Chapter 3

Android Classes and Basie's

3.1 Android Fundamentals

Android apps can be written using Kotlin, Java, and C++ languages. The Android SDK tools compile your code along with any data and resource files into an APK, an Android package, which is an archive file with an .apk suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app. Each Android app lives in its own security sandbox, protected by the following Android security features:

- The Android operating system is a multi-user Linux system in which each app is a different user.

- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.

- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.

- By default, every app runs in its own Linux process. The Android system starts the process when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

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

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

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

App components App components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can enter your app. Some components depend on others.
There are four different types of app components:

- Activities
- Services
- Broadcast receivers
- Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. The following sections describe the four types of app components.

Activities An activity is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app: An activity is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app:

- Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
- Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
- Helping the app handle having its process killed so the user can return to activities with their previous state restored.
- Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

You implement an activity as a subclass of the Activity class.
Services A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. There are actually two very distinct semantics services tell the system about how to manage an app: Started services tell the system to keep them running until their work is completed. This could be to sync some data in the background or play music even after the user leaves the app. Syncing data in the background or playing music also represent two different types of started services that modify how the system handles them:

- Music playback is something the user is directly aware of, so the app tells the system this by saying it wants to be foreground with a notification to tell the user about it; in this case the system knows that it should try really hard to keep that service's process running, because the user will be unhappy if it goes away.

- A regular background service is not something the user is directly aware as running, so the system has more freedom in managing its process. It may allow it to be killed (and then restarting the service sometime later) if it needs RAM for things that are of more immediate concern to the user.

Bound services run because some other app (or the system) has said that it wants to make use of the service. This is basically the service providing an API to another process. The system thus knows there is a dependency between these processes, so if process A is bound to a service in process B, it knows that it needs to keep process B (and its service) running for A. Further, if process A is something the user cares about, then it also knows to treat process B as something the user also cares about. Because of their flexibility (for better or worse), services have turned out to be a really useful building block for all kinds of higher-level system concepts. Live wallpapers, notification listeners, screen savers, input methods, accessibility services, and many other core system features are all built as services that applications implement and the system binds to when they should be running.
A service is implemented as a subclass of Service.

Broadcast receivers A broadcast receiver is a component that enables the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running. So, for example, an app can schedule an alarm to post a notification to tell the user about an upcoming event... and by delivering that alarm to a BroadcastReceiver of the app, there is no need for the app to remain running until the alarm goes off. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Apps can also initiate broadcasts—for example, to let other apps know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a gateway to other components and is intended to do a very minimal amount of work. For instance, it might schedule a JobService to perform some work based on the event with JobScheduler
A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

Content providers A content provider manages a shared set of app data that you can store in the file system, in a SQLite database, on the web, or on any other persistent storage location that your app can access. Through the content provider, other apps can query or modify the data if the content provider allows it. For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query the content provider, such as ContactsContract.Data, to read and write information about a particular person. It is tempting to think of a content provider as an abstraction on a database, because there is a lot of API and support built in to them for that common case. However, they have a different core purpose from a system-design perspective. To the system, a content provider is an entry point into an app for publishing named data items, identified by a URI scheme. Thus an app can decide how it wants to map the data it contains to a URI namespace, handing out those URIs to other entities which can in turn use them to access the data. There are a few particular things this allows the system to do in managing an app:

- Assigning a URI doesn't require that the app remain running, so URIs can persist after their owning apps have exited. The system only needs to make sure that an owning app is still running when it has to retrieve the app's data from the corresponding URI.

- These URIs also provide an important fine-grained security model. For example, an app can place the URI for an image it has on the clipboard, but leave its content provider locked up so that other apps cannot freely access it. When a second app attempts to access that URI on the clipboard,the system can allow that app to access the data via a temporary URI permission grant so that it is allowed to access the data only behind that URI, but nothing else in the second app.

Content providers are also useful for reading and writing data that is private to your app and not shared. For example, the Note Pad sample app uses a content provider to save notes.
A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions

3.1.1 Creating an Android App

Android Application initially creating using Eclipse IDE. With the advance development of Android Studio by JetBreans, Android Studio has get popularity it kills to Eclipse IDE for Android Development. Currently Google

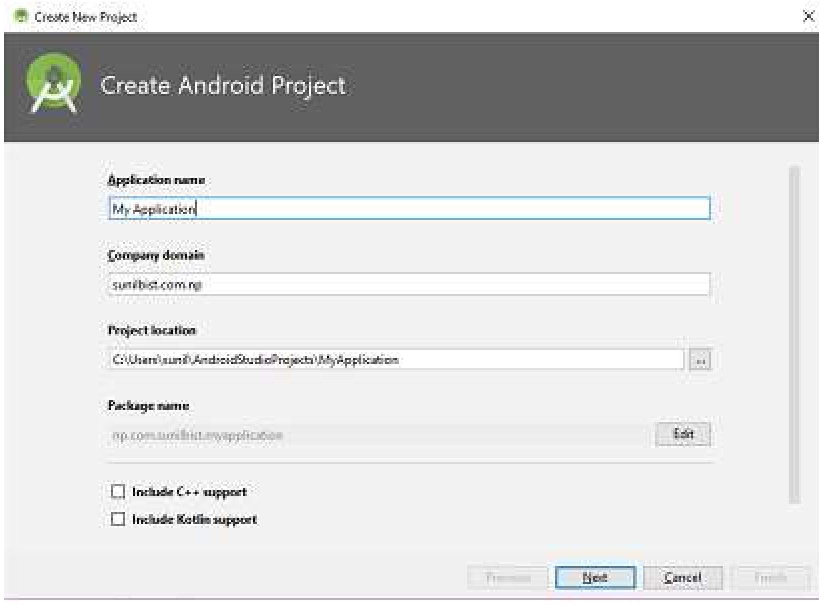
force to use Android Studio instead of Eclipse IDE.
Steps to Create Android App in Android Studio IDE

1. Create Virtual Device (ADV) use following steps:

   - Goto Tool and click on ADV Manager and you will seen an ADV Manager Screen
   - Click on Create Virtual Device and Select your ADV
   - Click on Next and Choose System image for your virtual Device [Note: You must have download System Image and HAXM for faster Access]
   - Now, Click on Next and Supply Necessary Information
   - Click on Finish and go further for Android Apps

2. Create Android Apps use following steps:

   - Goto File and Click on New Project
   - Supply Necessary Project Information as shown in figure below:



Application name: My Application

Company domain: sunilbist.com.np

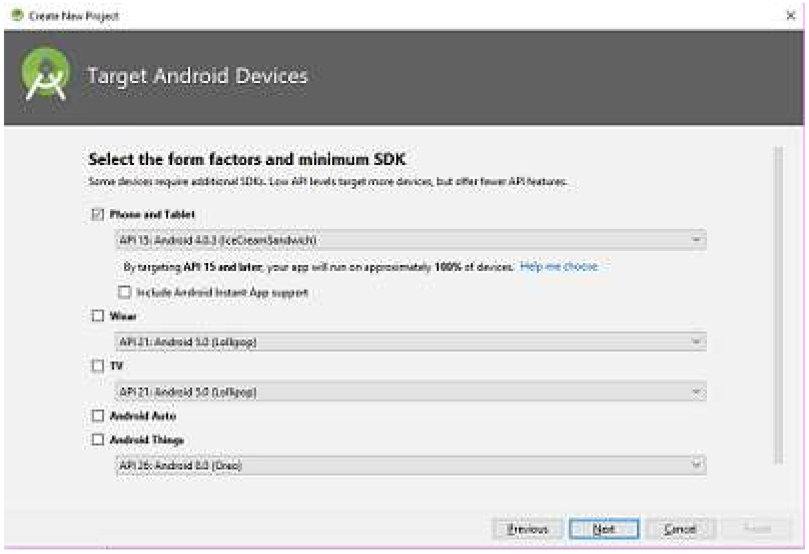
Project location: C:\Users\sunil\AndroidStudioProjects\MyApplication

Package name: np.com.sunilbist.myapplication

☐ Include C++ support

☐ Include Kotlin support

   - Choose Minimum SDK as shown in the figure below:

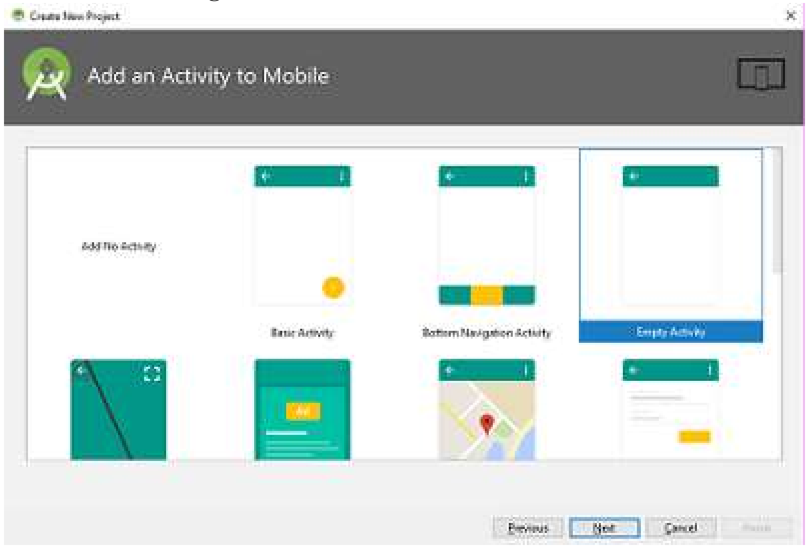


Select the form factors and minimum SDK

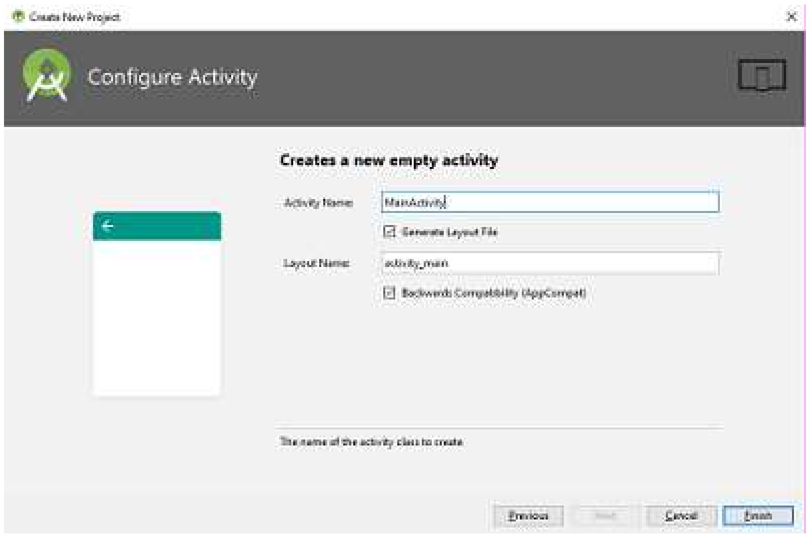
Some devices require additional SDKs. Low API levels target more devices, but offer fewer API features.

[✓] Phone and Tablet
API 15: Android 4.0.3 (IceCreamSandwich)
By targeting **API 15 and later**, your app will run on approximately **100%** of devices. Help me choose
[ ] Include Android Instant App support
[ ] Wear
API 21: Android 5.0 (Lollipop)
[ ] TV
API 21: Android 5.0 (Lollipop)
[ ] Android Auto
[ ] Android Things
API 26: Android 8.0 (Oreo)

- Select Empty Activity from the list of Available Activities as shown in the figure below



- Write Main Activity name for your App as shown in the figure below



Create New Project

Configure Activity

Creates a new empty activity

Activity Name: MainActivity

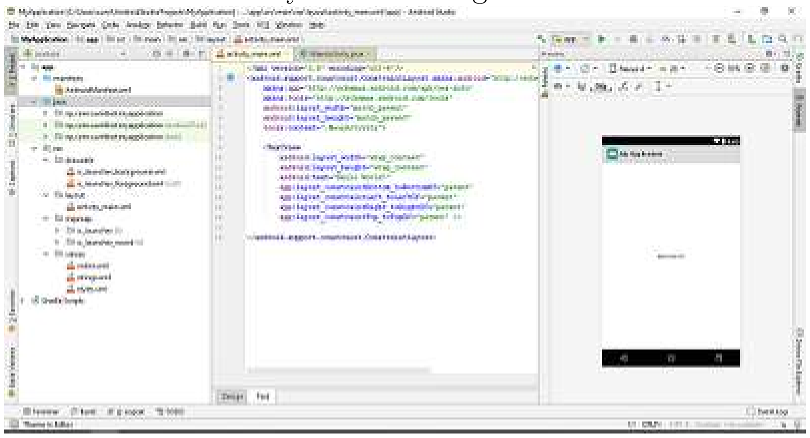
☑ Generate Layout File

Layout Name: activity_main

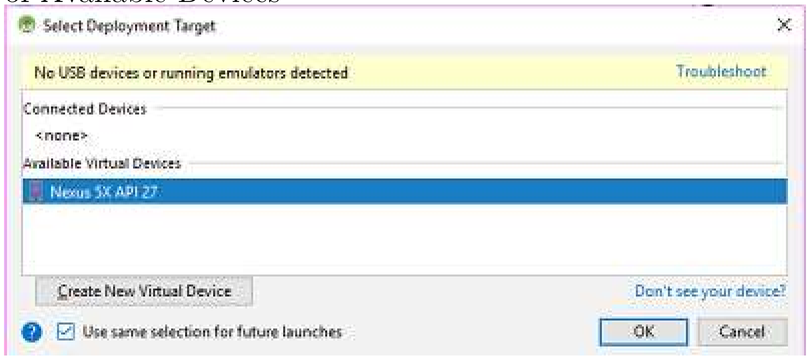
☑ Backwards Compatibility (AppCompat)

The name of the activity class to create

- Now your Android Studio started to create an app [Note: It will take time to synchronize and get create all necessary information]
- After Graddle build you will see figure below:



- Click on Run and you will see Select Deployment Target and List of Available Devices



Select Deployment Target

No USB devices or running emulators detected

Connected Devices

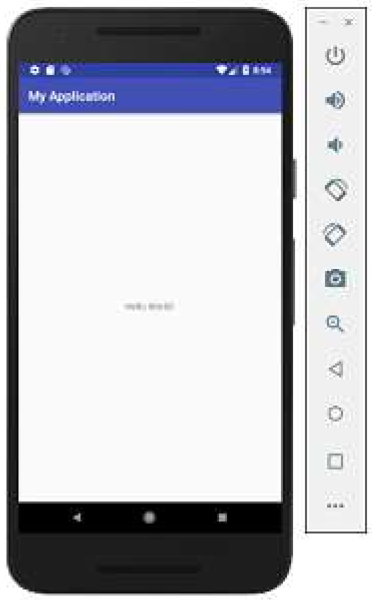
<none>

Available Virtual Devices

Nexus 5X API 27

☑ Use same selection for future launches

- Now click on Ok and you will see output as shown in the figure below:



Now You can move further going to develop Android Apps.

3.1.2 Android Manifest File

Every application must have an "AndroidManifest.xml" file (with precisely that name) in its root directory. The manifest file presents essential information about your app to the Android system, which the system must have before it can run any of the app's code. **Some of the works that are done by manifests are:**

- It names the Java package for the application. The package name serves as a unique identifier for the application.
- It describes the components of the application the activities, services, broadcast receivers, and content providers that the application is composed of. It names the classes that implement each of the components and publishes their capabilities (for example, which Intent messages they can handle). These declarations let the Android system know what the components are and under what conditions they can be launched.
- It determines which processes will host application components.
- It declares which permissions the application must have in order to access protected parts of the API and interact with other applications.

- It also declares the permissions that others are required to have in order to interact with the application's components.
- It lists the Instrumentation classes that provide profiling and other information as the application is running. These declarations are present in the manifest only while the application is being developed and tested; they're removed before the application is published.
- It declares the minimum level of the Android API that the application requires.

The AndroidManifest.xml file look like following: -

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android="http://schemas.android.com/apk/res/android"
   package="np.com.sunilbist.bca">

   <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
       <activity android:name=".MainActivity">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

3.2 The Activity Class

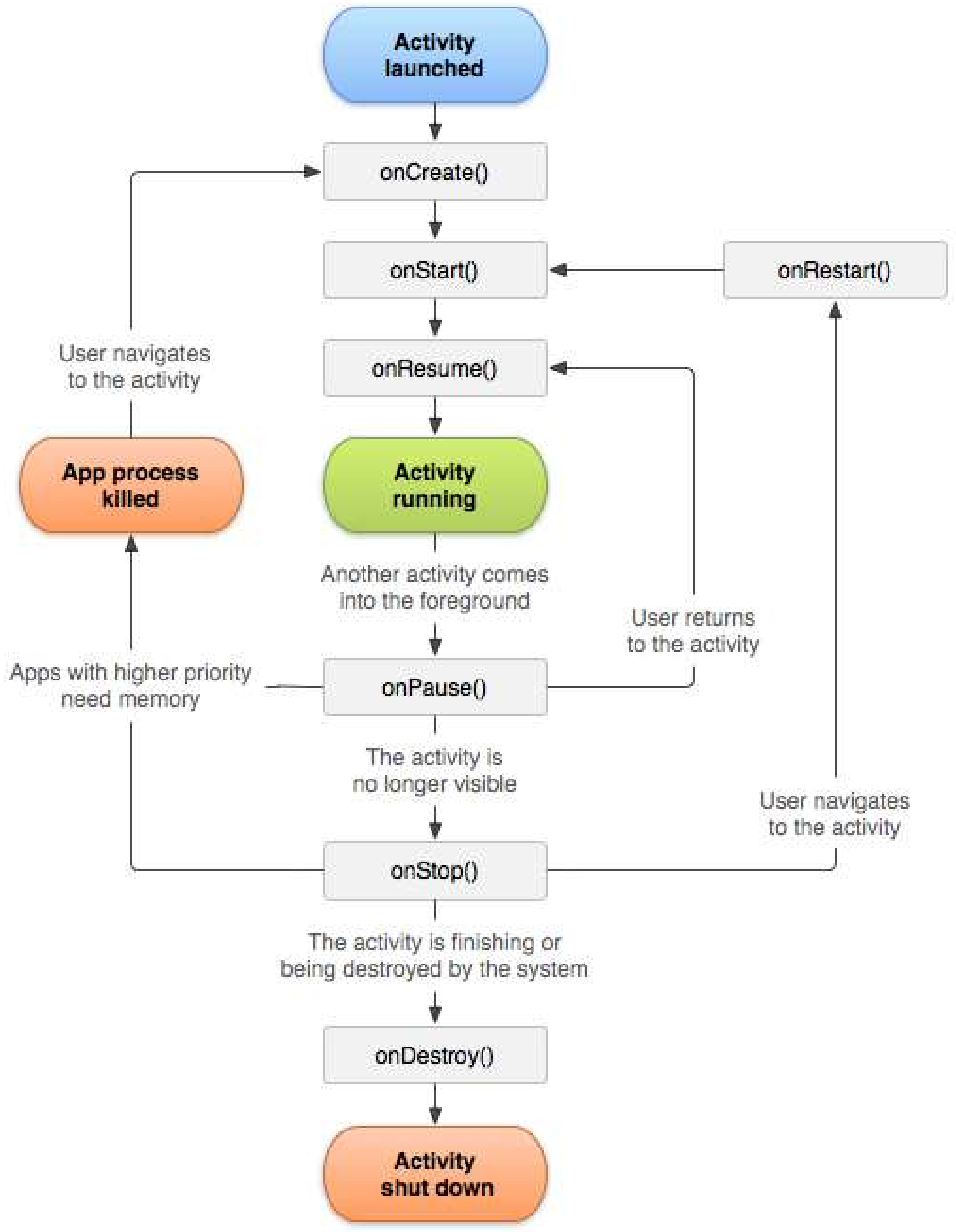
The Activity class is an important part of an application's overall lifecycle, and the way activities are launched and put together is a fundamental part

of the platform's application model.
Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

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

3.2.1 Activity Life-cycle

There are various states of activity like Running, Paused, Stopped and Killed. The process to represent these states of activity can be defined as Activity life cycle.


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

The major event that governs the life cycle of an activity is: **onCreate() :** called when the activity is first created.
onStart() : called when the activity becomes visible to the user.
onResume(): called when the activity starts interacting with the user.
onPause() : called when the current activity is being paused and previous activity is being resumed i.e. activity is going into the background but has

not yet been killed.
onStop() : called when the activity is no longer visible to the user.
onRestart() : called when the activity has been stopped and is restarting again.
onDestroy() : called before the activity is destroyed by the system.

3.2.2 Extending the Activity Class

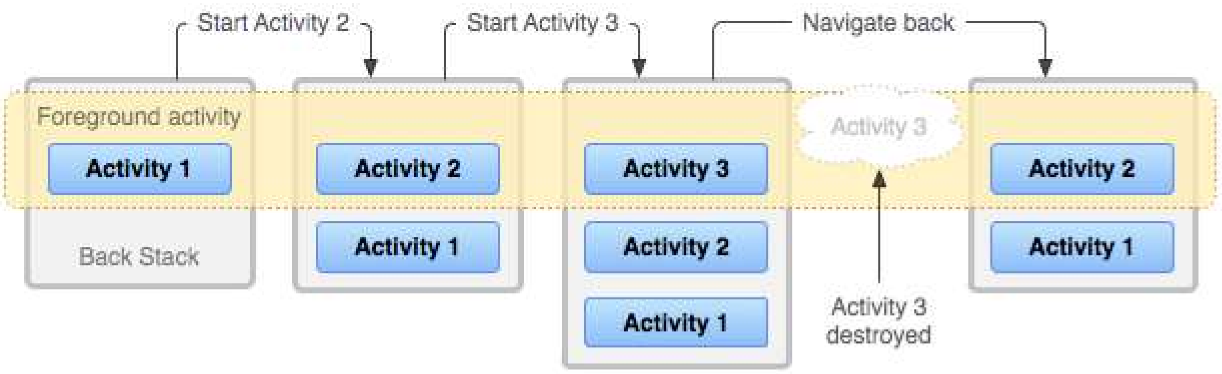
All Android Activities are sub-class of an Activity class. For example,

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

In this example, SubActivity is the child class of Activity class. Where Activity is built-in Super Class in Android.
In the above example, activity_sub is the layout file to display and designed in XML.

3.2.3 Understand Tasks and Back Stack

An application usually contains multiple activities. The activities can be designed to start the activities of the same application or that of the other applications. Even though the activities may be from different applications, Android maintains this seamless user experience keeping both activities in the same task. A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.



The device Home screen is the starting place for most tasks. When the user touches an icon in the application launcher (or a shortcut on the Home screen), that application's task comes to the foreground. If no task exists for the application (the application has not been used recently), then a new task is created and the ”main” activity for that application opens as the root activity in the stack.

When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface. When the user presses the Back button, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored).

Activities in the stack are never rearranged, only pushed and popped from the stack pushed onto the stack when started by the current activity and popped off when the user leaves it using the Back button. As such, the back stack operates as a ”last in, first out” object structure.

3.2.4 Creating Default Activity and Launcher Activity

The activity which is created by default while making our Android project is known as default activity. The activity via which we launch our application is known as launcher activity. The first activity on an Android project by default becomes the launcher activity. To change the launcher activity manually we need to make some changes in the file ”AndroidManifest.xml” Example:

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

```
    package="np.com.sunilbist.bca">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER"
                    />
            </intent-filter>
        </activity>
        <activity android:name=".LoginActivity"></activity>
    </application>
</manifest>
```

Here, in this example we have two activities i.e. 'MainActivity' and 'LoginActivity'. Here MainActivity is the launcher activity. The launcher activity can be distinguished from other activities with the help of "¡intent-filter¿". Here, "¡intentfilter¿" is inside of 'MainActivity' so it is the launcher activity. We can also have both the activities as the launcher activities if we have the code of "¡intent-filter¿" inside of both activities. On doing so, both of the activities will be seen on the application launcher screen and behaves as a launcher activity.

3.2.5 Creating Splash and Login Activities

Splash screens are typically still images that fill the screen of a mobile device. On a desktop computer, they can be full screen (such as for an operating system) or a portion of the screen (for example, Photoshop, Eclipse, and so on). They give feedback that the system has responded to the user's action of opening the application. Splash screens can include loading indicators but are often static images.

3.3 The Intent class

Intent is an abstract description of an operation to be performed. It can be used with

- startActivity to launch an Activity,
- broadcastIntent to send it to any interested BroadcastReceiver components, and
- startService(Intent) or bindService(Intent, ServiceConnection, int) to communicate with a background Service.

Intent provides a facility for performing late runtime binding between the codes in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed. There are two primary forms of intents we will use.

- Explicit Intents have specified a component (via setComponent(ComponentName) or setClass(Context, Class), which provides the exact class to be run. Often these will not include any other information, simply being a way for an application to launch various internal activities it has as the user interacts with the application.
- Implicit Intents have not specified a component; instead, they must include enough information for the system to determine which of the available components is best to run for that intent.

When using implicit intents, given such an arbitrary intent we need to know what to do with it. This is handled by the process of Intent resolution, which maps an Intent to an Activity, BroadcastReceiver, or Service (or sometimes two or more activities/receivers) that can handle it.

3.3.1 Creating Intent

To create an Intenet, we need to instantiate the Intenet class as:

```
Intent intent = new Intent(SourceActivity.this,
   DestinationActivity.class);
startActivity(intent);
finish();
```

3.3.2 Switching between Activities using Intent

To switch between two intent we use the following:

```
button = (Button) findViewById(R.id.button1);

button.setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View view) {

                Intent intent = new Intent(MainActivity.this,
                    LoginActivity.class);
                startActivity(intent);
                finish();
        }
});
```

In this example, MainActivity.java class have a button to switch to another activity, when a button is clicked in MainActivity this will go to the next activity called LoginActivity.java.

3.4 Permissions

Permission is a security mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad hoc access to specific pieces of data. A permission is a restriction limiting access to a part of the code or to data on the device. The limitation is imposed to protect critical data and code that could be misused to distort or damage the user experience. A basic Android application has no permissions associated with it by default, meaning it cannot do anything that would adversely impact the user experience or any data on the device. To make use of protected features of the device, we must include one or more ¡uses-permission¿ tags declaring the permissions in the "AndroidManifest.xml" file.
Then, when the application is installed on the device, the installer determines whether or not to grant the requested permission by checking the authorities that signed the application's certificates and, in some cases, asking the user. If the permission is granted, the application is able to use the protected features. If not, its attempts to access those features will simply fail without any notification to the user.

3.4.1 Allow App Permission in Android Manifest

An app must publicize the permissions it requires by including ¡uses-permission¿ tags in the app manifest. For example, an app that needs to calender information would have this line in the manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="np.com.sunilbist.bca">
    <uses-permission
        android:name="android.permission.READ_CALENDAR"/>
                .
                .
</manifest
```

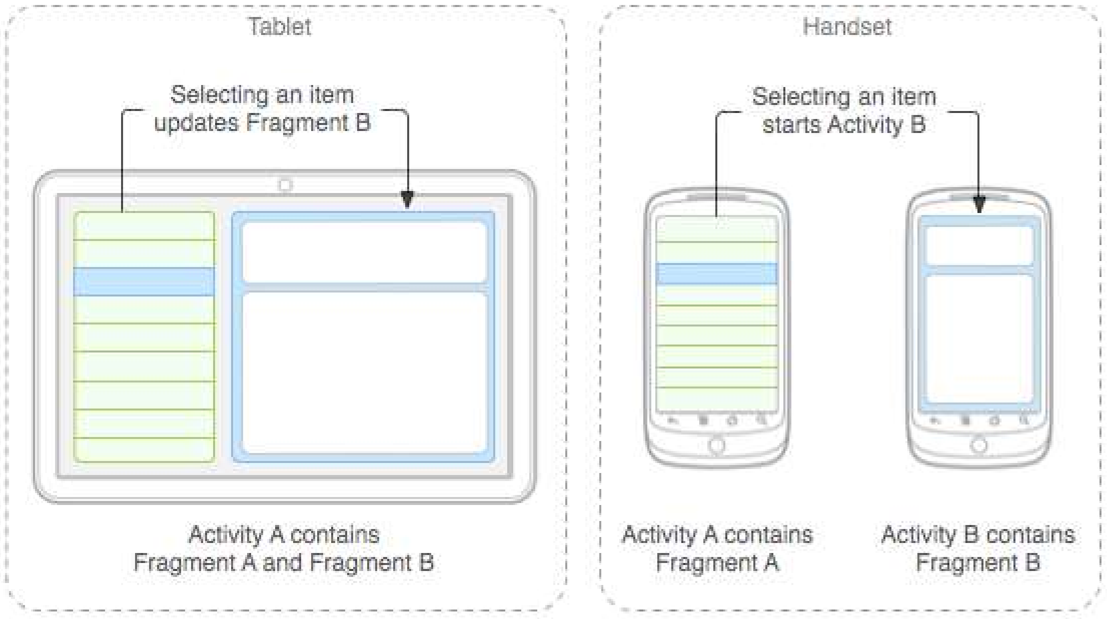
Here, "READ_CALENDAR" is the permission for read to the calender app. Some of the app permissions are:

- ACCESS_LOCATION_EXTRA_COMMANDS
- ACCESS_NETWORK_STATE
- ACCESS_NOTIFICATION_POLICY
- ACCESS_WIFI_STATE
- BLUETOOTH
- BLUETOOTH_ADMIN
- BROADCAST_STICKY
- CHANGE_NETWORK_STATE
- CHANGE_WIFI_MULTICAST_STATE
- CHANGE_WIFI_STATE
- DISABLE_KEYGUARD
- EXPAND_STATUS_BAR
- GET_PACKAGE_SIZE
- INSTALL_SHORTCUT

- INTERNET
- KILL_BACKGROUND_PROCESSES
- MANAGE_OWN_CALLS
- MODIFY_AUDIO_SETTINGS
- NFC
- READ_SYNC_SETTINGS
- READ_SYNC_STATS
- RECEIVE_BOOT_COMPLETED
- REORDER_TASKS
- REQUEST_COMPANION_RUN_IN_BACKGROUND
- REQUEST_COMPANION_USE_DATA_IN_BACKGROUND
- REQUEST_DELETE_PACKAGES
- REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
- SET_ALARM
- SET_WALLPAPER
- SET_WALLPAPER_HINTS
- TRANSMIT_IR
- USE_FINGERPRINT
- VIBRATE
- WAKE_LOCK
- WRITE_SYNC_SETTINGS

3.5 The Fragment Class and Its uses

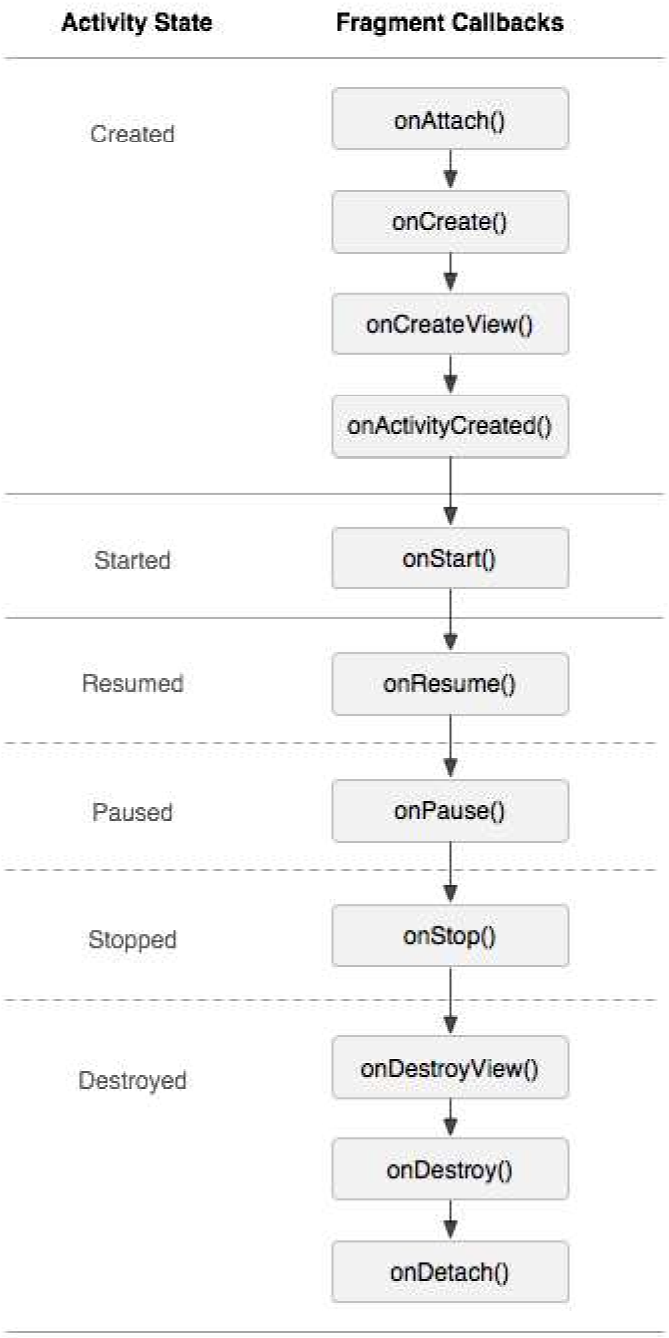
A fragment represents a behavior or a portion of user interface in an activity. Multiple fragments can be combines in a single activity to build a multi-pane UI and reuse a fragment in multiple activities.
A fragment must always be embedded in an activity and the fragment's lifecycle is directly affected by the host activity's lifecycle. For example, when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all fragments.

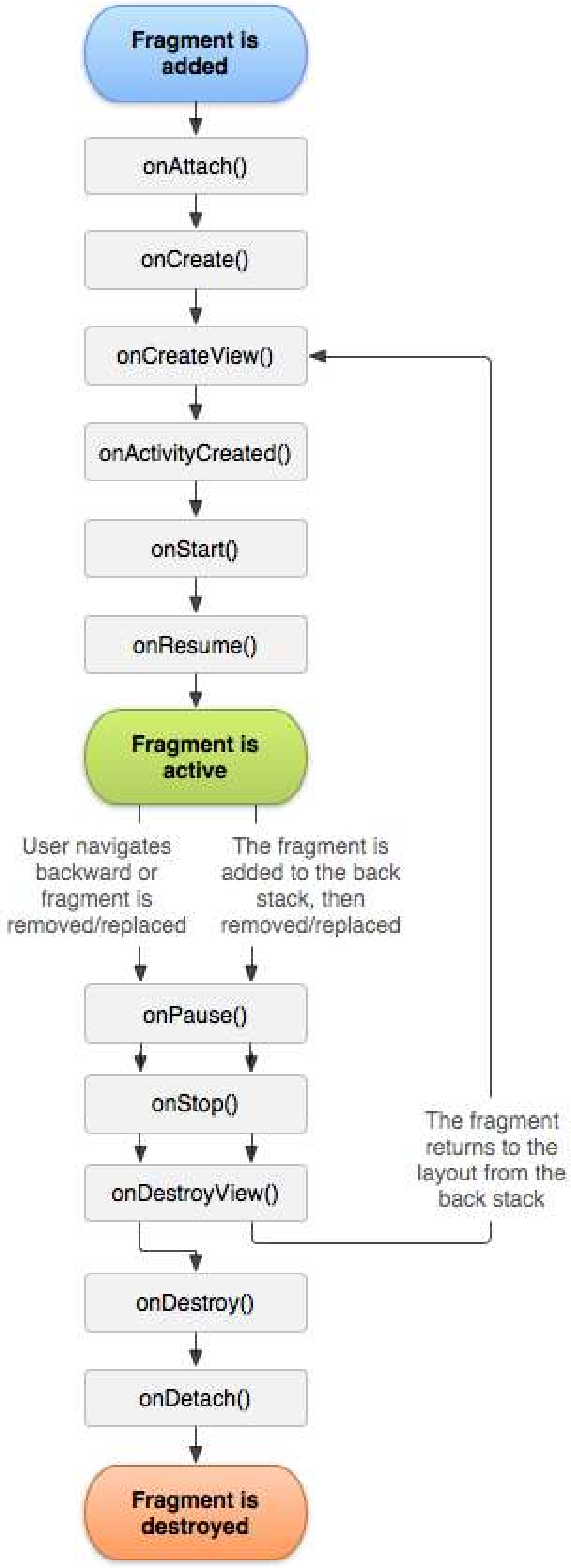

Tablet
Selecting an item updates Fragment B
Activity A contains Fragment A and Fragment B
Handset
Selecting an item starts Activity B
Activity A contains Fragment A
Activity B contains Fragment B

An example of how two UI modules defined by fragments can be combined into one activity for a tablet design, but separated for a handset design

3.5.1 Life Cycle of Fragment

To create a fragment, you must create a subclass of Fragment (or an existing subclass of it). The Fragment class has code that looks a lot like an Activity. It contains callback methods similar to an activity, such as onCreate(), onStart(), onPause(), and onStop()


Activity State
Fragment Callbacks
Created
onAttach()
onCreate()
onCreateView()
onActivityCreated()
Started
onStart()
Resumed
onResume()
Paused
onPause()
Stopped
onStop()
Destroyed
onDestroyView()
onDestroy()
onDetach()


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed

Usually, you should implement at least the following lifecycle methods:
onCreate() The system calls this when creating the fragment. Within your implementation, you should initialize essential components of the fragment that you want to retain when the fragment is paused or stopped, then resumed.
onCreateView() The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout. You can return null if the fragment does not provide a UI.
onPause() The system calls this method as the first indication that the user is leaving the fragment (though it does not always mean the fragment is being destroyed). This is usually where you should commit any changes that should be persisted beyond the current user session (because the user might not come back). Most applications should implement at least these three methods for every fragment, but there are several other callback methods we should also use to handle various stages of the fragment lifecycle. These additional callback methods are:
onAttach() Called when the fragment has been associated with the activity (the Activity is passed in here).
onActivityCreated() Called when the activity's onCreate() method has returned.
onDestroyView() Called when the view hierarchy associated with the fragment is being removed.
onDetach() Called when the fragment is being disassociated from the activity.

3.5.2 Activity vs Fragment

Activity	**Fragment**
An activity is a window with which the user interacts with.	A fragment is a part of an activity, which contributes its own UI to that Activity
An activity may contain zero or multiple number of fragments.	A fragment can be reused in multiple activities, so it acts like a reusable component in activities.
An activity can exist without any fragment in it.	A fragment has to live inside the activity. It should always be the part of an activity.
Activity is needed to be declared in "AndroidManifest.xml"	Fragment is not needed to be declared in "AndroidManifest.xml"
We cannot have nested activities.	We can have nested fragments.