**Activity Life Cycle:**

public class Activity

extends [ContextThemeWrapper](https://developer.android.com/reference/android/view/ContextThemeWrapper) implements [LayoutInflater.Factory2](https://developer.android.com/reference/android/view/LayoutInflater.Factory2), [Window.Callback](https://developer.android.com/reference/android/view/Window.Callback), [KeyEvent.Callback](https://developer.android.com/reference/android/view/KeyEvent.Callback), [View.OnCreateContextMenuListener](https://developer.android.com/reference/android/view/View.OnCreateContextMenuListener), [ComponentCallbacks2](https://developer.android.com/reference/android/content/ComponentCallbacks2)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| java.lang.Object | | | | | | | |
| ↳ | [android.content.Context](https://developer.android.com/reference/android/content/Context.html) | | | | | | |
|  | ↳ | [android.content.ContextWrapper](https://developer.android.com/reference/android/content/ContextWrapper.html) | | | | | |
|  |  | ↳ | [android.view.ContextThemeWrapper](https://developer.android.com/reference/android/view/ContextThemeWrapper.html) | | | | |
|  |  |  | ↳ | [android.app.Activity](https://developer.android.com/reference/android/app/Activity.html) | | | |
|  |  |  |  | ↳ | [androidx.activity.ComponentActivity](https://developer.android.com/reference/androidx/activity/ComponentActivity) | | |
|  |  |  |  |  | ↳ | [androidx.fragment.app.FragmentActivity](https://developer.android.com/reference/androidx/fragment/app/FragmentActivity) | |
|  |  |  |  |  |  | ↳ | androidx.appcompat.app.AppCompatActivity |

Activity is the basic one. Based on Activity , FragmentActivity provides the ability to use Fragment . Based on FragmentActivity , AppCompatActivity provides features to ActionBar

Let's see the 7 lifecycle methods of android activity.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **onCreate** | called when activity is first created. |
| **onStart** | called when activity is becoming visible to the user. (e.i. Checking the permission, register BR, check the login session) |
| **onResume** | called when user will start interacting with the Activity. (ragister initiate the class when we need to receive data in form of flow like Timer, scheduler, playing video, whether or chat, or video app) |
| **onPause** | called when activity is not visible to the user. (un-ragister or un-initiate the class when we need to receive data in form of flow like Timer, scheduler, pause video , whether or chat, or video app) |
| **onStop** | called when activity is no longer visible to the user. (expire the page base session, unregister the BR) |
| **onRestart** | called after your activity is stopped, prior to start. |
| **onDestroy** | called before the activity is destroyed. |

**Fragment Life Cycle:**

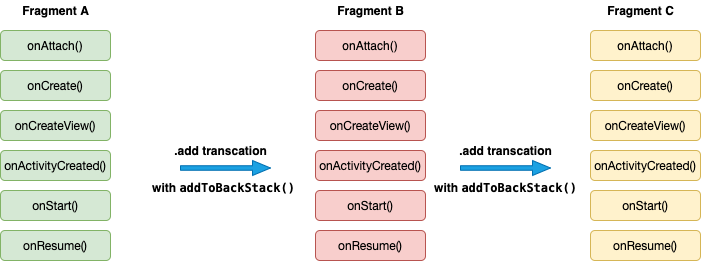
|  |  |
| --- | --- |
| **Method** | **Description** |
| onAttach(Activity) | it is called only once when it is attached with activity. |
| onCreate(Bundle) | It is used to initialize the fragment. |
| onCreateView(LayoutInflater, ViewGroup, Bundle) | creates and returns view hierarchy. |
| onActivityCreated(Bundle) | It is invoked after the completion of onCreate() method. |
| onViewStateRestored(Bundle) | It provides information to the fragment that all the saved state of fragment view hierarchy has been restored. |
| onStart() | makes the fragment visible. |
| onResume() | makes the fragment interactive. |
| onPause() | is called when fragment is no longer interactive. |
| onStop() | is called when fragment is no longer visible. |
| onDestroyView() | allows the fragment to clean up resources. |
| onDestroy() | allows the fragment to do final clean up of fragment state. |
| onDetach() | It is called immediately prior to the fragment no longer being associated with its activity. |

* **Fragment .add Transaction**

For example, there is a **Fragment A**to the **activity**’**s frame layout container**then Fragment A goes through life cycle methods from **onAttach()** to **onResume()**, when you add **Fragment B on top of Fragment A on same container**then Fragment B goes through life cycle methods from **onAttach()**to **onResume()** same happens when you put **Fragment C on top of Fragment B on the same container.**

**Note:**

1. **When we Add fragment B on top A, Then no any method will call from Fragment A**
2. **When Click on Back button Then Current fragment get destroyed and no any method call from backStack fragment**

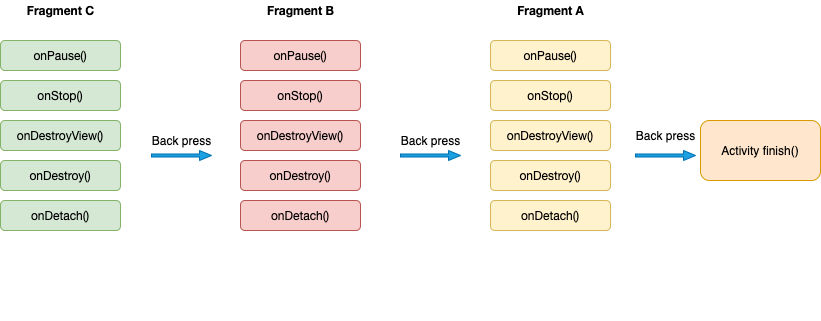


**.add fragment transaction of Fragment A, Fragment B, and Fragment C**

In short when you add a fragment then it calls life cycle methods from onAttach() to onResume().

When you click back press button on an android device then fragment C, fragment B, and Fragment A go through some life cycle methods like below.

Image



**Fragment .replace Transaction**

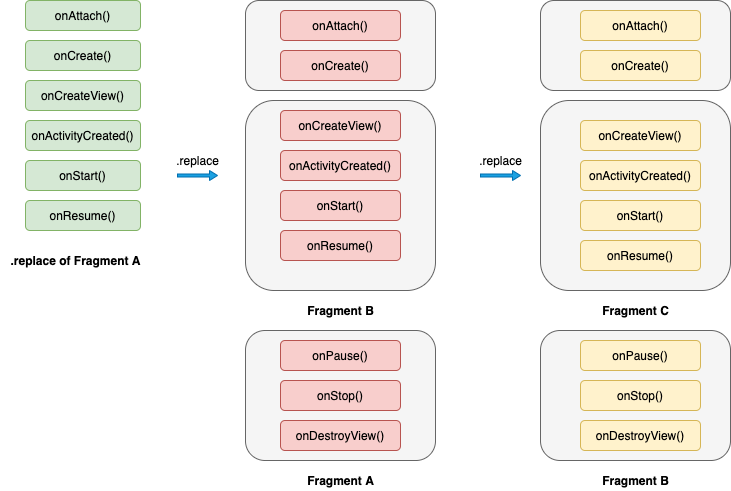
Let’s say you are replacing Fragment A with Fragment B **then first Fragment B**’**s life methods onAttach() and onCreate() gets called** then **Fragment A**’**s life cycle methods get called from onPause() to onDestroyView()**, here **Fragment A won**’**t be detached from the stack it is remembered by the stack.**

Let’s check the below example in which Fragment B is replacing Fragment A and after that Fragment C is replacing Fragment B.

**Note:**

1. When replace Fragment A with B, Then Fragment B life cycle method will from onAttached to OnResume and Fragment A life cycle method will call from onPouse to onDestroyedView.
2. When Press back Button, then current fragment life cycle method will from onPoused to onDeAttached and backStack fragment life cycle method will call onCreateView to onResume

Image

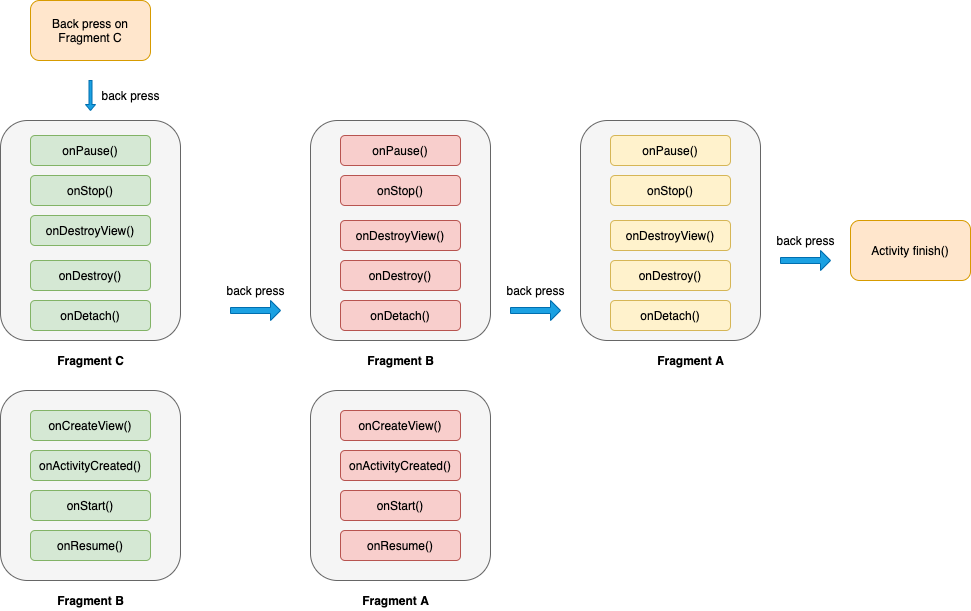


**.replace of Fragment A, then .replace of fragment B and then .replace of fragment C**

**Let**’**s check what happens when we click back press button**

When you click back press button on an android device then fragment C, fragment B, and Fragment A go through some life cycle methods like below.

Image



**back press clicks on fragment C, fragment B, and fragment A respectively**

**add()** method keeps on adding fragments **on top of** the previous fragment in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()

.add(R.id.fragmentContainer, fragment, "TAG")

.addToBackStack("TAG")

.commit();

**replace()** methods **clears all** the previous Fragment from Containers and then add it in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()

transaction.replace(R.id.fragment\_layout\_id, homeFragment())

transaction.addToBackStack("TAG")

transaction.commit()

**[addToBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentTransaction#addToBackStack(java.lang.String))**: Calling [addToBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentTransaction#addToBackStack(java.lang.String)) commits the transaction to the back stack. The user can later reverse the transaction and bring back the previous fragment by pressing the *Back* button. If you added or removed multiple fragments within a single transaction, all of those operations are undone when the back stack is popped. The optional name provided in the addToBackStack() call gives you the ability to pop back to that specific transaction using [popBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentManager#popBackStack(java.lang.String,%20int)).

**Activity Life Cycle @Important Point**

Suppose we have 4 activity ActvityA, ActvityB, ActvityC, ActvityD

1. **When App lunch or click on app icon (if activity A is lunching activity)**
   1. ActivityA : onCreate()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
2. **When Move to another Activity(From A To B)**
   1. ActivityA : onPause()
   2. ActivityB : onCreate()
   3. ActivityB : onStart()
   4. ActivityB : onResume()
   5. ActivityA : onStop()
   6. ActivityA : onSaveInstanceState()
3. **When click on Device Back Button(back from B To A)** 
   1. ActivityB : onPause()
   2. ActivityA : onRestart()
   3. ActivityA : onStart()
   4. ActivityA : onResume()
   5. ActivityB : onStop()
   6. ActivityB : onDestroy()
4. **When we press device back & have only activity A in backStack**
   1. ActivityA : onPause()
   2. ActivityA : onStop()
   3. ActivityA : onDestroy()
5. **When we relaunch app from app back stack (like we have App & press device back button and relaunch app from app stack ) or Open App via Recent**
   1. ActivityB : onCreate()
   2. ActivityB : onStart()
   3. ActivityB : onResume()
6. **When we press device Home**
   1. ActivityA : onPause()
   2. ActivityA : onStop()/
   3. ActivityA : onSaveInstanceState()
7. **When we press device Home and relaunch app from back stack or Open App via Recent**
   1. ActivityA : onRestart()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
8. **When we press device Luck button**
   1. ActivityA : onPause()
   2. ActivityA : onStop()
9. **When we press device release Luck button**
   1. ActivityA : onRestart()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
10. **Killing The app from recent back-stack**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onDestroy()
11. **When Device configuration change or rotate device screen** 
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
    4. ActivityA : onDestroy()
    5. ActivityA : onCreate()
    6. ActivityA : onStart()
    7. ActivityA : onRestoreInstanceState()
    8. ActivityA : onResume()
12. **Suppose, when you on ActivityA on your phone is ringing from top | notification** 
    1. Nothing activity will call
13. **Suppose, when you on ActivityA on your phone is ringing from top | notification and on decline button from notification only** 
    1. Nothing activity will call
14. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just click only on notification icon /message and your call manager activity open with call details**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
15. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just click only on notification icon /message and your call manager activity open with call details and you didn't Answer that call**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()
16. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just answer/pic/received that call**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
17. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just answer/pic/received that call and after some discussion back to activity**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()
18. **Switching A-App to B-App**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
19. **, Switching Back to from B-App to A-App**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()

**Activity Lunch Mode standard**

* **standard**
* **singleTop**
* **singleTask**
* **singleInstance**

<activity android:launchMode = [“standard” | “singleTop” | “singleTask” | “singleInstance”] ../>

**1. standard:** This is the **default launch mode** of activity. If you don’t set any launch mode to your activity, it will use the standard mode by default. **It creates a new instance of activity every time even if activity instance is already present.**

Suppose we have A, B, C, and D activities and your activity B has standard launch mode. Now again launching activity B

State of Activity Stack before launch B

A →B→ C→D

State of Activity Stack after launch B

A → B → C→D→ B

We can see that new instance of B is created again

**2. singleTop:** If an instance of activity already exists at the top of the current task, a new instance will not be created and the Android system will route the intent information through onNewIntent().

If an instance is not present on top of the task then a new instance will be created.

Suppose we have A, B, C, and D activities. A →B →C →D and set the C as **SingleTop**

If we launch C then a new instance of C will be created as it is not on top.

So it will look like A →B →C →D →C

Now suppose we have A →B →C →D →C like this

then we if again launch C activity then in this case new instance will not be created. Instead, we will receive the callback on onNewIntent() method.

**3. singleTask:** If An activity declared with launch mode as singleTask can have only one instance in the system (singleton). **At a time only one instance of activity will exist.**

**If activity instance is not present then the new instance will be created and if the instance is already present in the system then the onNewIntent() method will receive the callback.**

Suppose we have A, B, C activities(A →B →C ) and we are launching D that has a singleTask launch mode. In that case, the new instance of D will be created so the current state will look like this. (A →B →C →D)

Now let suppose if we launch B that also have has a singleTask launch mode then current state will look like

A →B

Here old instance gets called and intent data route through onNewIntent() callback. Also, notice that C and D activities get destroyed here.

**4. singleInstance :** It is similar to singleTask except that no other activities will be created in the same task. If another Activity is called from this kind of Activity, a new Task would be automatically created to place that new Activity.

Case 1:

Suppose you have A, B, and C activities(A →B →C) and your activity D has a singleInstance launch mode. In this case, if we launch D then D will be launch in the diffrent task. New task for D will be created.

Task1: A →B →C

Task2 : D (here D will be in the different task)

Now if you continue this and start E and D then Stack will look like

Task1: A →B →C →E

Task2: D

**android:exported:**  android:exported Whether or not the broadcast receiver can receive messages from sources outside its application — **"true" if it can, and "false" if not**. If "false", the only messages the broadcast receiver can receive are those sent by components of the same application or applications with the same user ID. The exported attribute is used to define if an activity, service, or receiver in your app is accessible and can be launched from an external application. As a practical example, if you try to share a file you'll see a set of applications available.

**<intent-filter>** : Specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle. Most of the contents of the filter are described by its**[<action>](https://developer.android.com/guide/topics/manifest/action-element), [<category>](https://developer.android.com/guide/topics/manifest/category-element), and [<data>](https://developer.android.com/guide/topics/manifest/data-element)** subelements.

<intent-filter android:[icon](https://developer.android.com/guide/topics/manifest/intent-filter-element#icon)="*drawable resource*"

               android:[label](https://developer.android.com/guide/topics/manifest/intent-filter-element#label)="*string resource*"

               android:[priority](https://developer.android.com/guide/topics/manifest/intent-filter-element#priority)="*integer*" >

    . . .

</intent-filter>

**<meta-data>:** It’s used to store arbitrary information about the app for other app/component/sdk. A name-value pair for an item of additional, arbitrary data that can be supplied to the parent component. A component element can contain any number of <meta-data> subelements. The values from all of them are collected in a single [Bundle](https://developer.android.com/reference/android/os/Bundle) object and made available to the component as the [PackageItemInfo.metaData](https://developer.android.com/reference/android/content/pm/PackageItemInfo#metaData) field.

Ordinary values are specified through the [value](https://developer.android.com/guide/topics/manifest/meta-data-element#value) attribute. However, to assign a resource ID as the value, use the [resource](https://developer.android.com/guide/topics/manifest/meta-data-element#resource) attribute instead.

<meta-data android:[name](https://developer.android.com/guide/topics/manifest/meta-data-element#nm)="*string*"

           android:**[resource](https://developer.android.com/guide/topics/manifest/meta-data-element#rsrc)**="*resource specification*"

           android:[value](https://developer.android.com/guide/topics/manifest/meta-data-element#val)="*string*" />

<meta-data

android:name="com.facebook.sdk.ApplicationId"

android:value="@string/facebook\_app\_id"/>

**Build Type:** Build types: In Android apps, build types usually refer to the environment in which you're testing. By default, when you create an app in Android Studio from any of the templates, you get two build types: debug and release.

The buildTypes block is where you can configure multiple build types. By default, the build system defines two build types: debug and release. The debug build type is not explicitly shown in the default build configuration, but it includes debugging tools and is signed with the debug key. The release build type applies Proguard settings and is not signed by default.

**File => Project structure => Build Variants => Select Module(Default App) => Build Type => (Add +, Delete -, Edit)**

debug {

applicationIdSsuffiix “.bebug”

versionNameSuffix “-Debug”

minifyEnabled false

debuggable true

}

qa {

applicationIdSsuffiix “.qa”

versionNameSuffix “-QA”

minifyEnabled true

debuggable false

}

staging {

applicationIdSsuffiix “.staging”

versionNameSuffix “-Staging”

minifyEnabled true

debuggable false

}

release {

minifyEnabled true

debuggable false

proguardFiles getDefaultProguardFile(‘proguard-android- optimize.txt’), ‘proguard-rules.pro’

}

Gradle:  Gradle, **an advanced build toolkit**, to automate and manage the build process, while allowing you to define flexible custom build configurations.

**compileSdkVersion**:

compileSdkVersion specifies the Android API level Gradle should use to compile your app.

This means your app can use the API features included in this API level and lower.

**targetSdkVersion:** Specifies the API level used to test the app.

**minSdkVersion:** Defines the minimum API level required to run the app.

**Product Flavours:** Product flavours lets you create multiple variants(Free/paid flavours/Stable/staining/experimental) of an android app while using a single codebase. To create product flavours you need to define rules in the build.

The productFlavors block is where you can configure multiple product flavors.This allows you to create different versions of your app that can override the defaultConfig block with their own settings. Product flavors are optional, and the build system does not create them by default.

This example creates a free and paid product flavor. Each product flavor then specifies its own application ID, so that they can exist on the Google Play Store, or an Android device, simultaneously

**File => Project structure => Build Variants => Select Module(Default App) => Flavours => (Add +, Delete -, Edit)**

flavorDimensions 'default'

productFlavors {

redFlavor {

dimension 'default'

applicationId "com.sample.productFlavors"

minSdkVersion 15

targetSdkVersion 28

buildConfigField "String", "HOST\_URL", "\"host url 1\""

}

greenFlavor {

dimension 'default'

applicationId "com.sample.productFlavors"

minSdkVersion 18

targetSdkVersion 28

buildConfigField "String", "HOST\_URL", "\"host url 2\""

}

}

sourceSets {

main {

manifest.srcFile 'AndroidManifest.xml'

java.srcDirs = ['src/commonFiles/java']

resources.srcDirs = ['src/commonFiles/java']

aidl.srcDirs = ['src/commonFiles/java']

renderscript.srcDirs = ['src/commonFiles/java']

res.srcDirs = ['res']

assets.srcDirs = ['assets']

}

redFlavor {

manifest.srcFile 'AndroidManifest-flavor2.xml'

res.srcDirs = ['res-flavor2', 'res']

}

greenFlavor{

manifest.srcFile 'AndroidManifestXXX.xml'

java.srcDirs = ['src\_qa']

}

**adb:** Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

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

adb is included in the Android SDK Platform-Tools package. You can download this package with the [SDK Manager](https://developer.android.com/studio/intro/update#sdk-manager), which installs it at *android\_sdk*/platform-tools/.

**How adb works**

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

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators

**Some ADB Command:**

**1. adb devices =>** show list of device

**2. adb install *path\_to\_apk:*** *Install the App from apk path*

*3.* **adb shell pm uninstall com.xxx.xx** => To uninstall the app from device