2015

# Android Activity Lifecycle & State

V1.0

LEE KELLY

# Table of Contents

## 1. Activities

An *activity* represents a single screen with a user interface. For example, an email app might have one activity that shows a list of emails, another activity to compose an email, and another activity for reading an email. Although the activities work together to form a consistent user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities (*if the email app allows it*). For example, a camera app can start the activity in the email app that composes new mail, in order for the user to share a picture.

## 2. The Activity Lifecycle

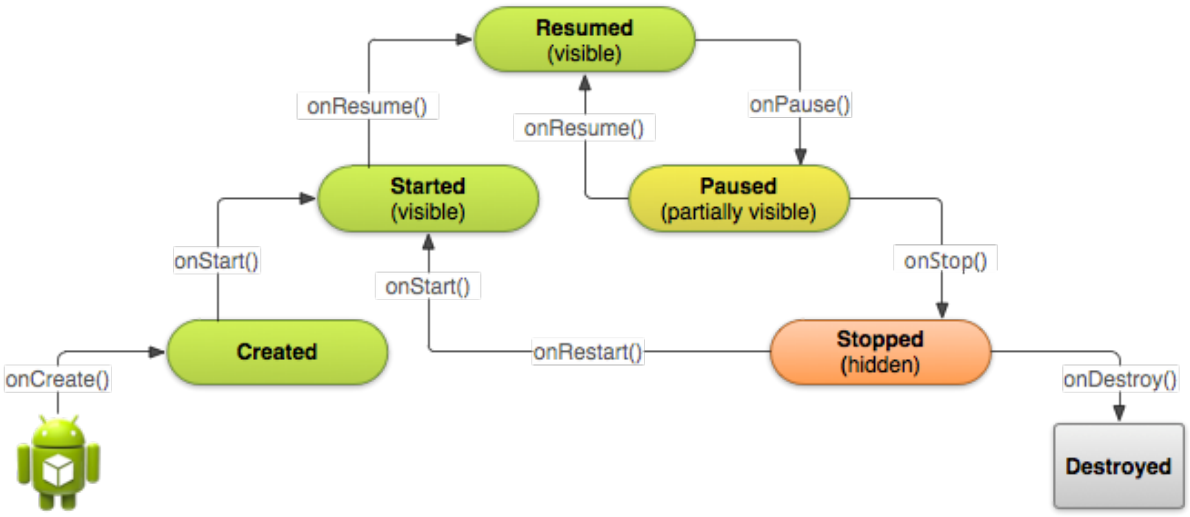
As a user navigates through, out of, and back to your app, the Activity instances in your app transition between different states in their lifecycle.

When your activity starts for the first time, it comes to the foreground of the system and receives user focus. During this process, the Android system calls a series of *lifecycle methods* on the activity where you set up the user interface and other components. If the user performs an action that starts another activity or switches to another app, the system calls another set of *lifecycle methods* on your activity as it moves into the background (*where the activity is no longer visible, but the instance and its state remains intact*).

Within the lifecycle *callback* methods, you can declare how your activity does what the user expects and does not consume system resources when your activity doesn't need them, when the user leaves and re-enters the activity.

## 3. Lifecycle Callbacks

There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity:




You don't need to implement all the lifecycle methods, however implementing your activity lifecycle methods properly ensures your app behaves well in several ways, including that it:

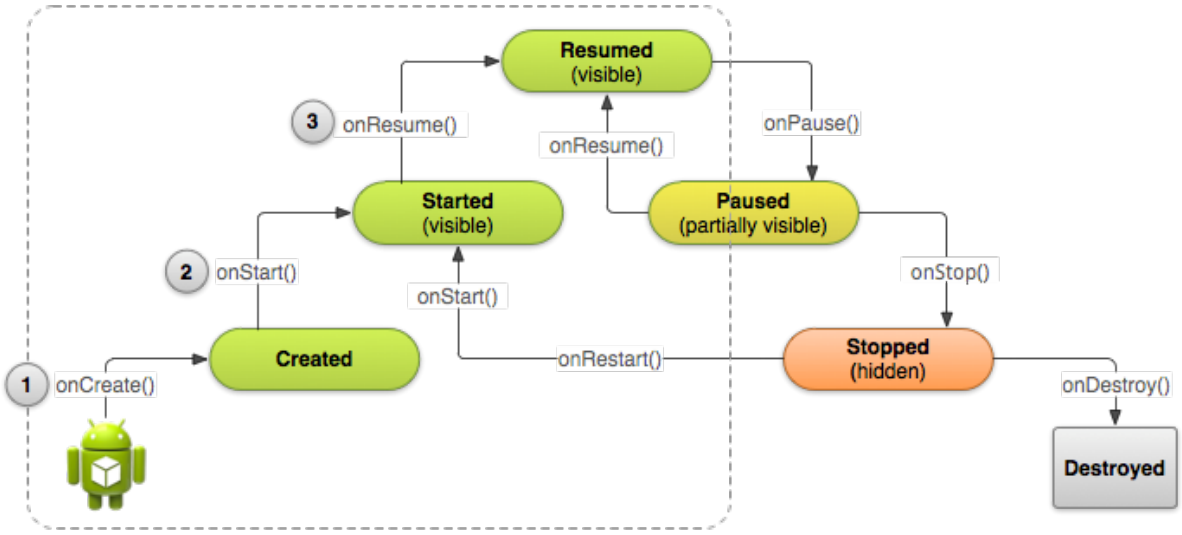
- Does not crash if the user receives a phone call or switches to another app.
- Does not consume valuable system resources when the user is not actively using it.
- Does not lose the user's progress if they leave your app and return to it at a later time.
- Does not crash or lose the user's progress when the screen orientation rotates.

## 3.1 Starting an Activity

Unlike other programming paradigms in which apps are launched with a ***main()*** method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

Whether an activity is the main activity that's created when the user clicks your app icon or a different activity that your app starts in response to a user action, the system creates every new instance of Activity by calling its ***onCreate()*** method.

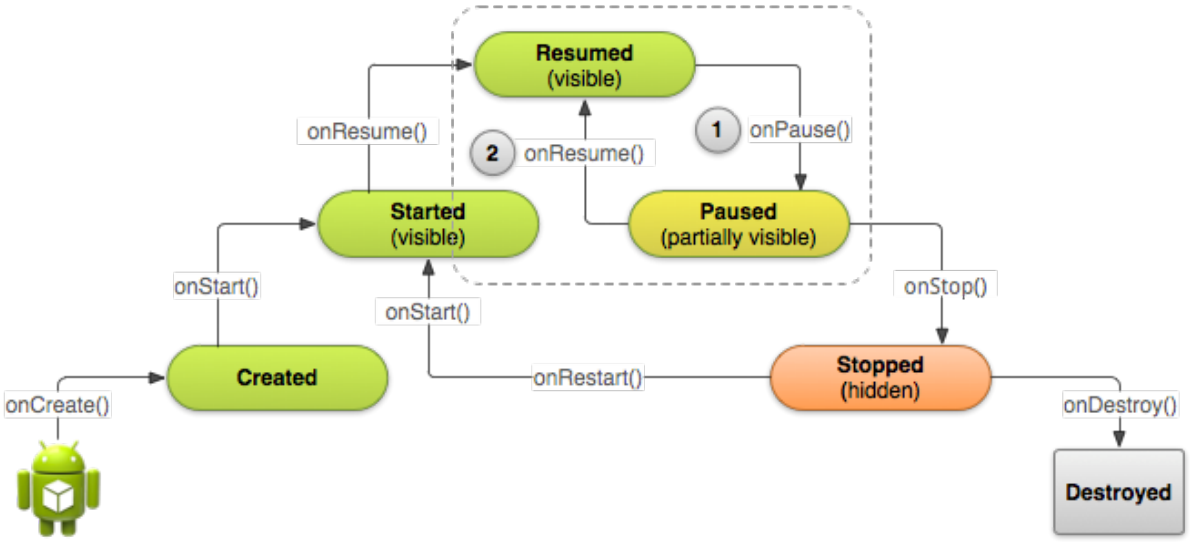
It is during this stage you should perform some fundamental setup for the activity, such as declaring the user interface (*defined in an XML layout file*), defining member variables, and configuring some of the UI.




Once the ***onCreate()*** finishes execution, the system calls the ***onStart()*** and ***onResume()*** methods in quick succession. Your activity never resides in the *Created* or *Started* states.

## 3.2 Pausing & Resuming

During normal app use, the foreground activity is sometimes obstructed by other visual components that cause the activity to *pause*. For example, when a semi-transparent activity opens (*such as one in the style of a dialog*), the previous activity pauses. As long as the activity is still partially visible but currently not the activity in focus, it remains *paused*.




When the system calls ***onPause()*** for your activity, it technically means your activity is still partially visible, but most often is an indication that the user is leaving the activity and it will soon enter the *Stopped* state. You should usually use the ***onPause()*** callback to:

- Stop animations or other ongoing actions that could consume CPU.
- Commit unsaved changes, but only if users expect such changes to be permanently saved when they leave (*such as a draft email*).
- Release system resources that may affect battery life while your activity is paused and the user does not need them.

You should avoid performing CPU-intensive work during ***onPause()***, such as writing to a database, because it can slow the visible transition to the next activity (*you should instead perform heavy-load shutdown operations during* ***onStop()***).

When your activity is *paused*, the Activity instance is kept resident in memory and is recalled when the activity resumes. You don't need to re-initialize components that were created during any of the callback methods leading up to the *Resumed* state.

When the user resumes your activity from the *Paused* state, the system calls the ***onResume()*** method.

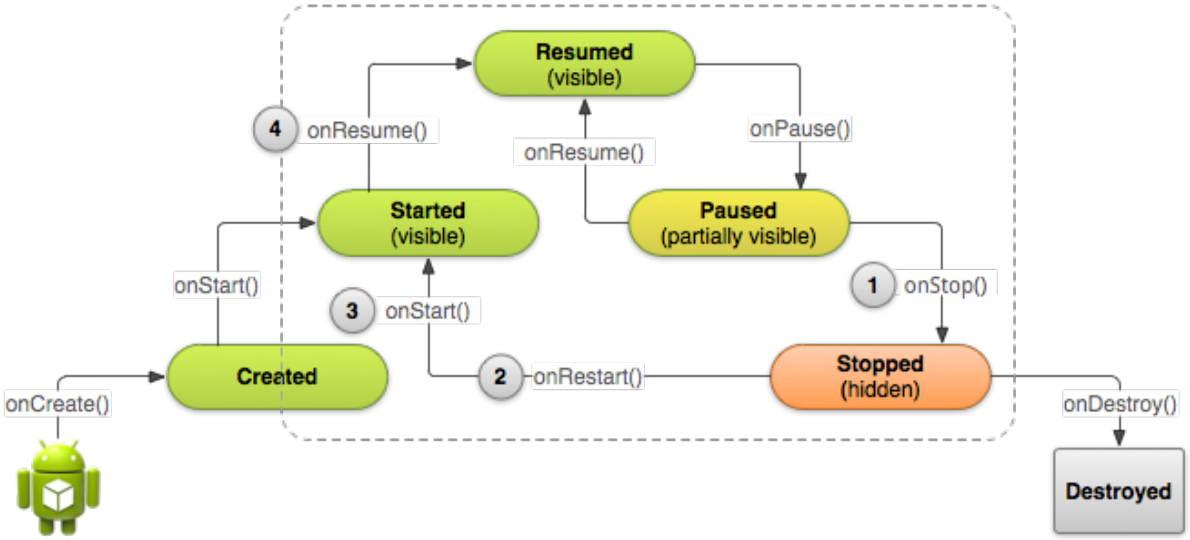
Be aware that the system calls this method every time your activity comes into the foreground, including when it's created for the first time. So, you should implement ***onResume()*** to initialize components that you release during ***onPause()***.

## 3.3 Stopping & Restarting

Properly stopping and restarting your activity is an important process in the activity lifecycle that ensures your users perceive that your app is always alive and doesn't lose their progress. There are a few key scenarios in which your activity is stopped and restarted:

- The user opens the Recent Apps window and switches from your app to another app. The activity in your app that's currently in the foreground is stopped. If the user returns to your app from the Home screen launcher icon or the Recent Apps window, the activity restarts.
- The user performs an action in your app that starts a new activity. The current activity is stopped when the second activity is created. If the user then presses the Back button, the first activity is restarted.
- The user receives a phone call while using your app on his or her phone.

The **Activity** class provides two lifecycle methods, ***onStop()*** and ***onRestart()***, which allow you to specifically handle how your activity handles being *stopped* and *restarted*. Unlike the *paused* state, which identifies a partial UI obstruction, the *stopped* state guarantees that the UI is no longer visible and the user's focus is in a separate activity (*or an entirely separate app*).




When the user leaves your activity, the system calls ***onStop()*** to stop the activity (*1*). If the user returns while the activity is *stopped*, the system calls ***onRestart()*** (*2*), quickly followed by ***onStart()*** (*3*) and ***onResume()*** (*4*). Notice that no matter what scenario causes the activity to stop, the system always calls ***onPause()*** before calling ***onStop()***.

When your activity receives a call to the ***onStop()*** method, it's no longer visible and should release almost *all* resources that aren't needed while the user is not using it. Once your activity is *stopped*, the system might destroy the instance if it needs to recover system memory. In extreme cases, the system might simply *kill* your app process without calling the activity's final ***onDestroy()*** callback, so it's important you use ***onStop()*** to release resources that might leak memory.

When your activity comes back to the foreground from the *stopped* state, it receives a call to ***onRestart()***. The system also calls the ***onStart()*** method, which happens every time your activity becomes visible (*whether being restarted or created for the first time*). The ***onRestart()*** method, however, is only called when the activity *resumes* from the *stopped* state, so you can use it to perform special restoration work that might be necessary only if the activity was previously stopped, but not destroyed.

## 4. Saving & Restoring State

There are a few scenarios where your activity is *destroyed* due to normal app behaviour, such as when the user presses the Back button, you call ***finish()***, the user rotates the screen, the system destroys the activity if it's currently *stopped* and hasn't been used in a long time, or the foreground activity requires more resources so the system shuts down processes to recover memory.

Depending upon app behaviour, the system calls the ***onSaveInstanceState()*** and ***onRestoreInstanceState()*** callback methods to save and restore "instance state" which is a collection of key-value pairs stored in a **Bundle** object.

When the screen changes orientation, the system *destroys* and *recreates* the foreground activity because the screen configuration has changed and your activity might need to load alternative resources (*such as the layout*).

By default, the system uses the **Bundle** instance state to save information about each **View** object in your activity layout (*such as the text value entered into an **EditText** object*). So, if your activity instance is destroyed and recreated, the state of the layout is restored to its previous state with no code required by you. However, your activity might have more state information that you'd like to restore, such as member variables that track the user's progress in the activity.


As the system begins to stop your activity, it calls ***onSaveInstanceState()*** (*1*) so you can specify additional state data you'd like to save in case the Activity instance must be recreated. If the activity is *destroyed* and the same instance must be recreated, the system passes the state data defined at (*1*) to both the ***onCreate()*** method (*2*) and the ***onRestoreInstanceState()*** method (*3*).

To save additional state information for your activity, you must implement ***onSaveInstanceState()*** and add key-value pairs to the **Bundle** object:

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle outState) {
    // Save the user's current game state
    outState.putInt(STATE_SCORE, mCurrentScore);
    outState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

Using ***onCreate()*** to restore some state data:

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

Instead of restoring the state during ***onCreate()*** you may choose to implement ***onRestoreInstanceState()***, which the system calls after the ***onStart()*** method, only if there is a saved state to restore, so you do not need to check whether the **Bundle** is null:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```