Android Permissions

Table of Contents

[Permission Types 3](#_Toc100927462)

[Install-Time Permissions 4](#_Toc100927463)

[Runtime Permissions 5](#_Toc100927464)

[Workflow of Permissions 6](#_Toc100927465)

[Workflow 7](#_Toc100927466)

[Code 9](#_Toc100927467)

[Best Practices 12](#_Toc100927468)

In an attempt to maintain security, Android applications are run in a **sandbox**. A sandbox is an isolated environment where programs are executed without affecting the system or platform. If the program wishes to use any data or features outside the sandbox, it must ask for permission.

By default, no application has permission to perform any operations that might adversely affect the system, other applications or the user. If the application wishes to access resources (e.g. camera, storage, network) or information (e.g. contacts), it must request permission to do so. The user reserves the right to decide whether to grant the permission or not.

The permissions that an application requires must be specified by the developer manually in the AndroidManifest.xml file.

<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/>  
<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"/>

XML

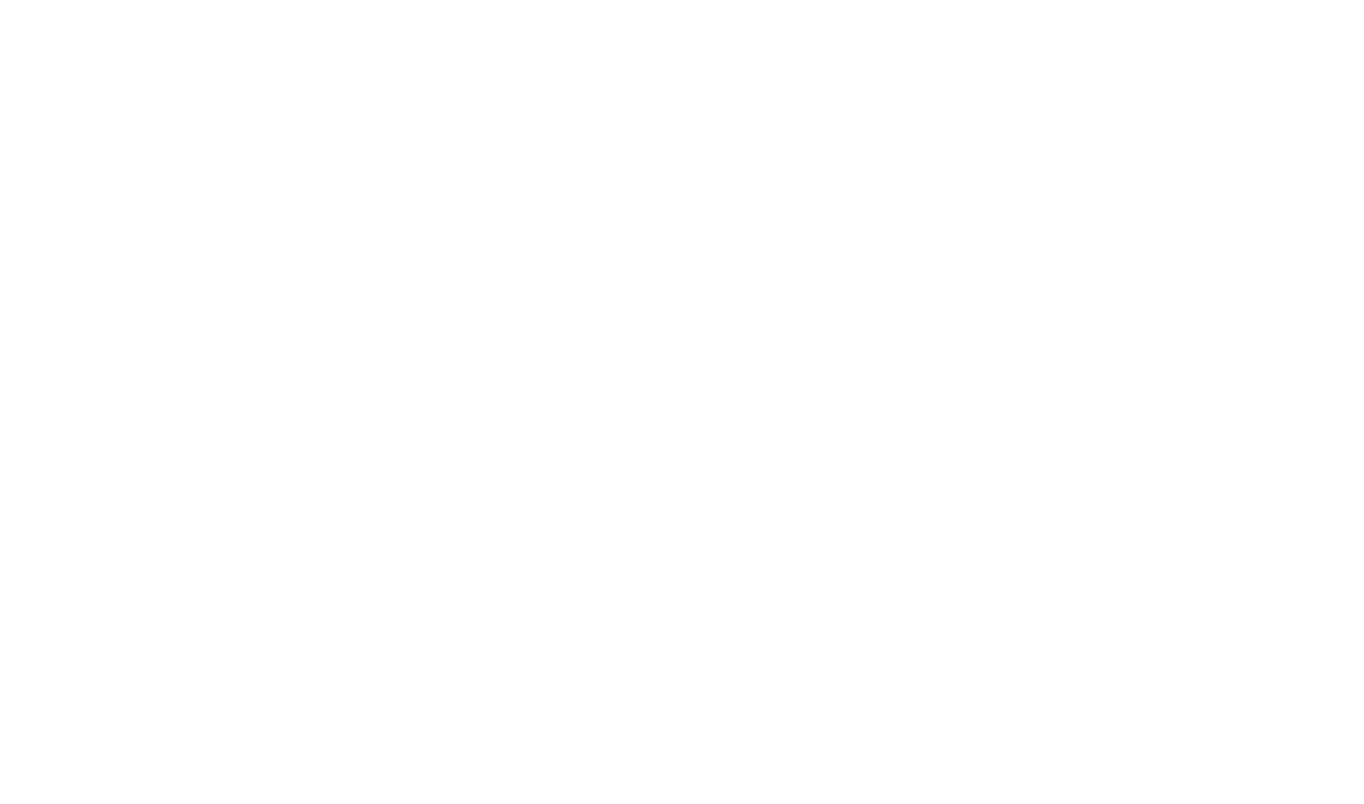
## Permission Types

Prior to Android 6.0 (API level 23), applications would have to request all the permissions it needed during **install time**. If the user refused to give the application the permissions it requested, it would simply not be installed.

If an installed application tried to perform an operation it did not have permission for (i.e. the permission had been revoked), it would **crash**.

The permissions given to an application can be viewed from Settings → Apps or using the adb command adb shell pm list permissions -s.

From Android 6.0 onwards, the way permissions are handled has changed. Permissions have been divided into three types, install-time permissions, runtime permissions and special permissions.



### Install-Time Permissions

**Install-time permissions** are requested when the application is first installed. These permissions are **normal permissions** which cannot adversely affect the system, other applications or the user. A few examples are given below:

* ACCESS\_LOCATION\_EXTRA\_COMMANDS
* ACCESS\_NETWORK\_STATE
* ACCESS\_NOTIFICATION\_POLICY
* ACCESS\_WIFI\_STATE
* BLUETOOTH
* BLUETOOTH\_ADMIN
* BROADCAST\_STICKY
* CHANGE\_NETWORK\_STATE
* CHANGE\_WIFI\_MULTICAST\_STATE
* CHANGE\_WIFI\_STATE
* DISABLE\_KEYGUARD
* EXPAND\_STATUS\_BAR
* GET\_PACKAGE\_SIZE
* INSTALL\_SHORTCUT
* INTERNET
* KILL\_BACKGROUND\_PROCESSES
* MODIFY\_AUDIO\_SETTINGS
* NFC
* READ\_SYNC\_SETTINGS
* READ\_SYNC\_STATS
* RECEIVE\_BOOT\_COMPLETED
* REORDER\_TASKS
* REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS
* REQUEST\_INSTALL\_PACKAGES
* SET\_ALARM
* SET\_TIME\_ZONE
* SET\_WALLPAPER
* SET\_WALLPAPER\_HINTS
* TRANSMIT\_IR
* UNINSTALL\_SHORTCUT
* USE\_FINGERPRINT
* VIBRATE
* WAKE\_LOCK
* WRITE\_SYNC\_SETTINGS

Since dangerous permissions are not requested at install-time, users have more confidence in installing the application. This was the primary reason behind dividing the permissions into groups.

### Runtime Permissions

**Runtime permissions** are requested at the time when the application requires the permission. These permissions are also called **dangerous permissions**, since they might adversely affect the system, other applications or the user. When an application requires one of these permissions, a dialog box is shown to the user requesting the permissions.

Some runtime permissions include:

* CALENDAR: READ\_CALENDAR, WRITE\_CALENDAR
* CAMERA: CAMERA
* CONTACTS: READ\_CONTACTS, WRITE\_CONTACTS, GET\_ACCOUNTS
* LOCATION: ACCESS\_FINE\_LOCATION, ACCESS\_COARSE\_LOCATION
* MICROPHONE: RECORD\_AUDIO
* PHONE: READ\_PHONE\_STATE, CALL\_PHONE, READ\_CALL\_LOG, WRITE\_CALL\_LOG, ADD\_VOICEMAIL, USE\_SIP PROCESS\_OUTGOING\_CALLS
* SENSORS: BODY\_SENSORS
* SMS: SEND\_SMS, RECEIVE\_SMS, READ\_SMS, RECEIVE\_WAP\_PUSH, RECEIVE\_MMS
* STORAGE: READ\_EXTERNAL\_STORAGE, WRITE\_EXTERNAL\_STORAGE

Notice that the permissions are divided into several groups. These are called **permission groups**. Permission groups are used to minimize the number of dialogs shown to the user. When a specific permission is requested by an application, the system shows a single dialog asking the user to grant permission for the entire group. Thus, instead of individually asking for 5 permissions, the system may simply ask if the user wishes to allow the application to access their SMS.

## Workflow of Permissions

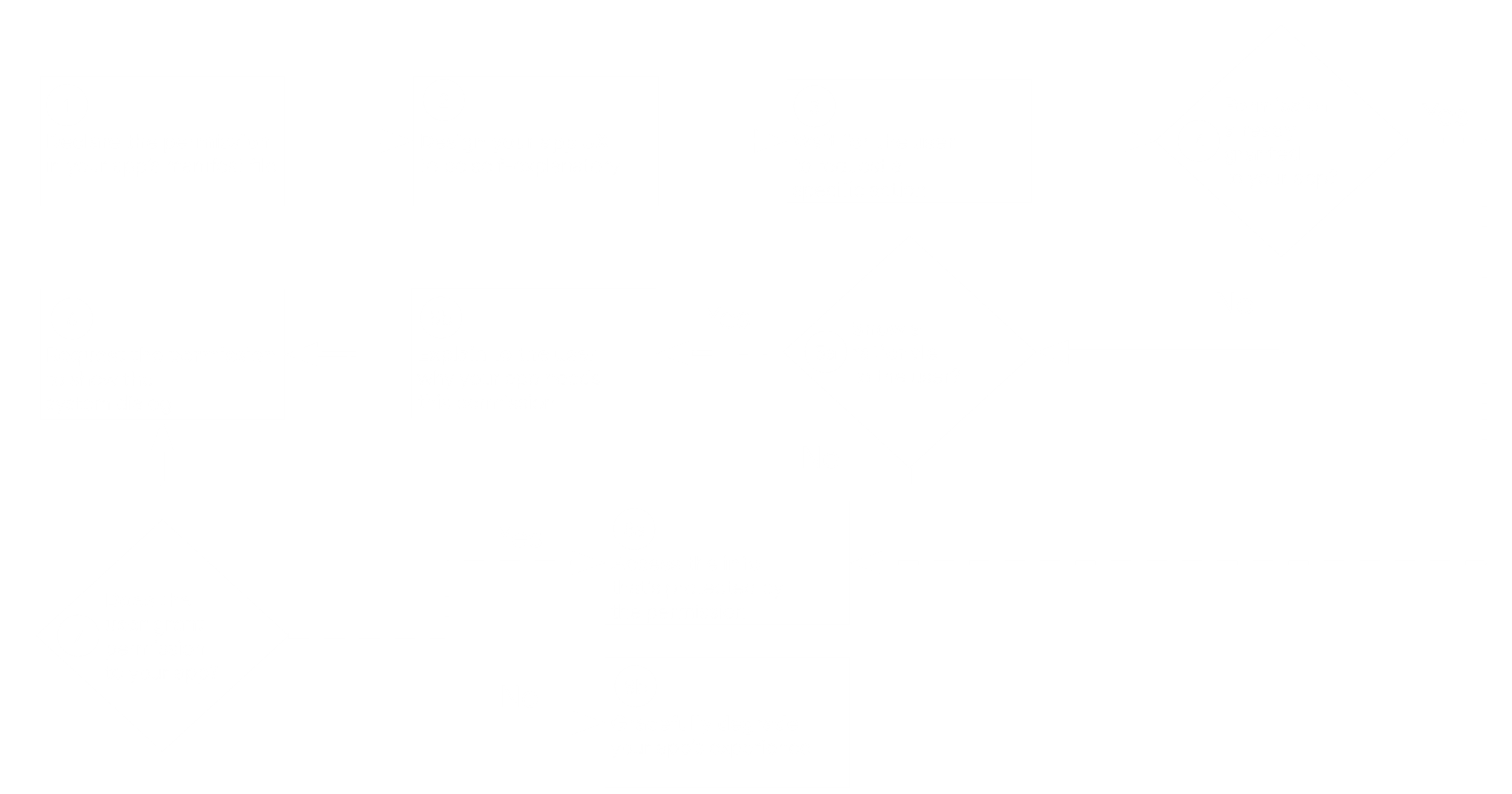
We need to decide whether or not we should ask for a permission. There are cases where we might not need a permission to achieve a goal.

For example, consider that we have an application that needs to access the user’s camera. This requires a permission. However, an alternative would be to use an **intent** to launch the built-in camera application. This would bypass the need for this specific permission.

Thus, we should only ask for a permission if there is no way to achieve our goal without the permission. If our only objective is to take a picture, we can use an intent to use the phones built-in camera application. However, if we plan on giving some **extra features** to the camera, such as filters, we need to use our own camera interface, which means we have to ask for permission. Applications such as Instagram do this.

If we do decide to use permissions, we first have to declare all the permissions in the AndroidManifest.xml file. Once this is done, we need to check if we are using any runtime permissions. If we are, we also need to write code to show the user a dialog asking for the permission at the moment we need the permissions.

## Workflow



The entire process of requesting permissions is provided in the diagram above.

1. The first step is to **declare** all the permissions our application will need in the AndroidManifest.xml file. This includes both normal permissions and runtime permissions.
2. We should **design** our application so that everything is very clear and self-explanatory. If our application requires access to the user’s camera, it should be very obvious that the application is doing this. We should not try to do anything immoral, such as tricking the user into giving us permission for something that they do not realize.
3. For dangerous permissions, we should **wait for the user** to request a specific action before asking for the permissions.
4. The rest of the steps detail how to handle requesting dangerous permissions. When the user performs an action that requires a dangerous permission, we first need to check if our application **already has the permission**. If it does, we can just go ahead with the functionality.
5. If this is the first time the user has tried to perform the action, a **system dialog** should be shown that requests the required permissions. This system dialog is standard and cannot be modified.
6. It is possible that the user has previously been requested this permission and has **denied** it. In that case, we should create a custom dialog to inform the user about **why** we need the permission. After this, the system dialog requesting the permission can be shown again.
7. If the user does not give us the required permission, we should **gracefully degrade** the application’s experience, i.e. we should prevent the user from accessing the feature without doing anything drastic like crashing the application.

## Code

// check if we already have the permissions  
int requestCode = 0;  
String[] requiredPermissions = {Manifest.permission.*ACCESS\_COARSE\_LOCATION*,  
 Manifest.permission.*ACCESS\_FINE\_LOCATION*};  
int permissionStatus = ContextCompat.*checkSelfPermission*(this,   
 requiredPermissions[0]);

if (permissionStatus == PackageManager.*PERMISSION\_GRANTED*) {  
 // start action  
}  
else if (ActivityCompat.*shouldShowRequestPermissionRationale*(this,  
 requiredPermissions[0])) {  
 // user has previously denied this permission  
 // show dialog with explanation  
 // this section MUST be asynchronous  
}  
else {  
 // this is the first time the user is trying this feature  
 // show system dialog requesting permission  
 ActivityCompat.*requestPermissions*(this, requiredPermissions,   
 requestCode);  
}

JAVA

In the above code, we have an array of permissions we require. The first thing we do is to check if we already have the permission we need. If we do, we can go forth normally, giving the user access to the feature.

The else-if block is somewhat interesting. *shouldShowRequestPermissionRationale* only returns true if the user has **previously denied** the required permission. In that case, we can do whatever we want to explain why we need the permission.

Finally, if the *shouldShowRequestPermissionRationale* method returns false, it means this is the first time the user is performing this action. In this case, we show the system dialog to the user asking for the permission.

The last stage is where things get a little tricky. The requestCode variable is defined by us to keep track of which permission we are requesting. When the system dialog is closed, a new method is automatically run, the one provided below. We need the request code in this method.

@Override  
public void onRequestPermissionsResult(int requestCode, String[] permissions,  
 int[] grantResults) {  
 switch (requestCode) {  
 case requestCode1:  
 if (grantResults.length > 0 &&  
 grantResults[0] == PackageManager.*PERMISSION\_GRANTED*) {  
 // user gave permission  
 }  
 else {  
 // user denied permission  
 }  
 return;  
 case requestCode2:  
 // more permission types  
 }  
}

JAVA

The onRequestPermissionsResult method is called whenever the system dialog is shown. Our app can ask for **multiple permissions**, with each permission showing a separate system dialog and thus running this method separately. This is why we need to check the result code, which will allow us to identify which specific permission caused the method to be triggered.

grantResults holds the user response for each of the permissions we asked for, i.e. requiredPermissions. We have to first check the **length** of grantResults. This is because granting and denying are not the only choices for the user. They can also cancel the process (by pressing the back button for example), which will cause grantResults to be empty.

The onRequestPermissionsResult method is very complicated. This is happening because we are handling the request codes by ourselves. Instead, we could allow the **system** to handle the process for us.

To do this, we first need to create the method which will handle the results of the user’s decision.

ActivityResultLauncher<String> requestPermissionLauncher =  
 registerForActivityResult(new ActivityResultContracts.RequestPermission(),   
 isGranted -> {  
 if (isGranted) { }  
 else { }  
 });

JAVA

Since this is a custom method which we are binding to a variable, we can have a different variable for each of the permissions we need to request. However, keep in mind that this method will only be **run once**, when the system dialog is shown requesting the user for permission.

Next, we call this method in the else block of the original code.

String[] requiredPermissions = {Manifest.permission.*ACCESS\_COARSE\_LOCATION*,  
 Manifest.permission.*ACCESS\_FINE\_LOCATION*};  
int permissionStatus = ContextCompat.*checkSelfPermission*(this,  
 requiredPermissions[0]);  
  
if (permissionStatus == PackageManager.*PERMISSION\_GRANTED*) {  
}  
else if (ActivityCompat.*shouldShowRequestPermissionRationale*(this,  
 requiredPermissions[0])) {  
}  
else {  
 requestPermissionLauncher.launch(requiredPermissions[0]);  
}

JAVA

## Best Practices

* Only use permissions absolutely necessary for the application.
* Pay attention to permissions required by libraries
* Be transparent about what permissions are being requested and why.
* Make it obvious when sensitive information is being accessed, e.g. by providing an indicator when the camera or microphone is being accessed. This will help gain the confidence of the user that the application is not trying to be secretive.

Additional information about best practices for permission usage can be found [here](https://developer.android.com/training/permissions/usage-notes).