**MAD (MCA 4th & IT 8th Sem) Unit-1**

**Android:-**

Android is a mobile operating system based on a modified version of the Linux kernel and other open source software, designed primarily for touchscreen mobile devices such as smartphones and tablets. Android is developed by a consortium of developers known as the Open Handset Alliance and commercially sponsored by Google. It was unveiled in November 2007, with the first commercial Android device, the HTC Dream, being launched in September 2008.

**BlackBerry OS:-**

BlackBerry OS is a proprietary mobile operating system designed specifically for Research In Motion's (RIM) BlackBerry devices. The BlackBerry OS runs on Blackberry variant phones like the BlackBerry Bold, Curve, and Pearl and Storm series.

The BlackBerry OS is designed for smartphone environments and is best known for its robust support for push Internet email. This push email functionality is carried out through the dedicated BlackBerry Enterprise Server (BES), which has versions for Microsoft Exchange, Lotus Domino and Novell GroupWise. BlackBerry OS can run only on BlackBerry phones. BlackBerry OS is similar to Apple's IOS in this regard. BlackBerry applications are written using Java, particularly the Java Micro Edition (Java ME) platform. However, **RIM** introduced the BlackBerry Web development platform in 2010, which makes use of the widget software development kit (SDK) to create small standalone Web apps made up of HTML, CSS and JavaScript code.

**RIM:** RIM first announced their Mobile Fusion service way back in November 2011, In IT departments to manage those myriad devices from a single web-based "control room." From their console, admins will be able to (among other things) remotely enable access to email and contacts, manage lost hardware, establish security policies and push notifications to the user.
Firefox OS[4] (project name: Boot to Gecko, also known as B2G)[5] is a discontinued open-source operating system – made for smartphones,[6] tablet computers,[7] smart TVs[8] and dongles designed by Mozilla and external contributors. It is based on the rendering engine of the Firefox web browser, Gecko, and on the Linux kernel. It was first commercially released in 2013.

**Firefox OS:**

Firefox OS was designed to provide a complete community-based alternative operating system, for running web applications directly or those installed from an application marketplace. The applications use open standards and approaches such as JavaScript and HTML-5. Firefox OS was publicly demonstrated in February 2012, on Android-compatible smartphones By December 16, 2014, fourteen operators in 28 countries throughout the world offered Firefox OS phones.

On December 8, 2015, Mozilla announced that it would stop sales of Firefox OS smartphones through carriers. Mozilla later announced that Firefox OS smartphones would be discontinued by May 2016, as the development of "Firefox OS for smartphones" would cease after the release of version 2.6. Around the same time, it was reported that Acadine Technologies, a startup founded by Li Gong (former president of Mozilla Corporation) with various other former Mozilla staff among its employees, would take over the mission of developing carrier partnerships, for its own Firefox OS derivative H5-OS.

In January 2016 Mozilla announced that Firefox OS would power Panasonic's UHD TVs (as previously announced Firefox OS "would pivot to connected devices"). In September 2016 Mozilla announced that work on Firefox OS had ceased and that all B2G-related code would be removed from Mozilla-central.

**ARM & MIPS Processor:**

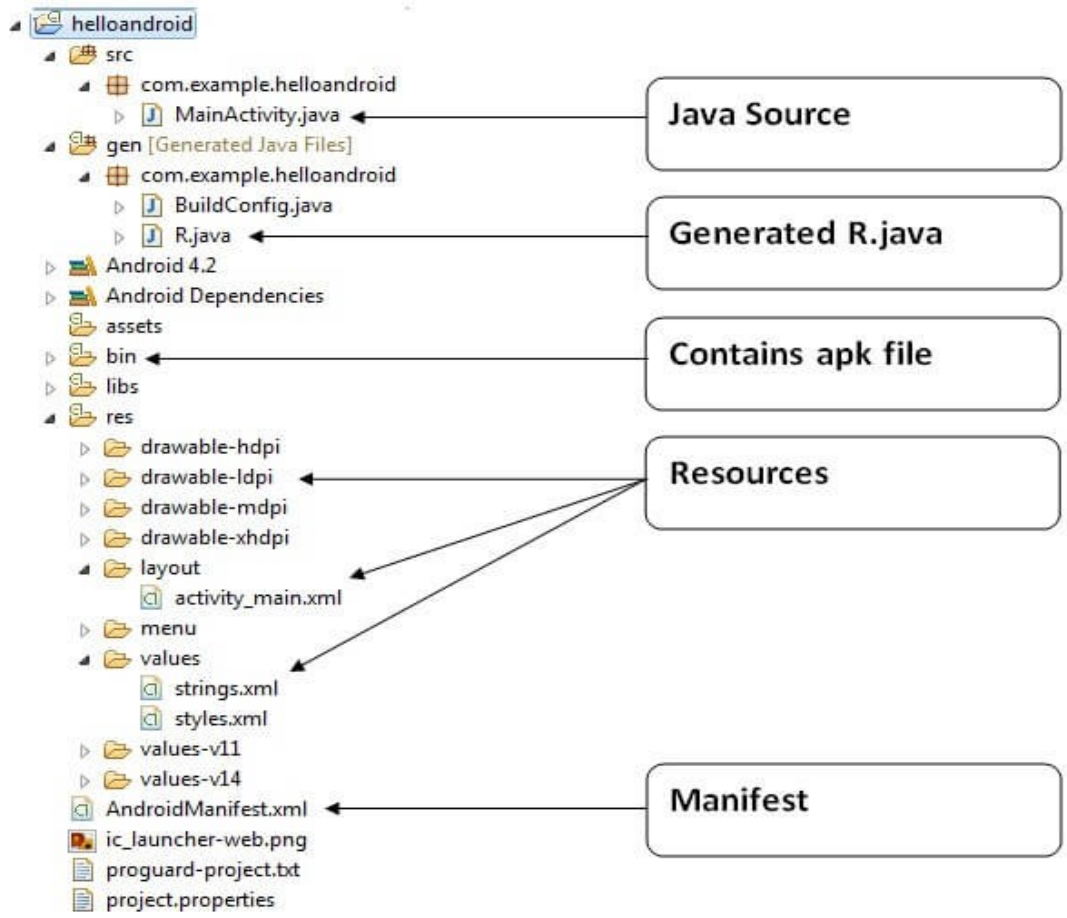
**ARM (Advanced RISC Machines and originally Acorn RISC Machine) is a family of reduced instruction set computer (RISC) instruction set architectures for computer processors, configured for various environments. Million instructions per second (MIPS) is an approximate measure of a computer's raw processing power.**
What is the difference between MIPS and ARM? MIPS and ARM are two different instruction set architectures in the family of RISC instruction set. Although both the instruction sets have a fixed and same instruction size, ARM has only 16 registers while MIPS has 32 registers. Although

both the instruction sets have a fixed and same instruction size, ARM has only 16 registers while MIPS has 32 registers. ARM has a high throughput and a great efficiency than MIPS because ARM processors support 64-bit data buses between the core and the caches.

MIPS (Microprocessor without Interlocked Pipelined Stages) is a family of reduced instruction set computer (RISC) instruction set architectures (ISA):
Page size: 4 KB, General purpose: 32 Bits: 64-bit (32 → 64) Floating point: 32 & MIPS Technologies are used. ·

**Hello: Andriod Example Internal:-** Here, we are going to learn the internal details or working of hello android example.

Android application contains different components such as java source code, string resources, images, manifest file, apk file etc. Let's understand the project structure of android application.




## Software Stack:-

Software stack is **a collection of independent components that work together to support the execution of an application**. The components, which may include an

operating system, architectural layers, protocols, runtime environments, databases and function calls, are stacked one on top of each other in a hierarchy.

One of popular software stacks is LAMP (Linux, Apache, MYSQL, Perl or PHP or Python). LAMP is an open source development platform for creating and

managing Web applications. Linux serves as the backend operating system (OS). Apache is the Web server, MySQL is the database, and either PHP, Perl or Python

are used for scripts.

**Core Building Blocks of Android:-**

An android **component** is simply a piece of code that has a well defined life cycle e.g. Activity, Receiver and Service etc. The **core building blocks** or

**fundamental components** of android are activities, views, intents, services, content providers, fragments and AndroidManifest.xml.

Activity:- An activity is a class that represents a single screen. It is like a Frame in AWT.

View :- A view is the UI (user Interface )element such as button, label, text field etc. Anything that you see is a view.

**Component in Android:-** In android, the component is an already pre-written code that has a life cycle. The Core building blocks include these components.

In which the Android Applications can build. Core building blocks are also known as the fundamental component. Understanding the fundamental Component is

important for Android Developers.

**Fundamental components of android are:-** Activity, Intent, Service, Content Provider, Broadcast Receiver.

**Activity:-** Activity is a screen. Activity is the main Screen which contains the User Interface (UI) and Logic. It contains XML and java code. The activity file contains

two files Activity.xml which is used for User Interface (UI) and the second file contains java file (Activity.java) used for writing Logic.

**Intent:-** The intent is message passing object. If you want to communicate from one activity to other then we can go for intent. Intent also helps to

communicate with other components like service or broadcast receiver.

**Uses of intent :**

- Start Activity.
- Start Service.
- Delivering broadcast.
- Opening a webpage.
- There are two types of intent
- Implicit Intent:
- Explicit Intent

**Services In Android:-** Services are an android component that can perform a long-running task in the background. Service doesn't have any user Interfaces.

Because services is runs in the background. To start a service in android another application is required. Service cannot start by itself. Android service is not a

thread or separate process.

**Examples of services are:--**

- Music player who plays music in the background.
- Network transaction Applications.
- The content provider does database-related tasks in the background.
- Performing file input and output
- Above are very common examples of services.

## Life Cycle of Service :--

When service gets started with the start Service() method then its life cycle starts. service will run in the background until it is stopped by stop Service()

or stop self() method. Bound service is another component that binds the service. Using bind Service() method. Service always runs on the main thread.

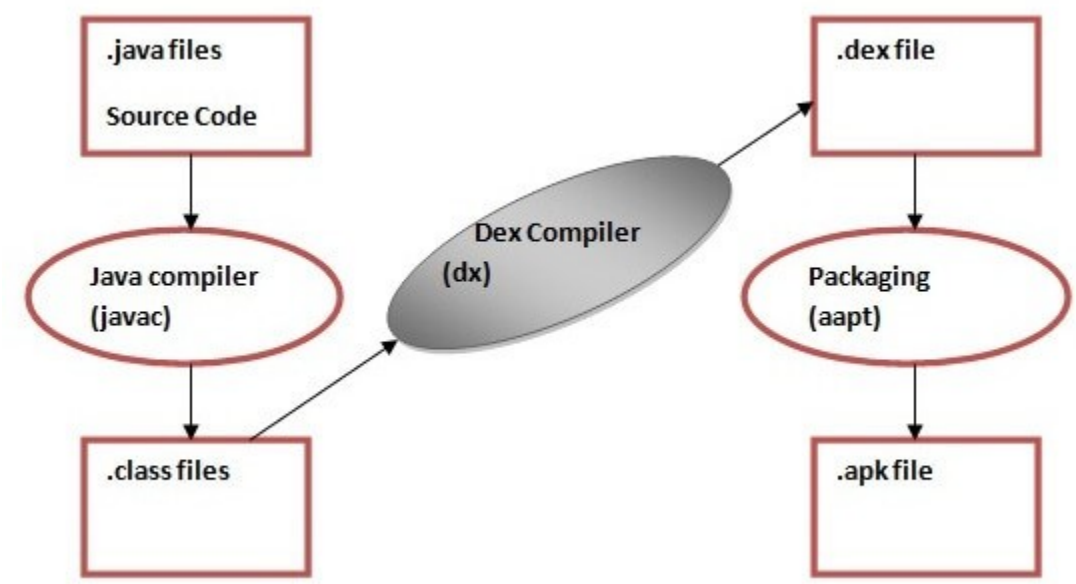
We can create two types of services in android

- Background service: A service that operates that does not directly notice by the user.
- Foreground service: A service that performs a certain operation that is noticeable to end-users is called foreground service.

An example is: Playing audio track. These services will continue to run until the user closes it. Foreground service must display a notification.

**Dalvik Virtual Machine (DVM) :-**

**DVM** is an android virtual machine optimized for mobile devices. It optimizes the virtual machine for *memory*, *battery life* and *performance*.
Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein. The Dex compiler converts the class files into the .dex file that run on the Dalvik VM. Multiple class files are converted into one dex file. JVM is high performance and provides excellent memory management. But it needs to be optimized for low-powered handheld devices as well.




Android Emulator:

The Android Emulator simulates Android devices on your computer so that you can test your application on a variety of devices and Android API levels without needing

to have each physical device. The emulator provides almost all of the capabilities of a real Android device. You can simulate incoming phone calls and text messages,
specify the location of the device, simulate different network speeds, simulate rotation and other hardware sensors, access the Google Play Store, and much more.

you **can even use the Play Store** if you want to go through the effort. Google also launched **Google Play Games on Windows 11 in beta**.

- **BlueStacks**
- **MuMu**
- **Nox**
- **Phoenix OS**
- **PrimeOS**
- **Remix OS Player**
- **Xamarin**
- **Make your own**

- **LDPlayer**

- **Android Studio**
- **ARChon**
- **Bliss OS**
- **GameLoop**
- **Genymotion**

**AndriodMenifest.xml:**

Every app project must have an AndroidManifest.xml file (with precisely that name) at the root of the project source set. The manifest file describes essential information

about your app to the Android build tools, the Android operating system, and Google Play. Among many other things, the manifest file is required to declare the following:

- The components of the app, which include all activities, services, broadcast receivers, and content providers. Each component must define basic properties such as the name
- of its Kotlin or Java class. It can also declare capabilities such as which device configurations it can handle, and intent filters that describe how the component can be started.
- Read more about app components. The permissions that the app needs in order to access protected parts of the system or other apps. It also declares any permissions
- that other apps must have if they want to access content from this app. Read more about permissions. The hardware and software features the app requires, which affects which
- devices can install the app from Google Play. Read more about device compatibility.

## Unit-2

**UI Widgets:**

There are given a lot of **android widgets** with simplified examples such as Button, EditText, AutoCompleteTextView, ToggleButton, DatePicker, TimePicker, ProgressBar etc.

Android widgets are easy to learn. The widely used android widgets with examples are given as:

Android Button:- Let's learn how to perform event handling on button click.

Android Toast:- Displays information for the short duration of time.

Custom Toast : -We are able to customize the toast, such as we can display image on the toast.

ToggleButton:- It has two states ON/OFF.

CheckBox :- Let's see the application of simple food ordering.

AlertDialog:- AlertDialog displays a alert dialog containing the message with OK and Cancel buttons.

Spinner:- Spinner displays the multiple options, but only one can be selected at a time.

AutoCompleteTextView:- Let's see the simple example of AutoCompleteTextView.

RatingBar :-RatingBar displays the rating bar.

DatePicker: -Datepicker displays the datepicker dialog that can be used to pick the date.

TimePicker: TimePicker displays the timepicker dialog that can be used to pick the time.

ProgressBar: ProgressBar displays progress task.

Android Analog clock and Digital clock:- The android.widget.AnalogClock **&** android.widget.DigitalClock classes provides the functionality to display analog and digital clocks.

- Android analog and digital clocks are used to show time in android application. Android Analog Clock is the subclass of View class.
- Android Digital Clock is the subclass of TextView class. Recommendable to use **Text Clock** Instead of that.

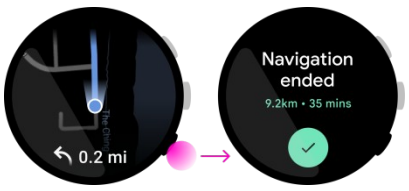
**Hardware buttons:--**

Wear OS watches may have different hardware button configurations. This guide goes over the best use cases for each of these button types.

**Button types**

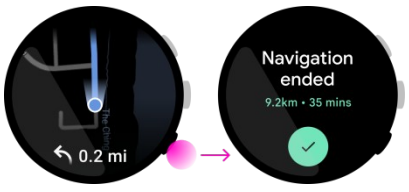
The following are the most common button types on Wear OS devices.

**OS buttons**




OS buttons are reserved for system actions: turning power on and off, and launching apps. All Wear OS watches have a power button and a launcher button.

**Multifunction buttons**




Buttons on the watch face or screen are OS-configurable and user-configurable. Any other buttons can be mapped to actions. Buttons can be mapped to convenient actions based on where they are located on the watch.

**Press states**

You can interact with Wear OS buttons in the following ways. **Single press**

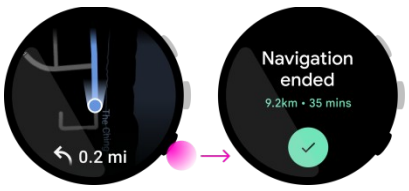



**Figure 1.** User presses the button and releases it quickly. **Press and hold**


**Figure 2.** User presses the botton and holds it for 500ms or longer.

**Multifunction button mapping**

Your app can assign multifunction buttons to actions if doing so fits your app's use case. Apps are not required to assign actions to multifunction buttons.

Use multifunction buttons in your app if one of the following conditions applies:

- Your app has obvious, binary actions (such as play/pause).
- The user primarily uses your app without the user looking at the display.

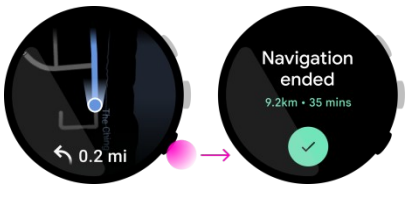



**Figure 3.** This fitness app has assigned a pause/resume action to a multifunction button, which allows the user to perform the action without looking at the screen.

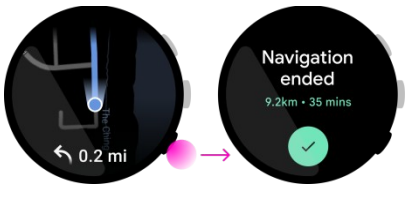



**Figure 4.** This messaging app includes a reply action, which requires multiple steps and can’t be completed with a single button press.

**Binary actions**

Binary actions help users understand what will happen each time they press a button. For example, "start" and "stop" on a stopwatch constitute a binary action, and represent a good use case for multifunction buttons.

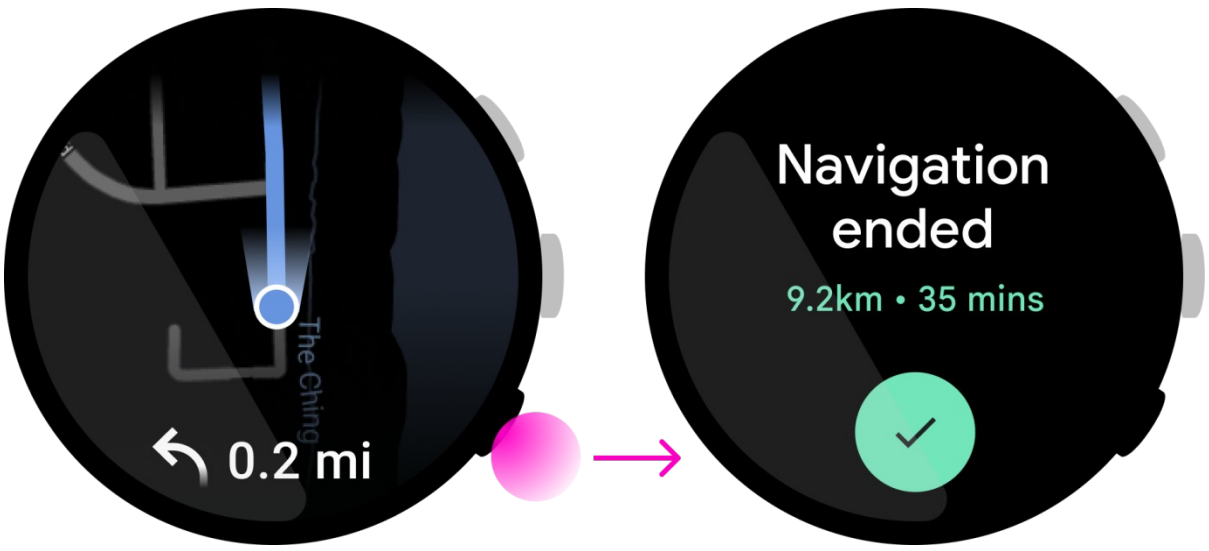



**Figure 5.** Pressing the multifunction button starts the clock, and pressing it again stops the clock.

**Multifunction buttons as alternatives**

Make multifunction button actions accessible via on-screen UI elements, as some watches don’t have multifunction buttons. But you can use multifunction buttons as alternatives for on-screen buttons.

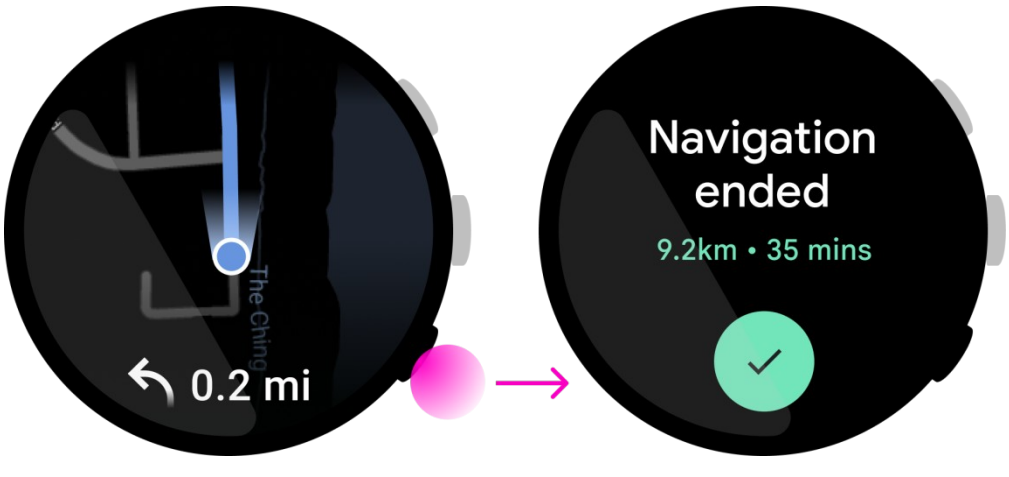



**Figure 6.** Use a multifunction button as an alternative for a start/stop action and show it as an on-screen button.

Don't use a multifunction button for an action that can’t be performed using on-screen UI elements.

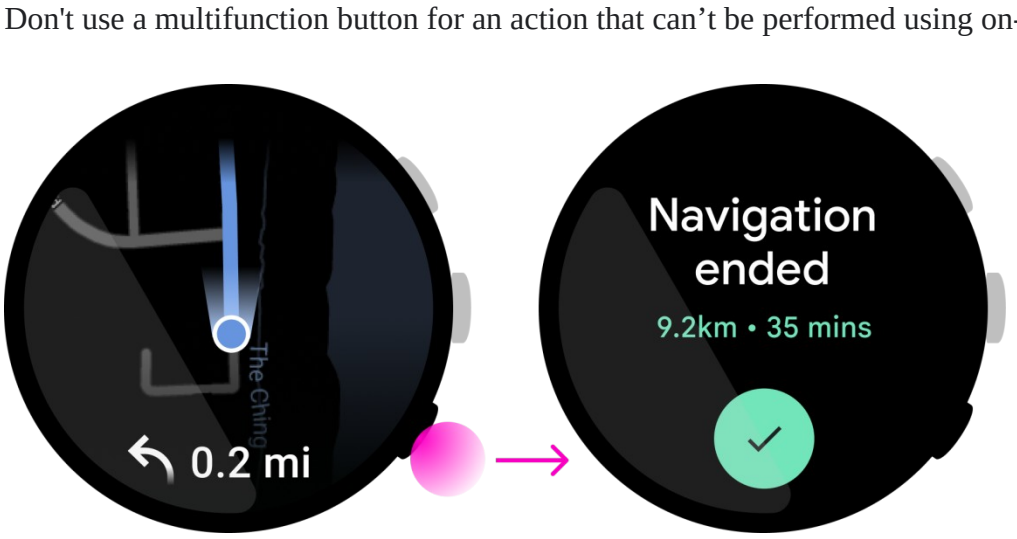



**Figure 7.** This stopwatch app uses the multifunction button to restart the stopwatch, but this isn’t clear and intuitive.

**Focus on simplicity and immediacy**

Pressing a multifunction button immediately conducts its assigned action. To prevent users from needing to look at the screen, use multifunction buttons for actions that can be completed with a single press.

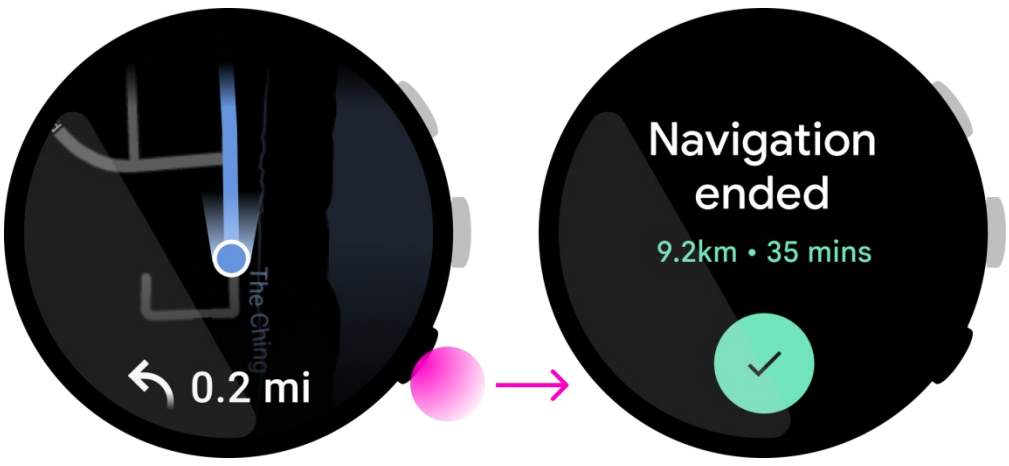



**Figure 8.** In this music app, the user can quickly pause a song using the multifunction button.

Don't use a multifunction button for complex actions.

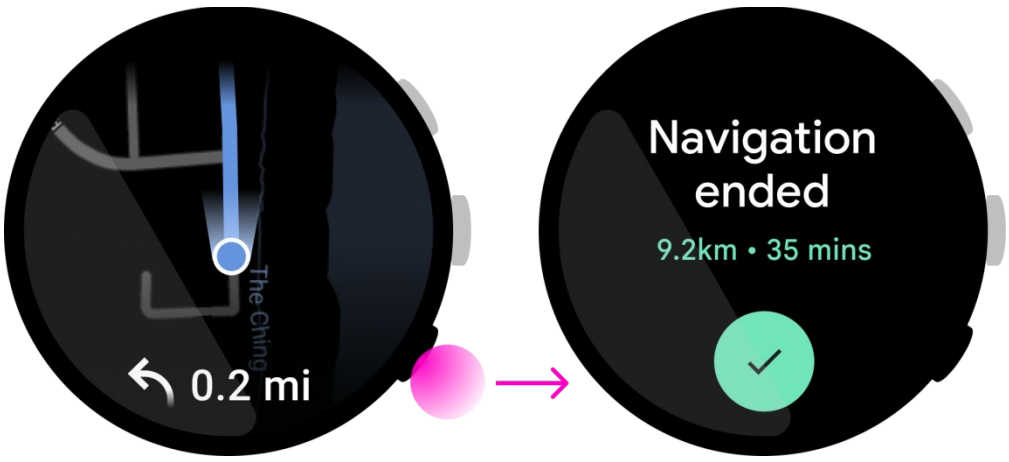



**Figure 9.** In this messaging app, pressing the button begins the action of replying, but the user may need to review the message before completing the action.

**Reversible**

Make button actions reversible. Don’t use a multifunction button to trigger a destructive action, such as deleting data or halting an ongoing activity.

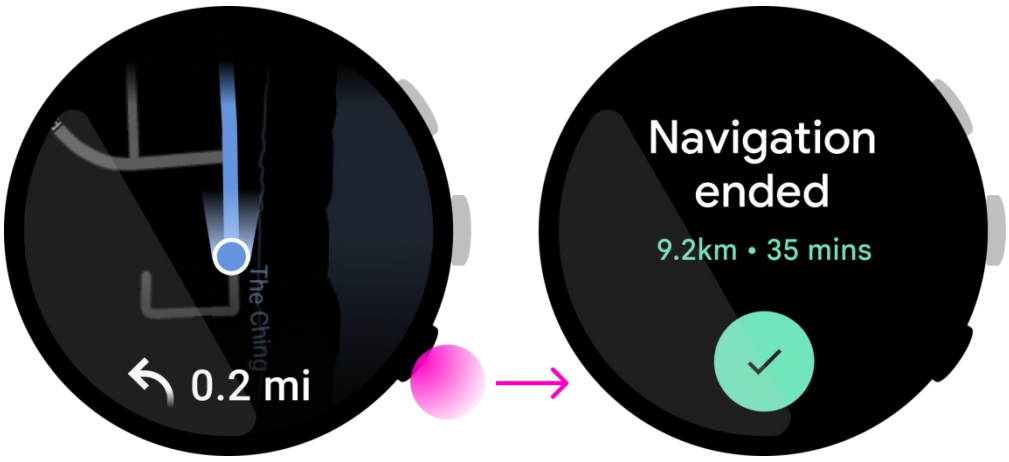



**Figure 10.** Pressing the multifunction button in this map app performs an action to “Stop navigation,” which can cause a user to lose directions at critical times.

# Unit-3

**Activity:**

The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model.

Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods

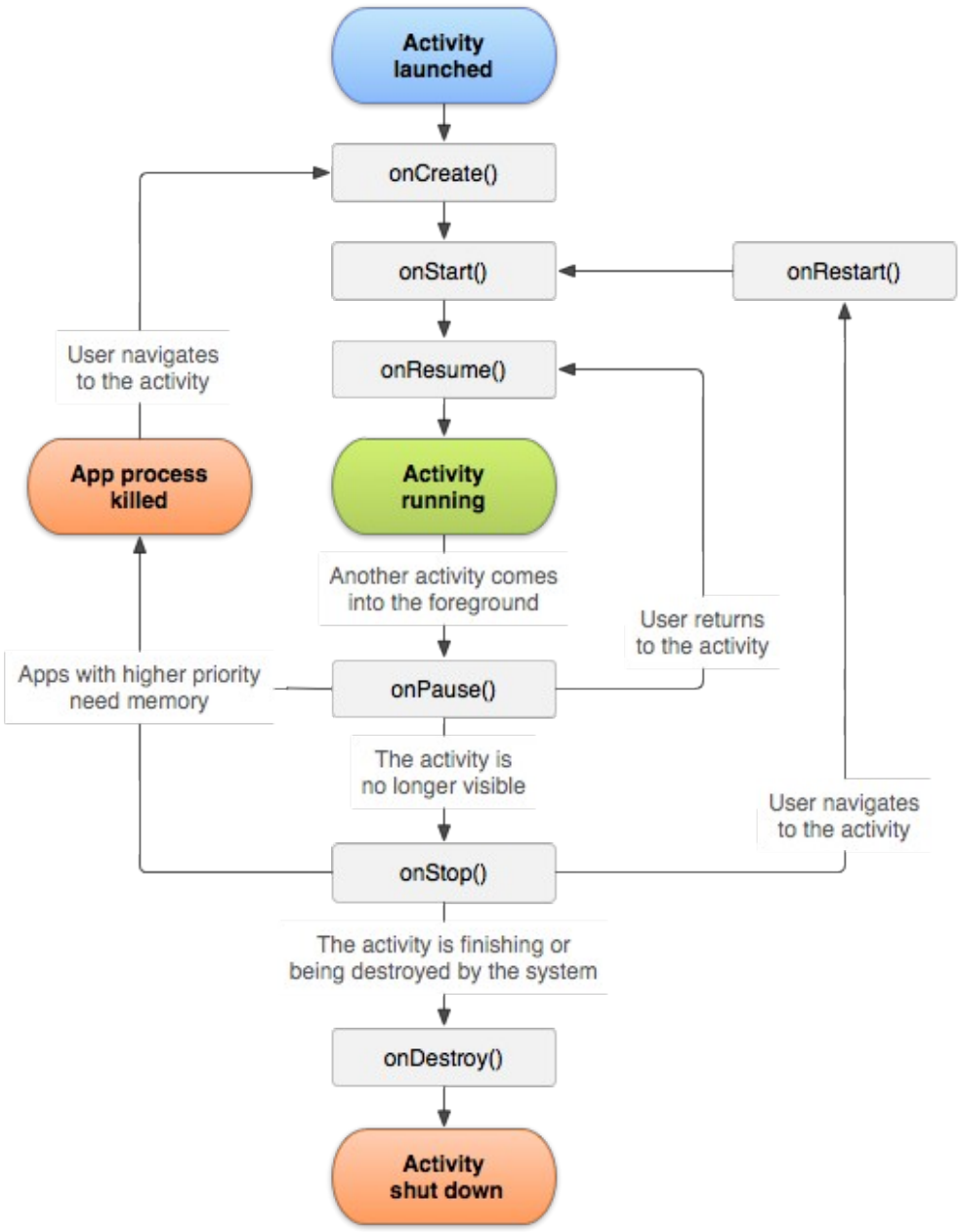
that correspond to specific stages of its lifecycle.

**Declare Activities:-**

To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element.

Activity Example:

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```




**An activity has essentially four states:**

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

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

**Activity Life Cycle:**

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

- The **entire lifetime** of an activity happens between the first call to onCreate(Bundle) through to a single final call to onDestroy(). An activity will do all setup of "global" state in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread in onDestroy().
- The **visible lifetime** of an activity happens between a call to onStart() until a corresponding call to onStop(). During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user. For example, you can register a BroadcastReceiver in onStart() to monitor for changes that impact your UI, and unregister it in onStop() when the user no longer sees what you are displaying. The onStart() and onStop() methods can be called multiple times, as the activity becomes visible and hidden to the user.

- The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause(). During this time the activity is in visible, active and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

  **Activity Syntax:**

  **syntax:**

```
<activity android:allowEmbedded=["true" | "false"]
      android:allowTaskReparenting=["true" | "false"]
      android:alwaysRetainTaskState=["true" | "false"]
      android:autoRemoveFromRecents=["true" | "false"]
      android:banner="drawable resource"
      android:clearTaskOnLaunch=["true" | "false"]
      android:colorMode=[ "hdr" | "wideColorGamut"]
      android:configChanges=["mcc", "mnc", "locale",
                  "touchscreen", "keyboard", "keyboardHidden",
                  "navigation", "screenLayout", "fontScale",
                  "uiMode", "orientation", "density",
                  "screenSize", "smallestScreenSize"]
      android:directBootAware=["true" | "false"]
      android:documentLaunchMode=["intoExisting" | "always" |
                  "none" | "never"]
      android:enabled=["true" | "false"]
      android:excludeFromRecents=["true" | "false"]
      android:exported=["true" | "false"]
      android:finishOnTaskLaunch=["true" | "false"]
      android:hardwareAccelerated=["true" | "false"]
      android:icon="drawable resource"
      android:immersive=["true" | "false"]
      android:label="string resource"
      android:launchMode=["standard" | "singleTop" |
                  "singleTask" | "singleInstance"]
      android:lockTaskMode=["normal" | "never" |
                  "if_whitelisted" | "always"]
      android:maxRecents="integer"
      android:maxAspectRatio="float"
      android:multiprocess=["true" | "false"]
      android:name="string"
      android:noHistory=["true" | "false"]
      android:parentActivityName="string"
      android:persistableMode=["persistRootOnly" |
                  "persistAcrossReboots" | "persistNever"]
      android:permission="string"
      android:process="string"
      android:relinquishTaskIdentity=["true" | "false"]
      android:resizeableActivity=["true" | "false"]
      android:screenOrientation=["unspecified" | "behind" |
                  "landscape" | "portrait" |
                  "reverseLandscape" | "reversePortrait" |
                  "sensorLandscape" | "sensorPortrait" |
                  "userLandscape" | "userPortrait" |
                  "sensor" | "fullSensor" | "nosensor" |
                  "user" | "fullUser" | "locked"]
      android:showForAllUsers=["true" | "false"]
      android:stateNotNeeded=["true" | "false"]
      android:supportsPictureInPicture=["true" | "false"]
      android:taskAffinity="string"
      android:theme="resource or theme"
      android:uiOptions=["none" | "splitActionBarWhenNarrow"]
      android:windowSoftInputMode=["stateUnspecified",
                  "stateUnchanged", "stateHidden",
                  "stateAlwaysHidden", "stateVisible",
                  "stateAlwaysVisible", "adjustUnspecified",
                  "adjustResize", "adjustPan"] >
</activity>
```

**Intent:-**

An intent is to perform an action on the screen. It is mostly used to start activity, send broadcast receiver, Start services and send message between two activities. There are two intents available in android as Implicit Intents and Explicit Intents.

An 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 Context.startService(Intent) or Context.bindService(Intent, ServiceConnection, int) to communicate with a background Service.

An Intent provides a facility for performing late runtime binding between the code 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.

**Intent Structure:**

The primary pieces of information in an intent are:

- **Action** -- The general action to be performed, such as ACTION_VIEW, ACTION_EDIT, ACTION_MAIN, etc.
- **Data** -- The data to operate on, such as a person record in the contacts database, expressed as a Uri.

Some examples of action/data pairs are:

- **ACTION_VIEW** ***content://contacts/people/1*** -- Display information about the person whose identifier is "1".
- **ACTION_DIAL** ***content://contacts/people/1*** -- Display the phone dialer with the person filled in.
- **ACTION_VIEW** ***tel:123*** -- Display the phone dialer with the given number filled in. Note how the VIEW action does what is considered the most reasonable thing for a particular URI.
- **ACTION_DIAL** ***tel:123*** -- Display the phone dialer with the given number filled in.
- **ACTION_EDIT** ***content://contacts/people/1*** -- Edit information about the person whose identifier is "1".
- **ACTION_VIEW** ***content://contacts/people/*** -- Display a list of people, which the user can browse through. This example is a typical top-level entry into the Contacts application, showing you the list of people. Selecting a particular person to view would result in a new intent { **ACTION_VIEW** ***content://contacts/people/N*** } being used to start an activity to display that person.

**Secondary attributes that you can also include with an Intent:**

- **Category** -- Gives additional information about the action to execute. For example, CATEGORY_LAUNCHER means it should appear in the Launcher as a top-level application, while CATEGORY_ALTERNATIVE means it should be included in a list of alternative actions the user can perform on a piece of data.
- **Type** -- Specifies an explicit type (a **MIME i.e copy ( type**) of the intent data. Normally the type is inferred from the data itself. By setting this attribute, you disable that evaluation and force an explicit type.
- **Component** -- Specifies an explicit name of a component class to use for the intent. Normally this is determined by looking at the other information in the intent (**the action, data-type and categories**) and matching that with a component that can handle it. If this attribute is set then none of the evaluation is performed, and this component is used exactly as is. By specifying this attribute, all of the other Intent attributes become optional.
- **Extras** -- This is a Bundle of any additional information. This can be used to provide extended information to the component. For example, if we have a action to send **an e-mail message**, we could also include **extra pieces of data here to supply a subject, body, etc**.

**Intents using some additional parameters:**

- **ACTION_MAIN with category CATEGORY_HOME** -- Launch the home screen.
- **ACTION_GET_CONTENT with MIME type** ***vnd.android.cursor.item/phone*** -- Display the list of people's phone numbers, allowing the user to browse through them and pick one and return it to the parent activity.
- **ACTION_GET_CONTENT with MIME(i.e copy) type and category CATEGORY_OPENABLE** -- Display all pickers for data that can be opened with ContentResolver.openInputStream(), allowing the user to pick one of them and then some data inside of it and returning the resulting URI to the caller. This can be used, for example, in an e-mail application to allow the user to pick some data to include as an attachment.

**Types of Intent:-**

Two types of primary forms of intents:-

- **Explicit Intents:** 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:** Implicit Intents **have not specified a component**; instead of that 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. The **intent resolution** mechanism basically revolves around **matching Intent against all of the <intent-filter> descriptions in the installed application packages.**

In the case of **broadcasts**, any BroadcastReceiver objects explicitly registered with Context#registerReceiver. More details on this can be found in the documentation on the IntentFilter class.

There are **Three pieces of information in Intent** that are used for resolution: the **action, type, and category.** Using this information, a query is done on the PackageManager for a component that can handle the intent. The appropriate **Component** is determined based on the **intent information** supplied in the AndroidManifest.xml file as follows:

- **Action**: if given, must be listed by the component as one it handles.
- **Type:** Type is retrieved from the Intent's data, if not already supplied in the Intent. If a type is included in the intent (either explicitly or implicitly in its data), then this must be listed by the component as one it handles.
- **Categories:** Categories **must all** *be* listed by the activity as it handles. If you include the categories CATEGORY_LAUNCHER and CATEGORY_ALTERNATIVE, then you will only resolve to components with an intent that lists *both* of those categories. Activities will very often need to support the CATEGORY_DEFAULT so that they can be found by Context.startActivity().

**Fragment:**

A Fragment is a **piece of an application's user interface** or behavior that can be placed in an Activity. Interaction with fragments is done through FragmentManager, which can be obtained via Activity.getFragmentManager() and Fragment.getFragmentManager(). In Fragment core, it represents a particular operation or interface that is running within a larger Activity. A Fragment is closely tied to the Activity it is in and cannot be used apart from one.

**Fragment Lifestyle:**

Fragment defines its own lifecycle. Fragment lifecycle is dependent on its activity: if the **activity is stopped**, **No fragments inside of it can be started.When the activity is destroyed, all fragments will be destroyed**. All subclasses of Fragment must include a public no-argument constructor. The framework will often re-instantiate a fragment class when needed or during state restore and needs to be able to find this constructor to instantiate it. If the no-argument constructor is not available, a runtime exception will occur in some cases during state restore.

**Fragment Lifecycle states and their relation both the fragment's lifecycle callbacks and the fragment's view Lifecycle**.

Below Figure shows each of the fragment's Lifecycle states and how they relate to both the fragment's lifecycle callbacks and the fragment's view Lifecycle. As a fragment progresses through its lifecycle, it moves upward and downward through its states.

For example, a fragment that is added to the top of the back stack moves upward from CREATED to STARTED to RESUMED. Conversely, when a fragment is popped off of the back stack, it moves downward through those states, going from RESUMED to STARTED to CREATED and finally DESTROYED.

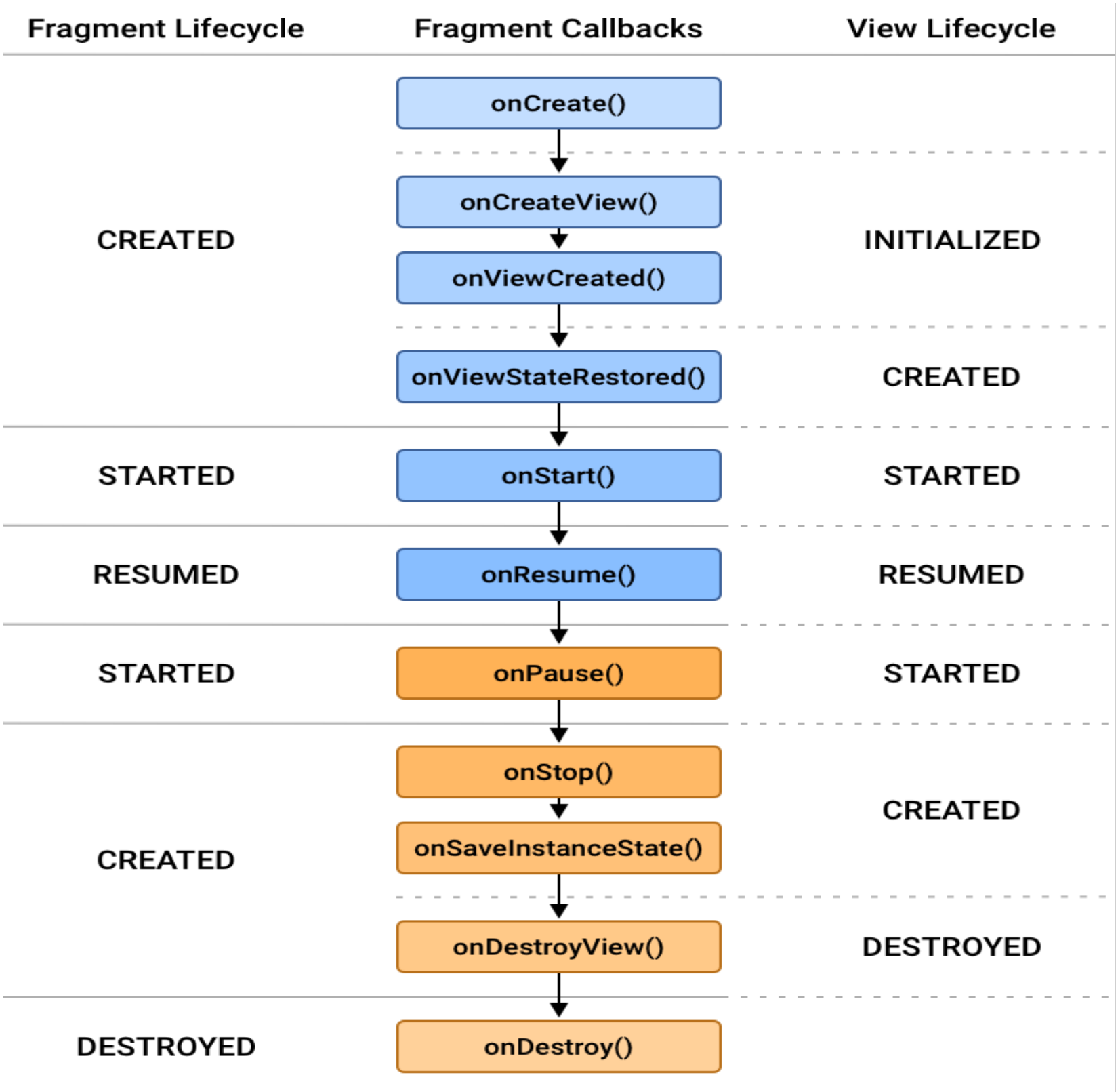



**Figure:** Fragment Lifecycle states and their relation both the fragment's lifecycle callbacks and the fragment's view Lifecycle.

**The core series of lifecycle methods that are called to bring a fragment up to resumed state (interacting with the user) are:**

- onAttach(Activity) called once the fragment is associated with its activity.
- onCreate(Bundle) called to do initial creation of the fragment.
- onCreateView(LayoutInflater, ViewGroup, Bundle) creates and returns the view hierarchy associated with the fragment.
- onActivityCreated(Bundle) tells the fragment that its activity has completed its own Activity.onCreate().
- onViewStateRestored(Bundle) tells the fragment that all of the saved state of its view hierarchy has been restored.
- onStart() makes the fragment visible to the user (based on its containing activity being started).
- onResume() makes the fragment begin interacting with the user (based on its containing activity being resumed).
- As a fragment is no longer being used, it goes through a reverse series of callbacks:
- onPause() fragment is no longer interacting with the user either because its activity is being paused or a fragment operation is modifying it in the activity.
- onStop() fragment is no longer visible to the user either because its activity is being stopped or a fragment operation is modifying it in the activity.
- onDestroyView() allows the fragment to clean up resources associated with its View.
- onDestroy() called to do final cleanup of the fragment's state.
- onDetach() called immediately prior to the fragment no longer being associated with its activity.

**Fragment Manager:**

Fragment Manager is the class responsible for performing actions on your app's fragments, such as adding, removing or replacing them and adding them to the back stack. You never interact with FragmentManager directly if you're using the Jetpack Navigation library, as it works with the FragmentManager on your behalf. Any app using fragments is using FragmentManager at some level. **Role of the FragmentManager** in relation to your activities and fragments, managing the back stack with FragmentManager and providing data and dependencies to your fragments.

**Dynamic Fragment:**

A Fragment can be a **static** part of the UI of an Activity, which means that the Fragment remains on the screen during the entire lifecycle of the Activity. However, the UI of an Activity may be more effective if it adds or removes the Fragment *dynamically* while the Activity is running.

Example of a **Dynamic Fragment** is the **DatePicker** object, which is an instance of DialogFragment, a subclass of Fragment. The date picker displays a dialog window floating on top of its Activity window when a user taps a button or an action occurs. The user can click **OK** or **Cancel** to close the Fragment.

This practical introduces the Fragment class and shows you how to include a Fragment as a static part of a UI, as well as how to use Fragment transactions to add, replace, or remove a Fragment dynamically.

## Android Menus:

Menus are a common user interface component in many types of applications. To provide a familiar and consistent user experience, you should use the Menu APIs to present user actions and other options in your activities

Android-powered devices are no longer required to provide a dedicated *Menu* button. With this change, Android apps should migrate away from a dependence on the traditional 6-item menu panel and instead provide an app bar to present common user actions.

**Options menu and app bar**

The options menu is the primary collection of menu items for an activity. It's where you should place actions that have a global impact on the app Such as "Search," "Compose email," and "Settings."

**Context menu and contextual action mode**

A context menu is a floating menu that appears when the user performs a long-click on an element. It provides actions that affect the selected content or context frame. The contextual action mode displays action items that affect the selected content in a bar at the top of the screen and allows the user to select multiple items.

**Popup menu**

A popup menu displays a list of items in a vertical list that's anchored to the view that invoked the menu. It's good for providing an overflow of actions that relate to specific content or to provide options for a second part of a command. Actions in a popup menu should **not** directly affect the corresponding content—that's what contextual actions are for. Rather, the popup menu is for extended actions that relate to regions of content in your activity.

**Defining a Menu in XML**

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in your activity's code, you should define a menu and all its items in an XML menu resource. You can then inflate the menu resource (load it as a Menu object) in your activity or fragment.

**Few Reasons for Menu Resource:**

- It's easier to visualize the menu structure in XML.
- It separates the content for the menu from your application's behavioral code.
- It allows you to create alternative menu configurations for different platform versions, screen sizes, and other configurations by leveraging the app resources framework.

**To define the menu, create an XML file inside your project's res/menu/ directory and build the menu**:

<menu>

Defines a Menu, which is a container for menu items. A <menu> element must be the root node for the file and can hold one or more <item> and <group> elements.

<item>

Creates a MenuItem, which represents a single item in a menu. This element may contain a nested <menu> element in order to create a submenu.

<group>

An optional, invisible container for <item> elements. It allows you to categorize menu items so they share properties such as active state and visibility. For more information, see the section about Creating Menu Groups.

Here's an example menu named game_menu.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/new_game"
        android:icon="@drawable/ic_new_game"
        android:title="@string/new_game"
        android:showAsAction="ifRoom"/>
   <item android:id="@+id/help"
        android:icon="@drawable/ic_help"
        android:title="@string/help" />
</menu>
```

The <item> element supports several attributes you can use to define an item's appearance and behavior. The items in the above menu include the following attributes:

android:id

A resource ID that's unique to the item, which allows the application to recognize the item when the user selects it.

android:icon

A reference to a drawable to use as the item's icon.

android:title

A reference to a string to use as the item's title.

android:showAsAction

Specifies when and how this item should appear as an action item in the app bar.

You can add a submenu to an item in any menu by adding a <menu> element as the child of an <item>. Submenus are useful when your application has a lot of functions that can be organized into topics, like items in a PC application's menu bar (File, Edit, View, etc.). For example:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/file"
        android:title="@string/file" >
      <!-- "file" submenu -->
      <menu>
         <item android:id="@+id/create_new"
              android:title="@string/create_new" />
         <item android:id="@+id/open"
              android:title="@string/open" />
      </menu>
   </item>
</menu>
```

## Creating an Options Menu:

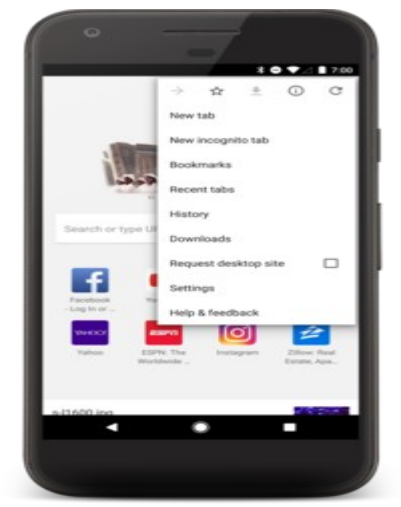



**Figure 1.** Options menu in the Browser.

The options menu is where you should include actions and other options that are relevant to the current activity context, such as "Search," "Compose email," and "Settings."

Where the items in your options menu appear on the screen depends on the version for which you've developed your application:

- If you've developed your application for **Android 2.3.x (API level 10) or lower**, the contents of your options menu appear at the top of the screen when the user presses the *Menu* button, as shown in figure 1. When opened, the first visible portion is the icon menu, which holds up to six menu items. If your menu includes more than six items, Android places the sixth item and the rest into the overflow menu, which the user can open by selecting *More*.
- If you've developed your application for **Android 3.0 (API level 11) and higher**, items from the options menu are available in the app bar. By default, the system places all items in the action overflow, which the user can reveal with the action overflow icon on the right side of the app bar (or by pressing the device *Menu* button, if available). To enable quick access to important actions, you can promote a few items to appear in the app bar by adding android:showAsAction="ifRoom" to the corresponding <item> elements (see figure 2).

  For more information about action items and other app bar behaviors, see the Adding the App Bar training class.


**Figure 2.** The Google Sheets app, showing several buttons, including the action overflow button.

You can declare items for the options menu from either your Activity subclass or a Fragment subclass. If both your activity and fragment(s) declare items for the options menu, they are combined in the UI. The activity's items appear first, followed by those of each fragment in the order in which each fragment is added to the activity. If necessary, you can re-order the menu items with the android:orderInCategory attribute in each <item> you need to move.

To specify the options menu for an activity, override onCreateOptionsMenu() (fragments provide their own onCreateOptionsMenu() callback). In this method, you can inflate your menu resource (defined in XML) into the Menu provided in the callback. For example:

KotlinJava

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.game_menu, menu)
    return true
}
```

You can also add menu items using add() and retrieve items with findItem() to revise their properties with MenuItem APIs.

If you've developed your application for Android 2.3.x and lower, the system calls onCreateOptionsMenu() to create the options menu when the user opens the menu for the first time. If you've developed for Android 3.0 and higher, the system calls onCreateOptionsMenu() when starting the activity, in order to show items to the app bar.

**Handling click events**

When the user selects an item from the options menu (including action items in the app bar), the system calls your activity's onOptionsItemSelected() method. This method passes the MenuItem selected. You can identify the item by calling getItemId(), which returns the unique ID for the menu item (defined by the android:id attribute in the menu resource or with an integer given to the add() method). You can match this ID against known menu items to perform the appropriate action. For example:

KotlinJava

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    // Handle item selection
    return when (item.itemId) {
        R.id.new_game -> {
            newGame()
            true
        }
        R.id.help -> {
            showHelp()
            true
        }
        else -> super.onOptionsItemSelected(item)
    }
}
```

When you successfully handle a menu item, return true. If you don't handle the menu item, you should call the superclass implementation of onOptionsItemSelected() (the default implementation returns false).

If your activity includes fragments, the system first calls onOptionsItemSelected() for the activity then for each fragment (in the order each fragment was added) until one returns true or all fragments have been called.

**Tip:** Android 3.0 adds the ability for you to define the on-click behavior for a menu item in XML, using the **android:onClick** attribute. The value for the attribute must be the name of a method defined by the activity using the menu. The method must be public and accept a single **MenuItem** parameter—when the system calls this method, it passes the menu item selected. For more information and an example, see the Menu Resource document.

**Tip:** If your application contains multiple activities and some of them provide the same options menu, consider creating an activity that implements nothing except the **onCreateOptionsMenu()** and **onOptionsItemSelected()** methods. Then extend this class for each activity that should share the same options menu. This way, you can manage one set of code for handling menu actions and each descendant class inherits the menu behaviors. If you want to add menu items to one of the descendant activities, override **onCreateOptionsMenu()** in that activity. Call **super.onCreateOptionsMenu(menu)** so the original menu items are created, then add new menu items with **menu.add()**. You can also override the super class's behavior for individual menu items.

**Changing menu items at runtime**

After the system calls onCreateOptionsMenu(), it retains an instance of the Menu you populate and will not call onCreateOptionsMenu() again unless the menu is invalidated for some reason. However, you should use onCreateOptionsMenu() only to create the initial menu state and not to make changes during the activity lifecycle.

If you want to modify the options menu based on events that occur during the activity lifecycle, you can do so in the onPrepareOptionsMenu() method. This method passes you the Menu object as it currently exists so you can modify it, such as add, remove, or disable items. (Fragments also provide an onPrepareOptionsMenu() callback.)

On Android 2.3.x and lower, the system calls onPrepareOptionsMenu() each time the user opens the options menu (presses the *Menu* button).

On Android 3.0 and higher, the options menu is considered to always be open when menu items are presented in the app bar. When an event occurs and you want to perform a menu update, you must call invalidateOptionsMenu() to request that the system call onPrepareOptionsMenu().

**Note:** You should never change items in the options menu based on the **`View`** currently in focus. When in touch mode (when the user is not using a trackball or d-pad), views cannot take focus, so you should never use focus as the basis for modifying items in the options menu. If you want to provide menu items that are context-sensitive to a **`View`**, use a Context Menu.

## Creating Contextual Menus:

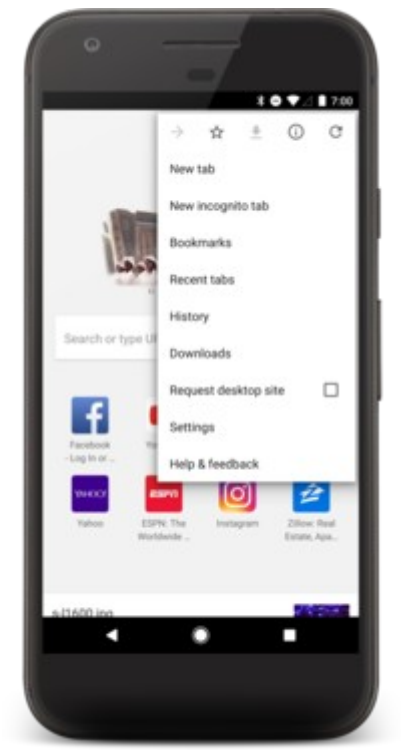



**Figure 3.** Screenshots of a floating context menu (left) and the contextual action bar (right).

A contextual menu offers actions that affect a specific item or context frame in the UI. You can provide a context menu for any view, but they are most often used for items in a ListView, GridView, or other view collections in which the user can perform direct actions on each item.

There are two ways to provide contextual actions:

- In a floating context menu. A menu appears as a floating list of menu items (similar to a dialog) when the user performs a long-click (press and hold) on a view that declares support for a context menu. Users can perform a contextual action on one item at a time.
- In the contextual action mode. This mode is a system implementation of ActionMode that displays a *contextual action bar* at the top of the screen with action items that affect the selected item(s). When this mode is active, users can perform an action on multiple items at once (if your app allows it).

**Note:** The contextual action mode is available on Android 3.0 (API level 11) and higher and is the preferred technique for displaying contextual actions when available. If your app supports versions lower than 3.0 then you should fall back to a floating context menu on those devices.

### Creating a floating context menu

To provide a floating context menu:

1. Register the View to which the context menu should be associated by calling registerForContextMenu() and pass it the View.

   If your activity uses a ListView or GridView and you want each item to provide the same context menu, register all items for a context menu by passing the ListView or GridView to registerForContextMenu().

2. Implement the onCreateContextMenu() method in your Activity or Fragment.

   When the registered view receives a long-click event, the system calls your onCreateContextMenu() method. This is where you define the menu items, usually by inflating a menu resource. For example:

   KotlinJava

```
override fun onCreateContextMenu(menu: ContextMenu, v: View,
                menuInfo: ContextMenu.ContextMenuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo)
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.context_menu, menu)
```

```
}
```

MenuInflater allows you to inflate the context menu from a menu resource. The callback method parameters include the View that the user selected and a ContextMenu.ContextMenuInfo object that provides additional information about the item selected. If your activity has several views that each provide a different context menu, you might use these parameters to determine which context menu to inflate.

3. Implement onContextItemSelected().

   When the user selects a menu item, the system calls this method so you can perform the appropriate action. For example:

   KotlinJava

```
override fun onContextItemSelected(item: MenuItem): Boolean {
    val info = item.menuInfo as AdapterView.AdapterContextMenuInfo
    return when (item.itemId) {
        R.id.edit -> {
            editNote(info.id)
            true
        }
        R.id.delete -> {
            deleteNote(info.id)
            true
        }
        else -> super.onContextItemSelected(item)
    }
}
```

The getItemId() method queries the ID for the selected menu item, which you should assign to each menu item in XML using the android:id attribute, as shown in the section about Defining a Menu in XML.

When you successfully handle a menu item, return true. If you don't handle the menu item, you should pass the menu item to the superclass implementation. If your activity includes fragments, the activity receives this callback first. By calling the superclass when unhandled, the system passes the event to the respective callback method in each fragment, one at a time (in the order each fragment was added) until true or false is returned. (The default implementation for Activity and android.app.Fragment return false, so you should always call the superclass when unhandled.)

### Using the contextual action mode

The contextual action mode is a system implementation of ActionMode that focuses user interaction toward performing contextual actions. When a user enables this mode by selecting an item, a *contextual action bar* appears at the top of the screen to present actions the user can perform on the currently selected item(s). While this mode is enabled, the user can select multiple items (if you allow it), deselect items, and continue to navigate within the activity (as much as you're willing to allow). The action mode is disabled and the contextual action bar disappears when the user deselects all items, presses the BACK button, or selects the *Done* action on the left side of the bar.

For views that provide contextual actions, you should usually invoke the contextual action mode upon one of two events (or both):

- The user performs a long-click on the view.
- The user selects a checkbox or similar UI component within the view.

How your application invokes the contextual action mode and defines the behavior for each action depends on your design. There are basically two designs:

- For contextual actions on individual, arbitrary views.
- For batch contextual actions on groups of items in a ListView or GridView (allowing the user to select multiple items and perform an action on them all).

The following sections describe the setup required for each scenario.

#### *Enabling the contextual action mode for individual views*

If you want to invoke the contextual action mode only when the user selects specific views, you should:

1. Implement the ActionMode.Callback interface. In its callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other lifecycle events for the action mode.

2. Call startActionMode() when you want to show the bar (such as when the user long-clicks the view).

   For example:

1. Implement the ActionMode.Callback interface:

   KotlinJava

```
private val actionModeCallback = object : ActionMode.Callback {
    // Called when the action mode is created; startActionMode() was called
    override fun onCreateActionMode(mode: ActionMode, menu: Menu): Boolean {
        // Inflate a menu resource providing context menu items
        val inflater: MenuInflater = mode.menuInflater
        inflater.inflate(R.menu.context_menu, menu)
        return true
    }

    // Called each time the action mode is shown. Always called after onCreateActionMode, but
    // may be called multiple times if the mode is invalidated.
    override fun onPrepareActionMode(mode: ActionMode, menu: Menu): Boolean {
        return false // Return false if nothing is done
    }

    // Called when the user selects a contextual menu item
    override fun onActionItemClicked(mode: ActionMode, item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.menu_share -> {
                shareCurrentItem()
                mode.finish() // Action picked, so close the CAB
                true
            }
            else -> false
        }
    }

    // Called when the user exits the action mode
    override fun onDestroyActionMode(mode: ActionMode) {
        actionMode = null
    }
}
```

   Notice that these event callbacks are almost exactly the same as the callbacks for the options menu, except each of these also pass the ActionMode object associated with the event. You can use ActionMode APIs to make various changes to the CAB, such as revise the title and subtitle with setTitle() and setSubtitle() (useful to indicate how many items are selected).

   Also notice that the above sample sets the actionMode variable to null when the action mode is destroyed. In the next step, you'll see how it's initialized and how saving the member variable in your activity or fragment can be useful.

2. Call startActionMode() to enable the contextual action mode when appropriate, such as in response to a long-click on a View:

   KotlinJava

```
someView.setOnLongClickListener { view ->
    // Called when the user long-clicks on someView
    when (actionMode) {
        null -> {
            // Start the CAB using the ActionMode.Callback defined above
            actionMode = activity?.startActionMode(actionModeCallback)
            view.isSelected = true
            true
        }
        else -> false
    }
}
```

   When you call startActionMode(), the system returns the ActionMode created. By saving this in a member variable, you can make changes to the contextual action bar in response to other events. In the above sample, the ActionMode is used to ensure that the ActionMode instance is not recreated if it's already active, by checking whether the member is null before starting the action mode.

**Enabling batch contextual actions in a ListView or GridView:-**

If you have a collection of items in a ListView or GridView (or another extension of AbsListView) and want to allow users to perform batch actions, you should:

- Implement the AbsListView.MultiChoiceModeListener interface and set it for the view group with setMultiChoiceModeListener(). In the listener's callback methods, you can specify the actions for the contextual action bar, respond to click events on action items, and handle other callbacks inherited from the ActionMode.Callback interface.
- Call setChoiceMode() with the CHOICE_MODE_MULTIPLE_MODAL argument.

For example:

KotlinJava

```
val listView: ListView = getListView()
with(listView) {
   choiceMode = ListView.CHOICE_MODE_MULTIPLE_MODAL
   setMultiChoiceModeListener(object : AbsListView.MultiChoiceModeListener {
      override fun onItemCheckedStateChanged(mode: ActionMode, position: Int,
                                             id: Long, checked: Boolean) {
         // Here you can do something when items are selected/de-selected,
         // such as update the title in the CAB
      }

      override fun onActionItemClicked(mode: ActionMode, item: MenuItem): Boolean {
         // Respond to clicks on the actions in the CAB
         return when (item.itemId) {
            R.id.menu_delete -> {
               deleteSelectedItems()
               mode.finish() // Action picked, so close the CAB
               true
            }
            else -> false
         }
      }

      override fun onCreateActionMode(mode: ActionMode, menu: Menu): Boolean {
         // Inflate the menu for the CAB
         val menuInflater: MenuInflater = mode.menuInflater
         menuInflater.inflate(R.menu.context, menu)
         return true
      }

      override fun onDestroyActionMode(mode: ActionMode) {
         // Here you can make any necessary updates to the activity when
         // the CAB is removed. By default, selected items are deselected/unchecked.
      }

      override fun onPrepareActionMode(mode: ActionMode, menu: Menu): Boolean {
         // Here you can perform updates to the CAB due to
         // an <code><a href="/reference/android/view/ActionMode.html#invalidate()">invalidate()</a></code> request
         return false
      }
   })
}
```

That's it. Now when the user selects an item with a long-click, the system calls the onCreateActionMode() method and displays the contextual action bar with the specified actions. While the contextual action bar is visible, users can select additional items.

In some cases in which the contextual actions provide common action items, you might want to add a checkbox or a similar UI element that allows users to select items, because they might not discover the long-click behavior. When a user selects the checkbox, you can invoke the contextual action mode by setting the respective list item to the checked state with setItemChecked().

## Creating a Popup Menu

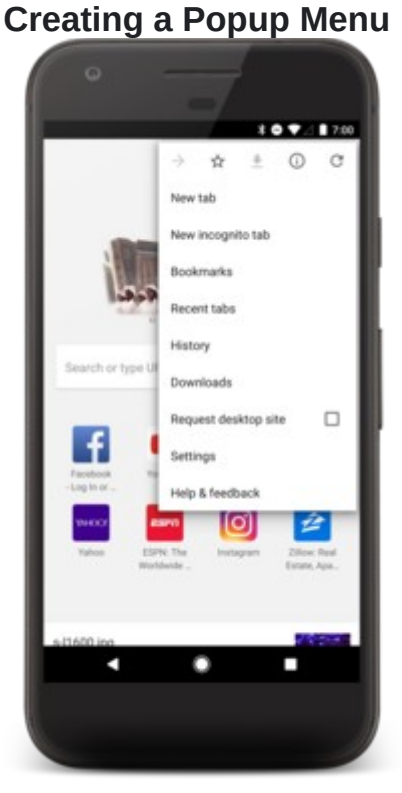



**Figure 4.** A popup menu in the Gmail app, anchored to the overflow button at the top-right.

A PopupMenu is a modal menu anchored to a View. It appears below the anchor view if there is room, or above the view otherwise. It's useful for:

- Providing an overflow-style menu for actions that *relate to* specific content (such as Gmail's email headers, shown in figure 4).

  **Note:** This is not the same as a context menu, which is generally for actions that *affect* selected content. For actions that affect selected content, use the contextual action mode or floating context menu.

- Providing a second part of a command sentence (such as a button marked "Add" that produces a popup menu with different "Add" options).
- Providing a drop-down similar to Spinner that does not retain a persistent selection.

  **Note:** **PopupMenu** is available with API level 11 and higher.

  If you define your menu in XML, here's how you can show the popup menu:

1. Instantiate a PopupMenu with its constructor, which takes the current application Context and the View to which the menu should be anchored.
2. Use MenuInflater to inflate your menu resource into the Menu object returned by PopupMenu.getMenu().
3. Call PopupMenu.show().

   For example, here's a button with the android:onClick attribute that shows a popup menu:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_overflow_holo_dark"
    android:contentDescription="@string/descr_overflow_button"
    android:onClick="showPopup" />
```

   The activity can then show the popup menu like this:

KotlinJava

```
fun showPopup(v: View) {
    val popup = PopupMenu(this, v)
    val inflater: MenuInflater = popup.menuInflater
    inflater.inflate(R.menu.actions, popup.menu)
    popup.show()
}
```

In API level 14 and higher, you can combine the two lines that inflate the menu with PopupMenu.inflate().

The menu is dismissed when the user selects an item or touches outside the menu area. You can listen for the dismiss event using PopupMenu.OnDismissListener.

### Handling click events

To perform an action when the user selects a menu i

# Relative Layout

RelativeLayout is a view group that displays child views in relative positions. The position o
be specified as relative to sibling elements (such as to the left-of or below another view) or
relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).

**Note:** For better performance and tooling support, you should instead build your layout with Constra

A RelativeLayout is a very powerful utility for designing a user interface because it can elim
view groups and keep your layout hierarchy flat, which improves performance. If you find y
several nested LinearLayout groups, you may be able to replace them with a single Relativ

## Positioning Views

RelativeLayout lets child views specify their position relative to the parent view or to each o
by ID). So you can align two elements by right border, or make one below another, centere
screen, centered left, and so on. By default, all child views are drawn at the top-left of the l
must define the position of each view using the various layout properties available
from RelativeLayout.LayoutParams.

Some of the many layout properties available to views in a RelativeLayout include:

**Relative Layout:**

RelativeLayout is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).

**Note:** For better performance and tooling support, you should instead build your layout with ConstraintLayout.

A RelativeLayout is a very powerful utility for designing a user interface because it can eliminate nested view groups and keep your layout hierarchy flat, which improves performance. If you find yourself using several nested LinearLayout groups, you may be able to replace them with a single RelativeLayout.

# Positioning Views

RelativeLayout lets child views specify their position relative to the parent view or to each other (specified by ID). So you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from RelativeLayout.LayoutParams.

Some of the many layout properties available to views in a RelativeLayout include:

android:layout_alignParentTop
: If "true", makes the top edge of this view match the top edge of the parent.

android:layout_centerVertical
: If "true", centers this child vertically within its parent.

android:layout_below

: Positions the top edge of this view below the view specified with a resource ID.

android:layout_toRightOf

: Positions the left edge of this view to the right of the view specified with a resource ID.

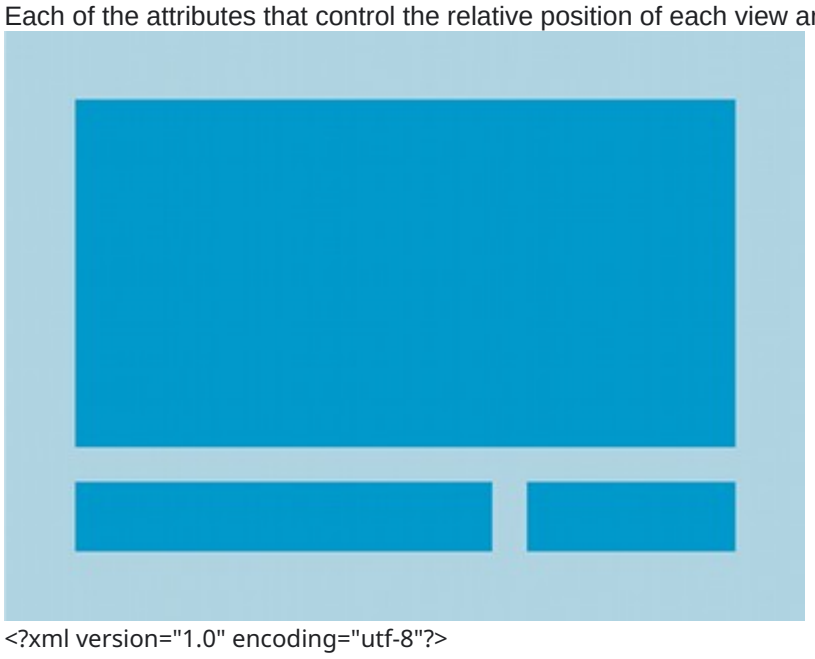
These are just a few examples. All layout attributes are documented at RelativeLayout.LayoutParams.

The value for each layout property is either a boolean to enable a layout position relative to the parent RelativeLayout or an ID that references another view in the layout against which the view should be positioned.

In your XML layout, dependencies against other views in the layout can be declared in any order. For example, you can declare that "view1" be positioned below "view2" even if "view2" is the last view declared in the hierarchy. The example below demonstrates such a scenario.

# Example

Each of the attributes that control the relative position of each view are emphasized.



```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```

# XML attributes

## android:gravity

`android:gravity`

Specifies how an object should position its content, on both the X and Y axes, within its own bounds.

Must be one or more (separated by '|') of the following constant values.

| Constant | Value | Description |
|---|---|---|
| bottom | 50 | Push object to the bottom of its container, not changing its size. |
| center | 11 | Place the object in the center of its container in both the vertical and horizontal axis, not changing its size. |
| center_horizontal | 1 | Place object in the horizontal center of its container, not changing its size. |
| center_vertical | 10 | Place object in the vertical center of its container, not changing its size. |
| clip_horizontal | 8 | Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges. |
| clip_vertical | 80 | Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges. |
| end | 800005 | Push object to the end of its container, not changing its size. |
| fill | 77 | Grow the horizontal and vertical size of the object if needed so it completely fills its container. |
| fill_horizontal | 7 | Grow the horizontal size of the object if needed so it completely fills its container. |
| fill_vertical | 70 | Grow the vertical size of the object if needed so it completely fills its container. |
| left | 3 | Push object to the left of its container, not changing its size. |
| right | 5 | Push object to the right of its container, not changing its size. |
| start | 800003 | Push object to the beginning of its container, not changing its size. |
| top | 30 | Push object to the top of its container, not changing its size. |

## android:ignoreGravity

`android:ignoreGravity`

Indicates what view should not be affected by gravity.

May be a reference to another resource, in the form "@[+][*package*:]*type*/*name*" or a theme attribute in the form "?[*package*:]*type*/*name*".

# Constants

## ABOVE

Added in API level 1

`static val ABOVE: Int`

Rule that aligns a child's bottom edge with another child's top edge.
Value: 2

## ALIGN_BASELINE

Added in API level 1

`static val ALIGN_BASELINE: Int`

Rule that aligns a child's baseline with another child's baseline.
Value: 4

## ALIGN_BOTTOM

Added in API level 1

`static val ALIGN_BOTTOM: Int`

Rule that aligns a child's bottom edge with another child's bottom edge.
Value: 8

## ALIGN_END

Added in API level 17

```
static val ALIGN_END: Int
```

Rule that aligns a child's end edge with another child's end edge.
Value: 19

## ALIGN_LEFT

Added in API level 1

```
static val ALIGN_LEFT: Int
```

Rule that aligns a child's left edge with another child's left edge.
Value: 5

## ALIGN_PARENT_BOTTOM

Added in API level 1

```
static val ALIGN_PARENT_BOTTOM: Int
```

Rule that aligns the child's bottom edge with its RelativeLayout parent's bottom edge.
Value: 12

## ALIGN_PARENT_END

Added in API level 17

```
static val ALIGN_PARENT_END: Int
```

Rule that aligns the child's end edge with its RelativeLayout parent's end edge.
Value: 21

## ALIGN_PARENT_LEFT

Added in API level 1

```
static val ALIGN_PARENT_LEFT: Int
```

Rule that aligns the child's left edge with its RelativeLayout parent's left edge.
Value: 9

## ALIGN_PARENT_RIGHT

Added in API level 1

```
static val ALIGN_PARENT_RIGHT: Int
```

Rule that aligns the child's right edge with its RelativeLayout parent's right edge.
Value: 11

## ALIGN_PARENT_START

Added in API level 17

```
static val ALIGN_PARENT_START: Int
```

Rule that aligns the child's start edge with its RelativeLayout parent's start edge.
Value: 20

## ALIGN_PARENT_TOP

Added in API level 1

```
static val ALIGN_PARENT_TOP: Int
```

Rule that aligns the child's top edge with its RelativeLayout parent's top edge.
Value: 10

## ALIGN_RIGHT

Added in API level 1

```
static val ALIGN_RIGHT: Int
```

Rule that aligns a child's right edge with another child's right edge.
Value: 7

## ALIGN_START

Added in API level 17

```
static val ALIGN_START: Int
```

Rule that aligns a child's start edge with another child's start edge.
Value: 18

## ALIGN_TOP

Added in API level 1

```
static val ALIGN_TOP: Int
```

Rule that aligns a child's top edge with another child's top edge.
Value: 6

## BELOW

Added in API level 1

```
static val BELOW: Int
```

Rule that aligns a child's top edge with another child's bottom edge.
Value: 3

## CENTER_HORIZONTAL

Added in API level 1

```
static val CENTER_HORIZONTAL: Int
```

Rule that centers the child horizontally with respect to the bounds of its RelativeLayout parent.
Value: 14

## CENTER_IN_PARENT

Added in API level 1

```
static val CENTER_IN_PARENT: Int
```

Rule that centers the child with respect to the bounds of its RelativeLayout parent.
Value: 13

## CENTER_VERTICAL

Added in API level 1

```
static val CENTER_VERTICAL: Int
```

Rule that centers the child vertically with respect to the bounds of its RelativeLayout parent.
Value: 15

## END_OF

Added in API level 17

```
static val END_OF: Int
```

Rule that aligns a child's start edge with another child's end edge.
Value: 17

## LEFT_OF

Added in API level 1

```
static val LEFT_OF: Int
```

Rule that aligns a child's right edge with another child's left edge.
Value: `0`

## RIGHT_OF

Added in API level 1

```
static val RIGHT_OF: Int
```

Rule that aligns a child's left edge with another child's right edge.
Value: `1`

## START_OF

Added in API level 17

```
static val START_OF: Int
```

Rule that aligns a child's end edge with another child's start edge.
Value: `16`

## TRUE

Added in API level 1

```
static val TRUE: Int
```

Value: `-1`

# Public constructors

## RelativeLayout

Added in API level 1

```
RelativeLayout(context: Context!)
```

## RelativeLayout

Added in API level 1

```
RelativeLayout(
    context: Context!,
    attrs: AttributeSet!)
```

## RelativeLayout

Added in API level 1

```
RelativeLayout(
    context: Context!,
    attrs: AttributeSet!,
    defStyleAttr: Int)
```

## RelativeLayout

Added in API level 21

```
RelativeLayout(
    context: Context!,
    attrs: AttributeSet!,
    defStyleAttr: Int,
    defStyleRes: Int)
```

# ViewGroup:

A View Group is a special view that can contain other views (called children.) The view group is the base class for layouts and views containers. This class also defines the View Group. Layout Params class which serves as the base class for layouts parameters.

# Layouts:

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 View Group objects. A View usually draws something the user can see and interact with. Whereas a View Group is an invisible container that defines the layout structure for View and other View Group objects.

**View** objects are usually called "widgets" and can be one of many subclasses, such as Button or Text View.

**View Group** objects are usually called "layouts" can be one of many types that provide a different layout structure, such Linear Layout or Constraint Layout.

**Declaration of  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.

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI.

## Relative Layout:-

Relative Layout is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent Relative Layout area (such as aligned to the bottom, left or center).

A Relative Layout is a very powerful utility for designing a user interface because it can eliminate nested view groups and keep your layout hierarchy flat, which improves performance.

## Unit _IV

# Adaptor

**Adaptor: -**

An Adapter object acts as a bridge between an AdapterView and the underlying data for that view. The Adapter provides access to the data items. The Adapter is also responsible for making a View for each item in the data set.

**Syntax of Adaptor:**

```
public interface Adapter
 android.widget.Adapter
Known indirect subclasses
```

ArrayAdapter<T>, BaseAdapter, CursorAdapter, HeaderViewListAdapter, ListAdapter, ResourceCursorAdapter, SimpleAdapter, SimpleCursorAdapter, SpinnerAdapter,
ThemedSpinnerAdapter, WrapperListAdapter.

**Types of Adapter:-**

- **Array Adaptor:--**

This adapter is used to provide views for an AdapterView, Returns a view for each object in a collection of data objects you provide, and can be used with list-based user interface widgets such as ListView or Spinner. By default, the array adapter creates a view by calling Object#toString() on each data object in the collection you provide and places the result in a TextView. You may also customize what type of view is used for the data object in the collection. To customize what type of view is used for the data object, override getView(int, android.view.View, android.view.ViewGroup) and inflate a view resource.

### Syntax of Array Adaptor:

```
public class ArrayAdapter
extends BaseAdapter implements Filterable, ThemedSpinnerAdapter
```

java.lang.Object
 android.widget.BaseAdapter
  android.widget.ArrayAdapter<T>

- **Spinner Adaptor:**

It extended Adapter that is the bridge between a Spinner and its data. A spinner adapter allows to define two different views: one that shows the data in the spinner itself and one that shows the data in the drop down list when the spinner is pressed.

**Syntax of Spinner Adaptor:**

public interface SpinnerAdapter
implements Adapter
android.widget.SpinnerAdapter
Known indirect subclasses
ArrayAdapter<T>, BaseAdapter, CursorAdapter, ResourceCursorAdapter, SimpleAdapter, SimpleCursorAdapter, ThemedSpinner
Adapter

### Base Adaptor:--

- **Base Adaptor:** Common base class of common implementation for an Adapter that can be used in both ListView (by implementing the specialized ListAdapter interface) and Spinner (by implementing the specialized SpinnerAdapter interface).

**Syntax of Base adaptor:**

```
public abstract class BaseAdapter
extends Object implements ListAdapter, SpinnerAdapter
```

Java.lang.Object:

Known direct subclasses
ArrayAdapter<T>, CursorAdapter, SimpleAdapter,
Known indirect subclasses
ResourceCursorAdapter, SimpleCursorAdapter,

- **Array List adapter:**

RecyclerView.Adapter base class for presenting List data in a RecyclerView, including computing diffs between Lists on a background thread.

This class is a convenience wrapper around AsyncListDiffer that implements Adapter common default behavior for item access and counting.

While using a LiveData<List> is an easy way to provide data to the adapter, it isn't required - you can use submitList(List) when new lists are available.  Advanced users that wish for more control over adapter behavior, or to provide a specific base class should refer to AsyncListDiffer, which provides  custom mapping from diff events to adapter positions.

**Syntax of Array List adaptor:**

```
public abstract class ListAdapter
extends Adapter<VH extends RecyclerView.ViewHolder>
```

Java.lang.Object
  androidx.recyclerview.widget.RecyclerView.Adapter<VH extends androidx.recyclerview.widget.RecyclerView.ViewHolder>
  androidx.recyclerview.widget.ListAdapter<T, VH extends androidx.recyclerview.widget.RecyclerView.ViewHolder>

---

## View:--

This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for *widgets*, which are used to create interactive UI components (buttons, text fields etc.) The ViewGroup subclass is the base class for *layouts* which are invisible containers that hold other Views (or other ViewGroups) and define their layout properties.

**Types of View:**

**Grid View:**

A view that shows items in two-dimensional scrolling grid. The items in the grid come from the `ListAdapter` associated with this view.

Syntax of Grid View: -

```
public class GridView
 extends AbsListView
```

Java.lang.Object
  android.view.View
   android.view.ViewGroup
   android.widget.AdapterView<android.widget.ListAdapter>
   android.widget.AbsListView
    android.widget.GridView.

**Web View :-**

In most cases, we recommend using a standard web browser, like Chrome, to deliver content to the user. To learn more about web browsers, read the guide on invoking a browser with an intent. WebView objects allow you to display web content as part of your activity layout, but lack some of the  features of fully-developed browsers. A WebView is useful when you need increased control over the UI and advanced configuration options that will allow you to embed web pages in a specially-designed environment for your app. To learn more about WebView and alternatives for serving web content, read the documentation on Web-based content.

**Syntax of WebView:**

**ScrollView**
**Search Voie**
**TabHost**
**DynamicListView**
**ExpandedList view**

**SQLite Spinner in Android:**

To create a spinner that populates its options from an SQLite database in Android, you'll need to follow these steps:

**1. Create an SQLite Database:**

First, create an SQLite database to store the data you want to display in the spinner. You can use the SQLiteOpenHelper class to manage database creation and versio management. Here's a simplified example of a database helper class:

Java

```java
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String DATABASE_NAME = "mydb.db";
    private static final int DATABASE_VERSION = 1;

    public static final String TABLE_NAME = "items";
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_NAME = "name";

    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String createTableQuery = "CREATE TABLE " + TABLE_NAME + " (" +
                COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
                COLUMN_NAME + " TEXT);";
        db.execSQL(createTableQuery);
    }

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

### 2. Insert Data into the Database:

Insert the data you want to display in the spinner into the database. You can do this in your activity or a dedicated data initialization method.

Java

```
DatabaseHelper dbHelper = new DatabaseHelper(this);
SQLiteDatabase db = dbHelper.getWritableDatabase();

// Insert sample data
ContentValues values = new ContentValues();
values.put(DatabaseHelper.COLUMN_NAME, "Item 1");
db.insert(DatabaseHelper.TABLE_NAME, null, values);

values.clear();
values.put(DatabaseHelper.COLUMN_NAME, "Item 2");
db.insert(DatabaseHelper.TABLE_NAME, null, values);

// Insert more items as needed

db.close();
```

### 3. Create the Spinner in Your Layout XML:

In your layout XML file, create a Spinner widget that will display the data from the database.

XML

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp" />
```

### 4. Populate the Spinner from the Database:

In your activity or fragment, retrieve data from the database and populate the spinner using an ArrayAdapter. Here’s an example:

Java

```
DatabaseHelper dbHelper = new DatabaseHelper(this);
SQLiteDatabase db = dbHelper.getReadableDatabase();

Cursor cursor = db.query(DatabaseHelper.TABLE_NAME,
        new String[]{DatabaseHelper.COLUMN_ID, DatabaseHelper.COLUMN_NAME},
        null, null, null, null, null);

Spinner spinner = findViewById(R.id.spinner);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

```
while (cursor.moveToNext()) {
    String itemName = cursor.getString(cursor.getColumnIndex(DatabaseHelper.COLUMN_NAME));
    adapter.add(itemName);
}

spinner.setAdapter(adapter);

cursor.close();
db.close();
```

**5. Handle Spinner Selection:**

If you want to perform actions when an item is selected from the spinner, set an OnItemSelectedListener for the spinner:

Java

```
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
    @Override
    public void onItemSelected(AdapterView<?> parentView, View selectedItemView, int position, long id) {
        // Handle item selection here
        String selectedItem = (String) parentView.getItemAtPosition(position);
        Toast.makeText(getApplicationContext(), "Selected: " + selectedItem, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onNothingSelected(AdapterView<?> parentView) {
        // Handle no item selected
    }
});
```

Now, your spinner should be populated with data from the SQLite database and you can handle item selections as

Needed. Make sure to adapt the code to your specific requirements and database schema.

**Android SQLite ListView:**

SQLite is an open-source lightweight relational database management system (RDBMS) to perform database operations, such as storing, updating, retrieving data from the database. Generally, in our android applications Shared Preferences, Internal Storage and External Storage options are useful to store and maintain a small amount of Data. In case, Deal with large amounts of data, then the **SQLite database** is the preferable option to store and maintain the data in a structured format.

How to create & insert data into SQLite Database and how to retrieve and show the data in custom listview in android application with examples.

**Android SQLite ListView Example:**

Following is the example of creating the SQLite database, insert and show the details from the SQLite database into an android listview using the **SQLiteOpenHelp** class. Create a new android application using android studio and give names as **SQLiteExample**. In case if you are not aware of creating an app in android studio check this article Android Hello World App.

Once we create an application, create a class file **DbHandler.java** in **\java\com.tutlane.sqliteexample** path to implement SQLite database related activities for that right-click on your application folder à Go to **New** à select **Java Class** and give name as **DbHandler.java**.

Once we create a new class file **DbHandler.java**, open it and write the code like as shown below

## DbHandler.java

```
package com.tutlane.sqliteexample;
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import java.util.ArrayList;
import java.util.HashMap;

/**
 * Created by tutlane on 06-01-2018.
 */

public class DbHandler extends SQLiteOpenHelper {
    private static final int DB_VERSION = 1;
    private static final String DB_NAME = "usersdb";
    private static final String TABLE_Users = "userdetails";
    private static final String KEY_ID = "id";
    private static final String KEY_NAME = "name";
    private static final String KEY_LOC = "location";
    private static final String KEY_DESG = "designation";
    public DbHandler(Context context){
        super(context,DB_NAME, null, DB_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db){
        String CREATE_TABLE = "CREATE TABLE " + TABLE_Users + "("
                + KEY_ID + " INTEGER PRIMARY KEY AUTOINCREMENT," + KEY_NAME + " TEXT,"
                + KEY_LOC + " TEXT,"
                + KEY_DESG + " TEXT"+ ")";
        db.execSQL(CREATE_TABLE);
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
        // Drop older table if exist
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_Users);
        // Create tables again
        onCreate(db);
    }
    // **** CRUD (Create, Read, Update, Delete) Operations ***** //

    // Adding new User Details
    void insertUserDetails(String name, String location, String designation){
        //Get the Data Repository in write mode
        SQLiteDatabase db = this.getWritableDatabase();
        //Create a new map of values, where column names are the keys
        ContentValues cValues = new ContentValues();
        cValues.put(KEY_NAME, name);
        cValues.put(KEY_LOC, location);
        cValues.put(KEY_DESG, designation);
        // Insert the new row, returning the primary key value of the new row
        long newRowId = db.insert(TABLE_Users,null, cValues);
        db.close();
    }
    // Get User Details
    public ArrayList<HashMap<String, String>> GetUsers(){
        SQLiteDatabase db = this.getWritableDatabase();
        ArrayList<HashMap<String, String>> userList = new ArrayList<>();
        String query = "SELECT name, location, designation FROM "+ TABLE_Users;
        Cursor cursor = db.rawQuery(query,null);
        while (cursor.moveToNext()){
            HashMap<String,String> user = new HashMap<>();
            user.put("name",cursor.getString(cursor.getColumnIndex(KEY_NAME)));
```

```
            user.put("designation",cursor.getString(cursor.getColumnIndex(KEY_DESG)));
            user.put("location",cursor.getString(cursor.getColumnIndex(KEY_LOC)));
            userList.add(user);
        }
        return  userList;
    }
    // Get User Details based on userid
    public ArrayList<HashMap<String, String>> GetUserByUserId(int userid){
        SQLiteDatabase db = this.getWritableDatabase();
        ArrayList<HashMap<String, String>> userList = new ArrayList<>();
        String query = "SELECT name, location, designation FROM "+ TABLE_Users;
        Cursor cursor = db.query(TABLE_Users, new String[]{KEY_NAME, KEY_LOC, KEY_DESG}, KEY_ID+ "=?",new String[]
{String.valueOf(userid)},null, null, null, null);
        if (cursor.moveToNext()){
            HashMap<String,String> user = new HashMap<>();
            user.put("name",cursor.getString(cursor.getColumnIndex(KEY_NAME)));
            user.put("designation",cursor.getString(cursor.getColumnIndex(KEY_DESG)));
            user.put("location",cursor.getString(cursor.getColumnIndex(KEY_LOC)));
            userList.add(user);
        }
        return  userList;
    }
    // Delete User Details
    public void DeleteUser(int userid){
        SQLiteDatabase db = this.getWritableDatabase();
        db.delete(TABLE_Users, KEY_ID+" = ?",new String[]{String.valueOf(userid)});
        db.close();
    }
    // Update User Details
    public int UpdateUserDetails(String location, String designation, int id){
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues cVals = new ContentValues();
        cVals.put(KEY_LOC, location);
        cVals.put(KEY_DESG, designation);
        int count = db.update(TABLE_Users, cVals, KEY_ID+" = ?",new String[]{String.valueOf(id)});
        return  count;
    }
}
```

If you observe above code, we implemented all SQLite Database related activities to perform CRUD operations in android application.

Now open **activity_main.xml** file from **\res\layout** folder path and write the code like as shown below.

## XML & JSON :

**XML Parsing:** Android - XML Parser - XML stands for Extensible Mark-up Language.XML is a very popular format and commonly used for sharing data on the internet.  A "read" method for each tag you want to include  such as readEntry() and readTitle() . The parser reads tags from the input stream.

**Android XML Parsing using SAX Parser:**

Android provides the facility to parse the xml file using SAX, DOM etc. parsers. The SAX parser cannot be used to create the XML file, It can be used to parse the xml file only.

**Advantage of SAX Parser over DOM:**

- It consumes less memory than DOM.

## Example of android SAX Xml parsing:

### *activity_main.xml*

Drag the one textview from the pallete. Now the activity_main.xml file will look like this:

*File: activity_main.xml*

```
<RelativeLayout xmlns:androclass="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:layout_marginLeft="75dp"
        android:layout_marginTop="46dp"
        android:text="TextView" />
</RelativeLayout>
```

**Android XML Parsing using DOM Parser:**

- Android DOM (Document Object Model) parser is a program that parses an XML document and extracts the required information from it.
- We can parse the xml document by DOM parser also. It can be used to create and parse the xml file.

**Advantage of DOM Parser over SAX:**

- It can be used to create and parse the xml file both but SAX parser can only be used to parse the xml file.

**Disadvantage of DOM Parser over SAX:**

- It consumes more memory than SAX.

## Example of android DOM Xml parsing

*activity_main.xml*

Drag the one textview from the pallete. Now the activity_main.xml file will look like this:

**JSON Basics in Android:**

**JSON Parsing in Android:**



**JSON (JavaScript Object Notation)** is a straightforward data exchange format to interchange the server's data and it is a better alternative for XML. Because JSON is a lightweight and structured language. Android supports all the JSON classes such as **JSONStringer, JSONObject, JSONArray** and all ot forms to parse the JSON data and fetch the required information by the program.
JSON's main advantage :

1. It is a language-independent and the JSON object will contain data like a key/value pair.
2. JSON nodes will start with a **square bracket ([)** or with a **curly bracket ({)**. The square and curly bracket's primary difference is that **square bracket ([)** represents the beginning of a **JSONArray node**. Whereas, **curly bracket ({)** represents a **JSONObject**. So one needs to call the appropriate method to get the data.
3. Sometimes JSON data start with [. We then need to use the **getJSONArray()** method to get the data. Similarly, if it starts with {, then we need to use the **getJSONobject()** method.

**Syntax of the JSON file:-**

```
{
"Name": "GeeksforGeeks",
"Estd": 2009,
```

```
"age": 10,
"address": {
    "buildingAddress": "5th & 6th Floor Royal Kapsons, A- 118",
    "city": "Sector- 136, Noida",
    "state": "Uttar Pradesh (201305)",
    "postalCode": "201305"
},
```

- To parse a JSON file in Android. For implement this project using **Kotlin** language.