



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF EEE

DEPARTMENT OF ECE

**UNIT – I - INTRODUCTION TO VARIOUS MOBILE PLATFORMS –
SECA5205**

MOBILE APPLICATION DEVELOPMENT

UNIT 1 INTRODUCTION TO VARIOUS MOBILE PLATFORMS

Introduction: Survey of prominent mobile platforms - Android - iOS - Windows Mobile.

What is Mobile Application Development?

Mobile application development is the process of creating software applications that run on a mobile device, and a typical mobile application utilizes a network connection to work with remote computing resources. Hence, the mobile development process involves creating installable software bundles (code, binaries, assets, etc.) , implementing backend services such as data access with an API, and testing the application on target devices

Mobile Applications and Device Platforms

There are two dominant platforms in the modern smartphone market. One is the iOS platform from Apple Inc. The iOS platform is the operating system that powers Apple's popular line of iPhone smartphones. The second is Android from Google. The Android operating system is used not only by Google devices but also by many other OEMs to built their own smartphones and other smart devices.

Although there are some similarities between these two platforms when building applications, developing for iOS vs. developing for Android involves using different software development kits (SDKs) and different development toolchain. While Apple uses iOS exclusively for its own devices, Google makes Android available to other companies provided they meet specific requirements such as including certain Google applications on the devices they ship. Developers can build apps for hundreds of millions of devices by targeting both of these platforms.

Alternatives for Building Mobile Apps

There are four major development approaches when building mobile applications

- Native Mobile Applications
- Cross-Platform Native Mobile Applications
- Hybrid Mobile Applications
- Progressive Web Applications

Each of these approaches for developing mobile applications has its own set of advantages and disadvantages. When choosing the right development approach for their projects, developers consider the desired user experience, the computing resources and native features required by the app, the development budget, time targets, and resources available to maintain the app.

ANDROID PLATFORM:-

Android is an open source and Linux-based operating system for mobile devices such as smartphones and tablet computers. Android was developed by the Open Handset Alliance, led by Google, and other companies.

What is Android?

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

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

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

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

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

Why Android ?



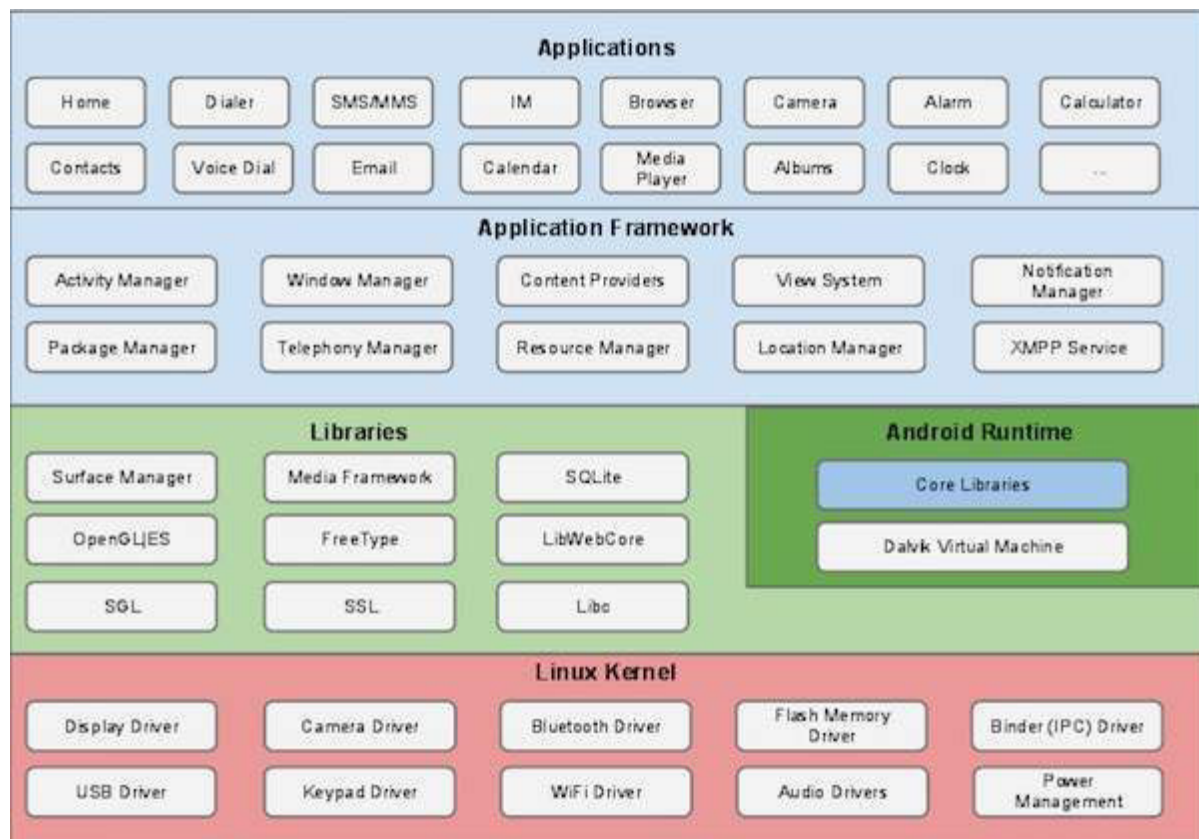
Android IDEs

There are so many sophisticated Technologies are available to develop android applications, the familiar technologies, which are predominantly using tools as follows

- Android Studio
- Eclipse IDE(Deprecated)

Android - Architecture

Android operating system is a stack of software components which is roughly divided into five sections and four main layers as shown below in the architecture diagram.



Linux kernel

At the bottom of the layers is Linux - Linux 3.6 with approximately 115 patches. This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc. Also, the kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, which take the pain out of interfacing to peripheral hardware.

Libraries

On top of Linux kernel there is a set of libraries including open-source Web browser engine WebKit, well known library libc, SQLite database which is a useful repository for storage and sharing of application data, libraries to play and record audio and video, SSL libraries responsible for Internet security etc.

Android Libraries

This category encompasses those Java-based libraries that are specific to Android development. Examples of libraries in this category include the application framework libraries in addition to those that facilitate user

interface building, graphics drawing and database access. A summary of some key core Android libraries available to the Android developer is as follows –

- **android.app** – Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** – Facilitates content access, publishing and messaging between applications and application components.
- **android.database** – Used to access data published by content providers and includes SQLite database management classes.
- **android.opengl** – A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** – Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.text** – Used to render and manipulate text on a device display.
- **android.view** – The fundamental building blocks of application user interfaces.
- **android.widget** – A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** – A set of classes intended to allow web-browsing capabilities to be built into applications.

Having covered the Java-based core libraries in the Android runtime, it is now time to turn our attention to the C/C++ based libraries contained in this layer of the Android software stack.

Android Runtime

This is the third section of the architecture and available on the second layer from the bottom. This section provides a key component called **Dalvik Virtual Machine** which is a kind of Java Virtual Machine specially designed and optimized for Android.

The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language.

The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Application Framework

The Application Framework layer provides many higher-level services to applications in the form of Java classes. Application developers are allowed to make use of these services in their applications.

The Android framework includes the following key services –

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

Applications

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

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

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

Sr.No	Components & Description
1	Activities They dictate the UI and handle the user interaction to the smart phone screen.
2	Services They handle background processing associated with an application.
3	Broadcast Receivers They handle communication between Android OS and applications.
4	Content Providers They handle data and database management issues.

Activities

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

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

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


Services

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

A service is implemented as a subclass of **Service** class as follows –

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

Broadcast Receivers

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

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

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

Content Providers

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

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

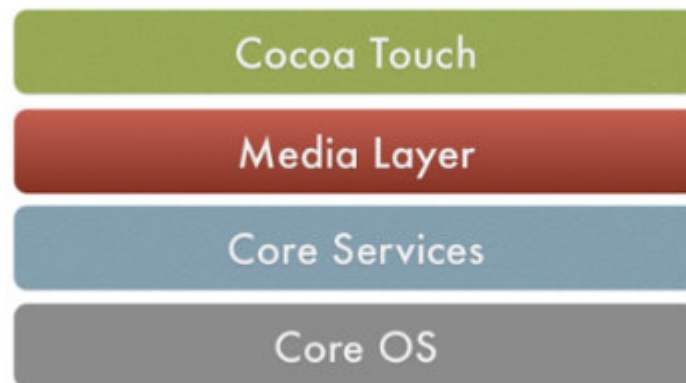
```
public class MyContentProvider extends ContentProvider  
{  
    public void onCreate() {}  
}
```

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

Apple iOS Architecture

Architecture of IOS is a layered architecture. At the uppermost level iOS works as an intermediary between the underlying hardware and the apps you make. Apps do not communicate to the underlying hardware directly. Apps talk with the hardware through a collection of well defined system interfaces. These interfaces make it simple to write apps that work constantly on devices having various hardware abilities. Lower layers gives the basic services which all application relies on and higher level layer gives sophisticated graphics and interface related services.

Apple provides most of its system interfaces in special packages called frameworks. A framework is a directory that holds a dynamic shared library that is .a files, related resources like as header files, images, and helper apps required to support that library. Every layer have a set of Framework which the developer use to construct the applications.



<https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/> 2/5

1. Core OS Layer:

The Core OS layer holds the low level features that most other technologies are built upon.

- Core Bluetooth Framework.
- Accelerate Framework.
- External Accessory Framework.
- Security Services framework.
- Local Authentication framework.

64-Bit support from IOS7 supports the 64 bit app development and enables the application to run faster.

2. Core Services Layer

Some of the Important Frameworks available in the core services layers are detailed:

- Address book framework – Gives programmatic access to a contacts database of user.
- Cloud Kit framework – Gives a medium for moving data between your app and iCloud.
- Core data Framework – Technology for managing the data model of a Model View Controller app.
- Core Foundation framework – Interfaces that gives fundamental data management and service features for ios apps.
- Core Location framework – Gives location and heading information to apps.
- Core Motion Framework – Access all motion based data available on a device. Using this core motion framework Accelerometer based information can be accessed.
- Foundation Framework – Objective C covering too many of the features found in the Core Foundation framework
- Healthkit framework – New framework for handling health-related information of user
- Homekit framework – New framework for talking with and controlling connected devices in a user's home.
- Social framework – Simple interface for accessing the user's social media accounts.
- StoreKit framework – Gives support for the buying of content and services from inside your iOS apps, a feature known as In-App Purchase.

3. Media Layer: Graphics, Audio and Video technology is enabled using the Media Layer.

Graphics Framework:

- UIKit Graphics – It describes high level support for designing images and also used for animating the content of your views.
- Core Graphics framework – It is the native drawing engine for iOS apps and gives support for custom 2D vector and image based rendering.
- Core Animation – It is an initial technology that optimizes the animation experience of your apps.
- Core Images – gives advanced support for controlling video and motionless images in a nondestructive way
- OpenGL ES and GLKit – manages advanced 2D and 3D rendering by hardware accelerated interfaces
- Metal – It permits very high performance for your sophisticated graphics rendering and computation works. It offers very low overhead access to the A7 GPU.

Audio Framework:

- Media Player Framework – It is a high level framework which gives simple use to a user's iTunes library and support for playing playlists.
- AV Foundation – It is an Objective C interface for handling the recording and playback of audio and video.
- OpenAL – is an industry standard technology for providing audio.
- Video Framework
- AV Kit – framework gives a collection of easy to use interfaces for presenting video.
- AV Foundation – gives advanced video playback and recording capability.
- Core Media – framework describes the low level interfaces and data types for operating media.

Cocoa Touch Layer

- UIKit framework – gives view controllers for showing the standard system interfaces for seeing and altering calendar related events
- GameKit Framework – implements support for Game Center which allows users share their game related information online

- iAd Framework – allows you deliver banner-based advertisements from your app.
- MapKit Framework – gives a scrollable map that you can include into your user interface of app.
- PushKitFramework – provides registration support for VoIP apps.
- Twitter Framework – supports a UI for generating tweets and support for creating URLs to access the Twitter service.
- UIKit Framework – gives vital infrastructure for applying graphical, event-driven apps in iOS. Some of the Important functions of UI Kit framework:

-Multitasking support.

- Basic app management and infrastructure.
- User interface management
- Support for Touch and Motion event.
- Cut, copy and paste support and many more.

Architecture of Windows

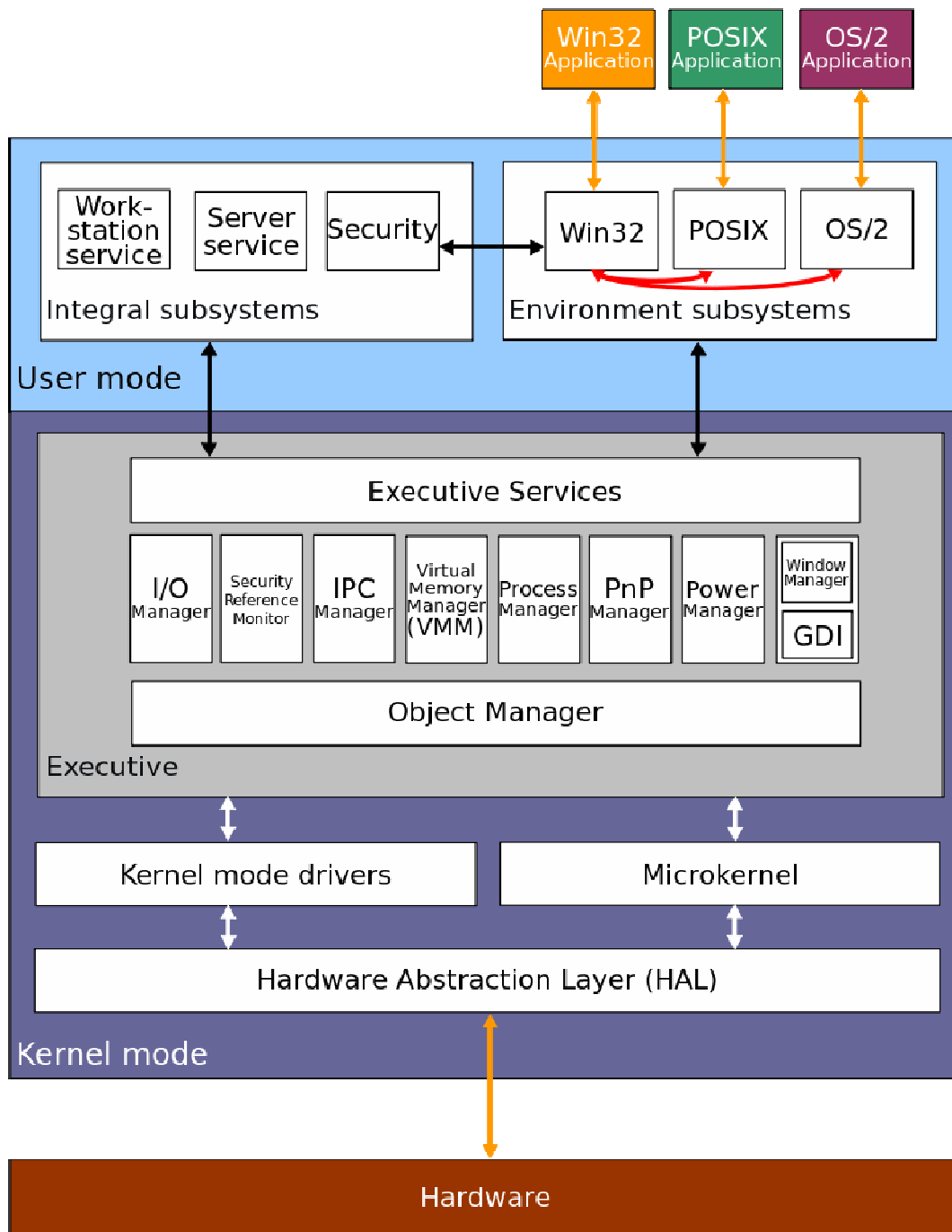
The architecture of Windows NT, a line of operating systems produced and sold by Microsoft, is a layered design that consists of two main components, user mode and kernel mode. It is a preemptive, reentrant multitasking operating system, which has been designed to work with uniprocessor and symmetrical multiprocessor (SMP)-based computers. To process input/output (I/O) requests, they use packet-driven I/O, which utilizes I/O request packets (IRPs) and asynchronous I/O. Starting with Windows XP, Microsoft began making 64-bit versions of Windows available; before this, there were only 32-bit versions of these operating systems.

Programs and subsystems in user mode are limited in terms of what system resources they have access to, while the kernel mode has unrestricted access to the system memory and external devices. Kernel mode in Windows NT has full access to the hardware and system resources of the computer. The Windows NT kernel is a hybrid kernel; the architecture comprises a simple kernel, hardware abstraction layer (HAL), drivers, and a range of services (collectively named Executive), which all exist in kernel mode.[1]

User mode in Windows NT is made of subsystems capable of passing I/O requests to the appropriate kernel mode device drivers by using the I/O manager. The user mode layer of Windows NT is made up of the "Environment subsystems", which run applications written for many different types of operating systems, and the "Integral subsystem", which operates system-specific functions on behalf of environment subsystems. The kernel mode stops user mode services and applications from accessing critical areas of the operating system that they should not have access to.

The Executive interfaces, with all the user mode subsystems, deal with I/O, object management, security and process management. The kernel sits between the hardware abstraction layer and the Executive to provide multiprocessor synchronization, thread and interrupt scheduling and dispatching, and trap handling and exception dispatching. The kernel is also responsible for initializing device drivers at bootup. Kernel mode drivers exist in three levels: highest level drivers, intermediate drivers and low-level drivers. Windows Driver Model (WDM) exists in the intermediate

layer and was mainly designed to be binary and source compatible between Windows 98 and Windows 2000. The lowest level drivers are either legacy Windows NT device drivers that control a device directly or can be a plug and play (PnP) hardware bus.



User mode

The user mode is made up of subsystems which can pass I/O requests to the appropriate kernel mode drivers via the I/O manager (which exists in kernel mode). Two subsystems make up the user mode layer of Windows 2000: the *Environment subsystem* and the *Integral subsystem*.

The environment subsystem was designed to run applications written for many different types of operating systems. None of the environment subsystems can directly access hardware, and must request access to memory resources through the Virtual Memory Manager that runs in kernel mode. Also, applications run at a lower priority than kernel mode processes. Currently, there are three main environment subsystems: the Win32 subsystem, an OS/2 subsystem and a POSIX subsystem.

The Win32 environment subsystem can run 32-bit Windows applications. It contains the console as well as text window support, shutdown and hard-error handling for all other environment subsystems. It also supports Virtual DOS Machines (VDMs), which allow MS-DOS and 16-bit Windows 3.x (Win16) applications to run on Windows. There is a specific MS-DOS VDM which runs in its own address space and which emulates an Intel 80486 running MS-DOS 5. Win16 programs, however, run in a Win16 VDM. Each program, by default, runs in the same process, thus using the same address space, and the Win16 VDM gives each program its own thread to run on. However, Windows 2000 does allow users to run a Win16 program in a separate Win16 VDM, which allows the program to be preemptively multitasked as Windows 2000 will pre-empt the whole VDM process, which only contains one running application. The OS/2 environment subsystem supports 16-bit character-based OS/2 applications and emulates OS/2 1.x, but not 32-bit or graphical OS/2 applications as used with OS/2 2.x or later. The POSIX environment subsystem supports applications that are strictly written to either the POSIX.1 standard or the related ISO/IEC standards.

The integral subsystem looks after operating system specific functions on behalf of the environment subsystem. It consists of a *security subsystem*, a *workstation service* and a *server service*. The security subsystem deals with security tokens, grants or denies access to user accounts based on resource permissions, handles logon requests and initiates logon

authentication, and determines which system resources need to be audited by Windows 2000. It also looks after Active Directory. The workstation service is an API to the network redirector, which provides the computer access to the network. The server service is an API that allows the computer to provide network services.

Kernel mode

Windows 2000 kernel mode has full access to the hardware and system resources of the computer and runs code in a protected memory area. It controls access to scheduling, thread prioritisation, memory management and the interaction with hardware. The kernel mode stops user mode services and applications from accessing critical areas of the operating system that they should not have access to as user mode processes ask the kernel mode to perform such operations on its behalf.

Kernel mode consists of *executive services*, which is itself made up on many modules that do specific tasks, *kernel drivers*, a *kernel* and a *Hardware Abstraction Layer*, or HAL.

Executive

The Executive interfaces with all the user mode subsystems. It deals with I/O, object management, security and process management. It contains various components, including the *I/O Manager*, the *Security Reference Monitor*, the *Object Manager*, the *IPC Manager*, the *Virtual Memory Manager* (VMM), a *PnP Manager* and *Power Manager*, as well as a *Window Manager* which works in conjunction with the *Windows Graphics Device Interface* (GDI). Each of these components exports a kernel-only support routine allows other components to communicate with one another. Grouped together, the components can be called *executive services*. No executive component has access to the internal routines of any other executive component.

The **object manager** is a special executive subsystem that all other executive subsystems must pass through to gain access to Windows 2000 resources — essentially making it a resource management infrastructure service. The object manager is used to reduce the duplication of object

resource management functionality in other executive subsystems, which could potentially lead to bugs and make development of Windows 2000 harder . To the object manager, each resource is an object, whether that resource is a physical resource (such as a file system or peripheral) or a logical resource (such as a file). Each object has a structure or *object type* that the object manager must know about. When another executive subsystem requests the creation of an object, they send that request to the object manager which creates an empty object structure which the requesting executive subsystem then fills in . Object types define the object procedures and any data specific to the object. In this way, the object manager allows Windows 2000 to be an object oriented operating system, as object types can be thought of as classes that define objects.

Each instance of an object that is created stores its name, parameters that are passed to the object creation function, security attributes and a pointer to its object type. The object also contains an object close procedure and a reference count to tell the object manager how many other objects in the system reference that object and thereby determines whether the object can be destroyed when a close request is sent to it . Every object exists in a hierarchical object namespace.

Further executive subsystems are the following:

- **I/O Manager:** allows devices to communicate with user-mode subsystems. It translates user-mode read and write commands in read or write IRPs which it passes to device drivers. It accepts file system I/O requests and translates them into device specific calls, and can incorporate low-level device drivers that directly manipulate hardware to either read input or write output. It also includes a cache manager to improve disk performance by caching read requests and write to the disk in the background
- **Security Reference Monitor (SRM):** the primary authority for enforcing the security rules of the security integral subsystem . It determines whether an object or resource can be accessed, via the use of access control lists (ACLs), which are themselves made up of access control entries (ACEs). ACEs contain a security identifier (SID) and a list of operations that the ACE gives a select group of trustees — a user

account, group account, or logon session — permission (allow, deny, or audit) to that resource.

- **IPC Manager:** short for Interprocess Communication Manager, this manages the communication between clients (the environment subsystem) and servers (components of the Executive). It can use two facilities: the *Local Procedure Call* (LPC) facility (clients and servers on the one computer) and the *Remote Procedure Call* (RPC) facility (where clients and servers are situated on different computers. Microsoft has had significant security issues with the RPC facility .
- **Virtual Memory Manager:** manages virtual memory, allowing Windows 2000 to use the hard disk as a primary storage device (although strictly speaking it is secondary storage). It controls the paging of memory in and out of physical memory to disk storage.
- **Process Manager:** handles process and thread creation and termination
- **PnP Manager:** handles Plug and Play and supports device detection and installation at boot time. It also has the responsibility to stop and start devices on demand — sometimes this happens when a bus gains a new device and needs to have a device driver loaded to support that device. Both FireWire and USB are hot-swappable and require the services of the PnP Manager to load, stop and start devices. The PnP manager interfaces with the HAL, the rest of the executive (as necessary) and with device drivers.
- **Power Manager:** the power manager deals with power events and generates power IRPs. It coordinates these power events when several devices send a request to be turned off it determines the best way of doing this.
- The display system has been moved from user mode into the kernel mode as a device driver contained in the file *Win32k.sys*. There are two components in this device driver — the Window Manager and the GDI:
 - **Window Manager:** responsible for drawing windows and menus. It controls the way that output is painted to the screen and handles input events (such as from the keyboard and mouse), then passes messages to the applications that need to receive this input

- **GDI:** the Graphics Device Interface is responsible for tasks such as drawing lines and curves, rendering fonts and handling palettes. Windows 2000 introduced native alpha blending into the GDI.

Kernel & kernel-mode drivers

The kernel sits between the HAL and the Executive and provides multiprocessor synchronization, thread and interrupt scheduling and dispatching, and trap handling and exception dispatching. The kernel often interfaces with the process manager. The kernel is also responsible for initialising device drivers at bootup that are necessary to get the operating system up and running.

Windows 2000 uses kernel-mode device drivers to enable it to interact with hardware devices. Each of the drivers has well defined system routines and internal routines that it exports to the rest of the operating system. All devices are seen by user mode code as a file object in the I/O manager, though to the I/O manager itself the devices are seen as device objects, which it defines as either file, device or driver objects. Kernel mode drivers exist in three levels: highest level drivers, intermediate drivers and low level drivers. The highest level drivers, such as file system drivers for FAT and NTFS, rely on intermediate drivers. Intermediate drivers consist of function drivers — or main driver for a device — that are optionally sandwiched between lower and higher level filter drivers. The function driver then relies on a bus driver — or a driver that services a bus controller, adapter, or bridge — which can have an optional bus filter driver that sits between itself and the function driver. Intermediate drivers rely on the lowest level drivers to function. The Windows Driver Model (WDM) exists in the intermediate layer. The lowest level drivers are either legacy Windows NT device drivers that control a device directly or can be a PnP hardware bus. These lower level drivers directly control hardware and do not rely on any other drivers..

REFERENCES

1. Erica Sadun, "The iOS 5 Developer's Cookbook: Core Concepts and Essential Recipes for iOS Programmers", Addison Wesley, 3rd Edition, 2011.
2. G. Blake Meike, Zigurd Mednieks, John Lombardo, Rick Rogers, "Android Application Development", O'reilly, 1st Edition, 2009.
3. R. Nageswara Rao, "Core JAVA: An Integrated Approach", Dreamtech Press, Wiley India, 1st Edition, 2015.
4. https://en.wikipedia.org/wiki/Mobile_app_development
5. <https://www.korcomptenz.com/mobile-app-development>
6. <http://garryowen.csisdmsz.ul.ie/~cs5212/resources/oth8.pdf>
7. https://en.wikipedia.org/wiki/Windows_Mobile
8. <https://www.slideshare.net/Bhavsidd/windows-phone-7-architecture-overview>
9. <https://www.tutorialspoint.com/apple-ios-architecture>
10. <https://intellipaat.com/blog/tutorial/ios-tutorial/ios-architecture/>
11. <https://www.besanttechnologies.com/what-is-ios>

UNIT - 1

INTRODUCTION TO VARIOUS MOBILE PLATFORMS

Part - A

1. Describe the characteristics of mobile application development?
2. Express briefly the advantages of various mobile platforms.
3. Demonstrate in detail about mobile security and its impact?
4. Summarize the features of Linux kernel of Android os?
5. Discuss about various android libraries in android platform?
6. Discuss in detail about ios hardware?
7. Describe about cocoa touch layer in ios?
8. Classify the various framework in windows os?
9. Describe about the windows os kernel?
10. Explain in detail the impact of J2ME in mobile application development?

Part - B

1. With neat blocks express the architecture of Android OS .
2. Discuss briefly the architecture of Windows OS with neat block diagram.
3. Interpret the concepts of ios and its impact on mobile application development with the neat diagram?



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF EEE

DEPARTMENT OF ECE

UNIT – II - ANDROID ACTIVITIES: STUDY AND CREATION – SECA5205

UNIT 2 ANDROID ACTIVITIES: STUDY AND CREATION

Introduction to Android, Why develop for android OS, Android SDK features, Creating android activities.

What is Android?

Android is an open-source operating system based on Linux with a Java programming interface for mobile devices such as Smartphone (Touch Screen Devices who supports Android OS) as well for Tablets too.

Android was developed by the Open Handset Alliance (OHA), which is led by Google. The Open Handset Alliance(OHA) is a consortium of multiple companies like Samsung, Sony, Intel and many more to provide services and deploy handsets using the android platform.

In 2007, Google released a first beta version of the Android Software Development Kit (SDK) and the first commercial version of Android 1.0 (with name Alpha), was released in September 2008.

In 2012, Google released another version of android, 4.1 Jelly Bean. It's an incremental update and it improved a lot in terms of the user interface, functionality, and performance.

In 2014, Google announced another Latest Version, 5.0 Lollipop. In Lollipop version Google completely revamped the UI by using Material Designs, which is good for the User Interface as well for the themes related.

All the source code for Android is available free on Git-Hub, Stack overflow, and many more websites. Google publishes most of the code under the Apache License version 2.0

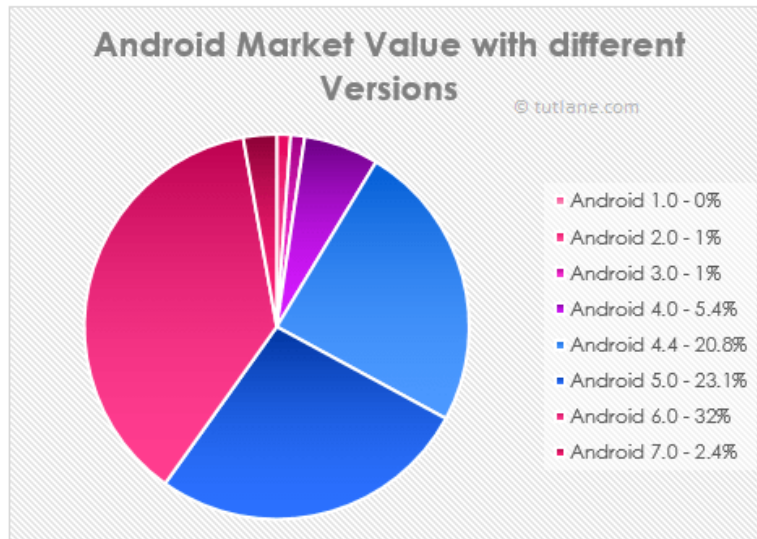
Android Features

Android is a powerful open-source operating system which provides a lot of great features, those are

- It supports connectivity for GSM, CDMA, WIFI, NFC, Bluetooth, etc. for telephony or data transfer. It will allow us to make or receive a calls / SMS messages and we can send or retrieve data across mobile networks
- By using WIFI technology we can pair with other devices using apps
- Android has multiple APIs to support location-based services such as GPS
- We can perform all data storage related activities by using lightweight database SQLite (/tutorial/sqlite).
- It has a wide range of media supports like AVI, MKV, FLV, MPEG4, etc. to play or record a variety of audio/video and having a different image format like JPEG, PNG, GIF, BMP, MP3, etc.
- It has extensive support for multimedia hardware control to perform playback or recording using camera and microphone

- It has an integrated open-source WebKit layout based web browser to support HTML5, CSS3
- It supports a multi-tasking, we can move from one task window to another and multiple applications can run simultaneously
- It will give a chance to reuse the application components and the replacement of native applications.
- We can access the hardware components like Camera, GPS, and Accelerometer
- It has support for 2D/3D Graphics

Following is the pictorial representation of using an android in the mobile phone market with different versions.



This is how Google released multiple versions of the android operating system and acquired a huge mobile phone market share with different versions.

Android Components

Application components are the essential building blocks of an Android application.



The main components of any android application are the following:

- Activities
- Services
- Content Providers
- Intent and broadcast receivers
- Widgets and Notifications

1. Activities

We can call the Activity as the presentation layer of an Android application. Simply put, an Activity represents the screen on your Android application which has its user interface. An application, for instance, an Email App can have many activities such as opening an email, composing an email, replying to an email – these all are different activities. So every Android application has more than one activity. When we start a new activity (like replying to an email), previous activity is pushed to the back stack and it gets stopped until the new activity is finished, however, if we push back button while ongoing activity, the current activity gets dissolved and is popped out of the stack and previous activity resumes.

2. Services

The other important component of an Android application is the service. It performs running operations (long or short) in the background for the activity that you perform on your screen. For example, a push notification from an email. It is possible that service still runs while you have terminated the application or you are not using it currently. For example, when you get an email, you get the notification while still, you are not using the application currently.

3. Content Providers

Content Providers manage the application Data and encapsulate it (Object Oriented Feature). This provides the data from one processor of an application to another one. The data might be stored in Database or in a file system or any other storage management systems. Android devices include several native Content Providers that expose useful databases such as the media store and contacts.

4. Intent and broadcast receivers

Android Intents are the means of communication that acts as a facilitator when the exchange of message occurs between different components within the same application or from one application to another. In order to start any service, we have to pass an intent to perform this task. Intents are of two types:

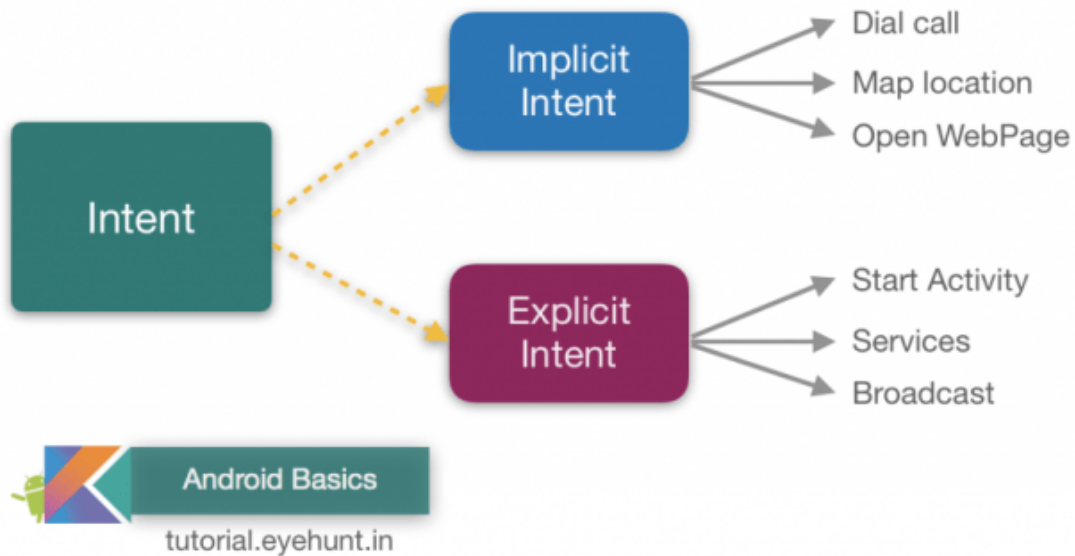
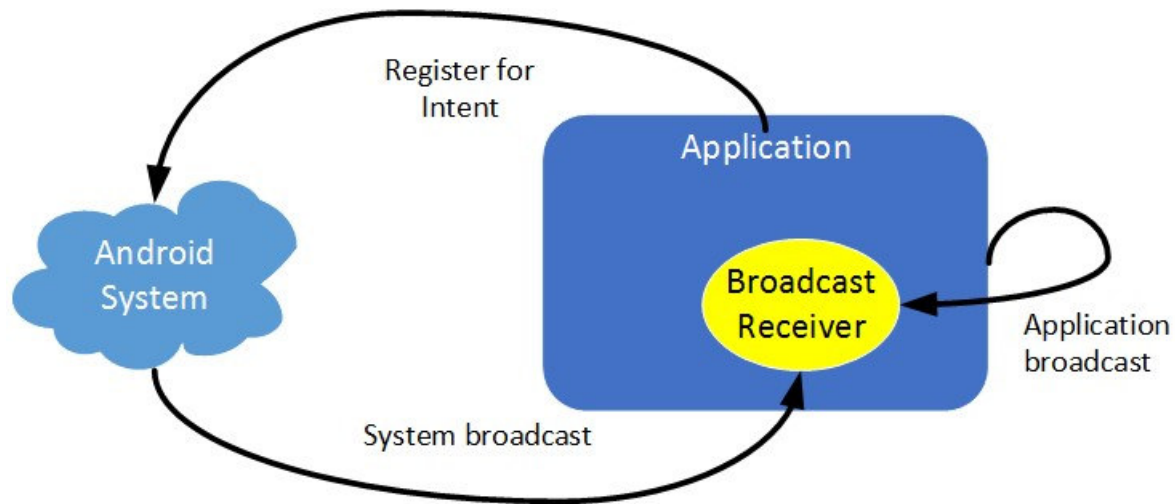


Image Source: <https://goo.gl/images/AqiSpu>

- **Implicit Intents:** It does not declare the name of the service to start but declares the action to perform.
- **Explicit Intents:** It specifies the exact activity to which intent should be given.

Broadcast Receivers enable your application to listen for intents that match the criteria you specify. As an example, applications can start the broadcasts to let other applications know that few data has been downloaded to the device and is available for them to use. There are two types of broadcast:

- **Normal Broadcast:** They are completely asynchronous and all receivers of the broadcast are run in an undefined order.
- **Ordered Broadcast:** They are synchronous and are delivered to one receiver one at a time.



5. Widgets and Notifications

Widgets display your app interesting or new content in the consolidated form on a mobile or tablet home screen. The user can do different activities like moving and resizing of widgets. There are basically four types of widgets:

- **Information Widget** – This widget displays only the important information to the users. For e.g. the clock on the home screen.
- **Collection Widget** – This widget displays multiple information of the same type and allows you to select any of them to open. For example, when you open an email application, you see multiple emails.
- **Control Widget** – This widget displays frequently used functions. For example, the music app widget allows the user to play, music from outside of an application.
- **Hybrid Widget** – This widget combines the information from above all three widgets.

Notifications allow informing users of any events that have occurred. For e.g., we use what's app application, as and when a message comes, we get a notification.

Characteristics of Android

- Android can run multiple applications at the same time.
- Android widgets let you display just about any feature you choose right on the home screen.
- Android supports multiple keyboards and it is super easy to install them.
- Android supports Video Graphics Array, 2D, and 3D graphics alongside.

- Android also supports Java applications.
- One can change settings quite faster when Android is running on the phone
- The very good app market
- Most Android devices support NFC, which allows electronic devices to easily interact across short distances.

Applications of Android

Android applications are software applications which are running on Android platform. We have already seen the components of the android application previously as composed of one or more application components like activities, services, content providers, and broadcast receivers. Android apps are written in the Java programming language and use Java core libraries. For Android app development, developers may download the Software Development Kit (SDK) from the android website. The SDK includes tools, sample code and relevant documents for creating Android apps.

What is the Android SDK?

Every time Google releases a new version, the corresponding SDK is also released. In order to work with Android, the developers must download and install each version's SDK for the particular device.

The Android SDK (Software Development Kit) is a set of development tools that are used to develop applications for the Android platform.

This SDK provides a selection of tools that are required to build Android applications and ensures the process goes as smoothly as possible. Whether you create an application using Java, Kotlin or C#, you need the SDK to get it to run on any Android device. You can also use an emulator in order to test the applications that you have built.

Nowadays, the Android SDK also comes bundled with Android Studio, the integrated development environment where the work gets done and many of the tools are now best accessed or managed.

Android SDK Features

Android SDK has a lot of amazing features. I've tried noting down most of them. So, have a look!

- Offline Mapping

SDK helps in dynamically downloading the maps for more than 190 countries in over 60 languages. You can view these offline. Also dealing with the map styles and the touch gesture. This SDK also has the ability to render raster tiles and map objects interleaved within different map layers.



- Dynamic Markers

In the previous versions, you could not have moved the position without a fallback or re-adding the icon. But in the latest edition, you can update the position of the icon dynamically.

- Improved API compatibility

With the latest release, it is much easier to migrate from the Google Maps Android API. This is another added advantage of using Android SDK in your program.

SDK Tools

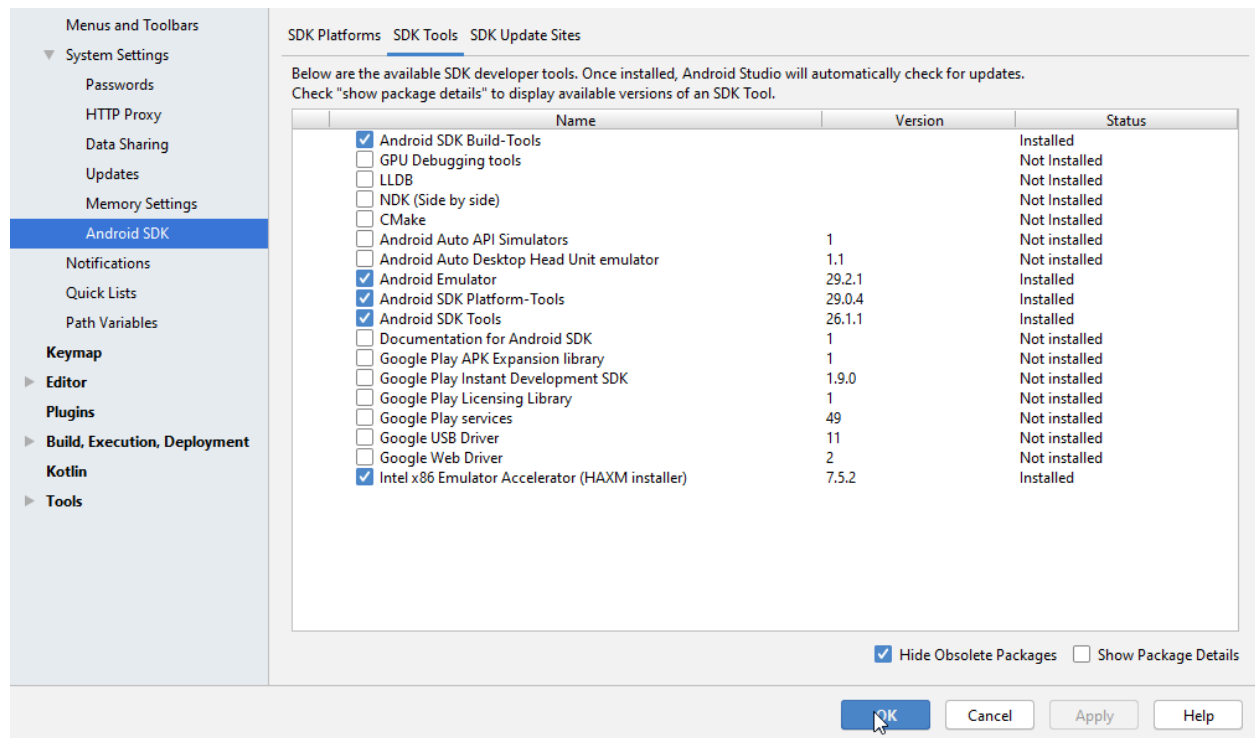
Android SDK Tools is a component for the Android SDK. This includes a complete set of development and debugging tools for Android. SDK tools are also included with Android Studio.

Android comes up with the revised version every now and then the latest release is SDK Tools, Revision 26.1.1 (September 2017)

In this release, they made a few changes. They are:

- A command-line version of the APK Analyzer has been added in tools/bin/apkanalyzer. It offers the same features as the APK Analyzer in Android Studio and can be integrated into build/CI servers and scripts for tracking size regressions, generating reports, and many more.
- ProGuard rules that are under the tools/proguard are no longer used by the Android Plugin for Gradle.

These change with each update.



SDK tools are generally platform-independent and they are required no matter which Android platform you are currently working on. There are a set of tools that get installed automatically when install Android Studio.

Tools	Description
Android	This tool lets you manage the AVD (Android Virtual Device), projects, and the installed components of the SDK.
Emulator	It lets you test your applications without using a physical device.
Proguard	This tool helps in shrinking, optimizing, and obscures your code by removing unused code.
ddms	It lets you debug your Android applications
Android Debug Bridge (Adb)	This is a versatile command-line tool that helps you communicate with an emulator instance or connected Android-powered device.

Android SDK manager

In order to download and install latest android APIs and development tools from the internet, Android helps us by having the Android SDK manager. This separates the APIs, tools and different platforms into different packages which you can download. Android SDK Manager comes with the Android SDK bundle. You can't download it separately.

Introduction to Activities

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

The concept of activities

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 user 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 that then launches your email app, you might go directly to the email app's screen for composing an email.

The Activity ([/reference/android/app/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 ([/reference/android/app/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. For instance, one of an app's activities may implement a Preferences screen, while another activity implements a Select Photo screen.

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.

To use activities in your app, you must register information about them in the app's manifest, and you must manage activity lifecycles appropriately. The rest of this document introduces these subjects.

Configuring the manifest

For your app to be able to use activities, you must declare the activities, and certain of their attributes, in the manifest.

Declare activities

To declare your activity, open your manifest file and add an `<activity>` ([/guide/topics/manifest/activity-element](#)) element as a child of the `<application>` ([/guide/topics/manifest/application-element](#)) element. For example:

```
<manifest ... >

<application ... >

  <activity android:name=".ExampleActivity" />

</application ... >

</manifest >
```

The only required attribute for this element is `android:name` ([/guide/topics/manifest/activity-element#nm](#)), which specifies the class name of the activity. You can also add attributes that define activity characteristics such as label, icon, or UI theme. For more information about these and other attributes, see the `<activity>` ([/guide/topics/manifest/activity-element](#)) element reference documentation.

Declare intent filters

Intent filters ([/guide/components/intents-filters](#)) are a very powerful feature of the Android platform. They provide the ability to launch an activity based not only on an explicit request, but also an implicit one. For example, an explicit request might tell the system to “Start the Send Email activity in the Gmail app”. By contrast, an implicit request tells the system to “Start a Send Email screen in any activity that can do the job.” When the system UI asks a user which app to use in performing a task, that’s an intent filter at work.

You can take advantage of this feature by declaring an `<intent-filter>` ([/guide/topics/manifest/intent-filter-element](#)) attribute in the `<activity>` ([/guide/topics/manifest/activity-element](#)) element. The definition of this element includes an `<action>` ([/guide/topics/manifest/action-element](#)) element and, optionally, a `<category>` ([/guide/topics/manifest/category-element](#)) element and/or a `<data>` ([/guide/topics/manifest/data-element](#)) element. These elements combine to specify the type of intent to which your activity can respond. For example, the following code snippet shows how to configure an activity that sends text data, and receives requests from other activities to do so:

```
<activity android:name=".ExampleActivity" android:icon="@drawable/app_icon"> <intent-filter>

  <action                android:name="android.intent.action.SEND"                />                <category
  android:name="android.intent.category.DEFAULT" /> <data android:mimeType="text/plain" />

</intent-filter>
```

</activity>

In this example, the <action> (/guide/topics/manifest/action-element) element specifies that this activity sends data. Declaring the <category> (/guide/topics/manifest/category-element) element as DEFAULT enables the activity to receive launch requests. The <data> (/guide/topics/manifest/data-element) element specifies the type of data that this activity can send. The following code snippet shows how to call the activity described above:

```
val sendIntent = Intent().apply {  
  
    action = Intent.ACTION_SEND  
  
    type = "text/plain"  
  
    putExtra(Intent.EXTRA_TEXT, textMessage)  
  
}  
  
startActivity(sendIntent)
```

If you intend for your app to be self-contained and not allow other apps to activate its activities, you don't need any other intent filters. Activities that you don't want to make available to other applications should have no intent filters, and you can start them yourself using explicit intents. For more information about how your activities can respond to intents, see [Intents and Intent Filters](#) (/guide/components/intent-filters).

Declare permissions

You can use the manifest's <activity> (/guide/topics/manifest/activity-element) tag to control which apps can start a particular activity. A parent activity cannot launch a child activity unless both activities have the same permissions in their manifest. If you declare a <uses-permission> (/guide/topics/manifest/uses-permission-element) element for a parent activity, each child activity must have a matching <uses-permission>

(/guide/topics/manifest/uses-permission-element) element.

For example, if your app wants to use a hypothetical app named SocialApp to share a post on social media, SocialApp itself must define the permission that an app calling it must have:

```
<manifest>  
  
    <activity android:name="...."  
  
        android:permission="com.google.socialapp.permission.SHARE_POST"  
  
    />
```

Then, to be allowed to call SocialApp, your app must match the permission set in SocialApp's manifest:

```
<manifest>
```

```
<uses-permission android:name="com.google.socialapp.permission.SHARE_POST" /> </manifest>
```

For more information on permissions and security in general, see [Security and Permissions \(/guide/topics/security/security\)](/guide/topics/security/security).

Managing the activity lifecycle

Over the course of its lifetime, an activity goes through a number of states. You use a series of callbacks to handle transitions between states. The following sections introduce these callbacks.

onCreate()

You must implement this callback, which fires when the system creates your activity. Your implementation should initialize the essential components of your activity: For example, your app should create views and bind data to lists here. Most importantly, this is where you must call `setContentView()` ([/reference/android/app/Activity#setContentView\(android.view.View\)](/reference/android/app/Activity#setContentView(android.view.View))) to define the layout for the activity's user interface.

When `onCreate()` ([/reference/android/app/Activity#onCreate\(android.os.Bundle\)](/reference/android/app/Activity#onCreate(android.os.Bundle))) finishes, the next callback is always `onStart()` ([/reference/android/app/Activity#onStart\(\)](/reference/android/app/Activity#onStart())).

onStart()

As `onCreate()` ([/reference/android/app/Activity#onCreate\(android.os.Bundle\)](/reference/android/app/Activity#onCreate(android.os.Bundle))) exits, the activity enters the Started state, and the activity becomes visible to the user. This callback contains what amounts to the activity's final preparations for coming to the foreground and becoming interactive.

onResume()

The system invokes this callback just before the activity starts interacting with the user. At this point, the activity is at the top of the activity stack, and captures all user input. Most of an app's core functionality is implemented in the `onResume()` ([/reference/android/app/Activity#onResume\(\)](/reference/android/app/Activity#onResume())) method. The `onPause()` ([/reference/android/app/Activity#onPause\(\)](/reference/android/app/Activity#onPause())) callback always follows `onResume()` ([/reference/android/app/Activity#onResume\(\)](/reference/android/app/Activity#onResume())).

onPause()

The system calls `onPause()` ([/reference/android/app/Activity#onPause\(\)](/reference/android/app/Activity#onPause())) when the activity loses focus and enters a Paused state. This state occurs when, for example, the user taps the Back or Recents button. When the system calls `onPause()` ([/reference/android/app/Activity#onPause\(\)](/reference/android/app/Activity#onPause())) for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.

An activity in the Paused state may continue to update the UI if the user is expecting the UI to update. Examples of such an activity include one showing a navigation map screen or a media player playing. Even if such activities lose focus, the user expects their UI to continue updating.

You should not use `onPause()` ([/reference/android/app/Activity#onPause\(\)](/reference/android/app/Activity#onPause())) to save application or user data, make network calls, or execute database transactions. For information about saving data, see [Saving and restoring activity state \(/guide/components/activities/activity-lifecycle#saras\)](/guide/components/activities/activity-lifecycle#saras).

`onStop()`

The system calls `onStop()` ([/reference/android/app/Activity#onStop\(\)](/reference/android/app/Activity#onStop())) when the activity is no longer visible to the user. This may happen because the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity. In all of these cases, the stopped activity is no longer visible at all. The next callback that the system calls is either `onRestart()` ([/reference/android/app/Activity#onRestart\(\)](/reference/android/app/Activity#onRestart())), if the activity is coming back to interact with the user, or by `onDestroy()` ([/reference/android/app/Activity#onDestroy\(\)](/reference/android/app/Activity#onDestroy())) if this activity is completely terminating.

`onRestart()`

The system invokes this callback when an activity in the Stopped state is about to restart. `onRestart()` ([/reference/android/app/Activity#onRestart\(\)](/reference/android/app/Activity#onRestart())) restores the state of the activity from the time that it was stopped. This callback is always followed by `onStart()` ([/reference/android/app/Activity#onStart\(\)](/reference/android/app/Activity#onStart())).

`onDestroy()`

The system invokes this callback before an activity is destroyed. This callback is the final one that the activity receives. `onDestroy()` ([/reference/android/app/Activity#onDestroy\(\)](/reference/android/app/Activity#onDestroy())) is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

Understand the Activity Lifecycle

As a user navigates through, out of, and back to your app, the [Activity](#) instances in your app transition through different states in their lifecycle. The [Activity](#) class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, or resuming an activity, or destroying the process in which the activity resides.

Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity. For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot. In other words, each callback allows you to perform specific work that's appropriate to a given change of state. Doing the right work at the right time and handling transitions properly make your app more robust and performant. For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:

- Crashing if the user receives a phone call or switches to another app while using your app.
- Consuming valuable system resources when the user is not actively using it.

- Losing the user's progress if they leave your app and return to it at a later time.
- Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

This document explains the activity lifecycle in detail. The document begins by describing the lifecycle paradigm. Next, it explains each of the callbacks: what happens internally while they execute, and what you should implement during them. It then briefly introduces the relationship between activity state and a process's vulnerability to being killed by the system. Last, it discusses several topics related to transitions between activity states.

For information about handling lifecycles, including guidance about best practices, see [Handling Lifecycles with Lifecycle-Aware Components](#) and [Saving UI States](#). To learn how to architect a robust, production-quality app using activities in combination with architecture components, see [Guide to App Architecture](#).

Activity-lifecycle concepts

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of six callbacks: [onCreate\(\)](#), [onStart\(\)](#), [onResume\(\)](#), [onPause\(\)](#), [onStop\(\)](#), and [onDestroy\(\)](#). The system invokes each of these callbacks as an activity enters a new state.

Figure below presents a visual representation of this paradigm.

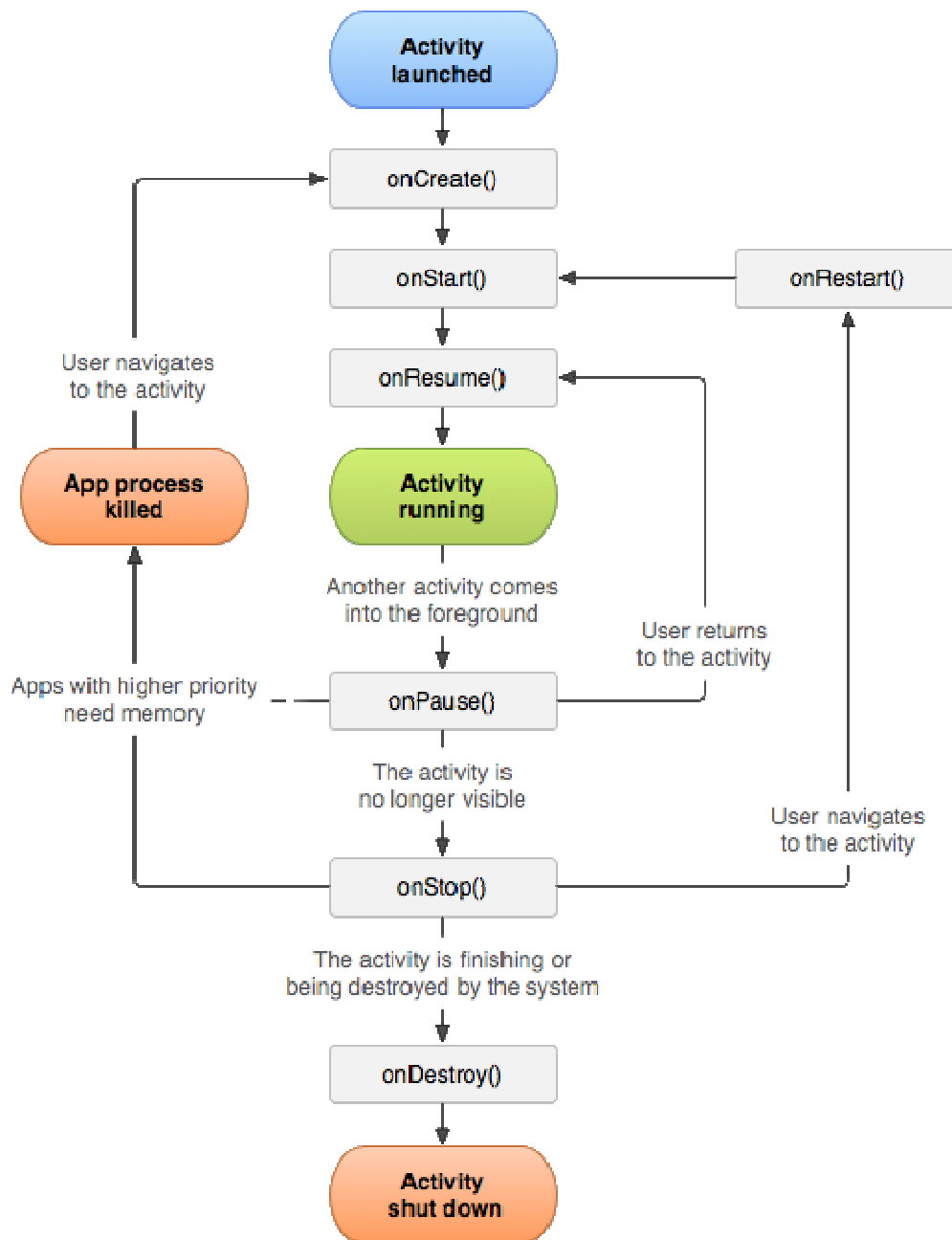


Figure. A simplified illustration of the activity lifecycle.

As the user begins to leave the activity, the system calls methods to dismantle the activity. In some cases, this dismantlement is only partial; the activity still resides in memory (such as when the user switches to another app), and can still come back to the foreground. If the user returns to that activity, the activity resumes from where the user left off. With a few exceptions, apps are [restricted from starting activities when running in the background](#).

The system's likelihood of killing a given process—along with the activities in it—depends on the state of the activity at the time. [Activity state and ejection from memory](#) provides more information on the relationship between state and vulnerability to ejection.

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

REFERENCES

1. G. Blake Meike, Zigurd Mednieks, John Lombardo, Rick Rogers, "Android Application Development", O'reilly, 1st Edition, 2009.
2. R. Nageswara Rao, "Core JAVA: An Integrated Approach", Dreamtech Press, Wiley India, 1st Edition, 2015.
3. Herbert Schildt, "Java: The Complete Reference", 9th Edition, 2014.
4. Gary Cornell, Cay S. Horstmann, "Core Java Volume I - Fundamentals", Prentice Hall, 9th Edition, 2012.
5. Cay S. Horstmann, "Core Java, Volume II - Advanced Features", Prentice Hall, 11th Edition, 2019. <https://www.besanttechnologies.com/what-is-ios>
6. <https://www.educba.com/introduction-to-android/>
7. <https://developer.android.com/guide/components/fundamentals>
8. <https://developer.android.com/studio/releases/platform-tools>
9. <https://code.tutsplus.com/tutorials/the-android-sdk-tutorial--cms-34623>
10. <https://www.javatpoint.com/android-life-cycle-of-activity>
11. https://www.tutorialspoint.com/android/android_activities.html
12. <https://developer.android.com/guide/components/activities/activity-lifecycle>

UNIT - 2

ANDROID ACTIVITIES: STUDY AND CREATION

Part – A

1. What is android? List any 5 android SDK features?
2. Explain in briefly any 5 android development tools?
3. Define an activity? Explain the life cycle activity?
4. Express the challenges faced in Android app development?
5. Mention few applications of Android platform?

Part – B

1. Define Activity? Explain the steps for creating activity? How to add more activity?
2. What is Activity Life Cycle? Explain with diagram and call back methods that support activity life cycle?
3. Explain the procedure for getting data back from an activity?
4. With neat notations discuss the concepts of Android SDK?



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF EEE

DEPARTMENT OF ECE

UNIT – III - ANDROID INTENT, THREAD AND SERVICES – SECA5205

UNIT 3 ANDROID INTENT, THREAD AND SERVICES

User Interface - Views, ViewGroups and Resources

An **Activity** interacts with the user, via a visual UI on a screen. The UI is placed on the **Activity** via the **Activity's setContentView()** method. In Android, the UI composes of **View** and **ViewGroup** objects, organized in a single view-tree structure.

A **View** is an interactive UI component (or widget or control), such as button and text field. It controls a *rectangular area* on the screen. It is responsible for drawing itself and handling events (such as clicking, entering texts). Android provides many ready-to-use Views such as **TextView**, **EditText**, **Button**, **RadioButton**, etc, in package **android.widget**. You can also create your custom **View** by extending **android.view.View**.

A **ViewGroup** is an *invisible container* used to *layout* the **View** components. Android provides many ready-to-use **ViewGroups** such as **LinearLayout**, **RelativeLayout**, **TableLayout** and **GridLayout** in package **android.widget**. You can also create your custom **ViewGroup** by extending from **android.view.ViewGroup**.

Views and **ViewGroups** are organized in a single tree structure called *view-tree*. You can create a view-tree either using programming codes or describing it in a XML layout file. XML layout is recommended as it separates the presentation view from the controlling logic, which provides modularity and flexibility in your program design. Once a view-tree is constructed, you can add the *root* of the view-tree to the **Activity** as the content view via **Activity's setContentView()** method.

There are two approaches in constructing the UI:

Build a simple user interface

In this lesson, you learn how to use the **Android Studio Layout Editor** (/studio/write/layout-editor)

to create a layout that includes a text box and a button. This sets up the next lesson, where you learn how to make the app send the content of the text box to another activity when the button is tapped.

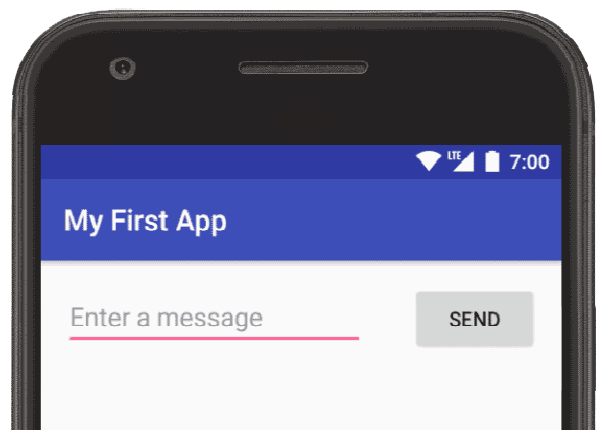
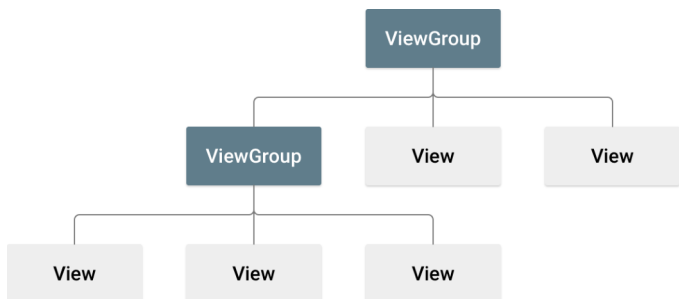


Figure 1. Screenshot of the final layout

The user interface (UI) for an Android app is built as a hierarchy of *layouts* and *widgets*. The layouts are **ViewGroup** (/reference/android/view/ViewGroup) objects, containers that control how their child views are positioned on the screen. Widgets are **View** (/reference/android/view/View) objects, UI components such as buttons and text boxes.

Figure 2. Illustration of how **ViewGroup** objects form branches in the layout and contain **View**

Android provides an XML vocabulary for `ViewGroup` and `View` classes, so most of your UI is defined in XML files. However, rather than teach you to write XML, this lesson shows you how to create a layout using Android Studio's Layout Editor. The Layout Editor writes the XML for you as you drag and drop views to build your layout.

This lesson assumes that you use [Android Studio v3.0 \(/studio\)](#) or higher and that you've completed the [create your Android project \(/training/basics/firstapp/creating-project\)](#) lesson.

Open the LayoutEditor

To get started, set up your workspace as follows:

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

2. To make room for the Layout Editor, hide the **Project** window. To do so, select **View > Tool Windows > Project**, or just click **Project**



on the left side of the Android Studio screen.

3. If your editor shows the XML source, click the **Design** tab at the bottom of the window.

4. Click **Select Design Surface**



and select **Blueprint**.

5. Click **Show**



in the Layout Editor toolbar and make sure that **Show All Constraints** is checked.

6. Make sure Autoconnect is off. A tooltip in the toolbar displays **Enable Autoconnection to Parent**



when Autoconnect is off.

7. Click **Default Margins** in the toolbar and select **16**. If needed, you can adjust the margins for each view later.

8. Click **Device for Preview**



in the toolbar and select **5.5, 1440 × 2560, 560 dpi (Pixel XL)**.

Your Layout Editor now looks as shown in figure 3.

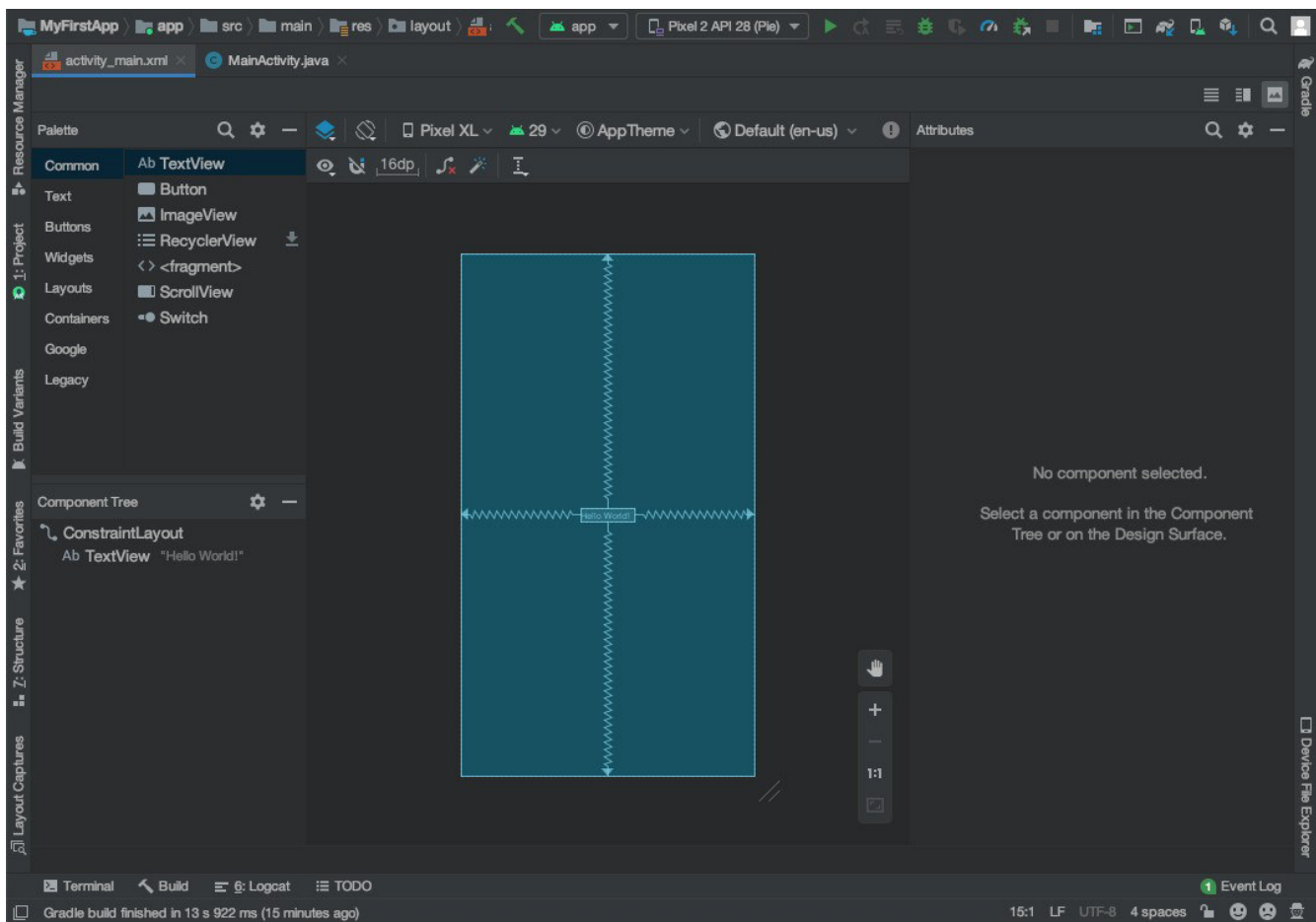


Figure 3. The Layout Editor showing `activity_main.xml`

For additional information, see [Introduction to the Layout Editor \(/studio/write/layout-editor#intro\)](#).

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

`ConstraintLayout` is a layout that defines the position for each view based on constraints to sibling views and the parent layout. In this way, you can create both simple and complex layouts with a flat view hierarchy. This type of layout avoids the need for nested layouts. A nested layout, which is a layout inside a layout, as shown in figure 2, can increase the time required to draw the UI.

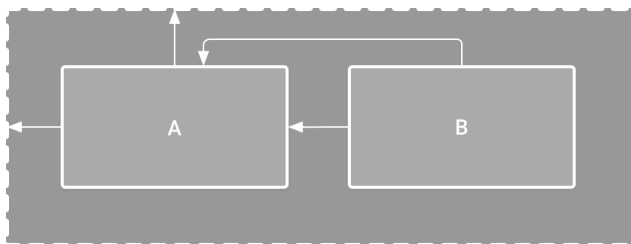


Figure 4. Illustration of two views positioned inside `ConstraintLayout`

For example, you can declare the following layout, which is shown in figure 4:

- ◆ View A appears 16 dp from the top of the parent layout. View A appears 16 dp from the left of the parent layout. View B appears 16 dp to the right of view A.
- ◆ View B is aligned to the top of view A.

In the following sections, you'll build a layout similar to the layout in figure 4.

Add a text box

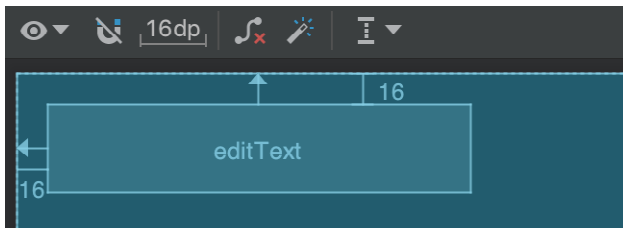


Figure 5. The text box is constrained to the top and left of the parent layout

Follow these steps to add a text box:

1. First, you need to remove what's already in the layout. Click **TextView** in the **Component Tree** panel and then press the Delete key.
2. In the **Palette** panel, click **Text** to show the available text controls.
3. Drag the **Plain Text** into the design editor and drop it near the top of the layout. This is an [EditText](/reference/android/widget/EditText) (/reference/android/widget/EditText) widget that accepts plain text input.
4. Click the view in the design editor. You can now see the square handles to resize the view on each corner, and the circular constraint anchors on each side. For better control, you might want to zoom in on the editor. To do so, use the **Zoom** buttons in the Layout Editor toolbar.
5. Click and hold the anchor on the top side, drag it up until it snaps to the top of the layout, and then release it. That's a constraint: it constrains the view within the default margin that was set. In this case, you set it to 16 dp from the top of the layout.

6. Use the same process to create a constraint from the left side of the view to the left side of the layout.

The result should look as shown in figure 5.

Add a button

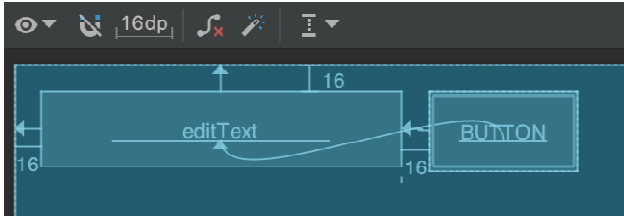


Figure 6. The button is constrained to the right side of the text box and its baseline

1. In the **Palette** panel, click **Buttons**.
2. Drag the **Button** widget into the design editor and drop it near the right side.
3. Create a constraint from the left side of the button to the right side of the text box.
4. To constrain the views in a horizontal alignment, create a constraint between the text baselines. To do so, right-click the button and then select **Show**



. The baseline anchor appears inside the button. Click and hold this anchor, and then drag it to the baseline anchor that appears in the adjacent text box.

The result should look as shown in figure 6.

Note: You can also use the top or bottom edges to create a horizontal alignment. However, the button image includes padding around it, so the visual alignment is wrong if created that way.

Change the UI strings

Follow these steps to change the UI strings:

1. Open the **Project** window and then open **app > res > values > strings.xml**.

This is a [string resources](/guide/topics/resources/string-resource) (/guide/topics/resources/string-resource) file, where you can specify all of your UI strings. It allows you to manage all of your UI strings in a single location, which makes them easier to find, update, and localize.

2. Click **Open editor** at the top of the window. This opens the [Translations Editor](/studio/write/translations-editor) (/studio/write/translations-editor), which provides a simple interface to add and edit your default strings. It also helps you keep all of your translated strings organized.

3. Click **Add Key**



to create a new string as the "hint text" for the text box. At this point, the window shown in figure 7 opens.

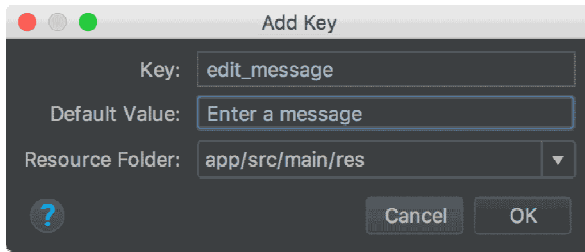


Figure 7. The dialog to add a new string

In the **Add Key** dialog box, complete the following steps:

- a. Enter "edit_message" in the **Key** field.
 - b. Enter "Enter a message" in the **Default Value** field.
 - c. Click **OK**.
4. Add another key named "button_send" with a value of "Send".

Now you can set these strings for each view. To return to the layout file, click **activity_main.xml** in the tab bar. Then, add the strings as follows:

1. Click the text box in the layout. If the **Attributes** window isn't already visible on the right, click **Attributes**
2. Locate the **text** property, which is currently set to "Name," and delete the value.
3. Locate the **hint** property and then click **Pick a Resource**
, which is to the right of the text box. In the dialog that appears, double-click **edit_message** from the list.
4. Click the button in the layout and locate its **text** property, which is currently set to "Button." Then, click **Pick a Resource**
and select **button_send**.

Make the text box size flexible

To create a layout that's responsive to different screen sizes, you need to make the text box stretch to fill all the horizontal space that remains after the button and margins are accounted for.

Before you continue, click **Select Design Surface** in the toolbar and select **Blueprint**.

To make the text box flexible, follow these steps:

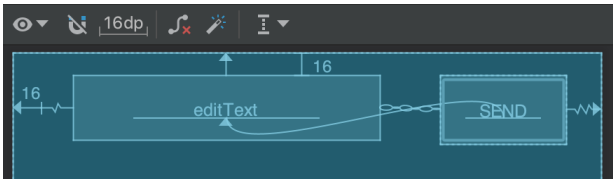


Figure 8. The result of choosing **Create Horizontal Chain**

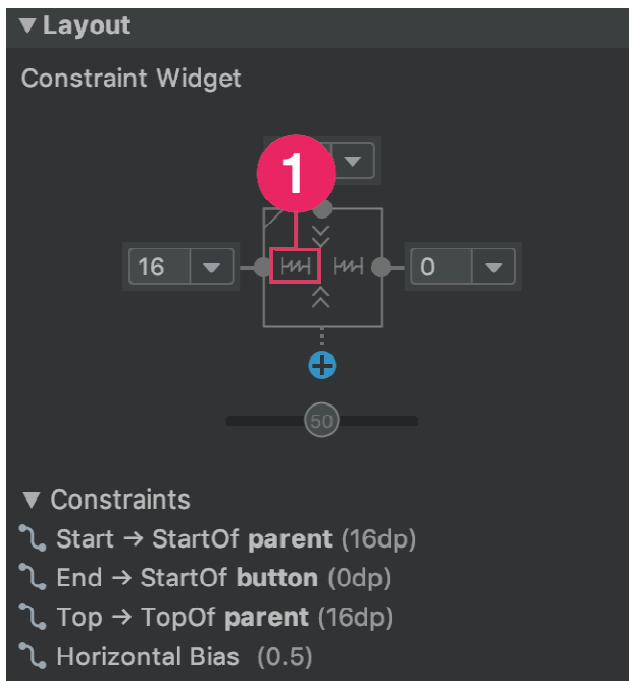


Figure 9. Click to change the width to **Match Constraints**

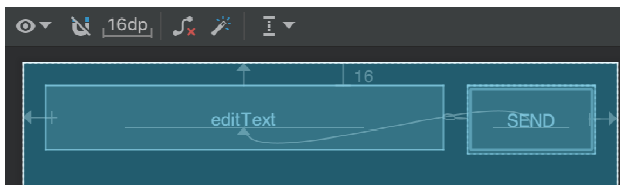


Figure 10. The text box now stretches to fill the remaining space

1. Select both views. To do so, click one, hold **Shift**, then click the other, and then right- click either one and select **Chains > Create Horizontal Chain**. The layout then appears as shown in figure 8.

A chain (</training/constraint-layout#constrain-chain>) is a bidirectional constraint between two or more views that allows you to lay out the chained views in unison.

2. Select the button and open the **Attributes** window. Then, use the view inspector at the top of the **Attributes** window to set the right margin to 16 dp.
3. Click the text box to view its attributes. Then, click the width indicator twice so it's set to **Match Constraints**, as indicated by callout 1 in figure 9.

Match constraints means that the width expands to meet the definition of the horizontal constraints and margins. Therefore, the text box stretches to fill the horizontal space that remains after the button and all the margins are accounted for.

Now the layout is done, as shown in figure 10.

If your layout didn't turn out as expected, click **See the final layout XML** below to see what your XML should look like. Compare it to what you see in the **Text** tab. If your attributes appear in a different order, that's okay.

[See the final layout XML](#)

```
<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.myfirstapp.MainActivity">

    <EditText

        android:id="@+id/editText"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginLeft="16dp"
        android:layout_marginTop="16dp"
        android:ems="10"
        android:hint="@string/edit_message"
        android:inputType="textPersonName"

        app:layout_constraintEnd_toStartOf="@+id/button"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button

        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:layout_marginStart="16dp"
        android:text="@string/button_send"
        app:layout_constraintBaseline_toBaselineOf="@+id/editText"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"

        app:layout_constraintStart_toEndOf="@+id/editText" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Activities and Intents :-

An [Activity](#) represents a single screen in your app with which your user can perform a single, focused task such as taking a photo, sending an email, or viewing a map. An activity is usually presented to the user as a full-screen window.

An app usually consists of multiple screens that are loosely bound to each other. Each screen is an activity. Typically, one activity in an app is specified as the "main" activity (MainActivity.java), which is presented to the user when the app is launched. The main activity can then start other activities to perform different actions.

Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, that new activity is pushed onto the back stack and takes user focus. The back stack follows basic "last in, first out" stack logic. When the user is done with the current activity and presses the Back button, that activity is popped from the stack and destroyed, and the previous activity resumes.

An activity is started or activated with an *intent*. An [Intent](#) is an asynchronous message that you can use in your activity to request an action from another activity, or from some other app component. You use an intent to start one activity from another activity, and to pass data between activities.

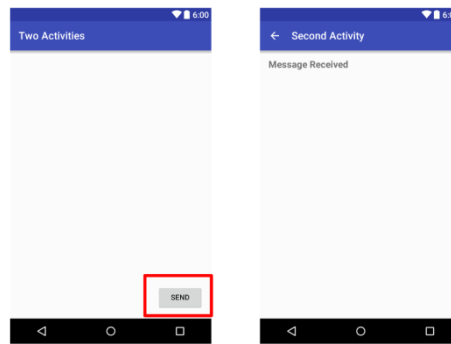
An Intent can be *explicit* or *implicit*:

- An *explicit intent* is one in which you know the target of that intent. That is, you already know the fully qualified class name of that specific activity.
- An *implicit intent* is one in which you do not have the name of the target component, but you have a general action to perform.

2. App overview

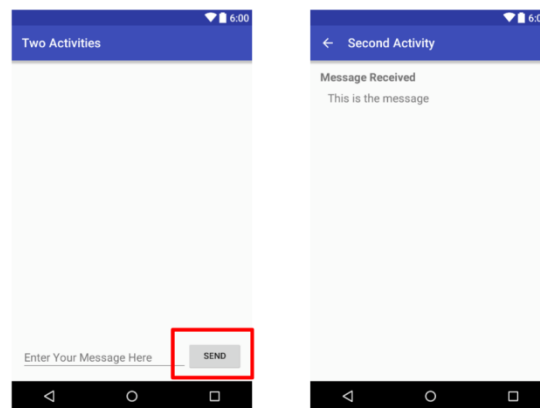
In this chapter you create and build an app called Two Activities that, unsurprisingly, contains two [Activity](#) implementations. You build the app in three stages.

In the first stage, you create an app whose main activity contains one button, **Send**. When the user clicks this button, your main activity uses an intent to start the second activity.



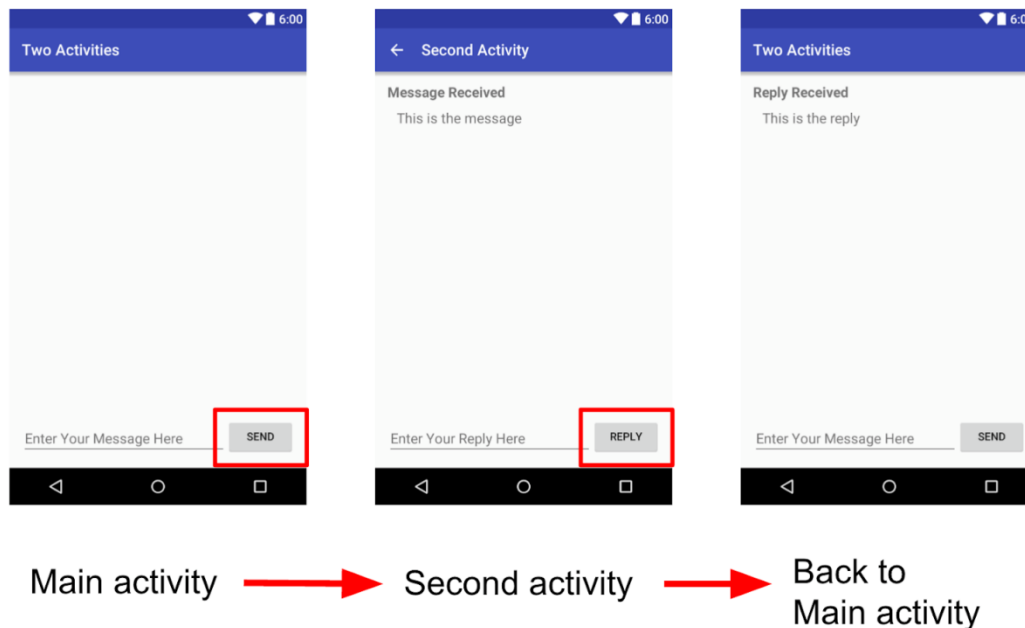
Main activity → Second activity

In the second stage, you add an `EditText` view to the main activity. The user enters a message and clicks **Send**. The main activity uses an intent to start the second activity and send the user's message to the second activity. The second activity displays the message it received.



Main activity → Second activity

In the final stage of creating the Two Activities app, you add an `EditText` and a **Reply** button to the second activity. The user can now type a reply message and tap **Reply**, and the reply is displayed on the main activity. At this point, you use an intent to pass the reply back from the second activity to the main activity.



3. Task 1: Create the TwoActivities project

In this task you set up the initial project with a main Activity, define the layout, and define a skeleton method for the `onClick` button event.

1.1 Create the TwoActivities project

1. Start Android Studio and create a new Android Studio project.

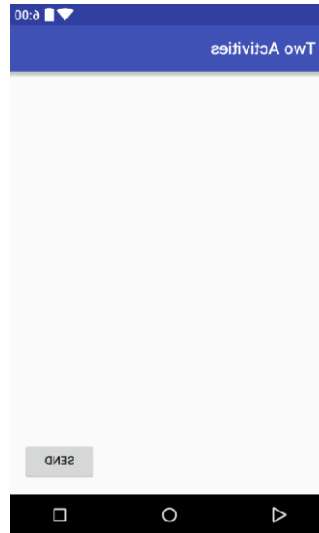
Name your app **Two Activities** and choose the same **Phone and Tablet** settings that you used in previous practicals. The project folder is automatically named `TwoActivities`, and the app name that appears in the app bar will be "Two Activities".

2. Choose **Empty Activity** for the Activity template. Click **Next**.
3. Accept the default Activity name (`MainActivity`). Make sure the **Generate Layout file** and **Backwards Compatibility (AppCompat)** options are checked.
4. Click **Finish**.

1.2 Define the layout for the main Activity

1. Open **res > layout > activity_main.xml** in the **Project > Android** pane. The layout editor appears.
2. Click the **Design** tab if it is not already selected, and delete the `TextView` (the one that says "Hello World") in the **Component Tree** pane.
3. With Autoconnect turned on (the default setting), drag a `Button` from the **Palette** pane to the lower right corner of the layout. Autoconnect creates constraints for the `Button`.

4. In the **Attributes** pane, set the **ID** to **button_main**, the **layout_width** and **layout_height** to **wrap_content**, and enter **Send** for the Text field. The layout should now look like this:



5. Click the **Text** tab to edit the XML code. Add the following attribute to the **Button**:

```
android:onClick="launchSecondActivity"
```

The attribute value is underlined in red because the `launchSecondActivity()` method has not yet been created. Ignore this error for now; you fix it in the next task.

6. Extract the string resource, as described in a previous practical, for "Send" and use the name `button_main` for the resource.

The XML code for the **Button** should look like the following:

```
<Button
    android:id="@+id/button_main"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginRight="16dp"
    android:text="@string/button_main"
    android:onClick="launchSecondActivity"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent" />
```

1.3 Define the Button action

In this task you implement the `launchSecondActivity()` method you referred to in the layout for the `android:onClick` attribute.

1. Click on **"launchSecondActivity"** in the `activity_main.xml` XML code.
2. Press **Alt+Enter** (Option+Enter on a Mac) and select **Create 'launchSecondActivity(View)' in 'MainActivity'**.

The MainActivity file opens, and Android Studio generates a skeleton method for the launchSecondActivity() handler.

3. Inside launchSecondActivity(), add a Log statement that says "Button Clicked!"

```
Log.d(LOG_TAG, "Button clicked!");
```

LOG_TAG will show as red. You add the definition for that variable in a later step.

4. At the top of the MainActivity class, add a constant for the LOG_TAG variable:

```
private static final String LOG_TAG =  
    MainActivity.class.getSimpleName();
```

This constant uses the name of the class itself as the tag.

5. Run your app. When you click the **Send** button you see the "Button Clicked!" message in the **Logcat** pane. If there's too much output in the monitor, type **MainActivity** into the search box, and the **Logcat** pane will only show lines that match that tag.

The code for MainActivity should look as follows:

```
package com.example.android.twoactivities;  
  
import android.support.v7.app.AppCompatActivity;  
import android.os.Bundle;  
import android.util.Log;  
import android.view.View;  
  
public class MainActivity extends AppCompatActivity {  
    private static final String LOG_TAG =  
        MainActivity.class.getSimpleName();  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
  
    public void launchSecondActivity(View view) {  
        Log.d(LOG_TAG, "Button clicked!");  
    }  
}
```


4. Task 2: Create and launch the second Activity

Each new activity you add to your project has its own layout and Java files, separate from those of the main activity. They also have their own `<activity>` elements in the `AndroidManifest.xml` file. As with the main activity, new activity implementations that you create in Android Studio also extend from the `AppCompatActivity` class.

Each activity in your app is only loosely connected with other activities. However, you can define an activity as a parent of another activity in the `AndroidManifest.xml` file. This parent-child relationship enables Android to add navigation hints such as left-facing arrows in the title bar for each activity.

An activity communicates with other activities (in the same app and across different apps) with an intent. An `Intent` can be *explicit* or *implicit*:

- An *explicit intent* is one in which you know the target of that intent; that is, you already know the fully qualified class name of that specific activity.
- An *implicit intent* is one in which you do not have the name of the target component, but have a general action to perform.

In this task you add a second activity to our app, with its own layout. You modify the `AndroidManifest.xml` file to define the main activity as the parent of the second activity. Then you modify the `launchSecondActivity()` method in `MainActivity` to include an intent that launches the second activity when you click the button.

2.1 Create the second Activity

1. Click the **app** folder for your project and choose **File > New > Activity > Empty Activity**.
2. Name the new Activity **SecondActivity**. Make sure **Generate Layout File** and **Backwards Compatibility (AppCompat)** are checked. The layout name is filled in as `activity_second`. Do *not* check the **Launcher Activity** option.
3. Click **Finish**. Android Studio adds both a new Activity layout (`activity_second.xml`) and a new Java file (`SecondActivity.java`) to your project for the new Activity. It also updates the `AndroidManifest.xml` file to include the new Activity.

2.2 Modify the AndroidManifest.xml file

1. Open **manifests > AndroidManifest.xml**.
2. Find the `<activity>` element that Android Studio created for the second Activity.

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

3. Replace the entire `<activity>` element with the following:

```

<activity android:name=".SecondActivity"
    android:label = "Second Activity"
    android:parentActivityName=".MainActivity">
    <meta-data
        android:name="android.support.PARENT_ACTIVITY"
        android:value=
            "com.example.android.twoactivities.MainActivity" />
</activity>

```

The `label` attribute adds the title of the `Activity` to the app bar.

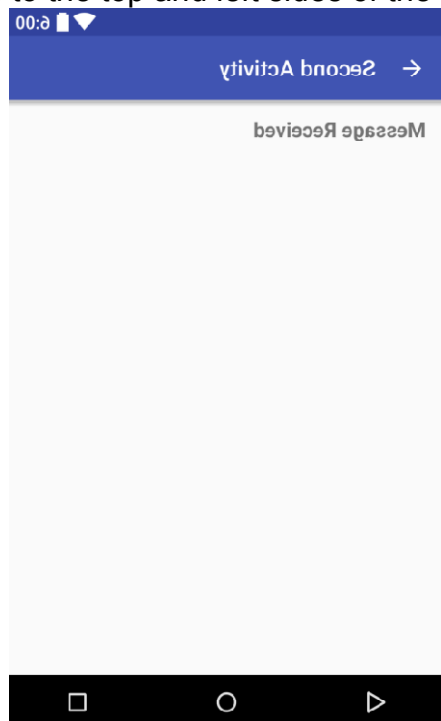
With the `parentActivityName` attribute, you indicate that the main activity is the parent of the second activity. This relationship is used for Up navigation in your app: the app bar for the second activity will have a left-facing arrow so the user can navigate "upward" to the main activity.

With the `<meta-data>` element, you provide additional arbitrary information about the activity in the form of key-value pairs. In this case the metadata attributes do the same thing as the `android:parentActivityName` attribute—they define a relationship between two activities for upward navigation. These metadata attributes are required for older versions of Android, because the `android:parentActivityName` attribute is only available for API levels 16 and higher.

4. Extract a string resource for "Second Activity" in the code above, and use `activity2_name` as the resource name.

2.3 Define the layout for the second Activity

1. Open `activity_second.xml` and click the **Design** tab if it is not already selected.
2. Drag a **TextView** from the **Palette** pane to the top left corner of the layout, and add constraints to the top and left sides of the layout. Set its attributes in the **Attributes** pane



as follows:

3. Click the **Text** tab to edit the XML code, and extract the "Message Received" string into a resource named `text_header`.

4. Add the `android:layout_marginLeft="8dp"` attribute to the `TextView` to complement the `layout_marginStart` attribute for older versions of Android.

The XML code for `activity_second.xml` should be as follows:

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.android.twoactivities.SecondActivity">

<TextView
    android:id="@+id/text_header"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginLeft="8dp"
    android:layout_marginTop="16dp"
    android:text="@string/text_header"
    android:textAppearance=
        "@style/TextAppearance.AppCompat.Medium"
    android:textStyle="bold"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
```

2.4 Add an Intent to the main Activity

In this task you add an explicit `Intent` to the main `Activity`. This `Intent` is used to activate the second `Activity` when the **Send** button is clicked.

1. Open **MainActivity**.
2. Create a new `Intent` in the `launchSecondActivity()` method.

The `Intent` constructor takes two arguments for an explicit `Intent`: an application `Context` and the specific component that will receive that `Intent`. Here you should use `this` as the `Context`, and `SecondActivity.class` as the specific class:

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

3. Call the `startActivity()` method with the new `Intent` as the argument.

```
startActivity(intent);
```

4. Run the app.

When you click the **Send** button, `MainActivity` sends the `Intent` and the Android system launches `SecondActivity`, which appears on the screen. To return to `MainActivity`, click the **Up** button (the left arrow in the app bar) or the Back button at the bottom of the screen.

5. Task 3: Send data from the main Activity to the second Activity

In the last task, you added an explicit intent to `MainActivity` that launched `SecondActivity`. You can also use an intent to *send data* from one activity to another while launching it.

Your intent object can pass data to the target activity in two ways: in the data field, or in the intent *extras*. The intent data is a URI indicating the specific data to be acted on. If the information you want to pass to an activity through an intent is not a URI, or you have more than one piece of information you want to send, you can put that additional information into the *extras* instead.

The intent *extras* are key/value pairs in a [Bundle](#). A `Bundle` is a collection of data, stored as key/value pairs. To pass information from one activity to another, you put keys and values into the intent extra `Bundle` from the sending activity, and then get them back out again in the receiving activity.

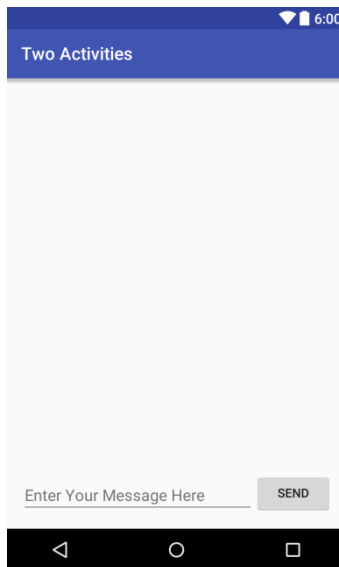
In this task, you modify the explicit intent in `MainActivity` to include additional data (in this case, a user-entered string) in the intent extra `Bundle`. You then modify `SecondActivity` to get that data back out of the intent extra `Bundle` and display it on the screen.

3.1 Add an EditText to the MainActivity layout

1. Open `activity_main.xml`.
2. Drag a **Plain Text** (`EditText`) element from the **Palette** pane to the bottom of the layout, and add constraints to the left side of the layout, the bottom of the layout, and the left side of the **Send Button**. Set its attributes in the **Attributes** pane as follows:

Attribute	Value
id	editText_main
Right margin	8
Left margin	8
Bottom margin	16
layout_width	match_constraint
layout_height	wrap_content
inputType	textLongMessage
hint	Enter Your Message Here
text	(Delete any text in this field)

The new layout in `activity_main.xml` looks like this:



3. Click the **Text** tab to edit the XML code, and extract the "Enter Your Message Here" string into a resource named `editText_main`.

The XML code for the layout should look something like the following.

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.android.twoactivities.MainActivity">

<Button
    android:id="@+id/button_main"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginRight="16dp"
    android:text="@string/button_main"
    android:onClick="launchSecondActivity"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintRight_toRightOf="parent" />

<EditText
    android:id="@+id/editText_main"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="16dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:ems="10"
    android:hint="@string/editText_main"
    android:inputType="textLongMessage"
    app:layout_constraintBottom_toBottomOf="parent"
```

```
app:layout_constraintEnd_toStartOf="@+id/button_main"
app:layout_constraintStart_toStartOf="parent" />
</android.support.constraint.ConstraintLayout>
```

3.2 Add a string to the Intent extras

The Intent *extras* are key/value pairs in a [Bundle](#). A Bundle is a collection of data, stored as key/value pairs. To pass information from one Activity to another, you put keys and values into the Intent extra Bundle from the sending Activity, and then get them back out again in the receiving Activity.

1. Open **MainActivity**.
2. Add a public constant at the top of the class to define the key for the Intent extra:

```
public static final String EXTRA_MESSAGE =
    "com.example.android.twoactivities.extra.MESSAGE";
```

3. Add a private variable at the top of the class to hold the EditText:

```
private EditText mMessageEditText;
```

4. In the onCreate() method, use [findViewById\(\)](#) to get a reference to the EditText and assign it to that private variable:

```
mMessageEditText = findViewById(R.id.editText_main);
```

5. In the launchSecondActivity() method, just under the new Intent, get the text from the EditText as a string:

```
String message = mMessageEditText.getText().toString();
```

6. Add that string to the Intent as an extra with the EXTRA_MESSAGE constant as the key and the string as the value:

```
intent.putExtra(EXTRA_MESSAGE, message);
```

The onCreate() method in MainActivity should now look like the following:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    mMessageEditText = findViewById(R.id.editText_main);
}
```

The launchSecondActivity() method in MainActivity should now look like the following:

```
public void launchSecondActivity(View view) {
    Log.d(LOG_TAG, "Button clicked!");
    Intent intent = new Intent(this, SecondActivity.class);
    String message = mMessageEditText.getText().toString();
    intent.putExtra(EXTRA_MESSAGE, message);
    startActivity(intent);
}
```

3.3 Add a TextView to SecondActivity for the message

1. Open **activity_second.xml**.
2. Drag another **TextView** to the layout underneath the **text_header** TextView, and add constraints to the left side of the layout and to the bottom of **text_header**.
3. Set the new **TextView** attributes in the **Attributes** pane as follows:

Attribute	Value
id	text_message
Top margin	8
Left margin	8
layout_width	wrap_content
layout_height	wrap_content
text	(Delete any text in this field)
textAppearance	AppCompat.Medium

The new layout looks the same as it did in the previous task, because the new **TextView** does not (yet) contain any text, and thus does not appear on the screen.

The XML code for the **activity_second.xml** layout should look something like the following:

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.android.twoactivities.SecondActivity">

<TextView
    android:id="@+id/text_header"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:text="@string/text_header"
    android:textAppearance=
        "@style/TextAppearance.AppCompat.Medium"
    android:textStyle="bold"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```

<TextView
    android:id="@+id/text_message"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/text_header" />
</android.support.constraint.ConstraintLayout>

```

3.4 Modify SecondActivity to get the extras and display the message

1. Open **SecondActivity** to add code to the `onCreate()` method.
2. Get the `Intent` that activated this `Activity`:

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

3. Get the string containing the message from the `Intent` extras using the `MainActivity.EXTRA_MESSAGE` static variable as the key:

```
String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
```

4. Use `findViewById()` to get a reference to the `TextView` for the message from the layout:

```
TextView textView = findViewById(R.id.text_message);
```

5. Set the text of the `TextView` to the string from the `Intent` extra:

```
textView.setText(message);
```

6. Run the app. When you type a message in `MainActivity` and click **Send**, `SecondActivity` launches and displays the message.

The `SecondActivity` `onCreate()` method should look as follows:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_second);
    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
    TextView textView = findViewById(R.id.text_message);
    textView.setText(message);
}

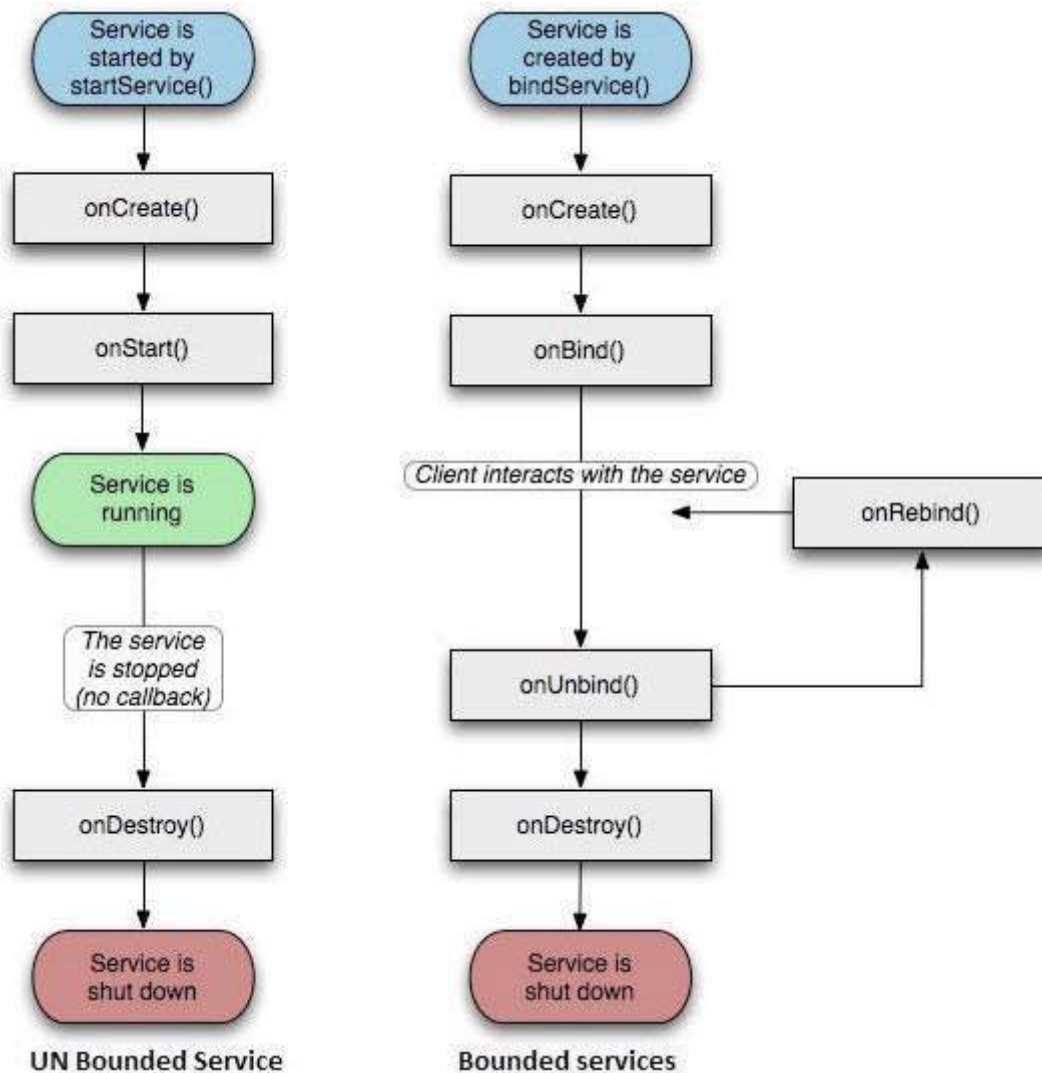
```


Android - Services

A **service** is a component that runs in the background to perform long-running operations without needing to interact with the user and it works even if application is destroyed. A service can essentially take two states –

Sr.No.	State & Description
1	Started A service is started when an application component, such as an activity, starts it by calling <i>startService()</i> . Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.
2	Bound A service is bound when an application component binds to it by calling <i>bindService()</i> . A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).

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



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

Sr.No.	Callback & Description
1	onStartCommand() The system calls this method when another component, such as an activity, requests that the service be started, by calling <code>startService()</code> . If you implement this method, it is your responsibility to stop the service when its work is done, by calling <code>stopSelf()</code> or <code>stopService()</code> methods.
2	onBind() The system calls this method when another component wants to bind with the service by calling <code>bindService()</code> . If you implement this method, you must provide an interface that clients use to

	communicate with the service, by returning an <i>IBinder</i> object. You must always implement this method, but if you don't want to allow binding, then you should return <i>null</i> .
3	onUnbind() The system calls this method when all clients have disconnected from a particular interface published by the service.
4	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 <i>onUnbind(Intent)</i> .
5	onCreate() The system calls this method when the service is first created using <i>onStartCommand()</i> or <i>onBind()</i> . This call is required to perform one-time set-up.
6	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.

The following skeleton service demonstrates each of the life cycle methods –

```
package com.tutorialspoint;

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

public class HelloService extends Service {

    /** indicates how to behave if the service is killed */
    int mStartMode;

    /** interface for clients that bind */
    IBinder mBinder;

    /** indicates whether onRebind should be used */
    boolean mAllowRebind;

    /** Called when the service is being created. */
    @Override
    public void onCreate() {

    }

}
```

```

/** The service is starting, due to a call to startService() */
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    return mStartMode;
}

/** A client is binding to the service with bindService() */
@Override
public IBinder onBind(Intent intent) {
    return mBinder;
}

/** Called when all clients have unbound with unbindService() */
@Override
public boolean onUnbind(Intent intent) {
    return mAllowRebind;
}

/** Called when a client is binding to the service with bindService() */
@Override
public void onRebind(Intent intent) {
}

/** Called when The service is no longer used and is being destroyed */
@Override
public void onDestroy() {
}
}

```

Example

This example will take you through simple steps to show how to create your own Android Service. Follow the following steps to modify the Android application we created in *Hello World Example* chapter –

Step	Description
1	You will use Android Studio IDE to create an Android application and name it as <i>My Application</i> under a package <i>com.example.tutorialspoint7.myapplication</i> as explained in the <i>Hello World Example</i> chapter.

2	Modify main activity file <i>MainActivity.java</i> to add <i>startService()</i> and <i>stopService()</i> methods.
3	Create a new java file <i>MyService.java</i> under the package <i>com.example.My Application</i> . This file will have implementation of Android service related methods.
4	Define your service in <i>AndroidManifest.xml</i> file using <code><service.../></code> tag. An application can have one or more services without any restrictions.
5	Modify the default content of <i>res/layout/activity_main.xml</i> file to include two buttons in linear layout.
6	No need to change any constants in <i>res/values/strings.xml</i> file. Android studio take care of string values
7	Run the application to launch Android emulator and verify the result of the changes done in the application.

Following is the content of the modified main activity file **MainActivity.java**. This file can include each of the fundamental life cycle methods. We have added *startService()* and *stopService()* methods to start and stop the service.

```
package com.example.tutorialspoint7.myapplication;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

import android.os.Bundle;
import android.app.Activity;
import android.util.Log;
import android.view.View;

public class MainActivity extends Activity {
    String msg = "Android : ";

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(msg, "The onCreate() event");
    }

    public void startService(View view) {
        startService(new Intent(getApplicationContext(), MyService.class));
    }

    // Method to stop the service
    public void stopService(View view) {
        stopService(new Intent(getApplicationContext(), MyService.class));
    }
}
```

Following is the content of **MyService.java**. This file can have implementation of one or more methods associated with Service based on requirements. For now we are going to implement only two methods *onStartCommand()* and *onDestroy()* –

```
package com.example.tutorialspoint7.myapplication;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.support.annotation.Nullable;
import android.widget.Toast;

/**
 * Created by Tutorialspoint7 on 8/23/2016.
 */

public class MyService extends Service {
    @Nullable
```

```

@Override
public IBinder onBind(Intent intent) {
    return null;
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    // Let it continue running until it is stopped.
    Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
    return START_STICKY;
}

@Override
public void onDestroy() {
    super.onDestroy();
    Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
}
}

```

Following will be the modified content of *AndroidManifest.xml* file. Here we have added `<service.../>` tag to include our service –

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.tutorialspoint7.myapplication">

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

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

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service android:name=".MyService" />
    </application>

</manifest>

```

Following will be the content of *res/layout/activity_main.xml* file to include two buttons –

```

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"

```

```
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">
```

```
<TextView
```

```
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Example of services"
    android:layout_alignParentTop="true"
    android:layout_centerHorizontal="true"
    android:textSize="30dp" />
```

```
<TextView
```

```
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tutorials point "
    android:textColor="#ff87ff09"
    android:textSize="30dp"
    android:layout_above="@+id/imageButton"
    android:layout_centerHorizontal="true"
    android:layout_marginBottom="40dp" />
```

```
<ImageButton
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imageButton"
    android:src="@drawable/abc"
    android:layout_centerVertical="true"
    android:layout_centerHorizontal="true" />
```


```
<Button
```

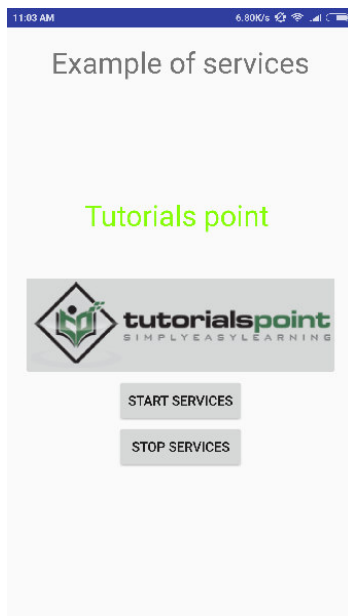
```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/button2"
    android:text="Start Services"
    android:onClick="startService"
    android:layout_below="@+id/imageButton"
    android:layout_centerHorizontal="true" />
```

```
<Button
```

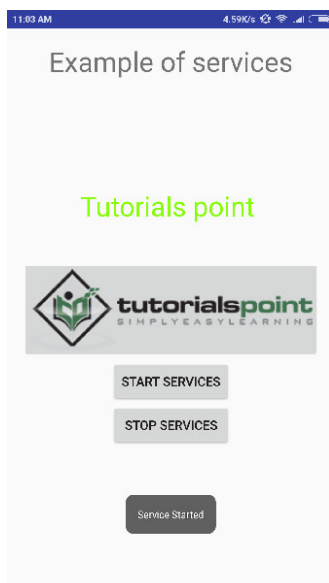
```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Stop Services"
    android:id="@+id/button"
    android:onClick="stopService"
    android:layout_below="@+id/button2"
    android:layout_alignLeft="@+id/button2"
    android:layout_alignStart="@+id/button2"
    android:layout_alignRight="@+id/button2"
    android:layout_alignEnd="@+id/button2" />
```

```
</RelativeLayout>
```


Let's try to run our modified **Hello World!** application we just modified. I assume you had created your **AVD** while doing environment setup. To run the app from Android studio, open one of your project's activity files and click Run  icon from the tool bar. Android Studio installs the app on your AVD and starts it and if everything is fine with your set-up and application, it will display following Emulator window –



Now to start your service, let's click on **Start Service** button, this will start the service and as per our programming in *onStartCommand()* method, a message *Service Started* will appear on the bottom of the the simulator as follows –



To stop the service, you can click the Stop Service button.

Threads

When an application is launched, the system creates a thread of execution for the application, called "main." This thread is very important because it is in charge of dispatching events to the appropriate user interface widgets, including drawing events. It is also almost always the thread in which your application interacts with components from the Android UI toolkit (components from the [android.widget](#) and [android.view](#) packages). As such, the main thread is also sometimes called the UI thread. However, under special circumstances, an app's main thread might not be its UI thread; for more information, see [Thread annotations](#).

The system does *not* create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks (such as [onKeyDown\(\)](#) to report user actions or a lifecycle callback method) always run in the UI thread of the process.

For instance, when the user touches a button on the screen, your app's UI thread dispatches the touch event to the widget, which in turn sets its pressed state and posts an invalidate request to the event queue. The UI thread dequeues the request and notifies the widget that it should redraw itself.

When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly. Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI. When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang. Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "[application not responding](#)" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy.

Additionally, the Android UI toolkit is *not* thread-safe. So, you must not manipulate your UI from a worker thread—you must do all manipulation to your user interface from the UI thread. Thus, there are simply two rules to Android's single thread model:

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

Worker threads

Because of the single threaded model described above, it's vital to the responsiveness of your application's UI that you do not block the UI thread. If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

However, note that you cannot update the UI from any thread other than the UI thread or the "main" thread.

The Application Main Thread

When an Android application is first started, the runtime system creates a single thread in which all application components will run by default. This thread is generally referred to as the main thread. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components that are started within the application will, by default, also run on the main thread.

Any component within an application that performs a time consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This will typically result in the operating system displaying an "Application is unresponsive" warning to the user. Clearly, this is far from the desired behavior for any application. In such a situation, this can be avoided simply by launching the task to be performed in a separate thread, allowing the main thread to continue unhindered with other tasks.

Thread Handlers

Clearly one of the key rules of application development is never to perform time-consuming operations on the main thread of an application. The second, equally important rule is that the code within a separate thread must never, under any circumstances, directly update any aspect of the user interface. Any changes to the user interface must always be performed from within the main thread. The reason for this is that the Android UI toolkit is not thread-safe. Attempts to work with non thread-safe code from within multiple threads will typically result in intermittent problems and unpredictable application behavior.

In the event that the code executing in a thread needs to interact with the user interface, it must do so by synchronizing with the main UI thread. This is achieved by creating a handler within the main thread, which, in turn, receives messages from another thread and updates the user interface accordingly.

REFERENCES

1. G. Blake Meike, Zigurd Mednieks, John Lombardo, Rick Rogers, "Android Application Development", O'reilly, 1st Edition, 2009.
2. R. Nageswara Rao, "Core JAVA: An Integrated Approach", Dreamtech Press, Wiley India, 1st Edition, 2015.
3. Herbert Schildt, "Java: The Complete Reference", 9th Edition, 2014.
4. Gary Cornell, Cay S. Horstmann, "Core Java Volume I - Fundamentals", Prentice Hall, 9th Edition, 2012.
5. Cay S. Horstmann, "Core Java, Volume II - Advanced Features", Prentice Hall, 11th Edition, 2019. <https://www.besanttechnologies.com/what-is-ios>
6. https://www3.ntu.edu.sg/home/ehchua/programming/android/Android_BasicsUI.html
7. <https://developer.android.com/guide/components/activities/intro-activities>
8. <https://livebook.manning.com/book/android-in-action-second-edition/chapter-3/>
9. <https://www.developer.com/languages/xml/understanding-user-interface-layout-and-ui-components-for-android-apps/>

UNIT - 3

ANDROID INTENT, THREAD AND SERVICES

PART - A

1. Define intent objects and discuss about it.
2. List the classification of intents in android platform?
3. Categorize various services in Android platform?
4. How to declare a service in manifest.xml file/
5. Explain how extras object of intent shall be useful?

PART – B

1. What is intent, fields? List & explain the types of intents?
2. What is implicit intent? List and explain the fields used in implicit intents?
3. Discuss in detail about various intent objects?
4. Discuss in detail how threads are handled in Android programming?
5. Explain in detail how Android services concepts implemented in Android ?



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF EEE

DEPARTMENT OF ECE

**UNIT – IV - RECEIVERS AND MULTIMEDIA TECHNIQUES IN ANDROID –
SECA5205**

UNIT 4 RECEIVERS AND MULTIMEDIA TECHNIQUES IN ANDROID

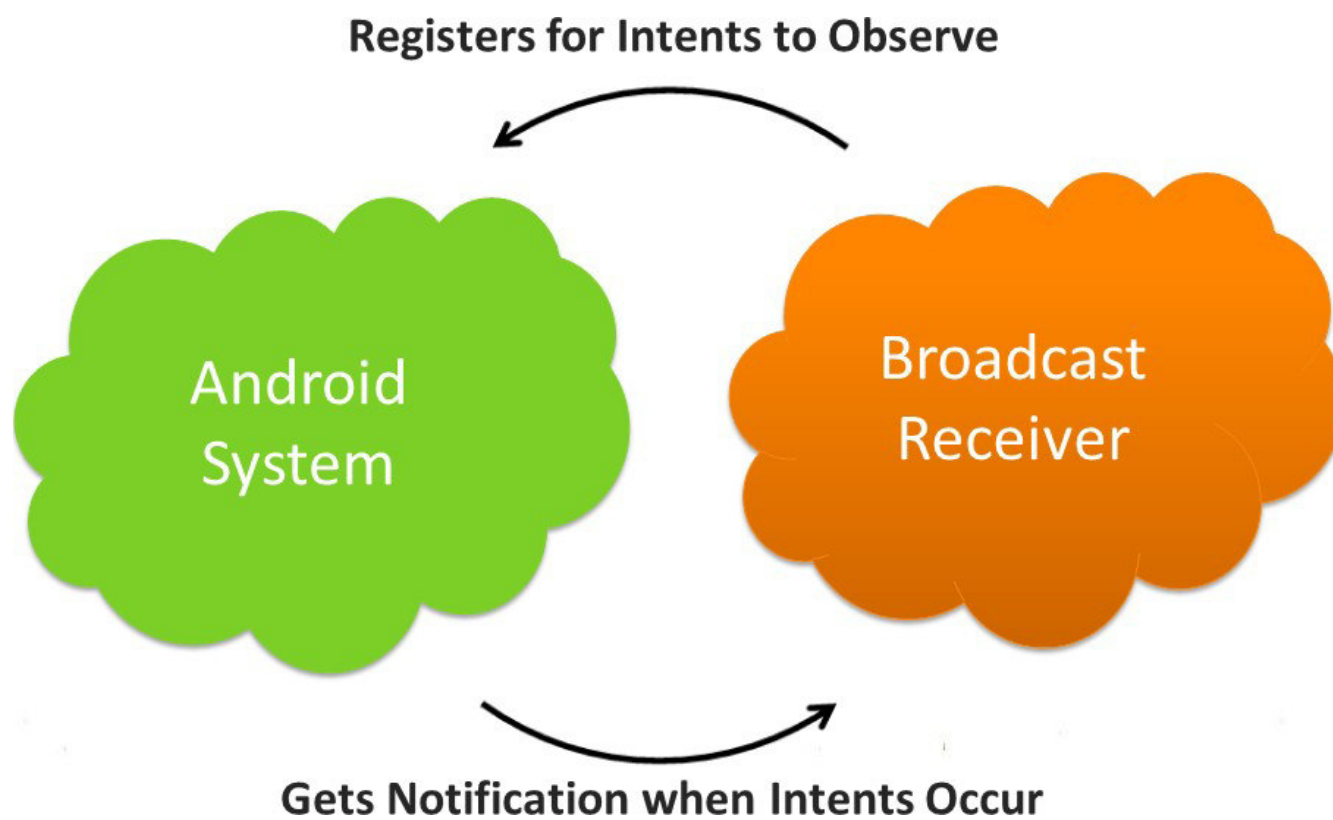
Receiver and Alerts, User Interface Layout, User Interface Events, Multimedia Techniques, Hardware Interfaces.

Android Broadcast Receiver

What is Android Broadcast Receiver?

A broadcast receiver is a dormant component of the Android system. Only an Intent (for which it is registered) can bring it into action. The Broadcast Receiver's job is to pass a notification to the user, in case a specific event occurs.

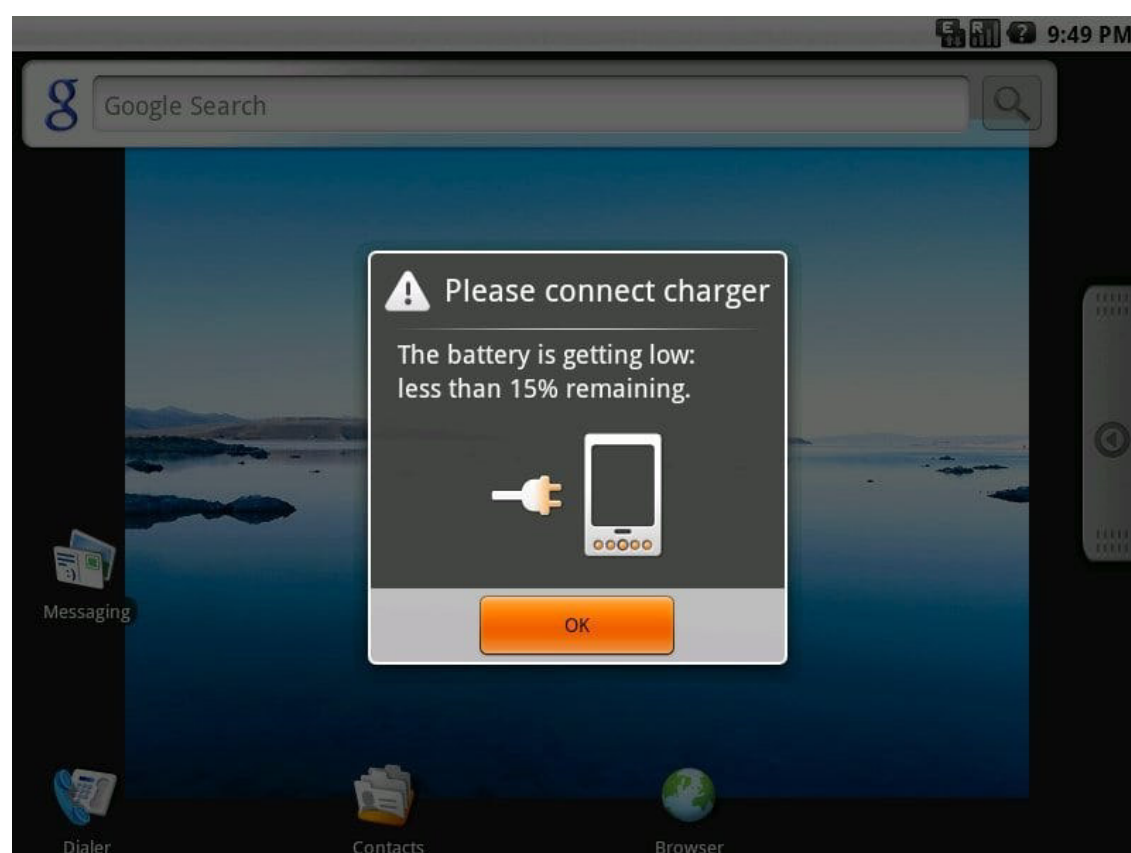
Using a Broadcast Receiver, applications can register for a particular event. Once the event occurs, the system will notify all the registered applications.



For instance, a Broadcast receiver triggers *battery Low notification that you see on your mobile screen.*

Other instances caused by a Broadcast Receiver are *new friend notifications, new friend feeds, new message etc. on your Facebook app.*

In fact, you see broadcast receivers at work all the time. Notifications like *incoming messages, WiFi Activated/Deactivated message etc. are all real-time announcements of what is happening in the Android system and the applications.*



How important is it to implement Broadcast Receivers correctly?

If you wish to create a good Android application, this is of utmost importance. If the broadcast events do not perform their job (of sending notifications to support the application's primary task) perfectly, the application would not be intuitive and user friendly.

Registration of Broadcast Receiver

There are two ways to register a Broadcast Receiver; one is Static and the other Dynamic.

- 1) Static: Use <receiver> tag in your Manifest file. (AndroidManifest.xml)
- 2) Dynamic: Use Context.registerReceiver () method to dynamically register an instance.

Classes of Broadcasts

The two major classes of broadcasts are:

- 1) Ordered Broadcasts: **These broadcasts are synchronous, and therefore follow a specific order. The order is defined using android: priority attribute. The receivers with greater priority would receive the broadcast first. In case there are receivers with same priority levels, the broadcast would not follow an order. Each receiver (when it receives the broadcast) can either pass on the notification to the next one, or abort the broadcast completely. On abort, the notification would not be passed on to the receivers next in line.**
- 2) Normal Broadcasts: **Normal broadcasts are not orderly. Therefore, the registered receivers often run all at the same time. This is very efficient, but the Receivers are unable to utilize the results.**

Sometimes to avoid system overload, the system delivers the broadcasts one at a time, even in case of normal broadcasts. However, the receivers still cannot use the results.

Difference between Activity Intent and Broadcasting Intent

You must remember that Broadcasting Intents are different from the Intents used to start an Activity or a Service (discussed in previous Android Tutorials). The intent used to start an Activity makes changes to an operation the user is interacting with, so the user is aware of the process. However, in case of broadcasting intent, the operation runs completely in the background, and is therefore invisible to the user.

Implementing the Broadcast Receiver

You need to follow these steps to implement a broadcast receiver:

- 1) Create a subclass of Android's BroadcastReceiver
- 2) Implement the onReceive() method: **In order for the notification to be sent, an onReceive() method has to be implemented. Whenever the event for which the receiver is registered occurs, onReceive() is called. For instance, in case of battery low notification, the receiver is registered to Intent.ACTION_BATTERY_LOW event. As soon as the battery level falls below the defined level, this onReceive() method is called.**

Following are the two arguments of the onReceive() method:

- ♦ Context: **This is used to access additional information, or to start services or activities.**
- ♦ Intent: **The Intent object is used to register the receiver.**

Security

As the broadcast receivers have a global work-space, security is very important concern here. If you do not define the limitations and filters for the registered receivers, other applications can abuse them.

Here are a few limitations that might help:

- ♦ **Whenever you publish a receiver in your application's manifest, make it unavailable to external applications by using `android: exported="false"`. You might think that specifying Intent filters while publishing the receiver would do the task for you, when in reality they are not enough.**
- ♦ **When you send a broadcast, it is possible for the external applications too to receive them. This can be prevented by specifying a few limitations.**
- ♦ **Similarly, when you register your receiver using registerReceiver, any application may send it broadcasts. This can be prevented using permissions as well.**

(PS: As of Android 3.1, the Android system will not receive any external Intent, so the system is comparatively secure now.)

Prolonged Operations

The Broadcast Receiver object is active only for the duration of onReceive (Context, Intent).

Therefore, if you need to allow an action after receiving the notification services should be triggered, and not broadcast receivers.

- ♦ **To show a dialogue, then you should use NotificationManager API**
- ♦ **If you wish to send a broadcast intent that would stick around even after the broadcast is complete, you must use sendStickyBroadcast (Intent) method.**

Broadcast Receiver Example

In this sample application, a notification is generated when you change the system time. The notification when clicked leads the user to the Contacts. This is how the application works:

```
1 public class MyBroadcastReceiver extends BroadcastReceiver {
```

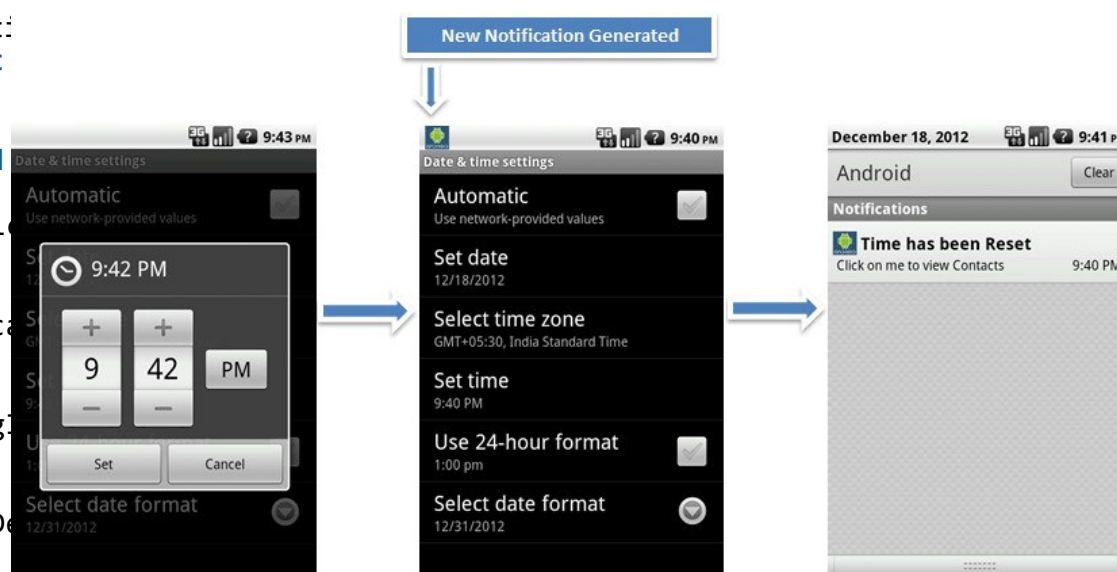
```
2  
3 private NotificationManager mNotif;  
4 private int SIMPLE_NOTIFICATION_ID;
```

```
5  
6 @Override  
7 public void onReceive(Context context, Intent intent) {
```

```
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```

Sample code

```
1 notifyDetails.flags |= Notification.FLAG_AUTO_CANCEL;  
2 notifyDetails.flags |= Notification.DEFAULT_SOUND;  
3  
4 mNotificationManager.notify(SIMPLE_NOTIFICATION_ID, notifyDetails);  
5 Log.i("hisham_debug", "Sucessfully Changed Time");  
6  
7 }  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```

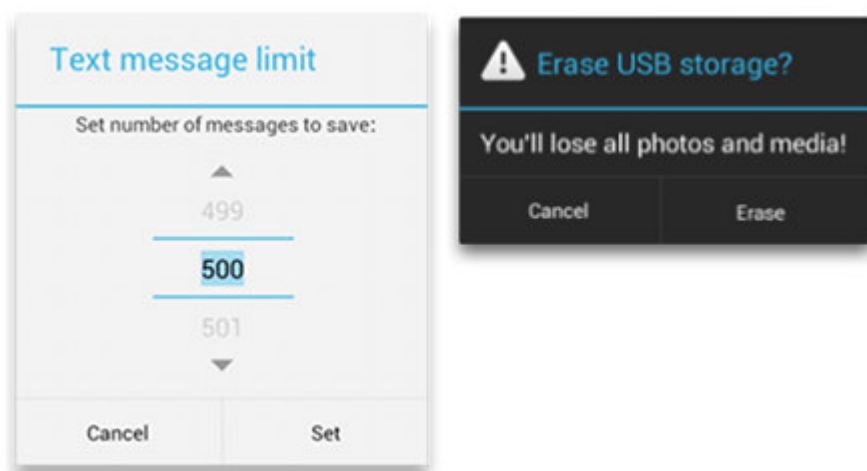


Dialogs

A dialog is a small window that prompts the user to make a decision or enter additional information. A dialog does not fill the screen and is normally used for modal events that require users to take an action before they can proceed.

Dialog Design

For information about how to design your dialogs, including recommendations for language, read the [Dialogs](#) design guide.



The [Dialog](#) class is the base class for dialogs, but you should avoid instantiating [Dialog](#) directly. Instead, use one of the following subclasses:

[AlertDialog](#)

A dialog that can show a title, up to three buttons, a list of selectable items, or a custom layout.

[DatePickerDialog](#) or [TimePickerDialog](#)

A dialog with a pre-defined UI that allows the user to select a date or time.

Caution: Android includes another dialog class called [ProgressDialog](#) that shows a dialog with a progress bar. This widget is deprecated because it prevents users from interacting with the app while progress is being displayed. If you need to indicate loading or indeterminate progress, you should follow the design guidelines for [Progress & Activity](#) and use a [ProgressBar](#) in your layout, instead of using [ProgressDialog](#).

These classes define the style and structure for your dialog, but you should use a [DialogFragment](#) as a container for your dialog. The [DialogFragment](#) class provides all the controls you need to create your dialog and manage its appearance, instead of calling methods on the [Dialog](#) object.

Using [DialogFragment](#) to manage the dialog ensures that it correctly handles lifecycle events such as when the user presses the *Back* button or rotates the screen. The [DialogFragment](#) class also allows you to reuse the dialog's UI as an embeddable component in a larger UI, just like a traditional [Fragment](#) (such as when you want the dialog UI to appear differently on large and small screens).

The following sections in this guide describe how to use a [DialogFragment](#) in combination with an [AlertDialog](#) object. If you'd like to create a date or time picker, you should instead read the [Pickers](#) guide.

Note: Because the [DialogFragment](#) class was originally added with Android 3.0 (API level 11), this document describes how to use the [DialogFragment](#) class that's provided with the [Support Library](#). By adding this library to your app, you can use [DialogFragment](#) and a variety of other APIs on devices running Android 1.6 or higher. If the minimum version your app supports is API level 11 or higher, then you can use the framework version of [DialogFragment](#), but be aware that the links in this document are for the support library APIs. When using the support library, be sure that you import `android.support.v4.app.DialogFragment` class and *not* `android.app.DialogFragment`.

You can accomplish a wide variety of dialog designs—including custom layouts and those described in the [Dialogs](#) design guide—by extending [DialogFragment](#) and creating a [AlertDialog](#) in the [onCreateDialog\(\)](#) callback method.

For example, here's a basic [AlertDialog](#) that's managed within a [DialogFragment](#):

KOTLINJAVA

```
class FireMissilesDialogFragment : DialogFragment() {

    override fun onCreateDialog(savedInstanceState: Bundle): Dialog {
        return activity?.let {
            // Use the Builder class for convenient dialog construction
            val builder = AlertDialog.Builder(it)
            builder.setMessage(R.string.dialog_fire_missiles)
                .setPositiveButton(R.string.fire,
                    DialogInterface.OnClickListener { dialog, id ->
                        // FIRE ZE MISSILES!
                    })
                .setNegativeButton(R.string.cancel,
                    DialogInterface.OnClickListener { dialog, id ->
                        // User cancelled the dialog
                    })

            // Create the AlertDialog object and return it
            builder.create()
        } ?: throw IllegalStateException("Activity cannot be null")
    }
}
```

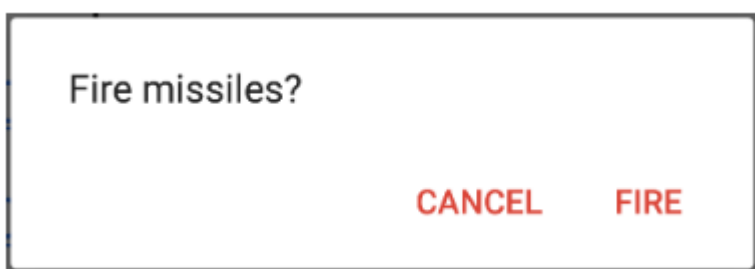


Figure 1. A dialog with a message and two action buttons.

Now, when you create an instance of this class and call [show\(\)](#) on that object, the dialog appears as shown in figure 1.

The next section describes more about using the [AlertDialog.Builder](#) APIs to create the dialog.

Depending on how complex your dialog is, you can implement a variety of other callback methods in the [DialogFragment](#), including all the basic [fragment lifecycle methods](#).

Building an Alert Dialog

The [AlertDialog](#) class allows you to build a variety of dialog designs and is often the only dialog class you'll need. As shown in figure 2, there are three regions of an alert dialog:

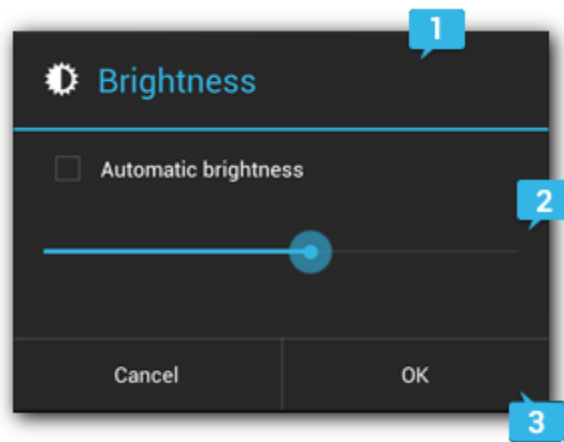


Figure 2. The layout of a dialog.

1. Title

This is optional and should be used only when the content area is occupied by a detailed message, a list, or custom layout. If you need to state a simple message or question (such as the dialog in figure 1), you don't need a title.

2. Content area

This can display a message, a list, or other custom layout.

3. Action buttons

There should be no more than three action buttons in a dialog.

The [AlertDialog.Builder](#) class provides APIs that allow you to create an [AlertDialog](#) with these kinds of content, including a custom layout.

Layouts **Part of [Android Jetpack](#).**

A layout defines the structure for a user interface in your app, such as in an [activity](#). All elements in the layout are built using a hierarchy of [View](#) and [ViewGroup](#) objects. A [View](#) usually draws something the user can see and interact with. Whereas a [ViewGroup](#) is an invisible container that defines the layout structure for [View](#) and other [ViewGroup](#) objects, as shown in figure 1.

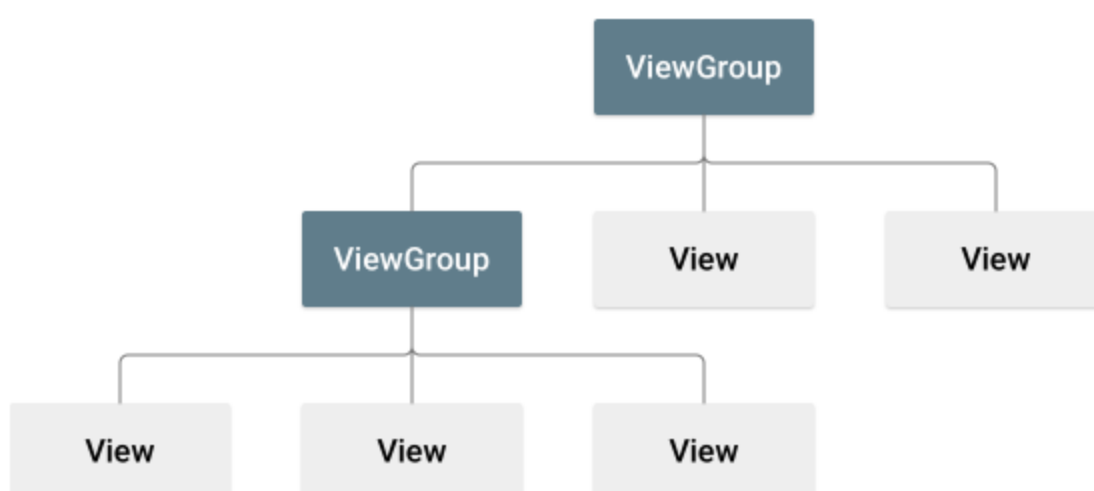


Figure 1. Illustration of a view hierarchy, which defines a UI layout

The [View](#) objects are usually called "widgets" and can be one of many subclasses, such as [Button](#) or [TextView](#). The [ViewGroup](#) objects are usually called "layouts" can be one of many types that provide a different layout structure, such as [LinearLayout](#) or [ConstraintLayout](#).

You can declare a layout in two ways:

- **Declare UI elements in XML.** Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

You can also use Android Studio's [Layout Editor](#) to build your XML layout using a drag-and-drop interface.

- **Instantiate layout elements at runtime.** Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (discussed further in [Supporting Different Screen Sizes](#)).

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

Tip:To debug your layout at runtime, use the [Layout Inspector](#) tool.

Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical [LinearLayout](#) to hold a [TextView](#) and a [Button](#):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

After you've declared your layout in XML, save the file with the `.xml` extension, in your Android project's `res/layout/` directory, so it will properly compile.

More information about the syntax for a layout XML file is available in the [Layout Resources](#) document.

Load the XML Resource

When you compile your app, each XML layout file is compiled into a [View](#) resource. You should load the layout resource from your app code, in your [Activity.onCreate\(\)](#) callback implementation. Do so by calling [setContentView\(\)](#), passing it the reference to your layout resource in the form of: `R.layout.layout_file_name`. For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

KOTLINJAVA

```
fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the [Activities](#) document).

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the `id` attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the [View](#) class) and you will use it very often. The syntax for an ID, inside an XML tag is:

```
android:id="@+id/my_button"
```


The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the `android` package namespace, like so:

```
android:id="@android:id/empty"
```

With the `android` package namespace in place, we're now referencing an ID from the `android.R` resources class, rather than the local resources class.

In order to create views and reference them from the app, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

2. Then create an instance of the view object and capture it from the layout (typically in the `onCreate()` method):

KOTLIN/JAVA

```
val myButton: Button = findViewById(R.id.my_button)
```

Defining IDs for view objects is important when creating a [RelativeLayout](#). In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

Note: With Android Studio 3.6 and higher, the [view binding](#) feature can replace `findViewById()` calls and provides compile-time type safety for code that interacts with views. Consider using view binding instead of `findViewById()`.

Layout Parameters

XML layout attributes named `layout_something` define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends [ViewGroup.LayoutParams](#). This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure 2, the parent view group defines layout parameters for each child view (including the child view group)

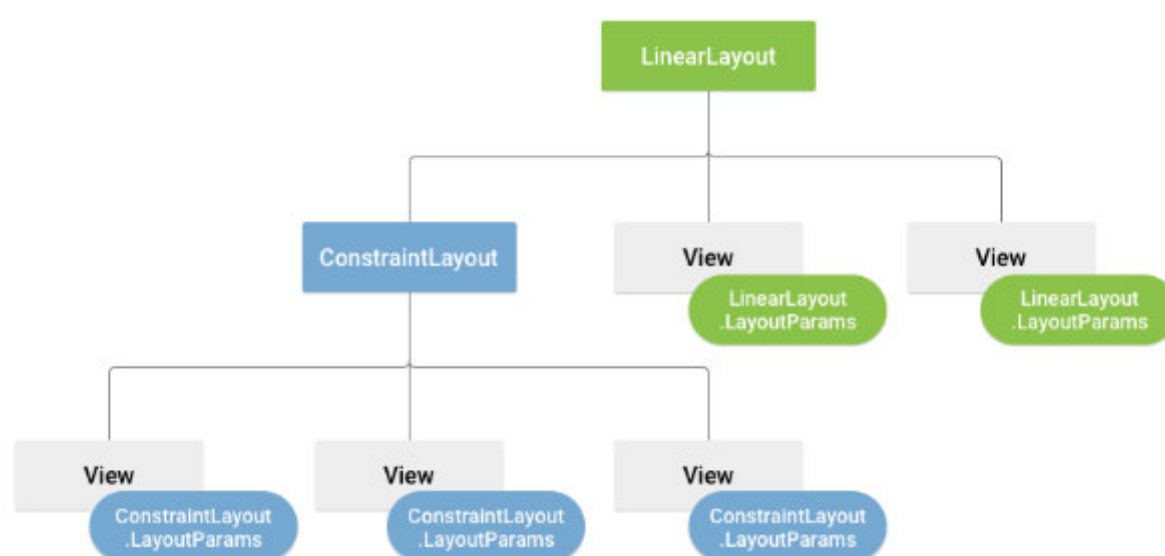


Figure 2. Visualization of a view hierarchy with layout parameters associated with each view

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (`layout_width` and `layout_height`), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

- **`wrap_content`** tells your view to size itself to the dimensions required by its content.

- ***match_parent*** tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (***dp***), ***wrap_content***, or ***match_parent***, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes. The accepted measurement types are defined in the [Available Resources](#) document.

Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of *left* and *top* coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods [getLeft\(\)](#) and [getTop\(\)](#). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when `getLeft()` returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely [getRight\(\)](#) and [getBottom\(\)](#). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling [getRight\(\)](#) is similar to the following computation: `getLeft() + getWidth()`.

Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling [getMeasuredWidth\(\)](#) and [getMeasuredHeight\(\)](#).

The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling [getWidth\(\)](#) and [getHeight\(\)](#).

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the [setPadding\(int, int, int, int\)](#) method and queried by calling [getPaddingLeft\(\)](#), [getPaddingTop\(\)](#), [getPaddingRight\(\)](#) and [getPaddingBottom\(\)](#).

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to [ViewGroup](#) and [ViewGroup.MarginLayoutParams](#) for further information.

For more information about dimensions, see [Dimension Values](#).

Common Layouts

Each subclass of the [ViewGroup](#) class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Note: Although you can nest one or more layouts within another layout to achieve your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

Linear Layout



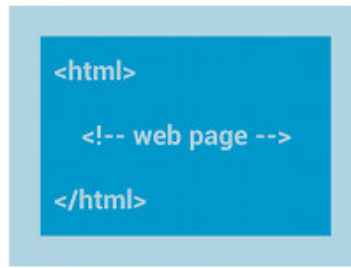
A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the

Relative Layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

Web View



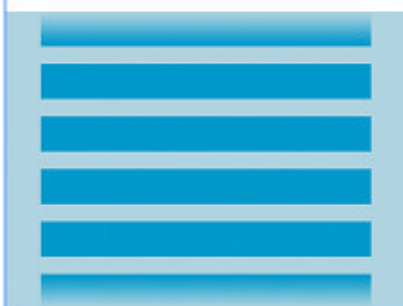
Displays web pages.

Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses [AdapterView](#) to populate the layout with views at runtime. A subclass of the [AdapterView](#) class uses an [Adapter](#) to bind data to its layout. The [Adapter](#) behaves as a middleman between the data source and the [AdapterView](#) layout—the [Adapter](#) retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the [AdapterView](#) layout.

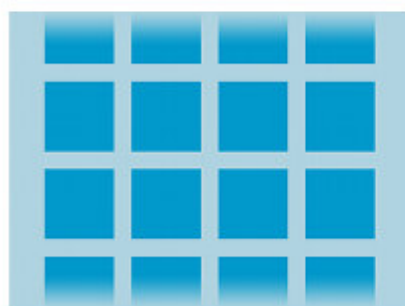
Common layouts backed by an adapter include:

List View



Displays a scrolling single column list.

Grid View



Displays a scrolling grid of columns and rows.

Filling an adapter view with data

You can populate an [AdapterView](#) such as [ListView](#) or [GridView](#) by binding the [AdapterView](#) instance to an [Adapter](#), which retrieves data from an external source and creates a [View](#) that represents each data entry.

Android provides several subclasses of [Adapter](#) that are useful for retrieving different kinds of data and building views for an [AdapterView](#). The two most common adapters are:

[ArrayAdapter](#)

Use this adapter when your data source is an array. By default, [ArrayAdapter](#) creates a view for each array item by calling [toString\(\)](#) on each item and placing the contents in a [TextView](#).

For example, if you have an array of strings you want to display in a [ListView](#), initialize a new [ArrayAdapter](#) using a constructor to specify the layout for each string and the string array:

KOTLINJAVA

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, myStringArray)
```

The arguments for this constructor are:

- Your app [Context](#)
- The layout that contains a [TextView](#) for each string in the array
- The string array

Then simply call [setAdapter\(\)](#) on your [ListView](#):

KOTLINJAVA

```
val listView: ListView = findViewById(R.id.listview)
listView.adapter = adapter
```

To customize the appearance of each item you can override the [toString\(\)](#) method for the objects in your array. Or, to create a view for each item that's something other than a [TextView](#) (for example, if you want an [ImageView](#) for each array item), extend the [ArrayAdapter](#) class and override [getView\(\)](#) to return the type of view you want for each item.

[SimpleCursorAdapter](#)

Use this adapter when your data comes from a [Cursor](#). When using [SimpleCursorAdapter](#), you must specify a layout to use for each row in the [Cursor](#) and which columns in the [Cursor](#) should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a [Cursor](#) containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the [Cursor](#) you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

KOTLINJAVA

```
val fromColumns = arrayOf(ContactsContract.Data.DISPLAY_NAME,
                          ContactsContract.CommonDataKinds.Phone.NUMBER)
val toViews = intArrayOf(R.id.display_name, R.id.phone_number)
```

When you instantiate the [SimpleCursorAdapter](#), pass the layout to use for each result, the [Cursor](#) containing the results, and these two arrays:

KOTLINJAVA

```
val adapter = SimpleCursorAdapter(this,
    R.layout.person_name_and_number, cursor, fromColumns, toViews, 0)
val listView = getListView()
listView.adapter = adapter
```

The [SimpleCursorAdapter](#) then creates a view for each row in the [Cursor](#) using the provided layout by inserting each `fromColumns` item into the corresponding `toViews` view.

.

If, during the course of your app's life, you change the underlying data that is read by your adapter, you should call [notifyDataSetChanged\(\)](#). This will notify the attached view that the data has been changed and it should refresh itself.

Handling click events

You can respond to click events on each item in an [AdapterView](#) by implementing the [AdapterView.OnItemClickListener](#) interface. For example:

KOTLINJAVA

```
listView.setOnItemClickListener = AdapterView.OnItemClickListener { parent, view, position, id ->
    // Do something in response to the click
}
```

Supported media formats

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

The tables below describe the media format support built into the Android platform. Codecs that are not guaranteed to be available on all Android platform versions are noted in parentheses, for example: (Android 3.0+). Note that any given mobile device might support other formats or file types that are not listed in the table.

Audio support

Audio formats and codecs

Format / Codec	Encoder	Decoder	Details	Supported File Type(s) / Container Formats
AAC LC	•	•	Support for mono/stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a) • ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported) • MPEG-TS (.ts, not seekable, Android 3.0+)
HE-AACv1 (AAC+)	• (Android 4.1+)	•		
HE-AACv2 (enhanced AAC+)		•	Support for stereo/5.0/5.1 content with standard sampling rates from 8 to 48 kHz.	
AAC ELD (enhanced low delay AAC)	• (Android 4.1+)	• (Android 4.1+)	Support for mono/stereo content with standard sampling rates from 16 to 48 kHz	
AMR-NB	•	•	4.75 to 12.2 kbps sampled @ 8kHz	3GPP (.3gp)
AMR-WB	•	•	9 rates from 6.60 kbit/s to 23.85 kbit/s sampled @ 16kHz	3GPP (.3gp)

FLAC	• (Android 4.1+)	• (Android 3.1+)	Mono/Stereo (no multichannel). Sample rates up to 48 kHz (but up to 44.1 kHz is recommended on devices with 44.1 kHz output, as the 48 to 44.1 kHz downsampler does not include a low-pass filter). 16-bit recommended; no dither applied for 24-bit.	FLAC (.flac) only
GSM		•	Android supports GSM decoding on telephony devices	GSM(.gsm)
MIDI		•	MIDI Type 0 and 1. DLS Version 1 and 2. XMF and Mobile XMF. Support for ringtone formats RTTTL/RTX, OTA, and iMelody	• Type 0 and 1 (.mid, .xmf, .mxmf) • RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy)
MP3		•	Mono/Stereo 8-320Kbps constant (CBR) or variable bit-rate (VBR)	MP3 (.mp3)
Opus		• (Android 5.0+)		Matroska (.mkv)
PCM/WAVE	• (Android 4.1+)	•	8- and 16-bit linear PCM (rates up to limit of hardware). Sampling rates for raw PCM recordings at 8000, 16000 and 44100 Hz.	WAVE (.wav)
Vorbis		•		• Ogg (.ogg) • Matroska (.mkv, Android 4.0+)

Video support

Video formats and codecs

Format / Codec	Encoder	Decoder	Details	Supported File Type(s) / Container Formats
H.263	•	•	Support for H.263 is optional in Android 7.0+	• 3GPP (.3gp) • MPEG-4 (.mp4)
H.264 AVC Baseline Profile (BP)	• (Android 3.0+)	•		• 3GPP (.3gp) • MPEG-4 (.mp4) • MPEG-TS (.ts, AAC audio only, not seekable, Android 3.0+)
H.264 AVC Main Profile (MP)	• (Android 6.0+)	•	The decoder is required, the encoder is recommended.	
H.265 HEVC		• (Android 5.0+)	Main Profile Level 3 for mobile devices and Main Profile Level 4.1 for Android TV	• MPEG-4 (.mp4)
MPEG-4 SP		•		3GPP (.3gp)
VP8	• (Android 4.3+)	• (Android 2.3.3+)	Streamable only in Android 4.0 and above	• WebM (.webm) • Matroska (.mkv, Android 4.0+)

Video encoding recommendations

The table below lists the Android media framework video encoding profiles and parameters recommended for playback using the H.264 Baseline Profile codec. The same recommendations apply to the Main Profile codec, which is only available in Android 6.0 and later.

	SD (Low quality)	SD (High quality)	HD 720p (N/A on all devices)
Video resolution	176 x 144 px	480 x 360 px	1280 x 720 px
Video frame rate	12 fps	30 fps	30 fps
Video bitrate	56 Kbps	500 Kbps	2 Mbps
Audio codec	AAC-LC	AAC-LC	AAC-LC
Audio channels	1 (mono)	2 (stereo)	2 (stereo)
Audio bitrate	24 Kbps	128 Kbps	192 Kbps

The table below lists the Android media framework video encoding profiles and parameters recommended for playback using the VP8 media codec.

	SD (Low quality)	SD (High quality)	HD 720p (N/A on all devices)	HD 1080p (N/A on all devices)
Video resolution	320 x 180 px	640 x 360 px	1280 x 720 px	1920 x 1080 px
Video frame rate	30 fps	30 fps	30 fps	30 fps
Video bitrate	800 Kbps	2 Mbps	4 Mbps	10 Mbps

Video decoding recommendations

Device implementations must support dynamic video resolution and frame rate switching through the standard Android APIs within the same stream for all VP8, VP9, H.264, and H.265 codecs in real time and up to the maximum resolution supported by each codec on the device.

Implementations that support the Dolby Vision decoder must follow these guidelines:

- Provide a Dolby Vision-capable extractor.
- Properly display Dolby Vision content on the device screen or on a standard video output port (e.g., HDMI).
- Set the track index of backward-compatible base-layer(s) (if present) to be the same as the combined Dolby Vision layer's track index.

Video streaming requirements

For video content that is streamed over HTTP or RTSP, there are additional requirements:

- For 3GPP and MPEG-4 containers, the `moov` atom must precede any `mdat` atoms, but must succeed the `ftyp` atom.
- For 3GPP, MPEG-4, and WebM containers, audio and video samples corresponding to the same time offset may be no more than 500 KB apart. To minimize this audio/video drift, consider interleaving audio and video in smaller chunk sizes.

Image support

Format / Codec	Encoder	Decoder	Details	Supported File Type(s) / Container Formats
BMP		•		BMP (.bmp)
GIF		•		GIF (.gif)
JPEG	•	•	Base+progressive	JPEG (.jpg)
PNG	•	•		PNG (.png)
WebP	• (Android 4.0+) (Transparency, Android 4.2.1+) (Lossless Android 10+)	• (Android 4.0+) (Lossless, Transparency, Android 4.2.1+)	Lossless encoding can be achieved on Android 10 using a quality of 100.	WebP (.webp)
HEIF		• (Android 8.0+)		HEIF (.heic; .heif)

Network protocols

The following network protocols are supported for audio and video playback:

- RTSP (RTP, SDP)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming [draft protocol](#):
 - MPEG-2 TS media files only
 - Protocol version 3 (Android 4.0 and above)
 - Protocol version 2 (Android 3.x)
 - Not supported before Android 3.0

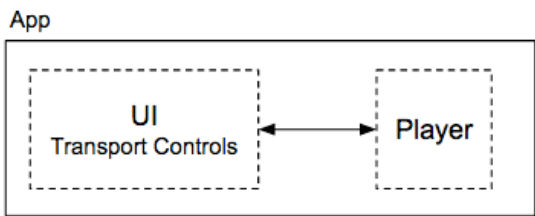
Media app architecture overview

This section explains how to separate a media player app into a media controller (for the UI) and a media session (for the actual player). It describes two media app architectures: a client/server design that works well for audio apps and a single-activity design for video players. It also shows how to make media apps respond to hardware controls and cooperate with other apps that use the audio output stream.

Player and UI

A multimedia application that plays audio or video usually has two parts:

- A player that takes digital media in and renders it as video and/or audio
- A UI with transport controls to run the player and optionally display the player's state



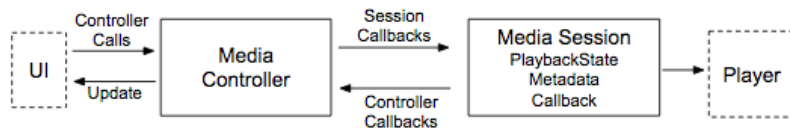
In Android you can build your own player from the ground up, or you can choose from these options:

- The [MediaPlayer](#) class provides the basic functionality for a bare-bones player that supports the most common audio/video formats and data sources.
- [ExoPlayer](#) is an open source library that exposes the lower-level Android audio APIs. ExoPlayer supports high-performance features like DASH and HLS streaming that are not available in `MediaPlayer`. You can customize the ExoPlayer code, making it easy to add new components. ExoPlayer can only be used with Android version 4.1 and higher.

Media session and media controller

While the APIs for the UI and player can be arbitrary, the nature of the interaction between the two pieces is basically the same for all media player apps. The Android framework defines two classes, a *media session* and a *media controller*, that impose a well-defined structure for building a media player app.

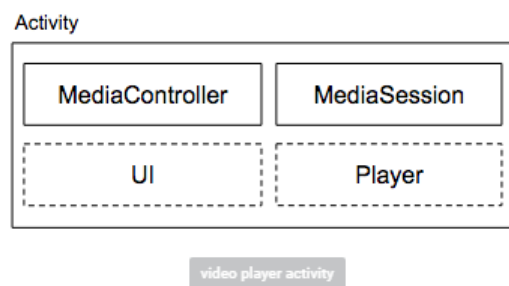
The media session and media controller communicate with each other using predefined callbacks that correspond to standard player actions (play, pause, stop, etc.), as well as extensible custom calls that you use to define special behaviors unique to your app.



Media session

A media session is responsible for all communication with the player. It hides the player's API from the rest of your app. The player is only called from the media session that controls it.

The session maintains a representation of the player's state (playing/paused) and information about what is playing. A session can receive [callbacks](#) from one or more media controllers. This makes it possible for your player to be controlled by your app's UI as well as companion devices running Wear OS and Android Auto. The logic that responds to callbacks must be consistent. The response to a `MediaSession` callback should be the same no matter which client app initiated the callback.



Audio app

An audio player does not always need to have its UI visible. Once it begins to play audio, the player can run as a background task. The user can switch to another app and work while continuing to listen.

To implement this design in Android, you can build an audio app using two components: an activity for the UI and a service for the player. If the user switches to another app, the service can run in the background. By factoring the two parts of an audio app into separate components, each can run more efficiently on its own. A UI is usually short-lived compared to a player, which may run for a long time without a UI.

Media controller

A media controller isolates your UI. Your UI code only communicates with the media controller, not the player itself. The media controller translates transport control actions into callbacks to the media session. It also receives callbacks from the media session whenever the session state changes. This provides a mechanism to automatically update the associated UI. A media controller can only connect to one media session at a time.

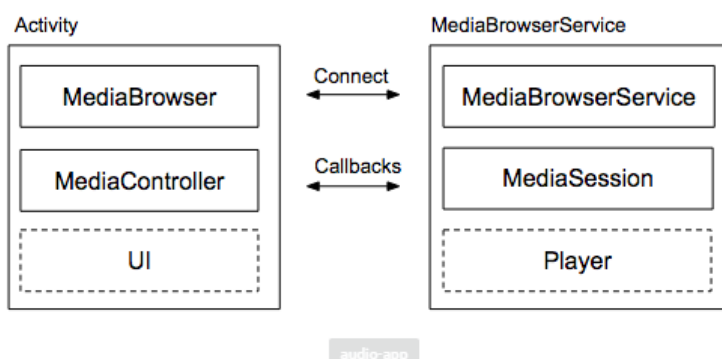
When you use a media controller and a media session, you can deploy different interfaces and/or players at runtime. You can change your app's appearance and/or performance independently depending on the capabilities of the device on which it's running.

Video apps versus audio apps

When playing a video, your eyes and ears are both engaged. When playing audio, you are listening, but you can also work with a different app at the same time. There's a different design for each use case.

Video app

A video app needs a window for viewing content. For this reason a video app is usually implemented as a single Android activity. The screen on which the video appears is part of the activity.

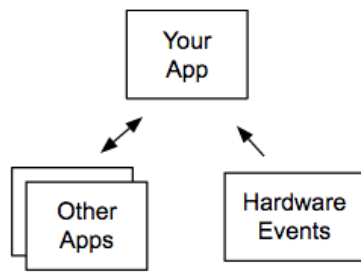


The support library provides two classes to implement this client/server approach: `MediaBrowserService` and `MediaBrowser`. The service component is implemented as a subclass of `MediaBrowserService` containing the media session and its player. The activity with the UI and the media controller should include a `MediaBrowser`, which communicates with the `MediaBrowserService`.

Using `MediaBrowserService` makes it easy for companion devices (like Android Auto and Wear) to discover your app, connect to it, browse for content, and control playback, without accessing your app's UI activity at all. In fact, there can be multiple apps connected to the same `MediaBrowserService` at the same time, each app with its own `MediaController`. An app that offers a `MediaBrowserService` should be able to handle multiple simultaneous connections.

Media apps and the Android audio infrastructure

A well-designed media app should "play well together" with other apps that play audio. It should be prepared to share the phone and cooperate with other apps on your device that use audio. It should also respond to hardware controls on the device.



The media-compat library

The [media-compat](#) library contains classes that are helpful for building apps that play audio and video. These classes are compatible with devices running Android 2.3 (API level 9) and higher. They also work with other Android features to create a comfortable, familiar Android experience.

The recommended implementation of media sessions and media controllers are the classes [MediaSessionCompat](#) and [MediaControllerCompat](#), which are defined in the [media-compat support library](#). They replace earlier versions of the classes `MediaSession` and `MediaController` that were introduced in Android 5.0 (API level 21). The compat classes offer the same functionality but make it easier to develop your app because you only need to write to one API. The library takes care of backward compatibility by translating media session methods to the equivalent methods on older platform versions when available.

If you already have a working app that's using the older classes, we recommend updating to the compat classes. When you use the compat versions you can remove all calls to [registerMediaButtonReceiver\(\)](#) and any methods from [RemoteControlClient](#).

Measuring performance

In Android 8.0 (API level 26) and later, the [getMetrics\(\)](#) method is available for some media classes. It returns a [PersistableBundle](#) object containing configuration and performance information, expressed as a map of attributes and values. The [getMetrics\(\)](#) method is defined for these media classes:

- [MediaPlayer.getMetrics\(\)](#)
- [MediaRecorder.getMetrics\(\)](#)
- [MediaCodec.getMetrics\(\)](#)
- [MediaExtractor.getMetrics\(\)](#)

Metrics are collected separately for each instance and persist for the lifetime of the instance. If no metrics are available the method returns null. The actual metrics returned depend on the class.

android.hardware

Added in API level 1

[Kotlin](#) | [Java](#)

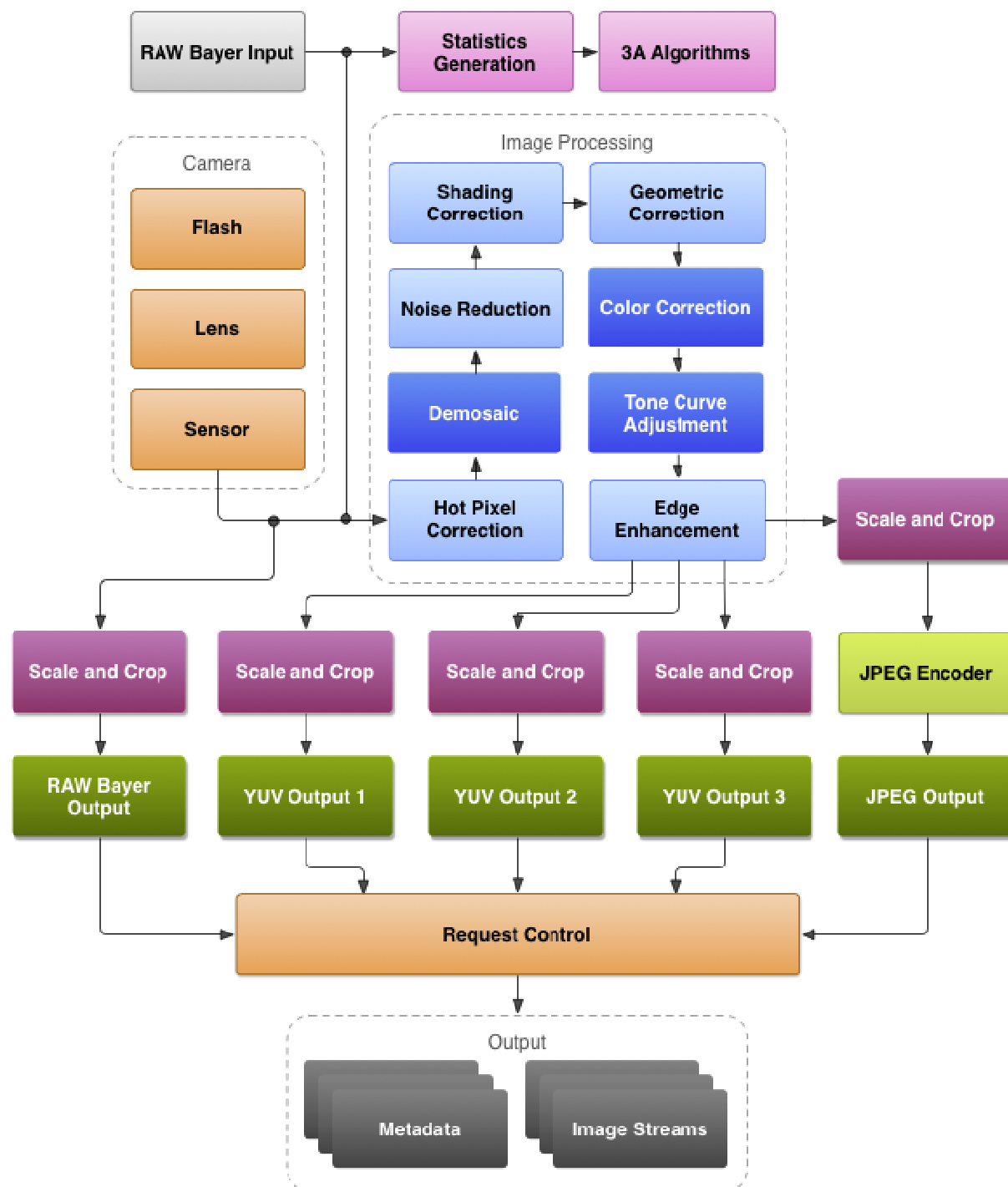
Provides support for hardware features, such as the camera and other sensors. Be aware that not all Android-powered devices support all hardware features, so you should declare hardware that your application requires using the [<uses-feature>](#) manifest element.

Interfaces

Camera.AutoFocusCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.AutoFocusMoveCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.ErrorCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.FaceDetectionListener	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.OnZoomChangeListener	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.PictureCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.PreviewCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
Camera.ShutterCallback	<i>This interface was deprecated in API level 21. We recommend using the new android.hardware.camera2 API for new applications.</i>
SensorEventListener	Used for receiving notifications from the SensorManager when there is new sensor data.
SensorEventListener2	Used for receiving a notification when a flush() has been successfully completed.
SensorListener	<i>This interface was deprecated in API level 3. Use SensorEventListener instead.</i>

GeomagneticField	Estimates magnetic field at a given point on Earth, and in particular, to compute the magnetic declination from true north.
HardwareBuffer	HardwareBuffer wraps a native AHardwareBuffer object, which is a low-level object representing a memory buffer accessible by various hardware units.
Sensor	Class representing a sensor.
SensorAdditionalInfo	This class represents a Sensor additional information frame, which is reported through listener callback onSensorAdditionalInfo .
SensorDirectChannel	Class representing a sensor direct channel.
SensorEvent	This class represents a Sensor event and holds information such as the sensor's type, the time-stamp, accuracy and of course the sensor's SensorEvent#values .
SensorEventCallback	Used for receiving sensor additional information frames.
SensorManager	SensorManager lets you access the device's sensors .
SensorManager.DynamicSensorCallback	Used for receiving notifications from the SensorManager when dynamic sensors are connected or disconnected.
TriggerEvent	This class represents a Trigger Event - the event associated with a Trigger Sensor.
TriggerEventListener	This class is the listener used to handle Trigger Sensors.

ANDROID CAMERA ARCHITECTURE



Android's camera HAL connects the higher level camera framework APIs in [android.hardware](#) to your underlying camera driver and hardware. The figure and list describe the components involved and where to find the source for each:

Application framework

At the application framework level is the app's code, which utilizes the [android.hardware.Camera](#) API to interact with the camera hardware. Internally, this code calls a corresponding JNI glue class to access the native code that interacts with the camera.

JNI

The JNI code associated with [android.hardware.Camera](#) is located in `frameworks/base/core/jni/android_hardware_Camera.cpp`. This code calls the lower level native code to obtain access to the physical camera and returns data that is used to create the [android.hardware.Camera](#) object at the framework level.

Native framework

The native framework defined in `frameworks/av/camera/Camera.cpp` provides a native equivalent to the [android.hardware.Camera](#) class. This class calls the IPC binder proxies to obtain access to the camera service.

Binder IPC proxies

The IPC binder proxies facilitate communication over process boundaries. There are three camera binder classes that are located in the `frameworks/av/camera` directory that calls into camera service. `ICameraService` is the interface to the camera service, `ICamera` is the interface to a specific opened camera device, and `ICameraClient` is the device's interface back to the application framework.

Camera service

The camera service, located in `frameworks/av/services/camera/libcameraservice/CameraService.cpp`, is the actual code that interacts with the HAL.

HAL

The hardware abstraction layer defines the standard interface that the camera service calls into and that you must implement to have your camera hardware function correctly.

Kernel driver

The camera's driver interacts with the actual camera hardware and your implementation of the HAL. The camera and driver must support YV12 and NV21 image formats to provide support for previewing the camera image on the display and video recording.

REFERENCES

1. G. Blake Meike, Zigurd Mednieks, John Lombardo, Rick Rogers, "Android Application Development", O'reilly, 1st Edition, 2009.
2. R. Nageswara Rao, "Core JAVA: An Integrated Approach", Dreamtech Press, Wiley India, 1st Edition, 2015.
3. Herbert Schildt, "Java: The Complete Reference", 9th Edition, 2014.
4. Cay S. Horstmann, "Core Java, Volume II - Advanced Features", Prentice Hall, 11th Edition, 2019. <https://www.besanttechnologies.com/what-is-ios>
5. <https://www.edureka.co/blog/android-tutorials-broadcast-receivers>
6. <https://developer.android.com/guide/components/broadcasts>
7. <https://www.journaldev.com/10356/android-broadcastreceiver-example-tutorial>
8. https://www.techotopia.com/index.php/Broadcast_Intent_and_Broadcast_Receivers_in_Android_Studio

UNIT - 4

RECEIVERS AND MULTIMEDIA TECHNIQUES IN ANDROID

PART – A

1. What do you understand from User Interface ?Discuss its hierarchy.
2. Discuss about various types of Layouts used in Android?
3. Enumerate about absolute layout in Android?
4. List the various multimedia events used in Android?

PART - B

1. Write short note on
 - i) Frame Layout
 - ii) Table Layout
 - iii) Relative Layout
2. Discuss in detail about the multimedia techniques available in Android platform?
3. Write in detail the hardware interfaces procedures and concepts in Android Platform?
4. Discuss in detail the handling of Alert Notification sequences in Android ?



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOL OF EEE

DEPARTMENT OF ECE

UNIT – V - CONNECTIVITY IN ANDROID – SECA5205

UNIT 5 CONNECTIVITY IN ANDROID

Connecting Gyro Sensor and Accelerometer -Wi-Fi and Bluetooth Connectivity with Mobile applications.

The sensor framework, which is a part of the Android SDK, allows you to read raw data from most sensors, be they hardware or software, in an easy and consistent manner. In this tutorial, I'll show you how to use the framework to read data from two very common sensors: proximity and gyroscope. I'll also introduce you to the rotation vector sensor, a composite sensor that can, in most situations, serve as an easier and more accurate alternative to the gyroscope.

Motion sensors

The Android platform provides several sensors that let you monitor the motion of a device.

The sensors' possible architectures vary by sensor type:

- The gravity, linear acceleration, rotation vector, significant motion, step counter, and step detector sensors are either hardware-based or software-based.
- The accelerometer and gyroscope sensors are always hardware-based.

Most Android-powered devices have an accelerometer, and many now include a gyroscope. The availability of the software-based sensors is more variable because they often rely on one or more hardware sensors to derive their data. Depending on the device, these software-based sensors can derive their data either from the accelerometer and magnetometer or from the gyroscope.

Motion sensors are useful for monitoring device movement, such as tilt, shake, rotation, or swing. The movement is usually a reflection of direct user input (for example, a user steering a car in a game or a user controlling a ball in a game), but it can also be a reflection of the physical environment in which the device is sitting (for example, moving with you while you drive your car). In the first case, you are monitoring motion relative to the device's frame of reference or your application's frame of reference; in the second case you are monitoring motion relative to the world's frame of reference. Motion sensors by themselves are not typically used to monitor device position, but they can be used with other sensors, such as the geomagnetic field sensor, to determine a device's position relative to the world's frame of reference (see [Position Sensors](#) for more information).

All of the motion sensors return multi-dimensional arrays of sensor values for each `SensorEvent`. For example, during a single sensor event the accelerometer returns acceleration force data for the three coordinate axes, and the gyroscope returns rate of rotation data for the three coordinate axes. These data values are returned in a `float` array (`values`) along with other `SensorEvent` parameters. Table 1 summarizes the motion sensors that are available on the Android platform.

The rotation vector sensor and the gravity sensor are the most frequently used sensors for motion detection and monitoring. The rotational vector sensor is particularly versatile and can be used for a wide range of motion-related tasks, such as detecting gestures, monitoring angular change, and monitoring relative orientation changes. For example, the rotational vector sensor is ideal if you are developing a game, an augmented reality application, a 2-dimensional or 3-dimensional compass, or a camera stabilization app. In most cases, using these sensors is a better choice than using the accelerometer and geomagnetic field sensor or the orientation sensor.

Android Open Source Project sensors

The Android Open Source Project (AOSP) provides three software-based motion sensors: a gravity sensor, a linear acceleration sensor, and a rotation vector sensor. These sensors were updated in Android 4.0 and now use a device's gyroscope (in addition to other sensors) to improve stability and performance. If you want to try these sensors, you can identify them by using the `getVendor()` method and the `getVersion()` method (the vendor is Google LLC; the version number is 3). Identifying these sensors by vendor and version number is necessary because the Android system considers these three sensors to be secondary sensors. For example, if a device manufacturer provides their own gravity sensor, then the AOSP gravity sensor shows up as a secondary gravity sensor. All three of these sensors rely on a gyroscope: if a device does not have a gyroscope, these sensors do not show up and are not available for use.

Use the gravity sensor

The gravity sensor provides a three dimensional vector indicating the direction and magnitude of gravity. Typically, this sensor is used to determine the device's relative orientation in space. The following code shows you how to get an instance of the default gravity sensor:

KOTLIN	JAVA
<pre>val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager val sensor: Sensor? = sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY)</pre>	

The units are the same as those used by the acceleration sensor (m/s^2), and the coordinate system is the same as the one used by the acceleration sensor.

★ **Note:** When a device is at rest, the output of the gravity sensor should be identical to that of the accelerometer.

Use the linear accelerometer

The linear acceleration sensor provides you with a three-dimensional vector representing acceleration along each device axis, excluding gravity. You can use this value to perform gesture detection. The value can also serve as input to an inertial navigation system, which uses dead reckoning. The following code shows you how to get an instance of the default linear acceleration sensor:

KOTLIN

JAVA

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val sensor: Sensor? = sensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION)
```



Conceptually, this sensor provides you with acceleration data according to the following relationship:

```
linear acceleration = acceleration - acceleration due to gravity
```



You typically use this sensor when you want to obtain acceleration data without the influence of gravity. For example, you could use this sensor to see how fast your car is going. The linear acceleration sensor always has an offset, which you need to remove. The simplest way to do this is to build a calibration step into your application. During calibration you can ask the user to set the device on a table, and then read the offsets for all three axes. You can then subtract that offset from the acceleration sensor's direct readings to get the actual linear acceleration.

The sensor [coordinate system](#) is the same as the one used by the acceleration sensor, as are the units of measure (m/s²).

Use the rotation vector sensor

The rotation vector represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis (x, y, or z). The following code shows you how to get an instance of the default rotation vector sensor:

1. Project Setup

If your app is simply unusable on devices that do not have all the hardware sensors it needs, it should not be installable on such devices. You can let Google Play and other app marketplaces know about your app's hardware requirements by adding one or more `<uses-feature>` tags to your Android Studio project's manifest file.

The app we'll be creating in this tutorial will not work on devices that lack a proximity sensor and a gyroscope. Therefore, add the following lines to your manifest file:

```
1 <uses-feature
2     android:name="android.hardware.sensor.proximity"
3     android:required="true" />
4 <uses-feature
5     android:name="android.hardware.sensor.gyroscope"
6     android:required="true" />
```

Note, however, that because the `<uses-feature>` tag doesn't help if a user installs your app manually using its APK file, you must still programmatically check if a sensor is available before using it.

KOTLIN JAVA

```
val sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
val sensor: Sensor? = sensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR)
```

The three elements of the rotation vector are expressed as follows:

$$\begin{aligned} x \cdot \sin\left(\frac{\theta}{2}\right) \\ y \cdot \sin\left(\frac{\theta}{2}\right) \\ z \cdot \sin\left(\frac{\theta}{2}\right) \end{aligned}$$

Where the magnitude of the rotation vector is equal to $\sin(\theta/2)$, and the direction of the rotation vector is equal to the direction of the axis of rotation.

The three elements of the rotation vector are equal to the last three components of a unit quaternion ($\cos(\theta/2)$, $x \cdot \sin(\theta/2)$, $y \cdot \sin(\theta/2)$, $z \cdot \sin(\theta/2)$). Elements of the rotation vector are unitless. The x, y, and z axes are defined in the same way as the acceleration sensor. The reference coordinate system is defined as a direct orthonormal basis (see figure 1). This coordinate system has the following characteristics:

- X is defined as the vector product $Y \times Z$. It is tangential to the ground at the device's current location and points approximately East.
- Y is tangential to the ground at the device's current location and points toward the geomagnetic North Pole.
- Z points toward the sky and is perpendicular to the ground plane.

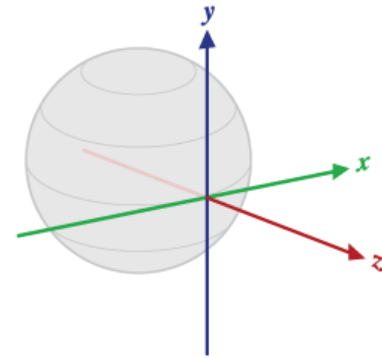


Figure 1. Coordinate system used by the rotation vector sensor.

3. Using the Gyroscope

The gyroscope allows you to determine the angular velocity of an Android device at any given instant. In simpler terms, it tells you how fast the device is rotating around its X, Y, and Z axes. Lately, even budget phones are being manufactured with a gyroscope built in, what with augmented reality and virtual reality apps becoming so popular.

By using the gyroscope, you can develop apps that can respond to minute changes in a device's orientation. To see how, let us now create an activity whose background color changes to blue every time you rotate the phone in the anticlockwise direction along the Z axis, and to yellow otherwise.

Step 1: Acquire the Gyroscope

To create a `Sensor` object for the gyroscope, all you need to do is pass the `TYPE_GYROSCOPE` constant to the `getDefaultSensor()` method of the `SensorManager` object.

```
1 | gyroscopeSensor =  
2 |     sensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```


Step 2: Register a Listener

Creating a listener for the gyroscope sensor is no different from creating one for the proximity sensor. While registering it, however, you must make sure that its sampling frequency is very high. Therefore, instead of specifying a polling interval in microseconds, I suggest you use the `SENSOR_DELAY_NORMAL` constant.

```
01 // Create a listener
02 gyroscopeSensorListener = new SensorEventListener() {
03     @Override
04     public void onSensorChanged(SensorEvent sensorEvent) {
05         // More code goes here
06     }
07
08     @Override
09     public void onAccuracyChanged(Sensor sensor, int i) {
10     }
11 };
12
13 // Register the listener
14 sensorManager.registerListener(gyroscopeSensorListener,
15     gyroscopeSensor, SensorManager.SENSOR_DELAY_NORMAL);
```

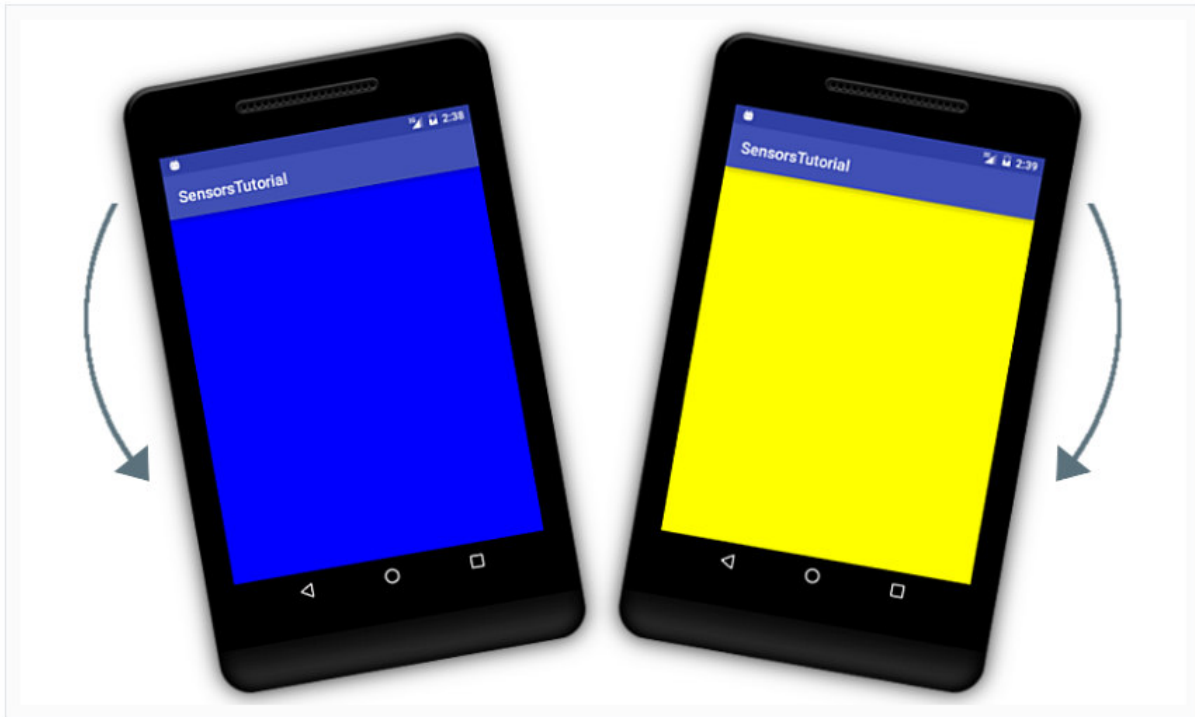
Step 3: Use the Raw Data

The gyroscope sensor's raw data consists of three `float` values, specifying the angular velocity of the device along the X, Y, and Z axes. The unit of each value is radians per second. In case of anticlockwise rotation along any axis, the value associated with that axis will be positive. In case of clockwise rotation, it will be negative.

Because we are currently interested only in rotation along the Z-axis, we'll be working only with the third element in the `values` array of the `SensorEvent` object. If it's more than `0.5f`, we can, to a large extent, be sure that the rotation is anticlockwise, and set the background color to blue. Similarly, if it's less than `-0.5f`, we can set the background color to yellow.

```
1 if(sensorEvent.values[2] > 0.5f) { // anticlockwise
2     getWindow().getDecorView().setBackgroundColor(Color.BLUE);
3 } else if(sensorEvent.values[2] < -0.5f) { // clockwise
4     getWindow().getDecorView().setBackgroundColor(Color.YELLOW);
5 }
```

If you run the app now, hold your phone in the portrait mode, and tilt it to the left, you should see the activity turn blue. If you tilt it in the opposite direction, it should turn yellow.



If you turn the phone too much, however, its screen orientation will change to landscape and your activity will restart. To avoid this condition, I suggest you set the `screenOrientation` of the activity to `portrait` in the manifest file.

```
1 <activity
2     android:name=".GyroscopeActivity"
3     android:screenOrientation="portrait">
4 </activity>
```

Android - Wi-Fi

Android allows applications to access to view the access the state of the wireless connections at very low level. Application can access almost all the information of a wifi connection.

The information that an application can access includes connected network's link speed, IP address, negotiation state, other networks information. Applications can also scan, add, save, terminate and initiate Wi-Fi connections.

Android provides **WifiManager** API to manage all aspects of WIFI connectivity. We can instantiate this class by calling **getSystemService** method. Its syntax is given below –

```
WifiManager mainWifiObj;
mainWifiObj = (WifiManager) getSystemService(Context.WIFI_SERVICE);
```

In order to scan a list of wireless networks, you also need to register your BroadcastReceiver. It can be registered using **registerReceiver** method with argument of your receiver class object. Its syntax is given below –

```
class WifiScanReceiver extends BroadcastReceiver {
    public void onReceive(Context c, Intent intent) {
    }
}

WifiScanReceiver wifiReceiver = new WifiScanReceiver();
registerReceiver(wifiReceiver, new
IntentFilter(WifiManager.SCAN_RESULTS_AVAILABLE_ACTION));
```

The wifi scan can be start by calling the **startScan** method of the WifiManager class. This method returns a list of ScanResult objects. You can access any object by calling the **get** method of list. Its syntax is given below –

```
List<ScanResult> wifiScanList = mainWifiObj.getScanResults();
String data = wifiScanList.get(0).toString();
```

Apart from just Scanning, you can have more control over your WIFI by using the methods defined in WifiManager class. They are listed as follows –

Sr.No	Method & Description
1	addNetwork(WifiConfiguration config) This method add a new network description to the set of configured networks.
2	createWifiLock(String tag) This method creates a new WifiLock.
3	disconnect() This method disassociate from the currently active access point.
4	enableNetwork(int netId, boolean disableOthers) This method allow a previously configured network to be associated with.
5	getWifiState() This method gets the Wi-Fi enabled state
6	isWifiEnabled()

	This method return whether Wi-Fi is enabled or disabled.
7	setWifiEnabled(boolean enabled) This method enable or disable Wi-Fi.
8	updateNetwork(WifiConfiguration config) This method update the network description of an existing configured network.

Example

Here is an example demonstrating the use of WIFI. It creates a basic application that open your wifi and close your wifi

To experiment with this example, you need to run this on an actual device on which wifi is turned on.

Steps	Description
1	You will use Android studio to create an Android application under a package com.example.sairamkrishna.myapplication.
2	Modify src/MainActivity.java file to add WebView code.
3	Modify the res/layout/activity_main to add respective XML components
4	Modify the AndroidManifest.xml to add the necessary permissions
5	Run the application and choose a running android device and install the application on it and verify the results.

Following is the content of the modified main activity file **src/MainActivity.java**.

```
package com.example.sairamkrishna.myapplication;

import android.net.wifi.WifiManager;
import android.os.Bundle;
import android.app.Activity;
import android.content.Context;
```

```

import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MainActivity extends Activity {
    Button enableButton,disableButton;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        enableButton=(Button) findViewById(R.id.button1);
        disableButton=(Button) findViewById(R.id.button2);

        enableButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                WifiManager wifi = (WifiManager)
getSystemService(Context.WIFI_SERVICE);
                wifi.setWifiEnabled(true);
            }
        });

        disableButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                WifiManager wifi = (WifiManager)
getSystemService(Context.WIFI_SERVICE);
                wifi.setWifiEnabled(false);
            }
        });
    }
}

```

Following is the modified content of the xml **res/layout/activity_main.xml**.

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    tools:context=".MainActivity">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"

```

```

        android:id="@+id/imageView"
        android:src="@drawable/abc"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />

<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="76dp"
    android:text="Enable Wifi"
    android:layout_centerVertical="true"
    android:layout_alignEnd="@+id/imageView" />

<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Disable Wifi"
    android:layout_marginBottom="93dp"
    android:layout_alignParentBottom="true"
    android:layout_alignStart="@+id/imageView" />

</RelativeLayout>

```

Following is the content of **AndroidManifest.xml** file.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.sairamkrishna.myapplication" >
    <uses-permission
android:name="android.permission.ACCESS_WIFI_STATE" />
    <uses-permission
android:name="android.permission.CHANGE_WIFI_STATE" />

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

        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >


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

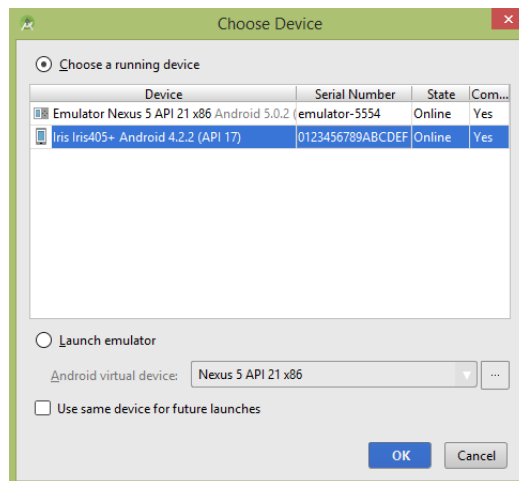
```

```
</intent-filter>

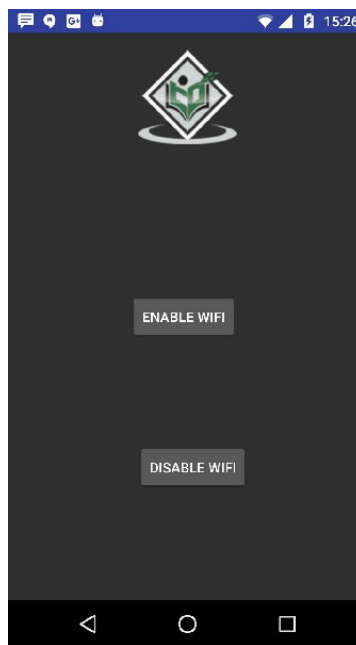
</activity>

</application>
</manifest>
```

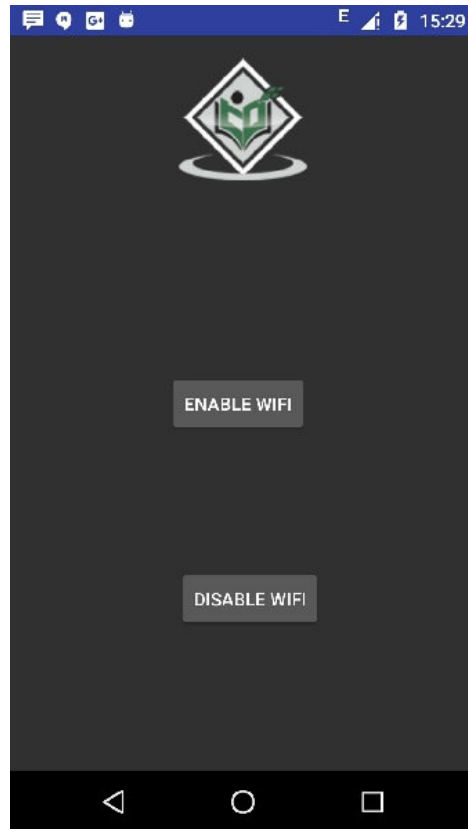
Let's try to run your application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run  icon from the toolbar. Before starting your application, Android studio will display following window to select an option where you want to run your Android application.



Select your mobile device as an option and It will shows the following image–



Now click on disable wifi button.then the sample output should be like this –



Android - Bluetooth

Among many ways, Bluetooth is a way to send or receive data between two different devices. Android platform includes support for the Bluetooth framework that allows a device to wirelessly exchange data with other Bluetooth devices.

Android provides Bluetooth API to perform these different operations.

- Scan for other Bluetooth devices
- Get a list of paired devices
- Connect to other devices through service discovery

Android provides BluetoothAdapter class to communicate with Bluetooth. Create an object of this calling by calling the static method getDefaultAdapter(). Its syntax is given below.

```
private BluetoothAdapter BA;  
BA = BluetoothAdapter.getDefaultAdapter();
```

In order to enable the Bluetooth of your device, call the intent with the following Bluetooth constant ACTION_REQUEST_ENABLE. Its syntax is.

```
Intent turnOn = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
startActivityForResult(turnOn, 0);
```

Apart from this constant, there are other constants provided the API , that supports different tasks. They are listed below.

Sr.No	Constant & description
1	ACTION_REQUEST_DISCOVERABLE This constant is used for turn on discovering of bluetooth
2	ACTION_STATE_CHANGED This constant will notify that Bluetooth state has been changed
3	ACTION_FOUND This constant is used for receiving information about each device that is discovered

Once you enable the Bluetooth , you can get a list of paired devices by calling `getBondedDevices()` method. It returns a set of bluetooth devices. Its syntax is.

```
private Set<BluetoothDevice>pairedDevices;  
pairedDevices = BA.getBondedDevices();
```

Apart from the paired Devices , there are other methods in the API that gives more control over Bluetooth. They are listed below.

Sr.No	Method & description
1	enable() This method enables the adapter if not enabled
2	isEnabled() This method returns true if adapter is enabled
3	disable() This method disables the adapter
4	getName() This method returns the name of the Bluetooth adapter
5	setName(String name) This method changes the Bluetooth name
6	getState() This method returns the current state of the Bluetooth Adapter.
7	startDiscovery() This method starts the discovery process of the Bluetooth for 120 seconds.

Example

This example provides demonstration of BluetoothAdapter class to manipulate Bluetooth and show list of paired devices by the Bluetooth.

To experiment with this example , you need to run this on an actual device.

Steps	Description
1	You will use Android studio to create an Android application a package com.example.sairamkrishna.myapplication.
2	Modify src/MainActivity.java file to add the code
3	Modify layout XML file res/layout/activity_main.xml add any GUI component if required
4	Modify AndroidManifest.xml to add necessary permissions.
5	Run the application and choose a running android device and install the application on it and verify the results.

Here is the content of **src/MainActivity.java**

```
package com.example.sairamkrishna.myapplication;

import android.app.Activity;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;

import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.ListView;
```

```

import android.widget.Toast;
import java.util.ArrayList;
import java.util.Set;

public class MainActivity extends Activity {
    Button b1,b2,b3,b4;
    private BluetoothAdapter BA;
    private Set<BluetoothDevice>pairedDevices;
    ListView lv;

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

        b1 = (Button) findViewById(R.id.button);
        b2=(Button) findViewById(R.id.button2);
        b3=(Button) findViewById(R.id.button3);
        b4=(Button) findViewById(R.id.button4);

        BA = BluetoothAdapter.getDefaultAdapter();
        lv = (ListView) findViewById(R.id.listView);
    }

    public void on(View v){
        if (!BA.isEnabled()) {
            Intent turnOn = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(turnOn, 0);
            Toast.makeText(getApplicationContext(), "Turned
on", Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(getApplicationContext(), "Already on",
Toast.LENGTH_LONG).show();
        }
    }

    public void off(View v){
        BA.disable();
        Toast.makeText(getApplicationContext(), "Turned off"
,Toast.LENGTH_LONG).show();
    }

    public void visible(View v){
        Intent getVisible = new
Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
        startActivityForResult(getVisible, 0);
    }
}

```

```

    }

    public void list(View v){
        pairedDevices = BA.getBondedDevices();

        ArrayList list = new ArrayList();

        for(BluetoothDevice bt : pairedDevices)
list.add(bt.getName());
        Toast.makeText(getApplicationContext(), "Showing Paired
Devices", Toast.LENGTH_SHORT).show();

        final ArrayAdapter adapter = new
ArrayAdapter(this, android.R.layout.simple_list_item_1, list);

        lv.setAdapter(adapter);
    }
}

```

Here is the content of **activity_main.xml**

Here abc indicates about logo of tutorialspoint.

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:paddingLeft="@dimen/activity_horizontal_margin"
android:paddingRight="@dimen/activity_horizontal_margin"
android:paddingTop="@dimen/activity_vertical_margin"
android:paddingBottom="@dimen/activity_vertical_margin"
tools:context=".MainActivity"
android:transitionGroup="true">

    <TextView android:text="Bluetooth Example"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/textview"
        android:textSize="35dp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Tutorials point"

```

```

        android:id="@+id/textView"
        android:layout_below="@+id/textview"
        android:layout_centerHorizontal="true"
        android:textColor="#ff7aff24"
        android:textSize="35dp" />

<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/imageView"
    android:src="@drawable/abc"
    android:layout_below="@+id/textView"
    android:layout_centerHorizontal="true"
    android:theme="@style/Base.TextAppearance.AppCompat" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Turn On"
    android:id="@+id/button"
    android:layout_below="@+id/imageView"
    android:layout_toStartOf="@+id/imageView"
    android:layout_toLeftOf="@+id/imageView"
    android:clickable="true"
    android:onClick="on" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Get visible"
    android:onClick="visible"
    android:id="@+id/button2"
    android:layout_alignBottom="@+id/button"
    android:layout_centerHorizontal="true" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="List devices"
    android:onClick="list"
    android:id="@+id/button3"
    android:layout_below="@+id/imageView"
    android:layout_toRightOf="@+id/imageView"
    android:layout_toEndOf="@+id/imageView" />

<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="turn off"

```

```

        android:onClick="off"
        android:id="@+id/button4"
        android:layout_below="@+id/button"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

<ListView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/listView"
    android:layout_alignParentBottom="true"
    android:layout_alignLeft="@+id/button"
    android:layout_alignStart="@+id/button"
    android:layout_below="@+id/textView2" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Paired devices:"
    android:id="@+id/textView2"
    android:textColor="#ff34ff06"
    android:textSize="25dp"
    android:layout_below="@+id/button4"
    android:layout_alignLeft="@+id/listView"
    android:layout_alignStart="@+id/listView" />

</RelativeLayout>

```

Here is the content of **Strings.xml**

```

<resources>
    <string name="app_name">My Application</string>
</resources>

```

Here is the content of **AndroidManifest.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.sairamkrishna.myapplication" >
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission
        android:name="android.permission.BLUETOOTH_ADMIN"/>

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

```

```


<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >

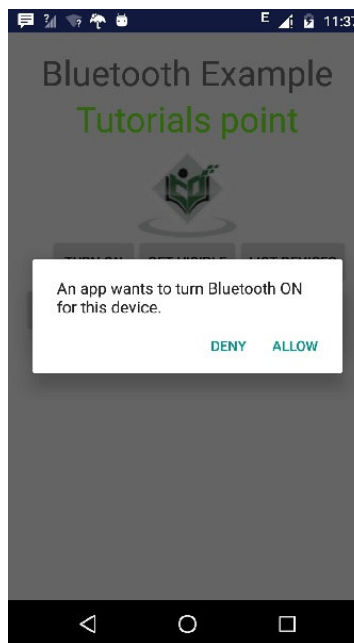
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category
android:name="android.intent.category.LAUNCHER" />
        </intent-filter>

    </activity>

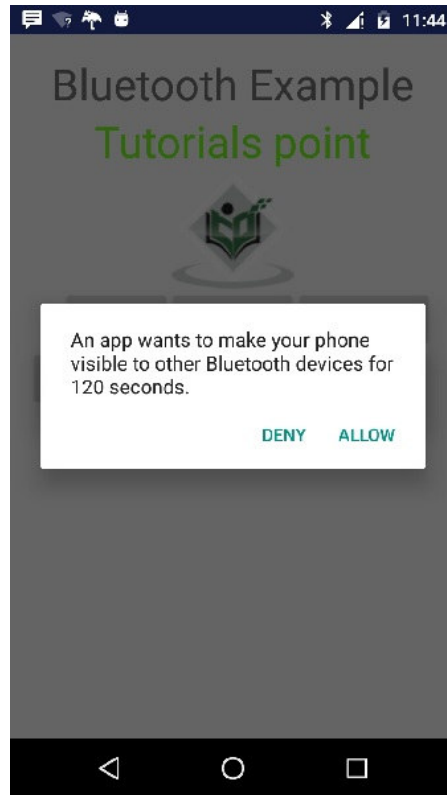
</application>
</manifest>

```

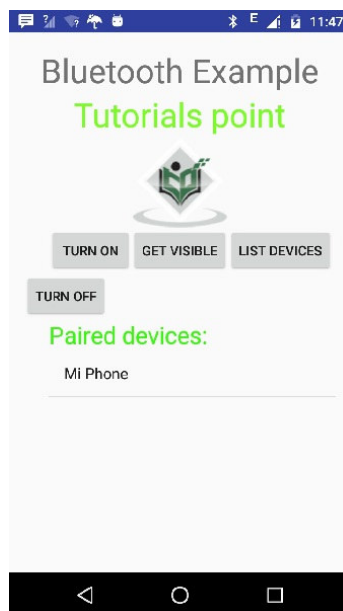
Let's try to run your application. I assume you have connected your actual Android Mobile device with your computer. To run the app from Android studio, open one of your project's activity files and click Run  icon from the tool bar. If your Bluetooth will not be turned on then, it will ask your permission to enable the Bluetooth.



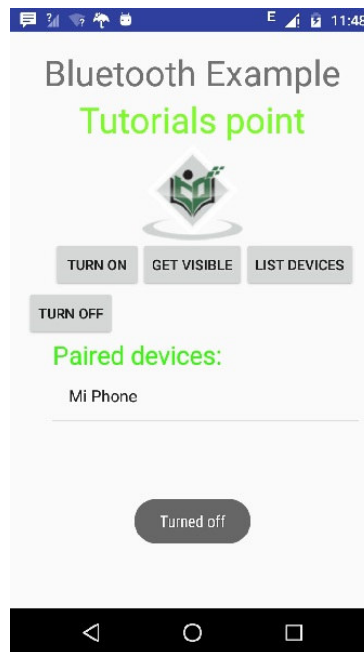
Now just select the Get Visible button to turn on your visibility. The following screen would appear asking your permission to turn on discovery for 120 seconds.



Now just select the List Devices option. It will list down the paired devices in the list view. In my case , I have only one paired device. It is shown below.



Now just select the Turn off button to switch off the Bluetooth. Following message would appear when you switch off the bluetooth indicating the successful switching off of Bluetooth.



REFERENCES

1. G. Blake Meike, Zigurd Mednieks, John Lombardo, Rick Rogers, "Android Application Development", O'reilly, 1st Edition, 2009.
2. Herbert Schildt, "Java: The Complete Reference", 9th Edition, 2014.
3. Gary Cornell, Cay S. Horstmann, "Core Java Volume I - Fundamentals", Prentice Hall, 9th Edition, 2012.
4. Cay S. Horstmann, "Core Java, Volume II - Advanced Features", Prentice Hall, 11th Edition, 2019.<https://www.besanttechnologies.com/what-is-ios>
5. https://developer.android.com/guide/topics/sensors/sensors_overview
6. https://subscription.packtpub.com/book/application_development/9781785285509/1/ch01lvl1sec10/components-of-the-sensor-framework
7. https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/sensors/sensors_overview.html
8. <https://blog.mindorks.com/using-android-sensors-android-tutorial>

UNIT - 5

CONNECTIVITY IN ANDROID

PART –A

1. Classify the various sensors availabilities in android platform?
2. Prepare a notes on the concepts of WIFI device in Android platform?
3. Explain the basic concepts of Bluetooth technology used in Android?
4. Write short note on
 - i) Gyroscope
 - ii) Accelerometer

PART - B

1. Discuss the concepts and procedure for extracting Accelerometer data from Android Mobile devices?
2. Explain the concepts and procedure for extracting Gyroscope data from Android Mobile devices?
3. With a code explain how to establish and communicate Bluetooth connectivity in Android mobile environment?
4. Explain the concepts of implementing WiFi services in Android devices with neat diagram?