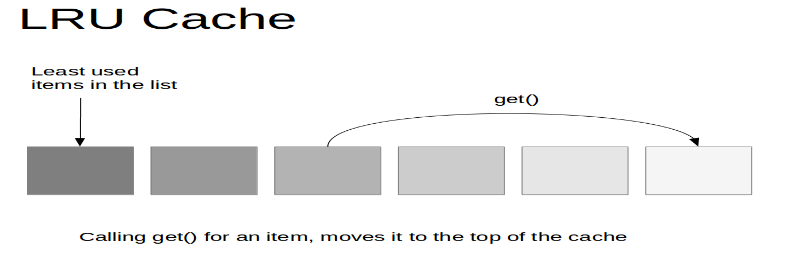
[Managing the application life cycle](http://www.vogella.com/tutorials/AndroidLifeCycle/article.html" \l "managing-the-application-life-cycle)

To manage limited system resources the Android system can terminate running applications. Each application is started in a new process with a unique ID under a unique user. If the Android system needs to free up resources it follows a simple set of rules.

| Table 1. Priorities | | |
| --- | --- | --- |
| **Process status** | **Description** | **Priority** |
| Foreground | An application in which the user is interacting with an activity, or which has a service which is bound to such an activity. Also if a service is executing one of its lifecycle methods or a broadcast receiver which runs its onReceive() method. | 1 |
| Visible | User is not interacting with the activity, but the activity is still (partially) visible or the application has a service which is used by an inactive but visible activity. | 2 |
| Service | Application with a running service which does not qualify for 1 or 2. | 3 |
| Background | Application with only stopped activities and without a service or executing receiver. Android keeps them in a least recent used (LRU) list and if requires terminates the one which was least used. | 4 |
| Empty | Application without any active components. | 5 |

All processes in the process list are added to a *least recently used* list (LRU list). The processes which are at the end of this lists will be the ones killed by the out-of-memory killer. If an application is restarted by the user, its gets moved to the top of this queue, if it reaches the lowest priority again, as indicated by the following graphic.



**Application object is an object whose lifecycle is same as our Application.**

When Application starts this object is created automatically by Android Framework. The **Application object** is already the first components started. It is also always the last component of the application, which is terminated.

When you want to share data between more than one activities of your application or you want to retain something at application level. In that scenario you can use Application object as a global store for your application.

**Note:** *There is normally no need to subclass Application. In most situations, static singletons can provide the same functionality in a more modular way. If your singleton[[1]](#footnote-1) needs a global context (for example to register broadcast receivers), include Context.getApplicationContext() as a Context argument when invoking your singleton's getInstance() method.*

To detect when an Android app goes to the background and come back to the foreground Using Application.ActivityLifecycleCallbacks and ComponentCallbacks2

You can specify a custom **Application** class in your Android manifest file.

This object provides the following main life-cycle methods:

* onCreate() - called before the first components of the application starts
* onLowMemory() - called when the Android system requests that the application cleans up memory
* onTrimMemory() - called when the Android system requests that the application cleans up memory. This message includes an indicator in which position the application is. For example, the constant TRIM\_MEMORY\_MODERATE indicates that the process is around the middle of the background LRU list; freeing memory can help the system keep other processes running later in the list for better overall performance.

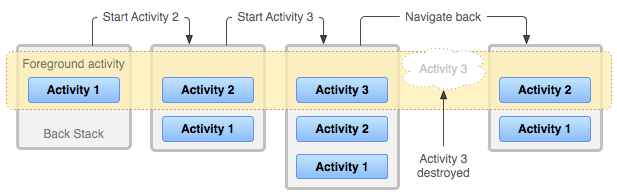
# Activity life cycle

In this lesson we will see:

* States of an Activity
* The live cycle methods
* An example
* Declaring the app’s launcher in the manifest
* Implementing the activity lifecycle callback

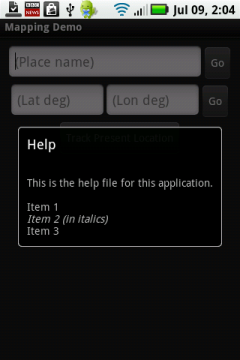
An Android application comprises one or more activities.

Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.



An activity has essentially four states:

* If an activity is in the foreground of the screen (at the top of the stack), it is *active* or *running*.
* If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus on top of your activity), it is *paused*. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can be killed by the system in extreme low memory situations.

 Consider the following case:

Here we see both activities at the same time. The first activity with the fields is obscured by another activity, and the user can no longer interact with it. However, it is still visible with all the resulting consequences.

* If an activity is completely obscured by another activity, it is *stopped*. It still retains all state and member information however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
* If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

## States of an activity

An activity is in different states, depending how it interacts with the user:

|  |  |
| --- | --- |
| Running | Activity is visible and interacts with the user. |
| Paused | Activity is still visible but partially obscured, instance is running but might be killed by the system. |
| Stopped | Activity is not visible, instance is running but might be killed by the system. |
| Killed | Activity has been terminated by the system of by a call to its finish() method. |

## The live cycle methods

To navigate transitions between stages of the activity lifecycle, the Activity class provides a core set of seven callbacks: onCreate(),onRestart(),onStart(), onResume(), onPause(), onStop(), and onDestroy(). The system invokes each of these callbacks as an activity enters a new state.

|  |  |
| --- | --- |
| onCreate() | Called then the activity is created.  Used to initialize the activity, for example create the user interface. |
| OnRestart() | Called after your activity has been stopped, prior to it being started again.  Always followed by onStart() |
| [onStart()](https://developer.android.com/reference/android/app/Activity.html#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. |
| onResume() | Called if the activity gets visible again and the user starts interacting with the activity again.  Used to initialize fields, register listeners, bind to services, etc. |
| onPause() | Called once another activity gets into the foreground. Always called before the activity is not visible anymore.  Used to release resources or save application data. For example, you unregister listeners, intent receivers, unbind from services or remove system service listeners. |
| onStop() | Called once the activity is no longer visible. Time or CPU intensive shut-down operations, such as writing information to a database should be down in the onStop() method. This method is guaranteed to be called as of API 11. |

The flow of these methods is depicted in the following diagram.

## Implementing the activity lifecycle callbacks

public class ExampleActivity extends Activity {

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

// *The activity is being created.*

}

@Override

protected void onRestart() {

super.onRestart();

// *The activity is coming from a stop state.*

}

@Override

protected void onStart() {

super.onStart();

// *The activity is about to become visible.*

}

@Override

protected void onResume() {

super.onResume();

// *The activity has become visible (it is now "resumed").*

}

@Override

protected void onPause() {

super.onPause();

// *Another activity is taking focus (this activity is about to be "paused").*

}

@Override

protected void onStop() {

super.onStop();

// *The activity is no longer visible (it is now "stopped")*

}

@Override

protected void onDestroy() {

super.onDestroy();

// *The activity is about to be destroyed.*

}

}

Using the previous application “The First Application”, add in the Main Activity the following codes one by one and execute the application. Use the Toast utility to display a message[[2]](#footnote-2)

@Override

protected void onCreate(){

super.onCreate(Bundle savedInstanceState);

Toast.*makeText*(getApplicationContext(), **"onCreate"**+

**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show(); }

@Override

**protected void** onRestart(){

**super**.onRestart();

Toast.*makeText*(getApplicationContext(), **"onStart "**+

**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();

}

@Override  
**protected void** onStart() {  
 **super**.onStart();  
 *// The activity is about to become visible.* Toast.*makeText*(getApplicationContext(), **"onStart "**+

**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();  
  
}

@Override  
**protected void** onResume() {  
 **super**.onResume();  
 *//display in long period of time* Toast.*makeText*(getApplicationContext(), **"onResume "**+

**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();  
}

@Override

**protected void** onPause() {  
 **super**.onPause();  
 *// Another activity is taking focus*

*//(this activity is about to be "paused").* Toast.*makeText*(getApplicationContext(), **"onPause "**+

**this**.getLocalClassName().toString()+

**" is about to be paused"**, Toast.***LENGTH\_LONG***).show();  
}

@Override  
**protected void** onStop() {  
 **super**.onStop();  
 *// The activity is no longer visible (it is now "stopped")* Toast.*makeText*(getApplicationContext(), **"onStop "**+

**this**.getLocalClassName().toString()+**"is not longer visible"**,

Toast.***LENGTH\_LONG***).show();  
}

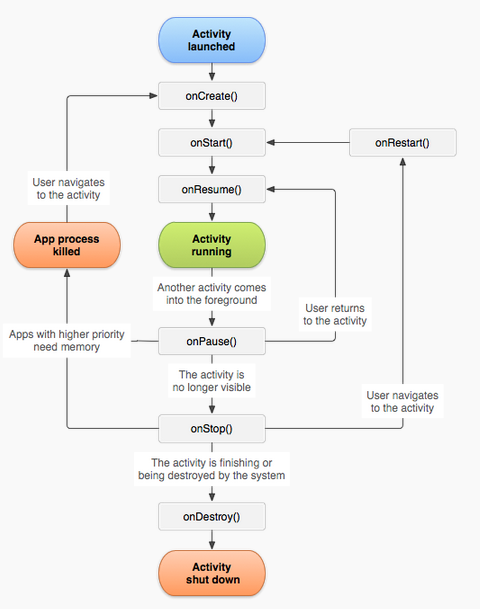
@Override  
**protected void** onDestroy() {  
 **super**.onDestroy();  
 *// The activity is about to be destroyed.* Toast.*makeText*(getApplicationContext(), **"onDestroy "**+**"Bye "**+

**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();  
}

The figure shown below shows:



* the ***back button***, which if clicked move back through app. The first activity is not destroyed, but the current activity yes;
* the ***home button***, which bring you to the home page where other apps can be launched; and
* the ***recent app button***, which presents a list of all stopped but not destroyed apps which can be clicked to bring to the foreground, or swiped which would destroy them -- in a similar manner to using the back button.
* the ***UP button***, which if clicked move back through app but after has destroyed both activities;



The figure above illustrates the life cycle. The figure captures the states and call back methods that get called as the app transitions. From the first moment an activity is created at the bottom of the pyramid each call back method (e.g., onCreate(), onStart(),onResume()`) moves the activity state up toward the top. At this point the activity is said to be in the *foreground* and the user freely interacts with it; for example, the activity transitions through *Created*, *Started* to *Resumed* state where it is *visible* and can be interacted with by the user.

Once the user is finished interacting with the app (the user starts another application), the activity moves to the background and then is destroyed or the user simply wants to start another application without destroying the current app (e.g., a navigation app). In the second case the app is not destroyed but pushed to the background by traversing down the pyramid from *Resumed* to *Paused* to *Stopped* -- at this point the application is *hidden* (not visible) from the user. The user can use the back key to navigate back to a background app and bring it back into focus, thereby moving back up the pyramid to the *Resumed* state via callback methods such as onRestart() and onResumed() -- in the case the code in the call back methods might restore the app so it could pick up where it left off.

## Declaring the app's launcher activity in the manifest

<activity android:name=".MyActivity"

android:launchMode="singleTask">

<intent-filter>

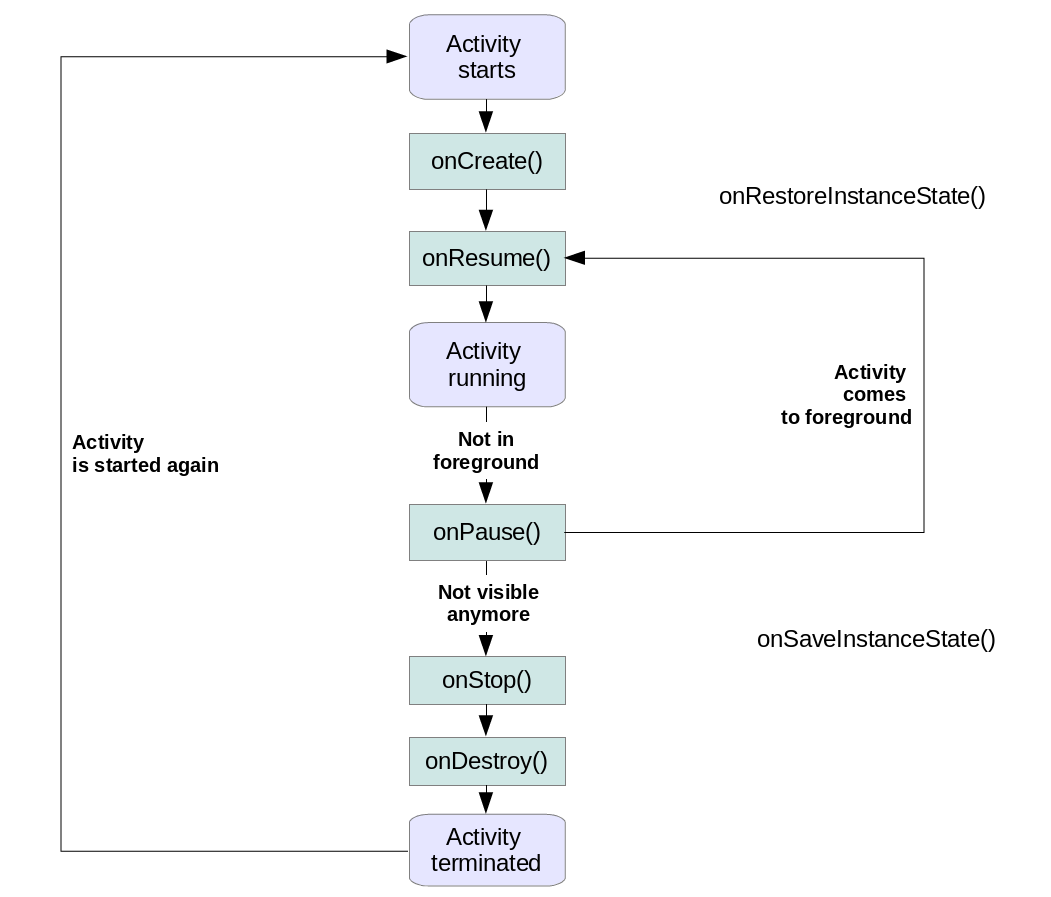
<action android:name="android.intent.action.MAIN" />

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

</intent-filter>

</activity>

## Saving and Restoring the instance state

The flow of these methods is depicted in the following diagram.

*Instance state* describes the UI state of an activity. This is non-persistent application data. It is passed between activities restarts due to a configuration change. The activity is responsible for saving and restoring its instance state.

The onSaveInstanceState() can be used to store this instance state as a Bundle. A Bundle can contain:

* primitive data types
* Strings
* objects which are of the Parcelable or Serializable type[[3]](#footnote-3)

(**Java** provides a mechanism, called object **serialization** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.)

* Arrays of the above types

The persisted Bundle data is passed at restart of the activity to the onCreate() method and onRestoreInstanceState() as parameter.

|  |  |
| --- | --- |
|  | If you override onSaveInstanceState() and onRestoreInstanceState(), you must call super. Android views store their data via a call to View.onSaveInstanceState from the onSaveInstanceState() method of the activity. For example, EditText stores its content via the default call of this method.  For this, the layout needs to define id’s for these views. |

The onRestoreInstanceState() or the onCreate() methods can be used to recreate the instance scope.

|  |  |
| --- | --- |
|  | Prefer using the onRestoreInstanceState() method for restoring the instance state. This approach separates the initial setup from restoring the state. |

If the user interacts with an activity and presses the Back button or if the finish() method of an activity is called, the activity is removed from the current activity stack and recycled. In this case there is no instance state to save and the onSaveInstanceState() method is not called.

If the user presses the Home button, the activity instance state must be saved. The onSaveInstanceState() method is called. If the user switches again to the activity and if Android terminated it, its activity stack is recreated. The saved bundle is then provided to the onRestoreInstanceState() and onCreate() methods.

@Override

protected void onRestoreInstanceState(Bundle savedInstanceState) {

super.onRestoreInstanceState(savedInstanceState);

Toast.*makeText*(getApplicationContext(), **"onSave**

**"**+**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();

}

@Override

protected void onSaveInstanceState(Bundle outState) {

super.onSaveInstanceState(outState);

Toast.*makeText*(getApplicationContext(), **"onRestore**

**"**+**this**.getLocalClassName().toString(), Toast.***LENGTH\_LONG***).show();

}

Note:

If onRestart() is called, the value of the instance variables would be maintained by the application stack itself and thus you do not need to restore them.

onCreate() method is only called when your Activity's onStop() is called and the process is killed.

## Getting a Result from an Activity and Attaching a Listener to a button

Let make some changes in TheFirstApplication project.

* In the first Application we activate a second activity from the first one. Some information is sent through an Intent to the second Intent. Let see how we can bring information from the second Activity into the first one.

From your first Activity call the second Activity using startActivityForResult() method

For example:

static final int PICK\_REQUEST = 1;// The request code

. . .

Intent i = new Intent(this, SecondActivity.class);

startActivityForResult(i, PICK\_REQUEST);

In your second Activity set the data which you want to return back to first Activity. If you don't want to return back, don't set any.

For example: In second Activity if you want to send back data:

Intent returnIntent = new Intent();

returnIntent.putExtra("result",result);

setResult(Activity.RESULT\_OK,returnIntent);

finish();

If you don't want to return data:

Intent returnIntent = new Intent();

setResult(Activity.RESULT\_CANCELED, returnIntent);

finish();

Now in your first Activity class write following code for the onActivityResult() method.

@Override

protected void onActivityResult(int requestCode, int resultCode, Intent data) {

// Check which request it is that we're responding to

if (requestCode == 1) {

  // Make sure the request was successful

if(resultCode == Activity.RESULT\_OK){

String result=data.getStringExtra("result");

//display your result in a TextView control

}

if (resultCode == Activity.RESULT\_CANCELED) {

//Write your code if there's no result

}

}

}

* Instead of using the line in button control, we can attach a listener to the button

**android:onClick="sendData"**

we can attach a listener to the button. For this we use the syntax:

. . .

**button** = (Button) findViewById(R.id.***button***);

. . .

**private void** sendData(){  
 **firstbutton**.setOnClickListener(**new** View.OnClickListener() {  
 **public void** onClick(View v) {  
 *// put in the Intent created earlier some info*

*// and start it* . . . .

}  
 });  
 }

## Resources

# <https://developer.android.com/guide/components/activities/index.ht>

# <http://www.vogella.com/tutorials/AndroidLifeCycle/article.html>

# <https://developer.android.com/guide/components/activities/activity-lifecycle.html>

# <https://www.nomtek.com/parcelable-sending-arbitrary-objects-using-intent-in-android/>

# https://code.tutsplus.com/tutorials/android-design-patterns-the-singleton-pattern--cms-29153

## Assignment

Modify the first assignment:

* Bringing some information from the second activity in the first one.
* Use listener attached to the buttons
* Overwrite the callback methods displaying just a message inside them.
* Add the methods onRestoreInstanceStatem, onSaveInstanceState, put a message in each of them.

Explain in what conditions the callback methods are called. (write this comment at each of callback method).

1. https://code.tutsplus.com/tutorials/android-design-patterns-the-singleton-pattern--cms-29153 [↑](#footnote-ref-1)
2. A toast is a view containing a quick little message for the user. The toast class helps you create and show those. [↑](#footnote-ref-2)
3. Parcelable is an interface for serialization/deserialization of Java objects used in Android [↑](#footnote-ref-3)