

# 1. Introduction to Mobile OS

## 1.1 – Introduction to Mobile OS

A **mobile operating system** (or **mobile OS**) is an operating system specifically designed for mobile devices like mobile phones, smartphones, PDAs, tablet computers and other handheld devices.

Much like the Linux or Windows operating system controls your desktop or laptop computer, a mobile operating system is the software platform on top of which other programs or applications can run on mobile devices. The operating system is responsible for determining the function and features available on your devices, such as Touchscreen, Cellular, Bluetooth, Wifi, GPS, Camera and many more. There are various mobile operating system available in the market. Some of them are open source and some of them are proprietary based. Currently there are few operating system dominating the market of which Android and iOS occupies the 88% and 11.9% respectively (*src from statista.com on end of 2018*). There are other operating system as well like Microsoft's windows for phone and blackberry but both of them are now nearly out of market as they couldn't compete with Google's Android and Apple inc's iOS.

### Types of Mobile Operating System:

#### a. Android

Android is a mobile operating system that is based on a modified version of Linux. It was originally developed by a startup of the same name, Android, Inc. In 2005, as part of its strategy to enter the mobile space, Google purchased Android, Inc. and took over its development work (as well as its development team).

Google wanted the Android OS to be open and free, so most of the Android code was released under the open source Apache License. That means anyone who wants to use Android can do so by downloading the full Android source code. Android is the by far most dominating mobile OS in market with 88% share occupied in the market as of end of 2018.

#### b. iOS

iOS (formerly iPhone OS) is a mobile operating system created and developed by Apple Inc. exclusively for its hardware. It is the operating system that presently powers many of the company's mobile devices, including the iPhone, iPad and iPod Touch. It is the second most popular mobile operating system globally after Android.

Originally unveiled in 2007 for the iPhone, iOS has been extended to support other Apple devices such as the iPod Touch(September 2007) and the iPad(January 2010).

iOS use XNU kernel at its core. XNU may also be referred to as the "OS X kernel" or "iOS kernel". XNU stands for "X is Not Unix". XNU was first made by NeXT(*cofounded by Steve Jobs*) in 1989 for their operating system called NeXTSTEP. On December 20, 1996, Apple Inc. purchased NeXT and its software. Afterwards, the XNU kernel was used to make OS X and related operating systems. Apple licensed XNU under the Apple Public Source License v2.0 (APSL). Therefore, XNU is open source.

A kernel is the core of an operating system. The operating system is all of the programs that manages the hardware and allows users to run applications on a computer. The kernel controls the hardware and applications. Applications do not communicate with the hardware directly, instead they go to the kernel. In summary, software runs on the kernel and the kernel operates the hardware. Without a kernel, a computer is a useless object.

XNU is a hybrid kernel (like Windows); although XNU is primarily monolithic (like Linux). This means that it uses concepts from both microkernel and monolithic kernels, but mainly monolithic ideas. The XNU kernel is a combination of the v3 Mach kernel, FreeBSD kernel, and a C++ API called I/O Kit. Because XNU uses parts from Mach, the kernel can run as separate processes. XNU itself is written in C/C++

### **c. BlackBerry**

BlackBerry OS is a proprietary mobile OS developed by BlackBerry Ltd for BlackBerry line of smartphone handheld devices. The operating system provided multitasking and supports specialized input devices that have been adopted by BlackBerry Ltd. For use in its handhelds, particularly the track wheel, trackball, and most recently, the trackpad and touchscreen. The BlackBerry platform is perhaps best known for its native support for corporate email, through MIDP 1.0 and, more recently, a subset of MIDP 2.0, which allows complete wireless activation and synchronization with Microsoft Exchange, Lotus Domino, or Novell GroupWise email, calendar, tasks, notes, and contacts, when used with BlackBerry Enterprise Server. The operating system also supports WAP 1.2

### **c. Symbian**

Symbian is a closed-source mobile operating system (OS) and computing platform designed for smartphones. Symbian was originally developed by Symbian Ltd. The current form of Symbian is an open-source platform developed by Symbian Foundation in 2009, as the successor of the original Symbian OS. Symbian was used by many major mobile phone brands, like Samsung, Motorola, Sony and above all by Nokia. It was the most popular smartphone OS on a worldwide average until the end of 2019, when it was overtaken by Android.

Symbian OS was created with three systems design principles in mind:

1. the integrity and security of user data is paramount
2. user time must not be wasted

3. all resources are scarce

#### **d. Ubuntu Touch**

Ubuntu Touch (also known as Ubuntu Phone) is a mobile version of the Ubuntu operating system developed by Canonical UK Ltd and Ubuntu community. It is designed primarily for touch screen mobile devices such as smartphone and tabled computers.

The Ubuntu Touch project was started in 2011 and released to manufactures on 16 September 2014.

#### **e. FireFox OS**

Firefox OS is a discontinued open-source operating system made for smartphones, tablet computers and smart TVs – designed by Mozilla and external contributors. It is based on the rendering engine of the Firefox web browser, Gecko, and on the Linux kernel. It was first commercially released in 2013 and later was discontinued in January 2017.

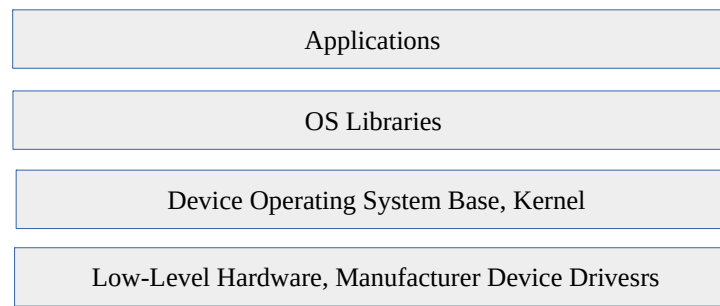
Kernel used in Firefox OS is similar to Android ie. Linux Kernel. Firefox OS used the Gecko engine on top of Linux kernel to render the screen output. Apps were written using HTML5, CSS and JavaScript all three being cooperative languages used in making internet webpages. In essence Apps on Firefox OS were web apps and the OS could be thought of as a Web browser that stored content off-line.

#### **f. Tizen OS**

Tizen is a Linux-based mobile operating system backed by the Linux Foundation but developed and used primarily by Samsung Electronics. The project was originally conceived as an HTML5-based platform for mobile devices to succeed MeeGo. Samsung merged its previous Linux-based OS effort, Bada into Tizen, and has since used it primarily on platforms such as wearable devices and smart TVs.

Much of Tizen is open source software, although the software development kit contains proprietary components owned by Samsung, and portions of the OS are licensed under the Flora License – a derivative of the Apache License that only grants a patent license to “Tizen certified platforms”

## 1.2 Structure of Mobile OS



### **Low Level Hardware:**

Low Level Hardware includes mobile processor and memory. Mobile processor use RISC architecture which are ARM Processors and MIPS. ARM processors is a 32 bit RISC architecture and consumes low power, Qualcomm Snapdragon is the example of ARM processors. MIPS (Microprocessor without Interlocked Pipeline Stages) is mainly used in embedded systems like Playstation 1 and 2, routers and gateways.

### **Kernel:**

It is responsible for services such as security, memory management, Process management. It includes the I/O components, File Systems, Networking Components.

### **Libraries:**

It is for providing libraries for the applications. It includes the Media libraries, 3D libraries, Audio libraries and Graphics Libraries etc.

### **Applications:**

It provides the set of applications to the mobile operating system. Libraries are exposed to developers through application frameworks.

## 1.3 Introduction to Development Environment

### Native Apps

Native apps are typically written in the native language of the device's operating system, primarily Android, iOS and Windows. Each platform provides a different toolset for developers, including the materials needed for marketplace submission and listing.

Native mobile apps provide the best overall user experience with multi-touch functionality, fast graphics API, fluid animation and built-in components. If your top priority is having a stunning user interface, then native is certainly your best bet. However, you'll pay substantially more in development and maintenance.

One of the biggest problems with native apps is making them available on a variety of platforms. Large portions of the app may need to be rewritten in the native language of each platform you are targeting. HTML5 runs everywhere a browser does, and does not require downloading and learning a new toolset per platform.

Undoubtedly, native development is best for gaming and other applications that require speedy performance and rely mostly on interactivity.

The truth is, there's no right or wrong answer when it comes to choosing how you're going to develop your mobile app. Each option has advantages and disadvantages, often determined by your desired functionality, budget, target group and timing.

### Pros of Native Apps

- **Better performance:** A native app has complete access to a phone's hardware resources. The app also interacts directly with the phone without the mediation of a web browser. This lends it significantly better performance, particularly when rendering graphics and animation.
- **Easier development:** Developers can easily leverage platform SDKs to create native apps. This makes for easier development, provided you already have access to experienced engineering talent.
- **Better distribution:** Native apps can be distributed directly through relevant app stores. This can be an incredibly powerful distribution channel that not only makes app installation and updates easy, but can also help cut down on marketing costs through app stores' built-in discovery features.
- **Better monetization:** Since native apps are associated with app stores, they can take advantage of the store's built-in monetization features, such as one-click payments.

### Cons of Native Apps

- **Increased development time and costs:** Since developers must build separate apps for each platform, total development time might be significantly higher than an equivalent HTML5 app. This also increases costs since developing for different platforms demands mastery over different languages and development environments. App developers themselves tend

to be more expensive than their HTML5/JavaScript counterparts. Higher maintenance costs: From the above, it follows that maintenance costs for native apps are also higher. A business might require the services of separate iOS, Android, and Windows Phone developers to keep its apps up and running, eating into profits.

- App store content restriction: Every app distributed through the app store must adhere to strict content guidelines. For certain businesses (such as gambling), this may be a deal-breaker.
- App store fees: App store fees (up to 30 percent) can eat into your profit margins and make an already costly undertaking prohibitively expensive.

## **HTML5 Apps**

In recent years, HTML5 has emerged as a popular way for building mobile applications. If you or one of your colleagues already has experience in web development, then HTML5 should be relatively easy to grasp. An HTML5 mobile app is essentially a web page (or multiple web pages) that is designed to work on a mobile-sized screen. As a result, HTML5 apps can be opened with most current mobile browsers.

HTML5 works across mobile platforms, so you won't have to build different apps for different mobile technologies. Additionally, HTML5 is cost-effective, fast to market and SEO-friendly, all of which could have tremendous benefits to your business. On the other hand, HTML5 has fewer offline capabilities and simply can't compete with a native user experience.

### **Pros of HTML5 Apps**

- Platform independent: HTML5 apps are completely platform independent. It does not matter whether your users are on Android, iOS, or Windows Phone, they can all access your app as long as they have a web browser.
- Easier updates: Native apps update much like desktop applications – the user has to download each update individually. HTML5 apps, on the other hand, support centralised updates. Every time the user visits your app through her/his browser, she/he sees the latest version of your app with no additional download requirements.
- Faster development: HTML5 apps are written in HTML, CSS, JavaScript, and server-side languages such as ASP.NET. This cuts down on development time since there are already vast libraries for these languages (such as jQuery for JavaScript) and a huge body of trained developers.
- Cheaper: Since HTML5 apps are built on relatively simple web technologies such as HTML5 and JavaScript, you will find it easier to hire affordable HTML5 developers than similarly talented native app developers.
- No content restrictions: Since HTML5 apps need not seek approval from a closed app store, they can carry whatever content you want.
- No fees: HTML5 apps are delivered directly through the browser. This cuts out the app store and saves you on the 30 percent app store fees.
- Better data: Any customer data you collect through a native app is subject to the rules of the app store in question. There are no such restrictions with HTML5 apps; you can collect whatever customer data you want. In fact, the Apple App Store withholding subscriber data was one of the reasons why Financial Times switched to a web app.

- Distribution through app store: Until a couple of years ago, there was no easy method to distribute a HTML5 app through the app store. However, new frameworks such as Phonegap and Trigger.io have eased the distribution problem considerably bundling up HTML5 apps as native apps (which can then be distributed through the app store). Although performance remains a concern and Phonegap still doesn't support every mobile OS fully, it's a capable alternative to building separate native apps for every platform.

### **Cons of HTML5 Apps**

- Poorer performance: An HTML5 app has limited access to a phone's hardware. This leads to poorer performance, especially when dealing with heavy graphics, although new platforms such as Famous are trying to mitigate this problem.
- Device fragmentation: Device fragmentation within web browsers is a real thing. Different devices might render the same app differently. This means your developers will need extensive testing and fine-tuning to get the UI right, especially when working on more complex apps.
- Technical limitations: An HTML5 does not have complete access to a device's features and events. This poses a limit on what you can do with an HTML5 app. The UI options for HTML5 apps are also limited.
- Limited monetization opportunities: A native app can make money through direct app sales and in-app purchases. These monetization options are not available for HTML5 apps. This can be particularly challenging for "freemium" apps.

### **Hybrid App**

Hybrid mobile application development use a combination of Native and HTML5 development methodologies to develop applications that have the advantages of both. A native container is used to develop hybrid apps while programming languages used are CSS Javascript and HTML5. These apps are available on different platforms and can be downloaded from the respective app stores.

Since users demand the same apps on different platforms, businesses are increasingly switching over to this methodology.

Hybrid apps are faster to develop and also provide a great user experience. They definitely bring the benefits of both Native and HTML5 methodologies.

### **Pros of Hybrid Apps**

- These apps do not require approval and can be updated while the users are using them
- The application is consistent across platforms
- Common web analytics tools can be used to analyze the traffic on hybrid apps
- Reduced costs and time

## **Cons of Hybrid App**

- Hybrid apps cannot benefit from the upgrade features of the respective platforms
- Creating large and heavy apps (like gaming) is difficult to manage with the Hybrid model
- The UX quality of Hybrid apps does not match that of Native

## **1.5 Introduction to Android**

Android is a mobile operating system that is based on a modified version of Linux. It was originally developed by a startup of the same name, Android, Inc. In 2005, as part of its strategy to enter the mobile space, Google purchased Android, Inc. and took over its development work (as well as its development team).

Google wanted the Android OS to be open and free, so most of the Android code was released under the open source Apache License. That means anyone who wants to use Android can do so by downloading the full Android source code. Moreover, vendors (typically hardware manufacturers) can add their own proprietary extensions to Android and customize Android to differentiate their products from others. This development model makes Android very attractive to vendors, especially those companies affected by the phenomenon of Apple's iPhone, which was a hugely successful product that revolutionized the smartphone industry. When the iPhone was launched, many smart- phone manufacturers had to scramble to find new ways of revitalizing their products. These manufacturers saw Android as a solution, meaning they will continue to design their own hardware and use Android as the operating system that powers it. Some companies that have taken advantage of Android's open source policy include Motorola and Sony Ericsson, which have been developing their own mobile operating systems for many years.

The main advantage to adopting Android is that it offers a unified approach to application development. Developers need only develop for Android in general, and their applications should be able to run on numerous different devices, as long as the devices are powered using Android. In the world of smartphones, applications are the most important part of the success chain.

### **API level**

API Level is an integer value that uniquely identifies the framework API revision offered by a version of the Android platform.

The Android platform provides a framework API that applications can use to interact with the underlying Android system. The framework API consists of:

- A core set of packages and classes
- A set of XML elements and attributes for declaring a manifest file
- A set of XML elements and attributes for declaring and accessing resources
- A set of Intents



- A set of permissions that applications can request, as well as permission enforcements included in the system

| Platform Version          | API Level | VERSION_CODE           |
|---------------------------|-----------|------------------------|
| Android 9                 | 28        | P                      |
| Android 8.1               | 27        | O_MR1                  |
| Android 8.0               | 26        | O                      |
| Android 7.1.1             | 25        | N_MR1                  |
| Android 7.1               |           |                        |
| Android 7.0               | 24        | N                      |
| Android 6.0               | 23        | M                      |
| Android 5.1               | 22        | LOLLIPOP_MR1           |
| Android 5.0               | 21        | LOLLIPOP               |
| Android 4.4W              | 20        | KITKAT_WATCH           |
| Android 4.4               | 19        | KITKAT                 |
| Android 4.3               | 18        | JELLY_BEAN_MR2         |
| Android 4.2, 4.2.2        | 17        | JELLY_BEAN_MR1         |
| Android 4.1, 4.1.1        | 16        | JELLY_BEAN             |
| Android 4.0.3, 4.0.4      | 15        | ICE_CREAM_SANDWICH_MR1 |
| Android 4.0, 4.0.1, 4.0.2 | 14        | ICE_CREAM_SANDWICH     |
| Android 3.2               | 13        | HONEYCOMB_MR2          |
| Android 3.1.x             | 12        | HONEYCOMB_MR1          |
| Android 3.0.x             | 11        | HONEYCOMB              |
| Android 2.3.4             |           |                        |
| Android 2.3.3             | 10        | GINGERBREAD_MR1        |
| Android 2.3.2             |           |                        |
| Android 2.3.1             | 9         | GINGERBREAD            |
| Android 2.3               |           |                        |
| Android 2.2.x             | 8         | FROYO                  |
| Android 2.1.x             | 7         | ECLAIR_MR1             |
| Android 2.0.1             | 6         | ECLAIR_0_1             |
| Android 2.0               | 5         | ECLAIR                 |
| Android 1.6               | 4         | DONUT                  |
| Android 1.5               | 3         | CUPCAKE                |
| Android 1.1               | 2         | BASE_1_1               |
| Android 1.0               | 1         | BASE                   |

## Pros and Cons of Android

### Pros:

- Android is open source Operating System and is customizable.
- Android has large number of OEM's (Original Equipment Manufacturers)
- It solves user's problems through Google support.
- Android has large number of apps in its play store.

**Cons:**

- Not as secure as iOS and windowsOS.
- Too many apps creating mess in the store, heavy OS in itself
- Most of the apps are dolphin browsers, wallpapers, malwares (considerable part) or are not even downloaded once.
- App compatibility issues for various versions of android. Not all phones run the same version at a given time.
- Over time phones become slow in operation.
- Quickly eat up the battery

**Comparison of Android with iOS***(only remaining true competitor and this comparison is biased towards Android as we are not learning iOS in this class ... :) )*

| Android   | iOS   |
|---|---|
| - Open source Operating System  | - Closed source or Proprietor based Operating System  |
| - familiar programming language(Java, now Kotlin)   | - new and less familiar programming language(Objective-C, now Swift)                              |
| - free developer tools and less costly to publish app on store(charges only 25\$ as one time payment for account) | - expensive for development as developers needs to pay 100\$ annually as developer account charge |
| - occupies more than 88% of mobile market, so big market for apps   | - occupies only about 11.8% of mobile market, so comparatively smaller market for apps            |

## 1.6 Android Architecture

To understand how Android works, take a look at Figure 1-1, which shows the various layers that make up the Android operating system (OS).

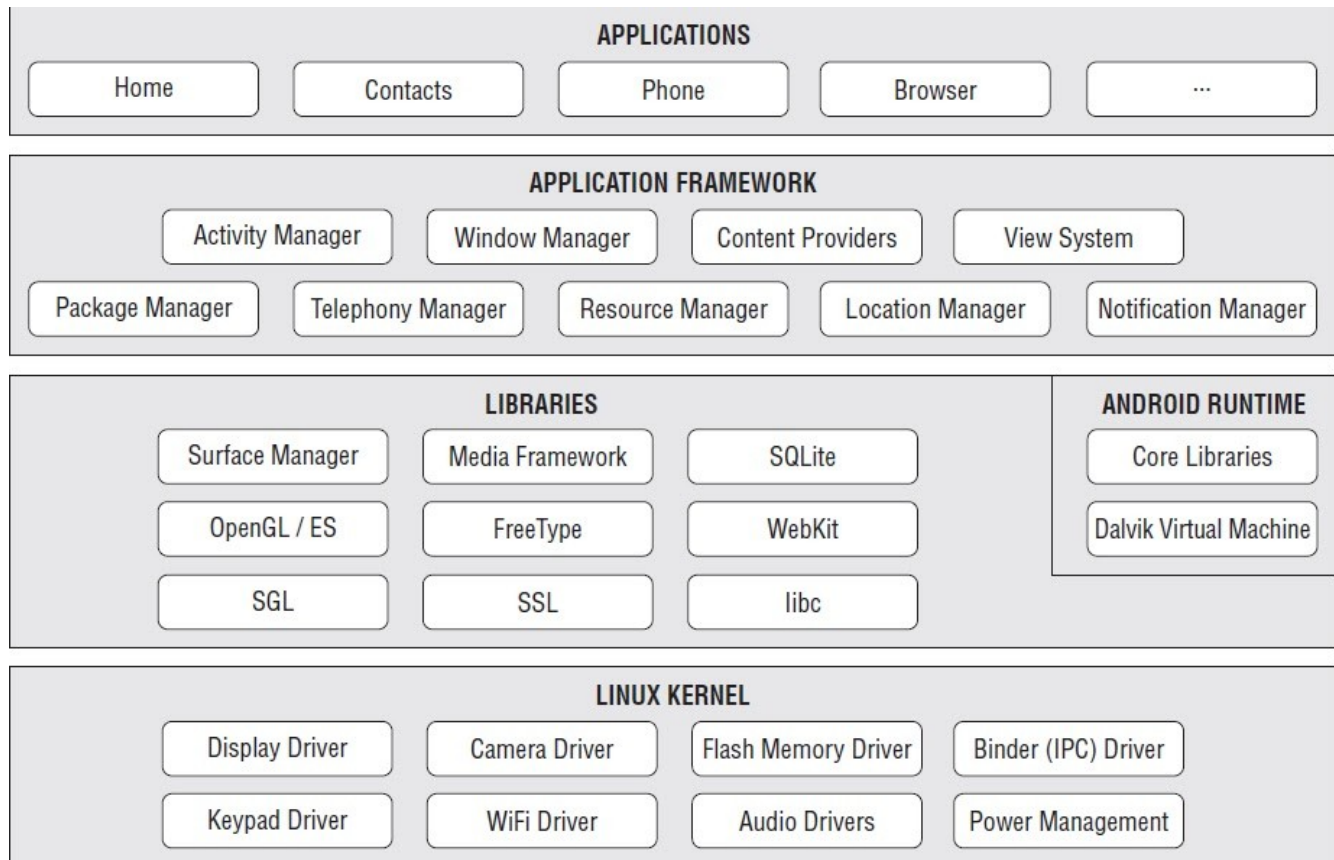


Figure 1-1

The Android OS is roughly divided into five sections in four main layers:

- **Linux kernel**—This is the kernel on which Android is based. This layer contains all the low-level device drivers for the various hardware components of an Android device.
- **Libraries**—These contain the code that provides the main features of an Android OS. For example, the SQLite library provides database support so that an application can use it for data storage. The WebKit library provides functionalities for web browsing.
- **Android runtime**—The Android runtime is located in the same layer with the libraries and provides a set of core libraries that enable developers to write Android apps using the Java programming language. The Android runtime also includes the Dalvik virtual machine, which enables every Android application to run in its own process, with its own instance

of the Dalvik virtual machine. (Android applications are compiled into Dalvik executables). Dalvik is a specialized virtual machine designed specifically for Android and optimized for battery-powered mobile devices with limited memory and CPU power.

- **Application framework**—The application framework exposes the various capabilities of the Android OS to application developers so that they can make use of them in their applications.
- **Applications**—At this top layer are the applications that ship with the Android device (such as Phone, Contacts, Browser, and so on), as well as applications that you download and install from the Android Market. Any applications that you write are located at this layer.

## 1.7 The Linux Kernel

Positioned at the bottom of the Android software stack, the Linux Kernel provides level of abstraction between the devices hardware and the upper layers of the Android software stack. Based on Linux version 2.6, the kernel provides preemptive multitasking, low-level core system services such as memory, process and power management in addition to providing a network stack and device drives for hardware such as the device display, Wi-Fi- and audio.

The original Linux kernel was developed in 1991 by Linus Torvalds and was combined with a set of tools, utilities and compilers developed by Richard Stallman at the Free Software Foundation to create a full operating system referred to as GNU/Linux. Various Linux distributions have been derived from these basic underpinnings such as Ubuntu and Red Hat Enterprise Linux.

It is important to note, however, that Android uses only the Linux kernel. That said, it is worth noting that the Linux kernel was originally developed for use in traditional computers in the form of desktops and servers. In fact, Linux is now most widely deployed in mission critical enterprise server environments. It is a testament to both the power of today's mobile devices and the efficiency and performance of the Linux kernel that we find this software at the heart of the Android software stack.

## 1.8 Android VM and Runtime (ART & Dalvik)

### Why Android use Virtual Machine?

Android makes use of a virtual machine as its runtime environment in order to run the APK files that constitute an Android application. Below are the advantages:

- The application code is isolated from the core OS. So even if any code contains some malicious code won't directly affect the system files. It makes the Android OS more stable and reliable.

- It provides cross compatibility or platform independency. It meaning even if an app is compiled on platform such as a PC, it can still be executed on the mobile platform using the virtual machine.

## **Android Runtime**

In a simplest term runtime is a system used by operating system which takes care of converting the code that you write in a high level language like Java to machine code and understand by CPU/Processor.

Runtime comprises of software instructions that execute when your program is running, even if they're not essentially are a part of the code of that piece of software in particular.

CPUs or more general term our computers understand only machine language (binary codes) so to make it run on CPU, the code must be converted to machine code, which is done by translator.

## **How Android code execution works?**

In Android Java classes converted into DEX bytecode. The DEX bytecode format is translated to native machine code via either ART or the Dalvik runtimes. Here DEX bytecode is independent of device architecture.

Dalvik is a JIT (Just in time) compilation based engine. There were drawbacks to use Dalvik hence from Android 4.4 (kitkat) ART was introduced as a runtime and from Android 5.0 (Lollipop) it has completely replaced Dalvik. Android 7.0 adds a just-in-time (JIT) compiler with code profiling to Android runtime (ART) that constantly improves the performance of Android apps as they run.

**Key Point:** Dalvik used JIT (Just in time) compilation whereas ART uses AOT (Ahead of time) compilation.

## **Just In Time (JIT)**

With the Dalvik JIT compiler, each time when the app is run, it dynamically translates a part of the Dalvik bytecode into machine code. As the execution progresses, more bytecode is compiled and cached. Since JIT compiles only a part of the code, it has a smaller memory footprint and uses less physical space on the device.

## **Ahead Of Time (AOT)**

ART is equipped with an Ahead-of-Time compiler. During the app's installation phase, it statically translates the DEX bytecode into machine code and stores in the device's storage. This is a one-time

event which happens when the app is installed on the device. With no need for JIT compilation, the code executes much faster.

As ART runs app machine code directly (native execution), it doesn't hit the CPU as hard as just-in-time code compiling on Dalvik. Because of less CPU usage results in less battery drain.

ART also uses same DEX bytecode as input for Dalvik. An application compiled using ART requires additional time for compilation when an application is installed and take up slightly larger amounts of space to store the compiled code.

## **Why Android use Virtual Machine?**

Android makes use of a virtual machine as its runtime environment in order to run the APK files that constitute an Android application. Below are the advantages:

- The application code is isolated from the core OS. So even if any code contains some malicious code won't directly affect the system files. It makes the Android OS more stable and reliable.
- It provides cross compatibility or platform independency. It meaning even if an app is compiled on platform such as a PC, it can still be executed on the mobile platform using the virtual machine.

## **Benefits of ART**

- Apps run faster as DEX bytecode translation done during installation.
- Reduces startup time of applications as native code is directly executed.
- Improves battery performance as power utilized to interpreted byte codes line by line is saved.
- Improved garbage collector.
- Improved developer tool.

## **Drawbacks of ART**

- App Installation takes more time because of DEX bytecodes conversion into machine code during installation.
- As the native machine code generated on installation is stored in internal storage, more internal storage is required.

*[Note : Refer to images of ART and Dalvik from internet to understand with much details]*

## 1.9 Installation and Configuration of Android SDKs and Android Studio IDE

For the installation of Android Studio and for configuration of Android SDKs.

Refer to <https://developer.android.com/studio/install>

*If any issue arises while installing Android Studio concern with your teacher as soon as possible as further chapters requires a perfectly operating Android Studio to workout with examples.*

## 1.10 Using ADB command line interfaces

The primary purpose of Android Debug Bridge (ADB) is to facilitate interaction between a development system, in this case Android Studio, and both AVD emulators and physical Android devices for the purpose of running and debugging applications.

The ADB consists of a client, a server process running in background on the development system and a daemon background process running in either AVDs or real Android devices such as phones and tablets.

The ADB client can take a variety of forms. For example, a client is provided in the form of a command-line tool named adb located in the Android SDK platform-tools sub-directory. Similarly, Android Studio also has a built-in client.

A variety of tasks may be performed using the adb command-line tool. For example, a listing of currently active virtual or physical devices may be obtained using the devices command-line argument. The following command output indicates the presence of an AVD on the system but no physical devices:

```
$ adb devices
```

```
List of devices attached
```

```
emulator-5554    device
```

Some other commands are:

```
$ adb kill-server (to kill adb server)
```

```
$ adb start-server (to start adb server)
```

```
$ adb -s emulator-5554 shell (for accessing root access to directory of virtual devices)
```

*Note : At the end of this chapter you should be familiar with:*

- *Types of Mobile OS*
- *Android OS, its architecture and kernel*
- *Native, HTML and Hybrid apps*
- *Android virtual machine and Runtime (Dalvik, ART, JIT, AOT)*
- *Android Studio, SDK and adb*

## 2. Java Architecture and OOPS

### Compilation in Java

Java combines both the approaches of compilation and interpretation. First, java compiler compiles the source code into byte code. At the run time, Java Virtual Machine (JVM) converts this byte code and generates machine code which will be directly executed by the machine in which java program runs.

### Java Virtual Machine (JVM)

JVM is a component which provides an environment for running Java programs. JVM converts the byte code into machine code which will be executed the machine in which the Java program runs.

### Why Java is Platform Independent?

Platform independence is one of the main advantages of Java. In another words, java is portable because the same java program can be executed in multiple platforms without making any changes in the source code. You just need to write the java code for one platform and the same program will run in any platforms. But how does Java make this possible?

First the Java code is compiled by the Java compiler and generates the byte code. This byte code will be stored in class files. Java Virtual Machine (JVM) is unique for each platform. Though JVM is unique for each platform, all response the same byte code and convert it into machine code required for its own platform and this machine code will be directly executed by the machine in which java program runs. This makes Java platform independent and portable.

### Java Runtime Environment (JRE) and Java Architecture in Detail

Java Runtime Environment contains JVM, class libraries and other supporting components. As you know the Java source code is compiled into byte code by Java compiler. This byte code will be stored in class files. During runtime, this byte code will be loaded, verified and JVM interprets the byte code into machine code which will be executed in the machine in which the Java program runs.

A Java Runtime Environment performs the following main tasks respectively.

1. Loads the class

This is done by the class loader

2. Verifies the byte code

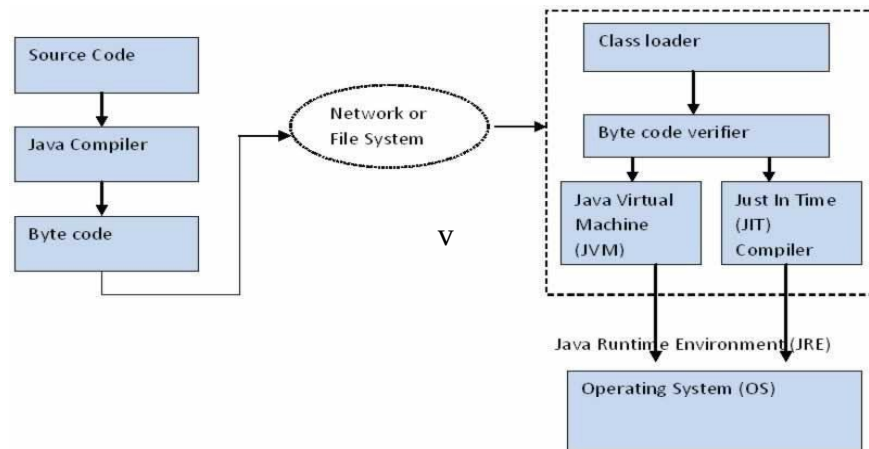
This is done by byte code verifier.

3. Interprets the byte code



This is done by the JVM

These tasks are described in detail in the subsequent sessions. A detailed Java architecture can be drawn as given below.



### Class loader

Class loader loads all the class files required to execute the program. Class loader makes the program secure by separating the namespace for the classes obtained through the network from the classes available locally. Once the byte code is loaded successfully, then next step is byte code verification by byte code verifier.

### Byte code verifier

The byte code verifier verifies the byte code to see if any security problems are there in the code. It checks the byte code and ensures the followings.

1. The code follows JVM specifications.
2. There is no unauthorized access to memory.
3. The code does not cause any stack overflows.
4. There are no illegal data conversions in the code such as float to object references.

Once this code is verified and proven that there is no security issues with the code, JVM will convert the byte code into machine code which will be directly executed by the machine in which the Java program runs.

### **Just in Time Compiler**

When the Java program is executed, the byte code is executed by JVM. But this interpretation is a slower process. To overcome this difficulty, JRE include the component JIT compiler. JIT makes the execution faster. If the JIT Compiler library exists, when a particular byte code is executed first time, JIT compiler compiles it into native machine code which can be directly executed by the machine in which the Java program runs. Once the byte code is recompiled by JIT compiler, the execution time needed will be much lesser. This compilation happens when the byte code is about to be executed and hence the name “Just in Time”.

Once the byte code is compiled into that particular machine code, it is cached by the JIT compiler and will be reused for the future needs. Hence the main performance improvement by using JIT compiler can be seen when the same code is executed again and again because JIT make use of the machine code which is cached and stored.

### **Why Java is Secure?**

As you have noticed in the prior session “Java Runtime Environment (JRE) and Java Architecture in Detail”, the byte code is inspected carefully before execution by Java Runtime Environment (JRE). This is mainly done by the “Class loader” and “Byte code verifier”. Hence a high level of security is achieved.

### **Garbage Collection**

Garbage collection is a process by which Java achieves better memory management. Whenever an object is created, there will be some memory allocated for this object. This memory will remain as allocated until there are some references to this object. When there is no reference to this object, Java will assume that this object is not used anymore. When garbage collection process happens, these objects will be destroyed and memory will be reclaimed.

## **2.1 Java Classes and Objects:**

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object- oriented programming.

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes*; and code, in the form of procedures, often known as *methods*. A distinguishing feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated.

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tangible and intangible). The example of intangible object is banking system.

An object has three characteristics:

- **State:**  
represents data (value) of an object.
- **Behavior:**  
represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **Identity:**  
Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Each object is said to be an instance of a particular class (for example, an object with its name field set to "Mary" might be an instance of class Employee). Procedures in object-oriented programming are known as methods, variables are also known as fields, members, attributes, or properties. This leads to the following terms:

- Class variables:  
belong to the *class as a whole*; there is only one copy of each one
- Instance variables or attributes:  
data that belongs to individual *objects*; every object has its own copy of each one
- Member variables:  
refers to both the class and instance variables that are defined by a particular class
- Class methods:  
belong to the *class as a whole* and have access only to class variables and inputs from the procedure call
- Instance methods:

belong to *individual objects*, and have access to instance variables for the specific object they are called on, inputs, and class variables

Objects are accessed somewhat like variables with complex internal structure, and in many languages are effectively pointers, serving as actual references to a single instance of said object in memory within a heap or stack. They provide a layer of abstraction which can be used to separate internal from external code. External code can use an object by calling a specific instance method with a certain set of input parameters, read an instance variable, or write to an instance variable. Objects are created by calling a special type of method in the class known as a constructor. A program may create many instances of the same class as it runs, which operate independently. This is an easy way for the same procedures to be used on different sets of data.

Object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, a significantly different yet

analogous terminology is used to define the concepts of *object* and *instance*.

## 2.2 Class Methods and Instances:

Data is encapsulated in a class by declaring variables inside the class declaration. Variables declared in these scopes are known as **instance variables**. Instance variables are declared in the same way as local variables except that, they are declared outside any particular method.

**Class variables** are global to class and to all the instances of class. They are useful in communicating between different objects of the same class for keeping track of global states.

**Methods** are functions that operate on instances of classes in which they are defined. Objects can communicate with each other using methods and can call methods in other classes.

Method definition has four parts:

- Name of the method
- Type of object
- List of parameters
- Body of method

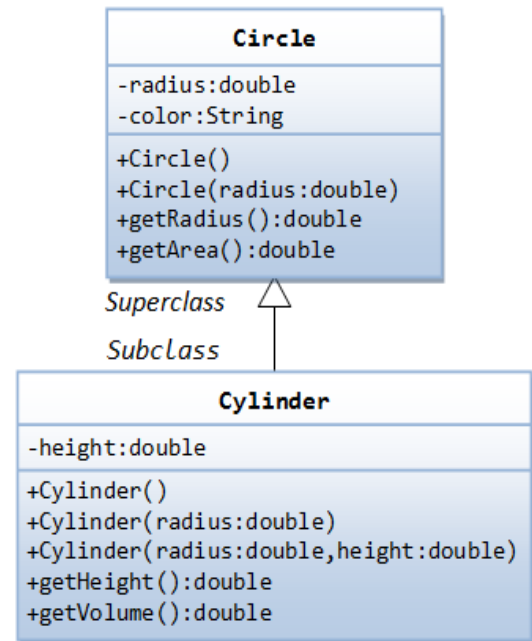
The methods which apply and operate on an instance are called instance methods and the methods which apply and operate on a class are called class methods.

Class methods, like class variables, are available to instances of the class and can be made available to other classes. Class methods can be used anywhere regardless of whether an instance of the class exists or not. Methods that provide some general utility but do not directly affect an instance of the class are declared as **class methods**.

### 2.3.1 Inheritance:

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

In OOP, we often organize classes in hierarchy to avoid duplication and reduce redundancy. The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies. A class in the lower hierarchy is called a subclass (or derived, child, extended class). A class in the upper hierarchy is called a superclass (or base, parent class). By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, redundancy can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,



A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors. It is important to note that a subclass is not a "subset" of a superclass. In contrast, subclass is a "superset" of a superclass. It is because a subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more variables and methods.

In this example, we derive a subclass called Cylinder from the superclass Circle. It is important to note that we reuse the class Circle. The Class Cylinder inherits all the member variables (radius and color) and methods

(getRadius(), getArea(), among others) from its superclass Circle. It further defines a variable called height, two public methods – getHeight() and getVolume() and its own constructors.

### Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

### **a. Public Inheritance:**

When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

### **b. Protected Inheritance:**

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

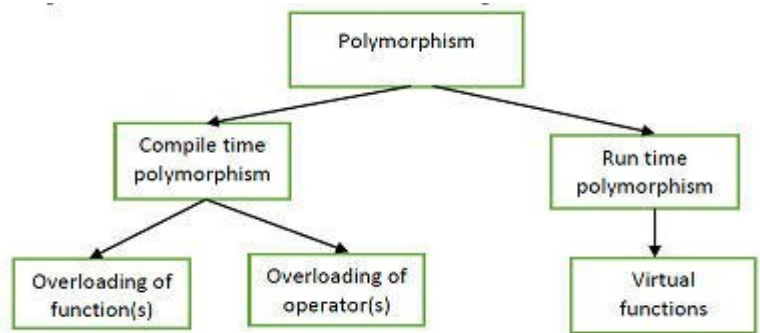
### **c. Private Inheritance:**

When deriving from a private base class, public and protected members of the base class become private members of the derived class.

## **2.3.2 Polymorphism:**

Polymorphism is an important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. Polymorphism means, “One name, multiple forms”.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific action associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.



### **Compile time/Static polymorphism** refers

to the binding of functions on the basis of their signature (number, type and sequence of parameters). It is also called early binding. In this the compiler selects the appropriate function during the compile time. The examples of compile time polymorphism are Function overloading (use the same function name to create functions that perform a variety of different tasks) and Operator overloading (assign multiple meanings to the operators).

### **Dynamic or Subtype or Runtime Polymorphism**

means the change of form by entity depending on the situation. If a member function is selected while the program is running, then it is called run time polymorphism. This feature makes the program more

flexible as a function can be called, depending on the context. This is also called late binding. The example of run time polymorphism is virtual function.

## 2.4 Interface and abstract Class:

**An interface** is a contract for what the classes can do. It, however, does not specify how the classes should do it. When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface. Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usages of interface is provide a communication contract between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.

A class containing one or more abstract methods is called an **abstract class**. An abstract class is incomplete in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class cannot be instantiated. In other words, you cannot create instances from an abstract class (otherwise, you will have an incomplete instance with missing method's body).

To use an abstract class, you have to derive a subclass from the abstract class. In the derived subclass, you have to override the abstract methods and provide implementation to all the abstract methods. The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.)

# 3. Android Classes and Basics

## 3.1 Android Fundamentals

Android apps are written in the Java or Kotlin Programming Language. The Android SDK tools compile your code-along with any data and resource files – into an APK: android Android package, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

Once installed on a device, each Android app lives in its own security sandbox:

- The Android operating system is a multi-user Linux system in which each app is a different user.
- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permission for all the files in an app so that only the user ID assigned to that app can access them.
- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.
- By default, every app runs in its own Linux process. Android starts the process when any of the app's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

In this way, the Android system implements the principle of least privilege. That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission.

However, there are ways for an app to share data with other apps and for an app to access system services:

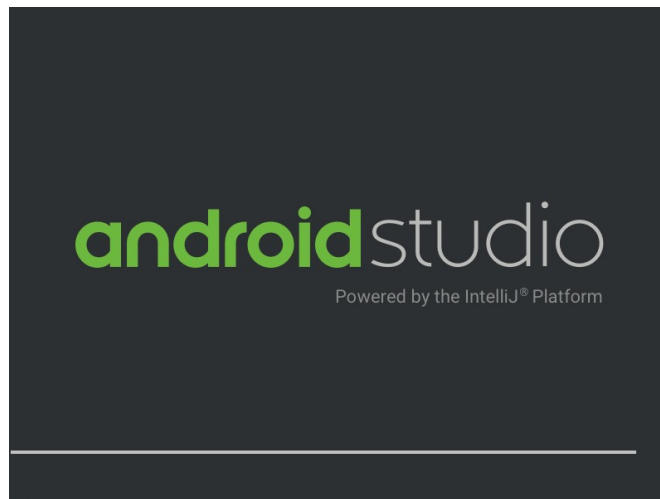
- It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM ( the apps must also be signed with the same certificate).
- An app can request permission to access device data such as the user's contacts, SMS messages, the mountable storage(SD card), camera, Bluetooth, and more. The user has to explicitly grant these permissions.



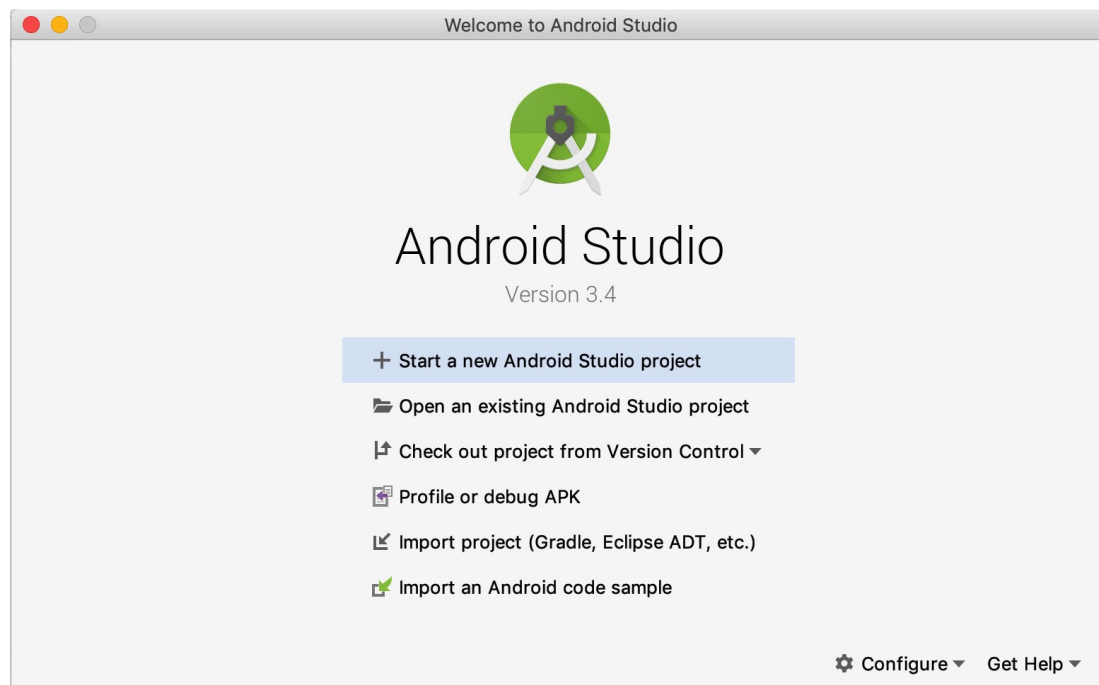
## 3.2 Creating an Android App

Steps to create an Android app are:

1. Open Android Studio from your application list in your computer.

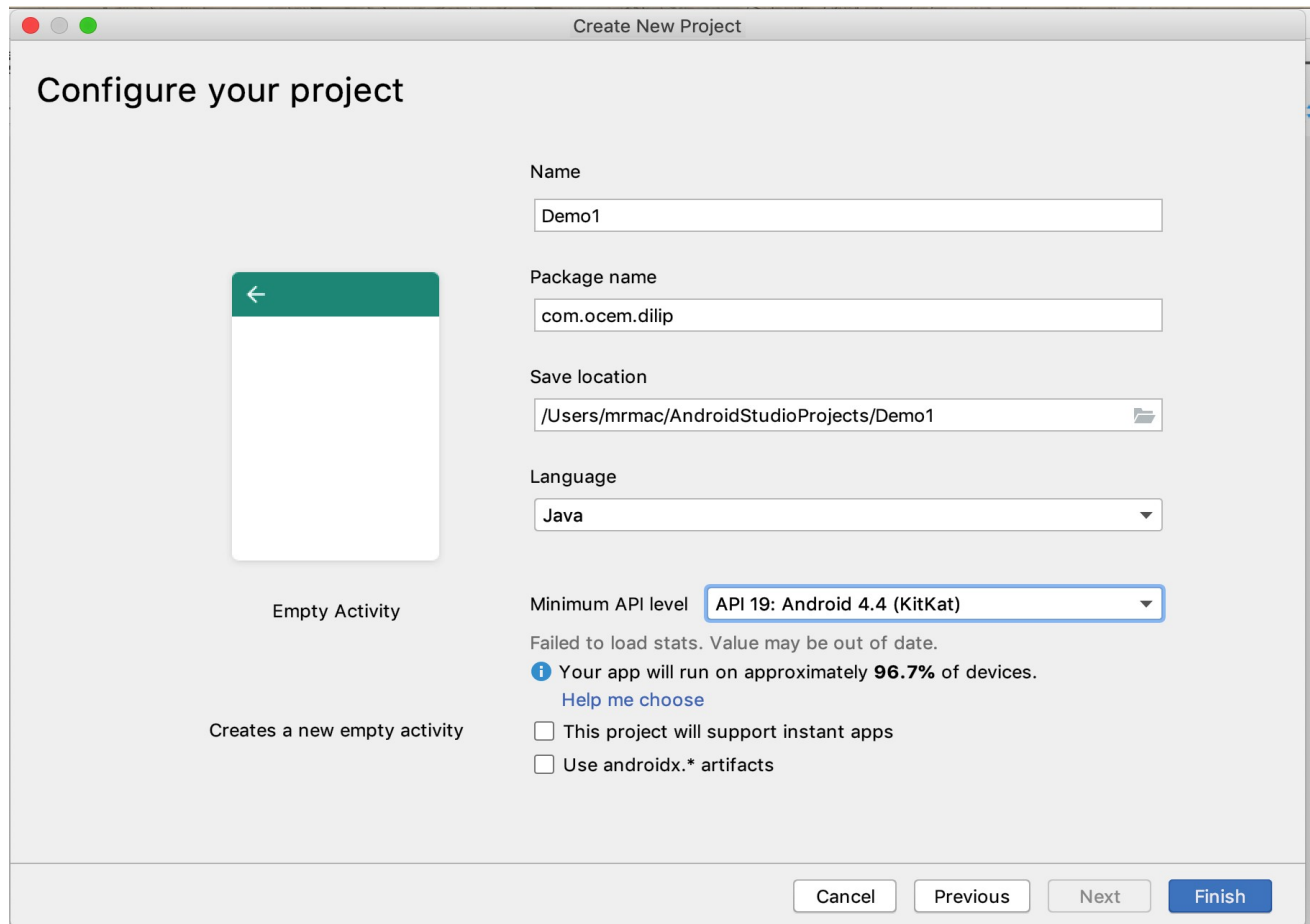


2. Click on “Start a new Android Studio project” from welcome screen's to open new project

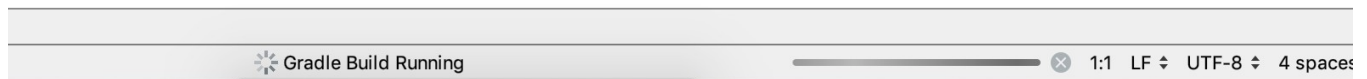


3. Select “Empty Activity” from next screen.

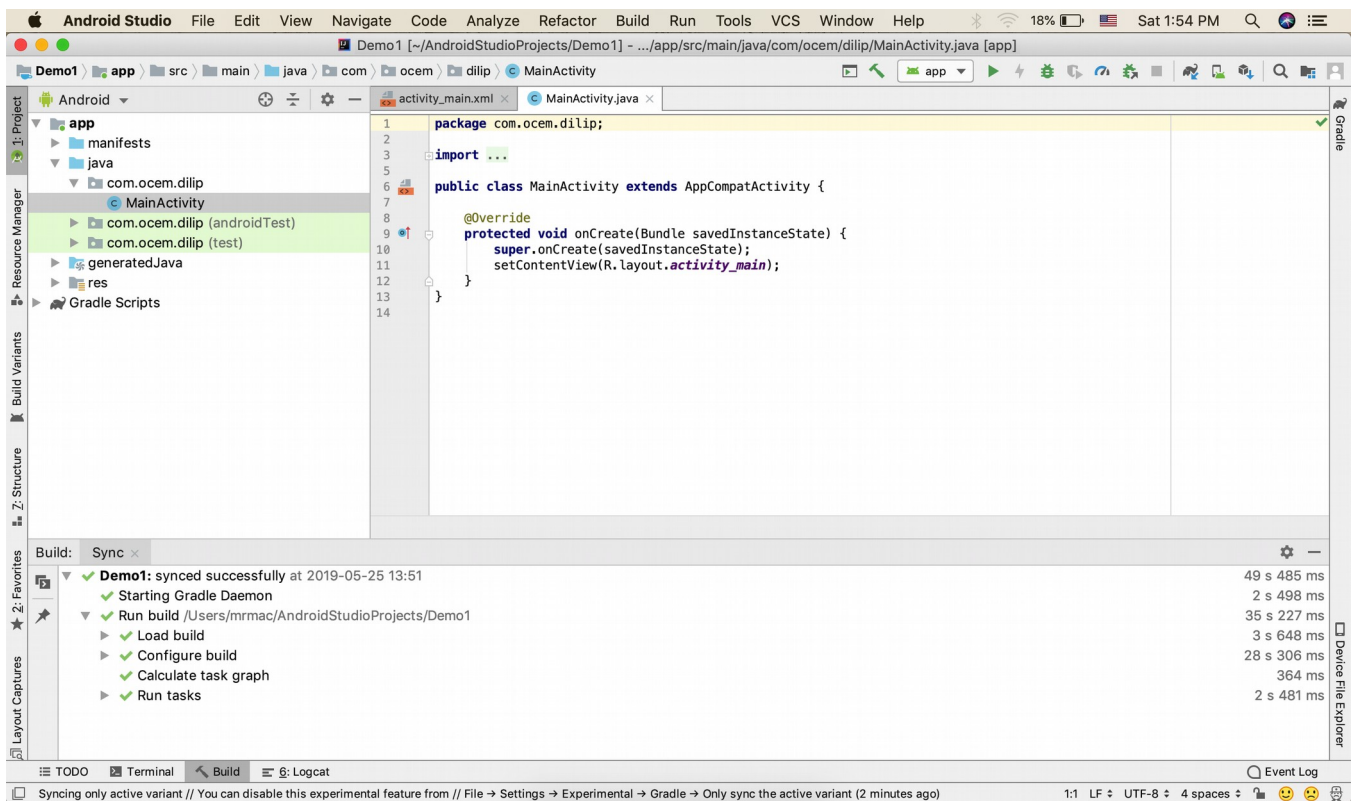
4. Fill out the fields on the next screen where you have to provide the application name, company domain name and select language for development. After all fields are filled select finish.



will take some time to load your projects and build it. During the process of building your project you can monitor the progress from the following progress bar at the bottom of Android Studio.

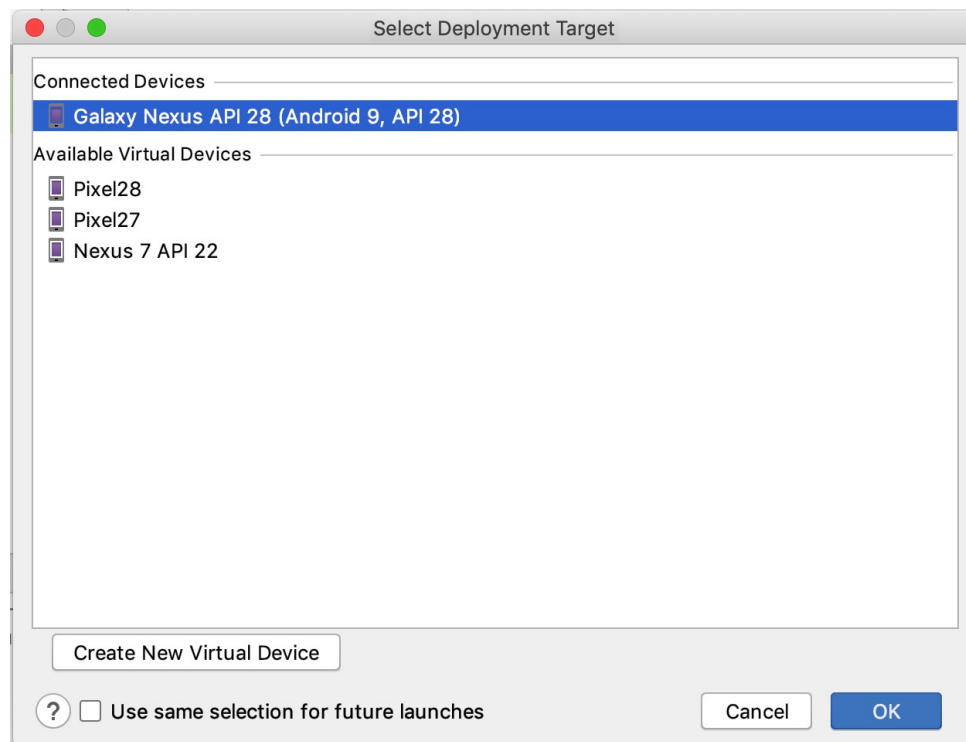


After the Build process is completed and the project is instantiated you will be able to write your code on the project and run it. Once build completes you should get following screen.

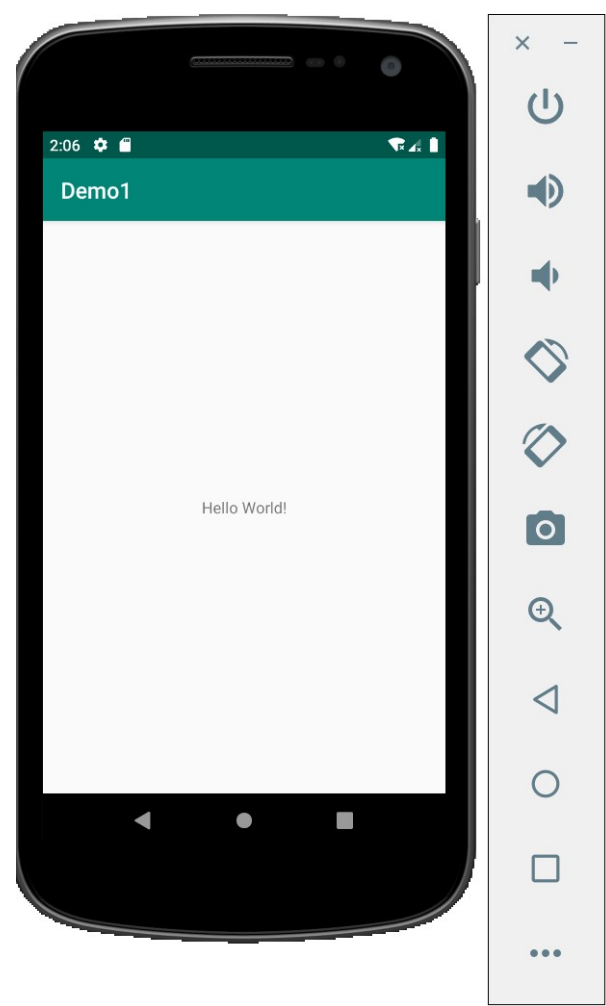


The image is not clear on the notes and is not possible to include clear image of the whole screen, but you will get idea when it will be ready on your device.

5. Click on run option from the toolbar and select an emulator or your android phone from the list to deploy your app.



Once the deployment is completed your will have your app deployed on emulator or your phone.



### 3.3 Android Manifest file

Every project in Android includes a manifest file, which is `AndroidManifest.xml`, stored in the root directory of its project hierarchy. The manifest file is an important part of our app because it defines the structure and metadata of our application, its components, and its requirements.

This file includes nodes for each of the Activities, Services, Content Providers and Broadcast Receiver that make the application and using Intent Filters and Permissions, determines how they co-ordinate with each other and other applications.

The manifest file also specifies the application metadata, which includes its icon, version number, themes etc. and additional top- level nodes can specify any required permissions, unit tests and define hardware, screen, or platform requirements.

The manifest comprises of a root manifest tag with a package attribute set to the project's package. It should also include an `xmlns:android` attribute that will supply several system attributes used within the file.

We use `versionCode` attribute is used to define the current application version in the form of an integer that increments itself with the iteration of version due to update. Also, `versionName` attribute is used to specify a public version that will be displayed to the users.

We can also specify whether our app should install on an SD card of the internal memory using the `installLocation` attribute.

A typical manifest node looks as:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ocem.dilip">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Some of the works that are done by manifests are:

- It names the Java package for the application. The package name serves as a unique identifier for the application.

- It describes the components of the application — the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.
- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.

### 3.4 The Activity Class

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.

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with `setContentView(View)`.

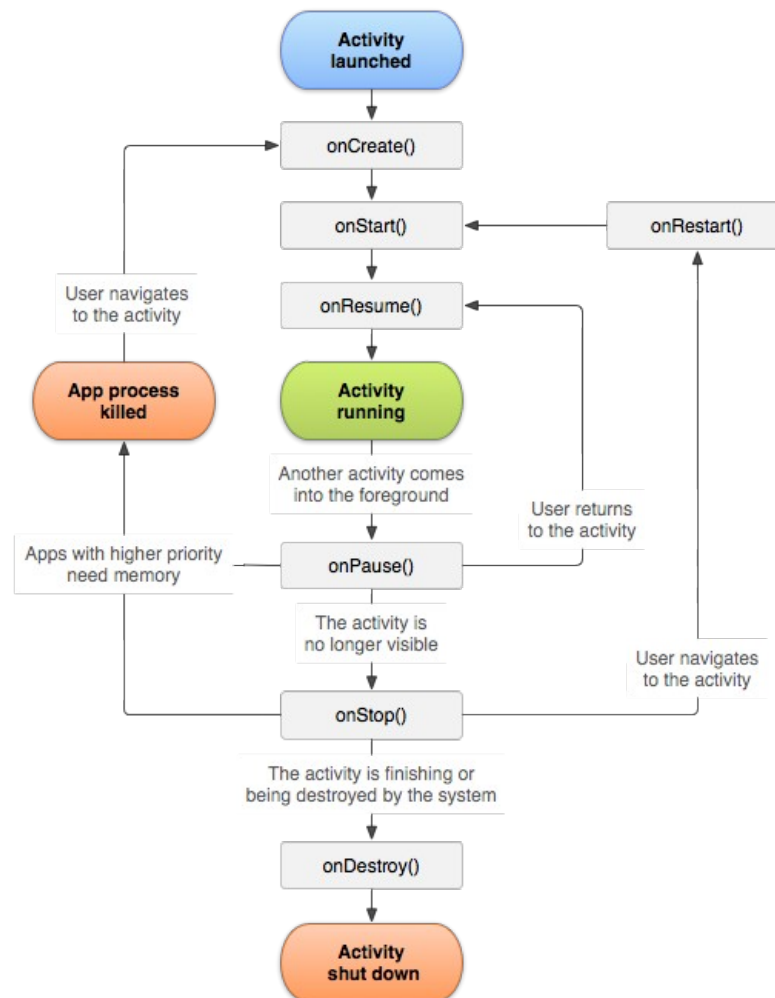
Activity for an app can be compared with a html page for a website or a web application. Android application can have from 0 to many activities according to the requirement or its operation. For instance while using social application we have Login Activity, NewsFeed activity, Profile activity and many more activity that combines and work to perform as a Social application.

For a general concept we can define activity as a single screen of any app where the UI components are placed and user can provide feedback to app with interacting on those components. As there are multiple screens on a normal app, we have multiple activities on single app to achieve the objective of the application.

Activity class are used to create those activities or screens in android by creating a Java class and extending the Activity class. And later the created activities needs to be registered on manifest file.

## 3.5 Activity Lifecycle

The Activity base class defines a series of events that govern the life cycle of an activity. Figure below shows the lifecycle of an Activity.



The Activity class defines the following events:

- **onCreate():**  
Called when the activity is first created
- **onStart():**  
Called when the activity becomes visible to the user

- `onResume()`:  
Called when the activity starts interacting with the user
- `onPause()`:  
Called when the current activity is being paused and the previous activity is being resumed
- `onStop()`:  
Called when the activity is no longer visible to the user
- `onDestroy()`:  
Called before the activity is destroyed by the system (either manually or by the system to conserve memory)
- `onRestart()`:  
Called when the activity has been stopped and is restarting again

## 3.6 Extending the activity class

To create an activity, we must create a Java class that extends the Activity base class. Example:

```
package com.ocem.dilip;
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

Here, “MainActivity” is the Java class that extends the Activity base class.

The activity loads its user interface (UI) component using the XML file defined in “res/layout” folder.

In above example, UI is loaded from “activity\_main.xml” file via,

```
setContentView(R.layout.activity_main);
```



## 3.7 Creating Default Activity

Default activity or launch activity can be created or declared by pointing out that the activity will be launching activity on manifest file for which we need to add following code to the activity attribute.

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

When the <intent-filter> is set on activity with action.Main as action and category.Launcher as category then the application running process assumes that the related activity is the launching or default activity and runs it as a first activity when the application is opened.

## 3.8 Creating Splash and Login Activity

*This is a practical topic and will be performed live on class or practical lectures.*

## 3.9 Intent

Intent is a powerful concept within the Android universe. An intent is a message that can be thought of as a request that is given to either an activity within your own app, an external application, or a built-in Android service.

Think of an intent as a way for an Activity to communicate with the outside Android world. A few key tasks that an intent might be used for within your apps:

- Take the user to another screen (activity) within your application
- Take the user to a particular URL within the Android web browser
- Take the user to the camera to have them take a picture
- Initiate a call for the user to a given number

As you can see, the Intent is a core part of user flows in Android development. The Intent object itself is a class that represents a particular "request" including the topic of the request and any request "parameters" which are called the Bundle

There are two types of intent:

### Explicit Intent

An "explicit" intent is used to launch other activities within your application. For example, if you the user presses the "compose" button and you want to bring up an activity for them to compose a message, you would launch that second activity using an explicit intent.

Using an intent is as simple as constructing the Intent with the correct parameters and then invoking that intent using the startActivity method:

```
Intent i = new Intent(ActivityOne.this, ActivityTwo.class);
startActivity(i);
```

**Note:** The first argument of the Intent constructor used above is a Context which at the moment is just the current Activity in scope.

In addition to specifying the activity that we want to display, an intent can also pass key-value data between activities. Think of this as specifying the "request parameters" for an HTTP Request. You can specify the parameters by putting key-value pairs into the intent bundle:

```
Intent i = new Intent(ActivityOne.this, ActivityTwo.class);
i.putExtra("username", "dilip");
i.putExtra("reply_to", "kushal");
startActivity(i);
```

## Implicit Intent

Implicit Intents are requests to perform an action based on a desired action and target data. This is in contrast to an explicit intent that targets a specific activity. For example, if I want to make a phone call for the user, that can be done with this intent:

```
Intent callIntent = new Intent(Intent.ACTION_CALL);
callIntent.setData(Uri.parse("tel:984592410"));
startActivity(callIntent);
```

## 3.10 Permission

By default, an Android app starts with zero permissions granted to it. When the app needs to use any of the protected features of the device (sending network requests, accessing the camera, sending an SMS, etc) it must obtain the appropriate permission from the user to do so.

Before Marshmallow, permissions were handled at install-time and specified in the AndroidManifest.xml within the project. After Marshmallow, permissions must now be **requested at runtime** before being used.

### Permissions before Marshmallow

Permissions were much simpler before Marshmallow (API 23). All permissions were handled at install-time. When a user went to install an app from the Google Play Store, the user was presented a list of permissions that the app required (some people referred to this as a "wall of permissions". The user could either accept all the permissions and continue to install the app or decide not to install the app. It was an all or nothing approach. There was no way to grant only certain permissions to the app and no way for the user to revoke certain permissions after the app was installed.

For an app developer, permissions were very simple. To request one of the many permission, simply specify it in the AndroidManifest.xml:

For example, an application that needs to read the user's contacts would add the following to its AndroidManifest.xml:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.ocem.dilip" >

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

That's all there was to it. The user had no way of changing permissions, even after installing the app. This made it easy for developers to deal with permissions, but wasn't the best user experience.

## Permission Updates in Marshmallow

Marshmallow brought large changes to the permissions model. It introduced the concept of runtime permissions. These are permissions that are requested while the app is running (instead of before the app is installed). These permission can then be allowed or denied by the user. For approved permissions, these can also be revoked at a later time. Permission are now categorized in two segment, first Normal Permission that are handled as before and second is Runtime Permission which is handled during the exact operation on app where the permission is required.

### Normal Permission

When you need to add a new permission, see if the permission is considered a PROTECTION\_NORMAL permission. In Marshmallow, Google has designated certain permissions to be "safe" and called these "Normal Permissions". These are things like ACCESS\_NETWORK\_STATE, INTERNET, etc. which can't do much harm. Normal permissions are automatically granted at install time and never prompt the user asking for permission.

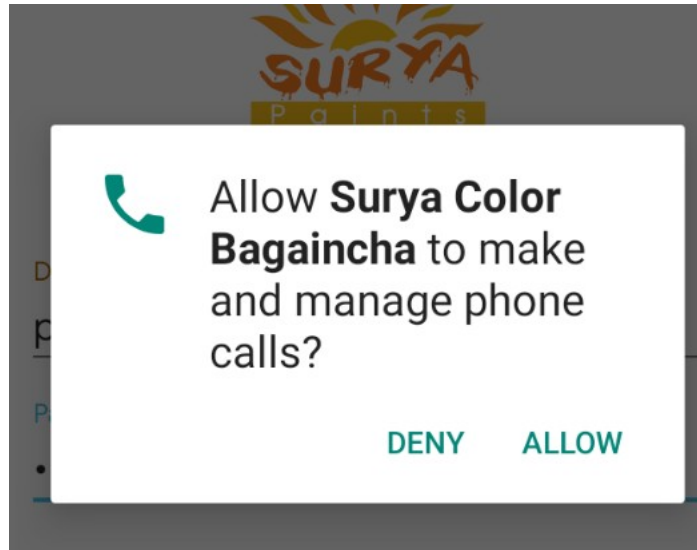
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.ocem.dilip" >

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

Normal Permission are always added to manifest file.

### Runtime Permission

If the permission you need to add isn't listed under the normal permissions, you'll need to deal with "Runtime Permissions". Runtime permissions are permissions that are requested as they are needed while the app is running. These permissions will show a dialog to the user, similar to the following one:



The first step when adding a "Runtime Permission" is to add it to the AndroidManifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.bihanitech.suryacolortone">

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

    ...

</manifest>
```

Next, you'll need to initiate the permission request and handle the result. The following code shows how to do this in the context of an Activity

// MainActivity.java

```
public class MainActivity extends AppCompatActivity {

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

        // In an actual app, you'd want to request a permission when the user performs an action
        // that requires that permission.
        getPermissionToReadUserPhoneState();
    }

    // Identifier for the permission request
    private static final int READ_PHONE_STATE_PERMISSIONS_REQUEST = 1;
```

```

// Called when the user is performing an action which requires the app to read the
// user's contacts
public void getPermissionToReadUserPhoneState() {
    // 1) Use the support library version ContextCompat.checkSelfPermission(...) to avoid
    // checking the build version since Context.checkSelfPermission(...) is only available
    // in Marshmallow
    // 2) Always check for permission (even if permission has already been granted)
    // since the user can revoke permissions at any time through Settings
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_PHONE_STATE)
        != PackageManager.PERMISSION_GRANTED) {

        // The permission is NOT already granted.
        // Check if the user has been asked about this permission already and denied
        // it. If so, we want to give more explanation about why the permission is needed.
        if (shouldShowRequestPermissionRationale(
            Manifest.permission.READ_PHONE_STATE)) {
            // Show our own UI to explain to the user why we need to read the contacts
            // before actually requesting the permission and showing the default UI
        }

        // Fire off an async request to actually get the permission
        // This will show the standard permission request dialog UI
        requestPermissions(new String[]{Manifest.permission.READ_PHONE_STATE},
            READ_CONTACTS_PERMISSIONS_REQUEST);
    }
}

// Callback with the request from calling requestPermissions(...)
@Override
public void onRequestPermissionsResult(int requestCode,
                                       @NonNull String permissions[],
                                       @NonNull int[] grantResults) {
    // Make sure it's our original READ_CONTACTS request
    if (requestCode == READ_PHONE_STATE_PERMISSIONS_REQUEST) {
        if (grantResults.length == 1 &&
            grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(this, "Read Phone State permission granted",
                Toast.LENGTH_SHORT).show();
        } else {
            // showRationale = false if user clicks Never Ask Again, otherwise true

```

```

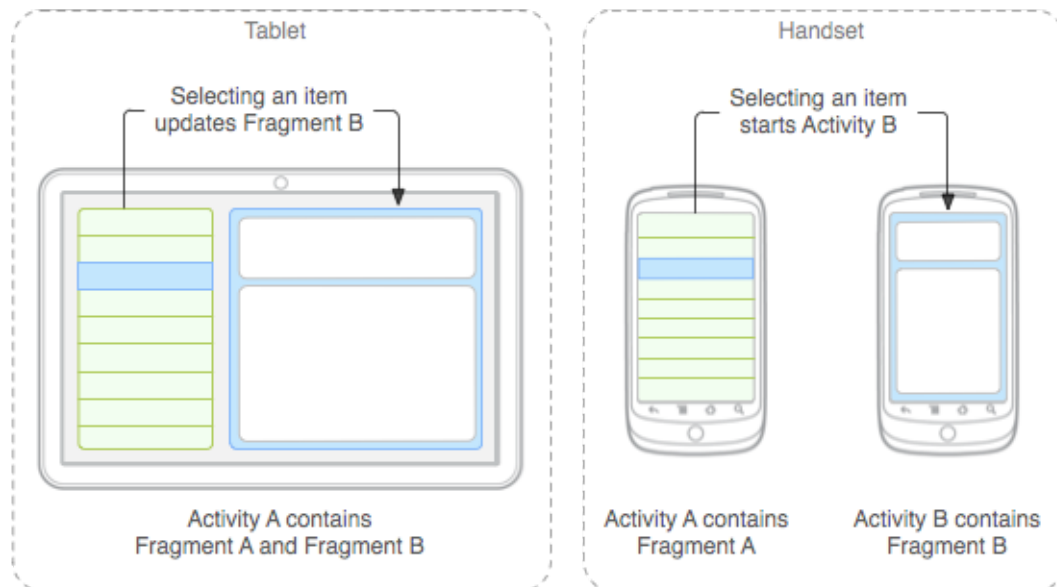
        boolean showRationale = shouldShowRequestPermissionRationale( this,
Manifest.permission.READ_PHONE_STATE);

        if (showRationale) {
            // do something here to handle degraded mode
        } else {
            Toast.makeText(this, "Read Phone State permission denied",
Toast.LENGTH_SHORT).show();
        }
    }
} else {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
}
}
}

```

### 3.10 The Fragment Class and its usage

A fragment is a reusable class implementing a portion of an activity. A Fragment typically defines a part of a user interface. Fragments must be embedded in activities; they cannot run independently of activities.



### Understanding Fragments

Here are the important things to understand about fragments:

- A Fragment is a combination of an XML layout file and a java class much like an Activity.
- Using the support library, fragments are supported back to all relevant Android versions.
- Fragments encapsulate views and logic so that it is easier to reuse within activities.

- Fragments are standalone components that can contain views, events and logic.

Within a fragment-oriented architecture, activities become navigational containers that are primarily responsible for navigation to other activities, presenting fragments and passing data.

## Importance of Fragments

There are many use cases for fragments but the most common use cases include:

- Reusing View and Logic Components**- Fragments enable re-use of parts of your screen including views and event logic over and over in different ways across many disparate activities. For example, using the same list across different data sources within an app.
- Tablet Support**- Often within apps, the tablet version of an activity has a substantially different layout from the phone version which is different from the TV version. Fragments enable device-specific activities to reuse shared elements while also having differences.
- Screen Orientation**- Often within apps, the portrait version of an activity has a substantially different layout from the landscape version. Fragments enable both orientations to reuse shared elements while also having differences.

## Fragment Lifecycle

Fragment has many methods which can be overridden to plug into the lifecycle (similar to Activity):

- onAttach()** is called when a fragment is connected to an activity.
- onCreate()** is called to do initial creation of the fragment.
- onCreateView()** is called by Android once the Fragment should inflate a view.
- onViewCreated()** is called after onCreateView() and ensures that the fragment's root view is non-null. Any view setup should happen here. E.g., view lookups, attaching listeners.
- onActivityCreated()** is called when host activity has completed its onCreate() method.
- onStart()** is called once the fragment is ready to be displayed on screen.
- onResume()** - Allocate “expensive” resources such as registering for location, sensor updates, etc.
- onPause()** - Release “expensive” resources. Commit any changes.
- onDestroyView()** is called when fragment's view is being destroyed, but the fragment is still kept around.
- onDestroy()** is called when fragment is no longer in use.
- onDetach()** is called when fragment is no longer connected to the activity.

# 4. Android User Interface

## 4.1 Introduction of Multiple Screen Size and Orientation Interfaces

Android runs on a variety of devices that offer different screen sizes and densities. That's why handling the multiple screen size is most important. We should design the user interface of our application in such a way so that it appears correctly on the widest possible range of devices. Here are some of the points that will help to achieve multiple screen size support:

### 1. Dimensions

- Always avoid hard-coded layout sizes.
- Use only **dp**(density-independent pixels), `match_parent` or `wrap_content` for layout elements dimensions.
- Use only **sp**(scale-independent pixels) for text size.
- Keep dimensions in **dimens.xml** files (Not hardcoded in app code or xml file).
- Provide dimensions for different **values** folder for different screen resolutions.

Values folder for different screen resolutions.

|                |               |                                   |
|----------------|---------------|-----------------------------------|
| values-sw720dp | 10.1"         | tablet 1280x800 mdpi              |
| values-sw600dp | 7.0"          | tablet 1024x600 mdpi              |
| values-sw480dp | 5.4"<br>5.1"  | 480x854 mdpi ,<br>480x800 mdpi    |
| values-xxhdpi  | 5.5"          | 1080x1920 xxhdpi                  |
| values-xxxhdpi | 5.5"          | 1440x2560 xxxhdpi                 |
| values-xhdpi   | 4.7"<br>4.65" | 1280x720 xhdpi,<br>1280x720 xhdpi |
| values-hdpi    | 4.0"<br>3.7"  | 480x800 hdpi,<br>480x854 hdpi     |
| values-mdpi    | 3.2"          | 320x480 mdpi                      |
| values-ldpi    | 3.4"<br>3.3"  | 240x432 ldpi,<br>240x400 ldpi,    |

We will find now-a-days maximum phone screen size 4" to 7". So we can create values folder xhdpi, xxhdpi, xxxhdpi, sw480dp and sw600dp with a base value folder.



## 2. Image and icon

- Provide different images and icon in different drawable folder for different screen resolutions.

|                  |             |
|------------------|-------------|
| drawable-ldpi    | //240x320   |
| drawable-mdpi    | //320x480   |
| drawable-hdpi    | //480x800   |
| drawable-xhdpi   | //720x1280  |
| drawable-xxhdpi  | //1080X1920 |
| drawable-xxxhdpi | //1440X2560 |

- Don't apply fixed values everywhere for image. Fixed dimensions can be vary with different device and image will be looked stretch. So use wrap\_content.
- Recommended minimum size for icons is 32 dp and you need 8 dp free space between another icon.
- You should put all your app icons in **mipmap** directories instead of **drawable** directories. Because all **mipmap** directories are retained in the APK even if you build density-specific APKs. This allows launcher apps to pick the best resolution icon to display on the home screen.

## 3. Layout Design

The configuration qualifiers you can use to provide size-specific resources are small, normal, large, and xlarge. For example, layouts for an extra large screen should go in layout-xlarge/.

But beginning with Android 3.2 (API level 13), the above size groups are deprecated and you should instead use the sw<N>dp (Smallest width device pixel) configuration qualifier to define the smallest available width required by your layout resources.

To create an alternative layout in Android Studio (using version 3.0 or higher), proceed as follows:

- 1.Open your default layout and then click **Orientation in Editor** in the toolbar.
- 2.In the drop-down list, click to create a suggested variant such as **Create Landscape Variant** or click **Create Other**.
- 3.If you selected **Create Other**, the **Select Resource Directory** appears. Here, select a screen qualifier on the left and add it to the list of **Chosen qualifiers**. When you're done adding qualifiers, click **OK**.

```
res/layout/my_layout.xml    #The default layout file  
  
res/layout-ldpi/my_layout.xml  
  
res/layout-mdpi/my_layout.xml  
  
res/layout-hdpi/my_layout.xml
```

```
res/layout-xhdpi/my_layout.xml  
res/layout-sw480dp/my_layout.xml  
res/layout-sw600dp/my_layout.xml  
res/layout-sw700dp/my_layout.xml
```

Use `LinearLayout` with proper using of `android:layout_weight`.

It is not a good practice to create different folders for layouts. Create your layout such that it works fine with all the screen sizes. To achieve this, play with the layout attributes. You only need to have different images for hdpi, mdpi and ldpi types. The rest will be managed by android OS.

If you want to use single layout and that should support all the screens like ldpi, , mdpi, hdpi, x-hdpi, xx-hdpi then you have to use `layout_weight` in your layout that will handle screen size for all the screens

## 4.2 User Interface Classes

In android, **Layout** is used to define the user interface for an app or activity and it will hold the UI elements that will appear to the user.

The user interface in android app is made with a collection of **View** and **ViewGroup** objects. Generally, the android apps will contain one or more activities and each activity is a one screen of app. The activities will contain a multiple UI components and those UI components are the instances of **View** and **ViewGroup** subclasses.

### View

View class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for *widgets*, which are used to create interactive UI components (buttons, text fields, etc.).

Commonly used Views are:

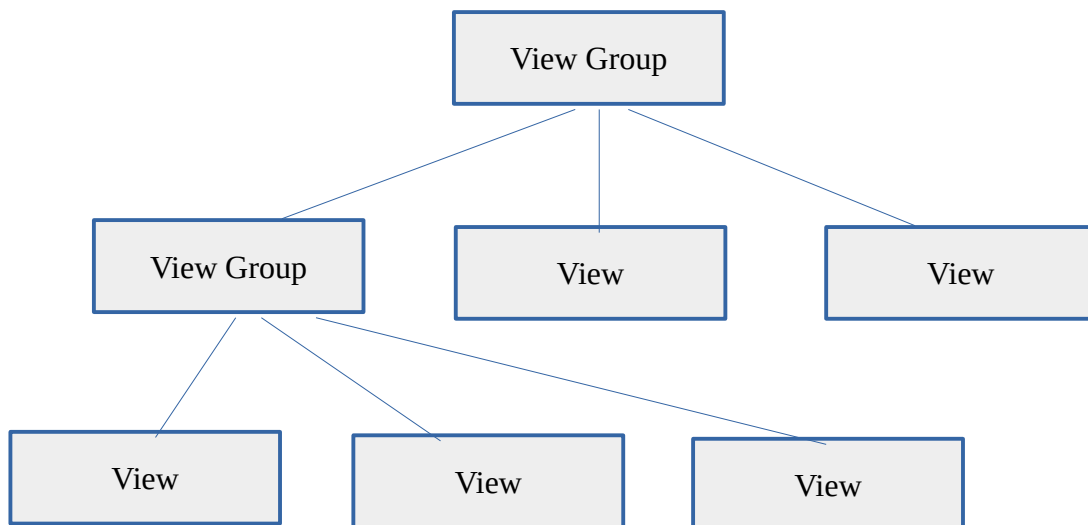
- **EditText**
  - a user interface element for entering and modifying text
- **ImageView**
  - displays image resources, for example `Bitmap` or `Drawable` resources

- TextView
  - a user interface that displays text to the user
- Button
  - a user interface element the user can tap or click to perform an action
- ImageButton
  - displays a button with an image (instead of text) that can be pressed or clicked by the user.
- CheckBox
  - a checkbox is a specific type of two-states button that can be either checked or unchecked.
- Spinner
  - a view that displays one child at a time and lets the user pick among them.

## ViewGroup

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

A group of view is known as ViewGroup. Top level ViewGroup is a parent, and under it, all the view and other view groups are its children.



Linear Layout, Constraint Layout, Relative Layout, Coordinator Layout, Table Layout, Frame Layout, Web View, List View, Recycler View, Grid View etc. are some of the famous ViewGroups used in android development.

## 4.3 Android UI Layouts

In android, Layout is used to define the user interface for an app or activity and it will hold the UI elements that will appear to the user.

The user interface in android app is made with a collection of View and ViewGroup objects. Generally, the android apps will contain one or more activities and each activity is a one screen of app. The activities will contain a multiple UI components and those UI components are the instances of View and ViewGroup subclasses.

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

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

In android, we can define a layouts in two ways, those are

- Declare UI elements in XML
- Instantiate layout elements at runtime

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

### Declare UI Elements in XML

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

Following is the example of defining a layout in XML file (activity\_main.xml) using LinearLayout to hold a TextView, EditText and Button.

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

```

<TextView
    android:id="@+id/tvName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Enter Name"
/>

<EditText
    android:id="@+id/etName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10">
</EditText>

<Button
    android:id="@+id/btGetName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Get Name"/>

</LinearLayout>

```

## Instantiate Layout Elements at Runtime

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

Following is the example of creating a layout using LinearLayout to hold a TextView, EditText and Button in an activity using custom View and ViewGroup objects programmatically.

```

public class MainActivity extends AppCompatActivity {

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

```

## Android Layout Attributes

When we define a layout using XML file we need to set width and height for every View and ViewGroup element using `layout_width` and `layout_height` attributes respectively. These two attributes are needed to be provided compulsorily to each and every View or ViewGroup on our layout file but there are others to. Some of the common layout attributes are:

- `android:id`
  - it is used to uniquely identify the view and ViewGroups
- `android:layout_width`
  - It is used to define the width for View and ViewGroup elements in layout
- `android:layout_height`
  - It is used to define the height for View and ViewGroup elements in layout
- `android:layout_marginLeft`
  - It is used to define the extra space in left side for View and ViewGroup elements in layout
- `android:layout_marginRight`
  - It is used to define the extra space in right side for View and ViewGroup elements in layout
- `android:layout_marginTop`
  - It is used to define the extra space on top for View and ViewGroup elements in layout
- `android:layout_marginBottom`
  - It is used to define the extra space in bottom side for View and ViewGroup elements in layout
- `android:paddingLeft`
  - It is used to define the left side padding for View and ViewGroup elements in layout files
- `android:paddingRight`
  - It is used to define the right side padding for View and ViewGroup elements in layout files
- `android:paddingTop`
  - It is used to define padding for View and ViewGroup elements in layout files on top side
- `android:paddingBottom`
  - It is used to define the bottom side padding for View and ViewGroup elements in layout files
- `android:layout_gravity`
  - It is used to define how child Views are positioned

## Android Layout Types

We have a different type of layouts available in android to implement user interface for our android applications with different designs based on our requirements.

Following are the commonly used layouts in android applications to implement required designs.

### a. **LinearLayout**

In android, LinearLayout is a ViewGroup subclass which is used to render all child View instances one by one either in Horizontal direction or Vertical direction based on the orientation property.

In android, we can specify the linear layout orientation using android:orientation attribute.

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

Example:

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

    <TextView
        android:id="@+id/tv1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="first text"/>

    <TextView
        android:id="@+id/tv2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="second text"/>

</LinearLayout>
```

### b. **RelativeLayout**

In android, RelativeLayout is a ViewGroup which is used to specify the position of child View instances relative to each other (Child A to the left of Child B) or relative to the parent (Aligned to the top of parent).

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

Example:

```

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

    <Button
        android:id="@+id/btn1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:text="Button1" />
    <Button
        android:id="@+id/btn2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_centerVertical="true"
        android:text="Button2" />

</RelativeLayout>

```

### c. **TableLayout**

In android, TableLayout is a ViewGroup subclass which is used to display the child View elements in rows and columns.

In android, TableLayout will position its children elements into rows and columns and it won't display any border lines for rows, columns or cells.

The TableLayout in android will work same as HTML table and table will have as many columns as the row with the most cells. The TableLayout can be explained as **<table>** and TableRow is like **<tr>** element.

Example:

```

<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="UserId" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="User Name" />
    </TableRow>
</TableLayout>

```



```

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Location"/>
    </TableRow>
    <TableRow>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="1"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Dilip Poudel"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Chitwan"/>
    </TableRow>
</TableLayout>

```

#### d. FrameLayout

In android, **FrameLayout** is a **ViewGroup** subclass which is used to specify the position of **View** instances it contains on the top of each other to display only single **View** inside the **FrameLayout**.

In simple manner, we can say **FrameLayout** is designed to block out an area on the screen to display a single item.

In android, **FrameLayout** will act as a placeholder on the screen and it is used to hold a single child view.

In **FrameLayout**, the child views are added in a stack and the most recently added child will show on the top. We can add multiple children views to **FrameLayout** and control their position by using gravity attributes in **FrameLayout**.

Example:

```

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

    <ImageView
        android:id="@+id/img1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/background_image"/>

```

```

<TextView
    android:id="@+id/tv1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="40dp"
    android:background="#4C374A"
    android:padding="10dp"
    android:text="Beautiful River, Narayani"
    android:textColor="#FFFFFF"
    android:textSize="20sp"/>

</FrameLayout>

```

## 4.4 Resources and Styles

Resources are the additional files and static content that your code uses, such as bitmaps, layout definitions, user interface strings and animation instructions. The Android resource system keeps track of all non-code assets associated with an application.

For any type of resource, you can specify *default* and multiple *alternative* resources for your application:

- Default resources are those that should be used regardless of the device configuration or when there are no alternative resources that match the current configuration.
- Alternative resources are those that you've designed for use with a specific configuration. To specify that a group of resources are for a specific configuration, append an appropriate configuration qualifier to the directory name.

The **res/** directory contains all the resources in various sub directories. Here we have an image resource, two layout resources, and a string resource file. Following table gives a detail about the resource directories supported inside project **res/** directory.

| Directory | Resource Type  |
|-----------|--|
| anim/     | XML files that define property animations. They are saved in res/anim/ folder and accessed from the <b>R.anim</b> class.   |
| color/    | XML files that define a state list of colors. They are saved in res/color/ and accessed from the <b>R.color</b> class.   |
| 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 <b>R.drawable</b> class. |
| layout/   | XML files that define a user interface layout. They are saved in res/layout/ and accessed from the <b>R.layout</b> class.  |

|         |   |
|---------|---|
| 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 <b>R.menu</b> class.   |
| raw/    | Arbitrary files to save in their raw form. You need to call <code>Resources.openRawResource()</code> with the resource ID, which is <code>R.raw.filename</code> to open such raw files.   |
| values/ | XML files that contain simple values, such as strings, integers, and colors. For example, here are some filename conventions for resources you can create in this directory – <ul style="list-style-type: none"> <li>• arrays.xml for resource arrays, and accessed from the <b>R.array</b> class.</li> <li>• integers.xml for resource integers, and accessed from the <b>R.integer</b> class.</li> <li>• bools.xml for resource boolean, and accessed from the <b>R.bool</b> class.</li> <li>• colors.xml for color values, and accessed from the <b>R.color</b> class.</li> <li>• dimens.xml for dimension values, and accessed from the <b>R.dimen</b> class.</li> <li>• strings.xml for string values, and accessed from the <b>R.string</b> class.</li> <li>• styles.xml for styles, and accessed from the <b>R.style</b> class.</li> </ul> |
| xml/    | Arbitrary XML files that can be read at runtime by calling <code>Resources.getXML()</code> . You can save various configuration files here which will be used at run time.  |

## Android Styles

A style is defined in an XML resource that is separate from the XML that specifies the layout. This XML file resides under **res/values/** directory of your project and will have **<resources>** as the root node which is mandatory for the style file. The name of the XML file is arbitrary, but it must use the .xml extension.

You can define multiple styles per file using **<style>** tag but each style will have its name that uniquely identifies the style. Android style attributes are set using **<item>** tag as shown below –

```
<resource>
  <style name="AppTheme" parent="android:Theme.Material">
    <item name="android:color/primary">@color/primary</item>
    <item name="android:color/primaryDark">@color/primary_dark </item>
    <item name="android:colorAccent/primary">@color/accent</item>
  </style>
</resource>
```

# 5. Advanced Topics

## 5.1 User Notifications

A notification is a message you can display to the user outside of your application's normal UI. Notifications appear in the phone's notification area and then can be expanded to see more information.

This is typically used to keep the user informed about events that are coming in that they should be aware of such as new email messages, new chat messages or an upcoming calendar event.

Notifications can include actions that let the user take a relevant action from within the notification drawer. At the minimum, all notifications consist of a base layout including:

- Icon of the app or relevant user image
- Title and message
- Timestamp

Starting with API level 26 (Oreo), notification channels were introduced to give more control over the notifications for users. Notification channels can be enabled/disabled by the user in the Settings app or upon long clicking the notification. Notifications are sorted based on the importance level of the channels. The developers targeting API level 26 and above need to set a channel to a notification at the minimum.

## Creating Notification

Notifications in Android are represented by the Notification class. To create notifications we use NotificationManager class which can be received from Context.

For API level 26 and above, we need to create a notification channel, provide a unique string id, name and importance level. Setting the description helps the user to identify the channel. Create the notification channel in the app's application class. Creating a notification channel again with same id updates the name and description with the new values.

```
// Configure the channel

int importance = NotificationManager.IMPORTANCE_DEFAULT;
NotificationChannel channel = new NotificationChannel("myChannelId", "My Channel",
importance);
channel.setDescription("Reminders");

// Register the channel with the notifications manager
NotificationManager mNotificationManager = (NotificationManager)
context.getSystemService(Context.NOTIFICATION_SERVICE);
mNotificationManager.createNotificationChannel(channel);
```

Let's build a basic notification using the **NotificationCompat.Builder**. Typically this will contain at least an icon, a title, body.

For **API level 26 and above**, we need to set a notification channel using the updated `.NotificationCompat.Builder`

*// Builder class for devices targeting API 26+ requires a channel ID*

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder (this,
"channel_id").setSmallIcon(R.drawable.notification_icon)

    .setContentTitle("My Notification")

    .setContentText("Notification body detail");
```

For **API level 25 and below**, we create the notification without any channel:

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder (
this).setSmallIcon(R.drawable.notification_icon)

    .setContentTitle("My Notification")

    .setContentText("Notification body detail");
```

And finally we have to append the notification using the **NotificationManager**

```
Notification mNotificationManager = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

mNotificationManager.notify(mId, mBuilder.build());
```

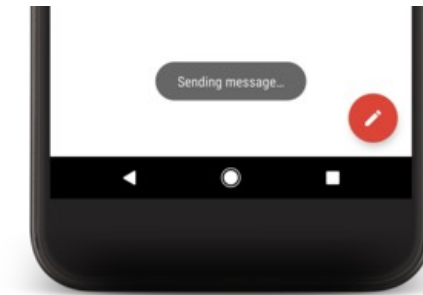
## 5.2 Toast

A toast is a view containing a quick little message for the user. The toast class helps you create and show those.

When the view is shown to the user, appears as a floating view over the application. It will never receive focus. The user will probably be in the middle of typing something else. The idea is to be as unobtrusive as possible, while still showing the user the information you want them to see. Two examples are the volume control, and the brief message saying that your settings have been saved.

A toast provides simple feedback about an operation in a small popup. It only fills the amount of space required for the message and the current activity remains visible and interactive. Toasts automatically disappear after a timeout.

For example, clicking **Send** on an email triggers a "Sending message..." toast, as shown in the following screen capture:



First, instantiate a Toast object with one of the `makeText()` methods. This method takes three parameters: the application Context, the text message, and the duration for the toast.

*// also supports Toast.LENGTH\_LONG*

```
Toast.makeText(getApplicationContext(), "some message", Toast.LENGTH_SHORT).show();
```

You can configure the position of a Toast. A standard toast notification appears near the bottom of the screen, centered horizontally. You can change this position with the `setGravity` method and specifying a Gravity constant.

```
Toast toast = Toast.makeText(getApplicationContext(), "some message",  
    Toast.LENGTH_SHORT);  
toast.setGravity(Gravity.CENTER_VERTICAL, 0, 0);  
toast.show();
```

## 5.3 SnackBar

SnackBar in android is a new widget introduced with the Material Design library as a replacement of a Toast.

Android SnackBar is light-weight widget and they are used to show messages in the bottom of the application with swiping enabled. SnackBar android widget may contain an optional action button.

### Difference between Toast and SnackBar

1. A Toast messages can be customized and printed anywhere on the screen, but a SnackBar can be only showed in the bottom of the screen
2. A Toast message don't have action button, but SnackBar may have action button optionally. Though, A SnackBar shouldn't have more than one action button
3. Toast message cannot be off until the time limit finish, but SnackBar can be swiped off before the time limit

**Note:** Toast message and SnackBar have display length property in common.

A code snippet to display a basic android SnackBar is shown below.

```
SnackBar snackbar = SnackBar .make(coordinatorLayout, "Hello Try Again..",
SnackBar.LENGTH_LONG);

snackbar.show();
```

We can assign an action button on snack bar to take up any action we want.

```
snackbar.setAction("RETRY", new View.OnClickListener() {

    @Override
    public void onClick(View view) {

        // your action here
    }
});
```

## 5.4 The Broadcast Receiver Class

A broadcast receiver (short receiver) is an Android component which allows you to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

For example, applications can register for the `ACTION_BOOT_COMPLETED` system event which is fired once the Android system has completed the boot process.

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

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

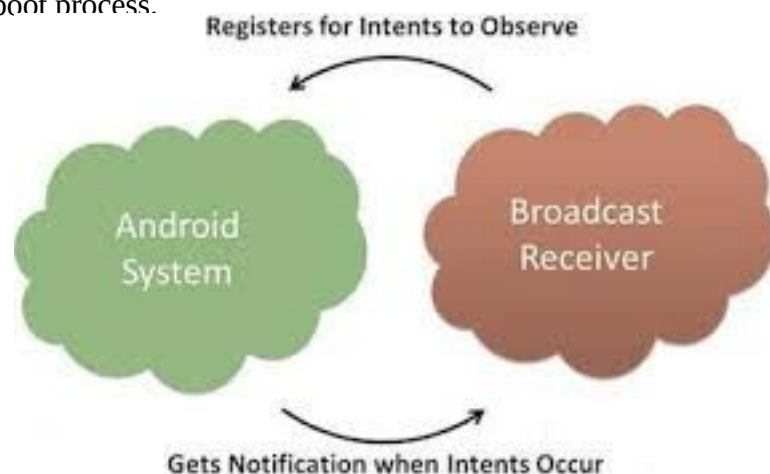
- **Creating the Broadcast Receiver**

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

```
Public class MyReceiver extends BroadcastReceiver {  
  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        Toast.makeText(context, "Intent Detected.", Toast.LENGTH_LONG)  
            .show()  
    }  
}
```

- **Registering Broadcast Receiver**

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





```

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

```

## 5.5 Threads

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new **Thread** object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named **main** of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The **exit** method of class **Runtime** has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the **run** method or by throwing an exception that propagates beyond the **run** method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of **Thread**. This subclass should override the **run** method of class **Thread**. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```

class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        ...
    }
}

```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);  
p.start();
```

The other way to create a thread is to declare a class that implements the **Runnable** interface. That class then implements the **run** method. An instance of the class can then be allocated, passed as an argument when creating **Thread**, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {  
    long minPrime;  
    PrimeRun(long minPrime) {  
        this.minPrime = minPrime;  
    }  
  
    public void run() {  
        // compute primes larger than minPrime  
        ...  
    }  
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

## 5.5 Threads in Android

In android when an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events and rendering the user interface and is usually called the UI thread. All components (activities, services, etc) and their executed code run in the same process and are instantiated by default in the UI thread.

But the problem here is performing long operations such as network access or database queries in the UI thread will block the entire app UI from responding. When the UI thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application will appear to freeze. Additionally, Android UI toolkit is not thread-safe and as such developers must not manipulate UI from a background thread.

In short, the two main rules about main UI thread are:

- Do not run long tasks on the main thread (to avoid blocking the UI)
- Do not change the UI at all from a background thread (only the main thread)

And to handle these long tasks on different thread we have solutions like AsyncTask, HandlerThread, ThreadPoolExecutor etc.

### 5.5.1 Async Task

AsyncTask is a mechanism for executing operations in a background thread without having to manually handle thread creation or execution. AsyncTasks were designed to be used for short operations (a few seconds at the most) and you might want to use a Service and/or Executor for very long running tasks.

This is typically used for long running tasks that cannot be done on UIThread, such as downloading network data from an API or indexing data from elsewhere on the device. An AsyncTask streamlines the following common background process:

- 1.**Pre-** Execute code on the UI thread before starting a task (e.g show ProgressBar)
- 2.**Task-** Run a background task on a thread given certain inputs (e.g fetch data)
- 3.**Updates-** Display progress updates during the task (optional)
- 4.**Post-** Execute code on UI thread following completion of the task (e.g show data)

Creating an AsyncTask is as simple as defining a class that extends from **AsyncTask** such as:

*// The types specified here are the input data type, the progress type, and the result type*

```
private class MyAsyncTask extends AsyncTask<String, Progress, Bitmap> {
    protected void onPreExecute() {
        // Runs on the UI thread before doInBackground
        // Good for toggling visibility of a progress indicator
        progressBar.setVisibility(ProgressBar.VISIBLE);
    }

    protected Bitmap doInBackground(String... strings) {
        // Some long-running task like downloading an image.
        Bitmap = downloadImageFromUrl(strings[0]);
        return someBitmap;
    }

    protected void onProgressUpdate(Progress... values) {
```

```

        // Executes whenever publishProgress is called from doInBackground
        // Used to update the progress indicator
        progressBar.setProgress(values[0]);
    }

    protected void onPostExecute(Bitmap result) {
        // This method is executed in the UIThread
        // with access to the result of the long running task
        imageView.setImageBitmap(result);
        // Hide the progress bar
        progressBar.setVisibility(ProgressBar.INVISIBLE);
    }
}

```

The worker once defined can be started anytime by creating an instance of the class and then invoke **.execute** to start the task:

```

public void onCreate(Bundle b) {
    // ...
    // Initiate the background task
    downloadImageAsync();
}

private void downloadImageAsync() {
    // Now we can execute the long-running task at any time.
    new MyAsyncTask().execute("http://images.com/image.jpg");
}

```

## Understanding the AsyncTask

AsyncTask accepts three generic types to inform the background work being done:

- **AsyncTask<Params, Progress, Result>**
  - **Params**- the type that is passed into the execute() method.
  - **Progress**- the type that is used within the task to track progress.
  - **Result**- the type that is returned by doInBackground().

For example AsyncTask<String, Void, Bitmap> means that the task requires a string input to execute, does not record progress and returns a Bitmap after the task is complete.

AsyncTask has multiple events that can be overridden to control different behavior:

- **onPreExecute**- executed in the main thread to do things like create the initial progress bar view.
- **doInBackground**- executed in the background thread to do things like network downloads.
- **onProgressUpdate**- executed in the main thread when **publishProgress** is called from doInBackground.
- **onPostExecute**- executed in the main thread to do things like set image views.

## Limitations of AsyncTask

An AsyncTask is tightly bound to a particular Activity. In other words, if the Activity is destroyed or the configuration changes then the AsyncTask will not be able to update the UI on completion. As a result, for short one-off background tasks **tightly coupled to updating an Activity**, we should consider using an AsyncTask as outlined above. A good example is for a several second network request that will populate data into a ListView within an activity.

So, as AsyncTask should be used for executing short operations (a few seconds at the most) in a sequential order. If you need to keep threads running for long periods of time or execute threads in parallel, it is highly recommended you use the **ThreadPoolExecutor** instead. If you need to have more control over how you are running sequential background tasks, see the **HandlerThread** below.

### 5.5.2 Handlers

A Handler is a component that can be attached to a thread and then made to perform some action on that thread via simple messages or Runnable tasks. It works in conjunction with another component, Looper, which is in charge of message processing in a particular thread.

When a Handler is created, it can get a Looper object in the constructor, which indicates which thread the handler is attached to. If you want to use a handler attached to the main thread, you need to use the looper associated with the main thread by calling `Looper.getMainLooper()`.

In this case, to update the UI from a background thread, you can create a handler attached to the UI thread, and then post an action as a Runnable:

```
Handler handler = new Handler(Looper.getMainLooper());
handler.post(new Runnable(){
    @Override
    public void run(){
        // update the ui from here
    }
});
```

A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it -- from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

There are two main uses for a Handler:

- (1) to schedule messages and runnables to be executed at some point in the future; and
- (2) to enqueue an action to be performed on a different thread than your own.

## 5.6 AlarmManager

This class provides access to the system alarm services. These allow you to schedule your application to be run at some point in the future. When an alarm goes off, the **Intent** that had been registered for it is broadcast by the system, automatically starting the target application if it is not already running. Registered alarms are retained while the device is asleep (and can optionally wake the device up if they go off during that time), but will be cleared if it is turned off and rebooted.

The Alarm Manager holds a CPU wake lock as long as the alarm receiver's `onReceive()` method is executing. This guarantees that the phone will not sleep until you have finished handling the broadcast. Once `onReceive()` returns, the Alarm Manager releases this wake lock. This means that the phone will in some cases sleep as soon as your `onReceive()` method completes. If your alarm receiver called **`Context.startService()`**, it is possible that the phone will sleep before the requested service is launched. To prevent this, your `BroadcastReceiver` and `Service` will need to implement a separate wake lock policy to ensure that the phone continues running until the service becomes available.

**Note:** Beginning with API 19 (**`Build.VERSION_CODES.KITKAT`**) alarm delivery is inexact: the OS will shift alarms in order to minimize wakeups and battery use

Suppose we need to set periodically executing background tasks. For example, we want to be able to check for new emails or content from a server every 15 minutes even if our application isn't running. This is useful for apps like email clients, news readers, instant messaging clients, et al. In this case, we don't necessarily need a long running task that runs forever. That would take drain battery life significantly and isn't what we want anyways.

For most of these common cases (checking for new data), what we really want to do is setup a **scheduler that triggers a background service at a regular interval** of our choosing. The best way to achieve this is to use an `IntentService` in conjunction with the `AlarmManager`. First, we have to define the `IntentService` to have periodically execute and setup a `BroadcastReceiver` that will be executed by the alarm and will launch the `IntentService`. After this the `IntentService` and `BroadcastReceiver` should be defined in manifest file.

Finally, we need to actually start the periodic alarm that will trigger the receiver by registering with the Alarm system service:

```
// Construct an intent that will execute the AlarmReceiver
Intent intent = new Intent(getApplicationContext(), MyAlarmReceiver.class);

// Create a PendingIntent to be triggered when the alarm goes off
final PendingIntent pIntent = PendingIntent.getBroadcast(this,
MyAlarmReceiver.REQUEST_CODE, intent, PendingIntent.FLAG_UPDATE_CURRENT);

// Setup periodic alarm every every half hour from this point onwards
long firstMillis = System.currentTimeMillis(); // alarm is set right away
```

```

AlarmManager alarm = (AlarmManager)
this.getSystemService(Context.ALARM_SERVICE);

// First parameter is the type: ELAPSED_REALTIME, ELAPSED_REALTIME_WAKEUP,
RTC_WAKEUP

// Interval can be INTERVAL_FIFTEEN_MINUTES, INTERVAL_HALF_HOUR,
INTERVAL_HOUR, INTERVAL_DAY
alarm.setInexactRepeating(AlarmManager.RTC_WAKEUP, firstMillis,
AlarmManager.INTERVAL_HALF_HOUR, pIntent);

```

This will cause the alarm to trigger immediately and then fire every half hour from that point forward. Each time the alarm fires, the MyAlarmReceiver broadcast intent is triggered which starts up the IntentService. The PendingIntent.FLAG\_UPDATE\_CURRENT flag ensures that if the alarm fires very quickly, that the events will replace each other rather than stack up.

## 5.7 Networking

### 5.7.1 Accessing the network in Android

Within an Android application you should avoid performing long running operations on the user interface thread. This includes file and network access.

The Android system crashes your application with a `NetworkOnMainThreadException` if you access network is accessed from the main thread.

Android contains the standard Java network `java.net` package which can be used to access network resources. The base class for HTTP network access in the `java.net` package is the `URLConnection` class.

Performing network operations with standard Java API can be cumbersome. You have to open and close a connections, enable caches and ensure to perform the network operation in a background thread.

To simplify these operations several popular Open Source libraries are available. The most popular ones are the following:

- OkHttp for efficient HTTP access
- Retrofit for REST based clients
- Glide for image processing

### 5.7.2 Permission to access the network

To access the Internet your application requires the `android.permission.INTERNET` permission. On modern Android API versions, this permission is automatically granted to your application.

### 5.7.3 Check the availability of the network

The network on an Android device is not always available. To check the network state your application requires the `android.permission.ACCESS_NETWORK_STATE` permission. You can check the network is currently available via the following code.

```
public boolean isNetworkAvailable() {
    ConnectivityManager cm = (ConnectivityManager) getSystemService
        (Context.CONNECTIVITY_SERVICE);

    NetworkInfo networkInfo = cm.getActiveNetworkInfo();
    // if no network is available networkInfo will be null
    // otherwise check if we are connected
    if (networkInfo != null && networkInfo.isConnected()) {
        return true;
    }
    return false;
}
```

A sample for network operation using Android.

```
private InputStream OpenHttpConnection(String urlString) throws IOException {
    InputStream in = null; int response = -1;
    URL url = new URL(urlString);
    URLConnection conn = url.openConnection();

    if (!(conn instanceof HttpURLConnection))
        throw new IOException("Not an HTTP connection");

    try{
        HttpURLConnection httpConn = (HttpURLConnection) conn;
        httpConn.setAllowUserInteraction(false);
        httpConn.setInstanceFollowRedirects(true);

        httpConn.setRequestMethod("GET");
        httpConn.connect();
        response = httpConn.getResponseCode();
        if (response == HttpURLConnection.HTTP_OK) {
            in = httpConn.getInputStream(); }
    }
    catch (Exception ex) {
        Log.d("Networking", ex.getLocalizedMessage());
        throw new IOException("Error connecting");
    }

    return in;
}
```



These kinds of Network Operation are permitted on main UI thread so we have to use methods like `AsyncTask` to call these functions on another thread.

# 6 . Graphics And Multimedia

## 6.1 Graphics and Animation

### 6.1.1 Graphics

If one talks about standard components that we can be used in UI, this is good but it is not enough when we want to develop a game or an app that requires graphic contents. Android SDK provides a set of API for drawing custom 2D and 3D graphics. When we write an app that requires graphics, we should consider how intensive the graphic usage is. In other words, there could be an app that uses quite static graphics without complex effects and there could be other app that uses intensive graphical effects like games. According to this usage, there are different techniques we can adopt:

- **Canvas and Drawable:**

In this case, we can extend the existing UI widgets so that we can customize their behavior or we can create custom 2D graphics using the standard method provided by the Canvas class.

- **Hardware acceleration:**

We can use hardware acceleration when drawing with the Canvas API. This is possible from Android 3.0.

- **OpenGL:**

Android supports OpenGL natively using NDK. This technique is very useful when we have an app that uses intensively graphic contents (i.e games).

The easiest way to use 2D graphics is extending the View class and overriding the onDraw method. We can use this technique when we do not need a graphics intensive app.

In this case, we can use the Canvas class to create 2D graphics. This class provides a set of method starting with draw that can be used to draw different shapes like:

- lines
- circle
- rectangle
- oval
- picture
- arc

For example let us suppose we want to draw a rectangle. We create a custom view and then we override onDraw method. Here we draw the rectangle:

```

public class TestView extends View {
    public TestView(Context context) {
        super(context);

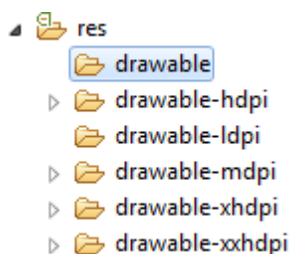
    }
    public TestView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }
    public TestView(Context context, AttributeSet attrs){
        super(context,attrs);
    }

    @Override
    protected void onDraw(Canvas canvas){
        super.onDraw(canvas);
        Paint p = new Paint();
        p.setColor(Color.GREEN);
        p.setStrokeWidth(1);
        canvas.drawRect(0,0,200,100,p);
    }
}

```

### 6.1.2 Drawable

In Android, a Drawable is a graphical object that can be shown on the screen. From API point of view all the Drawable objects derive from *Drawable* class. They have an important role in Android programming and we can use XML to create them. They differ from standard widgets because they are not interactive, meaning that they do not react to user touch. Images, colors, shapes, objects that change their aspect according to their state, object that can be animated are all drawable objects. In Android under res directory, there is a sub-dir reserved for Drawable, it is called *res/drawable*.



Under the drawable dir we can add binary files like images or XML files.

We can create several directories according to the screen density we want to support. These directories have a name like `drawable-xx`. This is very useful when we use images; in this case, we have to create several image versions: for example, we can create an image for the high dpi screen or another one for medium dpi screen.

Once we have our file under drawable directory, we can reference it, in our class, using

*`R.drawable.file_name`*

While it is very easy add a binary file to one of these directory, it is a matter of copy and paste, if we want to use a XML file we have to create it.

There are several types of drawable:

- Bitmap
- Nine-patch
- State list
- Level list
- Transition drawable
- Inset drawable
- Clip drawable
- Scale drawable
- Shape drawable

### **Shape drawable**

This is a generic shape. Using XML we have to create a file with shape element as root. This element as an attribute called

`android:shape`

where we define the type of shape like rectangle, oval, line and ring. We can customize the shape using child elements like:

For example, let us suppose we want to create an oval with solid background color. We create a XML file called for example `oval.xml`:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android" android:shape="oval" >
<solid android:color="#FF0000" />
<size android:height="100dp" android:width="120dp" />
</shape>
```

### 6.1.3 Animations

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

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

There are 3 types of Animations:

#### a. Property Animation

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

The property animation system lets you define the following characteristics of an animation:

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

#### b. View Animation

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

A tween animation can perform a series of simple transformations (position, size, rotation, and transparency) on the contents of a View object. So, if you have a TEXTVIEW object, you can move, rotate, grow, or shrink the text. If it has a background image, the background image will be transformed along with the text. The animation package provides all the classes used in a tween animation.

A sequence of animation instructions defines the tween animation, defined by either XML or Android code. As with defining a layout, an XML file is recommended because it's more readable, reusable, and swappable than hard-coding the animation.

Example:

```
imageView.setOnClickListener(new View.OnClickListener {  
    @Override  
    public void onClick(View v){  
        imageView.animate()  
            .alpha(0f)  
            .duration = 2000;  
    });
```

### c. Drawable Animation

This is used to do animation using drawables. An XML file specifying various list of drawables is made which are run one by one just like a roll of a film.

One way to animate **Drawables** is to load a series of Drawable resources one after another to create an animation. This is a traditional animation in the sense that it is created with a sequence of different images, played in order, like a roll of film. The **AnimationDrawable** class is the basis for Drawable animations.

While you can define the frames of an animation in your code, using the AnimationDrawable class API, it's more simply accomplished with a single XML file that lists the frames that compose the animation. The XML file for this kind of animation belongs in the **res/drawable/** directory of your Android project. In this case, the instructions are the order and duration for each frame of the animation.

The XML file consists of an **<animation-list>** element as the root node and a series of child **<item>** nodes that each define a frame: a drawable resource for the frame and the frame duration. Here's an example XML file for a Drawable animation:

Example:

```

<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
android:oneshot="true">
<item android:drawable="@drawable/rocket_thrust1" android:duration="200" />
<item android:drawable="@drawable/rocket_thrust2" android:duration="200" />
<item android:drawable="@drawable/rocket_thrust3" android:duration="200" />
</animation-list>

```

And then in JAVA file

```

AnimationDrawable rocketAnimation;
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    ImageView rocketImage = (ImageView) findViewById(R.id.rocket_image);
    rocketImage.setBackgroundResource(R.drawable.rocket_thrust);
    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
    rocketImage.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            rocketAnimation.start();
        }
    });
}

```

## 6.2 Multitouch and Gestures

Gesture recognition and handling touch events is an important part of developing user interactions. Handling standard events such as clicks, long clicks, key presses, etc are very basic and handled in other guides. This guide is focused on handling other more specialized gestures such as:

- Swiping in a direction
- Double tapping for zooming
- Pinch to zoom in or out
- Dragging and dropping
- Effects while scrolling a list

### Handling Touches

At the heart of all gestures is the `onTouchListener` and the `onTouch` method which has access to `MotionEvent` data. Every view has an `onTouchListener` which can be specified:

```
myView.setOnTouchListener(new OnTouchListener() {

    @Override
    public boolean onTouch(View v, MotionEvent event) {

        // Interpret MotionEvent data

        // Handle touch here
        return true;
    }
});
```

Each onTouch event has access to the MotionEvent which describe movements in terms of an action code and a set of axis values. The action code specifies the state change that occurred such as a pointer going down or up. The axis values describe the position and other movement properties:

- getAction() - Returns an integer constant such as MotionEvent.ACTION\_DOWN, MotionEvent.ACTION\_MOVE, and MotionEvent.ACTION\_UP
- getX() - Returns the x coordinate of the touch event
- getY() - Returns the y coordinate of the touch event

Note that every touch event can be propagated through the entire affected view hierarchy. Not only can the touched view respond to the event but every layout that contains the view has an opportunity as well.

## Handling Multi Touch Events

Note that getAction() normally includes information about both the action as well as the pointer index. In single-touch events, there is only one pointer (set to 0), so no bitmap mask is needed. In multiple touch events (i.e pinch open or pinch close), however, there are multiple fingers involved and a non-zero pointer index may be included when calling getAction(). As a result, there are other methods that should be used to determine the touch event:

- getActionMasked() - extract the action event without the pointer index
- getActionIndex() - extract the pointer index used

The events associated with other pointers usually start with MotionEvent.ACTION\_POINTER such as MotionEvent.ACTION\_POINTER\_DOWN and MotionEvent.ACTION\_POINTER\_UP. The getPointerCount() on the MotionEvent can be used to determine how many pointers are active in this touch sequence.

## Gesture Detectors

Within an onTouch event, we can then use a GestureDetector to understand gestures based on a series of motion events. Gestures are often used for user interactions within an app. Let's take a look at how to implement common gestures.



For easy gesture detection using a third-party library, check out the popular Sensey library which greatly simplifies the process of attaching multiple gestures to your views.

## Double Tapping

You can enable double tap events for any view within your activity using the `OnDoubleTapListener`. First, copy the code for `OnDoubleTapListener` into your application and then you can apply the listener with:

```
myView.setOnTouchListener(new OnDoubleTapListener(this) {  
    @Override  
    public void onDoubleTap(MotionEvent e) {  
        Toast.makeText(MainActivity.this, "Double Tap", Toast.LENGTH_SHORT  
    ).show();  
    }  
});
```

Now that view will be able to respond to a double tap event and you can handle the event accordingly.

## Swipe Gesture Detection

Detecting finger swipes in a particular direction is best done using the built-in `onFling` event in the `GestureDetector.OnGestureListener`.

A helper class that makes handling swipes as easy as possible can be found in the `OnSwipeTouchListener` class. Copy the `OnSwipeTouchListener` class to your own application and then you can use the listener to manage the swipe events with:

```
myView.setOnTouchListener(new OnSwipeTouchListener(this) {  
    @Override  
    public void onSwipeDown() {  
        Toast.makeText(MainActivity.this, "Down", Toast.LENGTH_SHORT).show();  
    }  
  
    @Override  
    public void onSwipeLeft() {  
        Toast.makeText(MainActivity.this, "Left", Toast.LENGTH_SHORT).show();  
    }  
  
    @Override  
    public void onSwipeUp() {  
        Toast.makeText(MainActivity.this, "Up", Toast.LENGTH_SHORT).show();  
    }  
  
    @Override  
    public void onSwipeRight() {  
        Toast.makeText(MainActivity.this, "Right", Toast.LENGTH_SHORT).show();  
    }  
});
```

With that code in place, swipe gestures should be easily manageable.

## 6.3 Multimedia

The Android SDK provides a set of APIs to handle multimedia files, such as audio, video and images. Moreover, the SDK provides other API sets that help developers to implement interesting graphics effects, like animations and so on.

The modern smart phones and tablets have an increasing storage capacity so that we can store music files, video files, images. etc. Not only the storage capacity is important, but also the high definition camera makes it possible to take impressive photos. In this context, the Multimedia API plays an important role.

### Multimedia API

Android supports a wide list of audio, video and image formats. You can give a look [here](#) to have an idea; just to name a few formats supported:

#### Audio

- MP3
- MIDI
- Vorbis (es: mkv)

#### Video

- H.263
- MPEG-4 SP

#### Images

- JPEG
- GIF
- PNG

All the classes provided by the Android SDK that we can use to add multimedia capabilities to our apps are under the

*android.media package.*

In this package, the heart class is called MediaPlayer. This class has several methods that we can use to play audio and video file stored in our device or streamed from a remote server. This class implements a state machine with well-defined states and we have to know them before playing a file. Simplifying the state diagram, as shown in the official documentation, we can define these macro-states:

- **Idle state:** When we create a new instance of the MediaPlayer class.

- **Initialization state:** This state is triggered when we use `setDataSource` to set the information source that `MediaPlayer` has to use.
- **Prepared state:** In this state, the preparation work is completed. We can enter in this state calling `prepare` method or `prepareAsync`. In the first case after the method returns the state moves to Prepared. In the async way, we have to implement a listener to be notified when the system is ready and the state moves to Prepared. We have to keep in mind that when calling the `prepare` method, the entire app could hang before the method returns because the method can take a long time before it completes its work, especially when data is streamed from a remote server. We should avoid calling this method in the main thread because it might cause a ANR (Application Not Responding) problem. Once the `MediaPlayer` is in prepared state we can play our file, pause it or stop it.
- **Completed state:** The end of the stream is reached. We can play a file in several ways:

```
// Raw audio file as resource
MediaPlayer mp = MediaPlayer.create(this, R.raw.audio_file);

// Local
file
MediaPlayer mp1 = MediaPlayer.create(this, Uri.parse("file:///..."));

// Remote
file
MediaPlayer mp2 = MediaPlayer.create(this, Uri.parse("http://website.com"));
```

or

we can use `setDataSource` in this way:

```
MediaPlayer mp3 = new MediaPlayer();
mp3.setDataSource("http://www.website.com");
```

Once we have created our `MediaPlayer` we can “prepare” it:

```
mp3.prepare();
```

and finally we can play it:

```
mp3.start();
```

## Using Android Camera

If we want to add to our apps the capability to take photos using the integrated smart phone camera, then the best way is to use an Intent. For example, let us suppose we want to start the camera as soon as we press a button and show the result in our app.

In the onCreate method of our Activity, we have to setup a listener of the Button and when clicked to fire the intent:

```
Button b = (Button) findViewById(R.id.btn1);

b.setOnClickListener(new View.OnClickListener() {

@Override
public void onClick(View v) {
// Here we fire the intent to start the camera
Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(i, 100);
}
```

# 7 . Packaging and Monetizing

## 7.1 Persisting Data to the Device

### Overview

The Android framework offers several options and strategies for persistence:

- Shared Preferences**- Easily save basic data as key-value pairs in a private persisted dictionary.
- Local Files**- Save arbitrary files to internal or external device storage.
- SQLite Database**- Persist data in tables within an application specific database.

### Use Cases

Each storage option has typical associated use cases as follows:

- Shared Preferences**- Used for app preferences, keys and session information.
- Local Files**- Often used for blob data or data file caches (i.e disk image cache)
- SQLite Database**- Used for complex local data manipulation or for raw speed

### Shared Preferences

If you need to store simple data items for your apps, the most straightforward approach is to use Shared Preferences. This is possibly the easiest data management option to implement, but it's only suitable for primitive type items such as numbers and text. Using Shared Preferences, you model your data items as key value pairs. The following code demonstrates acquiring a reference to the SharedPreferences object for an app and writing a value to it:

```
//get the preferences, then editor, set a data item
SharedPreferences appPrefs = getSharedPreferences("MyAppPrefs", 0);
SharedPreferences.Editor prefsEd = appPrefs.edit();
prefsEd.putString("dataString", "some string data");
prefsEd.commit();
```

Anything you save to Shared Preferences will still be available the next time your app runs, so it's ideal for saving items such as user preferences and settings. When your app starts up, you can check the Shared Preferences, then present your interface and functionality accordingly. The following code demonstrates retrieving the string data item:

```
//get the preferences then retrieve saved data, specifying a default value
SharedPreferences appPrefs = getSharedPreferences("MyAppPrefs", 0);
String savedData = appPrefs.getString("dataString", "");
```

## Local Files

Android can read/write files to internal as well as external storage. Applications have access to an application-specific directory where preferences and sqlite databases are also stored. Every Activity has helpers to get the writeable directory. File I/O API is a subset of the normal Java File API.

Writing files is as simple as getting the stream using `openFileOutput` method and writing to it using a `BufferedWriter`:

```
// Use Activity method to create a file in the writeable directory
FileOutputStream fos = openFileOutput("filename", MODE_WORLD_WRITEABLE);

// Create buffered writer
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(fos));
writer.write("Hi, I'm writing stuff");
writer.close();
```

Reading the file back is then just using a `BufferedReader` and then building the text into a `StringBuffer`:

```
BufferedReader input = null;
input = new BufferedReader( new InputStreamReader(openFileInput("filename")));
String line;
StringBuffer buffer = new StringBuffer();
while ((line = input.readLine()) != null) {
    buffer.append(line + "\n");
}
String text = buffer.toString();
```

## Request external storage permissions

To write to the public external storage, you must request the `WRITE_EXTERNAL_STORAGE` permission in your manifest file:

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

</manifest>

## SQLite

For maximum control over local data, developers can use SQLite directly by leveraging SQLiteOpenHelper for executing SQL requests and managing a local database.

To build a custom SQLite database from your Java code, you can extend the SQLiteOpenHelper class, then define and create your tables inside the "onCreate" method, as follows:

```
public void onCreate(SQLiteDatabase db) {  
    db.execSQL("CREATE TABLE Item (ItemID INTEGER, ItemName TEXT);");  
}
```

This statement creates a database table with two columns in it. The SQLiteDatabase class provides the means to manage the data, including query, insert and update methods. One potential downside to using an SQLite database in your Android apps is the amount of processing code required, and the necessary skill set.

## Defining the Database Handler

We need to write our own class to handle database operations such as creation, upgrading, reading and writing. Database operations are defined using the SQLiteOpenHelper:

```
public class UsersDatabaseHelper extends SQLiteOpenHelper {  
  
    // Database Info  
private static final String DATABASE_NAME = "userDatabase";  
private static final int DATABASE_VERSION = 1;  
  
    // Table Names  
private static final String TABLE_USERS = "users";  
    // User Table Columns  
private static final String KEY_USER_ID = "id";  
private static final String KEY_USER_NAME = "userName";  
private static final String KEY_USER_PROFILE_PICTURE_URL = "profilePictureUrl";  
  
    public UsersDatabaseHelper(Context context) {  
        super(context, DATABASE_NAME, null, DATABASE_VERSION);  
    }  
  
    // Called when the database is created for the FIRST time.  
    // If a database already exists on disk with the same DATABASE_NAME, this  
    // method will NOT be called.  
    @Override  
    public void onCreate(SQLiteDatabase db) {  
  
        String CREATE_USERS_TABLE = "CREATE TABLE " + TABLE_USERS +
```

```

        "(" +
        KEY_USER_ID + " INTEGER PRIMARY KEY," +
        KEY_USER_NAME + " TEXT," +
        KEY_USER_PROFILE_PICTURE_URL + " TEXT" +
        ")";

        db.execSQL(CREATE_USERS_TABLE);
    }

    // Called when the database needs to be upgraded.
    // This method will only be called if a database already exists on disk with
    // the same DATABASE_NAME,
    // but the DATABASE_VERSION is different than the version of the database that
    // exists on disk.

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        if (oldVersion != newVersion) {
            // Simplest implementation is to drop all old tables and recreate them
            db.execSQL("DROP TABLE IF EXISTS " + TABLE_USERS);
            onCreate(db);
        }
    }
}

```

**Important Note:** The SQLite database is **lazily initialized**. This means that it isn't actually created until it's first accessed through a call to **getReadableDatabase()** or **getWritableDatabase()**. This also means that any methods that call **getReadableDatabase()** or **getWritableDatabase()** should be done on a background thread as there is a possibility that they might be kicking off the initial creation of the database.

## CRUD Operations (Create, Read, Update, Delete)

### Inserting New Records

```

public class UsersDatabaseHelper extends SQLiteOpenHelper {

    // Insert a post into the database
    public void addPost(User user) {
        // Create and/or open the database for writing
        SQLiteDatabase db = getWritableDatabase();

        try{
            ContentValues values = new ContentValues();
            values.put(KEY_USER_ID, user.getId());
            values.put(KEY_USER_NAME, user.getName());
            values.put(KEY_USER_PROFILE_PICTURE_URL, user.getProfilePic());
            db.insert(TABLE_USERS, null, values);
        } catch (Exception e) {
            Log.d(TAG, "Error while trying to add post to database");
        }
    }
}

```



## Querying Records

We can retrieve anything from database using an object of the Cursor class. We will call a method of this class called `rawQuery` and it will return a resultset with the cursor pointing to the table. We can move the cursor forward and retrieve the data.

```
public List<User> getAllUsers() {
    List<User> users = new ArrayList<>();

    // SELECT * FROM Users
    String USER_SELECT_QUERY =
        String.format("SELECT * FROM %s ", TABLE_USERS);

    // "getReadableDatabase()" and "getWritableDatabase()" return the same object
    (except under low disk space scenarios)
    SQLiteDatabase db = getReadableDatabase();
    Cursor resultSet = db.rawQuery(POSTS_SELECT_QUERY, null);
    try {
        if (resultSet.moveToFirst()) {
            do {
                User newUser = new User();
                newUser.userName = cursor.getString(resultSet
.getColumnIndex(KEY_USER_NAME));
                newUser.profilePictureUrl = cursor.getString(resultSet
.getColumnIndex(KEY_USER_PROFILE_PICTURE_URL));
                users.add(newUser);

            } while(resultSet.moveToNext());
        }
    } catch (Exception e) {
        Log.d(TAG, "Error while trying to get users from database");
    } finally {
        if (resultSet != null && !resultSet.isClosed()) {
            resultSet.close();
        }
    }
    return users;
}
```

## Updating Records

```
// Update the user's profile picture url
public int updateUserProfilePicture(User user) {
    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(KEY_USER_PROFILE_PICTURE_URL, user.profilePictureUrl);

    // Updating profile picture url for user with that userName
    return db.update(TABLE_USERS, values, KEY_USER_NAME + " = ?",
        new String[] { String.valueOf(user.userName) });
}
```

```

public class UserDatabaseHelper extends SQLiteOpenHelper {

    public void deleteAllUsers() {
        SQLiteDatabase db = getWritableDatabase();
        db.beginTransaction();
        try {
            db.delete(TABLE_USERS, null, null);
        } catch (Exception e) {
            Log.d(TAG, "Error while trying to delete all posts and users");
        }
    }
}

```

## 7.1 The Content Provider Class

Content providers are Android's central mechanism that enables you to access data of other applications – mostly information stored in databases or flat files. As such content providers are one of Android's central component types to support the modular approach common to Android. Without content providers accessing data of other apps would be a mess.

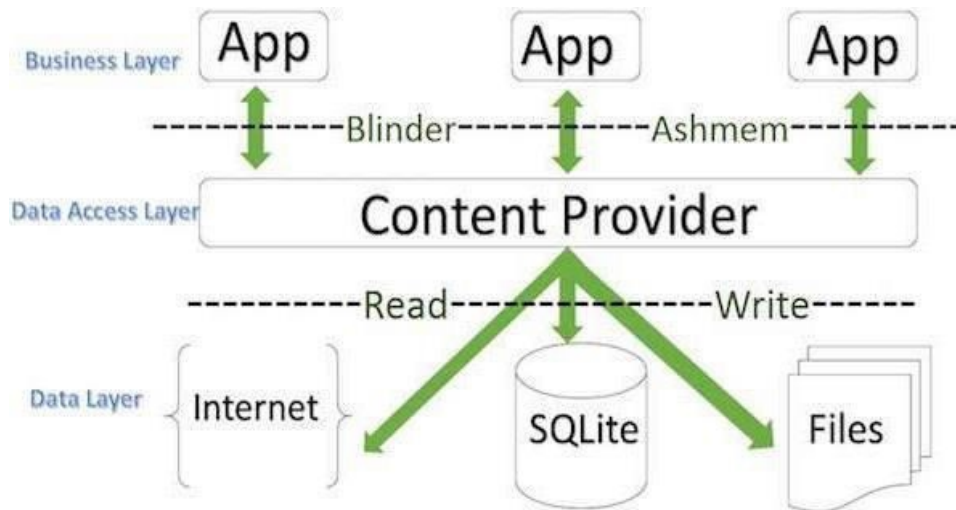
Content providers support the four basic operations, normally called CRUD-operations. CRUD is the acronym for create, read, update and delete. With content providers those objects simply represent data – most often a record (tuple) of a database – but they could also be a photo on your SD-card or a video on the web.

Content providers are one of the primary building blocks of Android applications, providing content to applications. They encapsulate data and provide it to applications through the single ContentResolver interface.

A content provider is only required if you need to share data between multiple applications. For example, the contacts data is used by multiple applications and must be stored in a content provider. If you don't need to share data amongst multiple applications you can use a database directly via SQLiteDatabase.

A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentResolver class. A content provider can use different ways to store its data and the data can be stored in a database, in files, or even over a network.

When a request is made via a ContentResolver the system inspects the authority of the given URI and passes the request to the content provider registered with the authority. The content provider can interpret the rest of the URI however it wants. The UriMatcher class is helpful for parsing URIs.



Content providers let you centralize content in one place and have many different applications access it as needed. A content provider behaves very much like a database where you can query it, edit its content, as well as add or delete content using `insert()`, `update()`, `delete()`, and `query()` methods. In most cases this data is stored in a **SQLite** database.

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 MyApplication extends ContentProvider{

}

```

Requests to `ContentResolver` are automatically forwarded to the appropriate `ContentProvider` instance, so subclasses don't have to worry about the details of cross-process calls.

## Creating Content Provider

This involves number of simple steps to create your own content provider.

- To create a content provider in android applications we should follow below steps.
- We need to create a content provider class that extends the `ContentProvider` base class.
- We need to define our content provider URI to access the content.
- The `ContentProvider` class defines a six abstract methods (`insert()`, `update()`, `delete()`, `query()`, `getType()`) which we need to implement all these methods as a part of our subclass.
- We need to register our content provider in `AndroidManifest.xml` using **<provider>** tag.

Following are the list of methods which need to implement as a part of ContentProvider class.

**query()**- It receives a request from the client. By using arguments it will get a data from requested table and return the data as a Cursor object.

**insert()**- This method will insert a new row into our content provider and it will return the content URI for newly inserted row.

**update()**- This method will update an existing rows in our content provider and it return the number of rows updated.

**delete()**- This method will delete the rows in our content provider and it return the number of rows deleted.

**getType()**- This method will return the MIME type of data to given content URI.

**onCreate()**- This method will initialize our provider. The android system will call this method immediately after it creates our provider.

## Android Content Provider Example

Following is the example of using **Content Provider** in android applications. Here we will create our own content provider to insert and access data in android application.

```
public class UsersProvider extends ContentProvider {
    static final String PROVIDER_NAME = "com.ocem.sampelApp.UserProvider";
    static final String URL = "content://" + PROVIDER_NAME + "/users";
    static final Uri CONTENT_URI = Uri.parse(URL);
    static final String id = "id";
    static final String name = "name";
    static final int uriCode = 1;
    static final UriMatcher uriMatcher;
    private static HashMap<String, String> values;
    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI(PROVIDER_NAME, "users", uriCode);
        uriMatcher.addURI(PROVIDER_NAME, "users/*", uriCode);
    }

    @Override
    public String getType(Uri uri) {
        return "vnd.android.cursor.dir/users"; default:
    }

    @Override
    public boolean onCreate() {
        Context context = getContext();
        DatabaseHelper dbHelper = new DatabaseHelper(context);
        db = dbHelper.getWritableDatabase();
        if (db != null) {
            return true;
        }
    }
}
```

```

        }
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
        qb.setTables(TABLE_NAME);
        qb.setProjectionMap(values);

        if (sortOrder == null || sortOrder == "") {
            sortOrder = id;
        }
        Cursor c = qb.query(db, projection, selection, selectionArgs, null,
            null, sortOrder);
        c.setNotificationUri(getContext().getContentResolver(), uri);
        return c;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        long rowID = db.insert(TABLE_NAME, "", values);
        if (rowID > 0) {
            Uri _uri = ContentUris.withAppendedId(CONTENT_URI, rowID);
            getContext().getContentResolver().notifyChange(_uri, null);
            return _uri;
        }
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
        String[] selectionArgs) {
        int count = 0;
        count = db.update(TABLE_NAME, values, selection,
            selectionArgs);
        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }

    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        int count = 0;
        count = db.delete(TABLE_NAME, selection, selectionArgs);
        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }

    private SQLiteDatabase db;
    static final String DATABASE_NAME = "EmpDB";
    static final String TABLE_NAME = "Employees";
    static final int DATABASE_VERSION = 1;
    static final String CREATE_DB_TABLE = " CREATE TABLE " + TABLE_NAME
    + " (id INTEGER PRIMARY KEY AUTOINCREMENT, "
    + " name TEXT NOT NULL);";
    private static class DatabaseHelper extends SQLiteOpenHelper {
        DatabaseHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            db.execSQL(CREATE_DB_TABLE);
        }
    }

```

```

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

```

## 7.2 The Service Class

### 7.2.1 What are services?

A *service* is a component which runs in the background without direct interaction with the user. As the service has no user interface, it is not bound to the lifecycle of an activity.

Services are used for repetitive and potentially long running operations, i.e., Internet downloads, checking for new data, data processing, updating content providers and the like.

Services run with a higher priority than inactive or invisible activities and therefore it is less likely that the Android system terminates them. Services can also be configured to be restarted if they get terminated by the Android system once sufficient system resources are available again.

It is possible to assign services the same priority as foreground activities. In this case it is required to have a visible notification active for the related service. It is frequently used for services which play videos or music.

### 7.2.2 Services and background processing

By default, a service runs in the same process as the main thread of the application.

Therefore, you need to use asynchronous processing in the service to perform resource intensive tasks in the background. A commonly used pattern for a service implementation is to create and run a new Thread in the service to perform the processing in the background and then to terminate the service once it has finished the processing.

Services which run in the process of the application are sometimes called local services.

### 7.2.3 Platform Service and Custom Services

The Android platform provides and runs predefined system services and every Android application can use them, given the right permissions. These system services are usually exposed via a specific Manager class. Access to them can be gained via the `getSystemService()` method. The Context class defines several constants for accessing these services.

An Android application can, in addition to consuming the existing Android platform services, define and use new services. Defining your custom services allows you to design responsive applications. You can

fetch the application data via it and once the application is started by the user, it can present fresh data to the user.

Custom services are started from other Android components, i.e., activities, broadcast receivers and other services.

## 7.2.4 Foreground services

A foreground service is a service that should have the same priority as an active activity and therefore should not be killed by the Android system, even if the system is low on memory. A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

```
Notification notification = new Notification(R.drawable.icon,
    getText(R.string.ticker_text), System.currentTimeMillis());
Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent,
    0);
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);
startForeground(ONGOING_NOTIFICATION_ID, notification);
```

## 7.3 Defining custom services

### 7.3.1 Implementation and declaration

A service needs to be declared in the *AndroidManifest.xml* file and the implementing class must extend the Service class or one of its subclasses.

The following code shows an example for a service declaration and its implementation.

In manifest:

```
<service android:name="MyService"
    android:icon="@drawable/icon"
    android:label="@string/service_name" >

</service>
```

In Java file:

```
public class MyService extends Service {
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        //TODO do something useful
        return Service.START_NOT_STICKY;
    }
    @Override
    public IBinder onBind(Intent intent) {
        //TODO for communication return IBinder implementation
        return null;
    }
}
```

### 7.3.2 Start a service

An Android component (service, receiver, activity) can trigger the execution of a service via the `startService(intent)` method.

```
// use this to start and trigger a service
Intent i= new Intent(context, MyService.class);

// potentially add data to the intent
i.putExtra("KEY1", "Value to be used by the service");
context.startService(i);
```

Alternatively, you can also start a service via the `bindService()` method call. This allows you to communicate directly with the *service*.

### 7.3.3 Service start process and execution

If the `startService(intent)` method is called and the service is not yet running, the service object is created and the `onCreate()` method of the service is called.

Once the *service* is started, the `onStartCommand(intent)` method in the *service* is called. It passes in the `Intent` object from the `startService(intent)` call.

If `startService(intent)` is called while the service is running, its `onStartCommand()` is also called. Therefore your service needs to be prepared that `onStartCommand()` can be called several times.

A service is only started once, no matter how often you call the `startService()` method.



### 7.3.4 Stopping a service

You stop a service via the `stopService()` method. No matter how frequently you called the `startService(intent)` method, one call to the `stopService()` method stops the service.

A service can terminate itself by calling the `stopSelf()` method. This is typically done if the service finishes its work.

## 7.4 Google Mobile Ads SDK

The Google Mobile Ads SDK for Ad Manager is a mobile advertising platform that you can use to generate revenue from your app.

The Google Mobile Ads SDK for Ad Manager is a mobile advertising platform that you can use to generate revenue from your app.

### Key capabilities

#### **Earn more from Ad Manager's in-app ads**

Show ads from millions of Google advertisers in real time, or use Mediation to earn from over 40 premium networks through the platform to simplify your ad operations, improve competition, and earn more.

#### **Improve user experience**

Mediation has ad network optimization built in, which automatically adjusts the positions of your other ad networks in your mediation stack to ensure you maximize your revenue.

Native and video ads create a positive user experience as you monetize by matching the look and feel of your app. Choose from different ad templates, customize them, and experiment with different layouts on the fly without republishing your app.

When your app's a global or domestic hit, you can monetize users quickly with Ad Manager, by showing ads to users in more than 200 markets.

#### **Scale fast**

More than one app? Ad Manager house ads is a free tool that enables you to cross-promote your apps to your userbase, across your family of apps.

### How does it work?

Ad Manager helps you monetize your mobile app through in-app advertising. Ads can be displayed as banner, interstitial, video, or native ads—which are seamlessly added to platform native UI components.

Before you can display ads within your app, you'll need an Ad Manager account and activate one or more ad unit IDs. This is a unique identifier for the places in your app where ads are displayed.

Ad Manager uses the Google Mobile Ads SDK which helps app developers gain insights about their users and maximize ad revenue. To do so, the default integration of the Mobile Ads SDK collects information such as device information and publisher-provided location information.

## Implementing in Android Studio

Integrating the Google Mobile Ads SDK into an app is the first step toward displaying ads and earning revenue. Once you've integrated the SDK, you can choose an ad format (such as native or rewarded video) and follow the steps to implement it.

### Import the Mobile Ads SDK

Apps can import the Google Mobile Ads SDK with a Gradle dependency that points to Google's Maven repository. In order to use that repository, you need to reference it in the app's project-level build.gradle file. Open yours and look for an allprojects section:

Example app-level build.gradle (excerpt)

```
dependencies {  
    implementation fileTree(dir: 'libs', include: ['*.jar'])  
    implementation 'com.android.support:appcompat-v7:26.1.0'  
    implementation 'com.google.android.gms:play-services-ads:17.2.1'  
}
```

Add the line in bold above, which instructs Gradle to pull in the latest version of the Mobile Ads SDK. Once that's done, save the file and perform a Gradle sync.

### Update your AndroidManifest.xml

Declare that your app is an Ad Manager app by adding the following <meta-data> tag in your AndroidManifest.xml.

```
<manifest>  
    <application>  
        <meta-data  
            android:name="com.google.android.gms.ads.AD_MANAGER_APP"  
            android:value="true"/>  
        </application>  
</manifest>
```

### Select an ad format

The Mobile Ads SDK is now imported and you're ready to implement an ad. Ad Manager offers a number of different ad formats, so you can choose the one that best fits your app's user experience.

Ads formats provided by Google are:

- **Banner:** Banner ads are rectangular image or text ads that occupy a spot within an app's layout. They stay on screen while users are interacting with the app, and can refresh automatically after a certain period of time.

- **Interstitial:** Interstitials are full-screen ads that cover the interface of an app until closed by the user. They're best used at natural pauses in the flow of an app's execution, such as between levels of a game or just after a task is completed.
- **Native:** Native is a component-based ad format that gives you the freedom to customize the way assets such as headlines and calls to action are presented in your apps. By choosing fonts, colors, and other details for yourself, you can create natural, unobtrusive ad presentations that can add to a rich user experience.
- **Rewarded:** Rewarded video ads are full-screen video ads that users have the option of watching in their entirety in exchange for in-app rewards.

Ads can be created on both xml file or in Java file

In XML file:

```
<com.google.android.gms.ads.doubleclick.PublisherAdView
    xmlns:ads="http://schemas.android.com/apk/res-auto"
    android:id="@+id/publisherAdView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_alignParentBottom="true"
    ads:adSize="BANNER"
    ads:adUnitId="/6499/example/banner">
</com.google.android.gms.ads.doubleclick.PublisherAdView>
```

In Java file:

```
PublisherAdView adView = new PublisherAdView(this);
adView.setAdSizes(AdSize.BANNER);

adView.setAdUnitId("/6499/example/banner");
```

## 7.5 Signing and Exporting an App

Android requires that all apps be digitally signed with a certificate before they can be installed. Android uses this certificate to identify the author of an app, and the certificate does not need to be signed by a certificate authority. Android apps often use self-signed certificates. The app developer holds the certificate's private key.

You can sign an app in debug or release mode. You sign your app in debug mode during development and in release mode when you are ready to distribute your app. The Android SDK generates a certificate to sign apps in debug mode. To sign apps in release mode, you need to generate your own certificate.

Signing in Debug Mode

In debug mode, you sign your app with a debug certificate generated by the Android SDK tools. This certificate has a private key with a known password, so you can run and debug your app without typing the password every time you make a change to your project.

Android Studio signs your app in debug mode automatically when you run or debug your project from the IDE. You can run and debug an app signed in debug mode on the emulator and on devices connected to your development machine through USB, but you cannot distribute an app signed in debug mode.

By default, the *debug* configuration uses a debug keystore, with a known password and a default key with a known password. The debug keystore is located in `$HOME/.android/debug.keystore`, and is created if not present. The debug build type is set to use this debug `SigningConfig` automatically.

### Signing in Release Mode

In release mode, you sign your app with your own certificate:

- *Create a keystore.* A **keystore** is a binary file that contains a set of private keys. You must keep your keystore in a safe and secure place.
- *Create a private key.* A **private key** represents the entity to be identified with the app, such as a person or a company.
- Add the signing configuration to the build file for the app module
- Invoke the `assembleRelease` build task from Android Studio.

The package in `app/build/apk/app-release.apk` is now signed with your release key.

## Publishing an App to the Play Store

There are basically three things to consider before publishing an android app.

- Registering for a Google Play publisher account
- Setting up a Google payments merchant account, if you will sell apps or in-app products.
- Exploring the Google Play Developer Console and publishing tools.

### 1. Register for a Publisher Account

1. Visit the Google Play Developer Console.
2. Enter basic information about your **developer identity** — name, email address, and so on. You can modify this information later.
3. Read and accept the **Developer Distribution Agreement** for your country or region. Note that apps and store listings that you publish on Google Play must comply with the Developer Program Policies and US export law.
4. Pay a **\$25 USD registration fee** using Google payments. If you don't have a Google payments account, you can quickly set one up during the process.

- When your registration is verified, you'll be notified at the email address you entered during registration.

## 2. Set Up a Google Payments Merchant Account

If you want to sell priced apps, in-app products, or subscriptions, you'll need a Google payments merchant account.

You can set one up at any time, but first review the list of merchant countries.

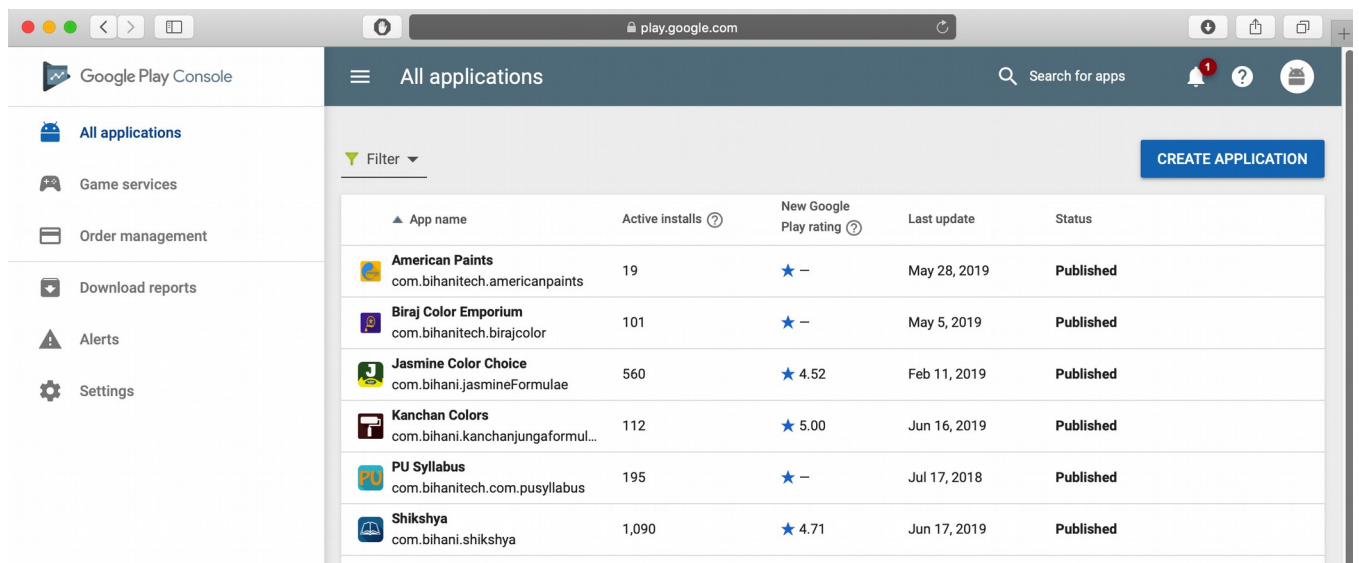
To set up a Google payments merchant account:

- Sign in** to your Google Play Developer Console at <https://play.google.com/apps/publish/>.
- Open **Financial reports** on the side navigation.
- Click **Setup a Merchant Account now**.

This takes you to the Google payments site; you'll need information about your business to complete this step.

## 3. Explore the Developer Console

When your registration is verified, you can sign in to your Developer Console, which is the home for your app publishing operations and tools on Google Play



The screenshot displays the Google Play Developer Console interface. On the left is a sidebar with navigation options: Google Play Console, All applications, Game services, Order management, Download reports, Alerts, and Settings. The main area is titled 'All applications' and features a search bar, a filter dropdown, and a 'CREATE APPLICATION' button. Below these is a table listing several applications with their details.

| App name  | Active installs | New Google Play rating | Last update  | Status    |
|---|-----------------|------------------------|--------------|-----------|
| <b>American Paints</b><br>com.bihanitech.americanpaints   | 19              | ★ —                    | May 28, 2019 | Published |
| <b>Biraj Color Emporium</b><br>com.bihanitech.birajcolor  | 101             | ★ —                    | May 5, 2019  | Published |
| <b>Jasmine Color Choice</b><br>com.bihani.jasmineFormulae | 560             | ★ 4.52                 | Feb 11, 2019 | Published |
| <b>Kanchan Colors</b><br>com.bihani.kanchanjungaformul... | 112             | ★ 5.00                 | Jun 16, 2019 | Published |
| <b>PU Syllabus</b><br>com.bihanitech.com.pusyllabus       | 195             | ★ —                    | Jul 17, 2018 | Published |
| <b>Shikshya</b><br>com.bihani.shikshya                    | 1,090           | ★ 4.71                 | Jun 17, 2019 | Published |

