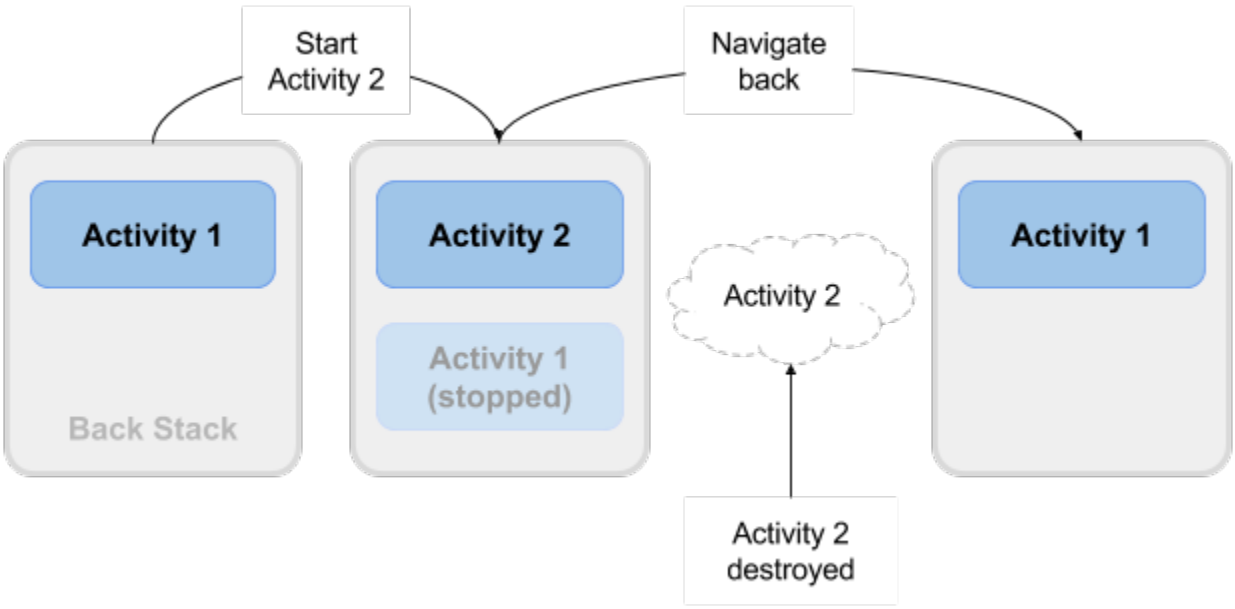
Unit-2

Topic-1 The Activity Lifecycle and Managing State

About the activity lifecycle

The activity lifecycle is the set of states an activity can be in during its entire lifetime, from the time it is initially created to when it is destroyed and the system reclaims that activity's resources. As the user interacts with your app and other apps on the device, the different activities move into different states.

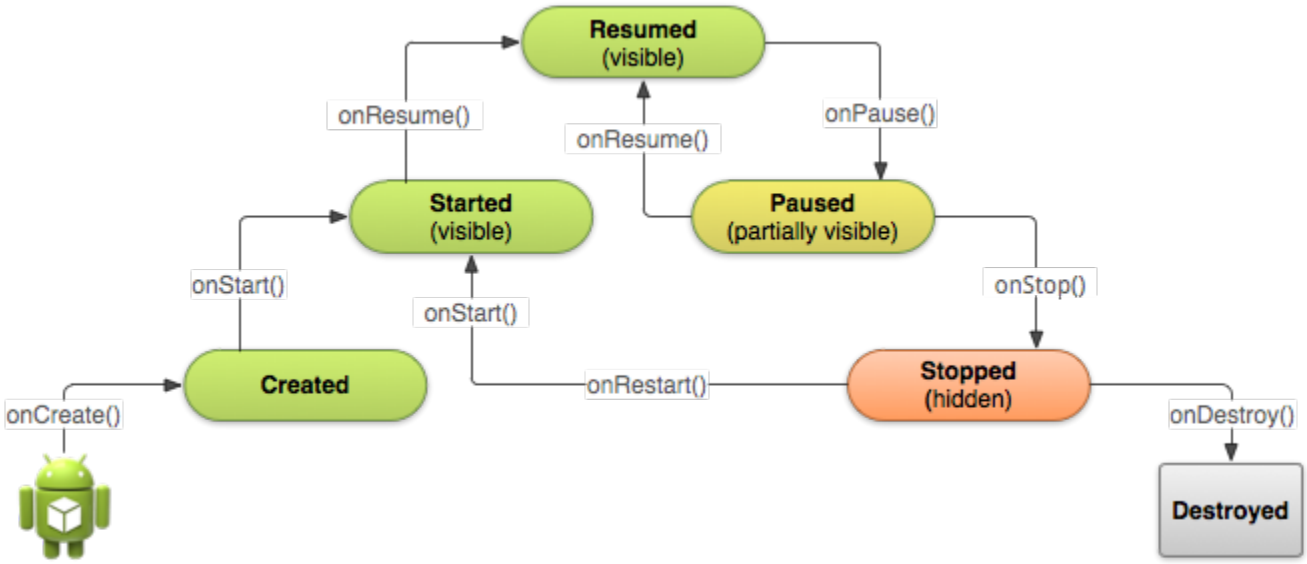
For example, when you start an app, the app's main activity (Activity 1) is started, comes to the foreground, and receives the user focus. When you start a second activity (Activity 2), that new activity is also created and started, and the main activity is stopped. When you're done with the second activity and navigate back, the first activity resumes. The second activity stops and is no longer needed; if the user does not resume the second activity, it is eventually destroyed by the system.



Activity states and lifecycle callback methods

When an activity transitions into and out of the different lifecycle states as it runs, the Android system calls several lifecycle callback methods at each stage. All of the callback methods are hooks that you can override in each of your Activity classes to define how that activity behaves when the user leaves and re-enters the activity. Keep in mind that the lifecycle states (and callbacks) are per activity, not per app, and you may implement different behavior at different points in the lifecycle for different activities in your app.

This figure shows each of the activity states and the callback methods that occur as the activity transitions between different states:



Depending on the complexity of your activity, you probably don't need to implement all the lifecycle callback methods in your activities. However, it's important that you understand each one and implement those that ensure your app behaves the way users expect. Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application.

Activity created (onCreate() method)

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   // The activity is being created.
}
```

Your activity enters into the created state when it is started for the first time. When an activity is first created the system calls the onCreate() method to initialize that activity. For example, when the user taps your app icon from the Home screen to start that app, the system calls the onCreate() method for the activity in your app that you've declared to be the "launcher" or "main" activity. In this case the main activity's onCreate() method is analogous to the main() method in other programs.

Similarly, if your app starts another activity with an intent (either explicit or implicit), the system matches your intent request with an activity and calls onCreate() for that new activity.

The onCreate() method is the only required callback you must implement in your activity class. In your onCreate() method you perform basic application startup logic that should happen only once, such as setting up the user interface, assigning class-scope variables, or setting up background tasks.

Created is a transient state; the activity remains in the created state only as long as it takes to run onCreate(), and then the activity moves to the started state.

Activity started (onStart() method)

```
@Override
protected void onStart() {
   super.onStart();
   // The activity is about to become visible.
}
```
After your activity is initialized with onCreate(), the system calls the onStart() method, and the activity is in the started state. The onStart() method is also called if a stopped activity returns to the foreground, such as when the user clicks the back or up buttons to navigate to the previous screen. While onCreate() is called only once when the activity is created, the onStart() method may be called many times during the lifecycle of the activity as the user navigates around your app.

When an activity is in the started state and visible on the screen, the user cannot interact with it until onResume() is called, the activity is running, and the activity is in the foreground.

Typically you implement onStart() in your activity as a counterpart to the onStop() method. For example, if you release hardware resources (such as GPS or sensors) when the activity is stopped, you can re-register those resources in the onStart() method.

Started, like created, is a transient state. After starting the activity moves into the resumed (running) state.

Activity resumed/running (onResume() method)

```
@Override
protected void onResume() {
   super.onResume();
   // The activity has become visible (it is now "resumed").
}
```
Your activity is in the resumed state when it is initialized, visible on screen, and ready to use. The resumed state is often called the running state, because it is in this state that the user is actually interacting with your app.

The first time the activity is started the system calls the onResume() method just after onStart(). The onResume() method may also be called multiple times, each time the app comes back from the paused state.

As with the onStart() and onStop() methods, which are implemented in pairs, you typically only implement onResume() as a counterpart to onPause(). For example, if in the onPause() method you halt any onscreen animations, you would start those animations again in onResume().

The activity remains in the resumed state as long as the activity is in the foreground and the user is interacting with it. From the resumed state the activity can move into the paused state.

Activity paused (onPause() method)

```
@Override
protected void onPause() {
```

```
    super.onPause();
    // Another activity is taking focus
    // (this activity is about to be "paused").
}
```
The paused state can occur in several situations:

- The activity is going into the background, but has not yet been fully stopped. This is the first indication that the user is leaving your activity.
- The activity is only partially visible on the screen, because a dialog or other transparent activity is overlaid on top of it.
- In multi-window or split screen mode (API 24), the activity is displayed on the screen, but some other activity has the user focus.

The system calls the onPause() method when the activity moves into the paused state. Because the onPause() method is the first indication you get that the user may be leaving the activity, you can use onPause() to stop animation or video playback, release any hardware-intensive resources, or commit unsaved activity changes (such as a draft email).

The onPause() method should execute quickly. Don't use onPause() for for CPU-intensive operations such as writing persistent data to a database. The app may still be visible on screen as it passed through the paused state, and any delays in executing onPause() can slow the user's transition to the next activity. Implement any heavy-load operations when the app is in the stopped state instead.

Note that in multi-window mode (API 24), your paused activity may still fully visible on the screen. In this case you do not want to pause animations or video playback as you would for a partially visible activity. You can use the inMultiWindowMode() method in the Activity class to test whether your app is running in multiwindow mode.

Your activity can move from the paused state into the resumed state (if the user returns to the activity) or to the stopped state (if the user leaves the activity altogether).

Activity stopped (onStop() method)

```
@Override
protected void onStop() {
    super.onStop();
    // The activity is no longer visible (it is now "stopped")
}
```
An activity is in the stopped state when it is no longer visible on the screen at all. This is usually because the user has started another activity, or returned to the home screen. The system retains the activity instance in the back stack, and if the user returns to that activity it is restarted again. Stopped activities may be killed altogether by the Android system if resources are low.

The system calls the onStop() method when the activity stops. Implement the onStop() method to save any persistent data and release any remaining resources you did not already release in onPause(), including those operations that may have

been too heavyweight for onPause().

Activity destroyed (onDestroy() method)

```
@Override
protected void onDestroy() {
   super.onDestroy();
   // The activity is about to be destroyed.
}
```

When your activity is destroyed it is shut down completely, and the Activity instance is reclaimed by the system. This can happen in several cases:

- You call finish() in your activity to manually shut it down.
- The user navigates back to the previous activity.
- The device is in a low memory situation where the system reclaims stopped activities to free more resources.
- A device configuration change occurs. You'll learn more about configuration changes later in this chapter.

Use onDestroy() to fully clean up after your activity so that no component (such as a thread) is running after the activity is destroyed.

Note that there are situations where the system will simply kill the activity's hosting process without calling this method (or any others), so you should not rely on onDestroy() to save any required data or activity state. Use onPause() or onStop() instead.

Activity restarted (onRestart() method)

```
@Override
protected void onRestart() {
   super.onRestart();
   // The activity is about to be restarted.
}
```

The restarted state is a transient state that only occurs if a stopped activity is started again. In this case the onRestart() method is called in between onStop() and onStart(). If you have resources that need to be stopped or started you typically implement that behavior in onStop() or onStart() rather than onRestart().

Configuration changes and activity state

Earlier in the section onDestroy() you learned that your activities may be destroyed when the user navigates back, by you with the finish() method, or by the system when it needs to free resources. The fourth time your activities are destroyed is when the device undergoes a *configuration change*.

Configuration changes occur on the device, in runtime, and invalidate the current layout or other resources in your activity The most common form of a configuration change is when the device is rotated. When the device rotates from portrait to landscape, or vice versa, the layout for your app also needs to change. The system recreates the activity to help that activity adapt to the new

configuration by loading alternative resources (such as a landscape-specific layout).

Other configuration changes can include a change in locale (the user chooses a different system language), or the user enters multi-window mode (Android 7). In multi-window mode, if you have configured your app to be resizeable, Android recreates your activities to use a layout definition for the new, smaller activity size.

When a configuration change occurs Android system shuts down your activity (calling onPause(), onStop(), and onDestroy()), and then starts it over again from the start (calling onCreate(), onStart(), and onResume()).

Activity instance state

When an activity is destroyed and recreated, there are implications for the runtime state of that activity. When an activity is paused or stopped, the state of the activity is retained because that activity is still held in memory. When an activity is recreated, the state of the activity and any user progress in that activity is lost, with these exceptions:

- Some activity state information is automatically saved by default. The state of views in your layout with a unique ID (as defined by the android:id attribute in the layout) are saved and restored when an activity is recreated. In this case, the user-entered values in EditText views are usually retained when the activity is recreated.
- The intent that was used to start the activity, and the information stored in that intent's data or extras, remains available to that activity when it is recreated.

The activity state is stored as a set of key/value pairs in a Bundle object called the *activity instance state*. The system saves default state information to instance state bundle just before the activity is stopped, and passes that bundle to the new activity instance to restore.

You can add your own instance data to the instance state bundle by overriding the onSaveInstanceState() callback. The state bundle is passed to the onCreate() method, so you can restore that instance state data when your activity is created. There is also a corresponding onRestoreInstanceState() callback you can use to restore the state data.

Because device rotation is a common use case for you app, make sure you test that your activity behaves correctly in response to this configuration change, and implement instance state if you need to.

Note: The activity instance state is particular to a specific instance of an activity, running in a single task. If the user force-quits the app, reboots the device, or if the Android system shuts down the entire app process to preserve memory, the activity instance state is lost. To keep state changes across app instances and device reboots, you need to write that data to shared preferences. You'll learn more about shared preferences in a later chapter.

Saving activity instance state

To save information to the instance state bundle, use the onSaveInstanceState()

callback. This is not a lifecycle callback method, but it is called when the user is leaving your activity (sometime before the onStop() method).

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);
    // save your state data to the instance state bundle
}
```

The onSaveInstanceState() method is passed a Bundle object (a collection of key/value pairs) when it is called. This is the instance state bundle to which you will add your own activity state information.

You learned about bundles in a previous chapter when you added keys and values to the intent extras. Add information to the instance state bundle in the same way, with keys you define and the various "put" methods defined in the Bundle class:

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    super.onSaveInstanceState(savedInstanceState);

    // Save the user's current game state
    savedInstanceState.putInt("score", mCurrentScore);
    savedInstanceState.putInt("level", mCurrentLevel);
}
```

Don't forget to call through to the superclass, to make sure the state of the view hierarchy is also saved to the bundle.

Restoring activity instance state

Once you've saved the activity instance state, you also need to restore it when the activity is recreated. You can do this one of two places:

- The onCreate() callback method, which is called with the instance state bundle when the activity is created.
- The onRestoreInstanceState() callback, which is called after onStart() after the activity is created.

Most of the time the better place to restore the activity state is in onCreate(), to ensure that your user interface including the state is available as soon as possible.

To restore the saved instances state in onCreate(), test for the existence of a state bundle before you try to get data out of it. When your activity is started for the first time there will be no state and the bundle will be null.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt("score");
```

```
        mCurrentLevel = savedInstanceState.getInt("level");
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Navigating between activities

An app is likely to enter and exit an activity, perhaps many times, during the app's lifetime. For example, the user may tap the device's Back button, or the activity may need to launch a different activity. This section covers topics you need to know to implement successful activity transitions. These topics include starting an activity from another activity, saving activity state, and restoring activity state.

Starting one activity from another

An activity often needs to start another activity at some point. This need arises, for instance, when an app needs to move from the current screen to a new one.

Depending on whether your activity wants a result back from the new activity it's about to start, you start the new activity using either the startActivity() or the startActivityForResult() method. In either case, you pass in an Intent object.

The Intent object specifies either the exact activity you want to start or describes the type of action you want to perform (and the system selects the appropriate activity for you, which can even be from a different application). An Intent object can also carry small amounts of data to be used by the activity that is started. For more information about the Intent class, see Intents and Intent Filters.

startActivity()

If the newly started activity does not need to return a result, the current activity can start it by calling thestartActivity() method.

When working within your own application, you often need to simply launch a known activity. For example, the following code snippet shows how to launch an activity called SignInActivity.

KOTLINJAVA

```
val intent = Intent(this, SignInActivity::class.java)
startActivity(intent)
```

Your application might also want to perform some action, such as send an email, text message, or status update, using data from your activity. In this case, your application might not have its own activities to perform such actions, so you can

instead leverage the activities provided by other applications on the device, which can perform the actions for you. This is where intents are really valuable: You can create an intent that describes an action you want to perform and the system launches the appropriate activity from another application. If there are multiple activities that can handle the intent, then the user can select which one to use. For example, if you want to allow the user to send an email message, you can create the following intent:

KOTLINJAVA

```
val intent = Intent(Intent.ACTION_SEND).apply {
    putExtra(Intent.EXTRA_EMAIL, recipientArray)
}
startActivity(intent)
```

The EXTRA_EMAIL extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, your activity resumes.

startActivityForResult()

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) method, where the integer parameter identifies the call. This identifier is meant to disambiguate between multiple calls to startActivityForResult(Intent, int)from the same activity. It's not global identifier and is not at risk of conflicting with other apps or activities.The result comes back through your onActivityResult(int, int, Intent) method.

When a child activity exits, it can call setResult(int) to return data to its parent. The child activity 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, the child activity can optionally return an Intent object containing any additional data it wants. The parent activity uses the onActivityResult(int, int, Intent) method, along with the integer identifier the parent activity originally supplied, to receive the information.

If a child activity fails for any reason, such as crashing, the parent activity receives a result with the code RESULT_CANCELED.

KOTLINJAVA

```
class MyActivity : Activity() {
    // ...

    override fun onKeyDown(keyCode: Int, event: KeyEvent?): Boolean {
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
            // When the user center presses, let them pick a contact.
```

```
            startActivityForResult(
                Intent(Intent.ACTION_PICK,Uri.parse("content://contacts")),
                PICK_CONTACT_REQUEST)
            return true
        }
        return false
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, intent: Intent?) {
        when (requestCode) {
            PICK_CONTACT_REQUEST ->
                if (resultCode == RESULT_OK) {
                    startActivity(Intent(Intent.ACTION_VIEW, intent?.data))
                }
        }
    }

    companion object {
        internal val PICK_CONTACT_REQUEST = 0
    }
}
```

Coordinating activities

When one activity starts another, they both experience lifecycle transitions. The first activity stops operating and enters the Paused or Stopped state, while the other activity is created. In case these activities share data saved to disc or elsewhere, it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process (app) and one is starting the other. Here's the order of operations that occur when Activity A starts Activity B:

1. Activity A's onPause() method executes.
2. Activity B's onCreate(), onStart(), and onResume() methods execute in sequence. (Activity B now has user focus.)
3. Then, if Activity A is no longer visible on screen, its onStop() method executes.

This predictable sequence of lifecycle callbacks allows you to manage the transition of information from one activity to another.

Handle Activity State Changes

Different events, some user-triggered and some system-triggered, can cause an Activity to transition from one state to another. This document describes some common cases in which such transitions happen, and how to handle those

transitions.

For more information about activity states, see Understand the Activity Lifecycle. To learn about how the ViewModel class can help you manage the activity lifecycle, see Understand the ViewModel Class.

Configuration change occurs

There are a number of events that can trigger a configuration change. Perhaps the most prominent example is a change between portrait and landscape orientations. Other cases that can cause configuration changes include changes to language or input device.

When a configuration change occurs, the activity is destroyed and recreated. The original activity instance will have the onPause(), onStop(), and onDestroy() callbacks triggered. A new instance of the activity will be created and have the onCreate(), onStart(), and onResume() callbacks triggered.

Use a combination of ViewModels, the onSaveInstanceState() method, and/or persistent local storage to preserve an activity's UI state across configuration changes. Deciding how to combine these options depends on the complexity of your UI data, use cases for your app, and consideration of speed of retrieval versus memory usage. For more information on saving your activity UI state, see Saving UI States.

Handling multi-window cases

When an app enters multi-window mode, available in Android 7.0 (API level 24)and higher, the system notifies the currently running activity of a configuration change, thus going through the lifecycle transitions described above. This behavior also occurs if an app already in multi-window mode gets resized. Your activity can handle the configuration change itself, or it can allow the system to destroy the activity and recreate it with the new dimensions.

For more information about the multi-window lifecycle, see the Multi-Window Lifecycle section of the Multi-Window Support page.

In multi-window mode, although there are two apps that are visible to the user, only the one with which the user is interacting is in the foreground and has focus. That activity is in the Resumed state, while the app in the other window is in the Paused state.

When the user switches from app A to app B, the system calls onPause() on app A, and onResume() on app B. It switches between these two methods each time the user toggles between apps.

For more detail about multi-windowing, refer to Multi-Window Support.

Topic-2 Thread

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of

the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
         . . .
    }
}
```

The following code would then create a thread and start it running:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.**See also:**

- Runnable
- Runtime.exit(int)
- run()
- stop()

Constants

MAX_PRIORITY

Added in API level 1

public static final int MAX_PRIORITY

The maximum priority that a thread can have.

Constant Value: 10 (0x0000000a)

MIN_PRIORITY

Added in API level 1

public static final int MIN_PRIORITY

The minimum priority that a thread can have.

Constant Value: 1 (0x00000001)

NORM_PRIORITY

Added in API level 1

public static final int NORM_PRIORITY

The default priority that is assigned to a thread.

Constant Value: 5 (0x00000005)

Public constructors

Thread

Added in API level 1

public Thread ()

Allocates a new Thread object. This constructor has the same effect as Thread (null, null, gname), where gname is a newly generated name. Automatically generated names are of the form "Thread-"+*n*, where *n* is an integer.

Thread

Added in API level 1

public Thread (Runnable target)

Allocates a new Thread object. This constructor has the same effect as Thread (null, target, gname), where gname is a newly generated name. Automatically generated

names are of the form "Thread-"+*n*, where *n* is an integer.

Parameters	
target	Runnable: the object whose run method is invoked when this thread is started. If null, this classes run method does nothing.

Thread

Added in API level 1

public Thread (ThreadGroup group,

Runnable target)

Allocates a new Thread object. This constructor has the same effect as Thread (group, target, gname),where gname is a newly generated name. Automatically generated names are of the form "Thread-"+*n*, where *n* is an integer.

Parameters	
group	ThreadGroup: the thread group. If null and there is a security manager, the group determined by SecurityManager#getThreadGroup. If there is not a security mana or SecurityManager.getThreadGroup() returns null, the group is set to the curren thread group.
target	Runnable: the object whose run method is invoked when this thread is started. I thread's run method is invoked.
Throws	
SecurityException	if the current thread cannot create a thread in the specified thread group

Thread

Added in API level 1

public Thread (String name)

Allocates a new Thread object. This constructor has the same effect as Thread (null, null, name).

Parameters

name	String: the name of the new thread

Thread

Added in API level 1

public Thread (ThreadGroup group,

String name)

Allocates a new Thread object. This constructor has the same effect as Thread (group, null, name).

Parameters	
group	ThreadGroup: the thread group. If null and there is a security manager, the group determined by SecurityManager#getThreadGroup. If there is not a security mana or SecurityManager.getThreadGroup() returns null, the group is set to the curren thread group.
name	String: the name of the new thread
Throws	
SecurityException	if the current thread cannot create a thread in the specified thread group

Thread

Added in API level 1

public Thread (Runnable target,

String name)

Allocates a new Thread object. This constructor has the same effect as Thread (null, target, name).

Parameters	
target	Runnable: the object whose run method is invoked when this thread is started. If null, this th method is invoked.
nam	String: the name of the new thread

e

Thread

Added in API level 1

public Thread (ThreadGroup group,

Runnable target,

String name)

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

If there is a security manager,
its SecurityManager#checkAccess(ThreadGroup) method is invoked with the ThreadGroup as its argument.

In addition, its checkPermission method is invoked with
the RuntimePermission("enableContextClassLoaderOverride") permission when invoked directly or indirectly by the constructor of a subclass which overrides the getContextClassLoader or setContextClassLoadermethods.

The priority of the newly created thread is set equal to the priority of the thread creating it, that is, the currently running thread. The method setPriority may be used to change the priority to a new value.

The newly created thread is initially marked as being a daemon thread if and only if the thread creating it is currently marked as a daemon thread. The
method setDaemon may be used to change whether or not a thread is a daemon.

Parameters	
group	ThreadGroup: the thread group. If null and there is a security manager, the group determined by SecurityManager#getThreadGroup. If there is not a security mana or SecurityManager.getThreadGroup() returns null, the group is set to the curren thread group.
target	Runnable: the object whose run method is invoked when this thread is started. I thread's run method is invoked.
name	String: the name of the new thread
Throws	
SecurityExcepti	if the current thread cannot create a thread in the specified thread group or cann

on	override the context class loader methods.

Thread

Added in API level 1

public Thread (ThreadGroup group,

Runnable target,

String name,

long stackSize)

Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*.

This constructor is identical to Thread(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String) with the exception of the fact that it allows the thread stack size to be specified. The stack size is the approximate number of bytes of address space that the virtual machine is to allocate for this thread's stack. **The effect of the stackSize parameter, if any, is highly platform dependent.**

On some platforms, specifying a higher value for the stackSize parameter may allow a thread to achieve greater recursion depth before throwing a StackOverflowError. Similarly, specifying a lower value may allow a greater number of threads to exist concurrently without throwing an OutOfMemoryError (or other internal error). The details of the relationship between the value of the stackSize parameter and the maximum recursion depth and concurrency level are platform-dependent. **On some platforms, the value of thestackSize parameter may have no effect whatsoever.**

The virtual machine is free to treat the stackSize parameter as a suggestion. If the specified value is unreasonably low for the platform, the virtual machine may instead use some platform-specific minimum value; if the specified value is unreasonably high, the virtual machine may instead use some platform-specific maximum. Likewise, the virtual machine is free to round the specified value up or down as it sees fit (or to ignore it completely).

Specifying a value of zero for the stackSize parameter will cause this constructor to behave exactly like theThread(ThreadGroup, Runnable, String) constructor.

Due to the platform-dependent nature of the behavior of this constructor, extreme care should be exercised in its use. The thread stack size necessary to perform a given computation will likely vary from one JRE implementation to another. In light of this variation, careful tuning of the stack size parameter may be required, and the tuning may need to be repeated for each JRE implementation on which an application is to run.

Implementation note: Java platform implementers are encouraged to document their

implementation's behavior with respect to the stackSize parameter.

Parameters	
group	ThreadGroup: the thread group. If null and there is a security manager, the group determined by SecurityManager#getThreadGroup. If there is not a security mana or SecurityManager.getThreadGroup() returns null, the group is set to the curren thread group.
target	Runnable: the object whose run method is invoked when this thread is started. I thread's run method is invoked.
name	String: the name of the new thread
stackSize	long: the desired stack size for the new thread, or zero to indicate that this para be ignored.
Throws	
SecurityException	if the current thread cannot create a thread in the specified thread group

Topic-3 AsyncTask

public abstract class AsyncTask
extends Object
java.lang.Object
↳ android.os.AsyncTask<Params, Progress, Result>

AsyncTask enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

Developer Guides

For more information about using tasks and threads, read the Processes and Threads developer guide.

Usage

AsyncTask must be subclassed to be used. The subclass will override at least one method (doInBackground(Params...)), and most often will override a second one (onPostExecute(Result).)

Here is an example of subclassing:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }

    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

Once created, a task is executed very simply:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

AsyncTask's generic types

The three types used by an asynchronous task are the following:

1. Params, the type of the parameters sent to the task upon execution.
2. Progress, the type of the progress units published during the background computation.
3. Result, the type of the result of the background computation.

   Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type Void:

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```

The 4 steps

When an asynchronous task is executed, the task goes through 4 steps:

1. onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. doInBackground(Params...), invoked on the background thread immediately after onPreExecute()finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step.
3. onProgressUpdate(Progress...), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
4. onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Cancelling a task

A task can be cancelled at any time by invoking cancel(boolean). Invoking this method will cause subsequent calls to isCancelled() to return true. After invoking this method, onCancelled(java.lang.Object), instead of onPostExecute(java.lang.Object) will be invoked after doInBackground(java.lang.Object[]) returns. To ensure that a task is cancelled as quickly as possible, you should always check the return value of isCancelled() periodically fromdoInBackground(java.lang.Object[]), if possible (inside a loop for instance.)

Threading rules

There are a few threading rules that must be followed for this class to work properly:

- The AsyncTask class must be loaded on the UI thread. This is done automatically as of Build.VERSION_CODES.JELLY_BEAN.
- The task instance must be created on the UI thread.
- execute(Params...) must be invoked on the UI thread.
- Do not call onPreExecute(), onPostExecute(Result), doInBackground(Params...), onProgressUpdate(Progress...) manually.
- The task can be executed only once (an exception will be thrown if a second execution is attempted.)

Memory observability

AsyncTask guarantees that all callback calls are synchronized to ensure the following without explicit synchronizations.

- The memory effects of onPreExecute(), and anything else executed before the call to execute(Params...), including the construction of the AsyncTask object, are visible to doInBackground(Params...).
- The memory effects of doInBackground(Params...) are visible to onPostExecute(Result).
- Any memory effects of doInBackground(Params...) preceding a call to publishProgress(Progress...) are visible to the corresponding onProgressUpdate(Progress...)call. (But doInBackground(Params...) continues to run, and care needs to be taken that later updates in doInBackground(Params...) do not interfere with an in-progress onProgressUpdate(Progress...) call.)
- Any memory effects preceding a call to cancel(boolean) are visible after a call to isCancelled() that returns true as a result, or during and after a resulting call to onCancelled().

Order of execution

When first introduced, AsyncTasks were executed serially on a single background thread. Starting with Build.VERSION_CODES.DONUT, this was changed to a pool of threads allowing multiple tasks to operate in parallel. Starting with Build.VERSION_CODES.HONEYCOMB, tasks are executed on a single thread to avoid common application errors caused by parallel execution.

If you truly want parallel execution, you can invoke executeOnExecutor(java.util.concurrent.Executor, java.lang.Object[]) with THREAD_POOL_EXECUTOR.

Topic-4 Android Service-LifeCycle and Working | With Example

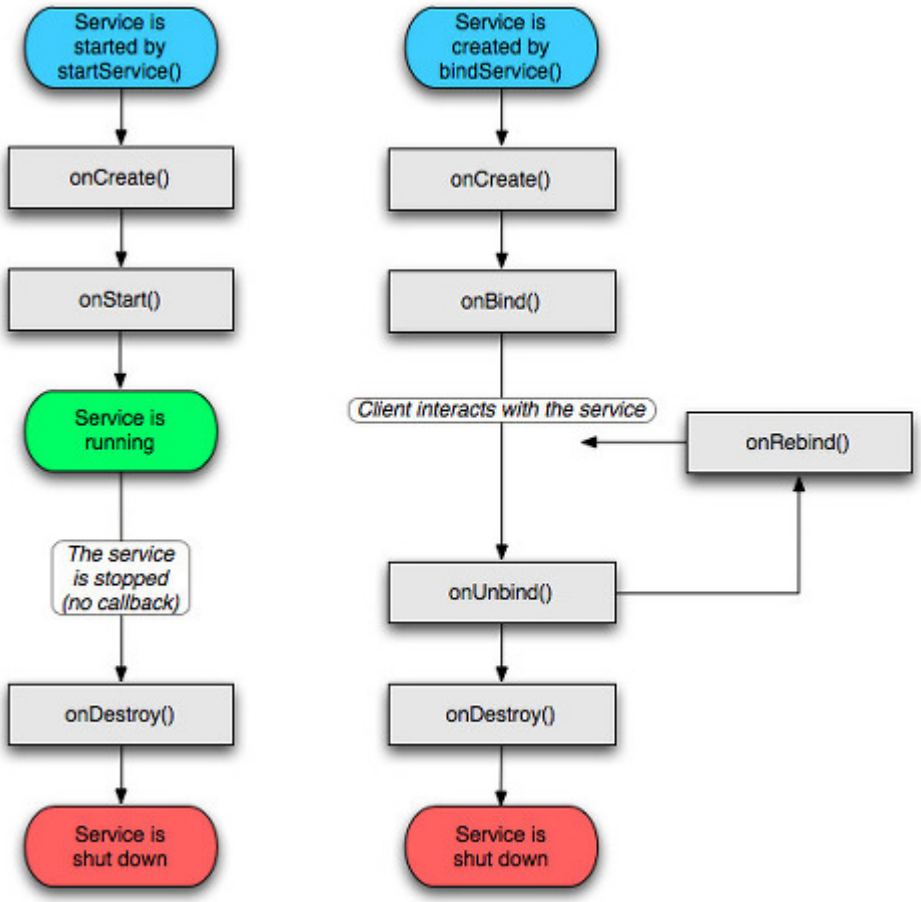
A Service is an application component that can perform long-running operations in the background, and it does not provide a user interface. Another application component can start a service, and it continues to run in the background even if the user switches to another application. Additionally, a component 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.

- Service is an Android Component without an UI
- It is used to perform long running operations in background. Services run indefinitly unless they are explicitly stopped or destroyed
- It can be started by any other application component. Components can even infact bind to a service to perform Interprocess- Comminication
- It can still be running even if the application is killed unless it stops itself by calling *stopself*() or is stopped by a Android component by calling *stopService*().
- If not stopped it goes on running unless is terminated by Android due to resource shortage
- The android.app.Service is subclass of ContextWrapper class.

Note: Service always runs on the main thread by default. When the doc says “long running processes in background” it means that processes without an UI. So if you are performing a time consuming task you must be creating a background thread in the service.

As explained above service can either be started or bound. You just need to call either *startService*() or *bindService*() from any of your android components. Based on how your service was started it will either be “started” or “bound”

- Started
  A service is **started** when an application component, such as an activity, starts it by calling *startService()*. Now the 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).



Android Service Lifecycle

Like any other components service also has callback methods. These will be invoked while the service is running to inform the application of its state. Implementing these in your custom service would help you in performing the right operation in the right state.

There is always only a single instance of service running in the app. If you are calling startService() for a single service multiple times in your application it just invokes the onStartCommand() on that service. Neither is the service restarted multiple times nor are its multiple instances created

onCreate()

This is the first callback which will be invoked when any component starts the service. If the same service is called again while it is still running this method wont be invoked. Ideally one time setup and intializing should be done in this callback.

onStartCommand()

This callback is invoked when service is started by any component by calling

startService(). It basically indicates that the service has started and can now run indefinetly. Now its your responsibilty to stop the service. This callback also has a int return type. This return type describes what happens to the service once it is destroyed by the system.There are three possible return types

- START_STICKY
  Returning this indicates that once the service is killed by the system it will be recreated and onStartCommand method will be invoked again.But the previous intent is not redelivered. Instead *onStartCommand* is called with a null intent
- START_NOT_STICKY
  Returning this indicates that the service wont be recreated if it is killed by the system
- START_REDELIVER_INTENT
  Returning this indicates that once the service is killed by the system it will be recreated and *onStartCommand* method will be invoked again.But here the original intent is redelivered again.

onBind()

This is invoked when any component starts the service by calling onBind. Basically the component has now binded with the service. This method needs to return your implementation of IBinder which will be used for Interprocess Communication. If you dont want your service to bind with any component you should just return null nevertheless this method needs to be implemented

onUnbind()

This is invoked when all the clients are disconnected from the service.

onRebind()

This is invoked when new clients are connected to the service. It is called after onUnbind

onDestroy()

This is a final clean up call from the system. This is invoked just before the service is being destroyed. Could be very useful to cleanup any resources such as threads, registered listeners, or receivers. But very rarely it happens that the service is destroyed with onDestroy not being invoked

Example :

In this example we will create a service with runs in background and prints logs until

we stop it.

Step-1- Create a Custom Service

As we have already discussed Service doesn't have a its own layout hence no xml file is required.

- Right click on your package and select New->Java Class. Name your class as MyService. This should extend the Android's Service class.
- We will be using a Handler and Runnable implemenation to print the logs every 5 seconds as long as the service is running

Then implement all the callbacks as shown in the code snippet below

```
1.  public class MyService extends Service {
2.
3.  //Declaring the handler
4.  private Handler handler;
5.  //Declaring your implementation of Runnable
6.  private Runner runner;
7.  /*
8.  Regardless of whether you want your service to be binded
9.  or not you should always implement onBind. You should return null if you
10. dont want it to bind
11. */
12. @Nullable
13. @Override
14. public IBinder onBind(Intent intent) {
15. return null;
16. }
17. /*
18. Initialization of Handler and Runner
19. */
20. public void onCreate() {
21. super.onCreate();
22.
23. Toast.makeText(this, "Service Started", Toast.LENGTH_LONG).show();
24. handler = new Handler();
25. runner = new Runner();
26.
27. }
28. // Starting the Runnable with handler
29. public int onStartCommand(Intent intent, int id, int startID) {
30. handler.post(runner);
31. return START_STICKY;
32. }
33. //Removing the callbacks from handler once the service is destroyed
```

```
@Override
public void onDestroy() {
super.onDestroy();
handler.removeCallbacks(runner);
Toast.makeText(this, "Service Destroyed", Toast.LENGTH_LONG).show();
}

/*
A runnable class designed in such a way that it is runs every 5 seconds
*/
public class Runner implements Runnable {

@Override
public void run() {

Log.d("AndroidClarified", "Running");
handler.postDelayed(this, 1000 * 5);
}
}

}
```

Step -2 Declaring the service in Manifest

As for every android component we first need to declare our service inside the AndroidManifest.xml. For this you just need to add the below code snippet under the <application> tag

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.androidclarified.serviceapp">
<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round"
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>
```

Step -3 -Starting the service

Since every service needs to be started from some other Android component. We will first create a Android Activity(If you are not sure how to create a activity read this) . You can just right click on your package and select New->Activity->Empty Activity. This will automatically create an Activity class and a layout file.

- We add two buttons to the layout file as shown in the snippet below

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="com.example.androidclarified.serviceapp.MainActivity">
<LinearLayout
android:layout_width="wrap_content"
android:layout_centerInParent="true"
android:layout_height="wrap_content"
android:orientation="horizontal">
<Button
android:id="@+id/start_button"
android:layout_margin="20dp"
android:text="Start Service"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
<Button
android:layout_width="wrap_content"
android:text="Stop Service"
android:layout_margin="20dp"
android:id="@+id/stop_button"
android:layout_height="wrap_content" />
</LinearLayout>
</RelativeLayout>
```

- Now we have to initialize both the buttons and add click listeners. This is how our activity class will look now
- Starting the service is similar to starting any other activity. You just need to create an Intent object specifying the service and then call *startService()* .

```
1. public class MainActivity extends AppCompatActivity implements
   View.OnClickListener {
2.
3. private Button startButton, stopButton;
4.
5. @Override
6. protected void onCreate(Bundle savedInstanceState) {
7. super.onCreate(savedInstanceState);
8. setContentView(R.layout.activity_main);
9.
10. startButton = (Button) findViewById(R.id.start_button);
11. stopButton = (Button) findViewById(R.id.stop_button);
12.
13. startButton.setOnClickListener(this);
14. stopButton.setOnClickListener(this);
15. }
16.
17. @Override
18. public void onClick(View v) {
19.
20. Intent intent = new Intent(this, MyService.class);
21.
22. switch (v.getId()) {
23. case R.id.start_button:
24. startService(intent);
25. break;
26. case R.id.stop_button:
27. stopService(intent);
28. break;
29.
30. }
31.
32. }
33. }
```

Topic-5 Broadcast receiver

Definition

A *broadcast receiver* (*receiver*) is an Android component which allows you to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens.

For example, applications can register for the ACTION_BOOT_COMPLETED system event which is fired once the Android system has completed the boot process.

Implementation

A receiver can be registered via the *AndroidManifest.xml* file.

Alternatively to this static registration, you can also register a receiver dynamically via theContext.registerReceiver() method.

The implementing class for a receiver extends the BroadcastReceiver class.

If the event for which the broadcast receiver has registered happens, the onReceive() method of the receiver is called by the Android system.

Life cycle of a broadcast receiver

After the onReceive() of the receiver class has finished, the Android system is allowed to recycle the receiver.

Asynchronous processing

Before API level 11, you could not perform any asynchronous operation in the onReceive()method, because once the onReceive() method had been finished, the Android system was allowed to recycle that component. If you have potentially long running operations, you should trigger a service instead.

Since Android API 11 you can call the goAsync() method. This method returns an object of thePendingResult type. The Android system considers the receiver as alive until you call thePendingResult.finish() on this object. With this option you can trigger asynchronous processing in a receiver. As soon as that thread has completed, its task calls finish() to indicate to the Android system that this component can be recycled.

Restrictions for defining broadcast receiver

As of Android 3.1 the Android system excludes all receiver from receiving intents by default if the corresponding application has never been started by the user or if the user explicitly stopped the application via the Android menu (in **Manage Application**).

This is an additional security feature as the user can be sure that only the applications he started will receive broadcast *intents*.

> This does not mean the user has to start the application again after a reboot. The Andr
> system remembers that the user already started it. Only one start is required without a
> stop by the user.

Send the broadcast to your application for testing

You can use the following command from the adb command line tool. The class name and package names which are targeted via the command line tool need to be as defined in the manifest. You should send the intent you generated to your specific component, for example if you send a general ACTION_BOOT_COMPLETED broadcast, this will trigger a lot of things in an Android system.

```
# trigger a broadcast and deliver it to a component
adb shell am activity/service/broadcast -a ACTION -c CATEGORY -n NAME

# for example (this goes into one line)

adb shell am broadcast -a
android.intent.action.BOOT_COMPLETED -c android.intent.category.HOME -n
package_name/class_name
```

Pending Intent

A *pending intent* is a token that you give to another application. For example, the notification manager, alarm manager or other 3rd party applications). This allows the other application to restore the permissions of your application to execute a predefined piece of code.

To perform a broadcast via a pending intent, get a PendingIntent via the getBroadcast()method of the PendingIntent class. To perform an activity via a pending intent, you receive the activity via PendingIntent.getActivity().

System broadcasts

Several system events are defined as final static fields in the Intent class. Other Android system classes also define events, e.g., the TelephonyManager defines events for the change of the phone state.

The following table lists a few important system events.

Table 1. System Events

Event	Description
Intent.ACTION_BOOT_COMPLETED	Boot completed. Requires the android.permission.RECEIVE_BOOT_COMPLETED on
Intent.ACTION_POWER_CONNECTED	Power got connected to the device.
Intent.ACTION_POWER_DISCONNECTED	Power got disconnected to the device.
Intent.ACTION_BATTERY_LOW	Triggered on low battery. Typically used to reduce act your app which consume power.
Intent.ACTION_BATTERY_OKAY	Battery status good again.

Automatically starting Services from a Receivers

A common requirement is to automatically start a service after a system reboot, i.e., for synchronizing data. For this you can register a receiver for theandroid.intent.action.BOOT_COMPLETED system event. This requires

theandroid.permission.RECEIVE_BOOT_COMPLETED permission.

The following example demonstrates the registration for the BOOT_COMPLETED event in the Android manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="de.vogella.android.ownservice.local"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk android:minSdkVersion="10" />

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

   <application
      android:icon="@drawable/icon"
      android:label="@string/app_name" >
      <activity
         android:name=".ServiceConsumerActivity"
         android:label="@string/app_name" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

      <receiver android:name="MyScheduleReceiver" >
         <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
         </intent-filter>
      </receiver>
      <receiver android:name="MyStartServiceReceiver" >
      </receiver>
   </application>

</manifest>
```

The receive would start the service as demonstrated in the following example code.

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class MyReceiver extends BroadcastReceiver {

   @Override
   public void onReceive(Context context, Intent intent) {
```

```
        // assumes WordService is a registered service
        Intent intent = new Intent(context, WordService.class);
        context.startService(intent);
    }
}
```

> If your application is installed on the SD card, then it is not available after theandroid.intent.action.BOOT_COMPLETED event. In this case register it for the `android.intent.action.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE ` event.
>
> Remember that as of Android API level 11 the user needs to have started the applicatio least once before your application can receiveandroid.intent.action.BOOT_COMPLETED events.

Exercise: Register a receiver for incoming phone calls

Target

In this exercise you define a broadcast receiver which listens to telephone state changes. If the phone receives a phone call, then our receiver will be notified and log a message.

Create project

Create a new project called de.vogella.android.receiver.phone. Also create an activity.

TIP:Remember that your receiver is only called if the user started it once. This requires an activity.

Implement receiver for the phone event

Create the MyPhoneReceiver class.

```
package de.vogella.android.receiver.phone;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.util.Log;

public class MyPhoneReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(Context context, Intent intent) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
      String state = extras.getString(TelephonyManager.EXTRA_STATE);
      Log.w("MY_DEBUG_TAG", state);
      if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
```

```
            String phoneNumber = extras
                  .getString(TelephonyManager.EXTRA_INCOMING_NUMBER);
            Log.w("MY_DEBUG_TAG", phoneNumber);
        }
    }
  }
}
```

Request permission

Add the android.permission.READ_PHONE_STATE permission to your manifest file which allows you to listen to state changes in your receiver. Also Register your receiver in your manifest file. The resulting manifest should be similar to the following listing.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="de.vogella.android.receiver.phone"
   android:versionCode="1"
   android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="15" />

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

   <application
      android:icon="@drawable/icon"
      android:label="@string/app_name" >
        <activity
         android:name=".MainActivity"
         android:label="@string/title_activity_main" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

      <receiver android:name="MyPhoneReceiver" >
         <intent-filter>
            <action android:name="android.intent.action.PHONE_STATE" >
            </action>
         </intent-filter>
      </receiver>
   </application>



</manifest>
```

Validate implementations

Install your application and simulate a phone call via the emulator controls. Validate that your receiver is called and logs a message to the LogCat view.

Exercise: System services and receiver

Target

In this chapter we will schedule a receiver via the Android alert manager system service. Once called, it uses the Android vibrator manager and a popup message (Toast) to notify the user.

Implement project

Create a new project called *de.vogella.android.alarm* with the activity called *AlarmActivity*.

Create the following layout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical" >

   <EditText
      android:id="@+id/time"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:hint="Number of seconds"
      android:inputType="numberDecimal" >
   </EditText>

   <Button
      android:id="@+id/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="startAlert"
      android:text="Start Counter" >
   </Button>

</LinearLayout>
```

Create the following broadcast receiver class. This class will get the vibrator service.

```
package de.vogella.android.alarm;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
```

```
import android.os.Vibrator;
import android.widget.Toast;

public class MyBroadcastReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      Toast.makeText(context, "Don't panik but your time is up!!!!.",
            Toast.LENGTH_LONG).show();
      // Vibrate the mobile phone
      Vibrator vibrator = (Vibrator)
context.getSystemService(Context.VIBRATOR_SERVICE);
      vibrator.vibrate(2000);
   }

}
```

Register this class as a broadcast receiver in *AndroidManifest.xml* and request authorization to vibrate the phone.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="de.vogella.android.alarm"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk android:minSdkVersion="15" />

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

   <application
      android:icon="@drawable/icon"
      android:label="@string/app_name" >
      <activity
         android:name=".AlarmActivity"
         android:label="@string/app_name" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

      <receiver android:name="MyBroadcastReceiver" >
      </receiver>
   </application>

</manifest>
```

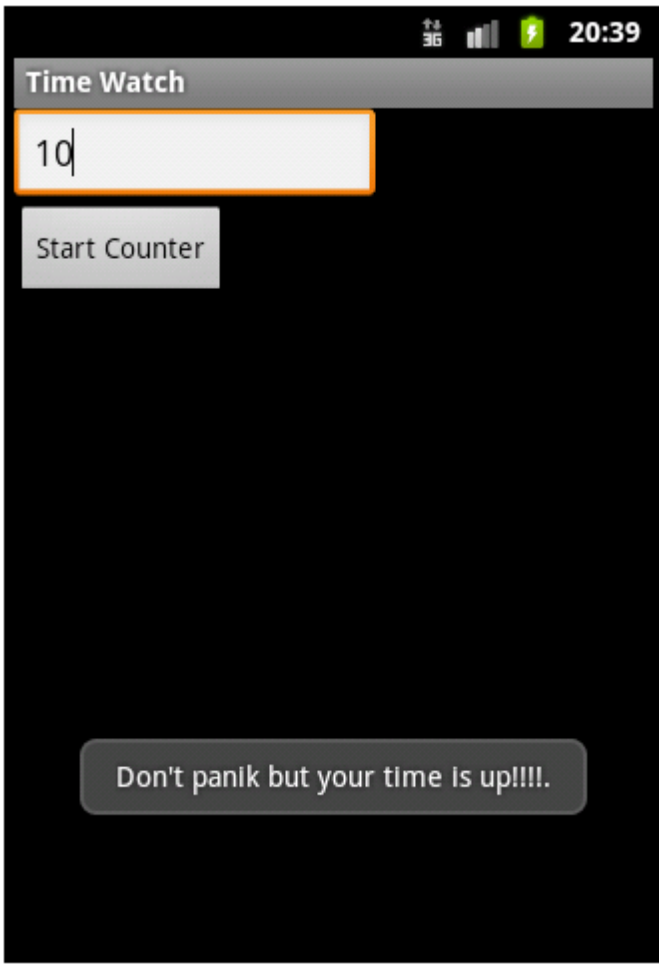
Change the code of your AlarmActivity class to the following. This activity creates an

intent to start the receiver and register this intent with the alarm manager service.

```
package de.vogella.android.alarm;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class AlarmActivity extends Activity {
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void startAlert(View view) {
    EditText text = (EditText) findViewById(R.id.time);
    int i = Integer.parseInt(text.getText().toString());
    Intent intent = new Intent(this, MyBroadcastReceiver.class);
    PendingIntent pendingIntent = PendingIntent.getBroadcast(
        this.getApplicationContext(), 234324243, intent, 0);
    AlarmManager alarmManager = (AlarmManager)
getSystemService(ALARM_SERVICE);
    alarmManager.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis()
        + (i * 1000), pendingIntent);
    Toast.makeText(this, "Alarm set in " + i + " seconds",
        Toast.LENGTH_LONG).show();
  }

}
```

Validate implementation

Run your application on the device. Set your time and start the alarm. After the defined number of seconds a *Toast* should be displayed. Keep in mind that the vibration alarm does not work on the Android emulator.



Dynamic broadcast receiver registration

Dynamically registered receiver

Receiver can be registered via the Android manifest file. You can also register and unregister a receiver at runtime via
the Context.registerReceiver() andContext.unregisterReceiver() methods.

> Do not forget to unregister a dynamically registered *receiver* by
> using Context.unregisterReceiver() method. If you forget this, the Android system repo
> a leaked broadcast receiver error. For instance, if you registered a receive
> in onResume() methods of your activity, you should unregister it in the onPause() meth

Using the package manager to disable static receivers

You can use the PackageManager class to enable or disable receivers registered in your*AndroidManifest.xml* file.

```
ComponentName receiver = new ComponentName(context, myReceiver.class);
PackageManager pm = context.getPackageManager();

pm.setComponentEnabledSetting(receiver,
   PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
```

```
    PackageManager.DONT_KILL_APP);
```

Sticky (broadcast) intents

An intent to trigger a receiver (*broadcast intent*) is not available anymore after it was sent and processed by the system. If you use
the sendStickyBroadcast(Intent) method, the corresponding *intent* is sticky, meaning the intent you are sending stays around after the broadcast is complete.

The Android system uses sticky broadcast for certain system information. For example, the battery status is send as sticky intent and can get received at any time. The following example demonstrates that.

```
// Register for the battery changed event
IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);

/ Intent is sticky so using null as receiver works fine
// return value contains the status
Intent batteryStatus = this.registerReceiver(null, filter);

// Are we charging / charged?
int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = status == BatteryManager.BATTERY_STATUS_CHARGING
    || status == BatteryManager.BATTERY_STATUS_FULL;

boolean isFull = status == BatteryManager.BATTERY_STATUS_FULL;

// How are we charging?
int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_USB;
boolean acCharge = chargePlug == BatteryManager.BATTERY_PLUGGED_AC;
```

You can retrieve that data through the return value
of registerReceiver(BroadcastReceiver, IntentFilter) `. This also works for a null `BroadcastReceiver.

In all other ways, this behaves just as sendBroadcast(Intent).

Sticky broadcast intents typically require special permissions.

Topic-6 Content providers

Content providers can help an application manage access to data stored by itself, stored by other apps, and provide a way to share data with other apps. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process. Implementing a content provider has many advantages. Most importantly you can configure a content provider to allow other applications to securely access and modify your app data as illustrated in figure 1.

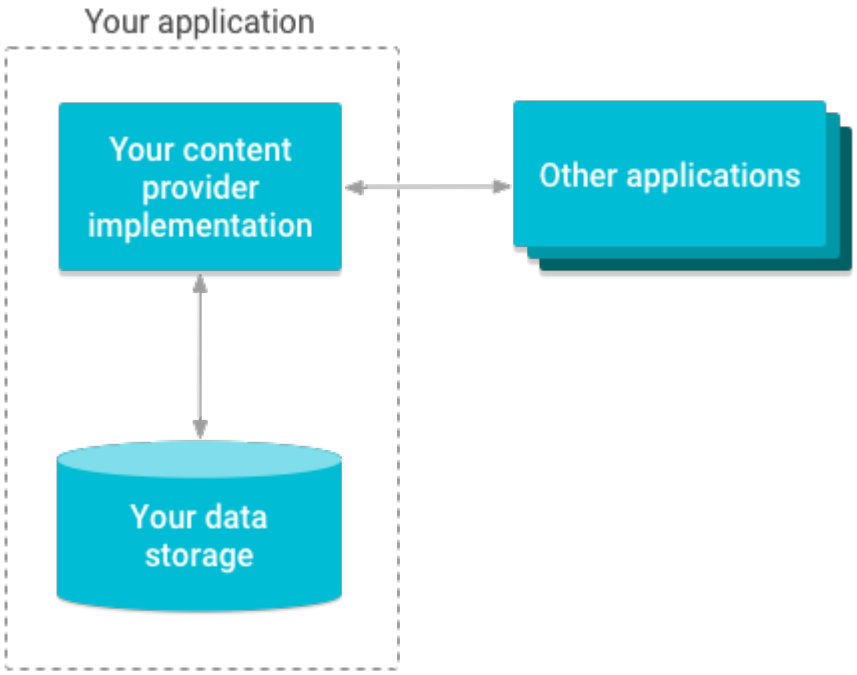


Figure 1. Overview diagram of how content providers manage access to storage.

Use content providers if you plan to share data. If you don't plan to share data, you may still use them because they provide a nice abstraction, but you don't have to. This abstraction allows you to make modifications to your application data storage implementation without affecting other existing applications that rely on access to your data. In this scenario only your content provider is affected and not the applications that access it. For example, you might swap out a SQLite database for alternative storage as illustrated in figure 2.

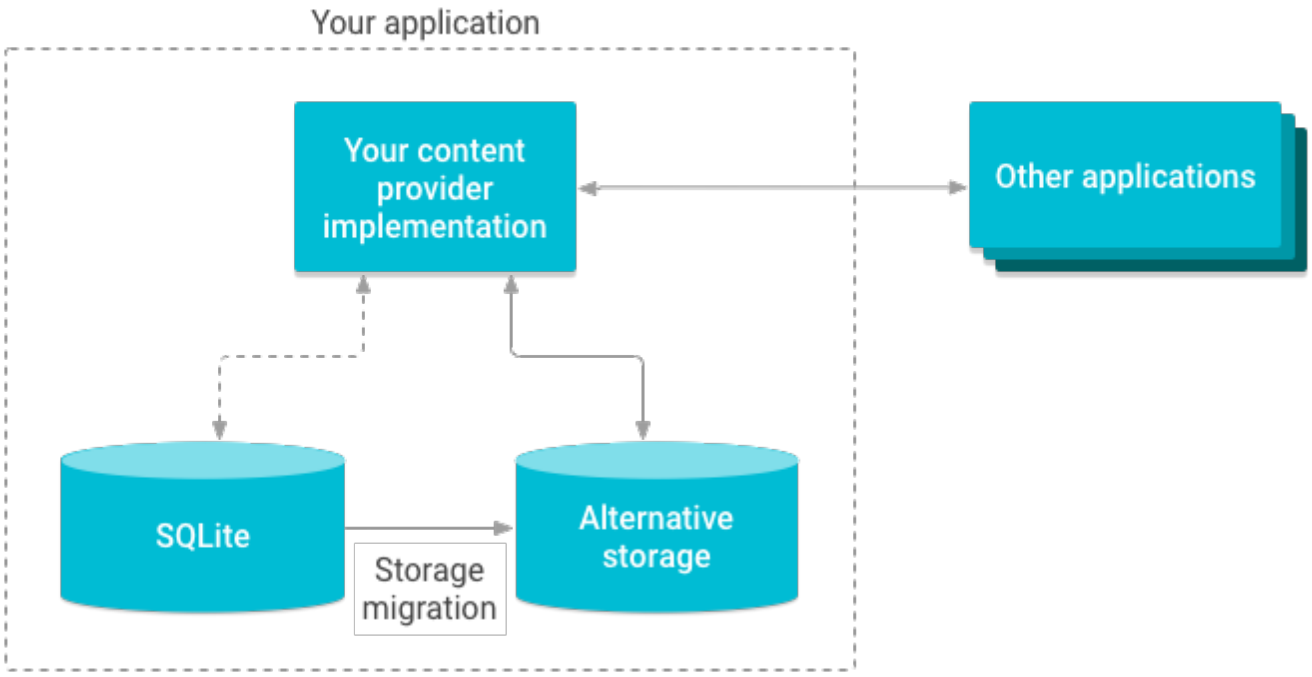


Figure 2. Illustration of migrating content provider storage.

A number of other classes rely on the ContentProvider class:

- AbstractThreadedSyncAdapter
- CursorAdapter
- CursorLoader

If you are making use of any of these classes you also need to implement a content provider in your application. Note that when working with the sync adapter framework you can also create a stub content provider as an alternative. For more information about this topic, see Creating a stub content provider. In addition, you need your own content provider in the following cases:

- You want to implement custom search suggestions in your application
- You need to use a content provider to expose your application data to widgets
- You want to copy and paste complex data or files from your application to other applications

The Android framework includes content providers that manage data such as audio, video, images, and personal contact information. You can see some of them listed in the reference documentation for theandroid.provider package. With some restrictions, these providers are accessible to any Android application.

A content provider can be used to manage access to a variety of data storage sources, including both structured data, such as a SQLite relational database, or unstructured data such as image files. For more information on the types of storage

available on Android, see Storage options, as well as Designing data storage.

Advantages of content providers

Content providers offer granular control over the permissions for accessing data. You can choose to restrict access to a content provider from solely within your application, grant blanket permission to access data from other applications, or configure different permissions for reading and writing data. For more information on using content providers securely, see Security tips for storing data, as well as Content provider permissions.

You can use a content provider to abstract away the details for accessing different data sources in your application. For example, your application might store structured records in a SQLite database, as well as video and audio files. You can use a content provider to access all of this data, if you implement this development pattern in your application.

Also note that CursorLoader objects rely on content providers to run asynchronous queries and then return the results to the UI layer in your application. For more information on using a CursorLoader to load data in the background, see Running a query with a CursorLoader.

Topic-7 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. If you want to start building your notifications, instead read Create a Notification.

For more information about the design and interaction patterns, see the Notifications design guide. Additionally, see the Android Notifications Sample for a demonstration of best practices in using the Notification.Style API in both mobile and wearable apps.

Appearances on a device

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

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.

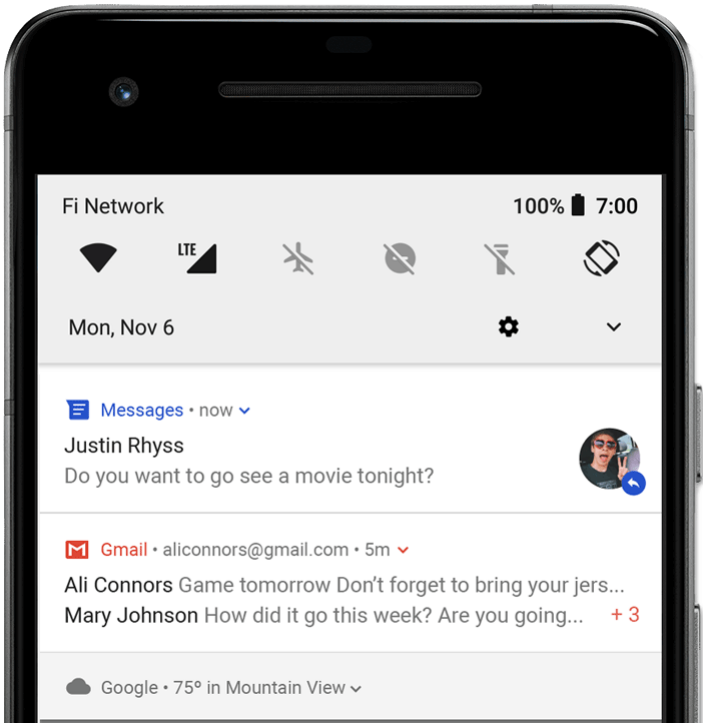


Figure 2. Notifications in the notification drawer

Users can drag down on a notification in the drawer to reveal the expanded view, which shows additional content and action buttons, if provided.

A notification remains visible in the notification drawer until dismissed by the app or the user.

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 should know about immediately, and it appears only if the device is unlocked.

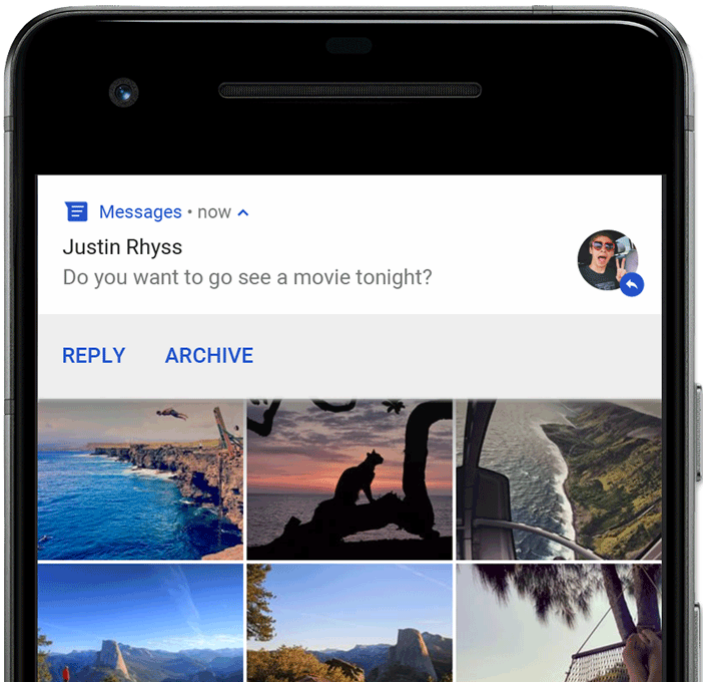


Figure 3. A heads-up notification appears in front of the foreground app

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

Example conditions that might trigger heads-up notifications include the following:

- The user's activity is in fullscreen mode (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.

Lock screen

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

You can programmatically set the level of detail visible in notifications posted by your app on a secure lock screen, or even whether the notification will show on the lock screen at all.

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

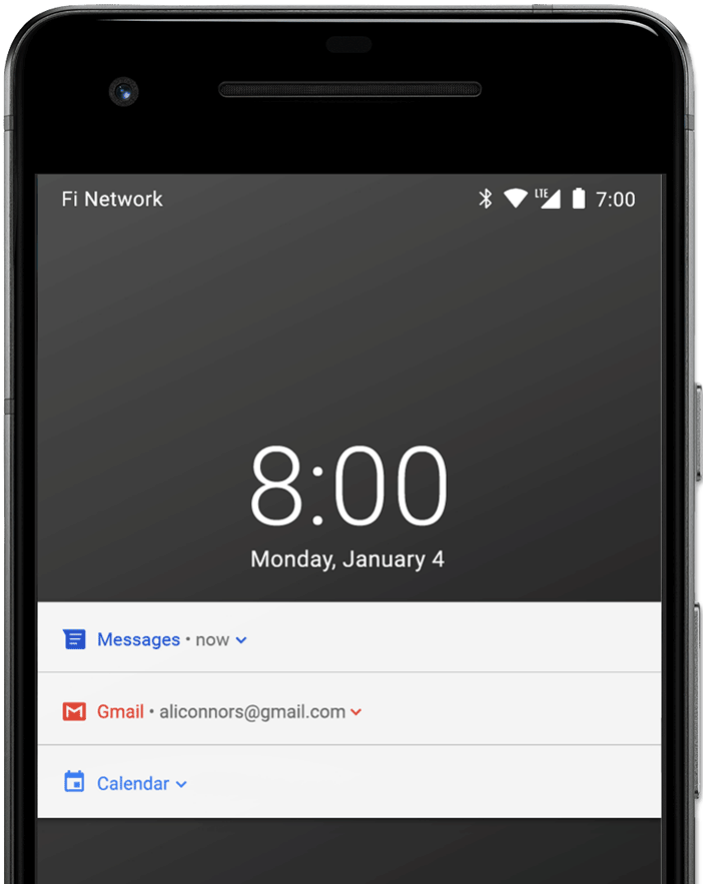


Figure 4. Notifications on the lock screen with sensitive content hidden

To learn more, see how to 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" (also known as a "notification dot") on the corresponding app launcher icon.

Users can long-press on an app icon to see the notifications for that app. Users can then dismiss or act on notifications from that menu, similar to the notification drawer.

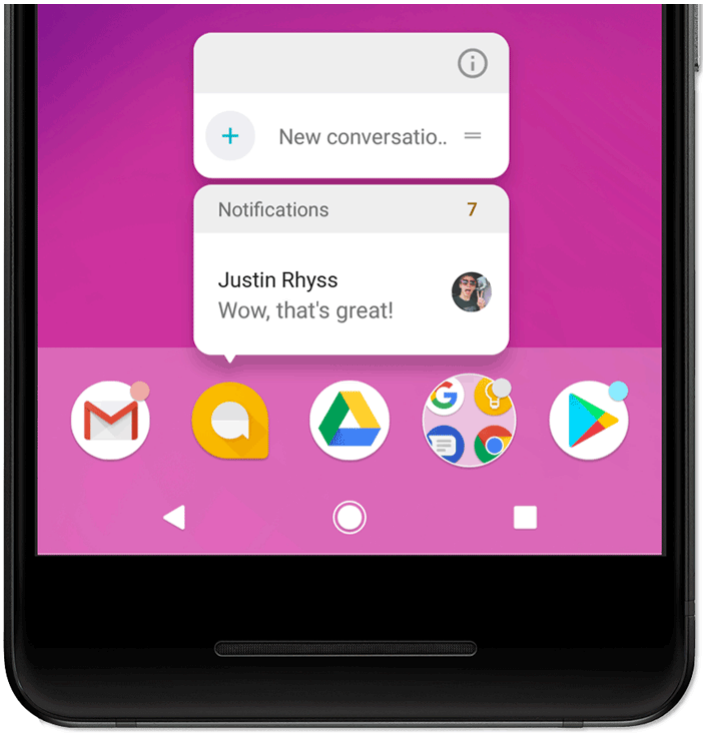


Figure 5. Notification badges and the long-press menu

To learn more about how badges work, read Notification badges.

Wear OS devices

If the user has a paired Wear OS device, all your notifications appear there automatically, including expandable detail and action buttons.

You can also enhance the experience by customizing some appearances for the notification on wearables and provide different actions, including suggested replies and voice input replies. For more information, see how toadd wearable-specific features to your notification.

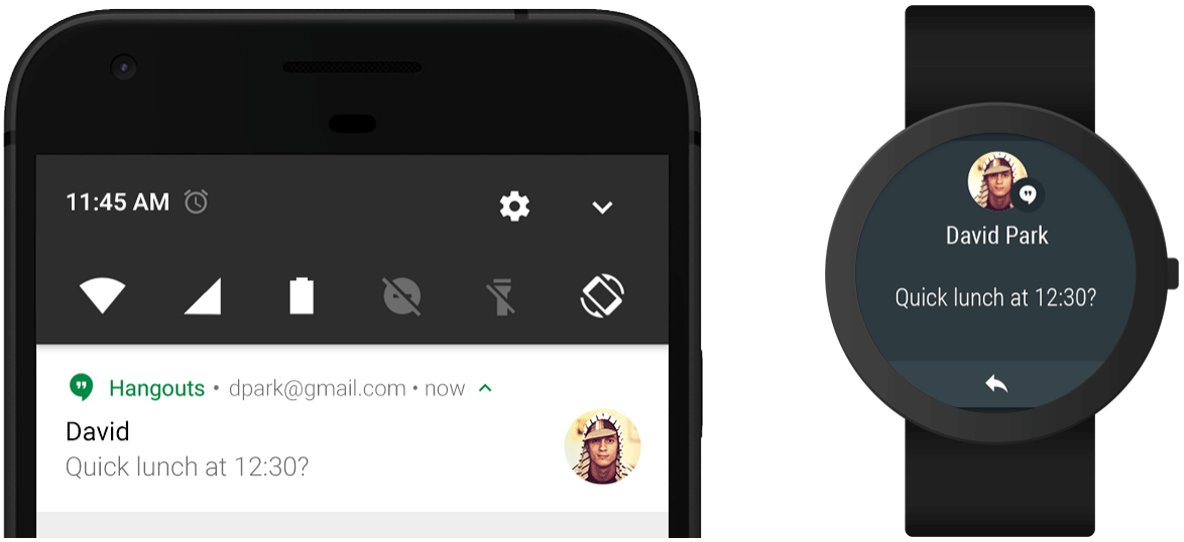


Figure 6. Notifications automatically appear on a paired Wear OS device

Notification anatomy

The design of a notification is determined by system templates—your app simply defines the contents for each portion of the template. Some details of the notification appear only in the expanded view.

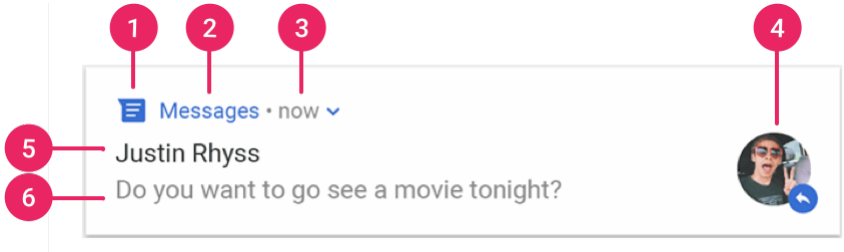


Figure 7. A notification with basic details

The most common parts of a notification are indicated in figure 7 as follows:

1. Small icon: This is required and set with setSmallIcon().
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with setWhen() or hide it with setShowWhen(false).
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set withsetLargeIcon().
5. Title: This is optional and set with setContentTitle().
6. Text: This is optional and set with setContentText().

For more information about how to create a notification with these features and more, read Create a Notification.

Notification actions

Although it's not required, every notification should open an appropriate app activity when tapped. In addition to this default notification action, you can add action buttons that complete an app-related task from the notification (often without opening an activity), as shown in figure 9.

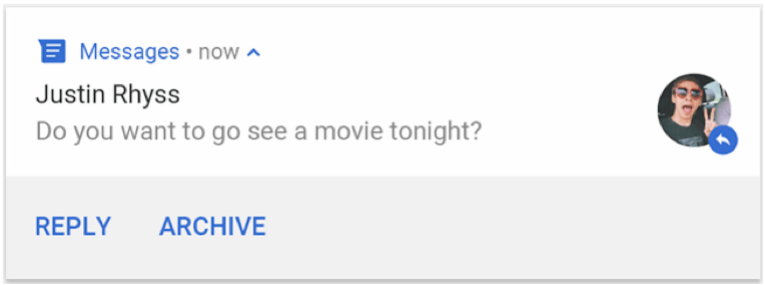


Figure 9. A notification with action buttons

Starting in Android 7.0 (API level 24), you can also add an action to reply to messages or enter other text directly from the notification.

Adding action buttons is explained further in Create a Notification.

Expandable notification

By default, the notification's text content is truncated to fit on one line. If you want your notification to be longer, you can enable a larger text area that's expandable by applying an additional template, as shown in figure 8.

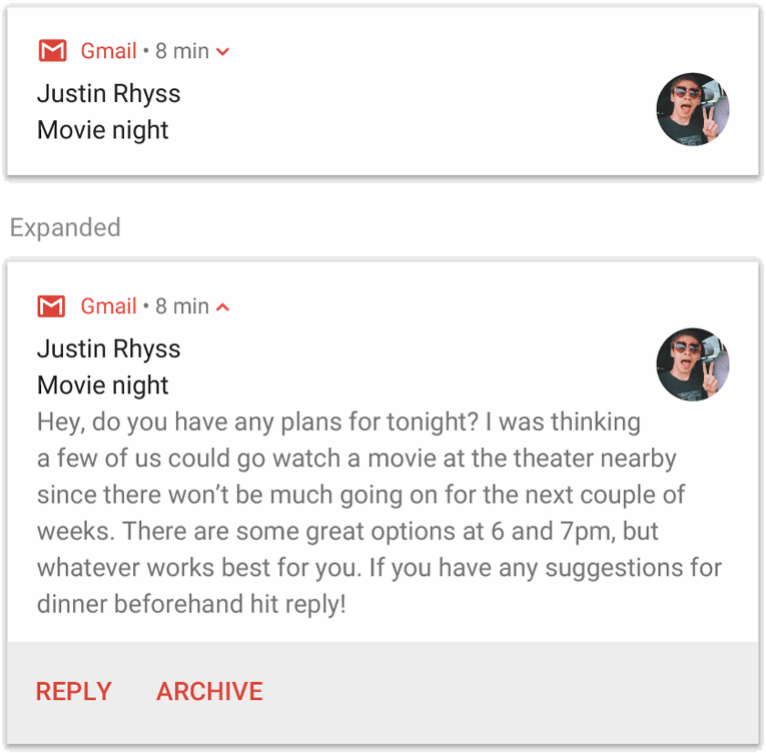


Figure 8. An expandable notification for large text

You can also create an expandable notification with an image, in inbox style, a chat conversation, or media playback controls. For more information, read Create an Expandable Notification.

And although we recommend you always use these templates to ensure proper design compatibility on all devices, if necessary, you can also create a custom notification layout.

Notification updates and groups

To avoid bombarding your users with multiple or redundant notifications when you have additional updates, you should consider updating an existing notification rather than issuing a new one, or consider using the inbox-style notification to show conversation updates.

However, if it's necessary to deliver multiple notifications, you should consider grouping those separate notifications into a group (available on Android 7.0 and higher). A notification group allows you to collapse multiple notifications into just one post in the notification drawer, with a summary. The user can then expand the notification to reveal the details for each individual notification.

The user can progressively expand the notification group and each notification within it for more details.

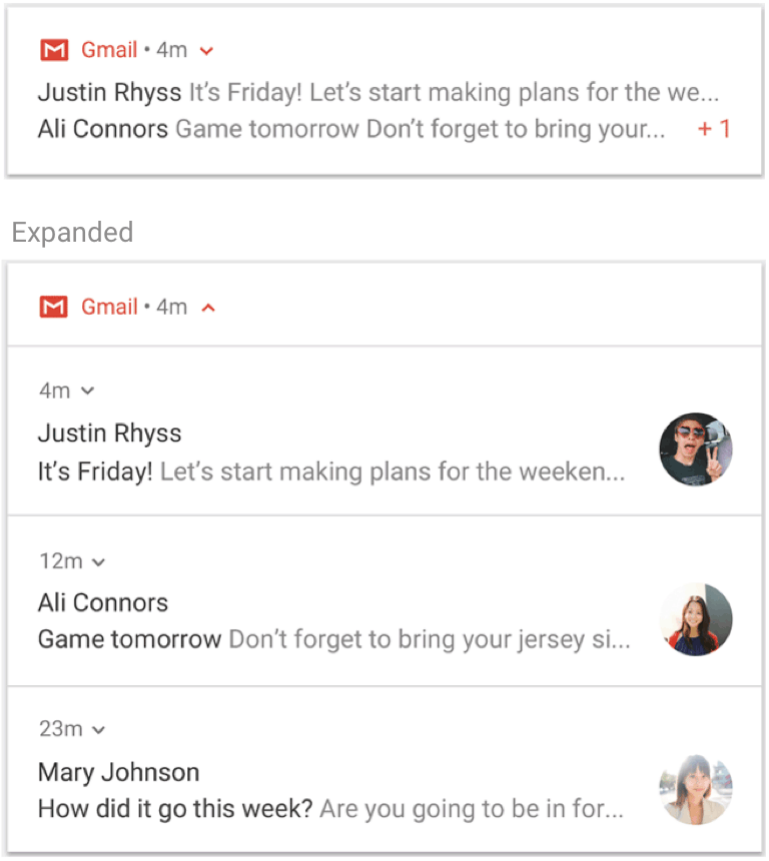


Figure 10. A collapsed and expanded notification group

To learn how to add notifications to a group, see Create a Group of Notifications.

Note: If the same app sends four or more notifications and does not specify a grouping, the system automatically groups them together.

Notification channels

Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel or it will not appear. By categorizing notifications into channels, users can disable specific notification channels for your app (instead of disabling *all* your notifications), and users can control the visual and auditory options for each channel—all from the Android system settings (figure 11). Users can also long-press a notification to change behaviors for the associated channel.

On devices running Android 7.1 (API level 25) and lower, users can manage notifications on a per-app basis only (effectively each app only has one channel on Android 7.1 and lower).

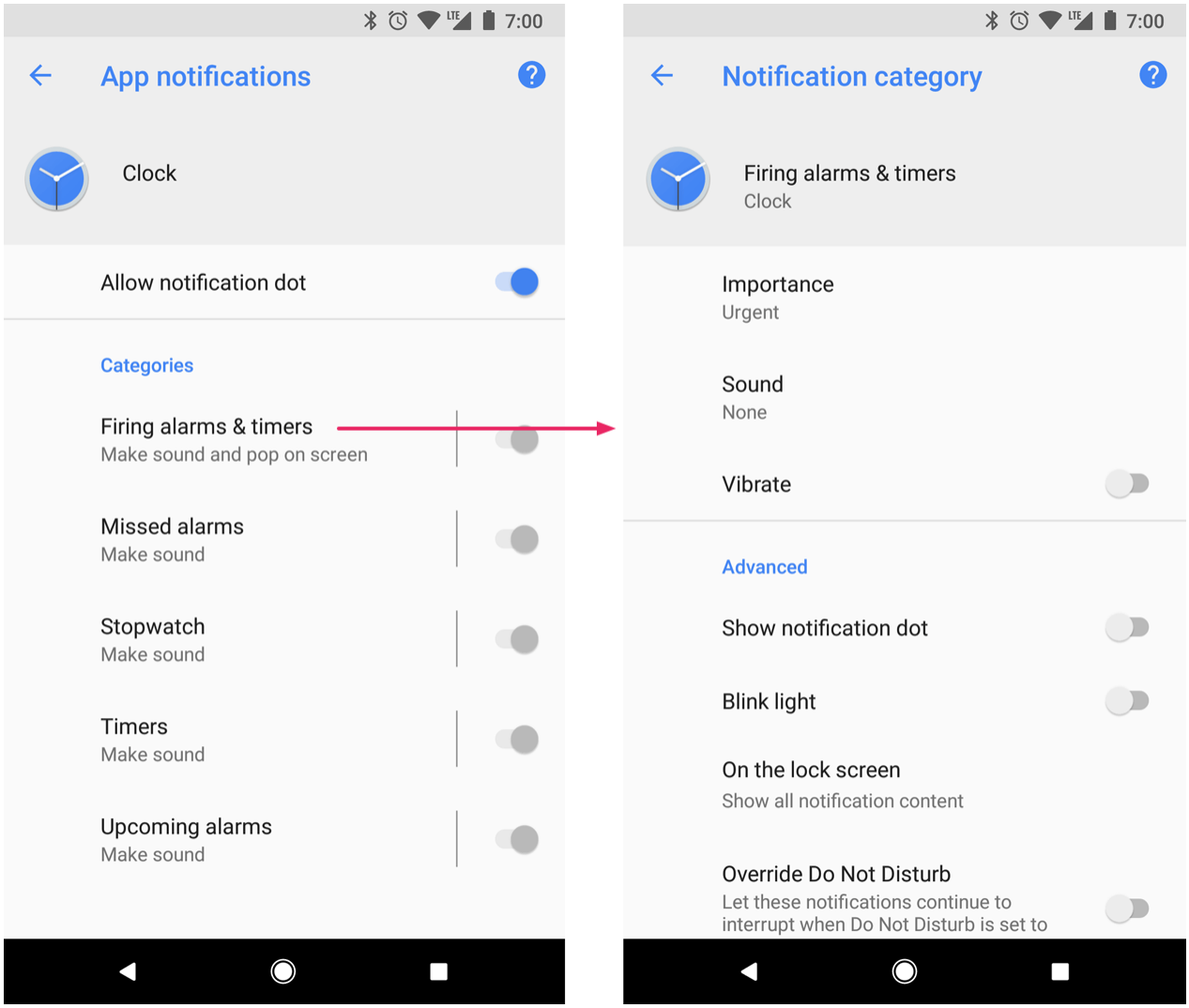


Figure 11. Notification settings for Clock app and one of its channels

Note: The user interface refers to channels as "categories."

One app can have multiple notification channels—a separate channel for each type of notification the app issues. An app can also create notification channels in response to choices made by users of your app. For example, you may set up separate notification channels for each conversation group created by a user in a messaging app.

The channel is also where you specify the importance level for your notifications on Android 8.0 and higher. So all notifications posted to the same notification channel have the same behavior.

For more information, see Create and Manage Notification Channels.

Notification importance

Android uses the *importance* of a notification to determine how much the notification should interrupt the user (visually and audibly). The higher the importance of a notification, the more interruptive the notification will be.

On Android 8.0 (API level 26) and above, importance of a notification is determined by the importance of the channel the notification was posted to. Users can change the importance of a notification channel in the system settings (figure 12). On Android 7.1 (API level 25) and below, importance of each notification is determined by the notification's priority.

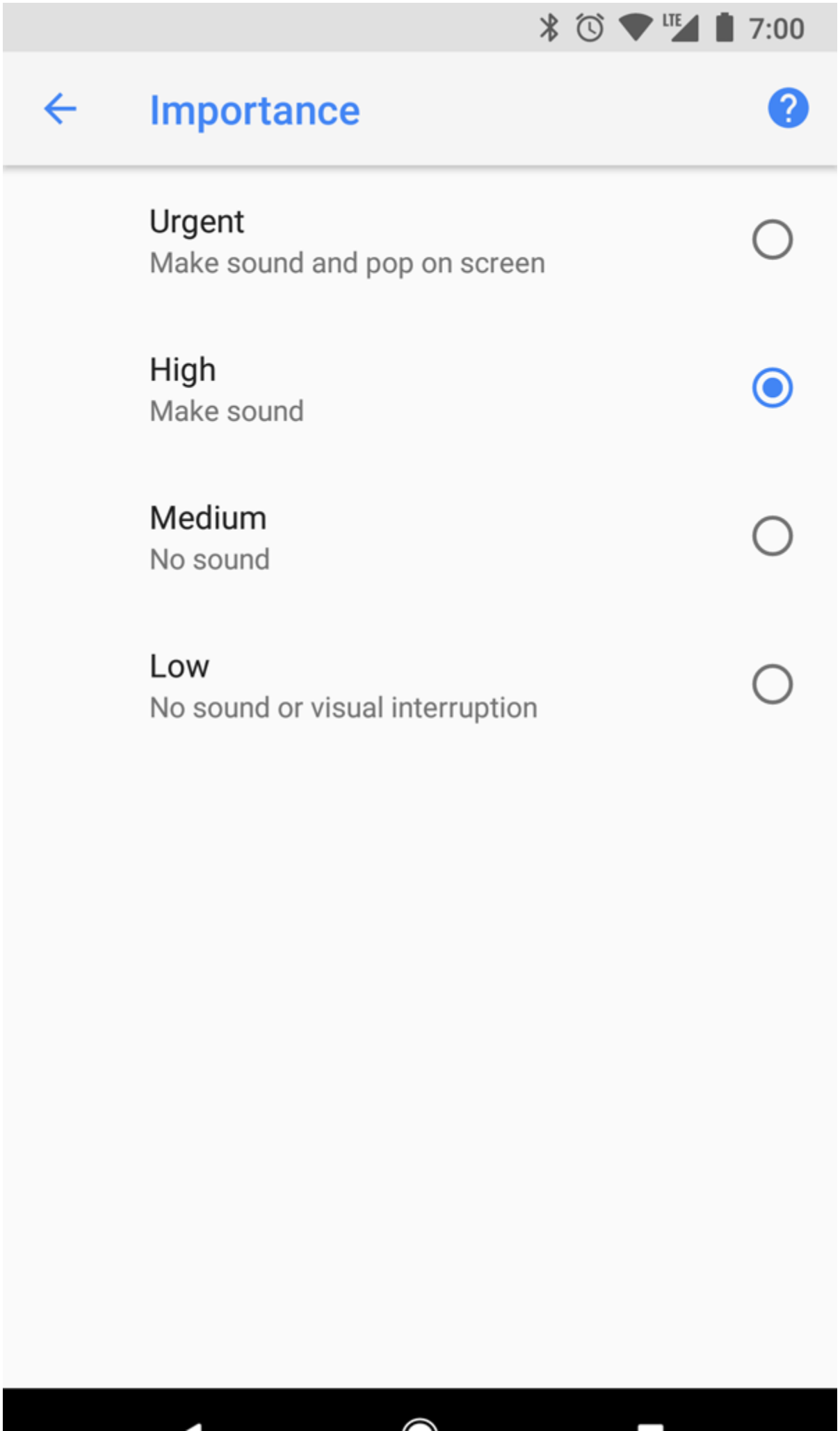

7:00
Importance
Urgent
Make sound and pop on screen
High
Make sound
Medium
No sound
Low
No sound or visual interruption

Figure 12. Users can change the importance of each channel on Android 8.0 and higher

The possible importance levels are the following:

- Urgent: Makes a sound and appears as a heads-up notification.
- High: Makes a sound.
- Medium: No sound.
- Low: No sound and does not appear in the status bar.

All notifications, regardless of importance, appear in non-interruptive system UI locations, such as in the notification drawer and as a badge on the launcher icon (though you can modify the appearance of the notification badge).