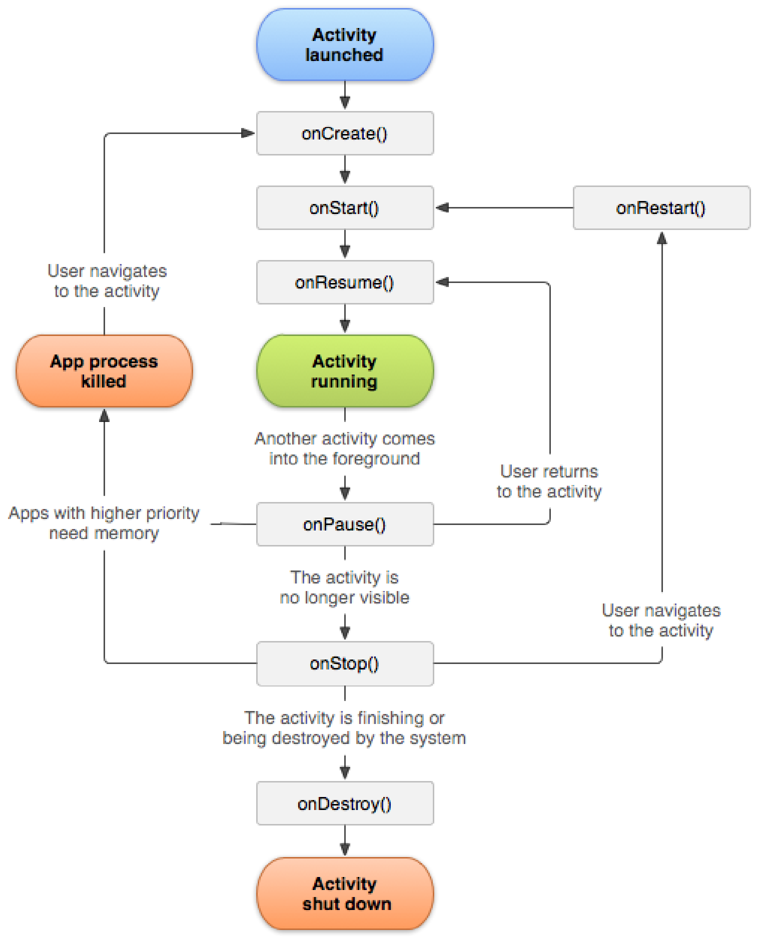
⇒ Activities

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

→ Activity State :

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

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


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

→ **Methods:**

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

→ **Example:**

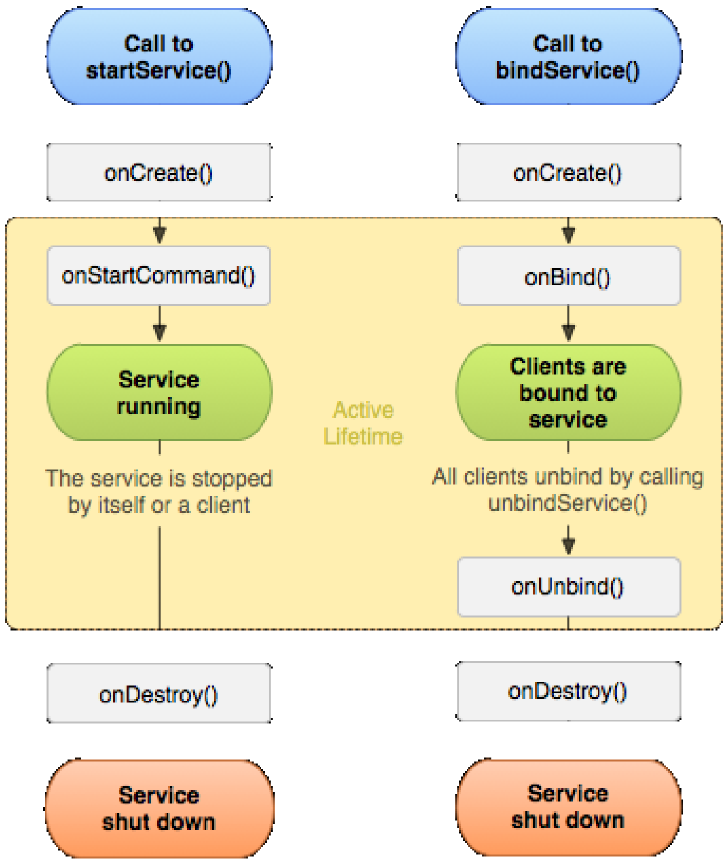
```
package com.example.myapplication

import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val toast = Toast.makeText(applicationContext, "onCreate Called", Toast.LENGTH_LONG).show()
    }

    override fun onStart() {
        super.onStart()
        val toast  = Toast.makeText(applicationContext, "onStart Called", Toast.LENGTH_LONG).show()
    }

    override fun onRestart() {
        super.onRestart()
        val toast = Toast.makeText(applicationContext, "onRestart Called",
Toast.LENGTH_LONG).show()
    }

    override fun onResume() {
        super.onResume()
        val toast = Toast.makeText(applicationContext, "onResume Called",
Toast.LENGTH_LONG).show()
    }

    override fun onPause() {
        super.onPause()
        val toast = Toast.makeText(applicationContext, "onPause Called", Toast.LENGTH_LONG).show()
    }

    override fun onStop() {
        super.onStop()
        val toast = Toast.makeText(applicationContext, "onStop Called", Toast.LENGTH_LONG).show()
    }
    override fun onDestroy() {
        super.onDestroy()
        val toast = Toast.makeText(applicationContext, "onDestroy Called",
Toast.LENGTH_LONG).show()
    }
}
```

⇒ Services

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

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

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



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

→ **Methods:**

Method	Detail
onStartCommand()	The system calls this method when another component, such as an activity, requests that the service be started, by calling startService(). If we implement this method, it is our responsibility to stop the service when its work is done, by calling stopSelf() or stopService() methods.
onBind()	The system calls this method when another component wants to bind with the service by calling bindService(). If we implement this method, we must provide an interface that clients use to communicate with the service, by returning an IBinder object. We must always implement this method, but if we don't want to allow binding, then we should return null.
onUnbind()	The system calls this method when all clients have disconnected from a particular interface published by the service.
onRebind()	The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its onUnbind(Intent).
onCreate()	The system calls this method when the service is first created using onStartCommand() or onBind(). This call is required to perform one-time set-up.
onDestroy()	The system calls this method when the service is no longer used and is being destroyed. Our service should implement this to clean up any resources such as threads, registered listeners, receivers, etc.

→ **Example :**

```
package com.example.myapplication

import android.app.Service
import android.content.Intent
import android.os.IBinder

class ExampleService : Service() {
   private var startMode: Int = 0            // indicates how to behave if the service is killed
   private var binder: IBinder? = null        // interface for clients that bind
   private var allowRebind: Boolean = false   // indicates whether onRebind should be used

   override fun onCreate() {
      // The service is being created
   }

   override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
      // The service is starting, due to a call to startService()
      return startMode
   }

   override fun onBind(intent: Intent): IBinder? {
      // A client is binding to the service with bindService()
      return binder
   }

   override fun onUnbind(intent: Intent): Boolean {
        // All clients have unbound with unbindService()
        return allowRebind
    }

    override fun onRebind(intent: Intent) {
        // A client is binding to the service with bindService(),
        // after onUnbind() has already been called
    }

    override fun onDestroy() {
        // The service is no longer used and is being destroyed
    }
}
```

⇒Intent and Intent Filter

➢ Intent

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

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

Starting an activity

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

Starting a service

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

Delivering a broadcast

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

➢ Intent types

→ There are two types of intents:

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

→ **Example:**

```
val intent = Intent(this, SecondActivity:: class.java).apply{
putExtra(“key”,”New Value”)
}
startActivity(intent)
```

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

→ **Example:**

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

OR

```
intent= Intent(Intent.ACTION_VIEW, Uri.parse("https://www.google.com/"))
startActivity(intent)
```

➢ Intent Object:

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

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

- **Component name**

→ The name of the component to start.

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

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

- **Action**

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

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

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

  - ACTION_VIEW
  - ACTION_SEND
  - ACTION_ANSWER
  - ACTION_BATTERY_LOW
  - ACTION_CALLED

- **Data**

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

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

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

- **Category**

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

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

- **Extras**

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

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

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

- **Flags**

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

➢ Intent Filter

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

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

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

→ **Syntax:**

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

→ **Contained in:**

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

→ **attributes:**

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

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

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

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

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

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

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

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

→ **Elements:**

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

attributes:

android:name : The name of the action.

<category>

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

attributes:

android:name : The name of the category.

<data>

Adds a data specification to an intent filter.

attributes:

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

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

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

→ **Example:**

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

Android R.java file

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

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

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

```
public final class R {
   public static final class attr {
   }
   public static final class drawable {
      public static final int ic_launcher=0x7f020000;
   }
   public static final class id {
      public static final int menu_settings=0x7f070000;
   }
   public static final class layout {
      public static final int activity_main=0x7f030000;
   }
   public static final class menu {
      public static final int activity_main=0x7f060000;
   }
```

```
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
        public static final int menu_settings=0x7f040002;
    }
    public static final class style {
        public static final int AppBaseTheme=0x7f050000;
        public static final int AppTheme=0x7f050001;
    }
}
```

AndroidManifest.xml file in android

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

→ It performs some other tasks also:

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

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

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.javatpoint.hello"
    android:versionCode="1"
    android:versionName="1.0" >

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

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

→ **Elements of the AndroidManifest.xml file**
The elements used in the above xml file are described below.

- *<manifest>*

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

- *<application>*

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

- *<activity>*

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

- *<intent-filter>*

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

- *<action>*

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

- *<category>*

It adds a category name to an intent-filter.

- *<uses-permission>*

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

- *<uses-sdk>*

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

- *<uses-library>*

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

- *<permission>*

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

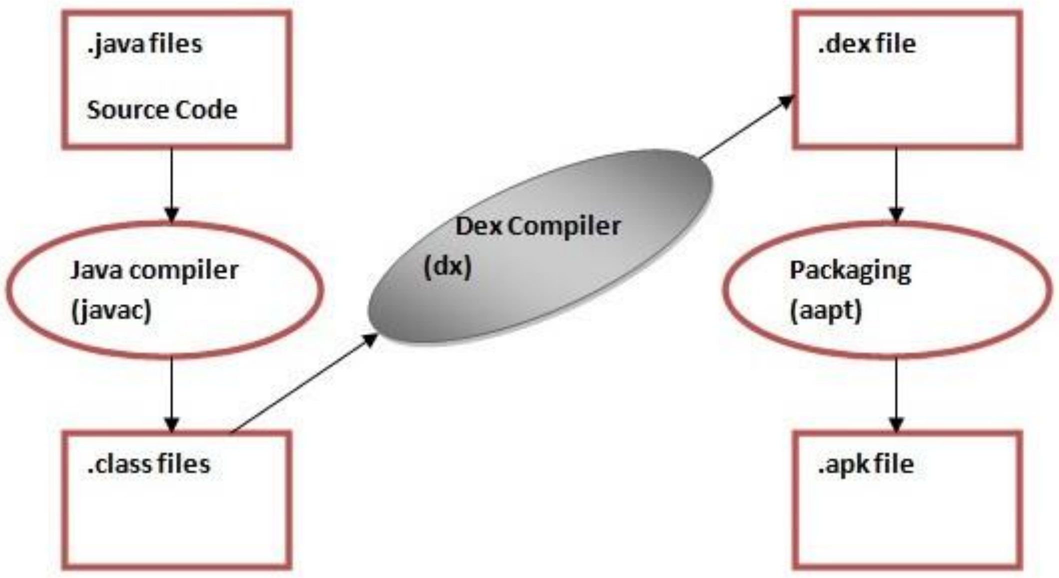
- *<meta-data>*

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

Dalvik Virtual Machine | DVM

→ As we know the modern JVM is high performance and provides excellent memory management. But it needs to be optimized for low-powered handheld devices as well.
→ The Dalvik Virtual Machine (DVM) is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for *memory*, *battery life* and *performance*.
→ Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein.

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



→ The javac tool compiles the java source file into the class file.
→ The dx tool takes all the class files of our application and generates a single .dex file. It is a platform-specific tool.
→ The Android Assets Packaging Tool (aapt) handles the packaging process.

Android Resources Organizing & Accessing

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

➢ Default Resources:

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

Sr.No.	Directory & Resource Type
1	anim/ XML files that define property animations. They are saved in res/anim/ folder and accessed from the R.anim class.
2	color/ XML files that define a state list of colors. They are saved in res/color/ and accessed from the R.color class.
3	drawable/ Image files like .png, .jpg, .gif or XML files that are compiled into bitmaps, state lists, shapes, animation drawable. They are saved in res/drawable/ and accessed from the R.drawable class.
4	layout/

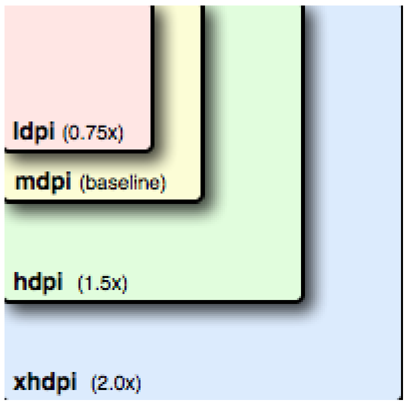
	XML files that define a user interface layout. They are saved in res/layout/ and accessed from the R.layout class.
5	menu/ XML files that define application menus, such as an Options Menu, Context Menu, or Sub Menu. They are saved in res/menu/ and accessed from the R.menu class.
6	raw/ Arbitrary files to save in their raw form. We need to call *Resources.openRawResource()* with the resource ID, which is *R.raw.filename* to open such raw files.
7	values/ XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources we can create in this directory − • arrays.xml for resource arrays, and accessed from the R.array class. • integers.xml for resource integers, and accessed from the R.integer class. • bools.xml for resource boolean, and accessed from the R.bool class. • colors.xml for color values, and accessed from the R.color class. • dimens.xml for dimension values, and accessed from the R.dimen class. • strings.xml for string values, and accessed from the R.string class. • styles.xml for styles, and accessed from the R.style class.
8	xml/ Arbitrary XML files that can be read at runtime by calling *Resources.getXML()*. We can save various configuration files here which will be used at run time.

➢ Alternative Resources

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

→ Common uses for alternate resources include:
  - Alternative layout files for different form factors (i.e phone vs tablet)
  - Alternative string resources for different languages (i.e English vs Italian)
  - Alternative drawable resources for different screen densities (shown below)
  - Alternate style resources for different platform versions (Holo vs Material)
  - Alternate layout files for different screen orientations (i.e portrait vs landscape)

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



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

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

Configuration	Qualifier Values	Description
MCC and MNC	Examples: mcc310 mcc310-mnc004 mcc208-mnc00 etc.	The mobile country code (MCC), optionally followed by mobile network code (MNC) from the SIM card in the device. For example, mcc310 is U.S. on any carrier, mcc310-mnc004 is U.S. on Verizon, and mcc208-mnc00 is France on Orange. If the device uses a radio connection (GSM phone), the MCC and MNC values come from the SIM card. We can also use the MCC alone (for example, to include country-specific legal resources in our app). If we need to specify based on the language only, then use the *language and region* qualifier instead (discussed next). If we decide to use the MCC and MNC qualifier, we should do so with care and test that it works as expected. Also see the configuration fields mcc, and mnc, which indicate the current mobile country code and mobile network code, respectively.
Language and region	Examples: en fr en-rUS fr-rFR fr-rCA b+en b+en+US b+es+419	The language is defined by a two-letter ISO 639-1 language code, optionally followed by a two letter ISO 3166-1-alpha-2 region code (preceded by lowercase r). The codes are *not* case-sensitive; the r prefix is used to distinguish the region portion. We cannot specify a region alone. Android 7.0 (API level 24) introduced support for BCP 47 language tags, which we can use to qualify language- and region-specific resources. A language tag is composed from a sequence of one or more subtags, each of which refines or narrows the range of language identified by the overall tag. For more information about language tags, see Tags for Identifying Languages.

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

		tablets. Thus, we likely care most about what the smallest possible width will be on each device. The smallest width of a device takes into account screen decorations and system UI. For example, if the device has some persistent UI elements on the screen that account for space along the axis of the smallest width, the system declares the smallest width to be smaller than the actual screen size, because those are screen pixels not available for our UI. Some values we might use here for common screen sizes: • 320, for devices with screen configurations such as: • 240x320 ldpi (QVGA handset) • 320x480 mdpi (handset) • 480x800 hdpi (high-density handset) 480, for screens such as 480x800 mdpi (tablet/handset). 600, for screens such as 600x1024 mdpi (7" tablet). 720, for screens such as 720x1280 mdpi (10" tablet). When our app provides multiple resource directories with different values for the smallestWidth qualifier, the system uses the one closest to (without exceeding) the device's smallestWidth. *Added in API level 13.* Also see the android:requiresSmallestWidthDp attribute, which declares the minimum smallestWidth with which our app is compatible, and the smallestScreenWidthDp configuration field, which holds the device's smallestWidth value. For more information about designing for different screens and using this qualifier, see the Supporting Multiple Screens developer guide.
Available width	w<N>dp Examples: w720dp w1024dp etc.	Specifies a minimum available screen width, in dp units at which the resource should be used—defined by the <N> value. This configuration value changes when the orientation changes between landscape and portrait to match the current actual width. This is often useful to determine whether to use a multi-pane layout, because even on a tablet device, we often won't want the same multi-pane layout for portrait orientation as we do for landscape. Thus, we can use this to specify the minimum width required for the layout, instead of using both the screen size and orientation qualifiers together. When our app provides multiple resource directories with different values for this configuration, the system uses the one closest to (without exceeding) the device's current screen width. The value here takes into account screen decorations, so if the device has some persistent UI elements on the left or right edge of the display, it uses a value for the width that is smaller than the real screen size, accounting for these UI elements and reducing the app's available space. *Added in API level 13.* Also see the screenWidthDp configuration field, which holds the current screen width.

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

		qualifiers that better match the current device configuration, the system may use whichever resources are the best match. Caution: If all our resources use a size qualifier that is *larger* than the current screen, the system will not use them and our app will crash at runtime (for example, if all layout resources are tagged with the xlarge qualifier, but the device is a normal-size screen). *Added in API level 4.* See Supporting Multiple Screens for more information. Also see the screenLayout configuration field, which indicates whether the screen is small, normal, or large.
Screen aspect	long notlong	• long: Long screens, such as WQVGA, WVGA, FWVGA • notlong: Not long screens, such as QVGA, HVGA, and VGA *Added in API level 4.* This is based purely on the aspect ratio of the screen (a "long" screen is wider). This isn't related to the screen orientation. Also see the screenLayout configuration field, which indicates whether the screen is long.
Round screen	round notround	• round: Round screens, such as a round wearable device • notround: Rectangular screens, such as phones or tablets *Added in API level 23.* Also see the isScreenRound() configuration method, which indicates whether the screen is round.
Wide Color Gamut	widecg nowidecg	• widecg: Displays with a wide color gamut such as Display P3 or AdobeRGB • nowidecg: Displays with a narrow color gamut such as sRGB *Added in API level 26.* Also see the isScreenWideColorGamut() configuration method, which indicates whether the screen has a wide color gamut.
High Dynamic Range (HDR)	highdr lowdr	• highdr: Displays with a high-dynamic range • lowdr: Displays with a low/standard dynamic range *Added in API level 26.* Also see the isScreenHdr() configuration method, which indicates whether the screen has a HDR capabilities.
Screen orientation	port land	• port: Device is in portrait orientation (vertical) • land: Device is in landscape orientation (horizontal) This can change during the life of our app if the user rotates the screen. See Handling Runtime Changes for information about how this affects our app during runtime.

		Also see the orientation configuration field, which indicates the current device orientation.
UI mode	car desk television appliance watch vrheadset	• car: Device is displaying in a car dock • desk: Device is displaying in a desk dock • television: Device is displaying on a television, providing a "ten foot" experience where its UI is on a large screen that the user is far away from, primarily oriented around DPAD or other non-pointer interaction • appliance: Device is serving as an appliance, with no display • watch: Device has a display and is worn on the wrist • vrheadset: Device is displaying in a virtual reality headset *Added in API level 8, television added in API 13, watch added in API 20.* For information about how our app can respond when the device is inserted into or removed from a dock, read Determining and Monitoring the Docking State and Type. This can change during the life of our app if the user places the device in a dock. We can enable or disable some of these modes using UiModeManager. See Handling Runtime Changes for information about how this affects our app during runtime.
Night mode	night notnight	• night: Night time • notnight: Day time *Added in API level 8.* This can change during the life of our app if night mode is left in auto mode (default), in which case the mode changes based on the time of day. We can enable or disable this mode using UiModeManager. See Handling Runtime Changes for information about how this affects our app during runtime.
Screen pixel density (dpi)	ldpi mdpi hdpi xhdpi xxhdpi xxxhdpi nodpi tvdpi anydpi *nnn*dpi	• ldpi: Low-density screens; approximately 120dpi. • mdpi: Medium-density (on traditional HVGA) screens; approximately 160dpi. • hdpi: High-density screens; approximately 240dpi. • xhdpi: Extra-high-density screens; approximately 320dpi. *Added in API Level 8* • xxhdpi: Extra-extra-high-density screens; approximately 480dpi. *Added in API Level 16* • xxxhdpi: Extra-extra-extra-high-density uses (launcher icon only, see the note in *Supporting Multiple Screens*); approximately 640dpi. *Added in API Level 18* • nodpi: This can be used for bitmap resources that we don't want to be scaled to match the device density. • tvdpi: Screens somewhere between mdpi and hdpi; approximately 213dpi. This isn't considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system scales them as appropriate. *Added in API Level 13* • anydpi: This qualifier matches all screen densities and takes precedence over other qualifiers. This is useful for vector drawables. *Added in API Level 21*

		• *nnn*dpi: Used to represent non-standard densities, where *nnn* is a positive integer screen density. This shouldn't be used in most cases. Use standard density buckets, which greatly reduces the overhead of supporting the various device screen densities on the market. There is a 3:4:6:8:12:16 scaling ratio between the six primary densities (ignoring the tvdpi density). So, a 9x9 bitmap in ldpi is 12x12 in mdpi, 18x18 in hdpi, 24x24 in xhdpi and so on. If we decide that our image resources don't look good enough on a television or other certain devices and want to try tvdpi resources, the scaling factor is 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi. Note: Using a density qualifier doesn't imply that the resources are *only* for screens of that density. If we don't provide alternative resources with qualifiers that better match the current device configuration, the system may use whichever resources are the best match. See Supporting Multiple Screens for more information about how to handle different screen densities and how Android might scale our bitmaps to fit the current density.
Touchscreen type	notouch finger	• notouch: Device doesn't have a touchscreen. • finger: Device has a touchscreen that is intended to be used through direction interaction of the user's finger. Also see the touchscreen configuration field, which indicates the type of touchscreen on the device.
Keyboard availability	keysexposed keyshidden keyssoft	• keysexposed: Device has a keyboard available. If the device has a software keyboard enabled (which is likely), this may be used even when the hardware keyboard *isn't* exposed to the user, even if the device has no hardware keyboard. If no software keyboard is provided or it's disabled, then this is only used when a hardware keyboard is exposed. • keyshidden: Device has a hardware keyboard available but it is hidden *and* the device does *not* have a software keyboard enabled. • keyssoft: Device has a software keyboard enabled, whether it's visible or not. If we provide keysexposed resources, but not keyssoft resources, the system uses the keysexposed resources regardless of whether a keyboard is visible, as long as the system has a software keyboard enabled. This can change during the life of our app if the user opens a hardware keyboard. See Handling Runtime Changes for information about how this affects our app during runtime. Also see the configuration fields hardKeyboardHidden and keyboardHidden, which indicate the visibility of a hardware keyboard and the visibility of any kind of keyboard (including software), respectively.
Primary text input method	nokeys qwerty 12key	• nokeys: Device has no hardware keys for text input. • qwerty: Device has a hardware qwerty keyboard, whether it's visible to the user or not.

		• 12key: Device has a hardware 12-key keyboard, whether it's visible to the user or not. Also see the keyboard configuration field, which indicates the primary text input method available.
Navigation key availability	navexposed navhidden	• navexposed: Navigation keys are available to the user. • navhidden: Navigation keys aren't available (such as behind a closed lid). This can change during the life of our app if the user reveals the navigation keys. See Handling Runtime Changes for information about how this affects our app during runtime. Also see the navigationHidden configuration field, which indicates whether navigation keys are hidden.
Primary non-touch navigation method	nonav dpad trackball wheel	• nonav: Device has no navigation facility other than using the touchscreen. • dpad: Device has a directional-pad (d-pad) for navigation. • trackball: Device has a trackball for navigation. • wheel: Device has a directional wheel(s) for navigation (uncommon). Also see the navigation configuration field, which indicates the type of navigation method available.
Platform Version (API level)	Examples: v3 v4 v7 etc.	The API level supported by the device. For example, v1 for API level 1 (devices with Android 1.0 or higher) and v4 for API level 4 (devices with Android 1.6 or higher). See the Android API levels document for more information about these values.

➢ Qualifier name rules

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

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

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

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

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

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

Creating alias resources

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

→ **Drawable**

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

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

→ **Layout**

- To create an alias to an existing layout, use the <include> element, wrapped in a <merge>.
- For example:

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

→ **Strings and other simple values**

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

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

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

- For example, a color:

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

Accessing Resources:

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

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

→ Although the R class is where resource IDs are specified, we should never need to look there to discover a resource ID. A resource ID is always composed of:
  - The *resource type*: Each resource is grouped into a "type," such as string, drawable, and layout. For more about the different types, see Resource Types.
  - The *resource name*, which is either: the filename, excluding the extension; or the value in the XML android:name attribute, if the resource is a simple value (such as a string).

→ There are two ways we can access a resource:
  - **In code:** Using a static integer from a sub-class of our R class
  - **In XML:** Using a special XML syntax that also corresponds to the resource ID defined in our R class.

➢ Accessing Resource in Code:

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

→ Syntax

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

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

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

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

➢ Accessing Resource in XML:

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

→ Syntax

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

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

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

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

➢ Referencing style attributes

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

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

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

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

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

➢ Accessing original files

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

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

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

➢ Accessing platform resources

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

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

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

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