Android -Introduction

* Welcome to the session on android programming!!
* Android was founded in Oct 2003 by Andy Rubin and his team including Rich Miner,Nick Sears and Chris White as Android Inc.
* Android means..” *A robot with human appearance*.”
* Later in Aug 2005 Google acquired Android Inc.
* Android was initially conceptualised for Digital cameras..But due to the smaller market for these when compared to the cell phones, the company focused and switched to cell phone designing..
* *The T-Mobile G1 (HTC Dream) was the first consumer device ever to run Android ( with version 1.0.)*

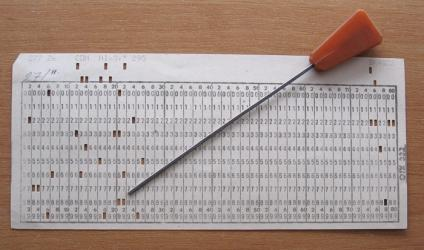
<https://www.androidcentral.com/android-history>

* First and foremost let us be clear that Android by itself is NOT a programming language.
* Android is well-known as an **open-source operating system**, meaning anyone can download the Android source code and build their own version of the OS. And the Android Open Source Project (AOSP) is how that's done. Once Google is done developing Android internally, it's publicly released through AOSP, allowing anyone from big manufacturers to hobbyist developers to tinker with the code.
* Android is a derived version of Linux as Linux was the derived version of unix.
* Unix was basically used for server machines and had all the i/o devices connected separately.
* Where as Android has all the interface devices integrated within one device.
* Android is a platform that was Internet-based and giving consumers an opportunity to put the Internet in their pockets.
* The first major updates to Android after the initial release were *versions 1.5 (Cupcake)* in Apr 2009 with touch screen only Android phones with built-in Keyboard.
* Android *1.6 (Donut)* later that year.*.* had quick search box which brought Google's mission statement of "organizing the world's information" to smartphones, with the ability to search not only the web, but contacts, music, apps and app data from one central location.
* These established the trend of naming Android versions in a cronological order after "sweet treats"..
* Eclair(2.0,2.1);
* Froyo(2.2);
* GingerBread(2.3);
* Honeycomb(3.0);
* Ice cream Sandwich(4.0);
* JellyBean(4.1,4.2,4.3);
* Kitkat(4.4);
* Lollipop(5.0,5.1);
* Marshmallow(6.0);
* Nougat(7.0,7.1);
* Oreo(8.0,8.1);
* Pie(9.0);

What is Android?

Prior to Android let us go thru the **history or timeline of Operating Systems.**

1. The earliest computers in **1890’s** were **mainframes** that ***lacked any form of operating system***. Each user use on punched paper cards and magnetic or paper tape.



1. The ***first operating systems (60th-70th years)***

Each manufacturer had such proprietary operating system, which differed from each other completely, and the operator had to always relearn everything from scratch.

The First OS for more computer came up in **1964**



1. In 1969 multi tasking multi-user operating system called **UNIX** started from *MULTICS* which means *Multiplexed Operating and Computing System*

which was bigger and complicated. Then A team of Bell Labs researchers led by Thompson and Ritchie, including Rudd Canaday, implemented a hierarchical file system, the concepts of computer processes and device files, a command-line interpreter, and some small utility programs, modeled on the corresponding features in Multics, but simplified. The resulting system, much smaller and simpler than Multics, was to become **Unix.**



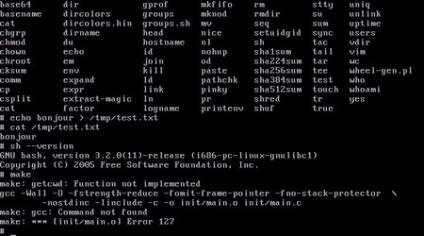
1. In **1978 Apple Dos** first appeared on Apple II computers and strongly influenced by the way it evolved operating systems over the next decade.



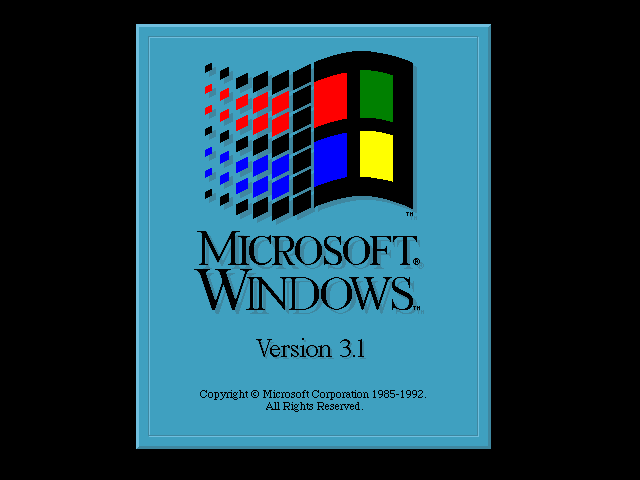
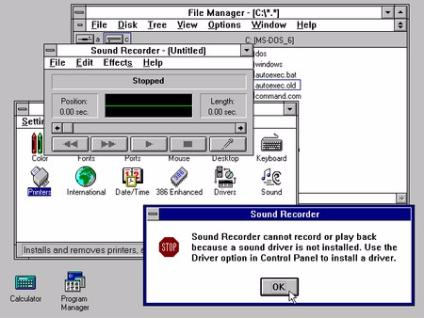
1. **1981 MS DOS (Microsoft Disk Operating Systems)**  was developed for x86 personal computers by Microsoft. MS-DOS was the main operating system for **IBM PC** compatible personal computers during the 1980s and the early 1990s.



1. **In 1991** an operating system kernel **Linux** is a family of open source Unix-like operating systems. It was originally developed for PC’s based on Intel x86 architecture. But since it was ported to more platforms than any other OS it is the leading OS for servers.



1. **1992 Windows 3.1** was released which had 16 bit operating environment. Truetype font was added and the dependency on third party font technology like Adobe Type manager was ended. Fonts like *Arial, Courier New, and Times New Roman, in regular, bold, italic, and bold-italic versions, as well as Symbol (a collection of scalable symbols)* were included.This effectively made Windows a *viable desktop publishing platform* for the first time.



1. **1995 Windows 95:** It was considered a ***complete operating system***, as it was not a graphical interface running on MS-DOS, and it also performed without any need of the MS-DOS environment after the boot process.

Windows 95 was capable of running **DOS and Windows-based** applications, although it had completely removed DOS as the underlying platform, unlike previous versions of Windows.

**Shortcuts, icons and the recycle bin** were introduced in Windows 95.

The **"plug & play"** feature was introduced, which allowed automatic recognition of hardware.

Another user-friendly feature introduced from Windows 95 was the representation of **files and folders as icons**. File modification was possible through menus and **the drives** were all listed in the folder called "**My Computer.**"



1. **1998 Windows 98:**A very significant feature in Windows 98 was the ***Web-based interface.***

***Active Desktop*** was introduced, which provided users the ability to customize the desktop with the look and feel of the Internet.

***Windows Media Player*** was introduced

It introduced ***Web-based applications*** like FrontPage, Windows Chat, Internet Explorer 4.01 and Outlook Express.

There was greater emphasis on ***security***,with ***auto-backing of the registry*** feature and enhanced ***networking***.

It also had the ***capability to convert the drive to FAT32 with no loss of data***

The ***disk cleanup tool*** was introduced, which helped in removing unnecessary files from the system.

1. **2001 Windows XP:** It was introduced in 2001 by Microsoft.

The "XP" in Windows XP stands for ***eXPerience***.It is the most important product since Windows 95 says Microsoft..

It comes in two versions,*Windows XP Home* and ***Windows XP Professional****.*

It included ***plug and play*** features for connecting to ***wireless networks***.

1. **2001Mac OS X**: MacOS is based on the **Unix** operating system. The "**X**" in Mac OS X and OS X is the Roman numeral for the number **10** and is pronounced as such. It is the ***second most widely used desktop OS***, after Microsoft Windows.



1. **2003 Android Inc.,:** The Android OS was originally created by Android, Inc., which was acquired by Google in 2005. It is based on **Linux kernel** and mostly used for **touch screen** based smart phones, tablets, supports many no.of applications in phones etc
2. **2007 Android:** Android was unveiled in 2007, with the [first commercial Android device](https://en.wikipedia.org/wiki/HTC_Dream) launched in September 2008.

Android has been the ***best-selling OS worldwide*** on smartphones since 2011 and on tablets since 2013. As of May 2017, it has over two billion [monthly active users](https://en.wikipedia.org/wiki/Monthly_active_users), the largest [installed base](https://en.wikipedia.org/wiki/Installed_base) of any operating system, and as of December 2018, the [Google Play](https://en.wikipedia.org/wiki/Google_Play) store features over 2.6 million apps!!!

<https://www.sutori.com/story/the-history-of-operating-systems--RrCp3bZLea46JfawcKHzXTzB>

Differences between Linux and Android.

Why Android is being compared only with Linux? And not any other OS?

SInce Android is derived from Linux, we compare Android with it.

Some of the major Difference Between Linux and Android are

|  |  |
| --- | --- |
| **Linux** | **Android** |
| Linux is developed majorly for personal and office system users(Big machines) | Android is peculiarly for mobile and tablet kind of devices(small devices) |
| Comparatively lesser footprint than Android | Holds a larger footprint than Linux |
| Linux systems use magnetic drives for memory | Android systems use flash memory |
| Does not use Virtual Machines for execution | Uses Dalvik VM(standard Java platform) and JVM |
| Uses GNU C library | Uses its own C library |
| Supports several architectures | Supports only 2 architectures ARM(platform for mobile phones) and  X86 (platform for Mobile Internet Devices (MID) used for desktops, servers, laptops) |
| Storage memory is high | Storage memory is limited. |
| Systems need external power backup. | Devices have limited battery power |

<https://www.educba.com/linux-vs-android/>

<https://www.westfloridacomponents.com/blog/android-vs-linux-os-better/>

# About Operating Systems

## Basics of OS:

## **Operating System** (**OS**) is a vital component of the system software that manages and allocates software resources which include the central processing unit (CPU), computer memory, file storage, input/output (I/O) devices, and network connections.An Operating System (OS) is an ***interface*** between a computer user and computer hardware.

## Role of OS: The operating system keeps things running smoothly across multiple hardware configurations, freeing up software developers to concentrate on making the best program they can write. The Kernel provides basic-level control over all of the computer hardware devices. **Main roles** include reading data from memory and writing data back to the memory, processing execution orders, determining how data is received and sent by devices such as the monitor, keyboard and mouse, and determining how to interpret data received from networks.

The ***kernel*** is the big chunk of executable code in charge of handling requests from system resources, like computing power, memory, network connectivity, or some other resource.

The **kernel's** role can be split into the following parts:

***Process management***

*The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the* ***scheduler****, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.*

***Memory management***

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance.The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc*/*free* pair to much more complex functionalities.

***Filesystems***

The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium.

***Device control***

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a device driver. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape drive.

***Networking***

Networking must be managed by the operating system, because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

***Security:*** By means of password and similar other techniques, it prevents unauthorized access to programs and data.

***Control over system performance****:* Recording delays between request for a service and response from the system.

***Job accounting****:*Keeping track of time and resources used by various jobs and users.

***Error detecting aids:*** Production of dumps, traces, error messages, and other debugging and error detecting aids.

***Coordination between other software and users:***Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

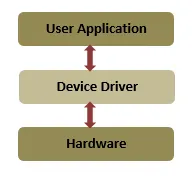
In all it loads and runs the programs.

<https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch01.html>

## Components of OS: The operating Systems comprises of the various **drivers** and **power management units**.

### **Device driver**

If any user-application wants to interact with the hardware, it must go through the corresponding device driver only and not directly as it might cause any damage to the hardware.



With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a ***device driver***. The kernel must have embedded in it a device driver for every peripheral present on a system,

A device driver (often referred to as driver’) is a piece of software that controls a particular type of device which is connected to the computer system. It provides a software interface to the hardware device, and enables access to the operating system and other applications.The device drivers can be mainly classified into three groups as below :-

1. **Character driver** – This category deals with the character devices i.e., which transfer data character-by-character. Examples include consoles, serial port, sensors etc.
2. **Block driver** – For the devices that transfer data by a block size (thus called block devices), the corresponding driver is called block device driver. For example, CD-ROM, USB devices etc.
3. **Network driver** – The devices that allows us to connect to a network and use the network interface services (called network devices) have a network driver in place for them. Example – Ethernet card, NIC (Network Interface Card) etc.

**Character drivers** are the most common drivers. They provide unbuffered, direct access to hardware devices. One can think of character drivers as a long sequence of bytes — same as regular files but can be accessed only in sequential order. Character drivers support at least the *open(), close(), read() and write()* operations.

**To make a device usable there must be a driver present for it.** So let us understand how an application accesses data from a device with the help of a driver. We will discuss the following four major entities.

***User-space application:*** This can be any simple utility like echo, or any complex application.

***Device file:*** This is a special file that provides an interface for the driver. It is present in the file system as an ordinary file. The application can perform all supported operation on it, just like for an ordinary file. It can move, copy, delete, rename, read and write these device files.

***Device driver:*** This is the software interface for the device and resides in the kernel space.

***Device:*** This can be the actual device present at the hardware level, or a pseudo device.

To send or receive data to and from the device or application, use the corresponding device file that is connected to the driver through the **Virtual File System (VFS) layer.** Whenever an application wants to perform any operation on the actual device, it performs this on the device file. The VFS layer redirects those operations to the appropriate functions that are implemented inside the driver. This means that whenever an application performs the open() operation on a device file, in reality the open() function from the driver is invoked, and the same concept applies to the other functions. The implementation of these operations is device-specific.

***But how does communication between the device file and the driver take place?*** It happens via a pair of numbers referred to as ***major and minor numbers.***

The major number identifies the driver associated with the device, i.e., which driver is to be used. The minor number is used by the kernel to determine exactly which device is being referred to.

*Two or more devices can have the same major number (if they are associated with the same driver) but not the same minor number.*

In a nutshell, minor number specifies the device and major number identifies the driver. **These both numbers come into picture while writing a device driver.**

For instance, a hard disk may have three partitions. Each partition will have a separate minor number but only one major number, because the same storage driver is used for all the partitions.

<https://opensourceforu.com/2014/10/an-introduction-to-device-drivers-in-the-linux-kernel/>

[***https://www.vlsifacts.com/device-drivers-role-and-types/***](https://www.vlsifacts.com/device-drivers-role-and-types/)

### **Power Management:**

Battery life is a perennial user concern. To extend battery life, Android continually adds new features to help the platform optimize the off-charger behavior of applications and devices.

Android includes the following battery life enhancements:

* [App Restrictions](https://source.android.com/devices/tech/power/app_mgmt" \l "app-restrictions). The platform can suggest apps that negatively affect battery life, so that users can choose to restrict those apps from consuming resources. Apps are not background restricted by default.
* [App Standby](https://source.android.com/devices/tech/power/app_mgmt" \l "app-standby). The platform can place unused applications in App Standby mode, temporarily restricting network access and deferring syncs and jobs for those applications.
* [Doze](https://source.android.com/devices/tech/power/platform_mgmt" \l "doze). The platform can enter a state of deep sleep (periodically resuming normal operations) if users have not actively used their device (screen off and stationary) for extended periods of time. Android 7.0 and later also enables Doze to trigger a lighter set of optimizations when users turn off the device screen yet continue to move around.
* [Exemptions](https://source.android.com/devices/tech/power/mgmt" \l "exempt-apps). Preloaded system apps and cloud messaging services are typically exempted from App Standby and Doze by default. App developers can use Intents to apply these settings to their apps. Users can exempt apps from App Standby and Doze power-saving modes in the Settings menu.

<https://source.android.com/devices/tech/power/mgmt>

#<http://people.cse.nitc.ac.in/jerrin/files/android_power_management.pdf>

# <https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf>

How a conventional OS works?

The operating system (OS) is the first thing loaded onto the computer -- without the operating system, a computer is useless.

# 1.Booting

When you turn on the computer, the operating system program is loaded into the main memory.This process of starting the device is called **BOOTING**. The whole booting process is performed by the chip called **BIOS(Basic Input Output System) Chip** on the motherboard and this is the only chip in your computer who is responsible for the whole system startup process from when you push the power button to the operating system startup.

The **BIOS is a computer chip on the motherboard l**ike CMOS except that its purpose is to **communicate** between the [processor](https://www.lifewire.com/what-is-a-cpu-2618150) and other hardware components like the [hard drive](https://www.lifewire.com/what-is-a-hard-disk-drive-2618152), [USB](https://www.lifewire.com/universal-serial-bus-usb-2626039) ports, [sound card](https://www.lifewire.com/what-is-a-sound-card-2618160), [video card](https://www.lifewire.com/what-is-a-video-card-2618161), and more.**BIOS** software **stored** **permanently**(\*) on a **ROM** chip on the motherboard A computer without a BIOS wouldn't understand how these pieces of the computer work together.

The BIOS [firmware](https://www.lifewire.com/what-is-firmware-2625881) is also what performs the **Power On Self Test** to test those pieces of hardware, and what ultimately runs the boot loader to launch the [operating system](https://www.lifewire.com/operating-systems-2625912).

It typically involves a chain of events /stages, in which at each stage a smaller, simpler program loads and then executes the larger, more complicated program of the next stage.

*The BIOS typically looks to the CMOS chip to tell it where to find the OS, and in most PCs, the OS loads from the C drive on the hard or solid state drive, even though the BIOS has the capability to load the OS from a floppy disk, CD or other storage device. The order of drives that the CMOS looks to in order to locate the OS is called the* [***boot sequence***](https://www.webopedia.com/TERM/B/boot_sequence.html)*, which can be changed by altering the CMOS setup. Looking to the appropriate boot drive, the BIOS will first encounter the boot record, which tells it where to find the beginning of the OS and the subsequent program file that will initialize the OS.*

The bootstrap loader performs the Power-On Self Test(**POST**). When the POST is running, you will typically see lights flashing and hear a series of beeps. If any errors are found during the POST, it will be notified by beep or messages displayed on the screen.The error at this point is always a hardware problem.

Basically the computer is performing a test to make sure all the attached hardware is communicating clearly with the CPU.

Once the POST is complete, the bootloader program opens or wakes up the **BIOS**, or Basic Input/Output System, and it is activated. The Bios typically checks the CMOS RAM and tells the CMOS to find the OS .

**CMOS** is also a computer **chip** on the motherboard, or more specifically a **RAM chip**, which means it would normally lose the settings it's storing when the computer is shut down (just like how the contents of [RAM](https://www.lifewire.com/what-is-random-access-memory-ram-2618159) are not maintained each time you restart your computer). However, the CMOS battery is used to provide constant power to the chip.

When the computer first [boots](https://www.lifewire.com/what-does-booting-mean-2625799) up, BIOS pulls information from the CMOS chip to understand the hardware settings, time, and anything else that's stored in it. The chip typically stores as little as 256 [bytes](https://www.lifewire.com/the-difference-between-bits-and-bytes-816248) of information.

The BIOS is actually stored in read only memory (ROM). Once the BIOS has found a bootable device,(like hard disk, floppy disk etc) it ***loads the boot loader (*** residing in The EPROM, ROM ) called the ***Master Boot Record (MBR)****,*  asshown below.



The conventional MBR code checks the MBR's partition table for a partition set as bootable (the one with active flag set). If an active partition is found, the MBR code loads the boot sector code from that partition, known as ***Volume Boot Record (VBR),*** and executes it. ***The VBR is often operating system specific while the MBR code is not operating system specific.*** However, in most operating systems its ***main function is to load and execute the operating system kernel, which continues the startup process.***

In general terms, the MBR code is the **first stage boot loader**, while the VBR code is the **Second-stage boot loader**. Second-stage boot loaders,such as GNU GRUB, BOOTMGR, Syslinux, or NTLDR, are not themselves operating systems, but are able to load an operating system properly and transfer execution to it. This is why we say that these codes are operating system specific.

The operating system subsequently initializes itself and may load extra device drivers.

**Booting** is of **two types**… Warm Booting and cold booting.

**Cold booting** is starting the system upon power on, where it reads all the instructions from the ROM and the Operating System Automatically gets loaded into the System.

**Warm Booting** is that in which System Automatically Starts when the system is already Running .For eg., this could be due to power fluctuation

*The process of returning a computer from a* ***state of*** [***hibernation***](https://en.wikipedia.org/wiki/Hibernation_(computing)) ***or*** [***sleep***](https://en.wikipedia.org/wiki/Sleep_mode) ***does not involve booting.***

The purpose of an operating system is to organize and control hardware and software so that the device it lives in behaves in a flexible but predictable way.

At the simplest level, an operating system does two things:

1. It manages the hardware and software resources of the system. In a desktop computer, these resources include such things as the processor, memory, disk space and more (On a cell phone, they include the keypad, the screen, the address book, the phone dialer, the battery and the network connection).
2. It provides a stable, consistent way for applications to deal with the hardware without having to know all the details of the hardware.

The first task, managing the hardware and software resources, is ***very important***, as various programs and input methods ***compete for the attention of the central processing unit (CPU) and demand memory, storage and input/output (I/O) bandwidth f***or their own purposes. In this capacity, the operating system plays the ***role of the good parent,*** making sure that each application gets the necessary resources while playing nicely with all the other applications, as well as husbanding the limited capacity of the system to the greatest good of all the users and applications.

# 2.Setup the Memory

The **Master Boot Record** ( MBR ) is a special type of boot sector at the very beginning of partitioned computer mass storage devices like fixed disks. The MBR holds the information on *how the logical partitions*, containing file systems are organized on that medium. Besides that, the MBR functions as an operating system independent chain boot loader in conjunction with each partition's **Volume Boot Record** (VBR).

*MBRs are not present on non-partitioned media like floppies, pendrive or other storage devices*

**MBR** is a **512 byte sector** located in the **first sector of the hard disk**. It contains both program code and partition table details.When allocating disk space for a partition, the first sector or data unit for each partition is always reserved for programmable code used in booting process.

**First 446 byte** are the primary boot loader which contains both executable code and error message text.

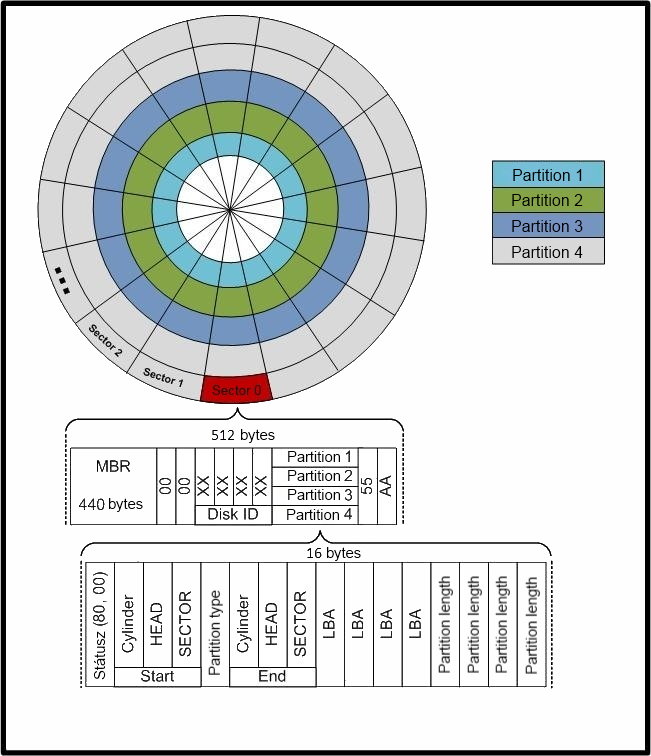
**Next 64** bytes contains the partition table. This section contains records for each of four partitions(P1,P2…).

**The last two bytes** known as magic number (0xAA55). This number is used for the validation check of MBR.

The bootstrap sequence in the BIOS will *load the first valid MBR t*hat it finds into the computer's physical memory at address 0000h : 7C00h . The last instruction executed in the BIOS code will be a "**jump**" to that address, to direct execution to the beginning of the MBR copy. The MBR is not located in a partition; it is located at a first sector of the device (physical offset 0), preceding the first partition.

Originating from the MBR,the total ***data storage*** space of a hard disk can be divided into *at most* ***four primary*** *partitions.* These partitions are described by ***16-byte*** entries that constitute the ***Partition Table***, located in the master boot record.

The ***partition type*** is identified by a ***1-byte code*** found in its partition table entry. Some of these codes may be used to indicate the presence of an extended partition.



* **Primary Partition**A partition that is needed to store and boot an operating system, though applications and user data can reside there as well, and what’s more, you can have a primary partition without any operating system on it. There can be up to a maximum of four primary partitions on a single hard disk, with only one of them set as active (“Active partition”).  
  Active (boot) partition is a primary partition that has an operating system installed on it. It is used for booting your machine. If you have a single primary partition, it is regarded as active. If you have more than one primary partition, only one of them is marked active.
* **Extended partition(Extended Boot Record EBR)**

It can be sub-divided into logical drives and is viewed as a container for logical drives, where **data is located**.The extended partition is used only for creating a desired number of logical partitions.

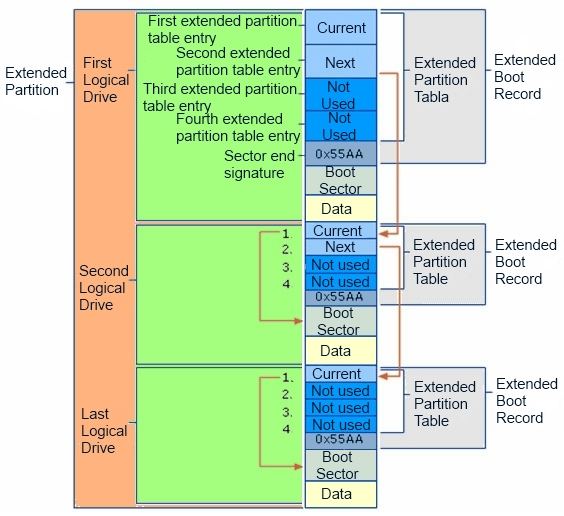
* **Logical drive** is created within an extended partition.

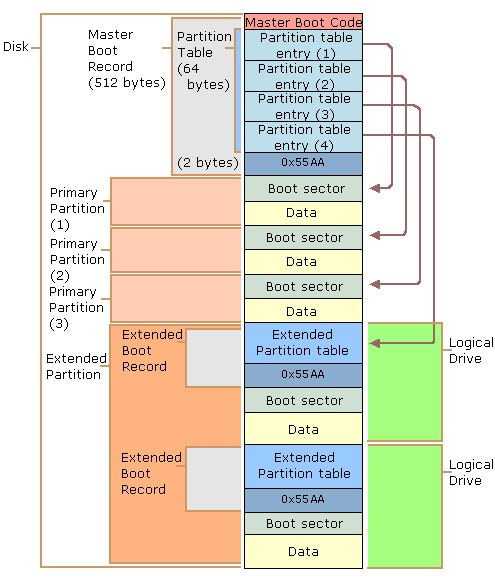
Logical partitions are used for **storing data mainly,** they can be formatted and assigned drive letters.

A logical partition is a way to extend the initial limitation of four partitions

Each EBR precedes the logical partition it describes. If another logical partition follows, then the first EBR will contain an entry pointing to the next EBR; thus, multiple EBRs form a linked list.

This means the number of logical drives that can be formed within an extended partition is **limited only by the amount of available disk space** in the given extended partition.





<https://gyires.inf.unideb.hu/GyBITT/20/ch02.html>

<https://gyires.inf.unideb.hu/GyBITT/20/ch02s02.html>

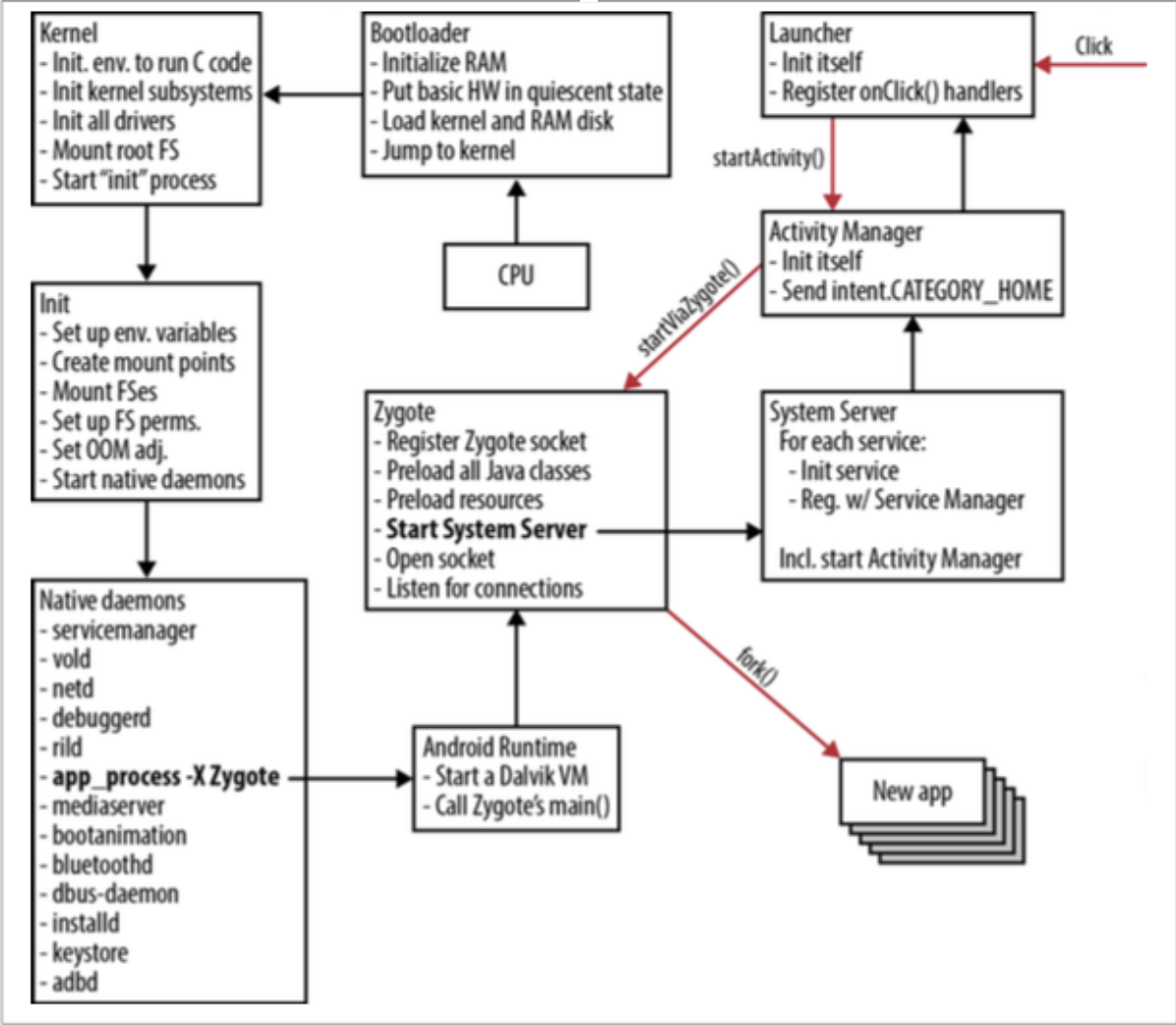
<http://www.c-jump.com/CIS24/Slides/Booting/Booting.html>

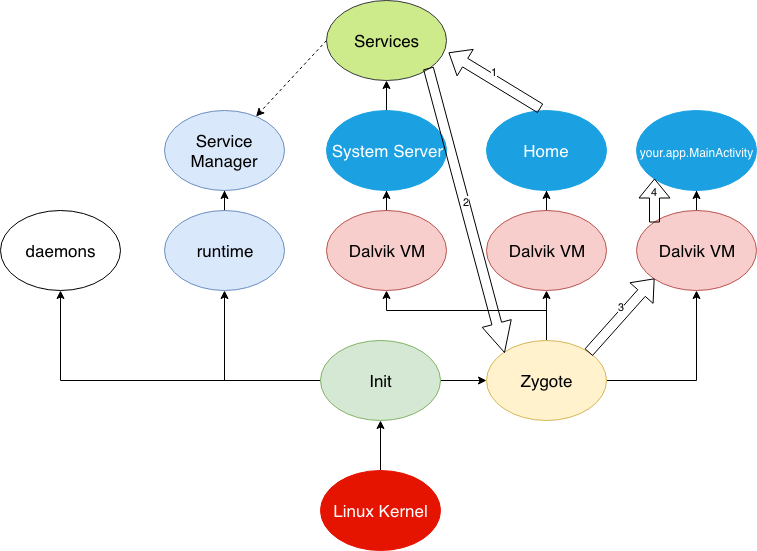
<https://www.geeksforgeeks.org/android-boot-process/>

How Android works?

## **How Does Android Boot Up?**

Here's the cycle of the bootup process in Android:

https://dzone.com/articles/android-internals-the-android-os-bootup-process



[*https://proandroiddev.com/how-android-boot-up-9864376d911c*](https://proandroiddev.com/how-android-boot-up-9864376d911c)

Android Internals

Android is an operating system based on a modified **Linux 2.6 with a Ja**[**va**](https://www.vogella.com/tutorials/JavaIntroduction/article.html) **programming interface.** Also several drivers and libraries have been modified to allow Android to run efficiently on mobile devices.

Android *doesn't use Java Virtual machine (JVM)* for executing class files; instead, it uses ***Dalvik virtual machine, which is not a true JVM and doesn't operate on Java bytecode.*** In order to run on Dalvik Virtual machines, class files are further compiled into Dalvik Executable or *DEX format*. After conversion to DEX format, class files along with other resources are bundled into Android Package (APK) for distribution and installation into various devices.

Android uses a special virtual machine, called the Dalvik Virtual Machine. Dalvik uses special **bytecode**. Therefore you **cannot** run standard Java bytecode on Android. Android provides a tool "**dx**" which allows to convert Java Class files into "dex" (**Dalvik Executable) files**. Android applications are packed into an .**apk** (Android Package) file by the program "aapt" (Android Asset Packaging Tool). To simplify development Google provides the Android Developer Tools (ADT) for Eclipse. The ADT performs automatically the conversion from class to dex files and creates the apk during deployment.

1. ***Boot ROM*** contains the initial code that’s run as soon as the device wakes up. It is a mask ROM or write protected flash drive. Embedded in the CPU chip, Boot ROM loads the Bootloader into RAM for execution.
2. ***Bootloader*** is a piece of code that runs before any operating system. It is responsible for loading an operating system from the device, setting up a minimal environment in which OS can run and beginning the startup process.The bootloader is used to do a ***low-level system initialization, before loading Linux Kernel***. Hence, ***it is not specific*** to **Android.**

One of the ***major tasks of bootloader*** involves setting up memory management,

security options. This is essential for the Kernel.

1. **Kernel**

Eventually, a **kernel is loaded into RAM** (normally this will be the kernel from the */boot* flash partition).

The Kernel does the bulk of hardware, driver and file system initialization.

* Core kernel initialization (Memory and I/O areas are initialized, Interrupts are started, and the process table is initialized)
* Driver initialization
* /root file system is mounted and kernel daemons are started
* Once it finishes system setup, it looks for init process in the system files and launch the first user-space processor root process.

At startup, Android loads the Linux Kernel and starts the **first process** called ***init***, which is similar to how any Linux system starts up.

**init program** is a key component of the Android bootup sequence, it initializes main elements of Android System.

**Bootloader** contains **two important files: init.s and main.c**

The Android **Init.s** is responsible for **initializing stacks**, and is a very crucial process. It is where the directories such as /dev, /sys, /proc are mounted.

The Android **Init.s** program process **two files**, executing the commands which it finds in them**.**

First one is generic init.rc, common for all Android Devices. The second one init.*<machine\_name>*.rc is device-specific initialization.These files have

commands that are responsible for starting processes called daemons. They sit on top of the hardware abstraction layer and listen on sockets for things like

* ***Android Debug Bridge Daemon (adbd)*** to manage ADB connections
* ***Debugger Daemon(debuggerd)*** to manage debug processes requests (dump memory, etc.)
* ***Radio Interface Layer Daemon (rild)*** to sit on top of the radio baseband interface with basement modem to manage communication with the radio.

It also start **ServiceManager** which will act as a registry for all system services.

**We can start these programs anytime on-demand.**

After starting these Linux daemons, one of the most important and neat process the init process does is “a call to **start the Zygote** “. Zygote process is basically a warmed-up ART/DalvikVM. *When it’s loaded it* ***goes to sleep***.

*Zygote is the first Android specific process when Android OS boots up!*

When new Application is required to start, the system sends an Intent to that process (with a message similar to “execute com.example.MainActivity”).

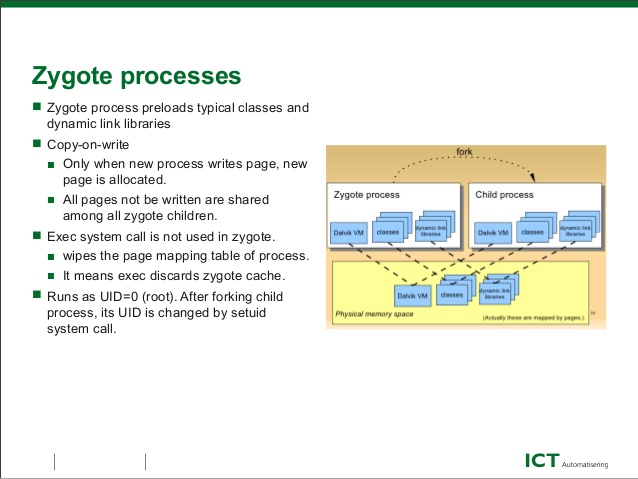
On receiving a new request,Zygote process ***wakes up***,*f****orks itself*** to ***create a new process*** which gets a pre-initialized VM instance and then goes ***back to sleep***.This **forking** is available due to ***copy-on-write*** resource management technique. *(Only when new process writes page, new page is allocated.All pages not be written are shared among all zygote children…. (info on slideshare.net)*

It doesn’t copy anything actually, just points to the pages of the parent process. The actual copying happens when there is a new write to the process’ pages.

The **forking** happens very efficiently. The child process, creates a new VM instance,loads the component that was requested(eg.com.example.MainActivity), process gets its own thread and resources to work with. This enables code sharing among VMs thus resulting in minimal startup time.

***Zygote preloads*** all the *system resources*,*core Java classes* and *dynamic link libraries* used by the Android framework and performs initial processing of them.

These classes can be reused by Android application.Every application ***runs its own process*** and we don’t want to do a cold start on the VM to start those up. *Zygote is the reason why Android apps start so fast.*



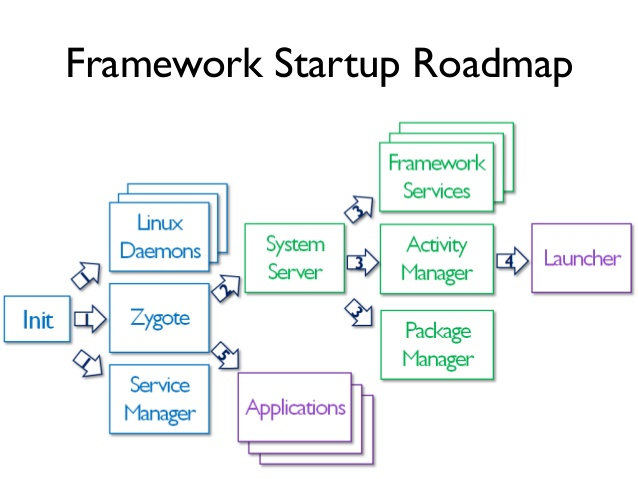
**Init starts Runtime process**

Runtime is an instance of class AppRuntime, which is sub-classed from AndroidRuntime(ART)

Runtime process does one very important thing i.e.,it starts the Service Manager and this is really the Context manager for Binder, that handles *service registration* and registers Service Manager as default context manager for Binder services.

Next, Zygote forks a new ART/Dalvik VM instance and starts the **System Server**.

1. **System Server**



**\*\*\***[***https://www.slideshare.net/ZongShenShen/toward-dynamic-analysis-of-obfuscated-android-malware***](https://www.slideshare.net/ZongShenShen/toward-dynamic-analysis-of-obfuscated-android-malware)

***\*\*\*\****[*https://www.slideshare.net/amraldo/android-booting-scenarios*](https://www.slideshare.net/amraldo/android-booting-scenarios)

***System Server is the first service started by the Zygote.***

The first thing system server does is ***starts the native library called android\_servers*** that provides interfaces to native functionalities.

It ***starts various Services and Managers*** like Power Manager, Activity Manager, Package Manager, Location Manager,Window Manager, Alarm Manager, Battery Service, Light Service, Bluetooth Service, Vibrator Service,Telephony registry and many more.

All of those services register themselves to the **ServiceManager**

It starts initializing each system service and registering them with the previously started Service Manager. Each service runs in a separate Dalvik thread in system server.

System Server also **starts other services beginning with com.android** such as com.android.phone, com.android.email.

*At this point, we are ready to launch the Home Application.*

1. **Activity Manager**

It is responsible ***for starting the Launcher App*** , activity thread process creation, activity stack management and activity lifecycle management.

**Activity Manager** will send a request to Zygote and say “I want to start an activity that subscribes to the Intent.ACTION\_MAIN and Intent.CATEGORY\_HOME. That will cause Zygote to fork off, and load this new process with the Home Activity and new ART/Dalvik VM.

Thus, it launches an intent to start Home Launcher and ***registers on click listeners*** to it. Whenever a click is detected, it launches new apps from icons on the home screen.

The click events are transferred to **AcitivityManagerService(AMS)** ***via Binder IPC.***

*And now your system is ready*.

When you press on your app icon on the home screen, Home application calls..

val launchIntent = packageManager.launchIntentForPackage("your.app.MainActivity")

startActivity(launchIntent)

which in cooperation of PackageManager and ActivityManager call Zygote with your package name ***your.app*** and an entry point which is ***your.app.MainActivity created.***

***The zygote then forks off with new ART/Dalvik VM and loads the activity component your.app.MainActivity.***

<https://proandroiddev.com/how-android-boot-up-9864376d911c>

1. **Binder**

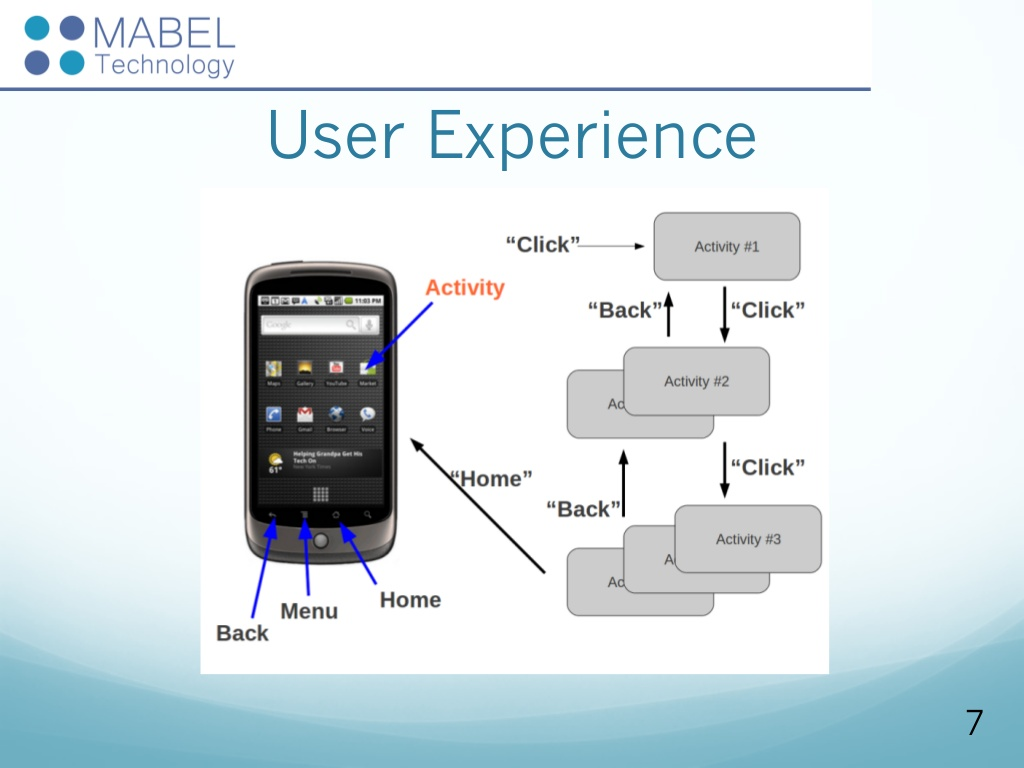
A special driver called ***Binder*** allow an efficient ***interprocess communications (IPC)*** in which **references to objects are passed between processes**. The real objects are stored in Shared Memory. This way the communication between the processes is optimized as less data must be transferred.

Android **does not provide a swap space**(A swap file (*or swap space or, in Windows NT, a pagefile) is a space on a hard disk used as the virtual memory extension of a computer's real memory (RAM). Having a swap file allows your computer's operating system to pretend that you have more RAM than you actually do*.) like other Linux systems therefore the amount of available memory is limited to the **memory** on the device.

Androids uses **a special C library "Bionic" instead of the standard Glibc**. This library is not compatible with Glibs but **requires less memory.** Bionic contains a special thread implementation which optimizes the memory consumption of each thread and **reduces the start time of new threads.**

<https://www.vogella.com/tutorials/AndroidInternals/article.html>

<https://proandroiddev.com/android-internals-101-how-android-os-starts-you-application-e1c98a014c05>



If you take a basic digital cell phone apart, you find that it contains just a few individual parts:

* A circuit board containing the brains of the phone
* An antenna
* A [liquid crystal display](https://electronics.howstuffworks.com/lcd.htm) (LCD)
* A keyboard (touch/button )
* A [microphone](https://electronics.howstuffworks.com/gadgets/audio-music/question309.htm)
* A [speaker](https://electronics.howstuffworks.com/speaker.htm)
* A [battery](https://electronics.howstuffworks.com/everyday-tech/battery.htm) etc



Google usually refers to the Android OS as a **software stack**. Each layer of the stack groups together several programs that support specific operating system functions.

The base of the stack is the **kernel**. Google used theLinux version 2.6 OS to build Android's kernel, which includes Android's memory management programs, security settings, power management software and several hardware drivers. **Drivers** are programs that control hardware devices.For Cell phones with camera, the Android kernel includes a camera driver, which allows the user to send commands to the camera hardware.

The next level of software includes Android's **libraries**, which are a set of instructions that tell the device how to handle different kinds of data. For example, the media framework library supports playback and recording of various audio, video and picture formats. Other libraries include a three-dimensional acceleration library (for devices with accelerometers) and a Web browser library.**The Android runtime(ART)** layer includes a set of core **Java** libraries -- Android application programmers build their apps using the Java programming language. It also includes the Dalvik Virtual Machine.

**There are 3 ways how an app talks to the native libraries…**

1.**Directly** ( Application-->thru Binder IPC to Runtime service-->thru JNI to Native service binding-->Dynamic Loading HAL Library→ Device Driver)

Eg...**Thru Location Manager…**

( Application-->thru Binder IPC to Location Manager service-->GPS Location Provider-->thru JNI GPS Location Provider Binder-->Dynamic Loading libgps.so--> GPS Device Driver)

2.**Through a service** ( Application-->thru Binder IPC to Runtime service-->thru JNI Native Service Binding-->thru Binder IPC to Native Service-->Dynamic Loading HAL Library→ Device Driver)

**Eg: Media Player** ( Application-->thru Binder IPC to Media Player→ thru JNI to Media Player Binder→thru Binder IPC to Audio Flinger→Dynamic Loading to libaudio.so→ LASA→ Speaker device driver)

3.**Through a daemon** ( Application-->thru Binder IPC to Runtime service-->thru JNI to Native service binding→ thru Sockets to Native Daemon-->Dynamic Loading to HAL Library→ Device Driver)

Depending upon the

1. Application
2. Native library implementation
3. Android version

The Android OS uses virtual machines to run each application as its own **process**. That's important for a few reasons.

First, no application is dependent upon another.

Second, if an application crashes, it shouldn't affect any other applications running on the device. Third, it simplifies memory management.

<https://electronics.howstuffworks.com/google-phone2.htm>

<https://javarevisited.blogspot.com/2013/06/introduction-of-how-android-works-Java-programmers.html>

<https://tektab.com/2015/10/31/android-bootloaderfastboot-mode-and-recovery-mode-explained/>

Development in Android

There is always a misconception that Android is a programming language.. But it is NOT.

Then how do you think programming is done in Android???

The ***Android Software Development Kit (Android SDK)*** and the ***Gradle*** tooling contains the necessary tools to Compile, package, deploy and start Android application.

Google provides a *specialized IDE* called *Android Studio t*o perform development tasks.

The Android SDK contains the ***Android debug bridge (adb)***. adb is a tool that allows you to connect to a virtual or real Android device. This allows managing the device or debugging your application.

# SDK or devkit

We make use of The software Development kit( SDK or devkit) which is a set of development tools provided by hardware and software providers which allows the developer to create or build applications, softwares, or frameworks and also to ensure that the software operates correctly and smoothly on a particular platform.

**It provides the infrastructure of all the libraries which are precoded.**

So nothing new can be created.. But using the features in precoded modules we can modify or extend the functionalities as per the requirement.

***While there are many different programming languages and a host of IDEs (Integrated Development Environments) you can use to create an app, the SDK is a constant.***

Android SDK Tools is a component for the Android SDK. It includes the complete set of development tools necessary to build, test, and debug apps for Android like,

* Required libraries
* Debugger
* An emulator
* Relevant documentation for the Android application program interfaces (APIs)
* Sample source code
* Tutorials for the Android OS

It is included with Android Studio.

Every time Google releases a new version of Android, a corresponding SDK is also released. To be able to write programs with the latest features, developers must download and install each version’s SDK for the particular phone.

Development platforms compatible with SDK

1. OS like Windows(XP or Later)
2. Linux
3. Mac OSX(10.4.9/ later)

SDK is used to write Android programs by using an integrated development environment (IDE), which serves as the central programming interface. ***Android Studio is the official Android IDE*** declared by Google. IDE’s provide graphical interface enabling the developers to perform the development tasks faster.

Android apps can be written using **Kotlin, Java, and C++ languages** using the Android software development kit (SDK).

**Java code** is most popularly used for Android apps and hence JDK(Java Development kit) must be installed.

Eg: Adding some more list items, enabling/ disabling/ cancelling certain buttons etc.

## Components of SDK:

The Android SDK can be broken down into several components. These include:

1. Platform tools
2. Build Tools
3. SDK tools
4. The Android Debug Bridge(ADB)
5. Android Emulator

The most important parts of this package are in the **SDKtools**. You will need these tools regardless of which version of Android you are targeting. These are what will actually create the APK – turning your Java program into an Android app that can be launched on a phone. These include a number of build tools, debugging tools, and image tools.

### **Platform tools**

The Platform tools are more specifically suited to the version of Android that you want to target. Generally, it is best to install the latest Platform tools, which will be installed by default. After first installation though, you need to keep your Platform-tools constantly updated. The tools should be backwards compatible, meaning that you will still be able to support older versions of Android.

### **Build tools**

The Build tools were once categorized under the same heading as the Platform tools but have since been decoupled so that they can be updated separately. As the name suggests, these are also needed to build your Android apps. This includes the zipalign tool for instance, which optimizes the app to use minimal memory when running prior to generating the final APK, and the apksigner which signs the APK (surprise!) for subsequent verification.

### **Android Debug Bridge**

The Android Debug Bridge (ADB) is a versatile command-line tool /program that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It relies on Platform-tools in order to understand the Android version that is being used on said device and hence it is included in the Platform-tools package. You can use ADB to access shell tools such as logcat, to query your device ID or even to install apps.It is a client-server program that includes three components:

* **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
* **A daemon (adbd)**, which runs commands on a device. The daemon runs as a background process on each device.
* **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

**How an ADB works** is, when you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators. Where the server finds an adb daemon (adbd), it sets up a connection to that port. Note that each emulator uses a pair of sequential ports — an *even-numbered port* for ***console*** connections and an odd-numbered port for adb connections.

For example:

Emulator 1, console: 5554

Emulator 1, adb: 5555

Emulator 2, console: 5556

Emulator 2, adb: 5557

and so on…

As shown, the emulator connected to adb on port 5555 is the same as the emulator whose console listens on port 5554.

Once the server has set up connections to all devices, you can use adb commands to access those devices. Because the server manages connections to devices and handles commands from multiple adb clients, you can control any device from any client (or from a script).

### 

### **Android Emulator**

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

The emulator provides almost all of the capabilities of a real Android device. You can

simulate incoming phone calls and text messages,

specify the location of the device,

simulate different network speeds,

simulate rotation and other hardware sensors,

access the Google Play Store,

and much more.

Testing your app on the emulator is in some ways faster and easier than doing so on a physical device. For example, you can transfer data faster to the emulator than to a device connected over USB.The emulator comes with predefined configurations for various Android phone, tablet, Wear OS, and Android TV devices.

You can use the emulator manually through its graphical user interface and programmatically through the command line and the emulator console.

#### **Android virtual devices:**

Each **instance** of the **Android Emulator** uses an *Android virtual device (AVD)* to specify the **Android version** and **hardware characteristics of the simulated device**.

An **Android** Virtual Device (AVD) represents a specific**Android** device. You can use an **Android emulator** as a target platform to run and test your **Android** applications on your PC.

To effectively test your app, you should create an AVD that models each device on which your app is designed to run. To create and manage AVDs, use the AVD Manager.

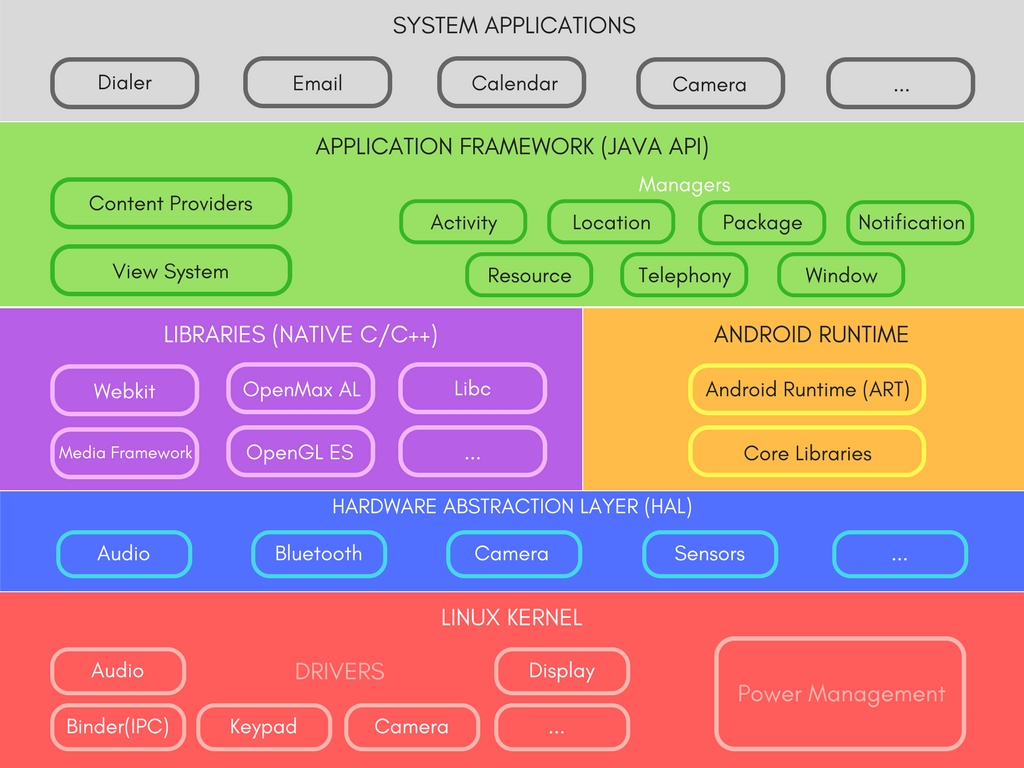
Each **AVD** functions as an independent device, with its **own private storage** for user data, SD card, and so on. By default, the emulator stores the user data, SD card data, and cache in a directory specific to that AVD. When you launch the emulator, it loads the user data and SD card data from the AVD directory.The Android Virtual Device manager( AVD Manager) is used in order to choose which version of Android you want to emulate, along with the device specifications (screen size, performance etc.).

<https://www.androidauthority.com/android-sdk-tutorial-beginners-634376/>

Installation Link:

<https://github.com/codepath/android_guides/wiki/Installing-Android-SDK-Tools>

Architecture



The Android operating system follows a layered architecture approach. All these layers are responsible for different roles and features.

# Linux Kernel

This layer is the ***foundation of the Android Platform.***

* Contains all low level drivers for various hardware components support.
* Android Runtime relies on Linux Kernel for core system services like,
  + Memory, process management, threading etc.
  + Network stack
  + Driver model
  + Security and more.

# Hardware Abstraction Layer (HAL)

* Provides Abstraction between hardware and rest of the software stack.

# Android Runtime (ART) (Instead of Dalvik)

* Designed to run apps in a constrained environment that has limited muscle power in terms of battery, processing and memory.
* Since Android 5.0, each app runs in its own process within its own instance of ART virtual machine, which makes process management more crucial.
* ART uses DEX files, which is a type of bytecode, specially designed for Android, which helps ART to manage memory more efficiently.
* Contains a set of core libraries that enables developers to write Android Apps using Java Programming.
* Prior to Android 5.0, Dalvik was used as Android runtime.
* ART is capable of both **Ahead-of-time** (AOT) and **Just-in-time** (JIT) compilation.
* It also has a very ***efficient garbage collection.***

# Libraries

* Exposed to developers through Android Application Framework.
* Contains C/C++ libraries used by components of Android Systems.
* Few features include,
  + SQLite Library used for data storage and light in terms of mobile memory footprints and task execution.
  + WebKit Library mainly provides Web Browsing engine and a lot more related features.
  + The surface manager library is responsible for rendering windows and drawing surfaces of various apps on the screen.
  + The media framework library provides media codecs for audio and video.
  + The OpenGl (Open Graphics Library) and SGL(Scalable Graphics Library) are the graphics libraries for 3D and 2D rendering, respectively.
  + The FreeType Library is used for rendering fonts.

# Application Framework

* It is a collection of APIs written in Java, which gives developers access to the complete feature set of Android OS.
* Developers have full access to the same framework APIs used by the core applications, so that they can enhance more in terms of functionalities of their application.
* Enables and simplify the reuse of core components and services, like:
  + **Activity Manager**: Manages the Lifecycle of apps & provide common navigation back stack.
  + **Window Manager**: Manages windows and drawing surfaces, and is an abstraction of the surface manager library.
  + **Content Providers**: Enables application to access data from other applications or to share their own data i.e it provides mechanism to exchange data among apps.
  + **View System**: Contains User Interface building blocks used to build an application's UI, including lists, grids, texts, boxes, buttons,etc. and also performs the event management of UI elements
  + **Package Manager**: Manages various kinds of information related to the application packages that are currently installed on the device.
  + **Telephony Manager**: Enables app to use phone capabilities of the device.
  + **Resource Manager**: Provides access to non-code resources (localized Strings, bitmaps, Graphics and Layouts).
  + **Location Manager**: Deals with location awareness capabilities.
  + **Notification Manager**: Enable apps to display custom alerts in the status bar.

# System Applications

* Top of the Android Application Stack, is occupied by the System apps and tonnes of other Apps that users can download from Android's Official Play Store, also known as Google Play Store.
* A set of Core applications are pre-packed in the handset like Email Client, SMS Program, Calendar, Maps, Browser, Contacts and few more.
* This layer uses all the layers below it for proper functioning of these mobile apps.

So as we can see and understand, ***Android holds layered or we can say grouped functionalities as software stack that makes Android work very fluently in any device.***

Android Devices in Market

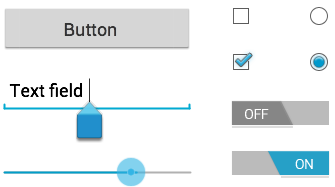
A device that runs Android OS comes in all shapes and sizes. Various devices that run Android OS and Apps are as follows,

* Smartphones
* Smart-watches
* Tablets
* E-reader Devices
* Netbooks
* MP4 Players
* Internet TVs and more.

https://www.studytonight.com/android/android-architecture

Android-UI Controls

Input controls are the interactive components in your app's user interface. Android provides a wide variety of controls you can use in your UI, such as buttons, text fields, seek bars, check box, zoom buttons, toggle buttons, etc



##### UI Elements

A view is just a widget that appears on the screen.

# **View**

A **View** is an object that draws something on the screen that the user can interact with and

*Views are used to create input and output fields in the an Android App. It can be input text field, radio field, image field etc. They are the same as, input text field, image tag to show images, radio field in HTML.*

There are a number of UI controls provided by Android that allow you to build the graphical user interface for your app.

|  |  |
| --- | --- |
| **Sr.No.** | **UI Control & Description** |
| 1 | TextView  This control is used to display text to the user. |
| 2 | EditText  EditText is a predefined subclass of TextView that includes rich editing capabilities. |
| 3 | AutoCompleteTextView  The AutoCompleteTextView is a view that is similar to EditText, except that it shows a list of completion suggestions automatically while the user is typing. |
| 4 | Button  A push-button that can be pressed, or clicked, by the user to perform an action. |
| 5 | ImageButton  An ImageButton is an AbsoluteLayout which enables you to specify the exact location of its children. This shows a button with an image (instead of text) that can be pressed or clicked by the user. |
| 6 | CheckBox  An on/off switch that can be toggled by the user. You should use check box when presenting users with a group of selectable options that are not mutually exclusive. |
| 7 | ToggleButton  An on/off button with a light indicator. |
| 8 | RadioButton  The RadioButton has two states: either checked or unchecked. |
| 9 | RadioGroup  A RadioGroup is used to group together one or more RadioButtons. |
| 10 | ProgressBar  The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background. |
| 11 | Spinner(Dropdown)  A drop-down list that allows users to select one value from a set. |
| 12 | TimePicker  The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode. |
| 13 | DatePicker  The DatePicker view enables users to select a date of the day. |
| 14 | ListView  It is a view which groups several items and display them in vertical scrollable list. The list items are automatically inserted to the list using an **Adapter** that pulls content from a source such as an array or database. |
| 15 | GridView  **It** shows items in two-dimensional scrolling grid (rows & columns) and the grid items are automatically inserted to the layout using a **ListAdapter** |
| 16 | Spinner(DropDown)  Spinners provide a quick way to select one value from a set. In the default state, a spinner shows its currently selected value. Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one. |
| 17 | Scroll View  It allows the view hierarchy placed within it to be scrolled. Scroll view supports vertical scrolling only. For horizontal scrolling, use HorizontalScrollView instead. |
| 118 | Recyler View  Recyler Viewis an advanced and flexible version of ListView and GridView. It is a container used for displaying large amount of data sets that can be scrolled very efficiently . *It provides an ability to implement the horizontal, vertical and Expandable List*. It is mainly used when we have data collections whose elements can change at run time based on user action or any network events. For using this widget we have to specify the Adapter and Layout Manager. |

# ViewGroup

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

**A ViewGroup** is an object that holds other View (and ViewGroup) objects in order to define the layout of the user interface.

One or more views can be grouped together into one **GroupView**. Example of ViewGroup includes **layouts**.

## **Attributes of Views/Viewgroups**

Some attributes that are common in all views and ViewGroups are listed below −

|  |  |
| --- | --- |
| **Sr.No** | **View & description** |
| 1 | **layout\_width**  Specifies the width of the View or ViewGroup |
| 2 | **layout\_height**  Specifies the height of the View or ViewGroup |
| 3 | **layout\_marginTop**  Specifies extra space on the top side of the View or ViewGroup |
| 4 | **layout\_marginBottom**  Specifies extra space on the bottom side of the View or ViewGroup |
| 5 | **layout\_marginLeft**  Specifies extra space on the left side of the View or ViewGroup |
| 6 | **layout\_marginRight**  Specifies extra space on the right side of the View or ViewGroup |
| 7 | **layout\_gravity**  Specifies how child Views are positioned |
| 8 | **layout\_weight**  Specifies how much of the extra space in the layout should be allocated to the View |

## Units of Measurement

When you are specifying the size of an element on an Android UI, you should remember the following units of measurement.

|  |  |
| --- | --- |
| **Sr.No** | **Unit & description** |
| 1 | **dp**  Density-independent pixel. 1 dp is equivalent to one pixel on a 160 dpi screen. |
| 2 | **sp**  Scale-independent pixel. This is similar to dp and is recommended for specifying font sizes |
| 3 | **pt**  Point. A point is defined to be 1/72 of an inch, based on the physical screen size. |
| 4 | **px**  Pixel. Corresponds to actual pixels on the screen |

## Screen Densities

|  |  |
| --- | --- |
| **Sr.No** | **Density & DPI** |
| 1 | **Low density (ldpi)** 120 dpi |
| 2 | **Medium density (mdpi)** 160 dpi |
| 3 | **High density (hdpi)** 240 dpi |
| 4 | **Extra High density (xhdpi)** 320 dpi |

# Android Layout

In Android the term layout refers to defining how the View components are displayed on the screen relative to each other. A layout is typically defined partly by the View and partly by the ViewGroup which contains the View. A typical layout defines the visual structure for an Android user interface.

## Types of layout

There are many types of layout. Some of which are listed below −

* Linear Layout
* Absolute Layout
* Table Layout
* Frame Layout
* Relative Layout
* Constraint Layout
* Grid Layout
* Scroll View

### **Linear Layout**

LinearLayout puts all its child elements into a single column or row depending on the android:orientation attribute.Possible values for this attribute are ***horizontal*** and ***vertical***. ***Horizontal is the default value.***LinearLayout can be nested to achieve more complex layouts.

### **AbsoluteLayout**

The AbsoluteLayout enables you to specify the exact location of its children.

### **TableLayout**

The TableLayout groups views into rows and columns.

### **4.** **RelativeLayout**

RelativeLayout allows positioning the widget relative to each other. This can be used for complex layouts. RelativeLayout is a complex layout manager and should only be used if such a complex layout is required, as it performs a resource intensive calculation to layout its children.The RelativeLayout enables you to specify how child views are positioned relative to each other.

### **5**. **FrameLayout**

The FrameLayout is a placeholder on screen that you can use to display a single view. FrameLayout is a layout manager which draws all child elements on top of each other. This allows to create nice visual effects.FrameLayout are used to display several buttons on top of another layout.

### **6. Constraint Layout**

ConstraintLayout is provided by an external library. It allows you to use a flat view hierarchy and has great performance. Also the design tools support constraint layout very well. New projects should prefer the usage of constraint layout. ConstraintLayout allows you to define constraints for views.

### **7. Grid Layout**

GridLayout allows you to organize views into a grid / table. GridLayout separates its drawing area into: rows, columns, and cells.

You can specify how many columns the grid should have. For each view you can specify in which row and column it should be placed and how many columns and rows it should use. If not specified, GridLayout uses defaults, e.g., one column, one row

### **8. Scroll View**

The ScrollView or the HorizontalScrollView class is useful to make views available, even if they do not fit onto the screen. A scroll view can contain one view, e.g., a layout manager containing more views. If the child view is too large, scroll view allows scrolling the content.

|  |
| --- |
|  |

Optimizing layouts

Here are some of the guidelines for creating efficient layouts.

* Avoid unnecessary nesting
* Avoid using too many Views
* Avoid deep nesting

# 

Components of Android Application:

The four essential components(building blocks) of an Android application are..

1.Activities

2.Services

3.Broadcast Receivers

4. Content Providers.

These are loosely coupled to the applications Manifest file “AndroidManifest.xml” which describes each component of the app and how they interact.

## Activity

The *first primary component of Android*  is, ***Activity***.

When we all started with coding, we know about the *main method* from where the program *begins execution.* Similarly, in Android, **Activity** is the one from where the Android Application ***starts its process***.

**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(user Interface).

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.

Activities are often presented to the user as full-screen windows, they can also be used in other ways: as floating windows or embedded into other windows.

**Concept:**

The mobile-app experience differs from its desktop counterpart in that a ***user's interaction with the app doesn't always begin in the same place****.* Instead, the ***users journey often begins non-deterministically.*** For instance, if you open an email app from your home screen, you might see a list of emails. By contrast, if you are using a social media app , then that launches your email app, you might go directly to the email app's screen for composing an email.

The Activity class is designed to facilitate this paradigm. ***When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. In this way, the activity serves as the entry point for an app's interaction with the user.*** You implement an activity as a subclass of the Activity class.

An activity provides the window in which the app draws its UI. This window typically fills the screen, but may be smaller than the screen and float on top of other windows. Generally, one activity implements one screen in an app. Most apps contain multiple screens, which means they comprise multiple activities.

Typically, ***one activity in an app is specified as the main activity***, which is the first screen to appear when the user launches the app. Each activity can then start another activity in order to perform different actions. For example, the main activity in a simple e-mail app may provide the screen that shows an e-mail inbox. From there, the main activity might launch other activities that provide screens for tasks like writing e-mails and opening individual e-mails.

Although activities work together to form a cohesive user experience in an app,***each activity is only loosely bound to the other activities; there are usually minimal dependencies among the activities in an app.*** In fact, activities often start up activities belonging to other apps. For example, a browser app might launch the Share activity of a social-media app.

Every activity you define for your application must be **declared and registered**  in your ***AndroidManifest.xml*** file and the main activity for your app must be declared in the manifest **with an <intent-filter>** that includes the MAIN action and LAUNCHER category .***If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps***.

### **Lifecycle of an Activity**

The cycle consists of 7 methods:

1. **onCreate()** — called when an activity first created; *when a user opens the app.*This is where you should do all of your *normal static set up*: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one.**Always followed by onStart().**
2. **onStart()** — called when an activity becomes visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
3. **onResume()** — called when a user starts interacting with the application.At this point your **activity is at the top of its activity stack**, with user input going to it. **Always followed by onPause().**
4. **onPause()** — Called when the activity loses foreground state, is no longer focusable or before transition to stopped/hidden or destroyed state. The activity is still visible to the user , so it is recommended to keep it visually active and continue updating the UI.**Implementations of this method must be very quick** because the next activity will not be resumed until this method returns. ***The paused activity does not receive user input and cannot execute any code.*** Followed by onResume() if the activity returns back to the front, or onStop()if it becomes invisible to the user.
5. **onStop()** — called when activity is ***no longer visible to the user***; ***when a user minimizes the app*.** This may happen either because a new activity is being started on top, an existing one is being brought in front of this one, or this one is being destroyed. This is ***typically used to stop animations and refreshing the UI,*** etc. Followed by either onRestart() if this activity is coming back to interact with the user, or onDestroy() if this activity is going away.
6. **onDestroy()** —The final call you receive before your activity is destroyed. This can happen either because the activity is finishing or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the Activity#isFinishing method.This callback is called before the activity is destroyed by the system, *when the user clears the application stack.*
7. onRestart() — This callback is called when the activity restarts after stopping it.

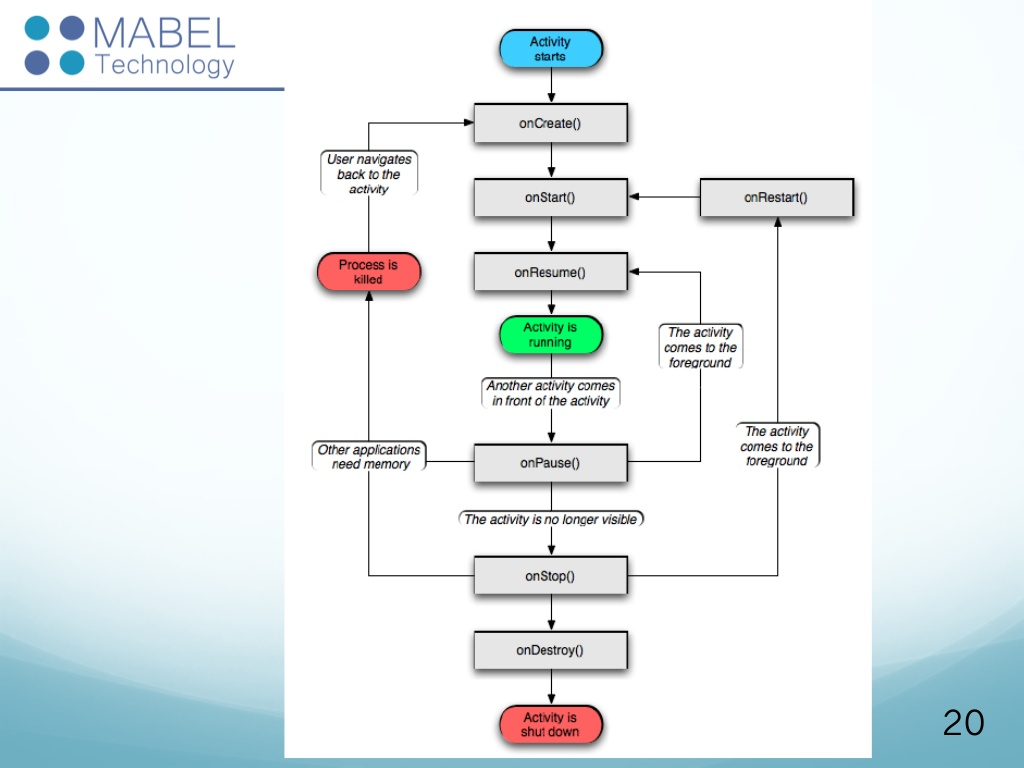
**So, here we can understand the single screen as a single activity.**

Activities in the system are managed as activity stacks. When a new activity is started, it is usually placed on the top of the current stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits. There can be one or multiple activity stacks visible on screen.

An activity has essentially **four states:**

* If an activity is in the foreground of the screen (at the highest position of the topmost stack), it is ***active* or *running*.** This is usually the activity that the user is currently interacting with.
* If an activity has lost focus but is still presented to the user, it is ***visible*.** It is possible if a new non-full-sized or transparent activity has focus on top of your activity, another activity has higher position in multi-window mode, or the activity itself is not focusable in current windowing mode. Such activity is completely alive (it maintains all state and member information and remains attached to the window manager).
* If an activity is completely obscured(covered or concealed) by another activity, it is ***stopped* or *hidden*.** It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will ***often be killed by the system*** when memory is needed elsewhere.
* The system can drop the activity from memory by either asking it to finish, or simply killing its process, making it ***destroyed***. When it is again displayed to the user, it must be completely restarted and restored to its previous state.

The following diagram shows the important state paths of an Activity. The ***square*** ***rectangles*** represent ***callback methods*** you can implement to perform operations when the Activity moves between states. The ***colored ovals are major states*** the Activity can be in.



There are **three key loops** you may be interested in monitoring **within your activity:**

* The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread in onDestroy().
* The **visible lifetime** of an activity happens between a call to [onStart()](https://developer.android.com/reference/android/app/Activity.html" \l "onStart()) until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden to the user.
* The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause(). During this time the activity is in visible, active and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

The entire lifecycle of an activity is defined by the following Activity methods. All of these are hooks that you can override to do appropriate work when the activity changes state. All activities will implement [onCreate(Bundle)](https://developer.android.com/reference/android/app/Activity.html" \l "onCreate(android.os.Bundle)) to do their initial setup; many will also implement onPause() to commit changes to data and prepare to pause interacting with the user, and onStop() to handle no longer being visible on screen. You should always call up to your superclass when implementing these methods.

<https://developer.android.com/guide/components/activities/intro-activities>

<https://developer.android.com/reference/android/app/Activity>

## 

## 2.**Services**:

* Service is an ***Android Component without an UI*** which runs on the main thread (of the hosting process) and hence it is ***not bound to the lifecycle of an activity.***
* By default, a service runs in the same process as the main thread of the application.
* A service in Android is designed to have a longer life.
* It is used to perform repetitive and long running operations in background without direct interaction with the user. i.e., Internet downloads, checking for new data, data processing, updating content providers and the like. ***Services run indefinitely unless they are explicitly stopped or destroyed.***
* It also has to be ***declared*** in the application’s manifest file ***AndroidManifest.xml***.
* It can be started by any other application component. Components can even infact bind to a service to interact with it and even perform Interprocess- Communication(IPC).For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.
* Services ***run with a higher priority than inactive or invisible activities*** and therefore it is less likely that the Android system terminates them.
* It can ***still be running even if the application is killed*** unless it stops itself by calling *stopself*() or is stopped by an Android component by calling *stopService*().
* If not stopped it goes on running unless is terminated by Android due to resource shortage. 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.
* The android.app.Service is subclass of ContextWrapper class.
* Services which run in the process of the application are sometimes called ***local services.***
* Custom services are started from other Android components, i.e., activities, broadcast receivers and other services.

**Note**: Service always runs on the main thread by default. When the doc says “long running processes in background” it means that processes without an UI. So if you are performing a time consuming task you must be creating a background thread in the service.

There are 2 main kinds of services —

1. Started Services
2. Bound Services

Other types are..

Bound and Started Services (both at the same time)

Foreground services

Background services

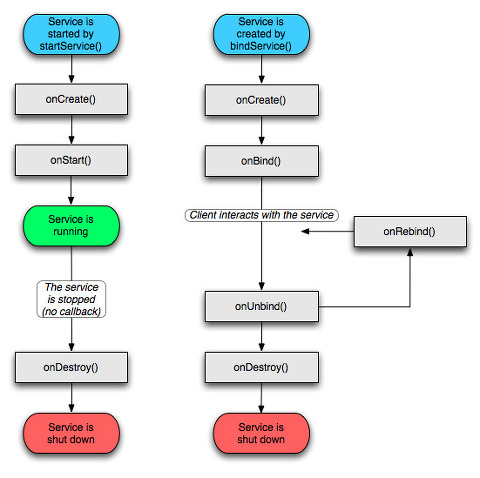
Service can either be started or bound. You just need to call either ***startService*()** where the service can run in the background indefinitely, even if the component that started it is destroyed; or **bindService()** ( from any of your android components), where if a component starts the service by calling **startService()** (which results in a call to **onStartCommand()**), the service continues to run until it stops itself with **stopSelf()** or another component stops it by calling **stopService().**

If a component calls **bindService()** to create the service and **onStartCommand()** is *not* called, the service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC), and ***the service runs only as long as the component is bound to it.*** After the service is unbound from all of its clients, the system destroys it.

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. It performs some operation that is ***noticeable*** to the user. For example, an audio app would use a foreground service to play an audio track.The foreground service always uses the notification to notify the user and using the notification you can actually interact with the service or the ongoing operation such as pause the music or play the next music.The ***notification cannot be dismissed*** unless the service is either stopped or removed from the foreground. So whenever in your app you see a notification that is performing some long running tasks that service is basically the foreground service and Foreground Service is always noticeable to the user that is, the user is aware of this ongoing process. Foreground services continue running even when the user isn't interacting with the app.

A ***background service*** performs an operation that ***isn't*** directly ***noticed*** by the user. 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 with the processing.

For example, if an app used a service to compact its storage, that would usually be a background service.



***There is always only a single instance of service running*** in the app. If you are calling startService() for a single service multiple times in your application it just invokes the onStartCommand() on that service. ***Neither is the service restarted multiple times nor are its multiple instances created.***

### **Comparison of services:**

|  |  |  |
| --- | --- | --- |
| **Unbound(Start)**  **Service** | **Bound**  **Service** | **Intent**  **Service** |
| Unbounded Service is used to perform long repetitive task | Bounded Service is used to perform background task in bound with another component | Intent Service is used to perform one time task i.e when the task completes the service destroys itself. |
| Unbound Service gets starts by calling startService(). | Bounded Service gets starts by calling bindService(). | Intent Service gets starts by calling startService(). |
| Unbound Service is stopped or destroyed explicitly by calling stopService(). | Bounded Service is unbind or destroyed by calling unbindService(). | IntentService Implicitly calls stopself() to destroy |
| Unbound Service is independent of the component in which it is started. | Bound Service dependents on the component in which it is started. | Intent Service is independent of the component in which it is started. |

The **callback methods** and description:

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

## 3. Broadcast Receivers(Receivers):

Android Broadcast Receiver is a dormant component of android that listens to system-wide broadcast events or intents.

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

When any of these events occur it brings the application into action by either creating a status bar notification or performing a task.

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.

Receiver doesn’t contain any user interface.It is generally implemented to delegate the tasks to services depending on the type of intent data that’s received.

Its **main job** is to **pass the notification** to the user, whenever a specific event occurs.

For instance, a Receiver triggers “Battery Low notification “, “new friend notification”, “new message notification” , “Wifi activated/deactivated” etc which you see on your mobile.

A BroadcastReceiver object is only valid for the duration of the call to **onReceive(Context, Intent)**. Once your code returns from this function, the system considers the object to be **finished** and **no longer active.**

Apps can **receive broadcasts in two ways**:

1.Through manifest-declared receivers

2. Through context-registered receivers.

### **1.Manifest-declared receivers**

If you declare a broadcast receiver in your manifest, the system launches your app (if the app is not already running) when the broadcast is sent.

The system package manager registers the receiver when the app is installed. The receiver then becomes a separate entry point into your app which means that the system can start the app and deliver the broadcast if the app is not currently running.

The system creates a new **BroadcastReceiver** component **object** to handle each broadcast that it receives. This object is valid only for the duration of the call to **onReceive(Context, Intent**. Once your code returns from this method, the system considers the component no longer active.

### **2.Context-registered receivers**

Context-registered receivers receive broadcasts ***as long as their registering context is valid*.** For an example, if you register within an **Activity** context, you receive broadcasts as long as the activity is not destroyed. If you register with the **Application context**, you receive broadcasts as long as the app is running.

**Registration** can be done in **two** ways--

1.**Static**( use<receiver> tag in your manifest file (AndroidManifest.xml))

2.**Dynamic**. (Use Context.registerReceiver() method to dynamically register the instance.)

If you register a receiver in **onCreate(Bundle)** using the activity's context, you should unregister it in **onDestroy()** to prevent leaking the receiver out of the activity context.

If you register a receiver in **onResume()**, you should unregister it in **onPause()** to prevent registering it multiple times (If you don't want to receive broadcasts when paused, and this can cut down on unnecessary system overhead).

Do not unregister in **onSaveInstanceState(Bundle),** because this isn't called if the user moves back in the history stack.

***Do not start long running background threads from a broadcast receiver.*** After **onReceive(),** the system can kill the process at any time to reclaim memory, and in doing so, it terminates the spawned thread running in the process.

To avoid this, you should either call **goAsync()** (if you want a little more time to process the broadcast in a background thread) or schedule a **JobService** from the receiver using the **JobScheduler**, so the system knows that the process continues to perform active work.

<https://androidclarified.com/android-service-lifecycle-and-working/>

<https://vogella.com/tutorials/AndroidServices/article.html>

<https://proandroiddev.com/deep-dive-into-android-services-4830b8c9a09>

[https://o7planning.org/en/10421/android-services-tutorial#a1172084](https://o7planning.org/en/10421/android-services-tutorial" \l "a1172084)

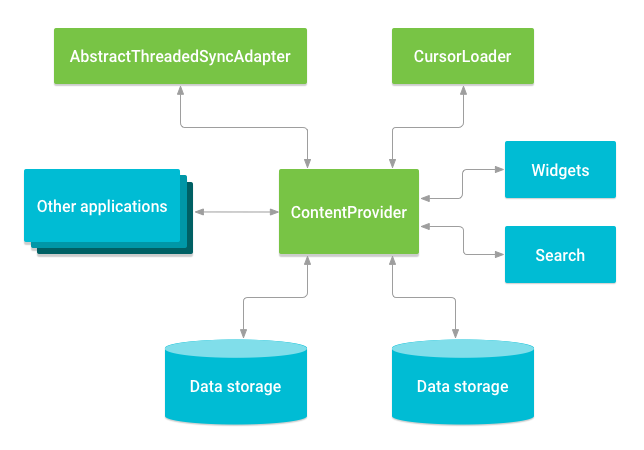
## 4. Content Provider(Provider):

* The **Content Provider** is one of the components of an android application.
* In order for a content provider to be **visible** within an Android system, it must be **declared** within the **Android manifest file** for the application in which it resides.
* It ***acts as a central repository*** to store the applications data in one place and make that data available for different applications to access whenever it’s required.
* It will act as more like relational database to store the app data.
* Content providers support the **four basic operations**, normally called **CRUD-operations**. CRUD is the acronym for **C**reate(insert),**R**ead(retrieve,query) **U**pdate and **D**elete. These multiple operations can be performed on the data stored in content provider
* 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.
* In android, we can use content provider whenever we want to ***share our app data*** with other apps and it allows us to make modifications to our application data without affecting other applications which depends on our app.
* By using content providers we can manage data such as audio, video, images and personal contact information.
* The content provider is having different ways to store app data. The app data can be stored in SQLite database or in files or even over a network based on our requirements. This essentially involves the implementation of a ***client/server*** ***arrangement*** whereby the application seeking access to the data is the client and the content provider is the server, performing actions and returning results on behalf of the client.
* We have a different type of access permissions available in content provider to share the data. We can set a restrict access permissions in content provider to restrict data access limited to only our application and we can configure different permissions to read or write a data. Permissions can be set to cover the entire content provider, or limited to specific tables and records.

*To summarize, without content providers accessing data of other apps would be a mess.*

A content provider coordinates access to the data storage layer in your application for a number of different APIs and components

* Sharing access to your application data with other applications
* Sending data to a widget
* Returning custom search suggestions for your application through the search framework using **SearchRecentSuggestionsProvider**
* Synchronizing application data with your server using an implementation of **AbstractThreadedSyncAdapter**
* Loading data in your UI using a **CursorLoader**



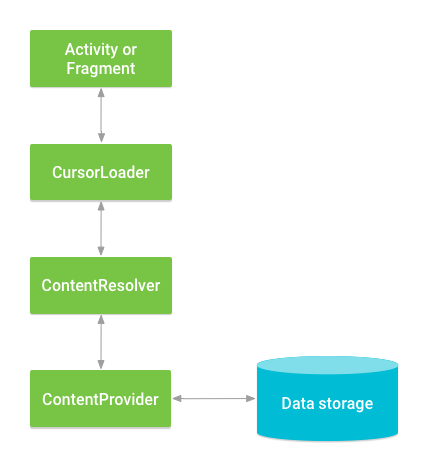
Relationship between content provider and other components.

### **Accessing a provider**

Access to a content provider is achieved via a ContentResolverobject**.** An application can obtain a reference to its content resolver by making a call to the getContentResolver() method of the application context.

The content resolver object contains a set of methods that mirror those of the content provider (insert, query, delete etc). The application simply makes calls to the methods, specifying the URI of the content on which the operation is to be performed. The content resolver and content provider objects then communicate to perform the requested task on behalf of the application and returns the results.

A common pattern for accessing aContentProvider from your UI uses a CursorLoader to run an asynchronous query in the background. The Activity orFragment in your UI call a CursorLoader to the query, which in turn gets the ContentProvider using the ContentResolver. This allows the UI to continue to be available to the user while the query is running. This pattern involves the interaction of a number of different objects, as well as the underlying storage mechanism.

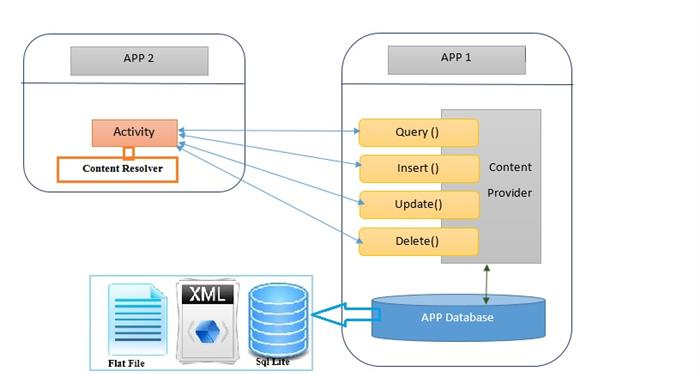


Interaction between ContentProvider, other classes, and storage.

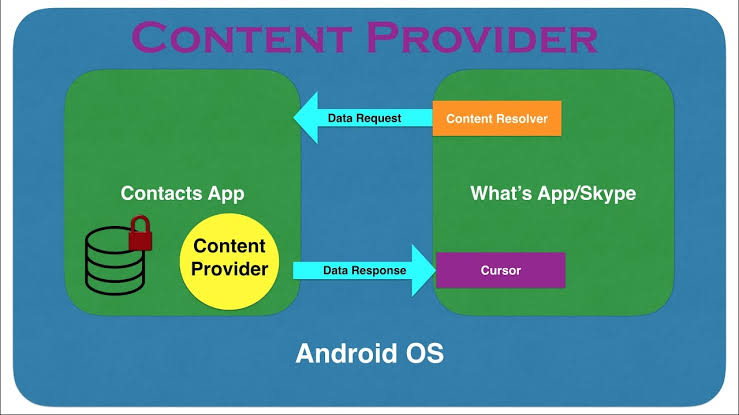
An Android device will potentially contain a number of content providers. The system must, therefore, provide some way of identifying one provider from another. Similarly, a single content provider may provide access to multiple forms of content (typically in the form of database tables). Client applications, therefore, need a way to specify the underlying data for which access is required. This is achieved through the use of **content URIs(**Uniform Resource Identifier).

The content URI is essentially used to identify specific data within a specific content provider. The Authority section of the URI identifies the content provider and usually takes the form of the package name of the content provider.

When implementing the Create, Read, Update and Delete methods in the content provider, it will be the responsibility of these methods to identify whether the incoming URI is targeting a specific row in a table, or references multiple rows and act accordingly.



<https://www.codeproject.com/Articles/818578/An-Absolute-Beginner-s-Guide-to-Building-and-Acces>



[https://guides.codepath.com/android/creating-content-providers#contract-classes](https://guides.codepath.com/android/creating-content-providers" \l "contract-classes)

<https://developer.android.com/guide/topics/providers/content-provider-basics>

<https://www.tutlane.com/tutorial/android/android-content-providers-with-examples>

<https://www.techotopia.com/index.php/Understanding_Android_Content_Providers>

# Fragments:

* Fragment is an Android programming aspect that represents a portion of the user interface of what a user sees on the application window.Fragment is a portion of user interface that can contain views, events and logic in an Activity Object.
* A **Fragment** is a combination of an **XML** layout file and a **java** class much like an Activity.
* A ***fragment*** is an independent Android component which can be used by an activity.
* Any activity using fragments should make sure to **extend** from *FragmentActivity* or *AppCompatActivity.*
* Fragments were added in Honeycomb version of Android i.e API version 11. Prior to this We could have only a single Activity on the screen at a given point of time, and there was no way to divide the screen and control the different parts separately.
* Fragments encapsulate views and logic so that it is easier to reuse within activities. You can build single-pane layouts for handsets (phones) and multi-pane UI/ layouts for tablets. You can also use fragments to support different layouts for landscape and portrait orientation on a smartphone.
* Fragments may only be used as **part of an activity** and ***cannot be instantiated*** as ***standalone*** application elements. So a fragment can be thought of as a

functional “**sub-activity**” with its own lifecycle similar to that of a full activity.It functions independently, but as it is linked to the Activity, when an activity is destroyed, the fragment also gets destroyed.

* Using the support library, fragments are supported back to all relevant Android versions.
* If you know Biology, and are aware of the concept of Host and Parasite, then in Android, ***Activity is the host*** while a ***Fragment is a parasite***.
* Fragment has its ***own lifecycle events***, which are different from an Activity's life cylce events.
* An Activity can have ***any number of fragments***in it, although it is suggested not to use too many fragments in a single activity.
* Also, a ***fragment is a reusable component***, hence, a single fragment can be included in multiple activities, if required.
* There are two ways to add a fragment to an activity: **dynamically** using **Java** and **statically** using **XML**. To add the fragment **statically**, simply embed the fragment in the activity's xml layout file. The second way is by adding the fragment **dynamically** in Java using the ***FragmentManager***. The *FragmentManager class* and the *FragmentTransaction class* allow you to add, remove and replace fragments in the layout of your activity at runtime.
* First the activity obtains a reference to the fragment. Then it gets a reference to the ViewGroup the fragment’s view will be rendered inside. Then the activity adds the fragment. The fragment then creates its view and returns it to the activity. The view is then inserted into the ViewGroup parent, and the fragment is alive.
* As it is possible to **dynamically add and remove fragments** from an activity, the usage of fragments allows to design very **flexible user interfaces**.
* A Fragment has it's own Layout for the UI(user interface), but we can even define a fragment without any layout, to implement behaviours which has no user interface, more like a background service.
* 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.

### **Main use of Fragments in Android**

Following are the 3 main usage of Fragments in Android, for which Fragments were introduced:

1. **Modularity**: If a single activity is having too many functional components, it is better to divide it into independent fragments, hence making the code more organized and easier to maintain.
2. **Reusability**: Fragments enable re-use of parts of your screen including views and event logic over and over in different ways across many activities. For example, using the same list across different data sources within an app.If we define any particular feature in a fragment, then that feature more or less becomes a reusable component which can be easily integrated into any activity.
3. **Adaptability/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. If we break UI components of an app screen into fragments, then it becomes easier to change their orientation and placement, based on screen size etc.-

### **Fragment Lifecycle**

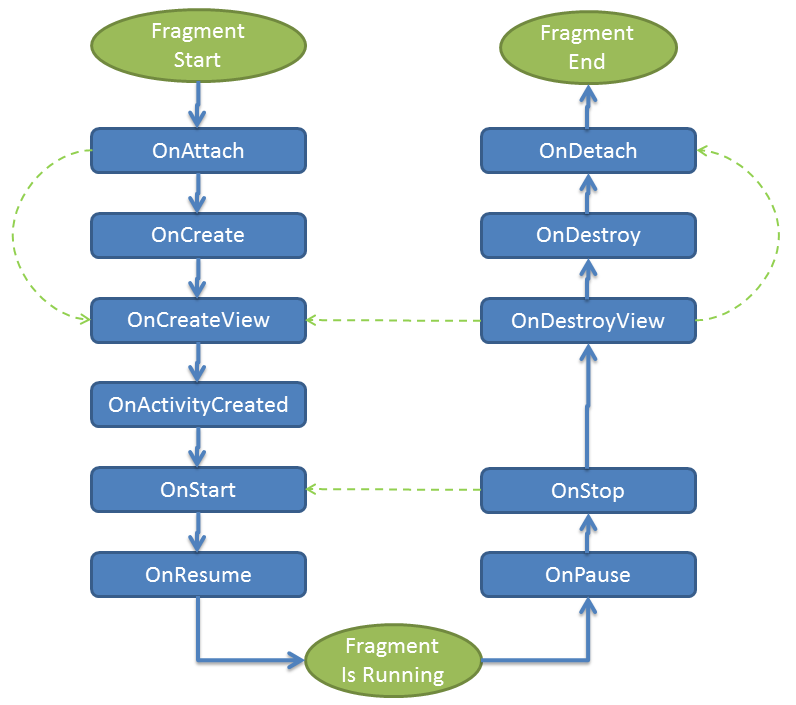
Fragment has many methods which can be overridden to plug into the lifecycle (similar to an Activity):

* **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 when it’s time for the fragment to draw its user interface for the first time.To draw a UI for your fragment, you must return a View component from this method that is the root of your fragment’s layout. You can return null if the fragment does not provide a UI.
* **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 the screen(visible).
* **onResume()** -This method called when a fragment is visible and allowing the user to interact with it. Allocates “expensive” resources such as registering for location, sensor updates, etc.***Fragment resumes only after activity resumes.***

When fragment goes out off the screen..

* **onPause()** - Release “expensive” resources. Commit any changes.
* **onStop()**-This method called when the fragment is no longer visible.
* **onDestroyView()** is called when fragment's view and related resources are being removed from the activity’s view hierarchy and destroyed, but the fragment is still kept around.
* **onDestroy()** is called when a fragment is no longer in use and when the fragment does its final clean up
* **onDetach()** is called when a fragment is no longer connected to or detached from the host activity.

The lifecycle execution order is mapped out below:



The most common ones to override are ***onCreateView*** which is in almost every fragment to setup the inflated view, ***onCreate*** for any data initialization and ***onActivityCreated*** used for setting up things that can only take place once the Activity has been fully created.

## **Handling device orientation-**

Android runs on a variety of devices that have different screen sizes and pixel densities. The system performs basic scaling and resizing to adapt your user interface to different screens, but there is more work you should do to ensure your UI gracefully adapts for each type of screen.

**Screen Sizes**

The screen size is the visible space provided for your app UI. The screen size as it's known to your app is not the actual size of the device screen—it takes into account the screen orientation, system decorations (such as the navigation bar), and window configuration changes (such as when the user enables *multi-window mode*).

**Flexible Layouts**

By default, Android resizes your app layout to fit the current screen. To ensure your layout resizes well for even small variations in screen size, you need to implement your layout with flexibility in mind. The **core principle** you must follow is to **avoid hard-coding** the position and size of your UI components. Instead, allow view sizes to stretch and specify view positions relative to the parent view or other sibling views so your intended order and relative sizes remain the same as the layout grows.

**Alternative layouts**

Although your layout should always respond to different screen sizes by stretching the space within and around its views, that might not provide the best user experience for every screen size. For example, the UI you designed for a phone, probably doesn't offer a good experience on a tablet. Therefore, your app should also provide **alternative layout resources** to optimize the UI design for certain screen sizes.

You can provide **screen-specific layouts** by creating additional res/layout/ directories—one for each screen configuration that requires a different layout—and then append a screen configuration qualifier to the layout directory name (such as layout-w600dp for screens that have 600dp of available width).

These **configuration qualifiers** represent the visible screen space available for your app UI. The system takes into account any system decorations (such as the navigation bar) and window configuration changes (such as when the user enables *multi-window mode*) when selecting the layout from your app.

**1.Use the smallest width(sw)** **qualifier**

The **"smallest width**"screen size qualifier allows you to provide alternative layouts for screens that have a minimum width measured in **density-independent pixels(dp or dip)**.

The **pixel density** is the number of pixels within a physical area of the screen and is referred to as **dpi (dots per inch)**. This is different from the resolution, which is the total number of pixels on a screen.

***Maintaining density independence is important*** because, without it, a UI element (such as a button) might appear larger on a low-density screen and smaller on a high-density screen (because when the pixels are larger—as shown in figure 2—a few pixels can go a long way).

The **Android** system helps you ***achieve density independence*** by providing density-independent pixels (dp or dip) as a unit of measurement that you should use instead of pixels (px).

In some cases, you will need to express dimensions in dp and then convert them to pixels. The **conversion** of dp units to screen pixels is simple:

***px = dp \* (dpi / 160)***

By describing the screen size as a measure of density-independent pixels, Android allows you to create layouts that are designed for very specific screen dimensions while avoiding any concerns you might have about different pixel densities..

For example

res/layout/main\_activity.xml # For handsets (smaller than 600dp available width)

res/layout-**sw600dp**/main\_activity.xml # For 7” tablets (600dp wide and bigger)

The smallest width qualifier specifies the smallest of the screen's two sides, **regardless of the device's current orientation**, so it's a simple way to specify the overall screen size available for your layout.

Remember that all the figures for the smallest width qualifier are ***density-independent pixels,*** because what matters is the amount of screen space available after the system accounts for pixel density (**not the raw pixel resolution**).

**2.Use the available width (w) and available height(h) qualifier**

Instead of changing the layout based on the smallest width of the screen, you might want to change your layout based on how much width or height is currently available. For example, if you have a two-pane layout, you might want to use that whenever the screen provides at least 600dp of width, which might change depending on whether the device is in landscape or portrait orientation. In this case, you should use the "available width" qualifier as follows:

res/layout/main\_activity.xml # For handsets (smaller than 600dp available width)

res/layout-w600dp/main\_activity.xml # For 7” tablets or any screen with 600dp

# available width (possibly landscape handsets)

If the **available height** is a concern for you, then you can do the same using the "available height" qualifier. For example, **layout-h600dp** for screens with at least 600dp of screen height.

**3.Add orientation qualifiers**

Although you may be able to support all size variations using only combinations of the "smallest width" and "available width" qualifiers, you might also want to change the user experience when the user switches between portrait and landscape orientations.

For that you can add the port or land qualifiers to your resource directory names. Just be sure these come after the other size qualifiers. For example:

res/layout/main\_activity.xml # For handsets

res/layout-land/main\_activity.xml # For handsets in landscape

res/layout-sw600dp/main\_activity.xml # For 7” tablets

res/layout-sw600dp-land/main\_activity.xml # For 7” tablets in landscape

If your app **supports Android 3.1 (API level 12)** or lower, you need to use the ***legacy size qualifiers*** in addition to the smallest/available width qualifiers from above.

From the example above, if you want a two pane layout on larger devices you need to use the "large" configuration qualifier to support version 3.1 and lower. So, to implement these layouts on those older versions, you might have the following files:

res/layout/main\_activity.xml # For **handsets** (smaller than 640dp x 480dp)

res/layout-large/main\_activity.xml # For **small** tablets (640dp x 480dp and bigger)

res/layout-xlarge/main\_activity.xml # For **large** tablets (960dp x 720dp and bigger)

**4.Modularize UI components with fragments**

When designing your app for multiple screen sizes you want to make sure you aren't needlessly duplicating your UI behavior across your activities. So you should use ***fragments*** to extract your UI logic into separate components. Then, you can then combine fragments to create multi-pane layouts when running on a large screen or place in separate activities when running on a handset.

For example, a news app on a tablet might show a list of articles on the left side and a full article on the right side—selecting an article on the left updates the article view on the right. On a handset, however, these two components should appear on separate screens—selecting an article from a list changes the entire screen to show that article.

**Provide alternative bitmaps**

To provide good graphical qualities on devices with different pixel densities, you should provide multiple versions of each bitmap in your app—one for each density bucket, at a corresponding resolution.

There are several density buckets available for use in your apps. Table 1 describes the different configuration qualifiers available and what screen types they apply to.

|  |  |
| --- | --- |
| Density Qualifier | Description |
| ldpi | Resources for low-density (*ldpi*) screens (~120dpi). |
| mdpi | Resources for medium-density (*mdpi*) screens (~160dpi). (This is the baseline density.) |
| hdpi | Resources for high-density (*hdpi*) screens (~240dpi). |
| xhdpi | Resources for extra-high-density (*xhdpi*) screens (~320dpi). |
| xxhdpi | Resources for extra-extra-high-density (*xxhdpi*) screens (~480dpi). |
| xxxhdpi | Resources for extra-extra-extra-high-density (*xxxhdpi*) uses (~640dpi). |
| nodpi | Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density. |
| tvdpi | Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33\*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi. |

To create alternative bitmap drawables for different densities, you should follow the **3:4:6:8:12:16 scaling ratio** between the six primary densities. For example, if you have a bitmap drawable that's 48x48 pixels for medium-density screens, all the different sizes should be:

* 36x36 (0.75x) for low-density (ldpi)
* 48x48 (1.0x baseline) for medium-density (mdpi)
* 72x72 (1.5x) for high-density (hdpi)
* 96x96 (2.0x) for extra-high-density (xhdpi)
* 144x144 (3.0x) for extra-extra-high-density (xxhdpi)
* 192x192 (4.0x) for extra-extra-extra-high-density (xxxhdpi)

Then, place the generated image files in the appropriate subdirectory under res/ and the system will pick the correct one automatically based on the pixel density of the device your app is running on:

res/

drawable-xxxhdpi/

awesome-image.png

drawable-xxhdpi/

awesome-image.png

drawable-xhdpi/

awesome-image.png

drawable-hdpi/

awesome-image.png

drawable-mdpi/

awesome-image.png

Then, any time you reference @drawable/awesomeimage, the system selects the appropriate bitmap based on the screen's dpi. If you don't provide a density-specific resource for that density, the system picks the next best match and scales it to fit the screen.

**Put app icons in mipmap directories**

Like all other bitmap assets, you need to provide density-specific versions of you app icon. However, ***some app launchers display your app icon as much as 25% larger than what's called for by the device's density bucket.***

For example, if a device's density bucket is xxhdpi and the largest app icon you provide is in drawable-xxhdpi, the launcher app scales up this icon, and that makes it appear less crisp. So you should provide an even higher density launcher icon in the mipmap-xxxhdpi directory. Now the launcher can use the xxxhdpi asset instead.

Because your app icon might be scaled up like this, you should put all your app icons in mipmap directories instead of drawable directories. Unlike the drawable directory, 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.

<https://developer.android.com/training/multiscreen/screensizes>

## 

<https://www.studytonight.com/android/fragments-in-android>

[https://www.vogella.com/tutorials/AndroidFragments/article.html#what-are-fragments](https://www.vogella.com/tutorials/AndroidFragments/article.html" \l "what-are-fragments)