Android:

The primary programming language for Android app development is Kotlin, although Java is still widely used. Kotlin is now the recommended language for Android development, with Google encouraging developers to leverage its modern features and benefits.

Kotlin is a modern, statically typed language that is increasingly favoured for Android development due to its conciseness, expressiveness, and ability to reduce common code errors. It also integrates well with existing Java code.

While Kotlin is the preferred language, Java remains a valid and popular choice for Android development. Many existing Android projects are written in Java, and it still provides a strong foundation for learning app development.

- Feature vs Smart Device:

  Any device that is designed to do minimal operations is called as feature device

  e.g.-> remote, mobile phones, washing machine etc.

  Any device that has got its own Operation system is called as smart device

  e.g.-> smartphones, smart TV, smartwatch, smart Refrigerators

- History:

Android is an operation system that is designed on top of Linux OS with some changes

It is designed for mobile devices

It is free and open-source OS

It is developed by Andy Rubin in 2003

It was acquired by google in 2005

In 2007, google announced the development of Android OS

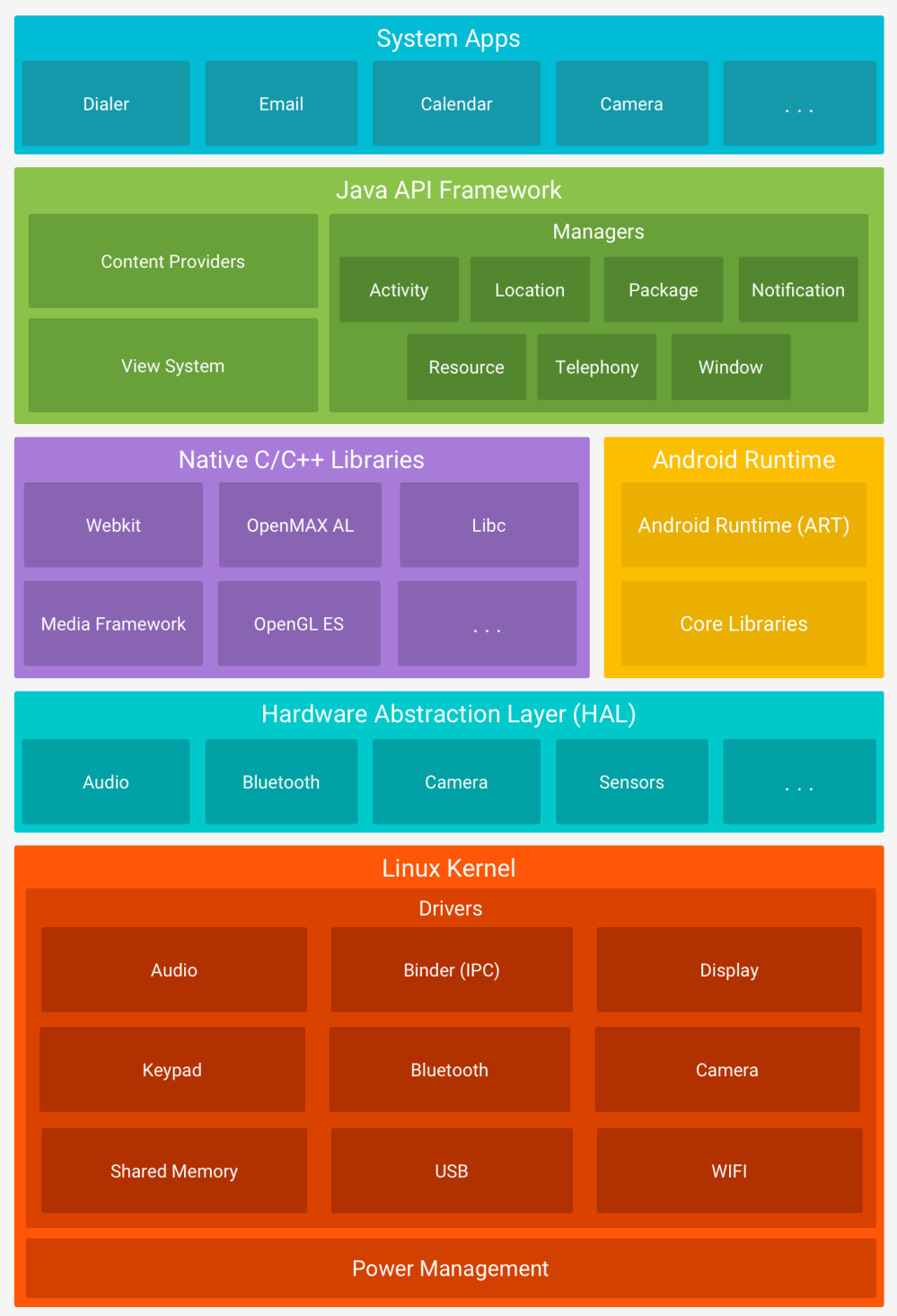
In 2008, HTC launched its first Android device

- Versions of Android:

  1.0 -> Android 1.0(September 2008)
  1.1 -> Android 1.1 (Feburary 2009)
  1.5 -> Android Cupcake (April 2009)
  1.6 -> Android Donut (September 2009)
  2.0 -> Android Eclair - 2.0 (October 2009) - 2.0.1 (December 2009) - 2.1 (January 2010)
  2.2 & 2.2.3 -> Android Froyo (May 2010)
  2.3 -> Android GingerBread - 2.3 - 2.3.2 (December 2010) - 2.3.3 - 2.3.7 (Feburary 2011)
  3.0 -> Android Honeycomb - 3.0 (Feburary 2011) - 3.1 (May 2011) - 3.2-3.2.6 (July 2011)
  4.0 -> Android Ice Cream Sandwich - 4.0 - 4.0.2 (October 2011) - 4.0.3 - 4.0.4 (December 2011)
  4.1 -> Android Jelly Bean - 4.1 – 4.1.2 (July 2012) - 4.2 – 4.2.2 (November 2012) - 4.3 – 4.3.1 (July 2013)
  4.2 -> Android KitKat - 4.4 – 4.4.4 (October 2013) - 4.4W – 4.4W.2 (June 2014)
  5.0 -> Android Lollipop - 5.0 – 5.0.2 (November 2014) - 5.1 – 5.1.1 (March 2015)
  6.0 & 6.0.1 -> Android Marshmallow (September 2015)

7.0 -> Android Nougat - 7.0 (August 2016) - 7.1 - 7.1.2 (October 2016)
8.0 -> Android Oreo - 8.0 (August 2017) - 8.1 (December 2017)
9.0 -> Android Pie (August 2018)
10 -> Android 10/ Android Q / Quince Tart (Sepetember 2019)
11 -> Android 11 / Red Velvet Cake (Sepettember 2020)
12 -> Android 12 - 12 (October 2021) / Snow Cone - 12.1 (March 2022) / Snow Cone V2
13 -> Android 13 / Tiramisu (August 2022)

Android Architecture: Android is an open source, Linux-based software stack created for a wide array of devices and form factors.



- Applications:
    - It is the top layer of android architecture.
    - The pre-installed applications like home, contacts, camera, gallery etc and third-party applications downloaded from the play store like chat applications, games etc. will be installed on this layer.
- Application Framework:
    - Application Framework provides several important classes which are used to create an Android application.
    - It includes different types of services like activity manager, notification manager, view system, package manager etc. which are helpful for the development of our application
- Android Runtime:
    - It is JVM for android, faster than desktop jvm.
    - It uses 30% native instructions and 70%-byte codes
    - It is possible because of double compilation that android uses.
    - The byte code is given to dex compiler which compiles and produces, 70%-byte code and 30% native code in. dex(dalvik executables) file.
    - For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). - ART is written to run multiple virtual machines on low-memory devices by executing Dalvik Executable format (DEX) files, a bytecode format designed specifically for Android that's optimized for a minimal memory footprint.
    - Some of the major features of ART include the following:
        - Ahead-of-time (AOT) and just-in-time (JIT) compilation
        - Optimized garbage collection (GC)
        - On Android 9 (API level 28) and higher, conversion of an app package's DEX files to more compact machine code.
    - Prior to Android version 5.0 (API level 21), Dalvik was the Android runtime. If your app runs well on ART, then it can work on Dalvik as well, but the reverse might not be true.
    - Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language, including some Java 8 language features, that the Java API framework uses.
- Platform Libraries:
    - The Platform Libraries includes various C/C++ core libraries and Java based libraries such as Media, Graphics, Surface Manager, OpenGL etc. to provide a support for android development.
    - Android has its own C runtime library.
    - The name for the Android’s runtime library is Bionic.

- Hardware abstraction layer (HAL):
  - The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework.
  - The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or Bluetooth module.
  - When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.
- Linux Kernel:
  - Linux Kernel is heart of the android architecture.
  - For android operating system the Linux kernel is customized.
  - Changes added to android from main line kernel is called as Androdism.
  - It manages all the available drivers such as display drivers, camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc. which are required during the runtime.
  - The Linux Kernel will provide an abstraction layer between the device hardware and the other components of android architecture.
  - The features of Linux kernel are:

    1. Security: The Linux kernel handles the security between the application and the system.
    2. Memory Management: It efficiently handles the memory management thereby providing the freedom to develop our apps.
    3. Process Management: It manages the process well, allocates resources to processes whenever they need them.
    4. Network Stack: It effectively handles the network communication.
    5. Driver Model: It ensures that the application works properly on the device and hardware manufacturers responsible for building their drivers into the Linux build.

File Structure:

Manifest folder-

- Manifests folder contains AndroidManifest.xml of our android application.

- This file contains information about our application such as the Android version, metadata and other application components.
- It acts as an intermediator between android OS and our application.

java Folder-

- The Java folder contains all the java source code (.java) files that we create during the app development, including other Test files.

res (Resource Folder)-

- The resource folder is the most important folder because it contains all the non-code sources like images layouts, and UI strings for our android application.
- the sub folders inside this res folder are
    1. Drawable
       It is used to store the images/logos used in our application
    2. Layout
       It is used to create the UI of our application
    3. mipmap
       It is used to have icon for our application
    4. values
       It is used to keep constants that we can use across our application
    5. anim
       It is used to keep the animation xml files
    6. menu
       It is used to keep the menu xml files for the menus that we create

Gradle Scripts-

In build. Gradle (Project) there are build scripts. In build. Gradle (Module) plugins and implementations are used to build configurations that can be applied to all our application modules.

Gradle:

It is a build automation tool It is an open-source tool It is used to build the software's that we design It can build any type of software's Android studio uses this Gradle for the automated build system for its application that we design.

Fundamental Components of Android:

The fundamental components are also called as Pillars of Android

There are 4 fundamental components for Android application:

Activity:

An activity is the entry point for interacting with the user.

It represents a single screen with a user interface.

Services:

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons.

It is a component that runs in the background to perform long-running operations or to perform work for remote processes.

A service does not provide a user interface.

Content Provider:

A content provider manages a shared set of app data that you can store in the file system, in a SQLite database or on any other persistent storage location that your app can access.

Through the content provider, other apps can query or modify the data, if the content provider permits it.

Broadcast Receiver:

A broadcast receiver is a component that lets the system deliver events to the app outside of a regular user flow so the app can respond to system-wide broadcast announcements.

Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to apps that aren't currently running.

Activity:

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI with setContentView(View). While activities are often presented to the user as full-screen windows,

they can also be used in other ways: as floating windows (via a theme with R.attr.windowIsFloating set), multi-window mode or embedded into other windows. There are two methods almost all subclasses of Activity will implement:

onCreate(Bundle) is where you initialize your activity. Most importantly, here you will usually call setContentView(int) with a layout resource defining your UI, and using findViewById(int) to retrieve the widgets in that UI that you need to interact with programmatically.

onPause() is where you deal with the user pausing active interaction with the activity. Any changes made by the user should at this point be committed (usually to the ContentProvider holding the data). In this state the activity is still visible on screen.

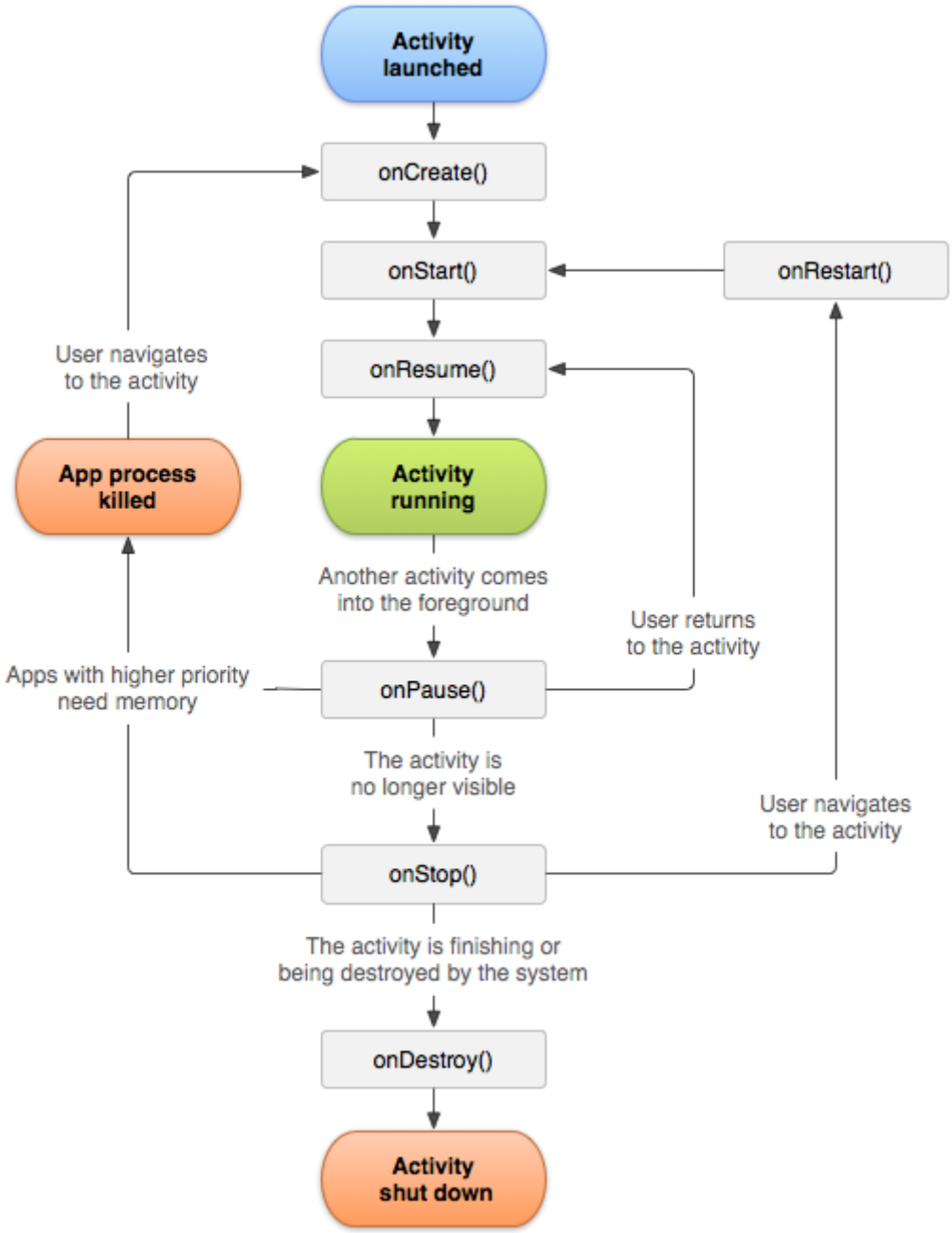
To be of use with Context.startActivity(), all activity classes must have a corresponding <activity> declaration in their package's AndroidManifest.xml.

Activity Lifecycle:

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

An activity has essentially four states:

- If an activity is in the foreground of the screen (at the highest position of the topmost stack), it is active or running. This is usually the activity that the user is currently interacting with.
- If an activity has lost focus but is still presented to the user, it is visible. It is possible if a new non-full-sized or transparent activity has focus on top of your activity, another activity has higher position in multi-window mode, or the activity itself is not focusable in current windowing mode. Such activity is completely alive (it maintains all state and member information and remains attached to the window manager).
- If an activity is completely obscured 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.



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 visible, active and interacting with the user. An activity can frequently go between the resumed and paused states -- for example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered -- so the code in these methods should be fairly lightweight.

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

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);

    protected void onStart();

    protected void onRestart();

    protected void onResume();

    protected void onPause();

    protected void onStop();

    protected void onDestroy();
}
```

Method	Description	Killable?	Next
`onCreate()`	Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one. Always followed by `onStart()`.	No	`onStart()`
`onRestart()`	Called after your activity has been stopped, prior to it being started again. Always followed by `onStart()`	No	`onStart()`
`onStart()`	Called when the activity is becoming visible to the user. Followed by `onResume()` if the activity comes to the foreground, or `onStop()` if it becomes hidden.	No	`onResume()` or `onStop()`
`onResume()`	Called when the activity will start interacting with the user. At this point your activity is at the top of its activity stack, with user input going to it. Always followed by `onPause()`.	No	`onPause()`

`onPause()`	Called when the activity loses foreground state, is no longer focusable or before transition to stopped/hidden or destroyed state. The activity is still visible to user, so it's recommended to keep it visually active and continue updating the UI. Implementations of this method must be very quick because the next activity will not be resumed until this method returns. Followed by either `onResume()` if the activity returns back to the front, or `onStop()` if it becomes invisible to the user.	Pre-`Build.VERSION_CODES.HONEYCOMB`	`onResume()` or `onStop()`
`onStop()`	Called when the activity is no longer visible to the user. This may happen either because a new activity is being started on top, an existing one is being brought in front of this one, or this one is being destroyed. This is typically used to stop animations and refreshing the UI, etc. Followed by either `onRestart()` if this activity is coming back to interact with the user, or `onDestroy()` if this activity is going away.	Yes	`onRestart()` or `onDestroy()`
`Destroy()`	The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called `Activity.finish` on it), or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the `isFinishing()` method.	Yes	*nothing*

Configuration Changes

If the configuration of the device (as defined by the `Resources.Configuration` class) changes, then anything displaying a user interface will need to update to match that configuration. Because Activity is the primary mechanism for interacting with the user, it includes special support for handling configuration changes.

Unless you specify otherwise, a configuration change (such as a change in screen orientation, language, input devices, etc.) will cause your current activity to be *destroyed*, going through the normal activity lifecycle process of `onPause()`, `onStop()`, and `onDestroy()` as appropriate. If the activity had been in the foreground or visible to the user, once `onDestroy()` is called in that instance then a new instance of the activity will be created, with whatever savedInstanceState the previous instance had generated from `onSaveInstanceState(Bundle)`.

This is done because any application resource, including layout files, can change based on any configuration value. Thus the only safe way to handle a configuration change is to re-retrieve all resources, including layouts, drawables, and strings. Because activities must already know how to save their state and re-create themselves from that state, this is a convenient way to have an activity restart itself with a new configuration.

In some special cases, you may want to bypass restarting of your activity based on one or more types of configuration changes. This is done with the `android:configChanges` attribute in its manifest. For any types of configuration changes you say that you handle there, you will receive a call to your current activity's `onConfigurationChanged(Configuration)` method instead of being restarted. If a configuration change involves any that you do not handle, however, the activity will still be restarted and `onConfigurationChanged(Configuration)` will not be called.

Starting Activities and Getting Results

The `startActivity(Intent)` method is used to start a new activity, which will be placed at the top of the activity stack. It takes a single argument, an `Intent`, which describes the activity to be executed.

Sometimes you want to get a result back from an activity when it ends. For example, you may start an activity that lets the user pick a person in a list of contacts; when it ends, it returns the person that was selected. To do this, you call the `startActivityForResult(Intent, int)` version with a second integer parameter identifying the call. The result will come back through your `onActivityResult(int, int, Intent)` method.

When an activity exits, it can call `setResult(int)` to return data back to its parent. It must always supply a result code, which can be the standard results RESULT_CANCELED, RESULT_OK, or any custom values starting at RESULT_FIRST_USER. In addition, it can optionally return back an Intent containing any additional data it wants. All of this information appears back on the parent's `Activity.onActivityResult()`, along with the integer identifier it originally supplied.

If a child activity fails for any reason (such as crashing), the parent activity will receive a result with the code RESULT_CANCELED.

```
public class MyActivity extends Activity {
    ...

    static final int PICK_CONTACT_REQUEST = 0;

    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
            // When the user center presses, let them pick a contact.
            startActivityForResult(
                new Intent(Intent.ACTION_PICK,
                new Uri("content://contacts")),
                PICK_CONTACT_REQUEST);
```

```
            return true;
        }
        return false;
    }

    protected void onActivityResult(int requestCode, int resultCode,
            Intent data) {
        if (requestCode == PICK_CONTACT_REQUEST) {
            if (resultCode == RESULT_OK) {
                // A contact was picked.  Here we will just display it
                // to the user.
                startActivity(new Intent(Intent.ACTION_VIEW, data));
            }
        }
    }
}
```

Permissions

The ability to start a particular Activity can be enforced when it is declared in its manifest's `<activity>` tag. By doing so, other applications will need to declare a corresponding `<uses-permission>` element in their own manifest to be able to start that activity.

When starting an Activity you can set `Intent.FLAG_GRANT_READ_URI_PERMISSION` and/or `Intent.FLAG_GRANT_WRITE_URI_PERMISSION` on the Intent. This will grant the Activity access to the specific URIs in the Intent. Access will remain until the Activity has finished (it will remain across the hosting process being killed and other temporary destruction). As of `Build.VERSION_CODES.GINGERBREAD`, if the Activity was already created and a new Intent is being delivered to `onNewIntent(android.content.Intent)`, any newly granted URI permissions will be added to the existing ones it holds.

Process Lifecycle

The Android system attempts to keep an application process around for as long as possible, but eventually will need to remove old processes when memory runs low. As described in Activity Lifecycle, the decision about which process to remove is intimately tied to the state of the user's interaction with it. In general, there are four states a process can be in based on the activities running in it, listed here in order of importance. The system will kill less important processes (the last ones) before it resorts to killing more important processes (the first ones).

1. The **foreground activity** (the activity at the top of the screen that the user is currently interacting with) is considered the most important. Its process will only be killed as a last resort, if it uses more memory than is available on the device. Generally at this point the device has reached a memory paging state, so this is required in order to keep the user interface responsive.
2. A **visible activity** (an activity that is visible to the user but not in the foreground, such as one sitting behind a foreground dialog or next to other activities in multi-window mode) is considered extremely important and will not be killed unless that is required to keep the foreground activity running.
3. A **background activity** (an activity that is not visible to the user and has been stopped) is no longer critical, so the system may safely kill its process to reclaim memory for other foreground or visible processes. If its process needs to be killed, when the user navigates back to the activity (making it visible on the screen again), its `onCreate(Bundle)` method will be called with the savedInstanceState it had previously supplied in `onSaveInstanceState(Bundle)` so that it can restart itself in the same state as the user last left it.
4. An **empty process** is one hosting no activities or other application components (such as `Service` or `BroadcastReceiver` classes). These are killed very quickly by the system as memory becomes low. For this reason, any background operation you do outside of an activity must be executed in the context of an activity BroadcastReceiver or Service to ensure that the system knows it needs to keep your process around.

ActivityManager

Added in API level 1

Kotlin | **Java**

```
public class ActivityManager
extends Object
```

java.lang.Object
↳ android.app.ActivityManager

This class gives information about, and interacts with, activities, services, and the containing process.

A number of the methods in this class are for debugging or informational purposes and they should not be used to affect any runtime behavior of your app. These methods are called out as such in the method level documentation.

Most application developers should not have the need to use this class, most of whose methods are for specialized use cases. However, a few methods are more broadly applicable. For instance, `isLowRamDevice()` enables your app to detect whether it is running on a low-memory device, and behave accordingly. `clearApplicationUserData()` is for apps with reset-data functionality.

In some special use cases, where an app interacts with its Task stack, the app may use the `ActivityManager.AppTask` and `ActivityManager.RecentTaskInfo` inner classes. However, in general, the methods in this class should be used for testing and debugging purposes only.

Nested classes	
class	`ActivityManager.AppTask` The AppTask allows you to manage your own application's tasks.
class	`ActivityManager.MemoryInfo` Information you can retrieve about the available memory through `ActivityManager.getMemoryInfo`.
class	`ActivityManager.ProcessErrorStateInfo` Information you can retrieve about any processes that are in an error condition.
class	`ActivityManager.RecentTaskInfo` Information you can retrieve about tasks that the user has most recently started or visited.
class	`ActivityManager.RunningAppProcessInfo` Information you can retrieve about a running process.
class	`ActivityManager.RunningServiceInfo` Information you can retrieve about a particular Service that is currently running in the system.
class	`ActivityManager.RunningTaskInfo` Information you can retrieve about a particular task that is currently "running" in the system.
class	`ActivityManager.TaskDescription` Information you can set and retrieve about the current activity within the recent task list.

Comparison to Java:

Kotlin fixes a series of issues that Java suffers from:

- Null references are controlled by the type system.
- No raw types
- Arrays in Kotlin are invariant
- Kotlin has proper function types, as opposed to Java's SAM-conversions
- Use-site variance without wildcards
- Kotlin does not have checked exceptions
- Separate interfaces for read-only and mutable collections.

What Java has that Kotlin does not?

- Checked exceptions
- Primitive types that are not classes. The byte-code uses primitives where possible, but they are not explicitly available.
- Static members are replaced with companion objects, top-level functions, extension functions, or @JvmStatic.
- Wildcard-types are replaced with declaration-site variance and type projections.
- Ternary-operator a ? b : c is replaced with if expression.
- Records
- Pattern Matching
- package-private visibility modifier

What Kotlin has that Java does not?

- Lambda expressions + Inline functions = performant custom control structures
- Extension functions
- Null-safety
- Smart casts (Java 16: Pattern Matching for instanceof)
- String templates (Java 21: String Templates (Preview))
- Properties
- Primary constructors
- First-class delegation
- Type inference for variable and property types (Java 10: Local-Variable Type Inference)
- Singletons
- Declaration-site variance & Type projections
- Range expressions
- Operator overloading
- Companion objects
- Data classes
- Coroutines
- Top-level functions
- Default arguments

- Named parameters
- Infix functions
- Expect and actual declarations
- Explicit API mode and better control of API surface

Getters and setters

Methods that follow the Java conventions for getters and setters (no-argument methods with names starting with `get` and single-argument methods with names starting with `set`) are represented as properties in Kotlin. Such properties are also called **synthetic properties**. `Boolean` accessor methods (where the name of the getter starts with `is` and the name of the setter starts with `set`) are represented as properties which have the same name as the getter method.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek(
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // call isLenient()
        calendar.isLenient = true // call setLenient()
    }
}
```

Kotlin has always allowed you to write `person.age`, where `age` is a synthetic property. Now, you can also create references to `Person::age` and `person::age`. The same applies for `name`, as well.

```
val persons = listOf(Person("Jack", 11), Person("Sofie", 12), Person("Peter", 11
    persons
         // Call a reference to Java synthetic property:
        .sortedBy(Person::age)
         // Call Java getter via the Kotlin property syntax:
        .forEach { person -> println(person.name) }
```

Methods returning void

If a Java method returns `void`, it will return `Unit` when called from Kotlin. If by any chance someone uses that return value, it will be assigned at the call site by the Kotlin compiler since the value itself is known in advance (being `Unit`).

Escaping for Java identifiers that are keywords in Kotlin

Some of the Kotlin keywords are valid identifiers in Java: `in`, `object`, `is`, and other. If a Java library uses a Kotlin keyword for a method, you can still call the method escaping it with the backtick (`) character:

```
foo.`is`(bar)
```

Null-safety and platform types

Any reference in Java may be `null`, which makes Kotlin's requirements of strict null-safety impractical for objects coming from Java. Types of Java declarations are treated in Kotlin in a specific manner and called **platform types**. Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java (see more below).

Consider the following examples:

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)
```

When you call methods on variables of platform types, Kotlin does not issue nullability errors at compile time, but the call may fail at runtime, because of a null-pointer exception or an assertion that Kotlin generates to prevent nulls from propagating:

```
item.substring(1) // allowed, throws an exception if item == null
```

Platform types are **non-denotable**, meaning that you can't write them down explicitly in the language. When a platform value is assigned to a Kotlin variable, you can rely on the type inference (the variable will have an inferred platform type then, as `item` has in the example above), or you can choose the type you expect (both nullable and non-nullable types are allowed):

```
val nullable: String? = item // allowed, always works
val notNull: String = item // allowed, may fail at runtime
```

If you choose a non-nullable type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-nullable variables from holding nulls. Assertions are also emitted when you pass platform values to Kotlin functions expecting non-null values and in other cases. Overall, the compiler does its best to prevent nulls from propagating far through the program although sometimes this is impossible to eliminate entirely, because of generics.

If you choose a non-nullable type, the compiler will emit an assertion upon assignment. This prevents Kotlin's non-nullable variables from holding nulls. Assertions are also emitted when you pass platform values to Kotlin functions expecting non-null values and in other cases. Overall, the compiler does its best to prevent nulls from propagating far through the program although sometimes this is impossible to eliminate entirely, because of generics.

Notation for platform types

As mentioned above, platform types can't be mentioned explicitly in the program, so there's no syntax for them in the language. Nevertheless, the compiler and IDE need to display them sometimes (for example, in error messages or parameter info), so there is a mnemonic notation for them:

- `T!` means "`T` or `T?`",
- `(Mutable)Collection<T>!` means "Java collection of `T` may be mutable or not, may be nullable or not",
- `Array<(out) T>!` means "Java array of `T` (or a subtype of `T`), nullable or not"

Nullability annotations

Java types that have nullability annotations are represented not as platform types, but as actual nullable or non-nullable Kotlin types. The compiler supports several flavors of nullability annotations, including:

- JetBrains ↗ (`@Nullable` and `@NotNull` from the `org.jetbrains.annotations` package)
- JSpecify ↗ (`org.jspecify.annotations`)
- Android (`com.android.annotations` and `android.support.annotations`)
- JSR-305 (`javax.annotation`, more details below)
- FindBugs (`edu.umd.cs.findbugs.annotations`)
- Eclipse (`org.eclipse.jdt.annotation`)
- Lombok (`lombok.NonNull`)
- RxJava 3 (`io.reactivex.rxjava3.annotations`)

Type parameters

By default, the nullability of plain type parameters in both Kotlin and Java is undefined. In Java, you can specify it using nullability annotations. Let's annotate the type parameter of the `Base` class:

```
public class Base<@NotNull T> {}
```

When inheriting from `Base`, Kotlin expects a non-nullable type argument or type parameter. Thus, the following Kotlin code produces a warning:

```
class Derived<K> : Base<K> {} // warning: K has undefined nullability
```

You can fix it by specifying the upper bound `K : Any`.

Kotlin also supports nullability annotations on the bounds of Java type parameters. Let's add bounds to `Base`:

```
public class BaseWithBound<T extends @NotNull Number> {}
```

Kotlin translates this just as follows:

```
class BaseWithBound<T : Number> {}
```

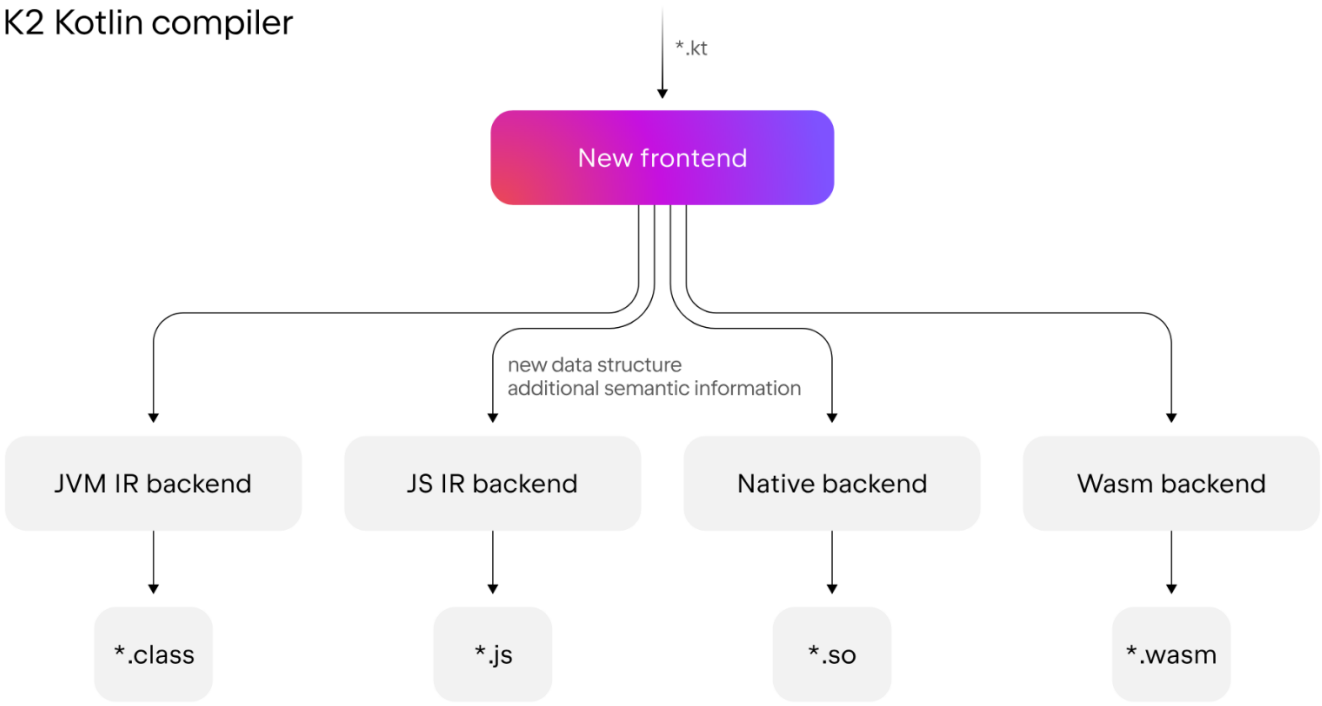
So passing nullable type as a type argument or type parameter produces a warning.

Annotating type arguments and type parameters works with the Java 8 target or higher. The feature requires that the nullability annotations support the `TYPE_USE` target (`org.jetbrains.annotations` supports this in version 15 and above).

> If a nullability annotation supports other targets that are applicable to a type in addition to the `TYPE_USE` target, `TYPE_USE` takes priority. For example, if `@Nullable` has both `TYPE_USE` and `METHOD` targets, the Java method signature `@Nullable String[] f()` becomes `fun f(): Array<String?>!` in Kotlin.

K2 compiler:

As the Kotlin language and ecosystem have continued to evolve, so has the Kotlin compiler. The first step was the introduction of the new JVM and JS IR (Intermediate Representation) backends that share logic, simplifying code generation for targets on different platforms. Now, the next stage of its evolution brings a new frontend known as K2.



With the arrival of the K2 compiler, the Kotlin frontend has been completely rewritten and features a new, more efficient architecture. The fundamental change the new compiler brings is the use of one unified data structure that contains more semantic information. This frontend is responsible for performing semantic analysis, call resolution, and type inference.

The new architecture and enriched data structure enables the K2 compiler to provide the following benefits:

- Improved call resolution and type inference. The compiler behaves more consistently and understands your code better.
- Easier introduction of syntactic sugar for new language features. In the future, you'll be able to use more concise, readable code when new features are introduced.
- Faster compilation times. Compilation times can be significantly faster.
- Enhanced IDE performance. Starting with 2025.1, IntelliJ IDEA uses K2 mode to analyze your Kotlin code, increasing stability and providing performance improvements. For more information, see Support in IDEs.

The new K2 compiler is enabled by default starting with 2.0.0. For more information on the new features provided in Kotlin 2.0.0, as well as the new K2 compiler, see What's new in Kotlin 2.0.0.

Basic syntax:

Package definition and imports:-

Package specification should be at the top of the source file:

```
package my.demo

import kotlin.text.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

Program entry point

An entry point of a Kotlin application is the `main` function:

```
fun main() {
    println("Hello world!")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Another form of `main` accepts a variable number of `String` arguments:

```
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Print to the standard output

`print` prints its argument to the standard output:

```
print("Hello ")
print("world!")
```

Open in Playground → Target: JVM Running on v.2.1.21

`println` prints its arguments and adds a line break, so that the next thing you print appears on the next line:

```
println("Hello world!")
println(42)
```

Open in Playground → Target: JVM Running on v.2.1.21

Read from the standard input

The `readln()` function reads from the standard input. This function reads the entire line the user enters as a string.

You can use the `println()`, `readln()`, and `print()` functions together to print messages requesting and showing user input:

```
// Prints a message to request input
println("Enter any word: ")

// Reads and stores the user input. For example: Happiness
val yourWord = readln()

// Prints a message with the input
print("You entered the word: ")
print(yourWord)
// You entered the word: Happiness
```

Functions

A function with two `Int` parameters and `Int` return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Open in Playground → Target: JVM Running on v.2.1.21

A function body can be an expression. Its return type is inferred:

```
fun sum(a: Int, b: Int) = a + b
```

Open in Playground → Target: JVM Running on v.2.1.21

A function that returns no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Variables

In Kotlin, you declare a variable starting with a keyword, `val` or `var`, followed by the name of the variable.

Use the `val` keyword to declare variables that are assigned a value only once. These are immutable, read-only local variables that can't be reassigned a different value after initialization:

```
// Declares the variable x and initializes it with the value of 5
val x: Int = 5
// 5
```

Open in Playground → Target: JVM Running on v.2.1.21

Use the `var` keyword to declare variables that can be reassigned. These are mutable variables, and you can change their values after initialization:

```
// Declares the variable x and initializes it with the value of 5
var x: Int = 5
// Reassigns a new value of 6 to the variable x
x += 1
// 6
```

Open in Playground → Target: JVM Running on v.2.1.21

Kotlin supports type inference and automatically identifies the data type of a declared variable. When declaring a variable, you can omit the type after the variable name:

```
// Declares the variable x with the value of 5;`Int` type is inferred
val x = 5
// 5
```

Open in Playground → Target: JVM Running on v.2.1.21

You can use variables only after initializing them. You can either initialize a variable at the moment of declaration or declare a variable first and initialize it later. In the second case, you must specify the data type:

```
// Initializes the variable x at the moment of declaration; type is not required
val x = 5
// Declares the variable c without initialization; type is required
val c: Int
// Initializes the variable c after declaration
c = 3
// 5
// 3
```

Open in Playground → Target: JVM Running on v.2.1.21

You can declare variables at the top level:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
// x = 0; PI = 3.14
// incrementX()
// x = 1; PI = 3.14
```

Open in Playground → Target: JVM Running on v.2.1.21

Creating classes and instances

To define a class, use the `class` keyword:

```
class Shape
```

Properties of a class can be listed in its declaration or body:

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
```

The default constructor with parameters listed in the class declaration is available automatically:

```
class Rectangle(val height: Double, val length: Double) {
    val perimeter = (height + length) * 2
}
fun main() {
    val rectangle = Rectangle(5.0, 2.0)
    println("The perimeter is ${rectangle.perimeter}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Inheritance between classes is declared by a colon (`:`). Classes are `final` by default; to make a class inheritable, mark it as `open` :

```
open class Shape

class Rectangle(val height: Double, val length: Double): Shape() {
    val perimeter = (height + length) * 2
}
```

Comments

Just like most modern languages, Kotlin supports single-line (or **end-of-line**) and multi-line (**block**) comments:

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

Block comments in Kotlin can be nested:

```
/* The comment starts here
/* contains a nested comment */
and ends here. */
```

String templates

```
fun main() {
    var a = 1
    // simple name in template:
    val s1 = "a is $a"

    a = 2
    // arbitrary expression in template:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
    println(s2)
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Conditional expressions

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

In Kotlin, `if` can also be used as an expression:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

for loop

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

Open in Playground → Target: JVM Running on v.2.1.21

or:

```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

while loop

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

Open in Playground → Target: JVM Running on v.2.1.21

See while loop.

when expression

```
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

Open in Playground → Target: JVM Running on v.2.1.21

Collections

Iterate over a collection:

```
for (item in items) {
    println(item)
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Check if a collection contains an object using `in` operator:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Use lambda expressions to filter and map collections:

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
    .filter { it.startsWith("a") }
    .sortedBy { it }
    .map { it.uppercase() }
    .forEach { println(it) }
```

Open in Playground → Target: JVM Running on v.2.1.21

Ranges

Check if a number is within a range using `in` operator:

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

Open In Playground → Target: JVM Running on v.2.1.21

Check if a number is out of range:

```
val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range, too")
}
```

Open In Playground → Target: JVM Running on v.2.1.21

Iterate over a range:

```
for (x in 1..5) {
    print(x)
}
```

Open In Playground → Target: JVM Running on v.2.1.21

Or over a progression:

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

Open In Playground → Target: JVM Running on v.2.1.21

Nullable values and null checks

A reference must be explicitly marked as nullable when `null` value is possible. Nullable type names have `?` at the end.

Return `null` if `str` does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

Open in Playground → Target: JVM Running on v.2.1.21

or:

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

Open in Playground → Target: JVM Running on v.2.1.21

Type checks and automatic casts

The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj)
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

Open in Playground → Target: JVM Running on v.2.1.21

or:

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj)
    }
    printLength("Incomprehensibilities")
    printLength(1000)
    printLength(listOf(Any()))
}
```

Open in Playground → Target: JVM Running on v.2.1.21

Idioms: A collection of random and frequently used idioms in Kotlin. If you have a favourite idiom, contribute it by sending a pull request.

Create DTOs (POJOs/POCOs)

```
data class Customer(val name: String, val email: String)
```

provides a `Customer` class with the following functionality:

- getters (and setters in case of `var`s) for all properties
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- `component1()`, `component2()`, …, for all properties (see Data classes)

Default values for function parameters

```
fun foo(a: Int = 0, b: String = "") { ... }
```

Filter a list

```
val positives = list.filter { x -> x > 0 }
```

Or alternatively, even shorter:

```
val positives = list.filter { it > 0 }
```

Check the presence of an element in a collection

```
if ("john@example.com" in emailsList) { ... }

if ("jane@example.com" !in emailsList) { ... }
```

String interpolation

```
println("Name $name")
```

Read standard input safely

```
// Reads a string and returns null if the input can't be converted into an integ
val wrongInt = readln().toIntOrNull()
println(wrongInt)
// null

// Reads a string that can be converted into an integer and returns an integer. I
val correctInt = readln().toIntOrNull()
println(correctInt)
// 13
```

Instance checks

```
when (x) {
    is Foo -> ...
    is Bar -> ...
    else   -> ...
}
```

Read-only list

```
val list = listOf("a", "b", "c")
```

Read-only map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

Access a map entry

```
println(map["key"])
map["key"] = value
```

Traverse a map or a list of pairs

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

`k` and `v` can be any convenient names, such as `name` and `age`.

Iterate over a range

```
for (i in 1..100) { ... }  // closed-ended range: includes 100
for (i in 1..<100) { ... } // open-ended range: does not include 100
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
(1..10).forEach { ... }
```

Lazy property

```
val p: String by lazy { // the value is computed only on first access
    // compute the string
}
```

Extension functions

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

Create a singleton

```
object Resource {
    val name = "Name"
}
```

Use inline value classes for type-safe values

```
@JvmInline
value class EmployeeId(private val id: String)

@JvmInline
value class CustomerId(private val id: String)
```

If you accidentally mix up `EmployeeId` and `CustomerId`, a compilation error is triggered.

The `@JvmInline` annotation is only needed for JVM backends.

Instantiate an abstract class

```
abstract class MyAbstractClass {
    abstract fun doSomething()
    abstract fun sleep()
}

fun main() {
    val myObject = object : MyAbstractClass() {
        override fun doSomething() {
            // ...
        }

        override fun sleep() { // ...
        }
    }
    myObject.doSomething()
}
```

If-not-null shorthand

```
val files = File("Test").listFiles()

println(files?.size) // size is printed if files is not null
```

If-not-null-else shorthand

```
val files = File("Test").listFiles()

// For simple fallback values:
println(files?.size ?: "empty") // if files is null, this prints "empty"

// To calculate a more complicated fallback value in a code block, use `run`
val filesSize = files?.size ?: run {
    val someSize = getSomeSize()
    someSize * 2
}
println(filesSize)
```

Execute a statement if null

```
val values = ...
val email = values["email"] ?: throw IllegalStateException("Email is missing!")
```

if expression

```
val y = if (x == 1) {
    "one"
} else if (x == 2) {
    "two"
} else {
    "other"
}
```

Builder-style usage of methods that return Unit

```
fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}
```

Map nullable value if not null

```
val value = ...

val mapped = value?.let { transformValue(it) } ?: defaultValue
// defaultValue is returned if the value or the transform result is null.
```

Return on when statement

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

try-catch expression

```
fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // Working with result
}
```

Configure properties of an object (apply)

```
val myRectangle = Rectangle().apply {
    length = 4
    breadth = 5
    color = 0xFAFAFA
}
```

This is useful for configuring properties that aren't present in the object constructor.

Java 7's try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

Single-expression functions

```
fun theAnswer() = 42
```

This is equivalent to

```
fun theAnswer(): Int {
    return 42
}
```

This can be effectively combined with other idioms, leading to shorter code. For example, with the `when` expression:

```
fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}
```

Call multiple methods on an object instance (with)

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { //draw a 100 pix square
    penDown()
    for (i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

https://kotlinlang.org/docs/coding-conventions.html

https://developer.android.com/kotlin/add-kotlin

Android KTX

Android KTX is a set of Kotlin extensions that are included with Android Jetpack and other Android libraries. KTX extensions provide concise, idiomatic Kotlin to Jetpack, Android platform, and other APIs. To do so, these extensions leverage several Kotlin language features, including the following:

- Extension functions
- Extension properties
- Lambdas
- Named parameters
- Parameter default values
- Coroutines

As an example, when working with SharedPreferences, you must create an editor before you can make modifications to the preferences data. You must also apply or commit those changes when you are finished editing, as shown in the following example:

```
sharedPreferences
    .edit()  // create an Editor
    .putBoolean("key", value)
    .apply() // write to disk asynchronously
```

Kotlin lambdas are a perfect fit for this use case. They allow you to take a more concise approach by passing a block of code to execute after the editor is created, letting the code execute, and then letting the SharedPreferences API apply the changes atomically.

Here's an example of one of the Android KTX Core functions, SharedPreferences.edit, which adds an edit function to SharedPreferences. This function takes an optional boolean flag as its first argument that indicates whether to commit or apply the changes. It also receives an action to perform on the SharedPreferences editor in the form of a lambda.

```
// SharedPreferences.edit extension function signature from Android KTX - Core
// inline fun SharedPreferences.edit(
//     commit: Boolean = false,
//     action: SharedPreferences.Editor.() -> Unit)


// Commit a new value asynchronously
sharedPreferences.edit { putBoolean("key", value) }


// Commit a new value synchronously
sharedPreferences.edit(commit = true) { putBoolean("key", value) }
```

The caller can choose whether to commit or apply the changes. The action lambda is itself an anonymous extension function on SharedPreferences.Editor which returns Unit, as indicated by its signature. This is why inside the block, you are able to perform the work directly on the SharedPreferences.Editor.

Use Android KTX in your project

To start using Android KTX, add the following dependency to your project's `build.gradle` file:

Groovy Kotlin

```
repositories {
    google()
}
```

AndroidX Modules

Android KTX is organized into modules, where each module contains one or more packages.

You must include a dependency for each module artifact in your app's `build.gradle` file. Remember to append the version number to the artifact. You can find the latest version numbers in each artifact's corresponding section in this topic.

Android KTX contains a single core module that provides Kotlin extensions for common framework APIs and several domain-specific extensions.

With the exception of the core module, all KTX module artifacts replace the underlying Java dependency in your `build.gradle` file. For example, you can replace a `androidx.fragment:fragment` dependency with `androidx.fragment:fragment-ktx`. This syntax helps to better manage versioning and does not add additional dependency declaration requirements.

Core KTX

The Core KTX module provides extensions for common libraries that are part of the Android framework. These libraries do not have Java-based dependencies that you need to add to build.gradle.

To include this module, add the following to your app's build.gradle file:

```
dependencies {
    implementation("androidx.core:core-ktx:1.16.0")
}
```

Here's a list of the packages that are contained in the Core KTX module:

- androidx.core.animation
- androidx.core.content
- androidx.core.content.res
- androidx.core.database
- androidx.core.database.sqlite
- androidx.core.graphics
- androidx.core.graphics.drawable
- androidx.core.location
- androidx.core.net
- androidx.core.os
- androidx.core.text
- androidx.core.transition
- androidx.core.util

- androidx.core.view
- androidx.core.widget

Collection KTX:

The Collection extensions contain utility functions for working with Android's memory-efficient collection libraries, including ArrayMap, LongSparseArray, LruCache, and others.

To use this module, add the following to your app's build.gradle file:

```
dependencies {
    implementation("androidx.collection:collection-ktx:1.5.0")
}
```

Fragment KTX:

The Fragment KTX module provides a number of extensions to simplify the fragment API.

To include this module, add the following to your app's build.gradle file:

```
dependencies {
    implementation("androidx.fragment:fragment-ktx:1.8.8")
}
```

With the Fragment KTX module, you can simplify fragment transactions with lambdas, for example:

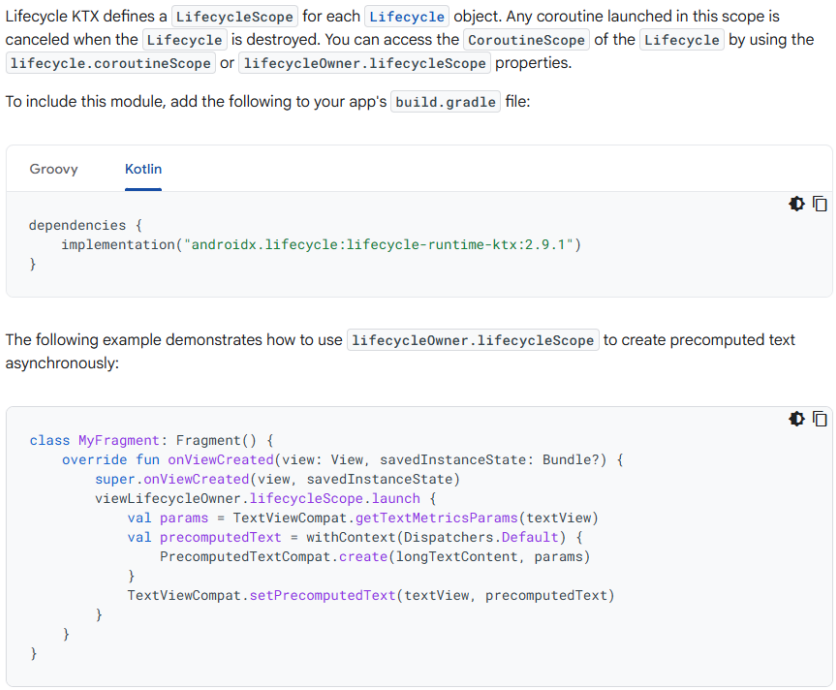
```
fragmentManager().commit {
   addToBackStack("...")
   setCustomAnimations(
         R.anim.enter_anim,
```

```
        R.anim.exit_anim)

    add(fragment, "...")

}
```

You can also bind to a ViewModel in one line by using the viewModels and activityViewModels property delegates:

```
// Get a reference to the ViewModel scoped to this Fragment

val viewModel by viewModels<MyViewModel>()

// Get a reference to the ViewModel scoped to its Activity

val viewModel by activityViewModels<MyViewModel>()
```

Lifecycle KTX:



https://developer.android.com/kotlin/ktx/extensions-list

Kotlin coroutines on Android:

A *coroutine* is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines were added to Kotlin in version 1.3 and are based on established concepts from other languages.

On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive. Over 50% of professional developers who use coroutines have reported seeing increased productivity. This topic describes how you can use Kotlin coroutines to address these problems, enabling you to write cleaner and more concise app code.

Features

Coroutines is our recommended solution for asynchronous programming on Android. Noteworthy features include the following:

- **Lightweight**: You can run many coroutines on a single thread due to support for *suspension*, which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations.
- **Fewer memory leaks**: Use *structured concurrency* to run operations within a scope.
- **Built-in cancellation support**: Cancellation is propagated automatically through the running coroutine hierarchy.
- **Jetpack integration**: Many Jetpack libraries include extensions that provide full coroutines support. Some libraries also provide their own coroutine scope that you can use for structured concurrency.

Use coroutines for main-safety

We consider a function *main-safe* when it doesn't block UI updates on the main thread. The `makeLoginRequest` function is not main-safe, as calling `makeLoginRequest` from the main thread does block the UI. Use the `withContext()` function from the coroutines library to move the execution of a coroutine to a different thread:

```
class LoginRepository(...) {
    ...
    suspend fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {

        // Move the execution of the coroutine to the I/O dispatcher
        return withContext(Dispatchers.IO) {
            // Blocking network request code
        }
    }
}
```

`withContext(Dispatchers.IO)` moves the execution of the coroutine to an I/O thread, making our calling function main-safe and enabling the UI to update as needed.

`makeLoginRequest` is also marked with the `suspend` keyword. This keyword is Kotlin's way to enforce a function to be called from within a coroutine.

Handling exceptions

To handle exceptions that the `Repository` layer can throw, use Kotlin's built-in support for exceptions. In the following example, we use a `try-catch` block:

```
class LoginViewModel(
    private val loginRepository: LoginRepository
): ViewModel() {

    fun login(username: String, token: String) {
        viewModelScope.launch {
            val jsonBody = "{ username: \"$username\", token: \"$token\"}"
            val result = try {
                loginRepository.makeLoginRequest(jsonBody)
            } catch(e: Exception) {
                Result.Error(Exception("Network request failed"))
            }
            when (result) {
                is Result.Success<LoginResponse> -> // Happy path
                else -> // Show error in UI
            }
        }
    }
}
```

In this example, any unexpected exception thrown by the `makeLoginRequest()` call is handled as an error in the UI.

Kotlin coroutines enable you to write clean, simplified asynchronous code that keeps your app responsive while managing long-running tasks such as network calls or disk operations.

This topic provides a detailed look at coroutines on Android. If you're unfamiliar with coroutines, be sure to read Kotlin coroutines on Android before reading this topic.

Manage long-running tasks

Coroutines build upon regular functions by adding two operations to handle long-running tasks. In addition to invoke (or call) and return, coroutines add suspend and resume:

- suspend pauses the execution of the current coroutine, saving all local variables.
- resume continues execution of a suspended coroutine from the place where it was suspended.

You can call suspend functions only from other suspend functions or by using a coroutine builder such as launch to start a new coroutine.

The following example shows a simple coroutine implementation for a hypothetical long-running task:

```
suspend fun fetchDocs() {                         // Dispatchers.Main
    val result = get("https://developer.android.com") // Dispatchers.IO for `get`
    show(result)                                  // Dispatchers.Main
}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /* ... */ }
```

In this example, get() still runs on the main thread, but it suspends the coroutine before it starts the network request. When the network request completes, get resumes the suspended coroutine instead of using a callback to notify the main thread.

Kotlin uses a *stack frame* to manage which function is running along with any local variables. When suspending a coroutine, the current stack frame is copied and saved for later. When resuming, the stack frame is copied back from where it was saved, and the function starts running again. Even though the code might look like an ordinary sequential blocking request, the coroutine ensures that the network request avoids blocking the main thread.

Use coroutines for main-safety

Kotlin coroutines use *dispatchers* to determine which threads are used for coroutine execution. To run code outside of the main thread, you can tell Kotlin coroutines to perform work on either the *Default* or *IO* dispatcher. In Kotlin, all coroutines must run in a dispatcher, even when they're running on the main thread. Coroutines can suspend themselves, and the dispatcher is responsible for resuming them.

To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

- **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling `suspend` functions, running Android UI framework operations, and updating `LiveData` objects.
- **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the Room component, reading from or writing to files, and running any network operations.
- **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

Continuing the previous example, you can use the dispatchers to re-define the `get` function. Inside the body of `get`, call `withContext(Dispatchers.IO)` to create a block that runs on the IO thread pool. Any code you put inside that block always executes via the `IO` dispatcher. Since `withContext` is itself a suspend function, the function `get` is also a suspend function.

```
suspend fun fetchDocs() {                      // Dispatchers.Main
    val result = get("developer.android.com")  // Dispatchers.Main
    show(result)                               // Dispatchers.Main
}

suspend fun get(url: String) =                 // Dispatchers.Main
    withContext(Dispatchers.IO) {              // Dispatchers.IO (main-safety block)
        /* perform network IO here */          // Dispatchers.IO (main-safety block)
    }                                          // Dispatchers.Main
}
```

Start a coroutine

You can start coroutines in one of two ways:

- launch starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.

- async starts a new coroutine and allows you to return a result with a suspend function called await.

Typically, you should launch a new coroutine from a regular function, as a regular function cannot call await. Use async only when inside another coroutine or when inside a suspend function and performing parallel decomposition.

Warning: launch and **async** handle exceptions differently. Since **async** expects an eventual call to **await**, it holds exceptions and rethrows them as part of the **await** call. This means if you use **async** to start a new coroutine from a regular function, you might silently drop an exception. These dropped exceptions won't appear in your crash metrics or be noted in logcat. For more information, see Cancellation and Exceptions in Coroutines.

Activity:

An activity is the entry point of an app for interacting with the user.

It represents a single screen with a user interface.

For example, an email app might have one activity that shows a list of new emails.

another activity to compose an email, and another activity for reading emails.

Although the activities work together to form a better user experience in the email app

An activity goes through a number of states over the course of its lifetime

a series of callbacks is used to handle transitions between states.

the callbacks are as follows:

- onCreate()
    - You must implement this callback, which fires when the system creates your activity.
    - You should initialize the essential components of your activity inside this callback
    - For example, your app should create views and bind data to lists here.
    - Most importantly, this is where you must call setContentView() to define the layout for the activity'suser interface.
    - When onCreate() finishes, the next callback is always onStart().
- onStart()
    - As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user.
    - This callback contains what amounts to the activity’s final preparations for coming to the foregroundand becoming interactive.
    - When onStart() finishes, the next callback is always onResume().
- onResume()
    - At this point, the activity is at the top of the activity stack
    - It captures all user input.
    - The onPause() callback always follows onResume().
- onPause()
    - The system calls onPause() when the activity loses focus and enters a Paused state. - This state occurs when, for example, the user taps the Back or Recents button.
    - When the system calls onPause() for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity, and the activity will soon enter the Stopped or Resumed state.
    - Once onPause() finishes executing the next callback is either onStop() or onResume(), depending on what happens after the activity enters the Paused state.
- onStop()
    - The system calls onStop() when the activity is no longer visible to the user.
    - This may happen because the activity is being destroyed or a new activity is starting and is covering the stopped activity.
    - In all of these cases, the stopped activity is no longer visible at all.
    - The next callback that the system calls is either onRestart(), if the activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating.
- onRestart()
    - The system invokes this callback when an activity in the Stopped state is about to restart. onRestart() restores the state of the activity from the time that it was stopped.

- This callback is always followed by onStart().

- onDestroy()
  - The system invokes this callback before an activity is destroyed.
  - This callback is the final one that the activity receives.
  - onDestroy() is usually implemented to ensure that all of an activity's resources are released when the activity, or the process containing it, is destroyed.

Logcat:

The Logcat window in Android Studio helps you debug your app by displaying logs from your device in real time—for example, messages that you added to your app with the Log class, messages from services that run on Android, or system messages, such as when a garbage collection occurs. When an app throws an exception, Logcat shows a message followed by the associated stack trace containing links to the line of code. Each log has a date, timestamp, process and thread ID, tag, package name, priority, and message associated with it. Different tags have a unique colour that helps identify the type of log. Each log entry has a priority of FATAL, ERROR, WARNING, INFO, DEBUG, or VERBOSE. we can Query logs using key-value search for example we can use

- ✓ tag: Matches against the tag field of the log entry.
- ✓ package: Matches against the package name of the logging app.
- ✓ message: Matches against the message part of the log entry.
- ✓ level: Matches the specified or higher severe log level–for example, DEBUG.

Log:

- ❖ API for sending log output.
- ❖ Generally, we should use the Log.v(), Log.d(), Log.i(), Log.w(), and Log.e() methods to write logs.
- ❖ we can then view the logs in logcat.
- ❖ Log.d : Send a DEBUG log message.
- ❖ Log.e : Send an ERROR log message.
- ❖ Log.i : Send a INFO log message.
- ❖ Log.v : Send a VERBOSE log message.
- ❖ Log.w : Send a WARN log message.

View, View Group and Layout:

A View is defined as the user interface which is used to create interactive UI components such as TextView, ImageView, EditText, RadioButton, etc. It is also responsible for event handling A ViewGroup act as a base class for that hold other Views or ViewGroups and is used to define the properties. These view groups are generally called layouts. Types of Android Layout:

- ❖ Linear Layout: It is a ViewGroup subclass, used to provide child View elements one by one in a particular direction either horizontally or vertically based on the orientation property. The default orientation is horizontal.
- ❖ Relative Layout: It is a ViewGroup subclass, used to specify the position of child View elements relative to each other like (A to the right of B) or relative to the parent (fix to the top of the parent).
- ❖ Constraint Layout: It is a ViewGroup subclass, used to specify the position of layout constraints for every child View relative to other views present. A ConstraintLayout is similar to a RelativeLayout, but having more power. used mostlty with drag and drop UI generation.
- ❖ Frame Layout: It is a ViewGroup subclass, used to specify the position of View elements it contains on the top of each other to display only a single View inside the FrameLayout.
- ❖ Table Layout: TableLayout is a ViewGroup subclass, used to display the child View elements in rows and columns.

Pixel in Android:

In Android, dp (density-independent pixels), sp (scale-independent pixels), and px (pixels), are units of measurement used to define dimensions in the layout and design of Android applications. Each of these units serves a specific purpose, particularly related to screen density and text size.

dp (Density-independent Pixels):

dp is a unit of measurement that is density-independent.

It is recommended to specify dimensions in layouts to ensure consistent visual size across different screen densities.

1 dp is equivalent to one physical pixel on a medium-density screen (160 dpi).

The actual size of a dp is adjusted based on the screen's density.

sp (Scale-independent Pixels):

sp is similar to dp, but it is specifically used for defining text size.

It takes into account the user's preferred text size setting in the device's system settings.

1 sp is the same as 1 dp, but it also considers the user's font size preference.

px(Pixels):

px is the most basic unit and represents a single pixel on the screen.

It is not recommended to specify dimensions in layouts due to its lack of density independence.

Text View:

It is a view used to display the textual data on screen We can set the text dynamically on this view from our program.

```
<TextView
android:textSize="20dp"
android:text="Phase-2 Hinjewadi"
android:layout_width="match_parent"
android:layout_height="wrap_content"/>
```

Edit Text: It is a view used to take the input from user.

```
<EditText
android:hint="Enter email"
android:inputType="textEmailAddress"
android:layout_width="match_parent"
android:layout_height="wrap_content"/>
```

Button:

It is a view that creates a clickable button on screen. It provides the onclick event that we can handle.

```
<Button
android:id="@+id/btnSave"
android:text="Save"
android:onClick="save"
android:layout_width="match_parent"
android:layout_height="wrap_content"/>


Button btnSave = findViewById(R.id.btnSave);
   btnSave.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View v) {
          Log.e(tag,"Save Button is Clicked");
      }
   });
// OR
public void save(View view) {
    Log.e(tag, "save() Button is Clicked");
 }
```

Checkbox:

It is a view that creates a checkbox

We can select multiple checkbox at same time

to add multpile ckeckbox add multiple checkBox Views

The view selection can be monitored by the method isSelected().

```
<CheckBox
android:id="@+id/checkboxC"
android:text="C"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
 <CheckBox
android:id="@+id/checkboxCPP"
android:text="CPP"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
```

Radio Button:

It is a view that creates a radioButton We can select only one radioButton at a time RadioButtons should be created inside a radioGroup to have only 1 selection The view selection can be monitored by the method isSelected().

```
<RadioGroup
android:orientation="horizontal"
android:layout_width="match_parent"
android:layout_height="wrap_content">

    <RadioButton
    android:id="@+id/radioMale"
    android:text="Male"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

    <RadioButton
    android:id="@+id/radioFemale"
    android:text="Female"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
</RadioGroup>
```

Spinner:

It is a view that creates a Dropdown Spinners provide a quick way to select one value from a set. - In the default state, a spinner shows its currently selected value. Tapping the spinner displays a menu showing all other values the user can select. To add a spinner to your layout.

```
<Spinner
    android:id="@+id/spinnerCourses"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

The choices you provide for the spinner can come from any source, but you must provide them through a SpinnerAdapter, such as an ArrayAdapter.

```
List<String> courses;
spinnerCourse = findViewById(R.id.spinnerCourse);
courses = new ArrayList<>();
courses.add("PG-DAC");
courses.add("PG-DMC");
courses.add("PG-DBDA");
```

```
ArrayAdapter arrayAdapter = new ArrayAdapter(this,

android.R.layout.simple_list_item_1,courses);

spinnerCourse.setAdapter(arrayAdapter);
```

When the user selects an item from the spinner's menu, the Spinner object receives an on-item selected event. To define the selection event handler for a spinner, implement the AdapterView.OnItemSelectedListener interface and the corresponding onItemSelected() callback method.

```
spinnerCourse.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {

@Override

public void onItemSelected(AdapterView<?> parent, View view int position, long id)

{

    Log.e("SpinnerActivity",courses.get(position));

}

@Override

 public void onNothingSelected(AdapterView<?> parent) {}

});

}
```

Array Adapter:

It provides views for an Adapter View 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.

Context:

This is an abstract class whose implementation is provided by the Android system. It provides global information about an application environment. It is used for application-level operations such as launching activities, broadcasting and receiving intents, etc. It allows us to access files and other resources such as pictures, Activities, Fragments and Services. The Context Class provides access to several resources and services that are needed to build an Android application.

There are two types of contexts:

- Application:
    - getApplicationContext() will give the application context
    - It is related with the life cycle of application
    - Application context is tied to the application itself and remains alive as long as the application is running.
    - This means that it can be used across multiple activities, and is useful for accessing global resources and classes that are not tied to any specific activity, such as shared preferences or database helpers.
- Activity:
    - getContext() will give the Activity context
    - It is related with the life cycle of activity

- An activity context should be used when you need access to resources or classes that are tied to a specific activity, such as views or resources that are specific to that activity.

Both "Activity" and "Application" are subclasses of the "Context"class, which provides access to application-specific resources and classes.

Toast:

- A toast provides simple feedback about an operation in a small popup.
- It only fills the amount of space required for the message and the current activity remains visible and interactive.
- Toasts automatically disappear after a timeout.
- If your app targets Android 12 (API level 31) or higher, its toast is limited to two lines of text and shows the application icon next to the text.
- Be aware that the line length of this text varies by screen size, so it's good to make the text as short as possible.
- To Instantiate a Toast object, Use the makeText() method, which takes the following parameters:
    1. The activity Context.
    2. The text that should appear to the user.
    3. The duration that the toast should remain on the screen.
- The makeText() method returns a properly initialized Toast object.
- To display the toast, call the show() method, as demonstrated in the following example:

  Toast.makeText(this, "Hello from toast", Toast.LENGTH_SHORT).show();
- If your app is in the foreground, consider using a snackbar instead of using a toast.
- Snackbars include user-actionable options, which can provide a better app experience.

SnackBar:

- It is similar to toast but with some advanced options.
- The snackbar component serves as a brief notification that appears at the bottom of the screen.
- It provides feedback about an operation or action without interrupting the user experience.
- Snackbars disappear after a few seconds.
- The user can also dismiss them with an action, such as tapping a button.
- Consider these three use cases where you might use a snack bar:
    1. Action Confirmation: After a user deletes an email or message, a snackbar appears to confirm the action and offer an "Undo" option.
    2. Network Status: When the app loses its internet connection, a snackbar pops up to note that it is now offline.
    3. Data Submission: Upon successfully submitting a form or updating settings, a snackbar notes that the change has saved successfully.

ImageView:

It is a view that can hold the images

It Displays image resources, for example Bitmap or Drawable resources.

```
<ImageView
    android:src="@drawable/my_image"
    android:layout_width="150dp"
    android:layout_height="150dp"/>
```

ScrollView:

- A view group that allows the view hierarchy placed within it to be scrolled.
- Scroll view may have only one direct child placed within it.
- To add multiple views within the scroll view, make the direct child you add a view group, for example
- LinearLayout, and place additional views within that LinearLayout.
- Scroll view supports vertical scrolling only. For horizontal scrolling, use HorizontalScrollView instead.
- Never add a RecyclerView or ListView to a scroll view, doing so results in poor user interface
- performance and a poor user experience.

Intent:

- It is a messaging object that is used to communicate between different components of android application
- It is mostly used to
    1. start an activity
    2. start a service
    3. deliver a broadcast.
- There are 2 types of intents:
    1. Implicit Intent:
        - It is used to start the component where we don't know the class name of the activity or service we want to start.
        - Here general action is specified to start the components
        - For example, if you want to show the user a location on a map, you can use an implicit intent to request that map application from your app to show a specified location.
        - When you use an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device.
        - If the intent matches an intent filter, the system starts that component and delivers it the Intent object.
        - If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.
        - To start the component of android application using Implicit Intent we need to have the information about the intent filter of that application
        - Intent filter is given for the component inside the manifest file
        - Intent filter consists of 3 elements
            1. Action:
                It determines the action for an intent
                It should be the string value of an action
                It should not be the classname string constant
            2. data:
                It determines what type of data can be passed to this component
                eg -> URI(scheme,host,port,path) and MIME type.
            3. category:
                It determines the intent action belong to which type of category
                It should be the string value of an category
                It should not be the classname string constant

```
public void btnCall(View view){
      Intent intent = new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:8983049388"));
      startActivity(intent);
   }
public void btnBrowse(View view){
   Intent intent = new
Intent(Intent.ACTION_VIEW,Uri.parse("https://sunbeaminfo.com/placements"));
   startActivity(intent);
   }
public void btnSendMessage(View view){
   Intent intent = new Intent(Intent.ACTION_SEND);
    intent.putExtra(Intent.EXTRA_TEXT, "Hello");
    intent.setType("text/plain");
    startActivity(intent);
}
```

2. Explicit Intent:

- It is used specify which component of which application will satisfy the intent, by specifying a full ComponentName
- It is used to start a component in your own app, because you know the class name of the activity or service you want to start. For example, you might start a new activity within your app in response to a user action, or start a service to download a file in the background.
- To start the component of android application using Explicit Intent we need to have the information about the Fully Qualified Classname of that component from android application.

```
public void btnSecond(View view){
Intent intent = new Intent(this, SecondActivity.class);
startActivity(intent);
}
//OR
public void btnSecond(View view){
Intent intent = new Intent(this, SecondActivity.class);
intent.putExtra("k_name","v_sunbeam");
startActivity(intent);
}
// in second activity get the data if passed through intent
Intent intent = getIntent();
String name =  intent.getStringExtra("k_name");
Log.e(tag,name);
```

Starting the Activity for Result:

- Starting another activity, whether it is one within your app or from another app, doesn't need to be a one-way operation.
- You can also start an activity and receive a result back.
- For example, your app can start a camera app and receive the captured photo as a result. Or you might start the Contacts app for the user to select a contact, and then receive the contact details as a result.
- The Activity Result APIs provide components for registering for a result, launching the activity that produces the result, and handling the result once it is dispatched by the system.
- When in a ComponentActivity or a Fragment, the Activity Result APIs provide a registerForActivityResult() API for registering the result callback. registerForActivityResult() takes an ActivityResultContract and an ActivityResultCallback and returns an ActivityResultLauncher, which you use to launch the other activity.

```
ActivityResultLauncher activityResultLauncher =
registerForActivityResult(new
ActivityResultContracts.StartActivityForResult(), new
ActivityResultCallback<ActivityResult>() {
@Override
public void onActivityResult(ActivityResult result) {
      }
});
public void getStudentData(View v){
      Intent intent = new Intent(this, InputStudentDetails.class);
      activityResultLauncher.launch(intent);
   }
```

APP Bar (Action Bar):

- The app bar, also known as the action bar, is one of the most important design elements in your app's
- activities, because it provides a visual structure and interactive elements that are familiar to users.
- It is a view which is present at top of the activity
- The key functions of the app bar are as follows:
    1. Dedicated space for giving your app an identity and indicating the user's location in the app.
    2. Predictable access to important actions, such as search.
    3. Support for navigation and view switching, using tabs or menus.
- In its most basic form, the action bar displays the title for the activity on one side and an overflow menu on the other.
- If you want the action bar on all activities then change the themes.xml file.
- Select the theme as Theme.Material3.DayNight into themes.xml file
- If you don’t want the action bar on specific activities and then hide it using below method.

  ```
  getSupportActionBar().hide();
  ```
- If you want to add the title to action bar then:

  ```
  getSupportActionBar().setTitle("Details");
  ```
- To provide the back action button on action bar add the parent attribute in manifest file for that corresponding activity

  ```
  <activity
  android:parentActivityName=".MainActivity"
  android:name=".SecondActivity"
  android:exported="false" />
  ```
- As we don’t require the action bar over here we can hide it using the hide().

ToolBar:

- It is similar to the ActionBar but is more cutomizable
- Action Bar is part of Activity where as the tool bar is a View Group
- A Toolbar is a generalization of action bars for use within application layouts. While an action bar is traditionally part of an Activity's opaque window decor controlled by the framework, a - Toolbar may be placed at any arbitrary level of nesting within a view hierarchy.

- An application may choose to designate a Toolbar as the action bar for an Activity using the setSupportActionBar() method.

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbBar"
    android:background="@android:color/darker_gray"
    app:title="Register"
    app:titleTextColor="@color/white"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

```
Toolbar toolbar = findViewById(R.id.toolBar);

setSupportActionBar(toolbar);
```

Menus:

- Menus are a common user interface component
- There are 3 types of menus:
    1. Options menu:
        - The options menu is the primary collection of menu items for an activity.
        - It's where you place actions such as "Search," "Compose email," and "Settings".
    2. Popup menu
        - A popup menu displays a vertical list of items that's attached to the view that invokes the menu.
        - we can display such menus on long click on the list view item.
    3. Context menu
        - A context menu is a floating menu that appears when the user performs a touch & hold on an element.
        - It provides actions that affect the selected content or context frame.
        - You can provide a context menu for any view, but they are most often used for items in a RecylerView or other view collections in which the user can perform direct actions on each item.

```
// Options Menu

 @Override

 public boolean onCreateOptionsMenu(Menu menu)

 {

//menu.add("Add");

//menu.add("Settings");

   getMenuInflater().inflate(R.menu.mainmenu,menu);

return super.onCreateOptionsMenu(menu);

}

@Override

 public boolean onOptionsItemSelected(@NonNull MenuItem item) {

if(item.getTitle().equals("Add"))

    Toast.makeText(this, "Add", Toast.LENGTH_SHORT).show();

if(item.getTitle().equals("Settings"))

    Toast.makeText(this, "Settings", Toast.LENGTH_SHORT).show();

return super.onOptionsItemSelected(item);

}
```

```
// Popup Menu
 public void popupMenu(View view){
   PopupMenu popupMenu = new PopupMenu(this,view);
   popupMenu.getMenu().add("Edit");
   popupMenu.getMenu().add("Delete");
   popupMenu.show();
   popupMenu.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
@Override
public boolean onMenuItemClick(MenuItem item) {
if(item.getTitle().equals("Edit"))
            Toast.makeText(MainActivity.this, "Edit",
Toast.LENGTH_SHORT).show();
if(item.getTitle().equals("Delete"))
            Toast.makeText(MainActivity.this, "Delete",
Toast.LENGTH_SHORT).show();
return false;
       }
   });
}
//Context Menu
@Override
 public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
super.onCreateContextMenu(menu, v, menuInfo);
  menu.add("Edit");
   menu.add("Delete");
}
@Override
public boolean onContextItemSelected(@NonNull MenuItem item) {
if(item.getTitle().equals("Edit"))
      Toast.makeText(MainActivity.this, "Edit", Toast.LENGTH_SHORT).show();
if(item.getTitle().equals("Delete"))
      Toast.makeText(MainActivity.this, "Delete", Toast.LENGTH_SHORT).show();
return super.onContextItemSelected(item);
}
```

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto">
<item
app:showAsAction="always"
android:icon="@drawable/add"
android:title="Add"/>
<item
app:showAsAction="collapseActionView"
android:title="Settings"/>
</menu>
```

ListView:

- Displays a vertically-scrollable collection of views
- each view is positioned immediately below the previous view in the list.
- It does not know the details, such as type and contents, of the views it contains.
- Instead list view requests views on demand from the Adapter as needed, such as to display new views as the user scrolls up or down.
- For more advanced features and performance it is recommended to use RecyclerView.

```
<ListView
android:id="@+id/listView"
android:layout_width="match_parent"
android:layout_height="wrap_content"/>
```

Recycler View:

- RecyclerView makes it easy to efficiently display large sets of data.
- You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're needed.
- As the name implies, RecyclerView recycles those individual elements.
- When an item scrolls off the screen, RecyclerView doesn't destroy its view. Instead, RecyclerView reuses the view for new items that have scrolled onscreen.
- RecyclerView improves performance and your app's responsiveness, and it reduces power consumption.
- It is a ViewGroup which is used to display the data in the form of List or Grid
- It is more efficient and customizable than the ListView
- RecyclerView is the ViewGroup that contains the views corresponding to your data.
- It's a view itself, so you add RecyclerView to your layout the way you would add any other UI element.

Implementation of Recylcer View, adapter and view holder:

activity_main.xml

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

CountryAdapter.java

```
public class CountryAdapter extends
        RecyclerView.Adapter<CountryAdapter.MyViewHolder> {
        Context context;
        List<String> countries;
        public CountryAdapter(Context context, List<String> countries) {
                this.context = context;
                this.countries = countries;
        }
        @NonNull
         @Override
         public MyViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int
        viewType) {
           View view =
        LayoutInflater.from(context).inflate(R.layout.countries_list,null);
                return new MyViewHolder(view);
        }
@Override
 public void onBindViewHolder(@NonNull MyViewHolder holder, int position) {
   holder.textCountry.setText(countries.get(position));
}
@Override
 public int getItemCount() {
return countries.size();
}
class MyViewHolder extends RecyclerView.ViewHolder{
   TextView textCountry;
public MyViewHolder(@NonNull View itemView) {
super(itemView);
      textCountry = itemView.findViewById(R.id.textCountry);
   }
}
}
```

MainActivity:

// Inside MainActivity

```
RecyclerView recyclerView;

List<String> countries = new ArrayList<>();

recyclerView = findViewById(R.id.recyclerView);

countries.add("India");

countries.add("USA");

countries.add("UK");

countries.add("China");

countries.add("Japan");

CountryAdapter countryAdapter = new CountryAdapter(this,countries);

recyclerView.setAdapter(countryAdapter);

recyclerView.setLayoutManager(new LinearLayoutManager(this));

// OR

recyclerView.setLayoutManager(new GridLayoutManager(this,1));
```

- The ViewHolder is a wrapper around a View that contains the layout for an individual item in the list.
- The Adapter creates ViewHolder objects as needed and also sets the data for those views.
- The process of associating views to their data is called binding.
- When you define your adapter, you override three key methods:
    1. onCreateViewHolder()
        - RecyclerView calls this method whenever it needs to create a new ViewHolder.
        - The method creates and initializes the ViewHolder and its associated View, but does not fill in the view's contents—the ViewHolder has not yet been bound to specific data.
    2. onBindViewHolder()
        - RecyclerView calls this method to associate a ViewHolder with data.
        - The method fetches the appropriate data and uses the data to fill in the view holder's layout.
        - For example, if the RecyclerView displays a list of names, the method might find the appropriate name in the list and fill in the view holder's TextView widget.
    3. getItemCount()
        - RecyclerView calls this method to get the size of the dataset.
        - For example, in an address book app, this might be the total number of addresses.
        - RecyclerView uses this to determine when there are no more items that can be displayed.

LayoutManager:

- The items in your RecyclerView are arranged by a LayoutManager class.
- The RecyclerView library provides three layout managers, which handle the most common layout situations
    1. LinearLayoutManager arranges the items in a one-dimensional list.
    2. GridLayoutManager arranges the items in a two-dimensional grid
        - If the grid is arranged vertically, GridLayoutManager tries to make all the elements in each row have the same width and height, but different rows can have different heights.
        - If the grid is arranged horizontally, GridLayoutManager tries to make all the elements in each column have the same width and height, but different columns can have different widths.

3. StaggeredGridLayoutManager is similar to GridLayoutManager, but it does not require that items in a row have the same height (for vertical grids) or items in the same column have the same width (for horizontal grids). The result is that the items in a row or column can end up offset from each other.

```
// GridLayoutManager ->
recyclerView.setLayoutManager(new
GridLayoutManager(this,1,LinearLayoutManager.VERTICAL,false));
recyclerView.setLayoutManager(new
GridLayoutManager(this,1,LinearLayoutManager.HORIZONTAL,false));
recyclerView.setLayoutManager(new
GridLayoutManager(this,5,LinearLayoutManager.VERTICAL,false));
recyclerView.setLayoutManager(new
GridLayoutManager(this,5,LinearLayoutManager.HORIZONTAL,false));
```

Click Animation:

- The recyclerview do not show any animation effect when clicked.
- If we want the ripple/hover effect on click of the view inside recyclerview then add the below attributes inside the view that needs to be animated when clicked.

```
<TextView
 android:background="?android:attr/selectableItemBackground"
 android:clickable="true"
 android:text="TextView that animate when clicked"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"/>
```

Fragments:

- A Fragment represents a reusable portion of your app's UI.
- It defines and manages its own layout, has its own lifecycle, and can handle its own input events.
- Fragments can't live on their own. They must be hosted by an activity or another fragment.
- Create a new Blank Fragment and name it as FirstFragment
- To load the fragment into mainActivity add the below code in activity_main.xml file.

```
<androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_container_view"
       android:name="com.sunbeaminfo.app1.FirstFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>
```

- Fragment lifecycle is managed by its callback methods which are as below:
    1. onCreate
    2. onCreateView
    3. onViewCreated
    4. onViewStateRestored
    5. onStart
    6. onResume
    7. onPause
    8. onStop
    9. onDestroy

ViewPager2:

It is used to display Fragments in a swipe able format. viewpager requires an custom adapter which will provide the information about the fragments.

Android Storage:

Android provides several options to save your apps data:

1. App specific Storage: Store files that are meant for your app's use only, either in dedicated directories within an internal storage volume or different dedicated directories within external storage.
2. Shared Storage: Store files that your app intends to share with other apps, including media, documents, and other files.
3. Preferences (Shared Preferences): Store private, primitive data in key-value pairs.
4. Database (Sqlite Database): Store structured data in a private database using the Room persistence library.

1. App Specific Storage:

- In many cases, your app creates files that other apps don't need to access, or shouldn't access.
- The system provides the following locations for storing such app-specific files:
    1. Internal storage directories:
        - These directories include both a dedicated location for storing persistent files,and another location for storing cache data.
        - The system prevents other apps from accessing these locations, and on Android 10 (API level 29) and higher, these locations are encrypted.
        - These characteristics make these locations a good place to store sensitive data that only your app itself can access.
    2. External storage directories:
        - These directories include both a dedicated location for storing persistent files, and another location for storing cache data.
        - Although it's possible for another app to access these directories if that app has the proper permissions, the files stored in these directories are meant for use only by your app.
        - If you specifically intend to create files that other apps should be able to access, your app should store these files in the shared storage part of external storage instead.
        - To Access and store files we can use the File API to access and store files.

2. Shared Storage:

- Use shared storage for user data that can or should be accessible to other apps and saved even if the user uninstalls your app.
- Android provides APIs for storing and accessing the following types of shareable data:
    1. Media content:
        - The system provides standard public directories for these kinds of files, so the user has a common location for all their photos, another common location for all their music and audio files, and so on.
        - Your app can access this content using the platform's MediaStore API.
    2. Documents and other files:
        - The system has a special directory for containing other file types, such as PDF documents and books.
        - Your app can access these files using the platform's Storage Access Framework.
    3. Datasets:
        - On Android 11 (API level 30) and higher, the system caches large datasets that multiple apps might use.
        - These datasets can support use cases like machine learning and media playback.

- Apps can access these shared datasets using the BlobStoreManager API.

3. Sqlite Database:

- Saving data to a database is ideal for repeating or structured data, such as contact information.
- It is a mini version of actual sql database for android.
- It is used to store the structured data inside it.
- The SQLiteOpenHelper class contains a useful set of APIs for managing your database.
- When you use this class to obtain references to your database, the system performs the potentially long-running operations of creating and updating the database only when needed and not during app startup.
- All you need to do is call getWritableDatabase() or getReadableDatabase().
- To use SQLiteOpenHelper, create a subclass that overrides the onCreate() and onUpgrade() callback methods.

```
public class DbHelper extends SQLiteOpenHelper {
// ctor that creates database with the given name and version
public DbHelper(@Nullable Context context) {
            super(context, "dmc_db", null, 1);
   }
// This creates the required tables in the database.
// gets called only once
@Override
public void onCreate(SQLiteDatabase db) {
      Log.e("Sqlite","oncreate called");
      String sql = "CREATE TABLE employee(empid INTEGER PRIMARY KEY
AUTOINCREMENT,name TEXT,sal REAL)";
      db.execSQL(sql);
   }
 // This method gets called if new tables are added or dropped.
// gets called only when version is changed
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {}
// method to insert the employee in database
public void insertEmployee(Employee employee){
      SQLiteDatabase db =  getWritableDatabase();
      ContentValues values = new ContentValues();
      values.put("name",employee.getName());
      values.put("sal",employee.getSal());
      db.insert("employee",null,values);
      db.close();
   }
// method to get all the employees from database
public List<Employee> getEmployees() {
      List<Employee> employeeList = new ArrayList()
      SQLiteDatabase db = getReadableDatabase();
      Cursor cursor = db.query("employee",null,null,null,null,null,null);
while(cursor.moveToNext()){
         Employee e = new Employee();
         e.setEmpid(cursor.getInt(0));
         e.setName(cursor.getString(1));
         e.setSal(cursor.getDouble(2));
         employeeList.add(e);
      }
      db.close();
return employeeList;
   }
// method to edit the employee from database
public void editEmployee(int empid,Employee employee) {
      SQLiteDatabase db = getWritableDatabase();
```

```
        ContentValues values = new ContentValues();
        values.put("sal",employee.getSal());
        db.update("employee",values,"empid=?",new String[]{empid+""});
        db.close();
    }
// method to delete the employee from database
public void deleteEmployee(int empid) {
        SQLiteDatabase db = getWritableDatabase();
        db.delete("employee","empid=?",new String[]{empid+""});
        db.close();
    }
}
```

Room Persistance Library

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite. In particular, Room provides the following benefits:

1. Compile-time verification of SQL queries.

2. Convenience annotations that minimize repetitive and error-prone boilerplate code.

3. Streamlined database migration paths.

Because of these considerations, it is highly recommend that you use Room instead of using the SQLite APIs directly.

To implement the crud operations using Room Persistance Library, add the dependencies inside your gradle

```
def room_version = "2.7.1"
implementation "androidx.room:room-runtime:$room_version"
annotationProcessor "androidx.room:room-compiler:$room_version"
```

```
val room_version = "2.7.1"
implementation("androidx.room:room-runtime:$room_version")
annotationProcessor("androidx.room:room-compiler:$room_version")
```

There are three major components in Room:

1. The Database class

that holds the database and serves as the main access point for the underlying connection to your app's persisted data.

2. Data Entities

that represent tables in your app's database.

3. Data Access Objects (DAOs)

that provide methods that your app can use to query, update, insert, and delete data in the database.

Steps for the implementation

1. Create and entity class Person with the annotaions that will represent the fields of table in database

```
@Entity(tableName = "person")
public class Person {
@PrimaryKey(autoGenerate = true)
private int id;
@ColumnInfo(name = "pname")
private  String name;
@ColumnInfo
private  int age;
}
```

2. Cretae the package db and inside it add the PersonDao interface which will have methods along with annotaion for perfoming the CRUD operations.

```
@Dao
public interface PersonDao {
@Insert
void insertPerson(Person person);
@Query("SELECT * FROM person")
List<Person> getData();
@Query("SELECT * FROM person WHERE age>:age")
List<Person> getSpecificPersons(int age);
@Query("DELETE FROM person WHERE id=:id")
void deletePerson(int id);
@Query("UPDATE person set name=:name where id=:id")
void updatePerson(int id,String name);
}
```

3. Create an abstract class PersonDatabase which represent the database, name, and the version. use proper annotations. It will be used tp return the Dao objects

here the class name should be kept as AppDatabase as it is an applicatio level database. however for better understanding and done for single person class we are usnig the name as PersonDatabase

Day08_Help.MD
Sunbeam Infotech
2025-05-20

```
@Database(entities = {Person.class},version = 1)
public abstract class PersonDatabase extends RoomDatabase {
private static PersonDatabase personDatabase = null;
public abstract PersonDao personDao();
//method that returns the object of PersonDatabase
public static PersonDatabase getInstance(Context context){
if(personDatabase == null){
        personDatabase =
Room.databaseBuilder(context,PersonDatabase.class,"person_db")
                . allowMainThreadQueries()
                . build();
      }
return personDatabase;
   }
}
```

4. use this class to call the methods declared in the PersonDao interface.

5. When any changes is done in the db and need to be reflected inside the recycler view it might not refresh the recyclerview with the method notifyDatasetChanged() as we are creating multiple objects of ArrayList, so for this we need to use setter for that collection use it inside the adapter.

```
// In Adapter
public void setPersonList(List<Person> personList) {
this.personList = personList;
 notifyDataSetChanged();
}
```

Content Provider:

- A content provider manages access to a central repository of data.
- A provider is part of an Android application, which often provides its own UI for working with the data.
- However, content providers are primarily used by other applications, which access the provider using a provider client object.
- Together, providers and provider clients offer a consistent, standard interface to data that also handles interprocess communication and secure data access.
- Typically, you work with content providers in one of two scenarios: implementing code to access an existing content provider in another application or creating a new content provider in your application to share data with other applications.

Content Resolver:

- When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client.
- The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider.
- The provider object receives data requests from clients, performs the requested action, and returns the results.
- This object has methods that call identically named methods in the provider object, an instance of one of the concrete subclasses of ContentProvider.
- The ContentResolver methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

Content URI's:

- A content URI is a URI that identifies data in a provider
- Content URIs include the symbolic name of the entire provider—its authority and a name that points to a table—a path
- When you call a client method to access a table in a provider, the content URI for the table is one of the arguments.
- The ContentProvider uses the path part of the content URI to choose the table to access.
- A provider usually has a path for each table it exposes.
  content://user_dictionary/words
- The content:// string is the scheme, which is always present and identifies this as a content URI.
- The user_dictionary string is the provider's authority.
- The words string is the table's path.

A content provider presents data to external applications as one or more tables that are similar to the tables found in a relational database. A row represents an instance of some type of data the provider collects, and each column in the row represents an individual piece of data collected for an instance.

A content provider coordinates access to the data storage layer in your application for a number of different APIs and components. As illustrated in figure 1, these include the following:

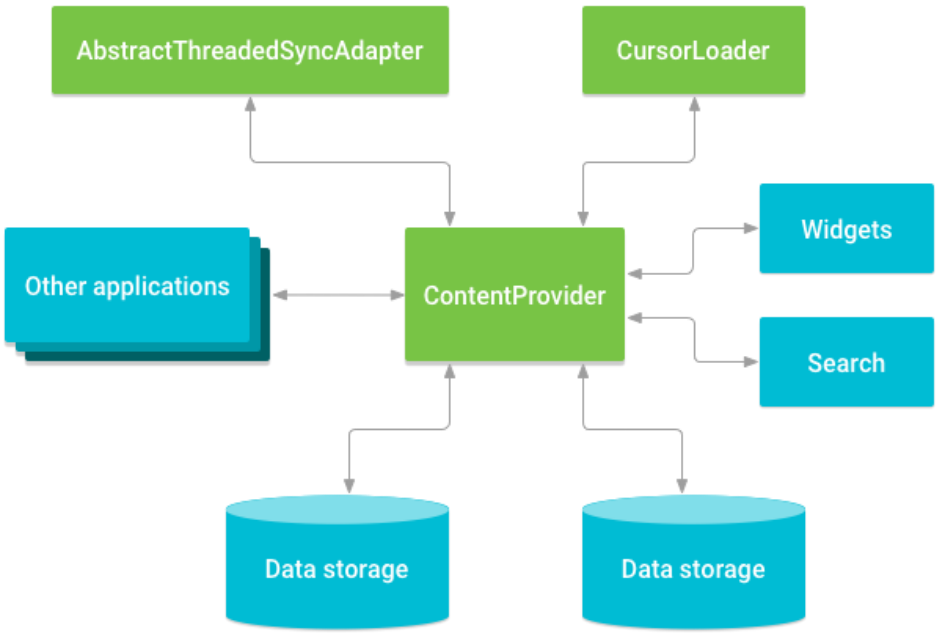
Sharing access to your application data with other applications

Sending data to a widget

Returning custom search suggestions for your application through the search framework using SearchRecentSuggestionsProvider

Synchronizing application data with your server using an implementation of AbstractThreadedSyncAdapter

Loading data in your UI using a CursorLoader



Access a provider:

When you want to access data in a content provider, you use the ContentResolver object in your application's Context to communicate with the provider as a client. The ContentResolver object communicates with the provider object, an instance of a class that implements ContentProvider.

The provider object receives data requests from clients, performs the requested action, and returns the results. This object has methods that call identically named methods in the provider object, an instance of one of the concrete subclasses of ContentProvider. The ContentResolver methods provide the basic "CRUD" (create, retrieve, update, and delete) functions of persistent storage.

A common pattern for accessing a ContentProvider from your UI uses a CursorLoader to run an asynchronous query in the background. The Activity or Fragment in your UI calls a CursorLoader to the query, which in turn gets the ContentProvider using the ContentResolver.

This lets the UI continue to be available to the user while the query is running. This pattern involves the interaction of a number of different objects, as well as the underlying storage mechanism, as illustrated in figure 2.

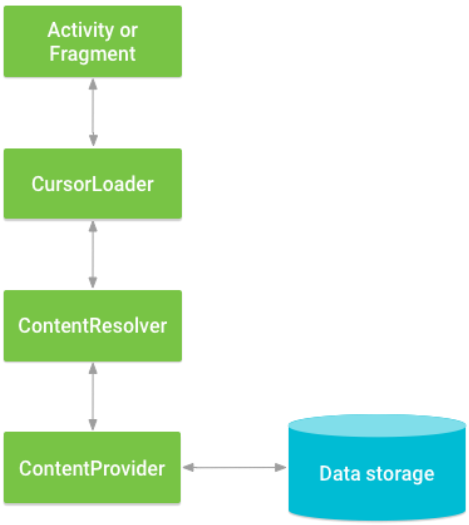


Table 2 shows how the arguments to `query(Uri,projection,selection,selectionArgs,sortOrder)` match a SQL SELECT statement:

Table 2: `query()` compared to SQL query.

`query()` argument	SELECT keyword/parameter	Notes
`Uri`	`FROM table_name`	`Uri` maps to the table in the provider named *table_name*.
`projection`	`col,col,col,...`	`projection` is an array of columns that is included for each row retrieved.
`selection`	`WHERE col = value`	`selection` specifies the criteria for selecting rows.
`selectionArgs`	No exact equivalent. Selection arguments replace `?` placeholders in the selection clause.	
`sortOrder`	`ORDER BY col, col,...`	`sortOrder` specifies the order in which rows appear in the returned `Cursor`.

Insert, update, and delete data

In the same way that you retrieve data from a provider, you also use the interaction between a provider client and the provider's `ContentProvider` to modify data. You call a method of `ContentResolver` with arguments that are passed to the corresponding method of `ContentProvider`. The provider and provider client automatically handle security and interprocess communication.

Insert data

To insert data into a provider, you call the `ContentResolver.insert()` method. This method inserts a new row into the provider and returns a content URI for that row. The following snippet shows how to insert a new word into the User Dictionary Provider:

Kotlin Java

```
// Defines a new Uri object that receives the result of the insertion
lateinit var newUri: Uri
...
// Defines an object to contain the new values to insert
val newValues = ContentValues().apply {
    /*
     * Sets the values of each column and inserts the word. The arguments to the "put"
     * method are "column name" and "value".
     */
    put(UserDictionary.Words.APP_ID, "example.user")
    put(UserDictionary.Words.LOCALE, "en_US")
    put(UserDictionary.Words.WORD, "insert")
    put(UserDictionary.Words.FREQUENCY, "100")

}

newUri = contentResolver.insert(
        UserDictionary.Words.CONTENT_URI,    // The UserDictionary content URI
        newValues                            // The values to insert
)
```

Update data

To update a row, you use a `ContentValues` object with the updated values, just as you do with an insertion, and selection criteria, as you do with a query. The client method you use is `ContentResolver.update()`. You only need to add values to the `ContentValues` object for columns you're updating. If you want to clear the contents of a column, set the value to `null`.

The following snippet changes all the rows whose locale has the language `"en"` to a have a locale of `null`. The return value is the number of rows that were updated.

Kotlin Java

```
// Defines an object to contain the updated values
val updateValues = ContentValues().apply {
    /*
     * Sets the updated value and updates the selected words.
     */
    putNull(UserDictionary.Words.LOCALE)
}

// Defines selection criteria for the rows you want to update
val selectionClause: String = UserDictionary.Words.LOCALE + "LIKE ?"
val selectionArgs: Array<String> = arrayOf("en_%")

// Defines a variable to contain the number of updated rows
var rowsUpdated: Int = 0
...
rowsUpdated = contentResolver.update(
        UserDictionary.Words.CONTENT_URI,  // The UserDictionary content URI
        updateValues,                      // The columns to update
        selectionClause,                   // The column to select on
        selectionArgs                      // The value to compare to
)
```

Delete data

Deleting rows is similar to retrieving row data. You specify selection criteria for the rows you want to delete, and the client method returns the number of deleted rows. The following snippet deletes rows whose app ID matches `"user"`. The method returns the number of deleted rows.

Kotlin Java

```
// Defines selection criteria for the rows you want to delete
val selectionClause = "${UserDictionary.Words.APP_ID} LIKE ?"
val selectionArgs: Array<String> = arrayOf("user")

// Defines a variable to contain the number of rows deleted
var rowsDeleted: Int = 0
...
// Deletes the words that match the selection criteria
rowsDeleted = contentResolver.delete(
        UserDictionary.Words.CONTENT_URI,  // The UserDictionary content URI
        selectionClause,                   // The column to select on
        selectionArgs                      // The value to compare to
)
```

Provider data types

Content providers can offer many different data types. The User Dictionary Provider offers only text, but providers can also offer the following formats:

- integer
- long integer (long)
- floating point
- long floating point (double)

Another data type that providers often use is a binary large object (BLOB) implemented as a 64 KB byte array. You can see the available data types by looking at the `Cursor` class "get" methods.

The data type for each column in a provider is usually listed in its documentation. The data types for the User Dictionary Provider are listed in the reference documentation for its contract class, `UserDictionary.Words`. Contract classes are described in the Contract classes section. You can also determine the data type by calling `Cursor.getType()`.

Providers also maintain MIME data type information for each content URI they define. You can use the MIME type information to find out whether your application can handle data that the provider offers or to choose a type of handling based on the MIME type. You usually need the MIME type when you work with a provider that contains complex data structures or files.

For example, the `ContactsContract.Data` table in the Contacts Provider uses MIME types to label the type of contact data stored in each row. To get the MIME type corresponding to a content URI, call `ContentResolver.getType()`.

Batch access

Batch access to a provider is useful for inserting a large number of rows, for inserting rows in multiple tables in the same method call, and in general for performing a set of operations across process boundaries as a transaction, called an *atomic operation*.

To access a provider in batch mode, create an array of `ContentProviderOperation` objects and then dispatch them to a content provider with `ContentResolver.applyBatch()`. You pass the content provider's authority to this method, rather than a particular content URI.

This lets each `ContentProviderOperation` object in the array work against a different table. A call to `ContentResolver.applyBatch()` returns an array of results.

The description of the `ContactsContract.RawContacts` contract class includes a code snippet that demonstrates batch insertion.

Data access using intents

Intents can provide indirect access to a content provider. You can let the user access data in a provider even if your application doesn't have access permissions by either getting a result intent back from an application that has permissions or by activating an application that has permissions and letting the user do work in it.

Get access with temporary permissions

You can access data in a content provider, even if you don't have the proper access permissions, by sending an intent to an application that does have the permissions and receiving back a result intent containing URI permissions. These are permissions for a specific content URI that last until the activity that receives them is finished. The application that has permanent permissions grants temporary permissions by setting a flag in the result intent:

- **Read permission:** `FLAG_GRANT_READ_URI_PERMISSION`
- **Write permission:** `FLAG_GRANT_WRITE_URI_PERMISSION`

★ **Note:** These flags don't give general read or write access to the provider whose authority is contained in the content URI. The access is only for the URI itself.

SharedPreferences:

- It is used to store primitive data in keyValue pairs
- If you have a relatively small collection of key-values that you'd like to save, you can use the SharedPreferences APIs.
- A SharedPreferences object points to a file containing key-value pairs and provides simple methods to read and write them.
- Each SharedPreferences file is managed by the framework and can be private or shared.
- To Get a handle to shared preferences we can create a new shared preference file or access an existing one by calling the method getSharedPreferences().
- we can call this from any Context in our app.
- When naming your shared preference files, you should use a name that's uniquely identifiable to your
- app.
- A good way to do this is prefix the file name with your application ID.
- For example: "com.example.myapp.PREFERENCE_FILE_KEY"
- To write to a shared preferences file, create a SharedPreferences.Editor by calling edit() on your SharedPreferences.
- Pass the keys and values you want to write with methods such as: putInt() andputString().
- Then call apply () or commit () to save the changes.
- To retrieve values from a shared preferences file, call methods such as getInt() and getString(), providing the key for the value you want, and optionally a default value to return if the key isn't present.

```
// To add the data in Shared preference
getSharedPreferences("com.sunbeaminfo.app1.PERSON_SP",MODE_PRIVATE).edit().putBool
 ean("login_status",true).apply();
// To get the data from Shared preference
 boolean status =
getSharedPreferences("com.sunbeaminfo.app1.PERSON_SP",MODE_PRIVATE).getBoolean("lo
 gin_status",false);
```

Animations:

There are two types of animation:

1. Property Animation
   - An animation defined in XML that modifies properties of the target object, such as background color or alpha value, over a set amount of time.
   - Here we use the res/animator dir to keep our animator xml file
2. Tween Animation
   - An animation defined in XML that performs transitions on a graphic such as rotating, fading, moving, and stretching.
   - Here we use the res/anim dir to keep our animator xml file

Tween Animation:

An animation defined in XML that performs transitions on a graphic such as rotating, fading, moving, and stretching. The file must have a single root element: either an <alpha>, <scale>, <translate>, <rotate>, or <set> element that holds a group (or groups) of other animation elements (including nested elements).

1. alpha

A fade-in or fade-out animation. Represents an AlphaAnimation.

Attributes:

android:fromAlpha

Float. Starting opacity offset, where 0.0 is transparent and 1.0 is opaque.

android:toAlpha

Float. Ending opacity offset, where 0.0 is transparent and 1.0 is opaque.

2. rotate

It rotates the view in clockwise direction pivoted at the position(0,0) of x and y from the top left cornerof Image view.

to keep the position(0,0) at the center of the view pivot the x and y values to 50%

Attributes:

android:fromDegrees

Float. Starting angular position, in degrees.

android:toDegrees

Float. Ending angular position, in degrees.

android:pivotX

Float or percentage. The X coordinate of the center of rotation. Expressed either in pixels relative to the object's left edge, such as "5"; in percentage relative to the object's left edge, such as "5%"; or in percentage relative to the parent container's left edge, such as "5%p".

android:pivotY

Float or percentage. The Y coordinate of the center of rotation. Expressed either in pixels relative to the object's top edge, such as "5"; in percentage relative to the object's top edge, such as "5%"; or in percentage relative to the parent container's top edge, such as "5%p".

3. scale

It expands the view in both x and y direction.

It starts the position(0,0) of x and y from the top left corner of Image view

to keep the position(0,0) at the center of the view pivot the x and y values to 50%

Attributes:

android:fromXScale

Float. Starting X size offset, where 1.0 is no change.

android:toXScale

Float. Ending X size offset, where 1.0 is no change.

android:fromYScale

Float. Starting Y size offset, where 1.0 is no change.

android:toYScale

Float. Ending Y size offset, where 1.0 is no change.

android:pivotX

Float. The X coordinate to remain fixed when the object is scaled.

android:pivotY

Float. The Y coordinate to remain fixed when the object is scaled.

4. translate

A vertical and/or horizontal motion.

Represents a TranslateAnimation.

Supports the following attributes in any of these three formats:

Values from -100 to 100 ending with "%", indicating a percentage relative to itself.

Values from -100 to 100 ending in "%p", indicating a percentage relative to its parent.

A float value with no suffix, indicating an absolute value.

Attributes:

android:fromXDelta

Float or percentage. Starting X offset.

android:toXDelta

Float or percentage. Ending X offset.

android:fromYDelta

Float or percentage. Starting Y offset.

android:toYDelta

Float or percentage. Ending Y offset.

we can give the values for the attributes in pixels relative to the normal position, such as "5".

In percentage relative to the element width, such as "5%".

Also, in percentage relative to the parent width, such as "5%p".

For animation the onclick attribute of button may not work so in this case provide the id for button and set the onclick listener.

MultiThreading:

- When we execute a code, os cretaes a processs and the code starts the execution inside this process.
- If in the same process we want to execute some tasks parallely then we need to create sub processes under it which are called as threads.
- Threads are also called as light weight process which executes under the main process.
- When the user launches your app, Android creates a new process along with an execution thread.
- This main thread, also known as the UI thread, is responsible for everything that happens onscreen.
- Threads persist throughout the lifetime of the activities that create them.
- Threads continue to execute, uninterrupted, regardless of the creation or destruction of activities, although they will be terminated when the application terminates.

Spalsh Activity:

- A splash screen is mostly the first screen of the app when it is opened.
- It is a constant screen that appears for a specific amount of time and generally shows for the first time when the app is launched.
- Splash screen is used to display some basic introductory information such as the company logo, content, etc just before the app loads completely.
- create a splash activity and make it launcher activity, add the animation on it and then launch the next activity after the animation is finished on splash activity.

Consuming REST API:

1. Add the dependency of retrofit to your application

```
implementation("com.squareup.retrofit2:retrofit:3.
implementation("com.squareup.retrofit2:converter-g
```

2. Create an interface called as API interface that will keep all the required methods to call the existing API's
3. In the interface provide the methods with the proper annotation that will map to the existing backend API's

```
public interface API {
    public static final String BASE_URL = "http://172.18.4.65:4000";

    @POST("/user")
    public Call<JsonObject> registerUser(@Body User user);
}
```

4. Create the Singleton class Retrofit client that will help us to get the object of this API interface.

```
public class RetrofitClient {
private static RetrofitClient retrofitClient;
private API api;

private RetrofitClient(){
      api = new Retrofit.Builder()
            .baseUrl(API.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(API.class);
   }
public static RetrofitClient getInstance(){
if(retrofitClient!=null)
         retrofitClient = new RetrofitClient();
return retrofitClient;
   }
public API getApi() {
return api;
   }
}
```

5. Start calling the api as per the requirement:

```
RetrofitClient.getInstance().getApi().registerUser(user).enqueue(new
Callback<JsonObject>() {

@Override
public void onResponse(Call<JsonObject> call, Response<JsonObject>
response) {
    Toast.makeText(RegisterActivity.this, "Registration is successful",
Toast.LENGTH_SHORT).show();
  }

@Override
public void onFailure(Call<JsonObject> call, Throwable t) {
    Toast.makeText(RegisterActivity.this, "Something went wrong",
Toast.LENGTH_SHORT).show();
  }
});
```

6. To make the call and get the response successfully we need the permission to use the internet and the permission of usesCleartextTraffic. In manifest add both the permissions

```
<uses-permission android:name="android.permission.INTERNET"/>
<application
android:usesCleartextTraffic="true">
</application>
```

Progress Bar

- A user interface element that indicates the progress of an operation.
- Progress bar supports two modes to represent progress.

1. determinate
2. indeterminate

```
<ProgressBar
        android:id="@+id/progressBarInDeterminate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
<!-- Add the style in progress bar as horizontal and it will be determinate -->
<ProgressBar
        style="@style/Widget.AppCompat.ProgressBar.Horizontal"
        android:id="@+id/progressBarDeterminate"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
```

- Determinate has the ability to set the progress to it.
- We can set the max and min progress to it and also update it as per the requirement
- for Indeterminate we cannot set the progress.
- the views should not be manipulated inside threads as they can cause unexpected behaviour.
- to manipulate them we should always use the main UI Thread.
- to get the main UI Thread use the post() method on the views.

```
progressBarDeterminate.post(new Runnable() {
@Override
public void run() {
    progressBarDeterminate.setVisibility(View.INVISIBLE);
  }
});
progressBarInDeterminate.post(new Runnable() {
@Override
public void run() {
        progressBarInDeterminate.setVisibility(View.INVISIBLE);
  }
});
```

Date and Time Picker Dialog:

```
// For Date Picker Dialog
DatePickerDialog datePickerDialog = new DatePickerDialog(this);
datePickerDialog.setOnDateSetListener(new DatePickerDialog.OnDateSetListener() {
@Override
```

```
public void onDateSet(DatePicker view, int year, int month, int dayOfMonth) {
        textViewDate.setText(dayOfMonth + "/" + month + "/" + year);
    }
});
datePickerDialog.show();
// optional to get Time to be displayed inside timepickerdailog
Calendar calendar = Calendar.getInstance();
int hr = calendar.get(Calendar.HOUR);
int min = calendar.get(Calendar.MINUTE);
// For Time Picker Dialog
TimePickerDialog timePickerDialog = new TimePickerDialog(this, new
TimePickerDialog.OnTimeSetListener() {
@Override
public void onTimeSet(TimePicker view, int hourOfDay, int minute) {
        String AM_PM = hourOfDay < 12 ? " AM" : " PM";
        textViewTime.setText(hourOfDay + " : " + minute + AM_PM);
    }
}, hr, min, false);
timePickerDialog.show();
```

In App Notifications:

To send notifications first add the permission in manifest and provide the permission to send the notification

```
<uses-permission android:name="android.permission.POST_NOTIFICATIONS"/>
```

To send the notification from app we require objects of

1. NotificationChannel

   It creates a new channel for notification that needs to be declared to the android

2. NotificationManager

   It is an object that is used to register our channel towards the NOtification Service in android

3. NotificationCompact

   To create the new notification along with the required data to be shown inside it.

4. NotificationManagerCompact

   It is used to display the notification in our status bar.

```
public void sendNotification(View view) {
// cretae a notification channel for your notification
NotificationChannel notificationChannel = new NotificationChannel("channel_id",
"channel_Name", NotificationManager.IMPORTANCE_DEFAULT);
// register your notification channel with the Android Notification service
NotificationManager notificationManager =
getSystemService(NotificationManager.class);
notificationManager.createNotificationChannel(notificationChannel);
// Cretae the notification thet you want to display and pass the channel_id
NotificationCompat.Builder builder = new NotificationCompat.Builder(this,
"channel_id");
builder.setSmallIcon(R.drawable.watch);
builder.setContentTitle("New Notification");
builder.setContentText("This is a new Notification sent.");
//To display larger texts
builder.setStyle(new NotificationCompat.BigTextStyle().bigText("This is a new
Notification sent from your app to check. wether it works or no"));
//before displaying notification it requires the permission to check if post
notification permission is given or no
 if (ActivityCompat.checkSelfPermission(this, android.Manifest.permission
POST_NOTIFICATIONS) != PackageManager.PERMISSION_GRANTED)
return;
// display the notification created immediately in your notifications
 // the int value in notifiy determines wether to create new notification if
already one exists or to manipulate the same
NotificationManagerCompat.from(MainActivity.this).notify(1, builder.build());
}
```

We can set the tap icon on the notification and launch the activity from our app.

For this we require object of PendingIntent

this should be done before we notify.

```
//Handle the click event of notification
Intent intent = new Intent(this, SplashActivity.class);
//only set this flag if you want to start the activity when you are already
working with the app.
 // however the activity launches only when the app is closed when this flag is not
set.
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
PendingIntent pendingIntent =PendingIntent.getActivity(this,0,intent,PendingIntent.FLAG_IMMUTABLE);
builder.setContentIntent(pendingIntent);
builder.setAutoCancel(true);
```

we can also set the progress to notifications.

it will be used when downloading needs to be shown in the notification.

```
new Thread(new Runnable() {
@Override
 public void run() {
int PROGRESS_MAX = 100;
int PROGRESS_CURRENT = 0;
for (int i = 1; i <= 10; i++) {
try {
        Thread.sleep(1000);
        PROGRESS_CURRENT = 10 * i;
        builder.setProgress(PROGRESS_MAX, PROGRESS_CURRENT, false);
        notificationManagerCompat.notify(3, builder.build());
     }
catch (InterruptedException e) {
throw new RuntimeException(e);
     }
  }
  builder.setContentText("Downlaod Finished");
// set the pending intent when download completes
  builder.setContentIntent(pendingIntent);
  notificationManagerCompat.notify(3,builder.build());
}
}).start();
```

Sensors:

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform supports three broad categories of sensors:

Motion sensors

These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

Environmental sensors

These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

Position sensors

These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

You can access sensors available on the device and acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

Determine which sensors are available on a device.

Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.

Acquire raw sensor data and define the minimum rate at which you acquire sensor data.

Register and unregister sensor event listeners that monitor sensor changes.

Sensor Framework

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the `android.hardware` package and includes the following classes and interfaces:

`SensorManager`

You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

`Sensor`

You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

`SensorEvent`

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

`SensorEventListener`

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

Sensor	Type	Description	Common Uses
`TYPE_ACCELEROMETER`	Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
`TYPE_AMBIENT_TEMPERATURE`	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
`TYPE_GRAVITY`	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
`TYPE_GYROSCOPE`	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
`TYPE_LIGHT`	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
`TYPE_LINEAR_ACCELERATION`	Software or Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
`TYPE_MAGNETIC_FIELD`	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	Creating a compass.
`TYPE_ORIENTATION`	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity sensor and the geomagnetic field sensor in conjunction with the `getRotationMatrix()` method.	Determining device position.
`TYPE_PRESSURE`	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
`TYPE_PROXIMITY`	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
`TYPE_RELATIVE_HUMIDITY`	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
`TYPE_ROTATION_VECTOR`	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
`TYPE_TEMPERATURE`	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the `TYPE_AMBIENT_TEMPERATURE` sensor in API Level 14	Monitoring temperatures.

Sensor Coordinate System

In general, the sensor framework uses a standard 3-axis coordinate system to express data values. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see figure 1). When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values. This coordinate system is used by the following sensors:

- Acceleration sensor
- Gravity sensor
- Gyroscope
- Linear acceleration sensor
- Geomagnetic field sensor

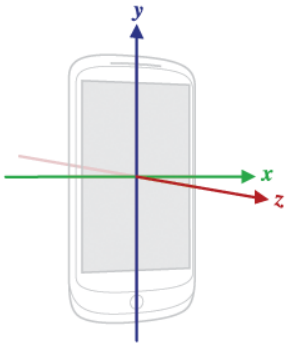


Figure 1. Coordinate system (relative to a device) that's used by the Sensor API.

The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves. This behavior is the same as the behavior of the OpenGL coordinate system.

Another point to understand is that your application must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And the sensor coordinate system is always based on the natural orientation of a device.

Finally, if your application matches sensor data to the on-screen display, you need to use the `getRotation()` method to determine screen rotation, and then use the `remapCoordinateSystem()` method to map sensor coordinates to screen coordinates. You need to do this even if your manifest specifies portrait-only display.

> **Note:** Some sensors and methods use a coordinate system that is relative to the world's frame of reference (as opposed to the device's frame of reference). These sensors and methods return data that represent device motion or device position relative to the earth. For more information, see the **`getOrientation()`** method, the **`getRotationMatrix()`** method, Orientation Sensor, and Rotation Vector Sensor.

Best Practices for Accessing and Using Sensors

As you design your sensor implementation, be sure to follow the guidelines that are discussed in this section. These guidelines are recommended best practices for anyone who is using the sensor framework to access sensors and acquire sensor data.

Only gather sensor data in the foreground

On devices running Android 9 (API level 28) or higher, apps running in the background have the following restrictions:

- Sensors that use the continuous reporting mode, such as accelerometers and gyroscopes, don't receive events.
- Sensors that use the on-change or one-shot reporting modes don't receive events.

Given these restrictions, it's best to detect sensor events either when your app is in the foreground or as part of a foreground service.

Unregister sensor listeners

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. The following code shows how to use the `onPause()` method to unregister a listener:

Kotlin Java

```
private lateinit var sensorManager: SensorManager
...
override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(this)
}
```

Don't block the onSensorChanged() method

Sensor data can change at a high rate, which means the system may call the `onSensorChanged(SensorEvent)` method quite often. As a best practice, you should do as little as possible within the `onSensorChanged(SensorEvent)` method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the `onSensorChanged(SensorEvent)` method.

Avoid using deprecated methods or sensor types

Several methods and constants have been deprecated. In particular, the `TYPE_ORIENTATION` sensor type has been deprecated. To get orientation data you should use the `getOrientation()` method instead. Likewise, the `TYPE_TEMPERATURE` sensor type has been deprecated. You should use the `TYPE_AMBIENT_TEMPERATURE` sensor type instead on devices that are running Android 4.0.

Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

Choose sensor delays carefully

When you register a sensor with the `registerListener()` method, be sure you choose a delivery rate that is suitable for your application or use-case. Sensors can provide data at very high rates. Allowing the system to send extra data that you don't need wastes system resources and uses battery power.

Services:

A Service is an application component that can perform long-running operations in the background. Service does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. A service runs in the main thread of its hosting process; the service does not create its own thread and does not run in a separate process unless you specify otherwise. You should run any blocking operations on a separate thread within the service to avoid Application Not Responding (ANR) errors. There are the three different types of services:

1. Foreground:
   - A foreground service performs some operation that is noticeable to the user.
   - Foreground services must display a Notification.
   - This notification cannot be dismissed unless the service is either stopped or removed from the foreground.
   - Foreground services continue running even when the user isn't interacting with the app.
   - For example, an audio app would use a foreground service to play an audio track.
2. Background:
   - A background service performs an operation that isn't directly noticed by the user.
   - For example, if an app used a service to compact its storage, that would usually be a background service.
3. Bound:
   - A service is bound when an application component binds to it by calling bindService().

- A bound service offers a client-server interface that allows components to interact with the service
- A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.
- Service has its own lifecycle which is managed by the callback methods
- We can create our service in two ways depending on which the callback method
    1. Bounded Service
        onCreate()
        onBind()
        onUnbind()
        onDestroy()
    2. UnBounded / Started Service
        onCreate()
        onStartCommand()
        onDestroy()

BoundedService:

- Right click on your package and select Service -> Service
- Check the entry of Service into your maninfest
- Add all the Lifecycle callback methods of service in MyService class
- Add 2 buttons into MainActivity to start and stop the service and handle the click event
- To start the service in bounded mode you have to create Service Connection object
- To start service use the method onBind to start the service in bounded mode
- To stop Service use the method onUnBind.
- To perform a download task we can perform it only once into the onBind Method
- If we want to perform the task multiple times then we need the Service object that is binded with that component
- For this in MyService add a nested class MyBinder that extends from Binder which is responsible to
- return MyService class object.
- From onBind() return the new object of MyBinder
- This will call the onServiceConnected method from Service Connection in MainActivity
- Inside this method you can fetch the MyService object which you can perform the task kept inside the service.

UnBounded Service:

- In this the service is not bound to the component of android.
- we can stop the service automatically when the task is completed by calling the method stopself() after task is completed.

Services overview

A `Service` is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

> **Caution:** A service runs in the main thread of its hosting process; the service does **not** create its own thread and does **not** run in a separate process unless you specify otherwise. You should run any blocking operations on a separate thread within the service to avoid Application Not Responding (ANR) errors.

Types of Services

These are the three different types of services:

Foreground

A foreground service performs some operation that is noticeable to the user. For example, an audio app would use a foreground service to play an audio track. Foreground services must display a Notification. Foreground services continue running even when the user isn't interacting with the app.

When you use a foreground service, you must display a notification so that users are actively aware that the service is running. This notification cannot be dismissed unless the service is either stopped or removed from the foreground.

Learn more about how to configure foreground services in your app.

> **Note:** The WorkManager API offers a flexible way of scheduling tasks, and is able to run these jobs as foreground services if needed. In many cases, using WorkManager is preferable to using foreground services directly.

Background

A background service performs an operation that isn't directly noticed by the user. For example, if an app used a service to compact its storage, that would usually be a background service.

> **Note:** If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground. In most situations, for example, you shouldn't access location information from the background. Instead, schedule tasks using WorkManager.

Bound

A service is *bound* when an application component binds to it by calling `bindService()`. A bound service offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

Although this documentation generally discusses started and bound services separately, your service can work both ways—it can be started (to run indefinitely) and also allow binding. It's simply a matter of whether you implement a couple of callback methods: `onStartCommand()` to allow components to start it and `onBind()` to allow binding.

Regardless of whether your service is started, bound, or both, any application component can use the service (even from a separate application) in the same way that any component can use an activity—by starting it with an `Intent`. However, you can declare the service as *private* in the manifest file and block access from other applications. This is discussed more in the section about Declaring the service in the manifest.

The basics

To create a service, you must create a subclass of `Service` or use one of its existing subclasses. In your implementation, you must override some callback methods that handle key aspects of the service lifecycle and provide a mechanism that allows the components to bind to the service, if appropriate. These are the most importa callback methods that you should override:

`onStartCommand()`

> The system invokes this method by calling `startService()` when another component (such as an activity) requests that the service be started. When this method executes, the service is started and can run in the background indefinitely. If you implement this, it is your responsibility to stop the service when its work is complete by calling `stopSelf()` or `stopService()`. If you only want to provide binding, you don't need to implement this method.

`onBind()`

> The system invokes this method by calling `bindService()` when another component wants to bind with the service (such as to perform RPC). In your implementation of this method, you must provide an interface that clients use to communicate with the service by returning an `IBinder`. You must always implement this metho however, if you don't want to allow binding, you should return null.

`onCreate()`

> The system invokes this method to perform one-time setup procedures when the service is initially created (before it calls either `onStartCommand()` or `onBind()`). If the service is already running, this method is not called.

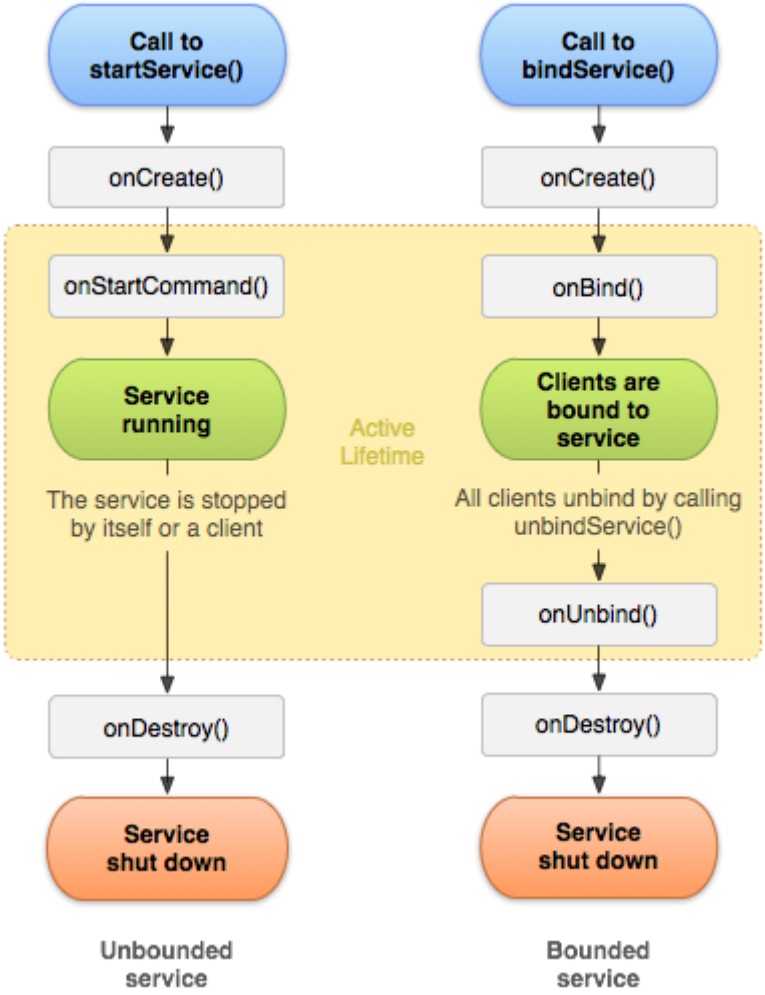
`onDestroy()`

> The system invokes this method when the service is no longer used and is being destroyed. Your service shoul implement this to clean up any resources such as threads, registered listeners, or receivers. This is the last call that the service receives.

If a component starts the service by calling `startService()` (which results in a call to `onStartCommand()`), the service continues to run until it stops itself with `stopSelf()` or another component stops it by calling `stopService()`.

If a component calls `bindService()` to create the service and `onStartCommand()` is *not* called, the service runs only as long as the component is bound to it. After the service is unbound from all of its clients, the system destroys

The Android system stops a service only when memory is low and it must recover system resources for the activity that has user focus. If the service is bound to an activity that has user focus, it's less likely to be killed; if the service declared to run in the foreground, it's rarely killed. If the service is started and is long-running, the system lowers its position in the list of background tasks over time, and the service becomes highly susceptible to killing—if your service is started, you must design it to gracefully handle restarts by the system. If the system kills your service, it restarts it as soon as resources become available, but this also depends on the value that you return from `onStartCommand()`. For more information about when the system might destroy a service, see the Processes and Threading document.



AIDL (Android Interface definition Language):

It lets you define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC).

1. inside the gradle.properties file and add the below line

```
android.defaults.buildfeatures.aidl=true
```

Inside the AIDL Provider add the aidl folder and an aidl interface

package aidl;

interface IRemoteService{

void downlaodFile();

}

Implement this interface inside your service class and return its stub/binder object from onBind method.

private IRemoteService.Stub stubBinder = new IRemoteService.Stub() {

@Override

public void downlaodFile() throws RemoteException {

new Thread(new Runnable() {

@Override

```
public void run() {
for (int i = 0; i < 10; i++) {
try {
                    Thread.sleep(1000);
                    Log.e(tag,"Counter - "+i);
                }
                }
              }
            }
  }).start();
  }
};
@Override
 public IBinder onBind(Intent intent) {
// return the object of your aidl interface that is implemeted above
// it will be received inside main activity in service connection
   Log.e(tag, "onBind()");
return stubBinder;
}
```

Inside MainActivity in Service connection object we can get the object of the aidl interface inside it

using the below method.

```
IRemoteService remoteService;
remoteService = IRemoteService.Stub.asInterface(service);
```

In second app that is AIDL Consumer we need to use the Provider service. In this inside manifest add the queries tag and add the intent.

```
<queries>
        <intent>
            <action android:name="myservice.AIDL"/>
        </intent>
 </queries>
```

Copy paste the aidl interface from the AIDL provider and add it inside Consumer app.

Inside the service connection get the obejct. Implement the startService onclick where we need to create the explicit intent from our implicit intent and start the service.

```
Intent intent = new Intent("myservice.AIDL");
//convert implicit intent to explicit intent
PackageManager packageManager = getPackageManager();
```

ResolveInfo resolveInfo = packageManager.resolveService(intent,0);

ComponentName componentName = new

ComponentName(resolveInfo.serviceInfo.packageName,resolveInfo.serviceInfo.name);

intent.setComponent(componentName);

bindService(intent, serviceConnection, BIND_AUTO_CREATE);

The Android Interface Definition Language (AIDL) is similar to other IDLs: it lets you define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC).

On Android, one process can't normally access the memory of another process. To talk, they need to decompose their objects into primitives that the operating system can understand and marshall the objects across that boundary for you. The code to do that marshalling is tedious to write, so Android handles it for you with AIDL.

> ★ **Note:** AIDL is necessary only if you let clients from different applications access your service for IPC and you want to handle multithreading in your service. If you don't need to perform concurrent IPC across different applications, create your interface by implementing a `Binder`. If you want to perform IPC but *don't* need to handle multithreading, implement your interface using a `Messenger`. Regardless, be sure that you understand bound services before implementing an AIDL.

Before you begin designing your AIDL interface, be aware that calls to an AIDL interface are direct function calls. Don't make assumptions about the thread in which the call occurs. What happens is different depending on whether the call is from a thread in the local process or a remote process:

- Calls made from the local process execute in the same thread that is making the call. If this is your main UI thread, that thread continues to execute in the AIDL interface. If it is another thread, that is the one that executes your code in the service. Thus, if only local threads are accessing the service, you can completely control which threads are executing in it. But if that is the case, don't use AIDL at all; instead, create the interface by implementing a `Binder`.
- Calls from a remote process are dispatched from a thread pool the platform maintains inside your own process. Be prepared for incoming calls from unknown threads, with multiple calls happening at the same time. In other words, an implementation of an AIDL interface must be completely thread-safe. Calls made from one thread on the same remote object arrive *in order* on the receiver end.
- The `oneway` keyword modifies the behavior of remote calls. When it is used, a remote call does not block. It sends the transaction data and immediately returns. The implementation of the interface eventually receives this as a regular call from the `Binder` thread pool as a normal remote call. If `oneway` is used with a local call, there is no impact, and the call is still synchronous.

Defining an AIDL interface

Define your AIDL interface in an `.aidl` file using the Java programming language syntax, then save it in the source code, in the `src/` directory, of both the application hosting the service and any other application that binds to the service.

When you build each application that contains the `.aidl` file, the Android SDK tools generate an `IBinder` interface based on the `.aidl` file and save it in the project's `gen/` directory. The service must implement the `IBinder` interface as appropriate. The client applications can then bind to the service and call methods from the `IBinder` to perform IPC.

To create a bounded service using AIDL, follow these steps, which are described in the sections that follow:

1. Create the `.aidl` file

   This file defines the programming interface with method signatures.

2. Implement the interface

   The Android SDK tools generate an interface in the Java programming language based on your `.aidl` file. This interface has an inner abstract class named `Stub` that extends `Binder` and implements methods from your AIDL interface. You must extend the `Stub` class and implement the methods.

3. Expose the interface to clients

   Implement a `Service` and override `onBind()` to return your implementation of the `Stub` class.

Caution: Any changes that you make to your AIDL interface after your first release must remain backward compatible to avoid breaking other applications that use your service. That is, because your `.aidl` file must be copied to other applications so they can access your service's interface, you must maintain support for the original interface.

Create the .aidl file

AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters and return values. The parameters and return values can be of any type, even other AIDL-generated interfaces.

You must construct the `.aidl` file using the Java programming language. Each `.aidl` file must define a single interface and requires only the interface declaration and method signatures.

By default, AIDL supports the following data types:

- All primitive types, with the exception of `short`, in the Java programming language (such as `int`, `long`, `char`, `boolean`, and so on)
- Arrays of any types, such as `int[]` or `MyParcelable[]`
- `String`
- `CharSequence`
- `List`

  All elements in the `List` must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you declare. A `List` can optionally be used as a parameterized type class, such as `List<String>`. The actual concrete class that the other side receives is always an `ArrayList`, although the method is generated to use the `List` interface.
- `Map`

  All elements in the `Map` must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you declare. Parameterized type maps, such as those of the form `Map<String,Integer>`, aren't supported. The actual concrete class that the other side receives is always a `HashMap`, although the method is generated to use the `Map` interface. Consider using a `Bundle` as an alternative to `Map`.

You must include an `import` statement for each additional type not listed previously, even if they are defined in the same package as your interface.

When defining your service interface, be aware that:

- Methods can take zero or more parameters and can return a value or void.
- All non-primitive parameters require a directional tag indicating which way the data goes: `in`, `out`, or `inout` (see the example below).

  Primitives, `String`, `IBinder`, and AIDL-generated interfaces are `in` by default and can't be otherwise.

  > **Caution:** Limit the direction to what is truly needed, because marshalling parameters is expensive.
- All code comments included in the `.aidl` file are included in the generated `IBinder` interface except comments before the import and package statements.
- String and int constants can be defined in the AIDL interface, such as `const int VERSION = 1;`.
- Method calls are dispatched by a `transact()` code, which normally is based on a method index in the interface. Because this makes versioning difficult, you can manually assign the transaction code to a method: `void method() = 10;`.
- Nullable arguments and return types must be annotated using `@nullable`.

BroadCastReceiver:

It is the fundamental component of android Android apps send and receive broadcast messages from the Android system and other Android apps The system and apps typically send broadcasts when certain events occur. For example, the Android system sends broadcasts when various system events occur, such as system boot, device charging,Screen On Off,etc. Apps also send custom broadcasts, for example, to notify other apps of something that might interest them (for example, new data download). Apps can register to receive specific broadcasts. When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast. Broadcasts can be used as a messaging system across apps and outside of the normal user flow. However, you must be careful not to abuse the opportunity to respond to broadcasts and run jobs in the background that can contribute to a slow system performance. The system automatically sends broadcasts when various system events occur, such as when the system switches in and out of Airplane Mode. All subscribed apps receive these broadcasts. The Intent object wraps the broadcast message. The action string identifies the event that occurred, such as android.intent.action.AIRPLANE_MODE.

The intent might also include additional information bundled into its extra field. For example, the Airplane Mode intent includes a boolean extra that indicates whether or not Airplane Mode is on. To get the state of Screen lock and unlock and to get the status of the bluetooth on and off.

For this add the permission of bluetooth in the maifest file:

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

The intent filter to be dynamically added in main activity.

```
MyReceiver receiver;
    @Override
     protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      receiver = new MyReceiver();
    IntentFilter intentFilter = new IntentFilter();
      intentFilter.addAction(Intent.ACTION_SCREEN_ON);
      intentFilter.addAction(Intent.ACTION_SCREEN_OFF);
      intentFilter.addAction(BluetoothAdapter.ACTION_STATE_CHANGED);
      registerReceiver(receiver,intentFilter);
    }
```

Android Permissions:

to ask for multiple permissions at same time use the below code:

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_PHONE_STATE)
==
  PackageManager.PERMISSION_GRANTED && ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_CALL_LOG)== PackageManager.PERMISSION_GRANTED){
//do the task
}
else{
requestPermissions(new String[]{Manifest.permission.READ_PHONE_STATE,
Manifest.permission.READ_CALL_LOG}, 1);
}
@Override
 public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
super.onRequestPermissionsResult(requestCode, permissions, grantResults);
if (requestCode == 1)
```

```
if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_DENIED)
    Toast.makeText(this, "both permissions are required",
Toast.LENGTH_SHORT).show();
else
//do the task
}
```

Location Based Services:

Use the location services to display the current location It will be displayed as latitude and longitude in the logcat display the latitude and longitude in textviews and add a button that opens the map Permissions required are:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

To get the location, internet shoud be on, location should be on. use the location manager to get the location service, add a location listner and register this to listen to location changes.

```
LocationManager locationManager;
        locationManager = getSystemService(LocationManager.class);
        LocationListener locationListener = new LocationListener() {
        @Override
        public void onLocationChanged(@NonNull Location location) {
            latitude = location.getLatitude();
                longitude = location.getLongitude();
                    Log.e("MainActivity", latitude + "," + longitude);
                    textLatitude.setText("Latitude - "+latitude);
                    textLongitude.setText("Lobgitude - "+longitude);
                 if (ActivityCompat.checkSelfPermission(this,
                android.Manifest.permission.ACCESS_FINE_LOCATION) !=
                PackageManager.PERMISSION_GRANTED &&
                ActivityCompat.checkSelfPermission(this,
                android.Manifest.permission.ACCESS_COARSE_LOCATION) !=
                PackageManager.PERMISSION_GRANTED) {
                   requestPermissions(new String[]{Manifest.permission.ACCESS_FINE_LOCATION,
                        Manifest.permission.ACCESS_COARSE_LOCATION}, 1);
                return;
                }
```

```
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER,
0, 0, locationListener);
```

Add a button to open the maps application. create an intent, add the uri and start the activity

```
Uri uri = Uri.parse("https://www.google.com/maps/place/"+latitude+","+longitude);

Intent intent = new Intent(Intent.ACTION_VIEW,uri);

startActivity(intent);
```

Notifications overview

A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information from your app. Users can tap the notification to open your app or take an action directly from the notification.

This page provides an overview of where notifications appear and the available features. To start building notifications, read Create a notification.

For more information about notification design and interaction patterns, see the Notifications design guide.

Appearances on a device

Notifications automatically appear to users in different locations and formats. A notification appears as an icon in the status bar, a more detailed entry in the notification drawer, and a badge on the app's icon. Notifications also appear on paired wearables.

Status bar and notification drawer

When you issue a notification, it first appears as an icon in the status bar.

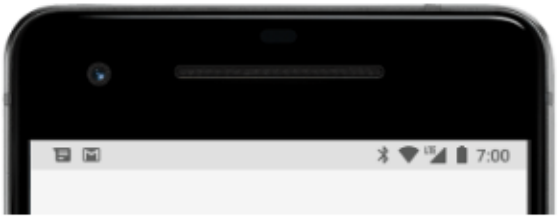
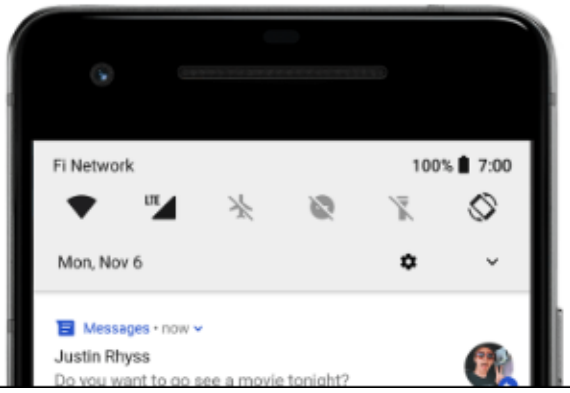


Figure 1. Notification icons appear on the left side of the status bar.

Users can swipe down on the status bar to open the notification drawer, where they can view more details and take actions with the notification.



Lock screen

Beginning with Android 5.0, notifications can appear on the lock screen.

You can programmatically set whether notifications posted by your app show on a secure lock screen and, if so, the level of detail visible.

Users can use the system settings to choose the level of detail visible in lock screen notifications or to disable all lock screen notifications. Starting with Android 8.0, users can disable or enable lock screen notifications for each notification channel.

Figure 4. Notifications on the lock screen with sensitive content hidden.

To learn more, see Set lock screen visibility.

App icon badge

In supported launchers on devices running Android 8.0 (API level 26) and higher, app icons indicate new notifications with a colored *badge* known as a *notification dot* on the corresponding app launcher icon.

Users can touch & hold an app icon to see the notifications for that app. Users can dismiss or act on notifications from that menu, similar to the notification drawer.

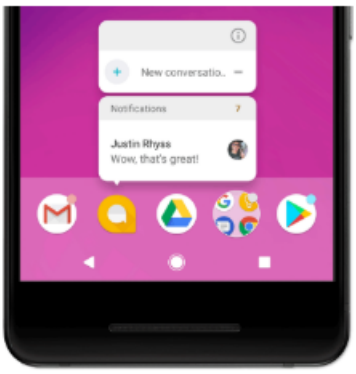


Figure 5. Notification badges and the touch & hold menu.

Heads-up notification

Beginning with Android 5.0, notifications can briefly appear in a floating window called a *heads-up notification*. This behavior is normally for important notifications that the user needs to know about immediately, and it only appears if the device is unlocked.

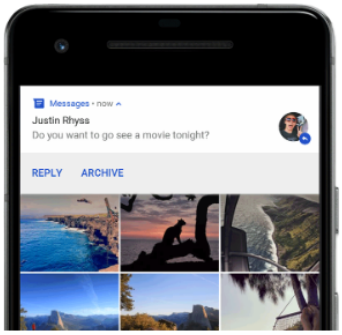


Figure 3. A heads-up notification appears in front of the foreground app.

The heads-up notification appears when your app issues the notification. It disappears after a moment, but it remains visible in the notification drawer as usual.

Conditions that might trigger heads-up notifications include the following:

- The user's activity is in fullscreen mode, such as when the app uses `fullScreenIntent`.
- The notification has high priority and uses ringtones or vibrations on devices running Android 7.1 (API level 25) and lower.
- The notification channel has high importance on devices running Android 8.0 (API level 26) and higher.

App capabilities depend on user choice in permissions dialog

In this dialog, users have the following actions available to them:

- Select **allow**
- Select **don't allow**
- Swipe away from the dialog, without pressing either button

The following sections describe how your app behaves, based on which action the user takes.

User selects "Allow"

If the user selects the **allow** option, your app can do the following:

- Send notifications. All notification channels are allowed.
- Post notifications related to foreground services. These notifications appear in the notification drawer.

User selects "Don't allow"

If the user selects the **don't allow** option, your app can't send notifications unless it qualifies for an exemption. All notification channels are blocked, except for a few specific roles. This is similar to the behavior that occurs when the user manually turns off all notifications for your app in system settings.

Caution: If your app targets 12L or lower and the user taps **Don't allow**, even just once, they aren't prompted again until one of the following occurs:

- The user uninstalls and reinstalls your app.
- You update your app to target Android 13 or higher.

Fragments

A `Fragment` represents a reusable portion of your app's UI. A fragment defines and manages its own layout, has its own lifecycle, and can handle its own input events. Fragments can't live on their own. They must be *hosted* by an activity or another fragment. The fragment's view hierarchy becomes part of, or *attaches to*, the host's view hierarchy.

> **Note:** Some Android Jetpack libraries, such as Navigation, `BottomNavigationView`, and `ViewPager2`, are designed to work with fragments.

Modularity

Fragments introduce modularity and reusability into your activity's UI by letting you divide the UI into discrete chunks. Activities are an ideal place to put global elements around your app's user interface, such as a navigation drawer. Conversely, fragments are better suited to define and manage the UI of a single screen or portion of a screen.

Consider an app that responds to various screen sizes. On larger screens, you might want the app to display a static navigation drawer and a list in a grid layout. On smaller screens, you might want the app to display a bottom navigation bar and a list in a linear layout.

Managing these variations in the activity is unwieldy. Separating the navigation elements from the content can make this process more manageable. The activity is then responsible for displaying the correct navigation UI, while the fragment displays the list with the proper layout.

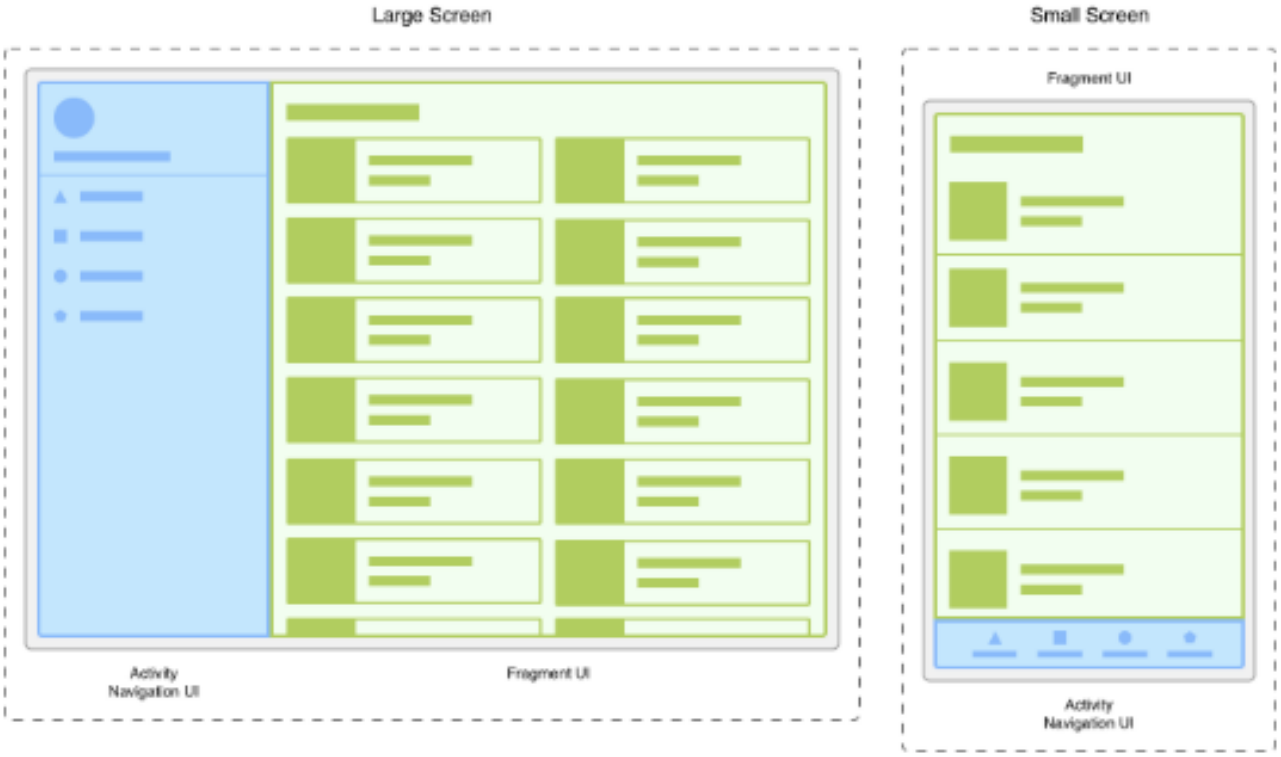


Figure 1. *Two versions of the same screen on different screen sizes. On the left, a large screen contains a navigation drawer that is controlled by the activity and a grid list that is controlled by the fragment. On the right, a small screen contains a bottom navigation bar that is controlled by the activity and a linear list that is controlled by the fragment.*

Dividing your UI into fragments makes it easier to modify your activity's appearance at runtime. While your activity is in the `STARTED` lifecycle state or higher, fragments can be added, replaced, or removed. And you can keep a record of these changes in a back stack that is managed by the activity, so that the changes can be reversed.

You can use multiple instances of the same fragment class within the same activity, in multiple activities, or even as a child of another fragment. With this in mind, only provide a fragment with the logic necessary to manage its own UI. Avoid depending on or manipulating one fragment from another.

Fragment manager

> **Note:** We strongly recommend using the Navigation library to manage your app's navigation. The framework follows best practices for working with fragments, the back stack, and the fragment manager. For more information about Navigation, see Get started with the Navigation component and Migrate to the Navigation component.

`FragmentManager` 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 might never interact with `FragmentManager` directly if you're using the Jetpack Navigation library, as it works with the `FragmentManager` on your behalf. However, any app using fragments is using `FragmentManager` at some level, so it's important to understand what it is and how it works.

This page covers:

- How to access the `FragmentManager`.
- The role of `FragmentManager` in relation to your activities and fragments.
- How to manage the back stack with `FragmentManager`.
- How to provide data and dependencies to your fragments.

Access the FragmentManager

You can access the `FragmentManager` from an activity or from a fragment.

`FragmentActivity` and its subclasses, such as `AppCompatActivity`, have access to the `FragmentManager` through the `getSupportFragmentManager()` method.

Fragments can host one or more child fragments. Inside a fragment, you can get a reference to the `FragmentManager` that manages the fragment's children through `getChildFragmentManager()`. If you need to access its host `FragmentManager`, you can use `getParentFragmentManager()`.

Here are a couple of examples to see the relationships between fragments, their hosts, and the `FragmentManager` instances associated with each.

The host fragment in Example 1 hosts two child fragments that make up a split-view screen. The host fragment in Example 2 hosts a single child fragment that makes up the display fragment of a swipe view.

Given this setup, you can think about each host as having a `FragmentManager` associated with it that manages its child fragments. This is illustrated in figure 2 along with property mappings between `supportFragmentManager`, `parentFragmentManager`, and `childFragmentManager`.

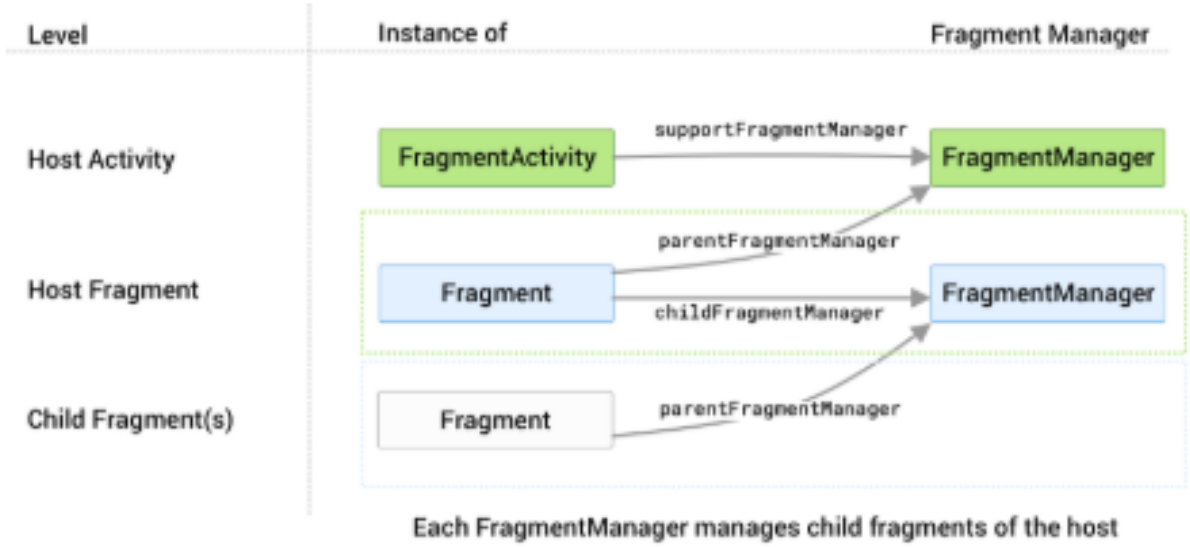


Figure 2. *Each host has its own* `FragmentManager` *associated with it that manages its child fragments.*

The appropriate `FragmentManager` property to reference depends on where the callsite is in the fragment hierarchy along with which fragment manager you are trying to access.

Once you have a reference to the `FragmentManager`, you can use it to manipulate the fragments being displayed to the user.

Child fragments

Generally speaking, your app consists of a single or small number of activities in your application project, with each activity representing a group of related screens. The activity might provide a point to place top-level navigation and a place to scope `ViewModel` objects and other view-state between fragments. A fragment represents an individual destination in your app.

If you want to show multiple fragments at once, such as in a split-view or a dashboard, you can use child fragments that are managed by your destination fragment and its child fragment manager.

Other use cases for child fragments are the following:

- Screen slides, using a `ViewPager2` in a parent fragment to manage a series of child fragment views.
- Sub-navigation within a set of related screens.
- Jetpack Navigation uses child fragments as individual destinations. An activity hosts a single parent `NavHostFragment` and fills its space with different child destination fragments as users navigate through your app.

Use the FragmentManager

The `FragmentManager` manages the fragment back stack. At runtime, the `FragmentManager` can perform back stack operations like adding or removing fragments in response to user interactions. Each set of changes is committed together as a single unit called a `FragmentTransaction`. For a more in-depth discussion about fragment transactions, see the fragment transactions guide.

When the user taps the Back button on their device, or when you call `FragmentManager.popBackStack()`, the top-most fragment transaction pops off of the stack. If there are no more fragment transactions on the stack, and if you aren't using child fragments, the Back event bubbles up to the activity. If you *are* using child fragments, see special considerations for child and sibling fragments.

When you call `addToBackStack()` on a transaction, the transaction can include any number of operations, such as adding multiple fragments or replacing fragments in multiple containers.

When the back stack is popped, all these operations reverse as a single atomic action. However, if you committed additional transactions prior to the `popBackStack()` call, and if you *didn't* use `addToBackStack()` for the transaction, these operations *don't* reverse. Therefore, within a single `FragmentTransaction`, avoid interleaving transactions that affect the back stack with those that don't.

Perform a transaction

To display a fragment within a layout container, use the `FragmentManager` to create a `FragmentTransaction`. Within the transaction, you can then perform an `add()` or `replace()` operation on the container.

For example, a simple `FragmentTransaction` might look like this:

Kotlin Java

```
supportFragmentManager.commit {
    replace<ExampleFragment>(R.id.fragment_container)
    setReorderingAllowed(true)
    addToBackStack("name") // Name can be null
}
```

In this example, `ExampleFragment` replaces the fragment, if any, that is currently in the layout container identified by the `R.id.fragment_container` ID. Providing the fragment's class to the `replace()` method lets the `FragmentManager` handle instantiation using its `FragmentFactory`. For more information, see the Provide dependencies to your fragments section.

`setReorderingAllowed(true)` optimizes the state changes of the fragments involved in the transaction so that animations and transitions work correctly. For more information on navigating with animations and transitions, see Fragment transactions and Navigate between fragments using animations.

Calling `addToBackStack()` commits the transaction to the back stack. The user can later reverse the transaction and bring back the previous fragment by tapping the Back button. If you added or removed multiple fragments within a single transaction, all those operations are undone when the back stack is popped. The optional name provided in the `addToBackStack()` call gives you the ability to pop back to a specific transaction using `popBackStack()`.

If you don't call `addToBackStack()` when you perform a transaction that removes a fragment, then the removed fragment is destroyed when the transaction is committed, and the user cannot navigate back to it. If you do call `addToBackStack()` when removing a fragment, then the fragment is only `STOPPED` and is later `RESUMED` when the user navigates back. Its view *is* destroyed in this case. For more information, see Fragment lifecycle.

Fragment lifecycle

Each `Fragment` instance has its own lifecycle. When a user navigates and interacts with your app, your fragments transition through various states in their lifecycle as they are added, removed, and enter or exit the screen.

To manage lifecycle, `Fragment` implements `LifecycleOwner`, exposing a `Lifecycle` object that you can access through the `getLifecycle()` method.

Each possible `Lifecycle` state is represented in the `Lifecycle.State` enum.

- `INITIALIZED`
- `CREATED`
- `STARTED`
- `RESUMED`
- `DESTROYED`

By building `Fragment` on top of `Lifecycle`, you can use the techniques and classes available for Handling Lifecycles with Lifecycle-Aware Components. For example, you might display the device's location on the screen using a lifecycle-aware component. This component could automatically start listening when the fragment becomes active and stop when the fragment moves to an inactive state.

As an alternative to using a `LifecycleObserver`, the `Fragment` class includes callback methods that correspond to each of the changes in a fragment's lifecycle. These include `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()`.

A fragment's view has a separate `Lifecycle` that is managed independently from that of the fragment's `Lifecycle`. Fragments maintain a `LifecycleOwner` for their view, which can be accessed using `getViewLifecycleOwner()` or `getViewLifecycleOwnerLiveData()`. Having access to the view's `Lifecycle` is useful for situations where a Lifecycle-aware component should only perform work while a fragment's view exists, such as observing `LiveData` that is only meant to be displayed on the screen.

This topic discusses the `Fragment` lifecycle in detail, explaining some of the rules that determine a fragment's lifecycle state and showing the relationship between the `Lifecycle` states and the fragment lifecycle callbacks.

Fragments and the fragment manager

When a fragment is instantiated, it begins in the `INITIALIZED` state. For a fragment to transition through the rest of its lifecycle, it must be added to a `FragmentManager`. The `FragmentManager` is responsible for determining what state its fragment should be in and then moving them into that state.

Beyond the fragment lifecycle, `FragmentManager` is also responsible for attaching fragments to their host activity and detaching them when the fragment is no longer in use. The `Fragment` class has two callback methods, `onAttach()` and `onDetach()`, that you can override to perform work when either of these events occur.

The `onAttach()` callback is invoked when the fragment has been added to a `FragmentManager` and is attached to its host activity. At this point, the fragment is active, and the `FragmentManager` is managing its lifecycle state. At this point, `FragmentManager` methods such as `findFragmentById()` return this fragment.

`onAttach()` is always called before any Lifecycle state changes.

The `onDetach()` callback is invoked when the fragment has been removed from a `FragmentManager` and is detached from its host activity. The fragment is no longer active and can no longer be retrieved using `findFragmentById()`.

`onDetach()` is always called after any Lifecycle state changes.

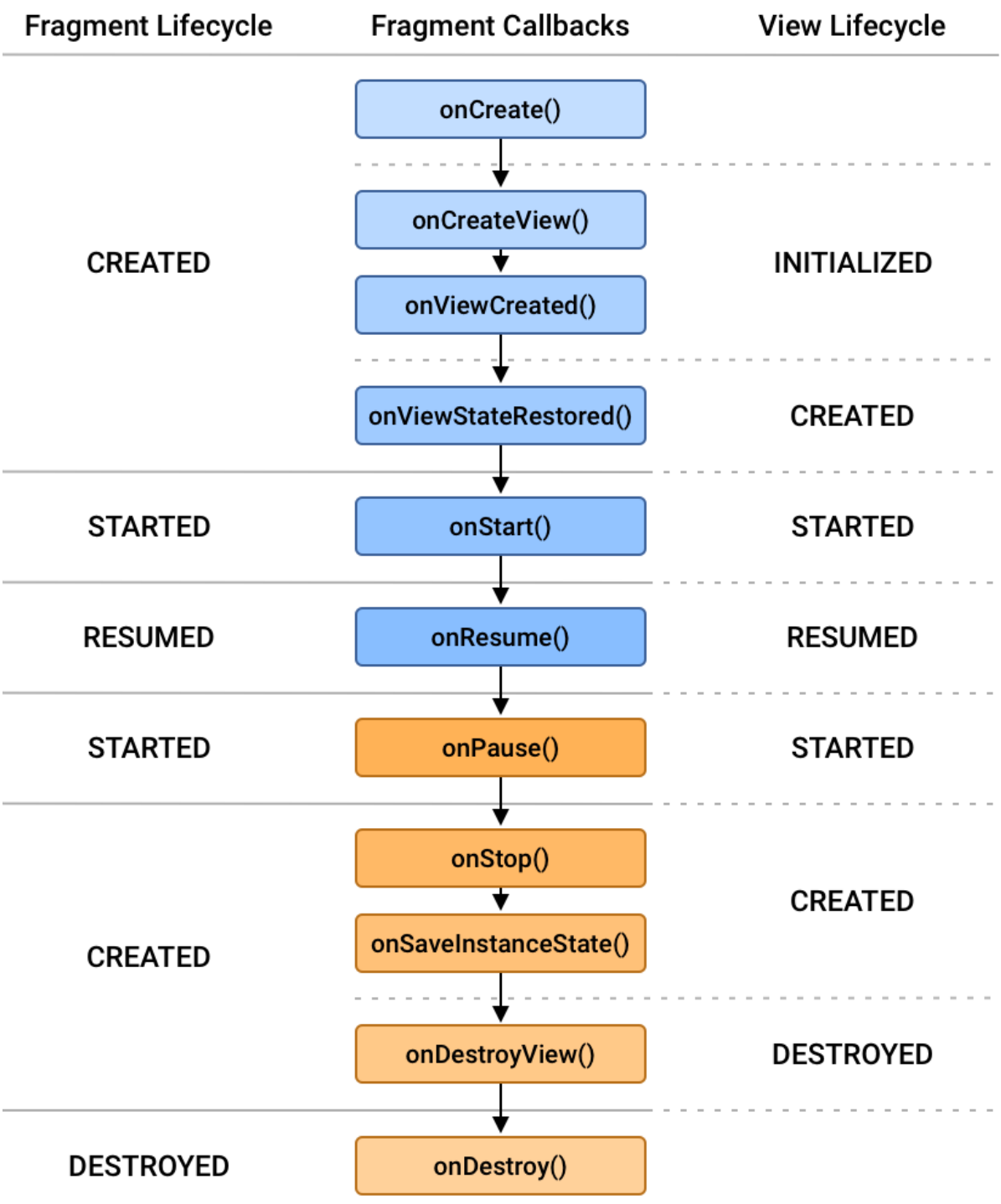
Note that these callbacks are unrelated to the `FragmentTransaction` methods `attach()` and `detach()`. For more information on these methods, see Fragment transactions.

Fragment lifecycle states and callbacks

When determining a fragment's lifecycle state, `FragmentManager` considers the following:

- A fragment's maximum state is determined by its `FragmentManager`. A fragment cannot progress beyond the state of its `FragmentManager`.
- As part of a `FragmentTransaction`, you can set a maximum lifecycle state on a fragment using `setMaxLifecycle()`.
- A fragment's lifecycle state can never be greater than its parent. For example, a parent fragment or activity must be started before its child fragments. Likewise, child fragments must be stopped before their parent fragment or activity.

Caution: Avoid using the `<fragment>` tag to add a fragment using XML, as the `<fragment>` tag allows a fragment to move beyond the state of its `FragmentManager`. Instead, always use `FragmentContainerView` for adding a fragment using XML.



Fragment CREATED

both the fragment's lifecycle callbacks and the fragment's vi
`Lifecycle`.

When your fragment reaches the `CREATED` state, it has been added to a `FragmentManager` and the `onAttach()` method has already been called.

This would be the appropriate place to restore any saved state associated with the fragment itself through the fragment's `SavedStateRegistry`. Note that the fragment's view has *not* been created at this time, and any state associated with the fragment's view should be restored only after the view has been created.

This transition invokes the `onCreate()` callback. The callback also receives a `savedInstanceState` `Bundle` argument containing any state previously saved by `onSaveInstanceState()`. Note that `savedInstanceState` has `null` value the first time the fragment is created, but it is always non-null for subsequent recreations, even if you d not override `onSaveInstanceState()`. See Saving state with fragments for more details.

Fragment CREATED and View INITIALIZED

The fragment's view `Lifecycle` is created only when your `Fragment` provides a valid `View` instance. In most cases, you can use the fragment constructors that take a `@LayoutId`, which automatically inflates the view at the appropriate time. You can also override `onCreateView()` to programmatically inflate or create your fragment's view

If and only if your fragment's view is instantiated with a non-null `View`, that `View` is set on the fragment and can b retrieved using `getView()`. The `getViewLifecycleOwnerLiveData()` is then updated with the newly `INITIALIZED` `LifecycleOwner` corresponding with the fragment's view. The `onViewCreated()` lifecycle callback is also called at this time.

This is the appropriate place to set up the initial state of your view, to start observing `LiveData` instances whose callbacks update the fragment's view, and to set up adapters on any `RecyclerView` or `ViewPager2` instances in your fragment's view.

Fragment and View CREATED

After the fragment's view has been created, the previous view state, if any, is restored, and the view's `Lifecycle` is then moved into the `CREATED` state. The view lifecycle owner also emits the `ON_CREATE` event to its observers. Her you should restore any additional state associated with the fragment's view.

This transition also invokes the `onViewStateRestored()` callback.

Fragment and View STARTED

It is strongly recommended to tie Lifecycle-aware components to the `STARTED` state of a fragment, as this state guarantees that the fragment's view is available, if one was created, and that it is safe to perform a `FragmentTransaction` on the child `FragmentManager` of the fragment. If the fragment's view is non-null, the fragment's view `Lifecycle` is moved to `STARTED` immediately after the fragment's `Lifecycle` is moved to `STARTED`.

When the fragment becomes `STARTED`, the `onStart()` callback is invoked.

★ **Note:** Components such as `ViewPager2` set the maximum `Lifecycle` of offscreen fragments to `STARTED`.

Fragment and View RESUMED

When the fragment is visible, all `Animator` and `Transition` effects have finished, and the fragment is ready for user interaction. The fragment's `Lifecycle` moves to the `RESUMED` state, and the `onResume()` callback is invoked.

The transition to `RESUMED` is the appropriate signal to indicate that the user is now able to interact with your fragment. Fragments that are not `RESUMED` should not manually set focus on their views or attempt to handle input method visibility.

Downward state transitions

When a fragment moves downward to a lower lifecycle state, the relevant `Lifecycle.Event` is emitted to observers by the fragment's view `Lifecycle`, if instantiated, followed by the fragment's `Lifecycle`. After a fragment's lifecycle event is emitted, the fragment calls the associated lifecycle callback.

Fragment and View STARTED

As the user begins to leave the fragment, and while the fragment is still visible, the `Lifecycle`s for the fragment and for its view are moved back to the `STARTED` state and emit the `ON_PAUSE` event to their observers. The fragment then invokes its `onPause()` callback.

Fragment and View CREATED

Once the fragment is no longer visible, the `Lifecycle`s for the fragment and for its view are moved into the `CREATED` state and emit the `ON_STOP` event to their observers. This state transition is triggered not only by the parent activity or fragment being stopped, but also by the saving of state by the parent activity or fragment. This behavior guarantees that the `ON_STOP` event is invoked before the fragment's state is saved. This makes the `ON_STOP` event the last point where it is safe to perform a `FragmentTransaction` on the child `FragmentManager`.

As shown in figure 2, the ordering of the `onStop()` callback and the saving of the state with `onSaveInstanceState()` differs based on API level. For all API levels prior to API 28, `onSaveInstanceState()` is invoked before `onStop()`. For API levels 28 and higher, the calling order is reversed.

Figure 2. Calling order differences for `onStop()` *and* `onSaveInstanceState()`.

Fragment CREATED and View DESTROYED

After all of the exit animations and transitions have completed, and the fragment's view has been detached from the window, the fragment's view `Lifecycle` is moved into the `DESTROYED` state and emits the `ON_DESTROY` event to its observers. The fragment then invokes its `onDestroyView()` callback. At this point, the fragment's view has reached the end of its lifecycle and `getViewLifecycleOwnerLiveData()` returns a `null` value.

At this point, all references to the fragment's view should be removed, allowing the fragment's view to be garbage collected.

Fragment DESTROYED

If the fragment is removed, or if the `FragmentManager` is destroyed, the fragment's `Lifecycle` is moved into the `DESTROYED` state and sends the `ON_DESTROY` event to its observers. The fragment then invokes its `onDestroy()` callback. At this point, the fragment has reached the end of its lifecycle.

Introduction

RecyclerView and ListView are two popular options for displaying long lists of data within an Android application. Both are subclasses of the ViewGroup class and can be used to display scrollable lists. However, they have different features, capabilities and implementation.

The process of implementing both may seem pretty similar, for example,

- You get list of data
- You create an adapter
- Find the view to which you have to display the list
- Set the adapter to that list

ListView was one of the earliest components introduced in Android development for displaying a scrollable list of items. Although it provided basic functionality and ease of implementation, it had its limitations, especially when it came to handling large data sets and customizing the appearance and behavior of the list.

As Android applications evolved and the need for more sophisticated list management became apparent, RecyclerView was introduced as a more versatile and efficient solution for displaying lists. As a developer, it's essential to understand the key differences between ListView and RecyclerView to appreciate their respective advantages and disadvantages.

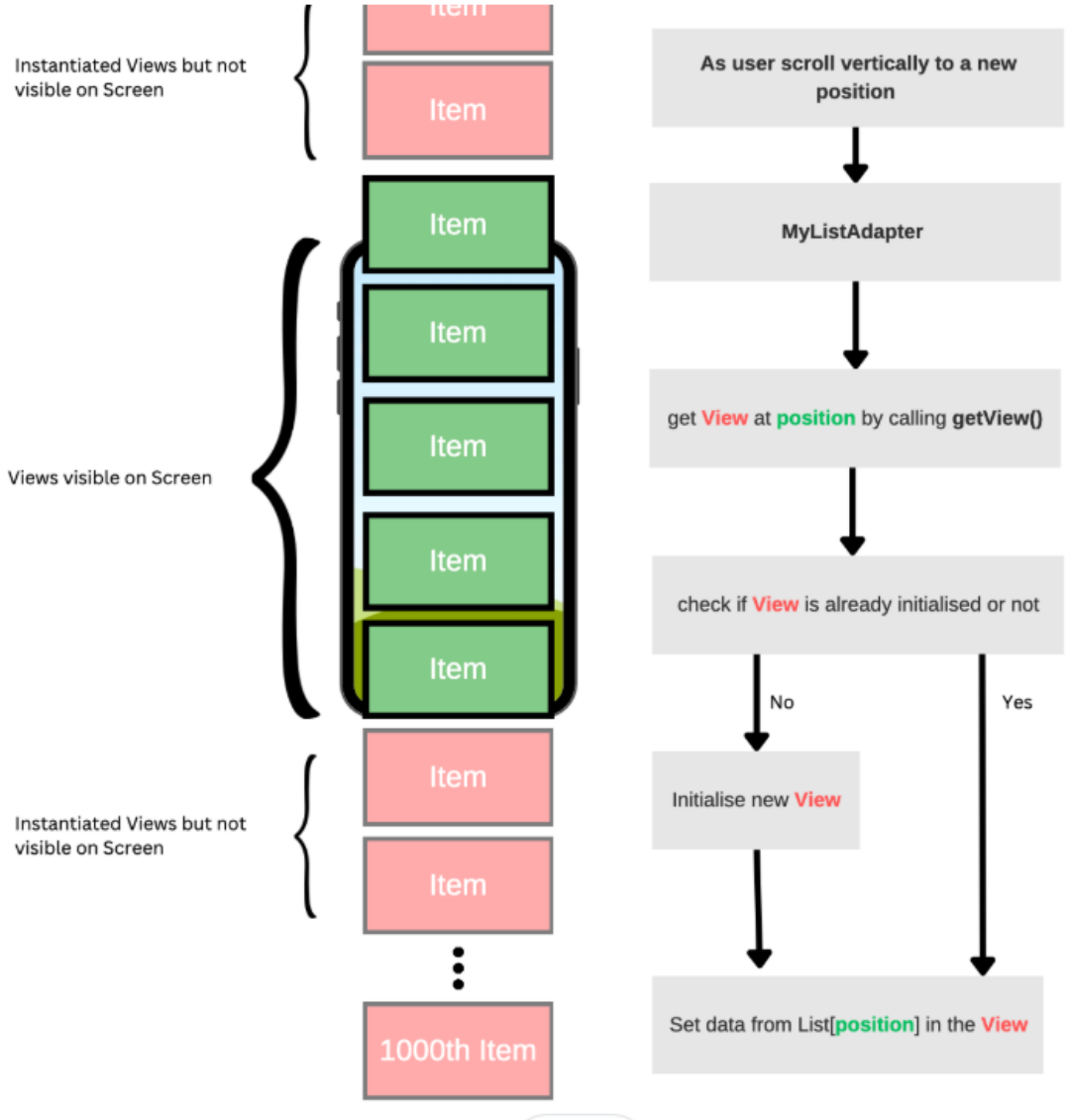
In this article, we'll explore the key differences between RecyclerView and ListView and give you a good understanding of when to use what and how and also appreciate why RecyclerView came into existence over ListView.

ListView

ListView was introduced in Android 1.0 and has been around since then. ListView was the go-to solution for displaying lists of data before RecyclerView was introduced.

One of the biggest advantages of using a ListView is that it's simpler to implement, easier to use. Here is an example of how simply ListViews can be implemented in Android.

```kotlin
class MyListAdapter(context: Context, private val data: ArrayList<String>)
    : ArrayAdapter<String>(context, R.layout.list_item, data) {

    // ViewHolder class to hold views for each item in the list
    private class ViewHolder(view: View) {
      // findViewById calls are made inside the ViewHolder
        val textView: TextView = view.findViewById(R.id.textView)
        val textView1: TextView = view.findViewById(R.id.textView1)
        val textView2: TextView = view.findViewById(R.id.textView2)
        val imageView: ImageView = view.findViewById(R.id.imageView)
    }

    // Override getView to create custom views for each item in the list
    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
        // Reference to the rowitem of the listview
        var row = convertView

        // Reference to the viewHolder
        val holder: ViewHolder

        // If the row hasn't been created yet, inflate the layout and create a ViewHolder to hold the
views
        if (row == null) {
            row = LayoutInflater.from(context).inflate(R.layout.list_item, parent, false)
            holder = ViewHolder(row)
            row.tag = holder
        } else {
            // If the row already exists, reuse the ViewHolder from the tag
            holder = row.tag as ViewHolder
        }

        // Set the data for the current position and set it on the views in the row
        holder.textView.text = data[position]
        holder.textView1.text = "Text 1"
        holder.textView2.text = "Text 2"
        holder.imageView.setImageResource(R.drawable.image)

        return row
    }
}
```



The getView function on a high level does the following:

- Gets each view item from the listview
- Find references to its child views
- Sets the correct data to those views depending upon the position
- Returns the created view item.

For each row item in the 1000 item list, we don't have to create 1000 different views, we can repopulate and reuse the **same set of views** with **different data** depending on the **position** in the list. This can be a major performance boost as we are saving tons of memory for a large list. This is called **View-Recycling** and is a major building block for RecyclerView which we will see in a while and here is a representation of how View Recycling works.

```
// Just an example of how simply ListView can be implemented
class MainActivity : AppCompatActivity() {

    // Reference to our view
    private lateinit var listView: ListView

    // List of data
    private val tutorials = arrayOf(
        "Item 1", "Item 2", "Item 3"
    )

    // OnCreate function
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Setting the reference to the ListView in the layout
        listView = findViewById(R.id.list)

        // Create an ArrayAdapter to populate the ListView with data
        val adapter = ArrayAdapter<String>(
            this,
            android.R.layout.simple_list_item_1,  // Use a built-in layout for the list item
            tutorials
        )

        // Set the adapter on the ListView
        listView.adapter = adapter

}
```

Now, we have recycled the views by a simple null check and saved memory but if we look inside the getView() function we ca see that we are trying to find the references to the child views by doing **findViewByID()** calls.

Depending upon how many child views there are, in my example code there are 4 so** for each item in the list, we are** **cal findViewByID() 4 times**.

Hence for a 1000 item list, there will be 4000 **findViewByID()** calls even though we have optimized the way in which the rowIt views are initialized. To help fix this problem for large lists, the **ViewHolder** pattern comes into play.

ViewHolder Pattern in Android

The ViewHolder pattern was created in Android to improve the performance of ListViews (and other AdapterView subclasses reducing the number of calls to findViewById().

When a ListView is scrolled, new views are created as needed to display the list items that become visible. Each time a new v is created, the findViewById() method is called to find the views in the layout and create references to them. This process car slow, especially for complex layouts with many views while also at the same time the instantiated views references are kept memory for the whole list which can grow directly proportional to the size of the list you are rendering.

The ViewHolder pattern addresses this performance issue by caching references to the views in the layout. When a view is recycled (i.e., reused for a different list item), the ViewHolder can simply update the views with new data, rather than having call findViewById() again.

Implementing ViewHolder Pattern in our ListView

Lets implement our ViewHolder class inside the MyListAdapter class.

```
// Creating our own adapter instead of using ArrayAdapter directly
class MyListAdapter(context: Context, private val data: ArrayList<String>)
    : ArrayAdapter<String>(context, R.layout.list_item, data) {

    // We have to Override getView to create custom views for each item in the list
    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
        var row = convertView

        // If the row hasn't been created yet, inflate the layout
        if (row == null) {
            row = LayoutInflater.from(context).inflate(R.layout.list_item, parent, false)
        }

        // Get the TextView in the row and set its text to the data for the current position
        // Note the findViewByID call here
        val textView = row.findViewById<TextView>(R.id.textView)
```

- Reuse the View for each item in the list instead of creating new ones for each item in the list.
- Reduce the number of **findViewById()** calls which in case of complex layouts and large number of items in the lists can take down the performance of the app significantly.

These are the two key things which are provided as a structure to the developers with RecyclerView apart from other features of customisations in RecyclerView.

Drawbacks of Using ListView

- Inefficient scrolling due to inefficient memory usage out of the box
- Lesser flexibility to customize how the list items should be positioned.
- Can only implement a vertically scrolling list.
- Implementing animations can be hard and complex out of the box
- Only offers notifyDataSetChanged() which is an inefficient way to handle updates.

RecyclerView

RecyclerView was introduced in Android 5.0 Lollipop as an upgrade over the ListView. It is designed to be more flexible and efficient, allowing developers to create complex layouts with minimal effort.

It uses "recycling" out of the box which we have seen above. It also has more flexible layout options, allowing you to create different types of lists with ease and also provides various methods to handle data set changes efficiently.

Let's use RecyclerView instead of ListView in our above implementation.

```
class MyListAdapter(private val data: Array<String>) :
    RecyclerView.Adapter<MyListAdapter.ViewHolder>() {

    // Inner ViewHolder class to hold views for each item in the list
    inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val textView: TextView = view.findViewById(R.id.textView)
    }

    // Called when RecyclerView needs a new ViewHolder of the given type to represent an item
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        // Inflate the layout for the item view
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.recycler_view_item, parent, false)

        // Create and return a new ViewHolder for the item view
        return ViewHolder(view)
    }

    // Called by RecyclerView to display the data at the specified position
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        // Get the data for the current position and set it on the views in the ViewHolder
        holder.textView.text = data[position]
    }

    // Returns the total number of items in the data set
    override fun getItemCount(): Int {
        return data.size
    }
}
```

As you can see there are multiple functions to override instead of just one getView() function of ArrayAdapter which makes the implementation of RecyclerViews not as beginner friendly as compared to ListView. It can also feel like an overkill implementation for the simplest of the lists in Android

Benefits of Using RecyclerView

- The major advantage of RecyclerView is its performance. It uses a view holder pattern out of the box, which reuses views from the RecyclerView pool and prevents the need to constantly inflate or create new views. This reduces the memory consumption of displaying a long list compared to ListViews and hence improves performance.
- With LayoutManager you can define how you want your list to be laid out, linearly, in a grid, horizontally, vertically rather than just vertically in a ListView.
- RecyclerView also offers a lot of customisation features over listview that make it easier to work with. For example, It supports drag and drop functionality, rearrange items in the list, item swiping gestures features like deleting or archiving items in the list. Below is an attached example code on how easy it is to extend the functionality to add swiping gestures.

```
// Set up the RecyclerView with a LinearLayoutManager and an adapter
recyclerView.layoutManager = LinearLayoutManager(this)
adapter = ItemAdapter(createItemList())
recyclerView.adapter = adapter

// Add support for drag and drop
val itemTouchHelper = ItemTouchHelper(object : ItemTouchHelper.Callback() {
    override fun getMovementFlags(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder
    ): Int {
        // Set the movement flags for drag and drop and swipe-to-dismiss
        val dragFlags = ItemTouchHelper.UP or ItemTouchHelper.DOWN
        val swipeFlags = ItemTouchHelper.START or ItemTouchHelper.END
        return makeMovementFlags(dragFlags, swipeFlags)
    }

    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean {
        // Swap the items in the adapter when dragged and dropped
        adapter.swapItems(viewHolder.adapterPosition, target.adapterPosition)
        return true
    }

    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        // Remove the item from the adapter when swiped to dismiss
        adapter.removeItem(viewHolder.adapterPosition)
    }
})

// Attach the ItemTouchHelper to the RecyclerView
itemTouchHelper.attachToRecyclerView(recyclerView)
```

MVP (Model View Presenter) Architecture Pattern in Android with Example

Last Updated : 14 Apr, 2025

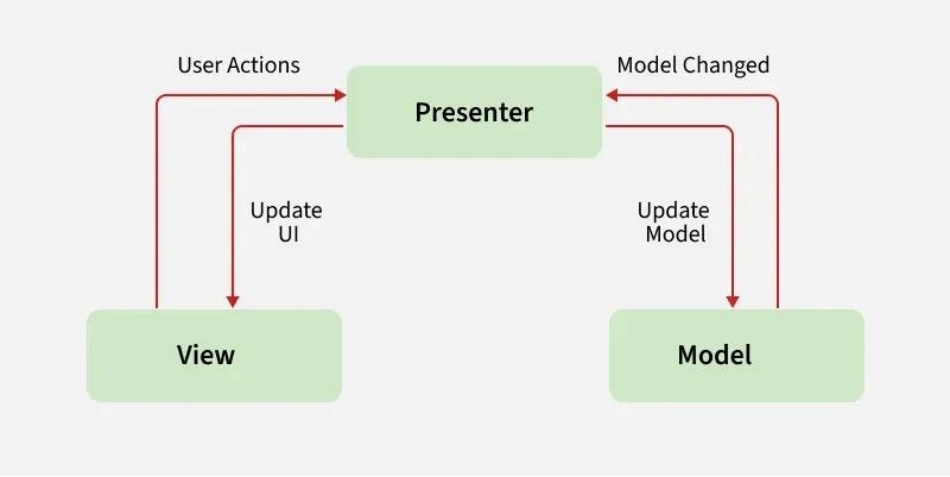
In the initial stages of Android development, learners do write codes in such a manner that eventually creates a **MainActivity** class which contains all the implementation logic(real-world business logic) of the application. This approach of app development leads to Android activity gets closely coupled to both UI and the application data processing mechanism. Further, it causes difficulties in the maintenance and scaling of such mobile applications. To avoid such problems in maintainability, readability, scalability, and refactoring of applications, developers prefer to define well-separated layers of code. By applying software architecture patterns, one can organize the code of the application to separate the concerns. **MVP (Model - View - Presenter)** architecture is one of the most popular architecture patterns and is valid in organizing the project.

MVP (Model - View - Presenter) comes into the picture as an alternative to the traditional **MVC (Model - View - Controller)** architecture pattern. Using MVC as the software architecture, developers end up with the following difficulties:

- Most of the core business logic resides in Controller. During the lifetime of an application, this file grows bigger and it becomes difficult to maintain the code.
- Because of tightly-coupled UI and data access mechanisms, both **Controller** and **View** layer falls in the same activity or fragment. This cause problem in making changes in the features of the application.
- It becomes hard to carry out Unit testing of the different layer as most of the part which are under testing needs Android SDK components.

MVP pattern overcomes these challenges of **MVC** and provides an easy way to structure the project codes. The reason why **MVP** is widely accepted is that it provides modularity, testability, and a more clean and maintainable codebase. It is composed of the following three components:

- **Model:** Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.
- **View:** UI(User Interface) layer. It provides the visualization of the data and keep a track of the user's action in order to notify the Presenter.
- **Presenter:** Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user's input notification from the View.



Key Points of MVP Architecture

1. Communication between View-Presenter and Presenter-Model happens via an **interface(also called Contract)**.
2. One Presenter class manages one View at a time i.e., there is a one-to-one relationship between Presenter and View.
3. Model and View class doesn't have knowledge about each other's existence.

Advantages of MVP Architecture

- No conceptual relationship in android components
- Easy code maintenance and testing as the application's model, view, and presenter layer are separated.

Disadvantages of MVP Architecture

- If the developer does not follow the single responsibility principle to break the code then the Presenter layer tends to expand to a huge all-knowing class.

Introduction to Model View View Model (MVVM)

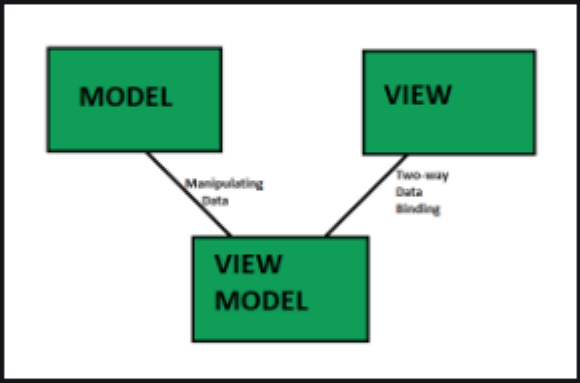
Last Updated : 01 Nov, 2023

Description of Model is as follows:

- **MODEL:** (Reusable Code – DATA) Business Objects that encapsulate data and behavior of application domain, Simply hold the data.
- **VIEW:** (Platform Specific Code - USER INTERFACE) What the user sees, The Formatted data.
- **VIEWMODEL:** (Reusable Code – LOGIC) Link between Model and View OR It Retrieves data from Model and exposes it to the View. This is the model specifically designed for the View.

Note: *Link between Model and View Model is Manipulating Data and between ViewModel and View is 2-way Data Binding.*

BASIC INTRODUCTION: [Way to Structure Code]



FEATURES:

- Life Cycle state of Application will be maintained.
- The application will be in the same position as where the user left it.
- UI Components are kept away from Business Logic.
- Business Logic is kept away from Database operations.
- Easy to understand and read.

BASIC EXAMPLE: We want to display Name in Purple Color (not written in the proper format, proper length) or Display Purple Color if Age of a person is > 18 years, Display Pink Color if Age of a person is < 18 years, Then the Logic of Purple and Pink Color would be present in ViewModel.

SUMMARY: From Server, Get Data(available in Model Objects), View Model reads Model Objects and then facilitates the easy presentation of data on the view.

The primary differences between MVVM AND MVC are as follows:

MVVM	MVC
The Model is somewhat similar to MVC but here we have ViewModels which are passed to the view and all the logic is in the ViewModel and hence no controller is there. Example: Knockout.js	In this pattern, we have models which are basic objects with no code and just properties, views that contribute to presentation items (HTML, WinForms, etc), client-side deletes, and Controllers that focus on the logic part. Examples: ASP.NET MVC, Angular
In MVVM your DeletePerson would be called off of your view model	We have a PersonController with an Action DeletePerson that delete a person
We are on the client side so we can hold on to objects and do a lot more logic in a non-disconnected state.	MVC is typically used when things are transactional and disconnected as is the case with server-side web. In ASP MVC we send the view through the wire and then the transaction with the client is over.

ADVANTAGES:

- Maintainability – Can remain agile and keep releasing successive versions quickly.
- Extensibility – Have the ability to replace or add new pieces of code.
- Testability - Easier to write unit tests against a core logic.
- Transparent Communication - The view model provides a transparent interface to the view controller, which it uses to populate the view layer and interact with the model layer, which results in a transparent communication between the layers of your application.

DISADVANTAGES:

- Some people think that for simple UIs, MVVM can be overkill.
- In bigger cases, it can be hard to design the ViewModel.
- Debugging would be a bit difficult when we have complex data bindings.

https://www.geeksforgeeks.org/how-to-publish-your-android-app-on-google-play-store/