https://developer.android.com/guide/components/processes-and-threads

<https://www.tutorialspoint.com/android/android_services.htm>

<https://www.geeksforgeeks.org/broadcast-receiver-in-android-with-example/>

<https://developer.android.com/guide/components/broadcasts>

<https://www.geeksforgeeks.org/how-to-send-sms-in-android-using-kotlin/>

https://www.geeksforgeeks.org/sms-full-form/

# Processes and threads overview

When an application component starts and the application does not have any other components running, the Android system starts a new Linux process for the application with a single thread of execution. By default, all components of the same application run in the same process and thread (called the "main" thread). If an application component starts and there already exists a process for that application (because another component from the application exists), then the component is started within that process and uses the same thread of execution. However, you can arrange for different components in your application to run in separate processes, and you can create additional threads for any process.

This document discusses how processes and threads work in an Android application.

## Processes

By default, all components of the same application run in the same process and most applications should not change this. However, if you find that you need to control which process a certain component belongs to, you can do so in the manifest file.

The manifest entry for each type of component element—[<activity>](https://developer.android.com/guide/topics/manifest/activity-element), [<service>](https://developer.android.com/guide/topics/manifest/service-element), [<receiver>](https://developer.android.com/guide/topics/manifest/receiver-element), and [<provider>](https://developer.android.com/guide/topics/manifest/provider-element)—supports an android:process attribute that can specify a process in which that component should run. You can set this attribute so that each component runs in its own process or so that some components share a process while others do not. You can also set android:process so that components of different applications run in the same process—provided that the applications share the same Linux user ID and are signed with the same certificates.

The [<application>](https://developer.android.com/guide/topics/manifest/application-element) element also supports an android:process attribute, to set a default value that applies to all components.

Android might decide to shut down a process at some point, when resources are required by other processes that are more immediately serving the user. Application components running in the process that's killed are consequently destroyed. A process is started again for those components when there's again work for them to do.

When deciding which processes to kill, the Android system weighs their relative importance to the user. For example, it more readily shuts down a process hosting activities that are no longer visible on screen, compared to a process hosting visible activities. The decision whether to terminate a process, therefore, depends on the state of the components running in that process.

The details of the process lifecycle and its relationship to application states are discussed in [Processes and Application Lifecycle](https://developer.android.com/guide/topics/processes/process-lifecycle).

## Threads

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

The system does not create a separate thread for each instance of a component. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread. Consequently, methods that respond to system callbacks (such as [onKeyDown()](https://developer.android.com/reference/android/view/View#onKeyDown(int, android.view.KeyEvent)) to report user actions or a lifecycle callback method) always run in the UI thread of the process.

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

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

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

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

### Worker threads

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

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

To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:

* [Activity.runOnUiThread(Runnable)](https://developer.android.com/reference/android/app/Activity#runOnUiThread(java.lang.Runnable))
* [View.post(Runnable)](https://developer.android.com/reference/android/view/View#post(java.lang.Runnable))
* [View.postDelayed(Runnable, long)](https://developer.android.com/reference/android/view/View#postDelayed(java.lang.Runnable, long))

[Kotlin](https://developer.android.com/guide/components/processes-and-threads#kotlin)[Java](https://developer.android.com/guide/components/processes-and-threads#java)

fun onClick(v: View) {  
    Thread(Runnable {  
        // a potentially time consuming task  
        val bitmap = processBitMap("image.png")  
        imageView.post {  
            imageView.setImageBitmap(bitmap)  
        }  
    }).start()  
}

This implementation is thread-safe: the background operation is done from a separate thread while the [ImageView](https://developer.android.com/reference/android/widget/ImageView) is always manipulated from the UI thread.

However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a [Handler](https://developer.android.com/reference/android/os/Handler) in your worker thread, to process messages delivered from the UI thread. See [Threading on Android](https://developer.android.com/training/multiple-threads) for a full explanation of how to schedule work on background threads and communicate back to the UI thread.

### Thread-safe methods

In some situations, the methods you implement might be called from more than one thread, and therefore must be written to be thread-safe.

This is primarily true for methods that can be called remotely—such as methods in a [bound service](https://developer.android.com/guide/components/bound-services). When a call on a method implemented in an [IBinder](https://developer.android.com/reference/android/os/IBinder) originates in the same process in which the [IBinder](https://developer.android.com/reference/android/os/IBinder) is running, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that the system maintains in the same process as the [IBinder](https://developer.android.com/reference/android/os/IBinder) (it's not executed in the UI thread of the process). For example, whereas a service's [onBind()](https://developer.android.com/reference/android/app/Service#onBind(android.content.Intent)) method would be called from the UI thread of the service's process, methods implemented in the object that [onBind()](https://developer.android.com/reference/android/app/Service#onBind(android.content.Intent)) returns (for example, a subclass that implements RPC methods) would be called from threads in the pool. Because a service can have more than one client, more than one pool thread can engage the same [IBinder](https://developer.android.com/reference/android/os/IBinder) method at the same time. [IBinder](https://developer.android.com/reference/android/os/IBinder) methods must, therefore, be implemented to be thread-safe.

Similarly, a content provider can receive data requests that originate in other processes. Although the [ContentResolver](https://developer.android.com/reference/android/content/ContentResolver) and [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider) classes hide the details of how the interprocess communication is managed, [ContentProvider](https://developer.android.com/reference/android/content/ContentProvider) methods that respond to those requests—the methods [query()](https://developer.android.com/reference/android/content/ContentProvider#query(android.net.Uri, java.lang.String[], android.os.Bundle, android.os.CancellationSignal)), [insert()](https://developer.android.com/reference/android/content/ContentProvider#insert(android.net.Uri, android.content.ContentValues)), [delete()](https://developer.android.com/reference/android/content/ContentProvider#delete(android.net.Uri, java.lang.String, java.lang.String[])), [update()](https://developer.android.com/reference/android/content/ContentProvider#update(android.net.Uri, android.content.ContentValues, java.lang.String, java.lang.String[])), and [getType()](https://developer.android.com/reference/android/content/ContentProvider#getType(android.net.Uri))—are called from a pool of threads in the content provider's process, not the UI thread for the process. Because these methods might be called from any number of threads at the same time, they too must be implemented to be thread-safe.

## Interprocess communication

Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller. This entails decomposing a method call and its data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, then reassembling and reenacting the call there. Return values are then transmitted in the opposite direction. Android provides all the code to perform these IPC transactions, so you can focus on defining and implementing the RPC programming interface.

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

State & Description

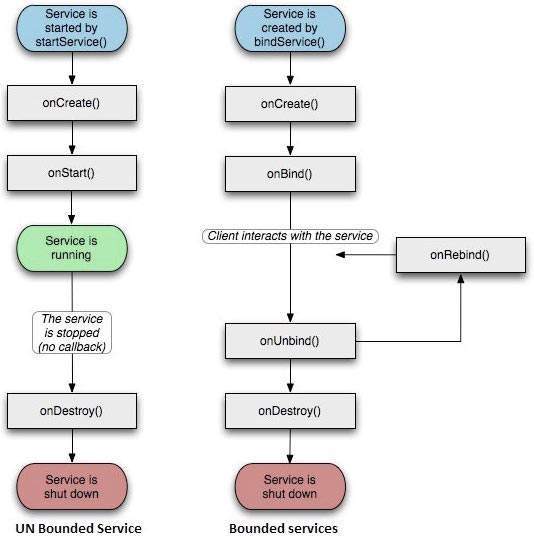
**Started**

A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Once started, a service can run in the background indefinitely, even if the component that started it is destroyed.

**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, get results, and even do so across processes with interprocess communication (IPC).

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



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

|  |  |
| --- | --- |
| **Sr.No.** | **Callback & Description** |
| 1 | **onStartCommand()**  The system calls this method when another component, such as an activity, requests that the service be started, by calling *startService()*. If you implement this method, it is your responsibility to stop the service when its work is done, by calling *stopSelf()* or *stopService()* methods. |
| 2 | **onBind()**  The system calls this method when another component wants to bind with the service by calling *bindService()*. If you implement this method, you must provide an interface that clients use to communicate with the service, by returning an *IBinder* object. You must always implement this method, but if you don't want to allow binding, then you should return *null*. |
| 3 | **onUnbind()**  The system calls this method when all clients have disconnected from a particular interface published by the service. |
| 4 | **onRebind()**  The system calls this method when new clients have connected to the service, after it had previously been notified that all had disconnected in its *onUnbind(Intent)*. |
| 5 | **onCreate()**  The system calls this method when the service is first created using *onStartCommand()* or *onBind()*. This call is required to perform one-time set-up. |
| 6 | **onDestroy()**  The system calls this method when the service is no longer used and is being destroyed. Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. |

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

package com.tutorialspoint;

import android.app.Service;

import android.os.IBinder;

import android.content.Intent;

import android.os.Bundle;

public class HelloService extends Service {

/\*\* indicates how to behave if the service is killed \*/

int mStartMode;

/\*\* interface for clients that bind \*/

IBinder mBinder;

/\*\* indicates whether onRebind should be used \*/

boolean mAllowRebind;

/\*\* Called when the service is being created. \*/

@Override

public void onCreate() {

}

/\*\* The service is starting, due to a call to startService() \*/

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

return mStartMode;

}

/\*\* A client is binding to the service with bindService() \*/

@Override

public IBinder onBind(Intent intent) {

return mBinder;

}

/\*\* Called when all clients have unbound with unbindService() \*/

@Override

public boolean onUnbind(Intent intent) {

return mAllowRebind;

}

/\*\* Called when a client is binding to the service with bindService()\*/

@Override

public void onRebind(Intent intent) {

}

/\*\* Called when The service is no longer used and is being destroyed \*/

@Override

public void onDestroy() {

}

}

**Example**

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

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | You will use Android StudioIDE to create an Android application and name it as *My Application* under a package *com.example.tutorialspoint7.myapplication* as explained in the *Hello World Example* chapter. |
| 2 | Modify main activity file *MainActivity.java* to add *startService()* and *stopService()* methods. |
| 3 | Create a new java file *MyService.java* under the package *com.example.My Application*. This file will have implementation of Android service related methods. |
| 4 | Define your service in *AndroidManifest.xml* file using <service.../> tag. An application can have one or more services without any restrictions. |
| 5 | Modify the default content of *res/layout/activity\_main.xml* file to include two buttons in linear layout. |
| 6 | No need to change any constants in *res/values/strings.xml* file. Android studio take care of string values |
| 7 | Run the application to launch Android emulator and verify the result of the changes done in the application. |

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

package com.example.tutorialspoint7.myapplication;

import android.content.Intent;

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.os.Bundle;

import android.app.Activity;

import android.util.Log;

import android.view.View;

public class MainActivity extends Activity {

String msg = "Android : ";

/\*\* Called when the activity is first created. \*/

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

Log.d(msg, "The onCreate() event");

}

public void startService(View view) {

startService(new Intent(getBaseContext(), MyService.class));

}

// Method to stop the service

public void stopService(View view) {

stopService(new Intent(getBaseContext(), MyService.class));

}

}

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

package com.example.tutorialspoint7.myapplication;

import android.app.Service;

import android.content.Intent;

import android.os.IBinder;

import android.support.annotation.Nullable;

import android.widget.Toast;

/\*\*

\* Created by TutorialsPoint7 on 8/23/2016.

\*/

public class MyService extends Service {

@Nullable

@Override

public IBinder onBind(Intent intent) {

return null;

}

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

// Let it continue running until it is stopped.

Toast.makeText(this, "Service Started", Toast.LENGTH\_LONG).show();

return START\_STICKY;

}

@Override

public void onDestroy() {

super.onDestroy();

Toast.makeText(this, "Service Destroyed", Toast.LENGTH\_LONG).show();

}

}

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

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.tutorialspoint7.myapplication">

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<activity android:name=".MainActivity">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

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

</intent-filter>

</activity>

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

</application>

</manifest>

Following will be the content of **res/layout/activity\_main.xml** file to include two buttons −

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools" android:layout\_width="match\_parent"

android:layout\_height="match\_parent" android:paddingLeft="@dimen/activity\_horizontal\_margin"

android:paddingRight="@dimen/activity\_horizontal\_margin"

android:paddingTop="@dimen/activity\_vertical\_margin"

android:paddingBottom="@dimen/activity\_vertical\_margin" tools:context=".MainActivity">

<TextView

android:id="@+id/textView1"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Example of services"

android:layout\_alignParentTop="true"

android:layout\_centerHorizontal="true"

android:textSize="30dp" />

<TextView

android:id="@+id/textView2"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Tutorials point "

android:textColor="#ff87ff09"

android:textSize="30dp"

android:layout\_above="@+id/imageButton"

android:layout\_centerHorizontal="true"

android:layout\_marginBottom="40dp" />

<ImageButton

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/imageButton"

android:src="@drawable/abc"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:id="@+id/button2"

android:text="Start Services"

android:onClick="startService"

android:layout\_below="@+id/imageButton"

android:layout\_centerHorizontal="true" />

<Button

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Stop Services"

android:id="@+id/button"

android:onClick="stopService"

android:layout\_below="@+id/button2"

android:layout\_alignLeft="@+id/button2"

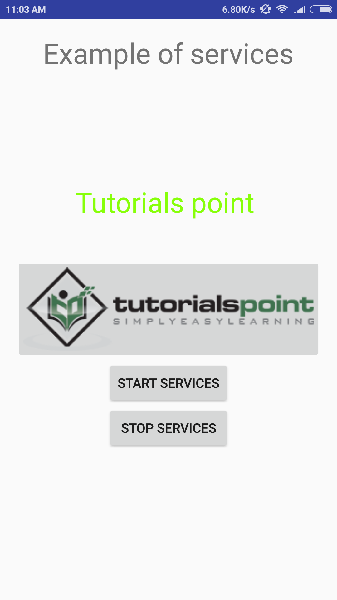
android:layout\_alignStart="@+id/button2"

android:layout\_alignRight="@+id/button2"

android:layout\_alignEnd="@+id/button2" />

</RelativeLayout>

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



Broadcast in android is the system-wide events that can occur when the device starts, when a message is received on the device or when incoming calls are received, or when a device goes to airplane mode, etc. Broadcast Receivers are used to respond to these system-wide events. Broadcast Receivers allow us to register for the system and application events, and when that event happens, then the register receivers get notified. There are mainly two types of Broadcast Receivers:

* **Static Broadcast Receivers:** These types of Receivers are declared in the manifest file and works even if the app is closed.
* **Dynamic Broadcast Receivers:** These types of receivers work only if the app is active or minimized.

# Broadcasts overview

Android apps can send or receive broadcast messages from the Android system and other Android apps, similar to the [publish-subscribe](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern) design pattern. These broadcasts are sent when an event of interest occurs. For example, the Android system sends broadcasts when various system events occur, such as when the system boots up or the device starts charging. Apps can also send custom broadcasts, for example, to notify other apps of something that they might be interested in (for example, some new data has been downloaded).

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.

Generally speaking, 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, as described in the following video.

## About system broadcasts

The system automatically sends broadcasts when various system events occur, such as when the system switches in and out of airplane mode. System broadcasts are sent to all apps that are subscribed to receive the event.

The broadcast message itself is wrapped in an [Intent](https://developer.android.com/reference/android/content/Intent) object whose action string identifies the event that occurred (for example android.intent.action.AIRPLANE\_MODE).

The intent may 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.

For more information about how to read intents and get the action string from an intent, see [Intents and Intent Filters](https://developer.android.com/guide/components/intents-filters).

For a complete list of system broadcast actions, see the BROADCAST\_ACTIONS.TXT file in the Android SDK. Each broadcast action has a constant field associated with it. For example, the value of the constant [ACTION\_AIRPLANE\_MODE\_CHANGED](https://developer.android.com/reference/android/content/Intent#ACTION_AIRPLANE_MODE_CHANGED) is android.intent.action.AIRPLANE\_MODE. Documentation for each broadcast action is available in its associated constant field.

### Changes to system broadcasts

As the Android platform evolves, it periodically changes how system broadcasts behave. Keep the following changes in mind if your app targets Android 7.0 (API level 24) or higher, or if it's installed on devices running Android 7.0 or higher.

#### Android 9

Beginning with Android 9 (API level 28), The [NETWORK\_STATE\_CHANGED\_ACTION](https://developer.android.com/reference/android/net/wifi/WifiManager#NETWORK_STATE_CHANGED_ACTION) broadcast doesn't receive information about the user's location or personally identifiable data.

In addition, if your app is installed on a device running Android 9 or higher, system broadcasts from Wi-Fi don't contain SSIDs, BSSIDs, connection information, or scan results. To get this information, call [getConnectionInfo()](https://developer.android.com/reference/android/net/wifi/WifiManager#getConnectionInfo()) instead.

#### Android 8.0

Beginning with Android 8.0 (API level 26), the system imposes additional restrictions on manifest-declared receivers.

If your app targets Android 8.0 or higher, you cannot use the manifest to declare a receiver for most implicit broadcasts (broadcasts that don't target your app specifically). You can still use a [context-registered receiver](https://developer.android.com/guide/components/broadcasts#context-registered-recievers) when the user is actively using your app.

#### Android 7.0

Android 7.0 (API level 24) and higher don't send the following system broadcasts:

* [ACTION\_NEW\_PICTURE](https://developer.android.com/reference/android/hardware/Camera#ACTION_NEW_PICTURE)
* [ACTION\_NEW\_VIDEO](https://developer.android.com/reference/android/hardware/Camera#ACTION_NEW_VIDEO)

Also, apps targeting Android 7.0 and higher must register the [CONNECTIVITY\_ACTION](https://developer.android.com/reference/android/net/ConnectivityManager#CONNECTIVITY_ACTION) broadcast using [registerReceiver(BroadcastReceiver, IntentFilter)](https://developer.android.com/reference/android/content/Context#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)). Declaring a receiver in the manifest doesn't work.

## Receiving broadcasts

Apps can receive broadcasts in two ways: through manifest-declared receivers and context-registered receivers.

### Manifest-declared receivers

If you declare a broadcast receiver in your manifest, the system launches your app (if the app is not already running) when the broadcast is sent.

**Note:** If your app targets API level 26 or higher, you cannot use the manifest to declare a receiver for implicit broadcasts (broadcasts that do not target your app specifically), except for a few implicit broadcasts that are [exempted from that restriction](https://developer.android.com/guide/components/broadcast-exceptions). In most cases, you can use [scheduled jobs](https://developer.android.com/topic/performance/scheduling) instead.

To declare a broadcast receiver in the manifest, perform the following steps:

1. Specify the [<receiver>](https://developer.android.com/guide/topics/manifest/receiver-element) element in your app's manifest.

<!-- If this receiver listens for broadcasts sent from the system or from  
     other apps, even other apps that you own, set android:exported to "true". -->  
<receiver android:name=".MyBroadcastReceiver" android:exported="false">  
    <intent-filter>  
        <action android:name="APP\_SPECIFIC\_BROADCAST

 " />  
    </intent-filter>  
</receiver>

The intent filters specify the broadcast actions your receiver subscribes to.

 Subclass [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver) and implement [onReceive(Context, Intent)](https://developer.android.com/reference/android/content/BroadcastReceiver#onReceive(android.content.Context, android.content.Intent)). The broadcast receiver in the following example logs and displays the contents of the broadcast:

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

1. private const val TAG = "MyBroadcastReceiver"  
     
   class MyBroadcastReceiver : BroadcastReceiver() {  
     
       override fun onReceive(context: Context, intent: Intent) {  
           StringBuilder().apply {  
               append("Action: ${intent.action}\n")  
               append("URI: ${intent.toUri(Intent.URI\_INTENT\_SCHEME)}\n")  
               toString().also { log ->  
                   Log.d(TAG, log)  
                   Toast.makeText(context, log, Toast.LENGTH\_LONG).show()  
               }  
           }  
       }  
   }

The system package manager registers the receiver when the app is installed. The receiver then becomes a separate entry point into your app which means that the system can start the app and deliver the broadcast if the app is not currently running.

The system creates a new [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver) component object to handle each broadcast that it receives. This object is valid only for the duration of the call to [onReceive(Context, Intent)](https://developer.android.com/reference/android/content/BroadcastReceiver#onReceive(android.content.Context, android.content.Intent)). Once your code returns from this method, the system considers the component no longer active.

### Context-registered receivers

Context-registered receivers receive broadcasts as long as their registering context is valid. For an example, if you register within an [Activity](https://developer.android.com/reference/android/app/Activity) context, you receive broadcasts as long as the activity is not destroyed. If you register with the Application context, you receive broadcasts as long as the app is running.

To register a receiver with a context, perform the following steps:

1. In your app's module-level build file, include version 1.9.0 or higher of the [AndroidX Core library](https://developer.android.com/jetpack/androidx/releases/core):

[Groovy](https://developer.android.com/guide/components/broadcasts#groovy)[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)

 dependencies {  
    def core\_version = "1.9.0"  
  
    // Java language implementation  
    implementation "androidx.core:core:$core\_version"  
    // Kotlin  
    implementation "androidx.core:core-ktx:$core\_version"  
  
    // To use RoleManagerCompat  
    implementation "androidx.core:core-role:1.0.0"  
  
    // To use the Animator APIs  
    implementation "androidx.core:core-animation:1.0.0-beta01"  
    // To test the Animator APIs  
    androidTestImplementation "androidx.core:core-animation-testing:1.0.0-beta01"  
  
    // Optional - To enable APIs that query the performance characteristics of GMS devices.  
    implementation "androidx.core:core-performance:1.0.0-alpha02"  
  
    // Optional - to use ShortcutManagerCompat to donate shortcuts to be used by Google  
    implementation "androidx.core:core-google-shortcuts:1.1.0"  
  
    // Optional - to support backwards compatibility of RemoteViews  
    implementation "androidx.core:core-remoteviews:1.0.0-beta03"  
  
    // Optional - APIs for SplashScreen, including compatibility helpers on devices prior Android 12  
    implementation "androidx.core:core-splashscreen:1.0.0"  
}

 Create an instance of [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver):

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

 val br: BroadcastReceiver = MyBroadcastReceiver()

 Create an instance of [IntentFilter](https://developer.android.com/reference/android/content/IntentFilter):

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

val filter = IntentFilter(APP\_SPECIFIC\_BROADCAST

 )

 Choose whether the broadcast receiver should be exported and visible to other apps on the device. If this receiver is listening for broadcasts sent from the system or from other apps—even other apps that you own—use the RECEIVER\_EXPORTED flag. If instead this receiver is listening only for broadcasts sent by your app, use the RECEIVER\_NOT\_EXPORTED flag.

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

 val listenToBroadcastsFromOtherApps = false  
val receiverFlags = if (listenToBroadcastsFromOtherApps) {  
    ContextCompat.RECEIVER\_EXPORTED  
} else {  
    ContextCompat.RECEIVER\_NOT\_EXPORTED  
}

**Caution:** If the broadcast receiver is exported, other apps could send unprotected broadcasts to your app.

 Register the receiver by calling [registerReceiver()](https://developer.android.com/reference/androidx/core/content/ContextCompat#registerReceiver(android.content.Context,android.content.BroadcastReceiver,android.content.IntentFilter,int)):

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

1. ContextCompat.registerReceiver(context, br, filter, receiverFlags)
2. To stop receiving broadcasts, call [unregisterReceiver(android.content.BroadcastReceiver)](https://developer.android.com/reference/android/content/Context#unregisterReceiver(android.content.BroadcastReceiver)). Be sure to unregister the receiver when you no longer need it or the context is no longer valid.

Be mindful of where you register and unregister the receiver, for example, if you register a receiver in [onCreate(Bundle)](https://developer.android.com/reference/android/app/Activity#onCreate(android.os.Bundle)) using the activity's context, you should unregister it in [onDestroy()](https://developer.android.com/reference/android/app/Activity#onDestroy()) to prevent leaking the receiver out of the activity context. If you register a receiver in [onResume()](https://developer.android.com/reference/android/app/Activity#onResume()), you should unregister it in [onPause()](https://developer.android.com/reference/android/app/Activity#onPause()) to prevent registering it multiple times (If you don't want to receive broadcasts when paused, and this can cut down on unnecessary system overhead). Do not unregister in [onSaveInstanceState(Bundle)](https://developer.android.com/reference/android/app/Activity#onSaveInstanceState(android.os.Bundle)), because this isn't called if the user moves back in the history stack.

### Effects on process state

The state of your [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver) (whether it is running or not) affects the state of its containing process, which can in turn affect its likelihood of being killed by the system. For example, when a process executes a receiver (that is, currently running the code in its [onReceive()](https://developer.android.com/reference/android/content/BroadcastReceiver#onReceive(android.content.Context, android.content.Intent)) method), it is considered to be a foreground process. The system keeps the process running except under cases of extreme memory pressure.

However, once your code returns from onReceive(), the BroadcastReceiver is no longer active. The receiver's host process becomes only as important as the other app components that are running in it. If that process hosts only a manifest-declared receiver (a common case for apps that the user has never or not recently interacted with), then upon returning from onReceive(), the system considers its process to be a low-priority process and may kill it to make resources available for other more important processes.

For this reason, you should not start long running background threads from a broadcast receiver. After onReceive(), the system can kill the process at any time to reclaim memory, and in doing so, it terminates the spawned thread running in the process. To avoid this, you should either call [goAsync()](https://developer.android.com/reference/android/content/BroadcastReceiver#goAsync()) (if you want a little more time to process the broadcast in a background thread) or schedule a [JobService](https://developer.android.com/reference/android/app/job/JobService) from the receiver using the [JobScheduler](https://developer.android.com/reference/android/app/job/JobScheduler), so the system knows that the process continues to perform active work. For more information, see [Processes and Application Life Cycle](https://developer.android.com/guide/topics/processes/process-lifecycle).

The following snippet shows a [BroadcastReceiver](https://developer.android.com/reference/android/content/BroadcastReceiver) that uses [goAsync()](https://developer.android.com/reference/android/content/BroadcastReceiver#goAsync()) to flag that it needs more time to finish after onReceive() is complete. This is especially useful if the work you want to complete in your onReceive() is long enough to cause the UI thread to miss a frame (>16ms), making it better suited for a background thread.

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

class MyBroadcastReceiver : BroadcastReceiver() {  
    private val scope = CoroutineScope(SupervisorJob())  
  
    override fun onReceive(context: Context, intent: Intent) {  
        val pendingResult: PendingResult = goAsync()  
  
        scope.launch(Dispatchers.Default) {  
            try {  
                // Do some background work  
                buildString {  
                    append("Action: ").append(intent.action).append("\n")  
                    append("URI: ").append(intent.toUri(Intent.URI\_INTENT\_SCHEME)).append("\n")  
                }.also { log ->  
                    Log.d(TAG, log)  
                }  
            } finally {  
                // Must call finish() so the BroadcastReceiver can be recycled  
                pendingResult.finish()  
            }  
        }  
    }  
  
    companion object {  
        private const val TAG = "MyBroadcastReceiver"  
    }  
}

## Sending broadcasts

Android provides three ways for apps to send broadcast:

* The [sendOrderedBroadcast(Intent, String)](https://developer.android.com/reference/android/content/Context#sendOrderedBroadcast(android.content.Intent, java.lang.String)) method sends broadcasts to one receiver at a time. As each receiver executes in turn, it can propagate a result to the next receiver, or it can completely abort the broadcast so that it won't be passed to other receivers. The order receivers run in can be controlled with the android:priority attribute of the matching intent-filter; receivers with the same priority will be run in an arbitrary order.
* The [sendBroadcast(Intent)](https://developer.android.com/reference/android/content/Context#sendBroadcast(android.content.Intent)) method sends broadcasts to all receivers in an undefined order. This is called a Normal Broadcast. This is more efficient, but means that receivers cannot read results from other receivers, propagate data received from the broadcast, or abort the broadcast.
* The [LocalBroadcastManager.sendBroadcast](https://developer.android.com/reference/androidx/localbroadcastmanager/content/LocalBroadcastManager#sendBroadcast(android.content.Intent)) method sends broadcasts to receivers that are in the same app as the sender. If you don't need to send broadcasts across apps, use local broadcasts. The implementation is much more efficient (no interprocess communication needed) and you don't need to worry about any security issues related to other apps being able to receive or send your broadcasts.

The following code snippet demonstrates how to send a broadcast by creating an Intent and calling [sendBroadcast(Intent)](https://developer.android.com/reference/android/content/Context#sendBroadcast(android.content.Intent)).

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

Intent().also { intent ->  
    intent.setAction("com.example.broadcast.MY\_NOTIFICATION")  
    intent.putExtra("data", "Nothing to see here, move along.")  
    sendBroadcast(intent)  
}

The broadcast message is wrapped in an [Intent](https://developer.android.com/reference/android/content/Intent) object. The intent's action string must provide the app's Java package name syntax and uniquely identify the broadcast event. You can attach additional information to the intent with [putExtra(String, Bundle)](https://developer.android.com/reference/android/content/Intent#putExtra(java.lang.String, android.os.Bundle)). You can also limit a broadcast to a set of apps in the same organization by calling [setPackage(String)](https://developer.android.com/reference/android/content/Intent#setPackage(java.lang.String)) on the intent.

**Note:** Although intents are used for both sending broadcasts and starting activities with [startActivity(Intent)](https://developer.android.com/reference/android/content/Context#startActivity(android.content.Intent)), these actions are completely unrelated. Broadcast receivers can't see or capture intents used to start an activity; likewise, when you broadcast an intent, you can't find or start an activity.

## Restricting broadcasts with permissions

Permissions allow you to restrict broadcasts to the set of apps that hold certain permissions. You can enforce restrictions on either the sender or receiver of a broadcast.

### Sending with permissions

When you call [sendBroadcast(Intent, String)](https://developer.android.com/reference/android/content/Context#sendBroadcast(android.content.Intent, java.lang.String)) or [sendOrderedBroadcast(Intent, String, BroadcastReceiver, Handler, int, String, Bundle)](https://developer.android.com/reference/android/content/Context#sendOrderedBroadcast(android.content.Intent, java.lang.String, android.content.BroadcastReceiver, android.os.Handler, int, java.lang.String, android.os.Bundle)), you can specify a permission parameter. Only receivers who have requested that permission with the tag in their manifest (and subsequently been granted the permission if it is dangerous) can receive the broadcast. For example, the following code sends a broadcast:

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

sendBroadcast(Intent("com.example.NOTIFY"), Manifest.permission.SEND\_SMS)

To receive the broadcast, the receiving app must request the permission as shown below:

<uses-permission android:name="android.permission.SEND\_SMS"/>

You can specify either an existing system permission like [SEND\_SMS](https://developer.android.com/reference/android/Manifest.permission#SEND_SMS) or define a custom permission with the [<permission>](https://developer.android.com/guide/topics/manifest/permission-element) element. For information on permissions and security in general, see the [System Permissions](https://developer.android.com/guide/topics/security/permissions).

**Note:** Custom permissions are registered when the app is installed. The app that defines the custom permission must be installed before the app that uses it.

### Receiving with permissions

If you specify a permission parameter when registering a broadcast receiver (either with [registerReceiver(BroadcastReceiver, IntentFilter, String, Handler)](https://developer.android.com/reference/android/content/Context#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter, java.lang.String, android.os.Handler)) or in [<receiver>](https://developer.android.com/guide/topics/manifest/receiver-element) tag in your manifest), then only broadcasters who have requested the permission with the [<uses-permission>](https://developer.android.com/guide/topics/manifest/uses-permission-element) tag in their manifest (and subsequently been granted the permission if it is dangerous) can send an Intent to the receiver.

For example, assume your receiving app has a manifest-declared receiver as shown below:

<receiver android:name=".MyBroadcastReceiver"  
          android:permission="android.permission.SEND\_SMS">  
    <intent-filter>  
        <action android:name="android.intent.action.AIRPLANE\_MODE"/>  
    </intent-filter>  
</receiver>

Or your receiving app has a context-registered receiver as shown below:

[Kotlin](https://developer.android.com/guide/components/broadcasts#kotlin)[Java](https://developer.android.com/guide/components/broadcasts#java)

var filter = IntentFilter(Intent.ACTION\_AIRPLANE\_MODE\_CHANGED)  
registerReceiver(receiver, filter, Manifest.permission.SEND\_SMS, null )

Then, to be able to send broadcasts to those receivers, the sending app must request the permission as shown below:

<uses-permission android:name="android.permission.SEND\_SMS"/>

## Security considerations and best practices

Here are some security considerations and best practices for sending and receiving broadcasts:

* If you don't need to send broadcasts to components outside of your app, then send and receive local broadcasts with the [LocalBroadcastManager](https://developer.android.com/reference/androidx/localbroadcastmanager/content/LocalBroadcastManager) which is available in the [Support Library](https://developer.android.com/topic/libraries/support-library). The [LocalBroadcastManager](https://developer.android.com/reference/androidx/localbroadcastmanager/content/LocalBroadcastManager) is much more efficient (no interprocess communication needed) and allows you to avoid thinking about any security issues related to other apps being able to receive or send your broadcasts. Local Broadcasts can be used as a general purpose pub/sub event bus in your app without any overheads of system wide broadcasts.
* If many apps have registered to receive the same broadcast in their manifest, it can cause the system to launch a lot of apps, causing a substantial impact on both device performance and user experience. To avoid this, prefer using context registration over manifest declaration. Sometimes, the Android system itself enforces the use of context-registered receivers. For example, the [CONNECTIVITY\_ACTION](https://developer.android.com/reference/android/net/ConnectivityManager#CONNECTIVITY_ACTION) broadcast is delivered only to context-registered receivers.
* Do not broadcast sensitive information using an implicit intent. The information can be read by any app that registers to receive the broadcast. There are three ways to control who can receive your broadcasts:
  + You can specify a permission when sending a broadcast.
  + In Android 4.0 and higher, you can specify a [package](https://developer.android.com/guide/topics/manifest/manifest-element#package) with [setPackage(String)](https://developer.android.com/reference/android/content/Intent#setPackage(java.lang.String)) when sending a broadcast. The system restricts the broadcast to the set of apps that match the package.
  + You can send local broadcasts with [LocalBroadcastManager](https://developer.android.com/reference/androidx/localbroadcastmanager/content/LocalBroadcastManager).
* When you register a receiver, any app can send potentially malicious broadcasts to your app's receiver. There are three ways to limit the broadcasts that your app receives:
  + You can specify a permission when registering a broadcast receiver.
  + For manifest-declared receivers, you can set the [android:exported](https://developer.android.com/guide/topics/manifest/receiver-element#exported) attribute to "false" in the manifest. The receiver does not receive broadcasts from sources outside of the app.
  + You can limit yourself to only local broadcasts with [LocalBroadcastManager](https://developer.android.com/reference/androidx/localbroadcastmanager/content/LocalBroadcastManager).
* The namespace for broadcast actions is global. Make sure that action names and other strings are written in a namespace you own, or else you may inadvertently conflict with other apps.
* Because a receiver's [onReceive(Context, Intent)](https://developer.android.com/reference/android/content/BroadcastReceiver#onReceive(android.content.Context, android.content.Intent)) method runs on the main thread, it should execute and return quickly. If you need to perform long running work, be careful about spawning threads or starting background services because the system can kill the entire process after onReceive() returns. For more information, see [Effect on process state](https://developer.android.com/guide/components/broadcasts#effects-on-process-state) To perform long running work, we recommend:
  + Calling [goAsync()](https://developer.android.com/reference/android/content/BroadcastReceiver#goAsync()) in your receiver's onReceive() method and passing the [BroadcastReceiver.PendingResult](https://developer.android.com/reference/android/content/BroadcastReceiver.PendingResult) to a background thread. This keeps the broadcast active after returning from onReceive(). However, even with this approach the system expects you to finish with the broadcast very quickly (under 10 seconds). It does allow you to move work to another thread to avoid glitching the main thread.
  + Scheduling a job with the [JobScheduler](https://developer.android.com/reference/android/app/job/JobScheduler). For more information, see [Intelligent Job Scheduling](https://developer.android.com/topic/performance/scheduling.html).
* Do not start activities from broadcast receivers because the user experience is jarring; especially if there is more than one receiver. Instead, consider displaying a [notification](https://developer.android.com/develop/ui/views/notifications).

**SMS** stands for **Short Message Service**.  
It is basically a standardized communication protocol that is used for the exchange of text messages between mobile devices. It is one of the most widely used data applications. One can send SMS by using a smartphone, any other mobile phone, or a computer system equipped with the web, etc. The maximum length that a text message can have is 160 alpha-numeric characters. In the current scenario, the SMS facility is supported by GSM, CDMA, and TDMA mobile networks. The facility of SMS is provided by all the telecommunication companies like Vodafone, airtel, idea, etc.



#### History

The text message facility was first added to the mobile devices in the early 1980s. The action plan GSM for providing the services available to public data networks also to the mobile systems was approved in December 1982. According to this plan, the exchange of text messages was allowed either via mobile stations or via handling stations. Finally, the concept of SMS was thus developed in Franco-German cooperation by Friedhelm Hillebrand and Bernard Ghillebaert in 1984. The main idea behind the SMS concept was to use telephone optimized systems and to thus transport messages by making the use of signaling paths. The maximum length of text messages to be of 160 alpha-numeric characters was decided by Friedhelm Hillebrand.

#### Advantages

* It is a very reliable and attested platform for sending important information which ensures a guaranteed delivery of the information.
* It is a more discrete form of sending information in comparison to a phone call.
* It is provided automatically to all mobile devices and there is no requirement for an internet connection.
* The conversations among people are automatically stored until deleted by the people themselves.
* It does not require any app to be downloaded as it is already provided in all the mobile devices.

#### Disadvantages

* It has a limited length as it provides only 160 characters per text message.
* In order to send a text message one has to pay the cost for sending it as SMS facility is not free of cost.
* There is no possibility to unsend a message once sent.
* SMS Manager is a class in Android which is used to send the SMS to a specific contact from the android application. We can send text messages, data messages, and multimedia messages using this class. There are different methods that are provided to send different types of messages. In this article, we will take a look at How to implement SMS Manager in Android using Kotlin. A sample video is given below to get an idea about what we are going to do in this article.
* Video Player
* 00:00
* 00:28
* **Note**: If you are looking to use SMS manager in android application using JAVA. *Check out the following article:*[*Send SMS in Android using Java*](https://www.geeksforgeeks.org/sending-a-text-message-over-the-phone-using-smsmanager-in-android/)

### **Step by Step Implementation**

* **Step 1: Create a New Project in Android Studio**
* To create a new project in Android Studio please refer to [How to Create/Start a New Project in Android Studio](https://www.geeksforgeeks.org/android-how-to-create-start-a-new-project-in-android-studio/). Note that select **Kotlin** as the programming language.
* **Step 2: Working with the activity\_main.xml file**
* Navigate to the **app > res > layout > activity\_main.xml** and add the below code to that file. Below is the code for the **activity\_main.xml** file. Comments are added inside the code to understand the code in more detail.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout      xmlns:android="<http://schemas.android.com/apk/res/android>"      xmlns:tools="<http://schemas.android.com/tools>"      android:layout\_width="match\_parent"      android:layout\_height="match\_parent"      android:orientation="vertical"      tools:context=".MainActivity">        <!--on below line we are displaying a simple text view-->      <TextView          android:layout\_width="match\_parent"          android:layout\_height="wrap\_content"          android:layout\_gravity="center"          android:layout\_marginStart="10dp"          android:layout\_marginTop="30dp"          android:layout\_marginEnd="10dp"          android:padding="10dp"          android:text="SMS Manager App"          android:textAlignment="center"          android:textColor="@color/black"          android:textSize="20sp" />        <!--on below line we are creating a          simple edit text for phone number-->      <EditText          android:id="@+id/idEdtPhone"          android:layout\_width="match\_parent"          android:layout\_height="wrap\_content"          android:layout\_marginStart="10dp"          android:layout\_marginTop="20dp"          android:layout\_marginEnd="10dp"          android:hint="Enter phone number"          android:inputType="phone" />        <!--on below line we are creating a           message edit text for message-->      <EditText          android:id="@+id/idEdtMessage"          android:layout\_width="match\_parent"          android:layout\_height="wrap\_content"          android:layout\_marginStart="10dp"          android:layout\_marginTop="20dp"          android:layout\_marginEnd="10dp"          android:hint="Enter Message" />        <!--on below line we are creating          a button for sending message-->      <Button          android:id="@+id/idBtnSendMessage"          android:layout\_width="match\_parent"          android:layout\_height="wrap\_content"          android:layout\_marginStart="10dp"          android:layout\_marginTop="20dp"          android:layout\_marginEnd="10dp"          android:padding="5dp"          android:text="Send Message"          android:textAllCaps="false" />    </LinearLayout> |

* **Step 3: Working with the** **MainActivity.kt file**
* Go to the **MainActivity.kt** file and refer to the following code. Below is the code for the **MainActivity.kt** file. Comments are added inside the code to understand the code in more detail.

|  |
| --- |
| package com.gtappdevelopers.kotlingfgproject    import android.os.Bundle  import android.telephony.SmsManager  import android.widget.Button  import android.widget.EditText  import android.widget.Toast  import androidx.appcompat.app.AppCompatActivity    class MainActivity : AppCompatActivity() {        // on below line we are creating variable      // for edit text phone and message and button      lateinit var phoneEdt: EditText      lateinit var messageEdt: EditText      lateinit var sendMsgBtn: Button        override fun onCreate(savedInstanceState: Bundle?) {          super.onCreate(savedInstanceState)          setContentView(R.layout.activity\_main)            // initializing variables of phone edt,          // message edt and send message btn.          phoneEdt = findViewById(R.id.idEdtPhone)          messageEdt = findViewById(R.id.idEdtMessage)          sendMsgBtn = findViewById(R.id.idBtnSendMessage)            // adding on click listener for send message button.          sendMsgBtn.setOnClickListener {                // on below line we are creating two              // variables for phone and message              val phoneNumber = phoneEdt.text.toString()              val message = messageEdt.text.toString()                // on the below line we are creating a try and catch block              try {                    // on below line we are initializing sms manager.                  //as after android 10 the getDefault function no longer works                 //so we have to check that if our android version is greater                 //than or equal toandroid version 6.0 i.e SDK 23                  val smsManager:SmsManager                  if (Build.VERSION.SDK\_INT>=23) {                    //if SDK is greater that or equal to 23 then                    //this is how we will initialize the SmsManager                      smsManager = this.getSystemService(SmsManager::class.java)                  }                  else{                    //if user's SDK is less than 23 then                    //SmsManager will be initialized like this                      smsManager = SmsManager.getDefault()                  }                    // on below line we are sending text message.                  smsManager.sendTextMessage(phoneNumber, null, message, null, null)                    // on below line we are displaying a toast message for message send,                  Toast.makeText(applicationContext, "Message Sent", Toast.LENGTH\_LONG).show()                } catch (e: Exception) {                    // on catch block we are displaying toast message for error.                  Toast.makeText(applicationContext, "Please enter all the data.."+e.message.toString(), Toast.LENGTH\_LONG)                      .show()              }          }      }  } |

* **Step 4: Adding permission to send SMS in AndroidManifest.xml**
* Navigate to manifest > AndroidManifest.xml and add the below permissions in it.

|  |
| --- |
| <!--on below line we are adding sms permissions-->  <uses-permission android:name="android.permission.READ\_PHONE\_STATE" />  <uses-permission android:name="android.permission.SEND\_SMS" />  <dist:module dist:instant="true" /> |

* Now run your application to see the output of it.
* **Output:**
* Video Player
* 00:00
* 00:27