# Android: Activities, Intents and Intent Filters

http://developer.android.com/guide/components/activities.html

http://developer.android.com/guide/components/intents-filters.html

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2018-2019)

## Outline

## Outline

## Overview

An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map.

- Each activity is given a window in which to draw its user interface.
- The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually contains multiple activities. Each activity should be designed around a specific kind of action the user can perform and can start other activities.

**Example**

An email application might have one activity to show a list of new messages. When the user selects a message, a new activity opens to view that message.

An activity can even start activities that exist in other applications on the device.

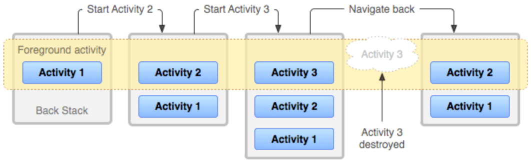
**Example**

If your application wants to send an email message, you can define an intent to perform a "send" action and include some data, such as an email address and a message. An activity from another application that declares itself to handle this kind of intent then opens. In this case, the intent is to send an email, so an email application's "compose" activity starts (if multiple activities support the same intent, then the system lets the user select which one to use). When the email is sent, your activity resumes and it seems as if the email activity was part of your application. Even though the activities may be from different applications, Android maintains this seamless user experience by keeping both activities in the same *task*.

# Tasks and Back Stack

A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.

- Each activity can start another activity
  - Each time a new activity starts, the previous activity is stopped
- The system preserves the activities in a LIFO stack
  - The new activity is pushed on top of the stack and takes the focus
  - When the user presses the Back button, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored)



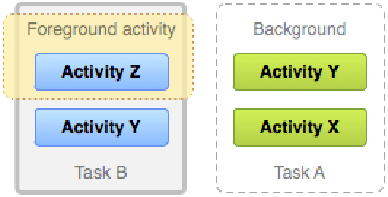
Representation of how each new activity in a task adds an item to the back stack.

When the user presses the Back button, the current activity is destroyed.

- The device Home screen is the starting place for most tasks.
  - When the user touches an icon in the application launcher (or a shortcut on the Home screen), that application's task comes to the foreground.
    - If no task exists for the application (the application has not been used recently), then a new task is created and the "main" activity for that application opens as the root activity in the stack.
  - If the user continues to press Back, then each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen (or to whichever activity was running when the task began).
    - When all activities are removed from the stack, the task no longer exists.

- A task is a cohesive unit that can move to the "background" when users begin a new task or go to the Home screen, via the Home button.
  - While in the background, all the activities in the task are stopped, but the back stack for the task remains intact—the task has simply lost focus while another task takes place, as shown in figure below.

Two tasks: Task B receives user interaction in the foreground, while Task A is in the background, waiting to be resumed.



  - Multiple tasks can be held in the background at once. However, if the user is running many background tasks at the same time, the system might begin destroying background activities in order to recover memory, causing the activity states to be lost.

The activity lifecycle has three different phases:

**Entire lifetime**

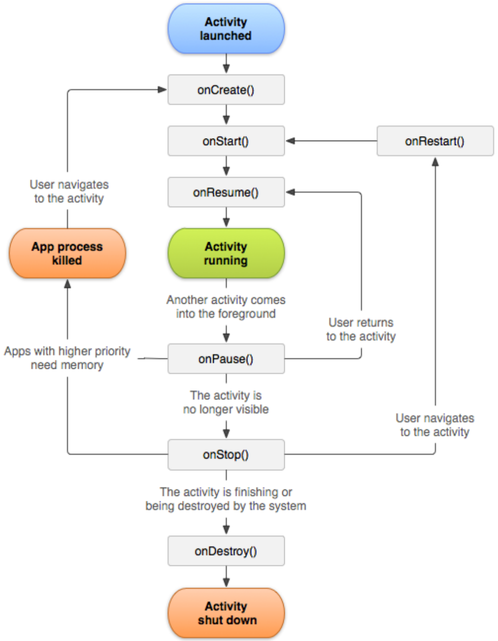
- Between onCreate() and onDestroy()
- Setup of global state in onCreate()
- Release remaining resources in onDestroy()

**Visible lifetime**

- Between onStart() and onStop()
- Maintain resources that have to be shown to the user

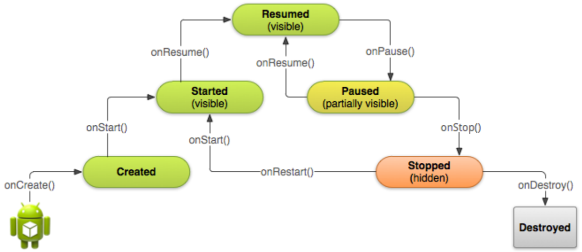
**Foreground lifetime**

- Between onResume() and onPause()


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

A simplified illustration of the Activity lifecycle, expressed as a step pyramid.

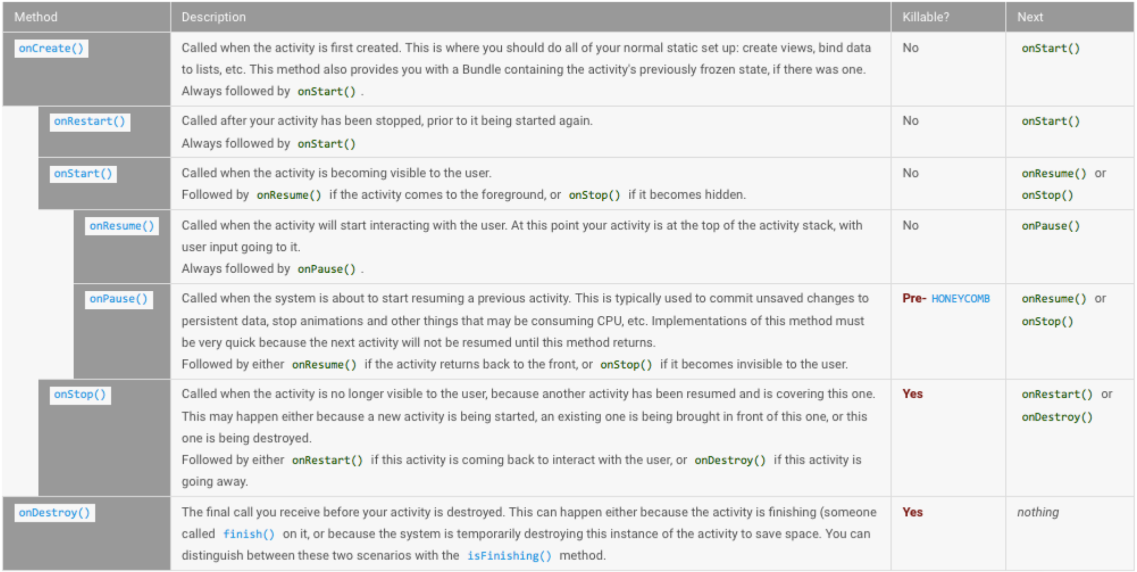
Only three of these states can be static. That is, the activity can exist in one of only three states for an extended period of time:



- **Resumed.** In this state, the activity is in the foreground and the user can interact with it. (Also sometimes referred to as the "running" state.)
- **Paused.** The activity is partially obscured by another activity—the other activity that's in the foreground is semi-transparent or doesn't cover the entire screen. The paused activity does not receive user input and cannot execute any code.
- **Stopped.** The activity is completely hidden and not visible to the user; it is considered to be in the background. While stopped, the activity instance and all its state information such as member variables is retained, but it cannot execute any code.

The other states (Created and Started) are transient and the system quickly moves from them to the next state by calling the next lifecycle callback method. That is, after the system calls onCreate(), it quickly calls onStart(), which is quickly followed by onResume().

## A summary of the activity lifecycle's callback methods.



| Method | Description | Killable? | Next |
|---|---|---|---|
| `onCreate()` | Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one. Always followed by `onStart()` . | No | `onStart()` |
| `onRestart()` | Called after your activity has been stopped, prior to it being started again. Always followed by `onStart()` | No | `onStart()` |
| `onStart()` | Called when the activity is becoming visible to the user. Followed by `onResume()` if the activity comes to the foreground, or `onStop()` if it becomes hidden. | No | `onResume()` or `onStop()` |
| `onResume()` | Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it. Always followed by `onPause()` . | No | `onPause()` |
| `onPause()` | Called when the system is about to start resuming a previous activity. This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. Implementations of this method must be very quick because the next activity will not be resumed until this method returns. Followed by either `onResume()` if the activity returns back to the front, or `onStop()` if it becomes invisible to the user. | **Pre-** `HONEYCOMB` | `onResume()` or `onStop()` |
| `onStop()` | Called when the activity is no longer visible to the user, because another activity has been resumed and is covering this one. This may happen either because a new activity is being started, an existing one is being brought in front of this one, or this one is being destroyed. Followed by either `onRestart()` if this activity is coming back to interact with the user, or `onDestroy()` if this activity is going away. | **Yes** | `onRestart()` or `onDestroy()` |
| `onDestroy()` | The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called `finish()` on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the `isFinishing()` method. | **Yes** | *nothing* |

## Example (from PDM18kotlin2)

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin2.git]

File: activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    <Button
        android:text="@string/button_send1"
        android:onClick="sendMessage"
        ... />
    ...
</android.support.constraint.ConstraintLayout>
```

File: MainActivity.kt

```kotlin
const val EXTRA_MESSAGE = "it.unito.di.educ.pdm18kotlin2.MESSAGE"

class MainActivity : AppCompatActivity() {
    ...
    fun sendMessage(view: View) {
        val intent = Intent(this, DisplayMessageActivity::class.java).apply {
            putExtra(EXTRA_MESSAGE, edit_message.text.toString())
        }
        startActivity(intent)
    }
}
```

File: AndroidManifest.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="it.unito.di.educ.pdm18kotlin2">

    <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>
        <activity android:name=".ShowTimeActivity" />
        <activity android:name=".DisplayMessageActivity" />
        <activity android:name=".ClearOnTopActivity"/>
    </application>

</manifest>
```

File: DisplayMessageActivity.kt

```kotlin
package it.unito.di.educ.pdm18kotlin2

import ...

class DisplayMessageActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        saveLog("onCreate 1")

        setContentView(R.layout.activity_display_message)

        dm_message.apply {
            text = intent.getStringExtra(EXTRA_MESSAGE)
        }
    }

    ...
}
```

File: activity_display_message.xml

```xml
<android.support.constraint.ConstraintLayout
    ...
    tools:context=".DisplayMessageActivity">

    <TextView
        android:id="@+id/dm_title"
        android:text="@string/display_message"
        ... />

    <TextView
        android:id="@+id/dm_message"
        android:text="@string/startMessage"
        ... />

    <Button
        android:id="@+id/dm_home"
        android:onClick="goHome"
        ... />

</android.support.constraint.ConstraintLayout>
```

## Check the back stack

For viewing the list of activities stored into the Back Stack is possible to use this command:

1. From the installation of the SDK (e.g. Android/sdk/platform-tools):
2. **adb shell "dumpsys activity activities | grep Run"**

## Applications

- There is no main
- There is a class Application
  - A base class for keeping a global application state
- Class Activity takes care of creating a window
  - We can place our UI with setContentView(int)
- Activities are presented
  - As full-screen windows
  - As floating windows
    - via a theme with R.attr.windowIsFloating set
  - Embedded inside of another activity
    - using ActivityGroup (deprecated in API level 13)

## Specify Your App's Launcher Activity

- When the user selects your app icon from the Home screen, the system calls the onCreate() method for the Activity in your app that you've declared to be the "launcher" (or "main") activity. This is the activity that serves as the main entry point to your app's user interface.
- The main activity for your app must be declared in the manifest with an <intent-filter> that includes the MAIN action and LAUNCHER category. For example:

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

- Nothing forces each app to have one and only one starting activity
  - The launcher screen would have different icons for the same program
  - If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps

## Example: constraining instances with FLAG_ACTIVITY_CLEAR_TOP (1/2) (from PDM18kotlin2) [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/PDM18kotlin2.git]

File: MainActivity.java

```
class MainActivity : AppCompatActivity() {


    ...
    /**
     * Called when the user clicks the ClearOnTop button
     */
    fun clearOnTop(view: View) {
        val intent = Intent(this, ClearOnTopActivity::class.java).apply {
            flags = Intent.FLAG_ACTIVITY_CLEAR_TOP
        }
        startActivity(intent)
    }
}
```

## Example: constraining instances with FLAG_ACTIVITY_CLEAR_TOP (2/2)

- If the activity being launched is already running in the current task
  - Instead of launching a new instance of that activity
    - All of the other activities on top of it will be closed
    - The intent will be delivered to the (now on top) old activity as a new Intent

**Example**

Consider a task consisting of activities A, B, C, D

- If D calls startActivity() with an Intent that resolves to activity B
  - C and D will be finished and B receives the given Intent
  - The stack now is: A, B

Other flags are available

## Outline

## Intents

- Allow for late binding between components
  - Activities, Services, Broadcast receivers
  - Components can also exchange data
- Indirect communication
  - "Android, please do that with this data"
- Reuse existing and installed applications
- Two possible types

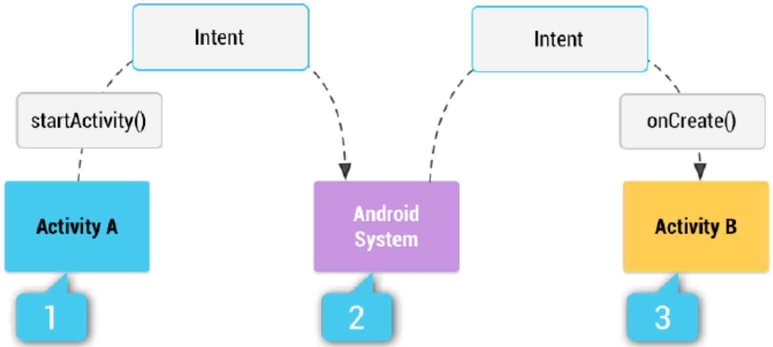
  **Explicit** The target receiver is specified through the Component Name[1]

  **Implicit** The target is specified by data type/names. The system chooses the receiver that matches the request[2]

[1] When you create an explicit intent to start an activity or service, the system immediately starts the app component specified in the Intent object.

[2] When you create an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the Intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use.

Illustration of how an implicit intent is delivered through the system to start another activity: [1] Activity A creates an Intent with an action description and passes it to startActivity(). [2] The Android System searches all apps for an intent filter that matches the intent. When a match is found, [3] the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

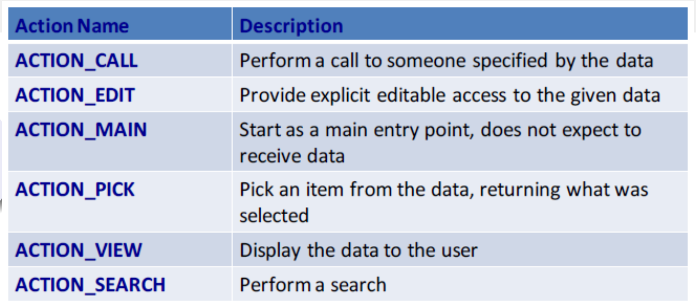


## Intent definition

- Component Name: the receiver of the intent
  - It is optional
- Action name: the action to be performed
  - Predefined actions exist, the programmer can define new ones
  - The action largely determines how the rest of the intent is structured—particularly what is contained in the data and extras
- Data: information exchanged between caller and callee
- Category: the kind of component that should handle the Intent
  - Any number of category descriptions can be placed in an intent, but most intents do not require a category
- Extras: additional information in the form of key-value pairs
- Flags: additional information to instruct Android how to launch an activity, and how to treat it after execution

## Action

A string that specifies the generic action to perform (such as view or pick).



| Action Name | Description |
|---|---|
| ACTION_CALL | Perform a call to someone specified by the data |
| ACTION_EDIT | Provide explicit editable access to the given data |
| ACTION_MAIN | Start as a main entry point, does not expect to receive data |
| ACTION_PICK | Pick an item from the data, returning what was selected |
| ACTION_VIEW | Display the data to the user |
| ACTION_SEARCH | Perform a search |

Examples of action/data pairs

- ACTION_VIEW content://contacts/people/1
  - Display information about the person whose identifier is “1”
- ACTION_DIAL content://contacts/people/1
  - Display the phone dialer with the person filled in
- ACTION_VIEW tel:123
  - Display the phone dialer with the given number filled in
- ACTION_DIAL tel:123
  - Display the phone dialer with the given number filled in
- ACTION_EDIT content://contacts/people/1
  - Edit information about the person whose identifier is “1”
- ACTION_VIEW content://contacts/people/
  - Display a list of people, which the user can browse through

## Explicit intents

- startActivity(intent: Intent!) starts a new activity, and places it on top of the stack
  - The Intent parameter describes the activity we want to execute
  - Often they do not include any other information
    - It is a way for an application to launch internal activities
- Intent(packageContext: Context!, cls: Class<*>!)
  - Context is a wrapper for global information about an application environment
  - Activity subclasses Context

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

```
val intent = Intent().apply {
    component = ComponentName(this@MainActivity, SndAct::class.java)
}
startActivity(intent)
```

### Example

If you built a service in your app, named DownloadService, designed to download a file from the web, you can start it with the following code:

```
// Executed in an Activity, so 'this' is the Context,
// The fileUrl is a string URL, such as "http://www.example.com/image.png"
val downloadIntent = Intent(this, DownloadService::class.java).apply {
    data = Uri.parse(fileUrl)
}
startService(downloadIntent);
```

## Intents with results

- Activities can return results
  - startActivityForResult(intent: Intent!, requestCode: Int)
  - onActivityResult(requestCode: Int, resultCode: Int, data: Intent?)

```
const val PICK_CONTACT_REQUEST = 1
// The request code
val pickContactIntent = Intent(Intent.ACTION_PICK, Uri.parse("content://contacts")).apply {
    type = ContactsContract.CommonDataKinds.Phone.CONTENT_TYPE
}
// Show user only contacts with phone numbers
startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST)
```

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
      // Check which request we're responding to
      if (requestCode == PICK_CONTACT_REQUEST) {
             // Make sure the request was successful
             if (resultCode == RESULT_OK) {
                    // The user picked a contact.
                    // The Intent's data Uri identifies which contact was selected.
                  // Do something with the contact here
}      }      }
```

- setResult()
  - void setResult(resultCode: int, data: Intent!)
  - The result is delivered to the caller component only after invoking the finish() method!

## Implicit intents

- Target component is not named
  - component name is left blank
- When Intent is launched, Android tries to find an activity that can answer it
  - If at least one is found, then that activity is started!
  - If more than one matching activity is found, the user is prompted to make a choice

Implicit intents are very useful to re-use code and to launch external applications

```
val intent = Intent(android.content.Intent.ACTION_VIEW, Uri.parse("http://www.unito.it"))
startActivity(intent)
```

## Intent filters

<intent-filter> tag in AndroidManifest.xml

```
<intent-filter>
       <action android:name="android.intent.action.EDIT" />
       <action android:name="android.intent.action.VIEW" />
       ...
</intent-filter>
```

- The action specified in the Intent must match one of the actions listed in the filter
  - Fail if filter does not specify any action
  - Success if intent does not specify an action but the filter contains at least one action

## Data

The URI (a Uri object) that references the data to be acted on and/or the MIME type of that data. The type of data supplied is generally dictated by the intent's action.

### Example

If the action is ACTION_EDIT, the data should contain the URI of the document to edit.

### Example

An activity that's able to display images probably won't be able to play an audio file, even though the URI formats could be similar. So specifying the MIME type of your data helps the Android system find the best component to receive your intent. However, the MIME type can sometimes be inferred from the URI.

The URI of the intent is compared with the parts of the URI mentioned in the filter

- Both URI and type are compared

```
<intent-filter>
      <data android:mimeType="audio/*" android:scheme="http"/>
      <data android:mimeType="video/mpeg" android:scheme="http"/>
</intent-filter>
```

## Category

A string containing additional information about the kind of component that should handle the intent.

Every category in the Intent must match a category of the filter

- If the category is not specified in the Intent
  - Android assumes it is DEFAULT
  - The filter must include this category to handle the intent

```
<intent-filter>
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Here are some common categories:

- CATEGORY_BROWSABLE
  - The target activity allows itself to be started by a web browser to display data referenced by a link—such as an image or an e-mail message.
- CATEGORY_LAUNCHER
  - The activity is the initial activity of a task and is listed in the system's application launcher.