ACTIVITIES

COMP 41690

DAVID COYLE > D.COYLE@UCD.IE

BASIC BUILDING BLOCKS

Activities: UI building blocks

Content Providers: handling and storing data

Services: background processes

Broadcase receivers: handle system wide messages

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an **intent**. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your app or another.

THE MANIFEST FILE

Before the Android system can start an app component, the system must know that the component exists by reading the app's **AndroidManifest.xml** file.

Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as:

- Identify any user permissions the app requires, such as Internet access or read-access to the user's contacts.
- Declare the minimum API Level required by the app, based on which APIs the app uses.
- Declare hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
- API libraries the app needs to be linked against (other than the Android framework APIs), such as the Google Maps library.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
        <activity android:name="com.example.project.ExampleActivity"
                  android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

In the <application> element, the android:icon attribute points to resources for an icon that identifies the app.

In the <activity> element, the android:name attribute specifies the fully qualified class name of the Activity subclass and the android:label attributes specifies a string to use as the user-visible label for the activity.

You must declare all app components this way:
<activity> elements for activities
<service> elements for services
<receiver> elements for broadcast receivers
<provider> elements for content providers

ACTIVITIES:
THE BUILDING BLOCKS OF THE USER INTERFACE

An activity is a single, focused thing that the user can do.

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 consists of multiple activities that are loosely bound to each other.

For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails.

DECLARING AN ACTIVITY

You must declare your activity in the manifest file in order for it to be accessible to the system.

To declare your activity, open your manifest file and add an <activity> element as a child of the <application> element.

```
<manifest ... >
  <application ... >
      <activity android:name=".ExampleActivity" />
      ...
  </application ... >
  ...
</manifest >
```

The android:name attribute is the only required attribute - it specifies the class name of the activity.

SPECIFY AN APP LAUNCHER

When the user selects your app icon from the Home screen, the system calls the onCreate( ) method for the Activity that you've declared to be the "launcher" (or "main") activity. This activity is the main entry point to your app's user interface.

You define which activity to use as the main activity in the Android manifest file, AndroidManifest.xml, which is at the root of your project directory.

The main activity for your app must be declared in the manifest with an <intent-filter> that includes the MAIN action and LAUNCHER category.

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

CREATING AN ACTIVITY

To create an activity, you must create a subclass of Activity (or an existing subclass of it).

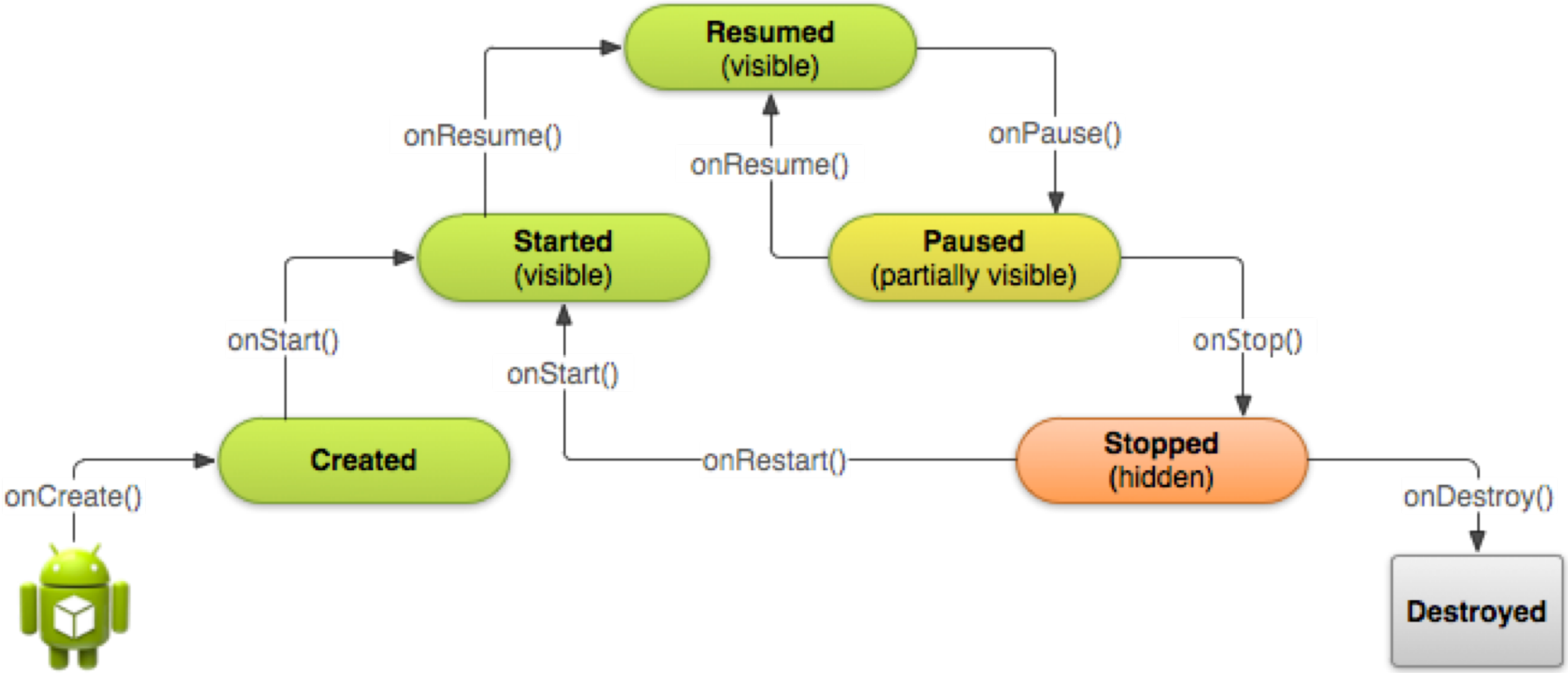
In your subclass, you need to implement *lifecycle callback methods* that the system calls when the activity transitions between states of its lifecycle, such as when the activity is being created, stopped, resumed, or destroyed.

Managing the lifecycle of your activities by implementing callback methods is crucial to developing a strong and flexible application.

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

https://developer.android.com/reference/android/app/Activity.html

THE ACTIVITY LIFECYCLE



ACTIVITIES CONTD.

Depending on the complexity of your activity, you probably don't need to implement all the lifecycle methods. However, it's important that you understand each one.

Implementing your activity lifecycle methods properly ensures your app behaves well in several ways, including that it:

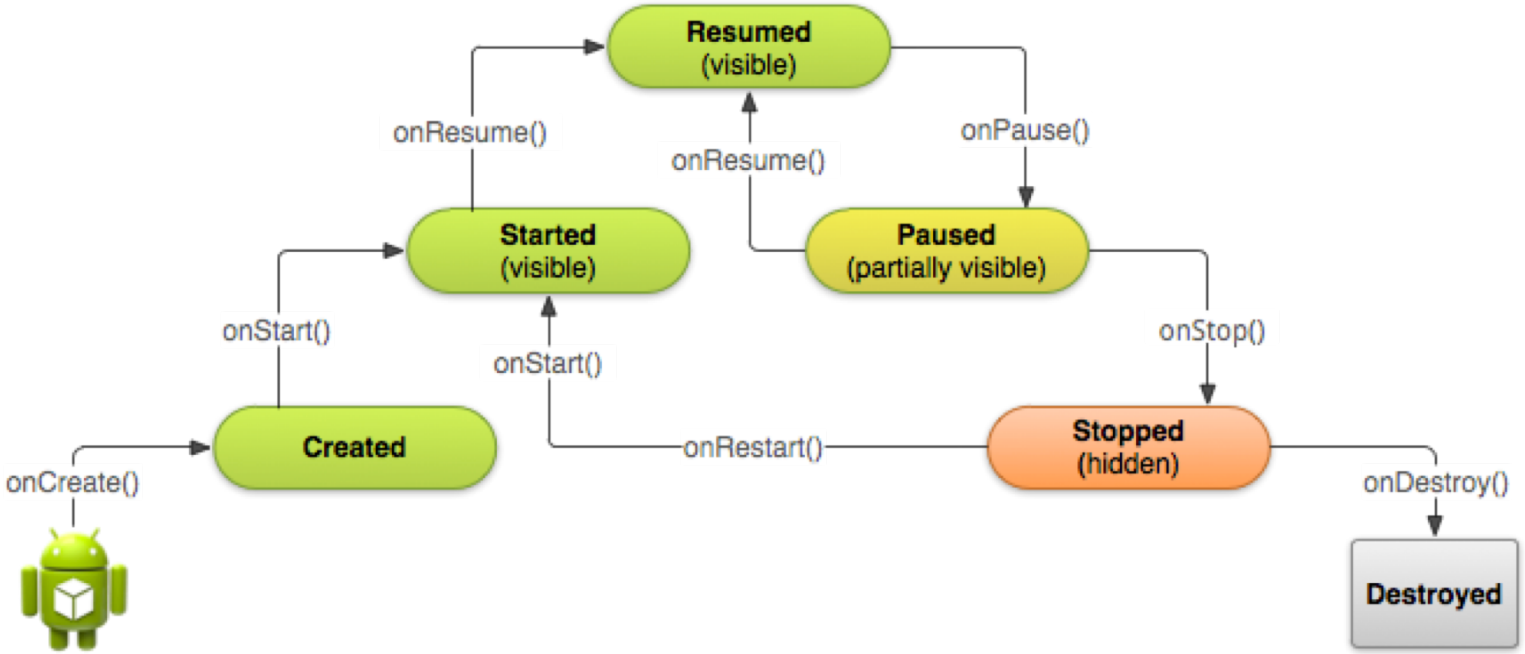
- Does not crash if the user receives a phone call or switches to another app while using your app.
- Does not consume valuable system resources when the user is not actively using it.
- Does not lose the user's progress if they leave your app and return to it at a later time.
- Does not crash or lose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

onCreate( )

You must implement this method. The system calls this when creating your activity. This is where you should initialize the essential components of your activity.

Most importantly, this is where you must call setContentView() to define the layout for the activity's user interface.

Always followed by onStart( )



```java
TextView mTextView; // Member variable for text view in the layout

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

    // Set the user interface layout for this Activity
    // The layout file is defined in the project res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // Initialize member TextView so we can manipulate it later
    mTextView = (TextView) findViewById(R.id.text_message);

    // Make sure we're running on Honeycomb or higher to use ActionBar APIs
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
        // For the main activity, make sure the app icon in the action bar
        // does not behave as a button
        ActionBar actionBar = getActionBar();
        actionBar.setHomeButtonEnabled(false);
    }
}
```

onRestart( )

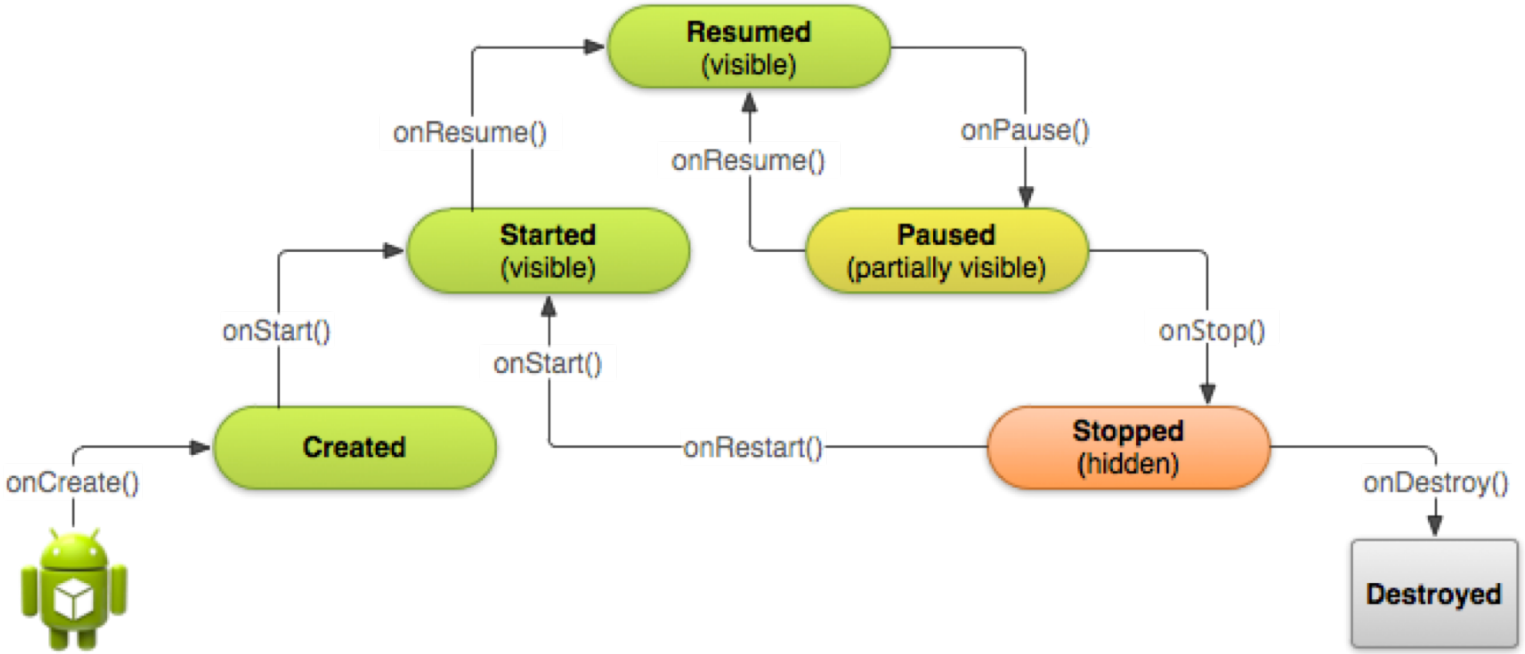
Called after your activity has been stopped, prior to it being started again.

Always followed by onStart()

onStart( )

Called just before the activity becomes visible to the user.

Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.

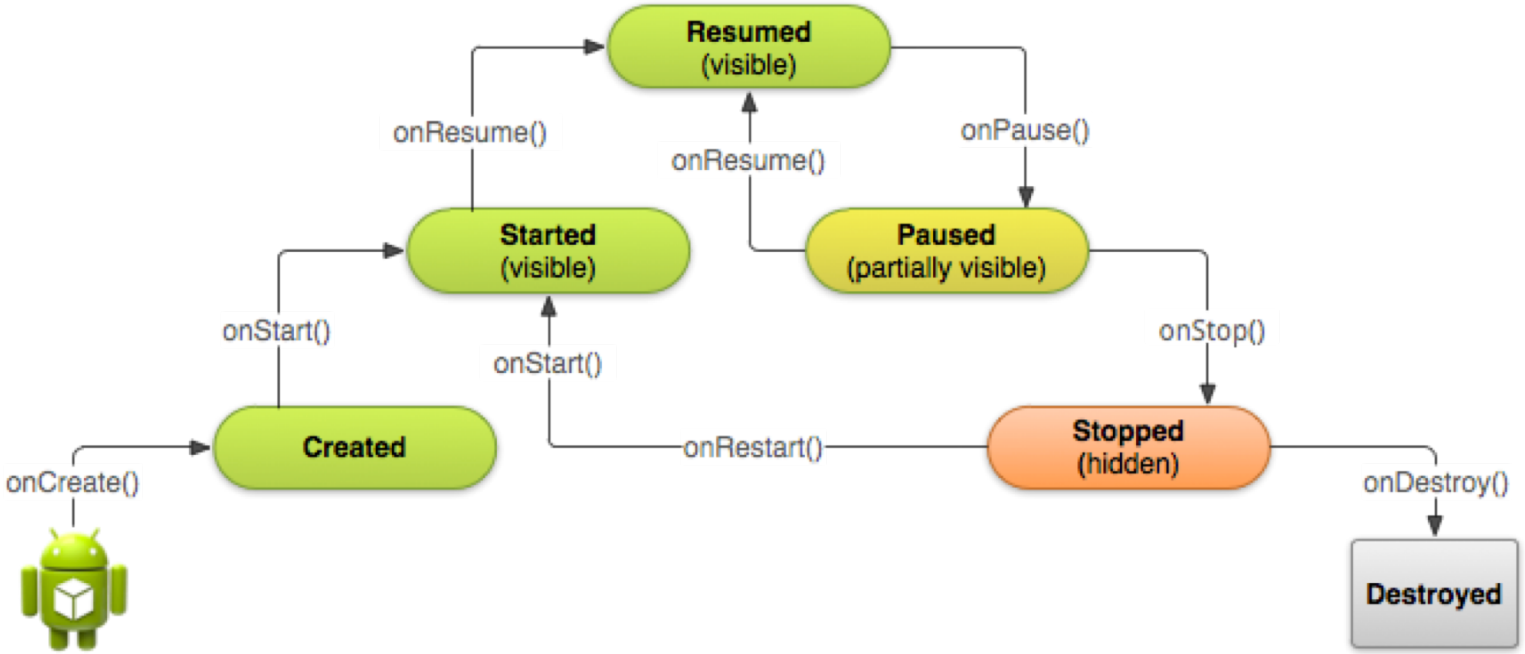


```java
@Override
protected void onStart() {
    super.onStart();  // Always call the superclass method first

    // The activity is either being restarted or started for the first time
    // so this is where we should make sure that GPS is enabled
    LocationManager locationManager =
            (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    boolean gpsEnabled = locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);

    if (!gpsEnabled) {
        // Create a dialog here that requests the user to enable GPS, and use an intent
        // with the android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS action
        // to take the user to the Settings screen to enable GPS when they click "OK"
    }
}

@Override
protected void onRestart() {
    super.onRestart();  // Always call the superclass method first

    // Activity being restarted from stopped state
}
```

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.

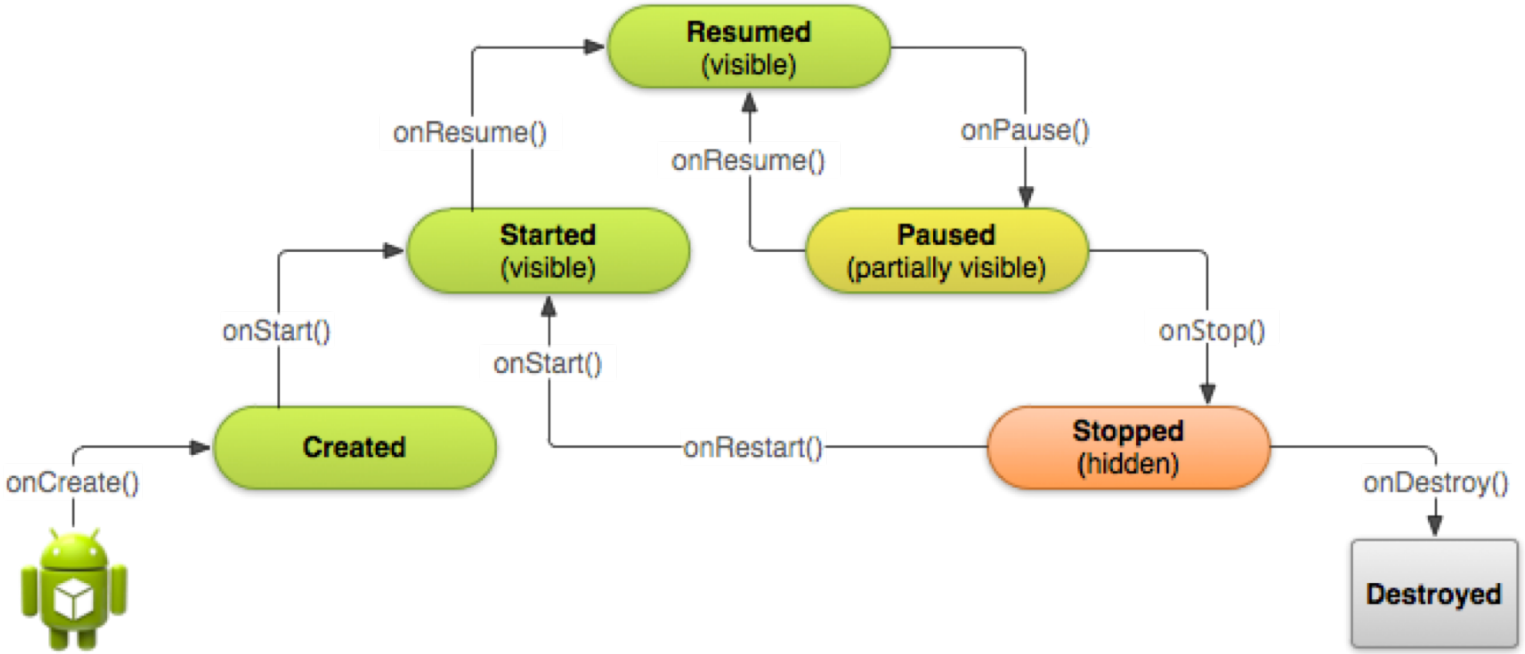
Always followed by onPause().



onPause( )

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 a dialog), the previous activity pauses.

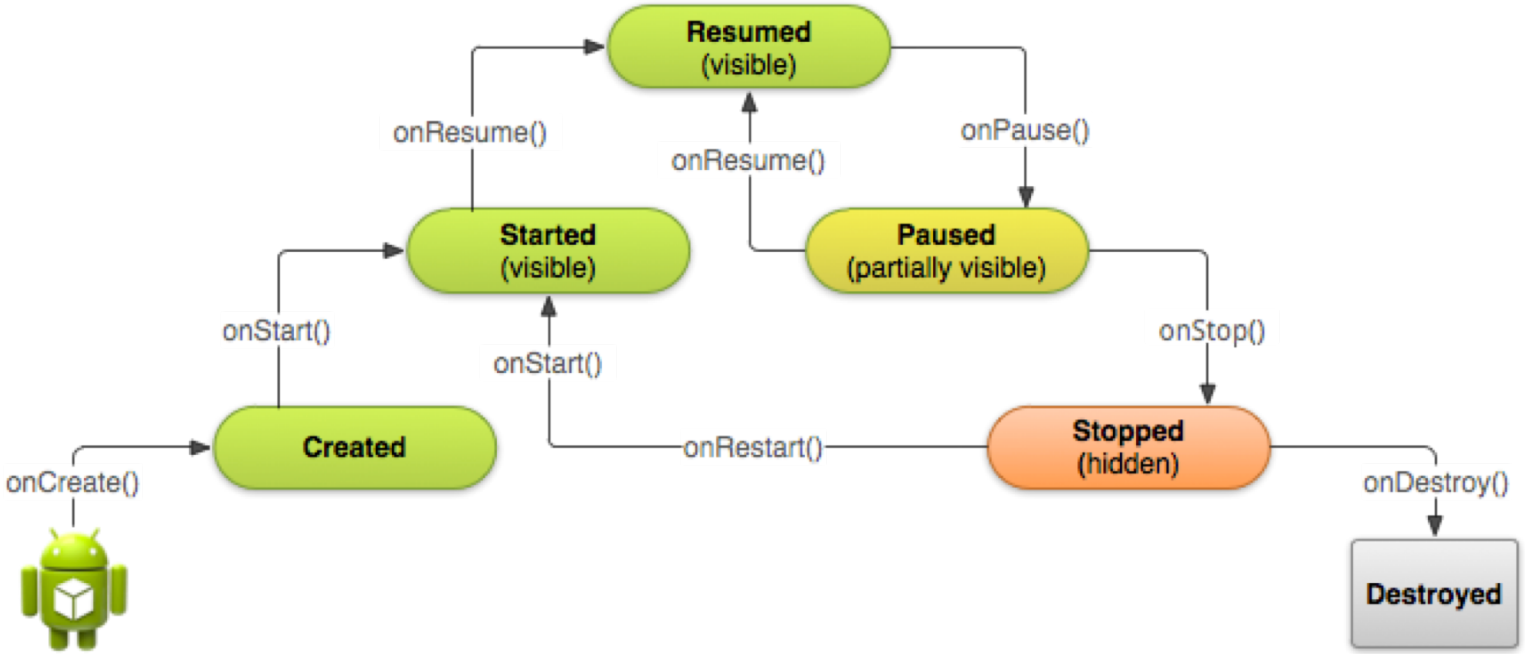
As your activity enters the paused state, the system calls the onPause() method.



onPause( )

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.

What should happen here?



onPause( ) – example actions

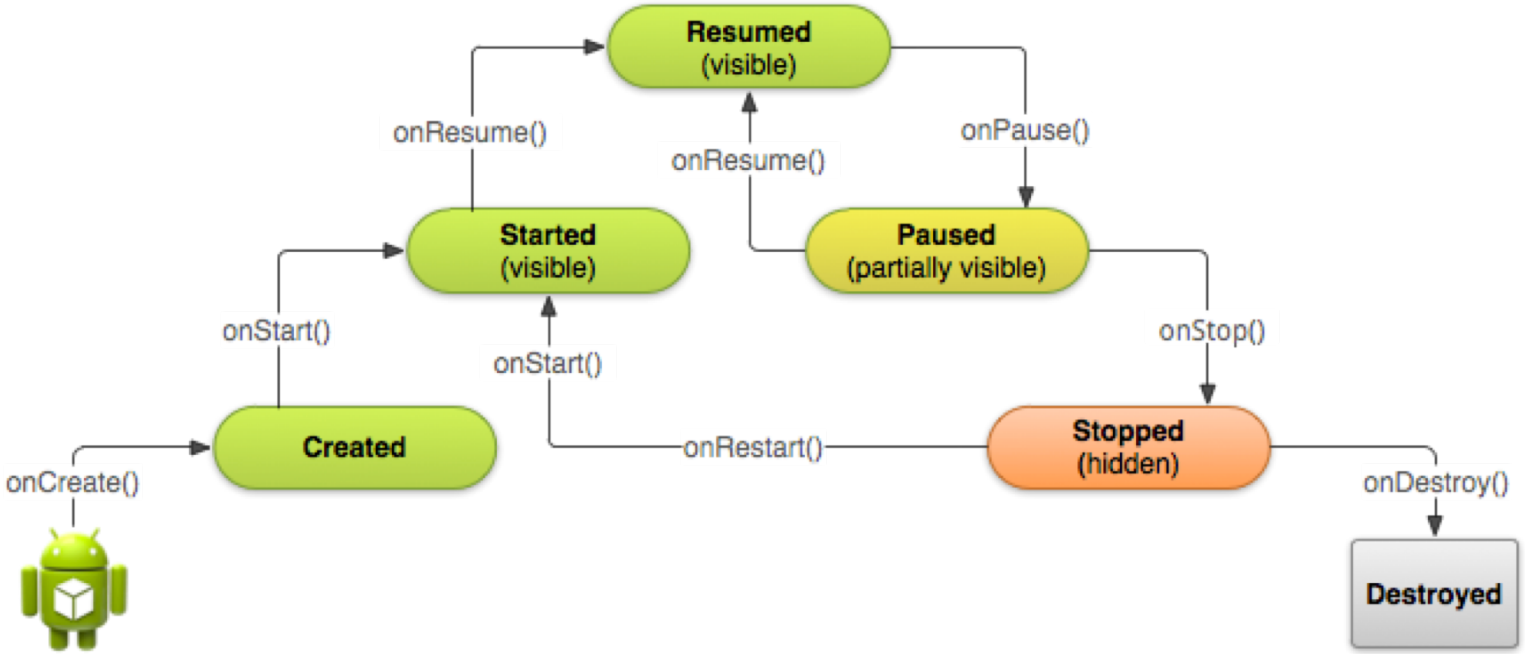
- Stop animations or other ongoing actions that could consume CPU.
- Release system resources, such as broadcast receivers, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them.
- Commit unsaved changes, but only if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

```
@Override
public void onPause() {
    super.onPause();  // Always call the superclass method first

    // Release the Camera because we don't need it when paused
    // and other activities might need to use it.
    if (mCamera != null) {
        mCamera.release();
        mCamera = null;
    }
}
```

onPause( )

Implementations of this method must be very quick because the next activity will not be resumed until this method returns.

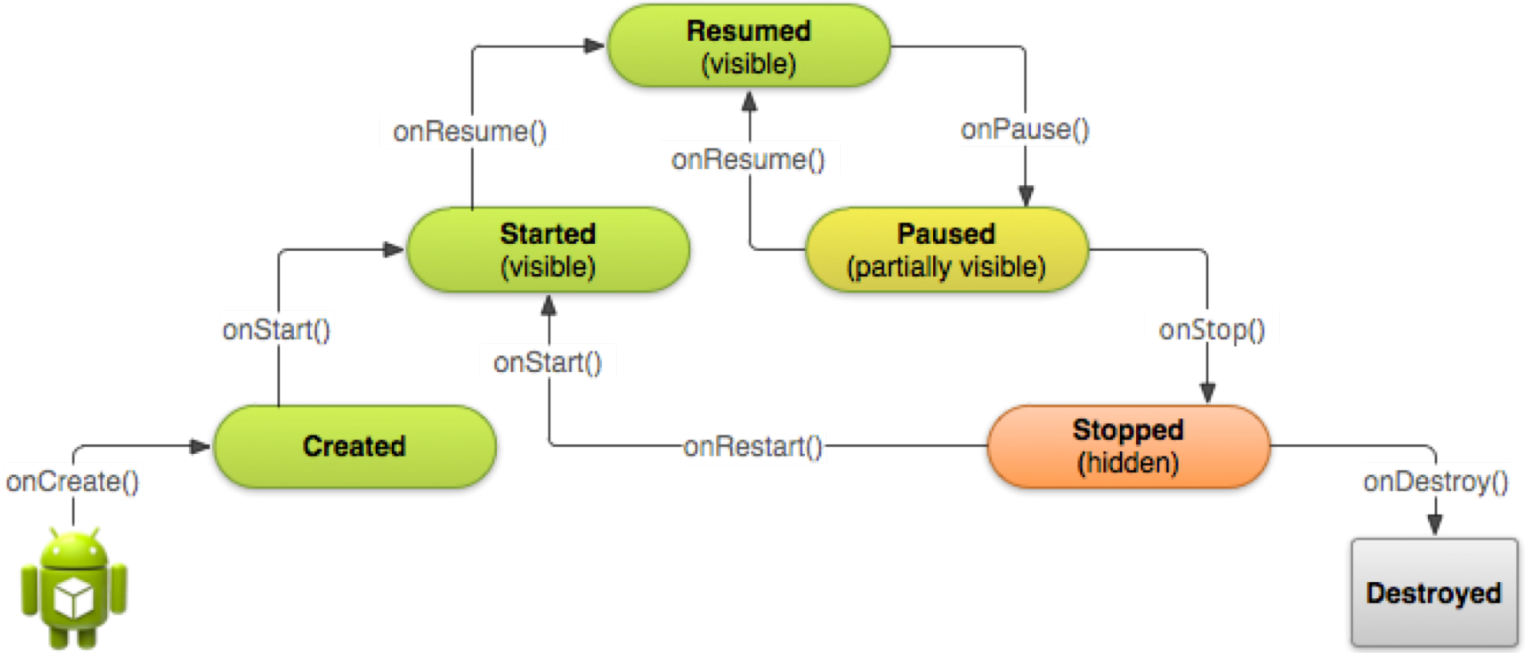
Keep the amount of operations in onPause() relatively simple in order to allow for a speedy transition to the user's next destination if your activity is actually being stopped.



onPause( )

The only time you should persist user changes to permanent storage within onPause() is when you're certain users expect the changes to be auto-saved (such as when drafting an email).

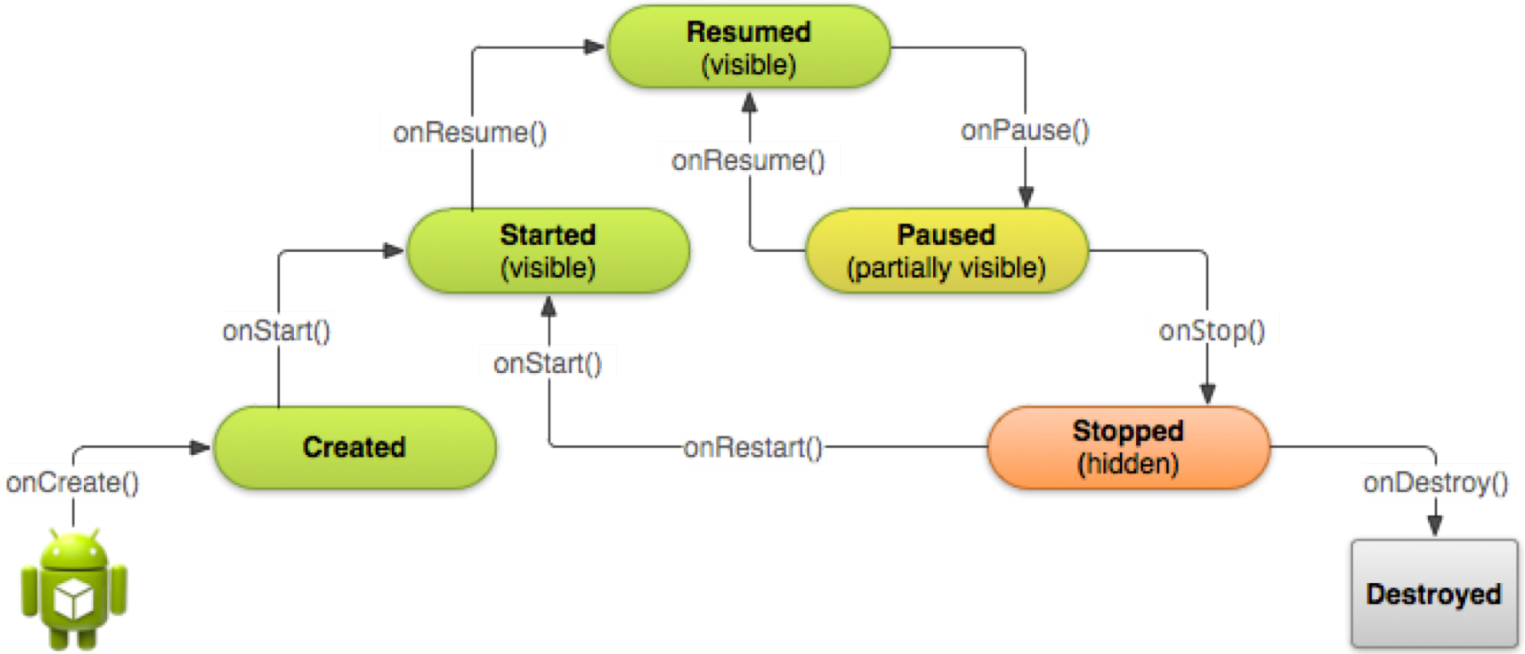
Avoid performing CPU-intensive work such as writing to a database, because *it can slow the visible transition* to the next activity (you should instead perform heavy-load shutdown operations during onStop()).



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.

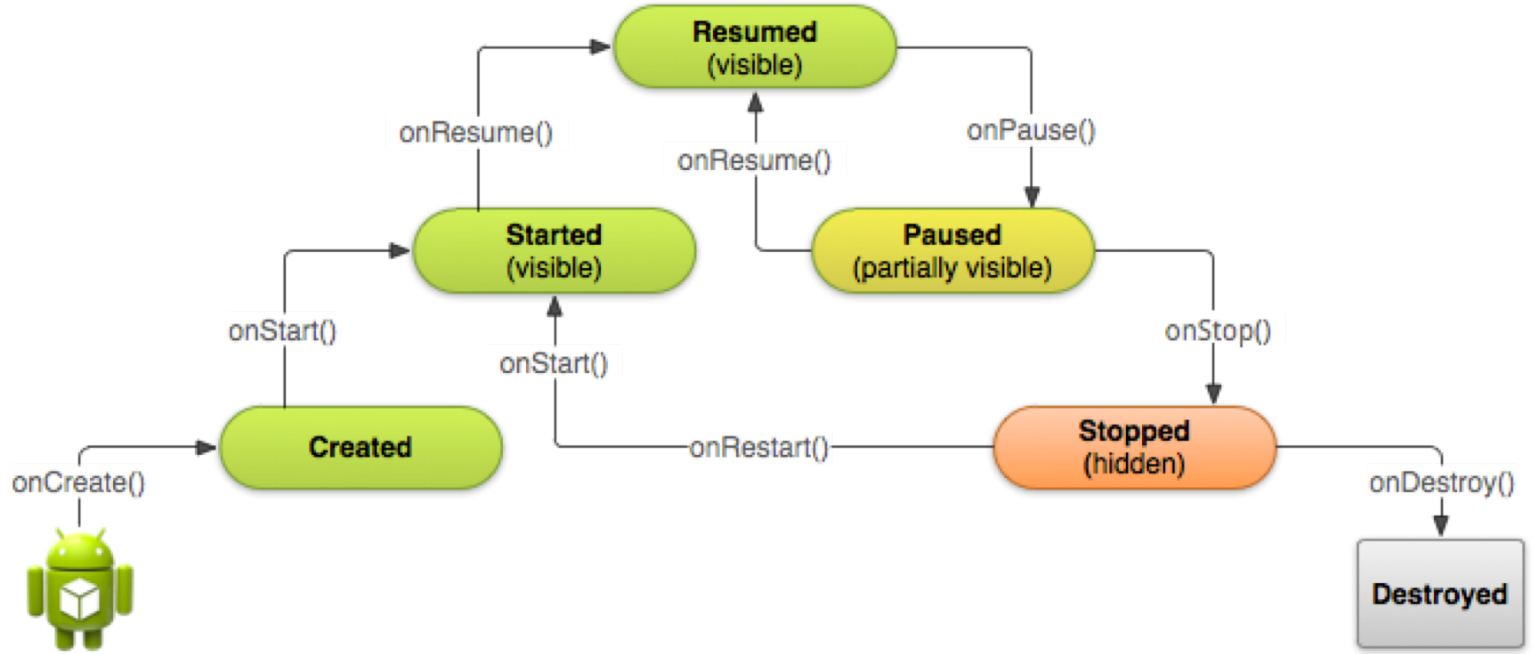


onStop( )

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.

It's important you use onStop() to release resources that might leak memory.



onStop( )

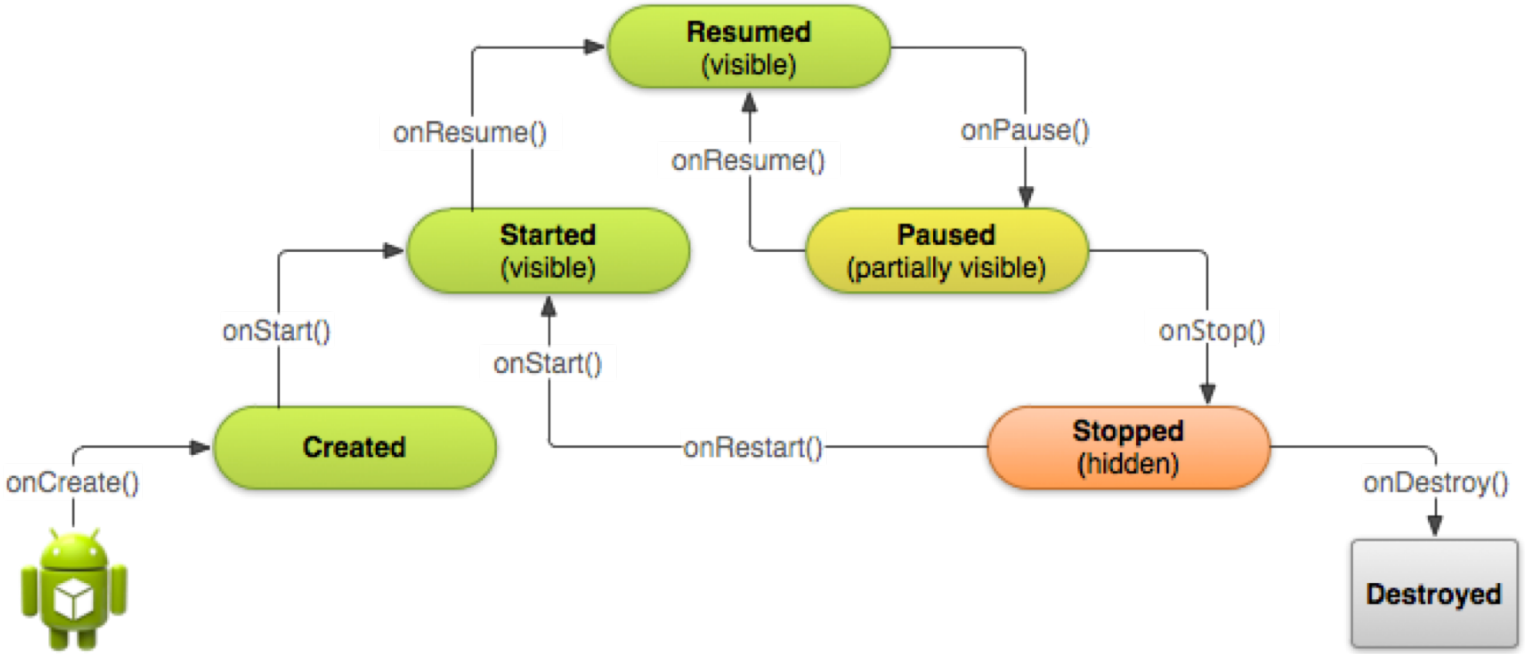
Although onPause( ) is called before onStop( ), you should use onStop( ) to perform larger, more CPU intensive shut-down operations, such as writing information to a database.

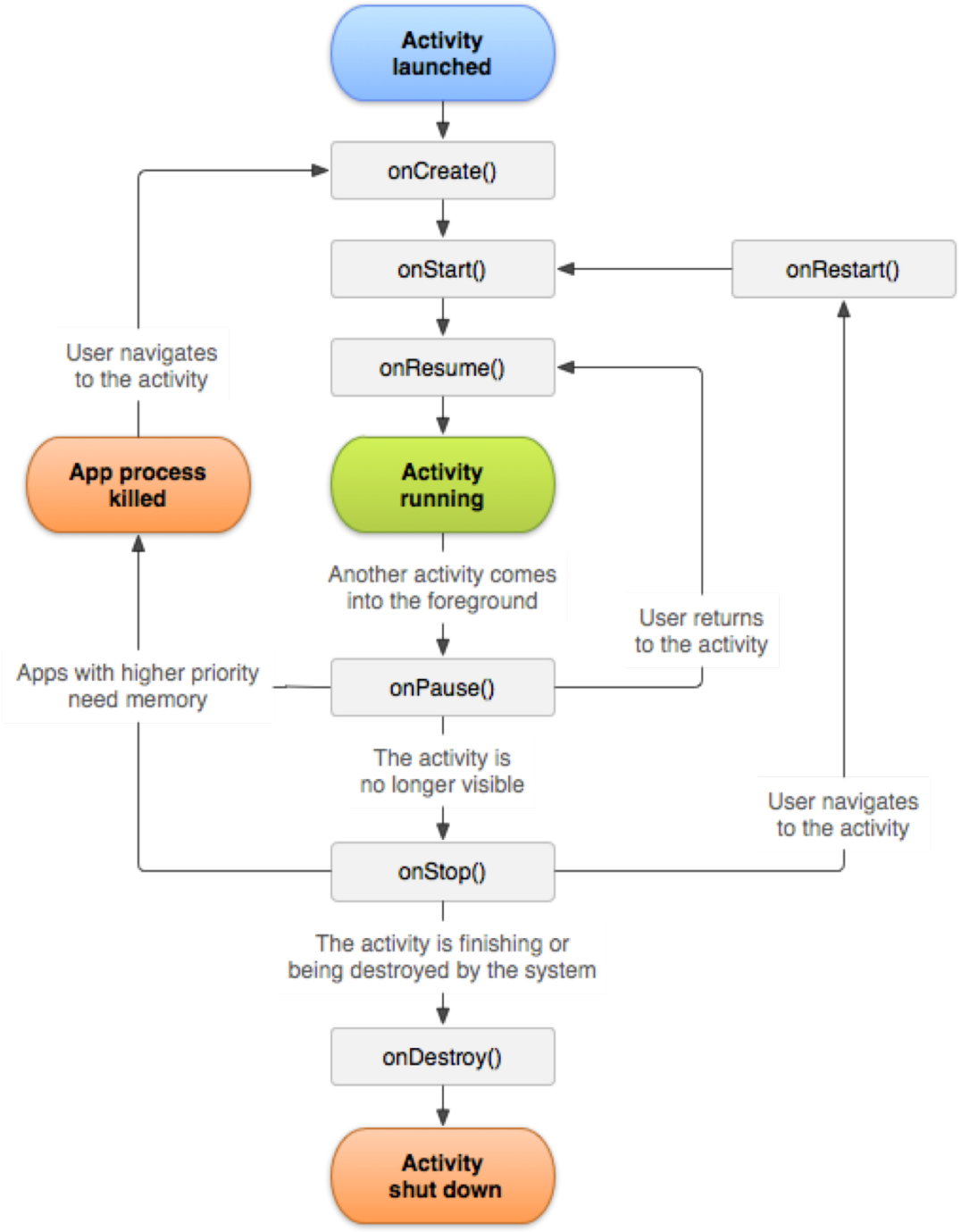
```
@Override
protected void onStop() {
    super.onStop();  // Always call the superclass method first

    // Save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE, getCurrentNoteTitle());

    getContentResolver().update(
            mUri,    // The URI for the note to update.
            values,  // The map of column names and new values to apply to them.
            null,    // No SELECT criteria are used.
            null     // No WHERE columns are used.
            );
}
```

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.





Taken together, these methods define the entire lifecycle of an activity.

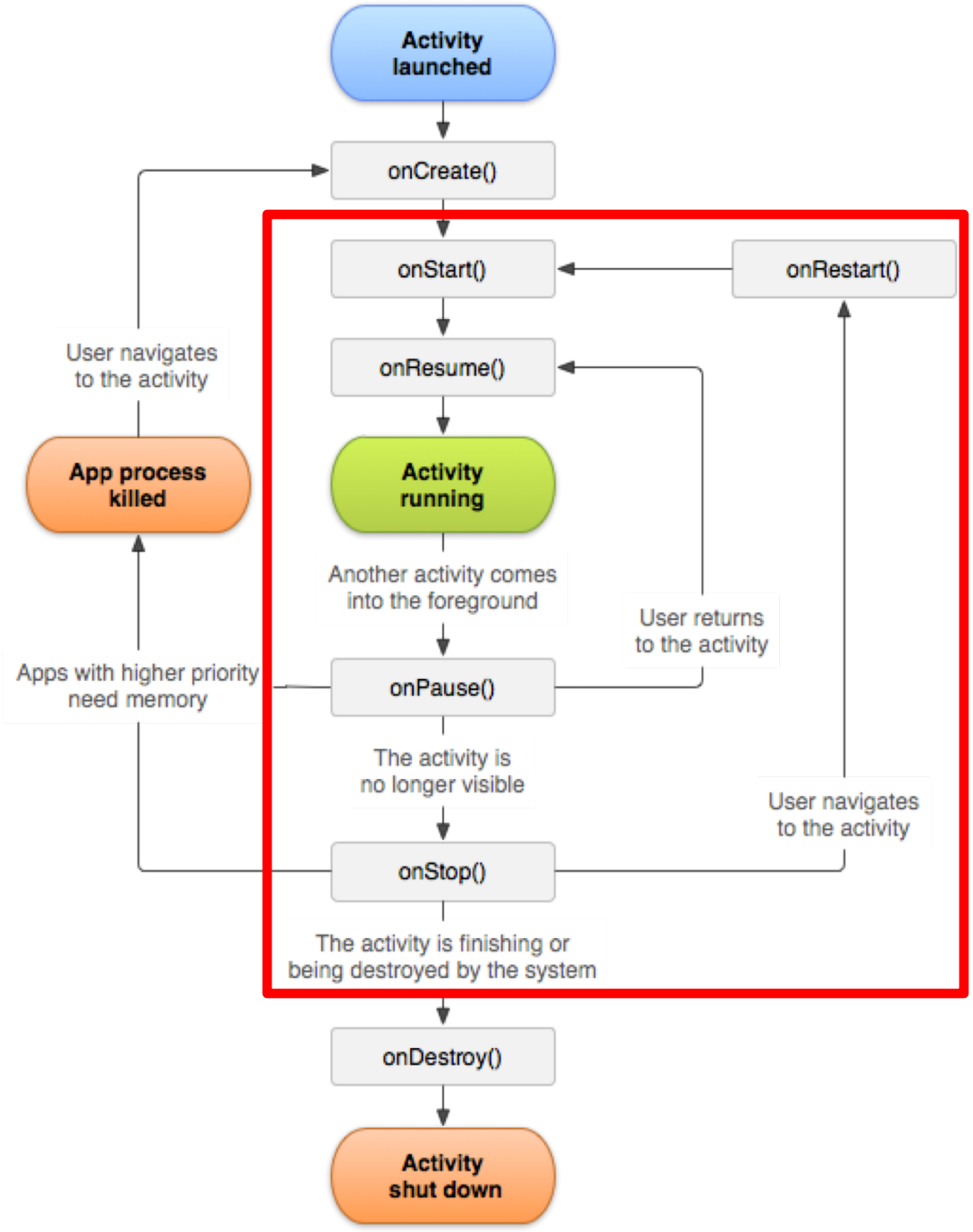
By implementing these methods, you can monitor three nested loops in the activity lifecycle:

1. **The entire lifetime**

1. **The visible lifetime**

2. **The foreground lifetime**

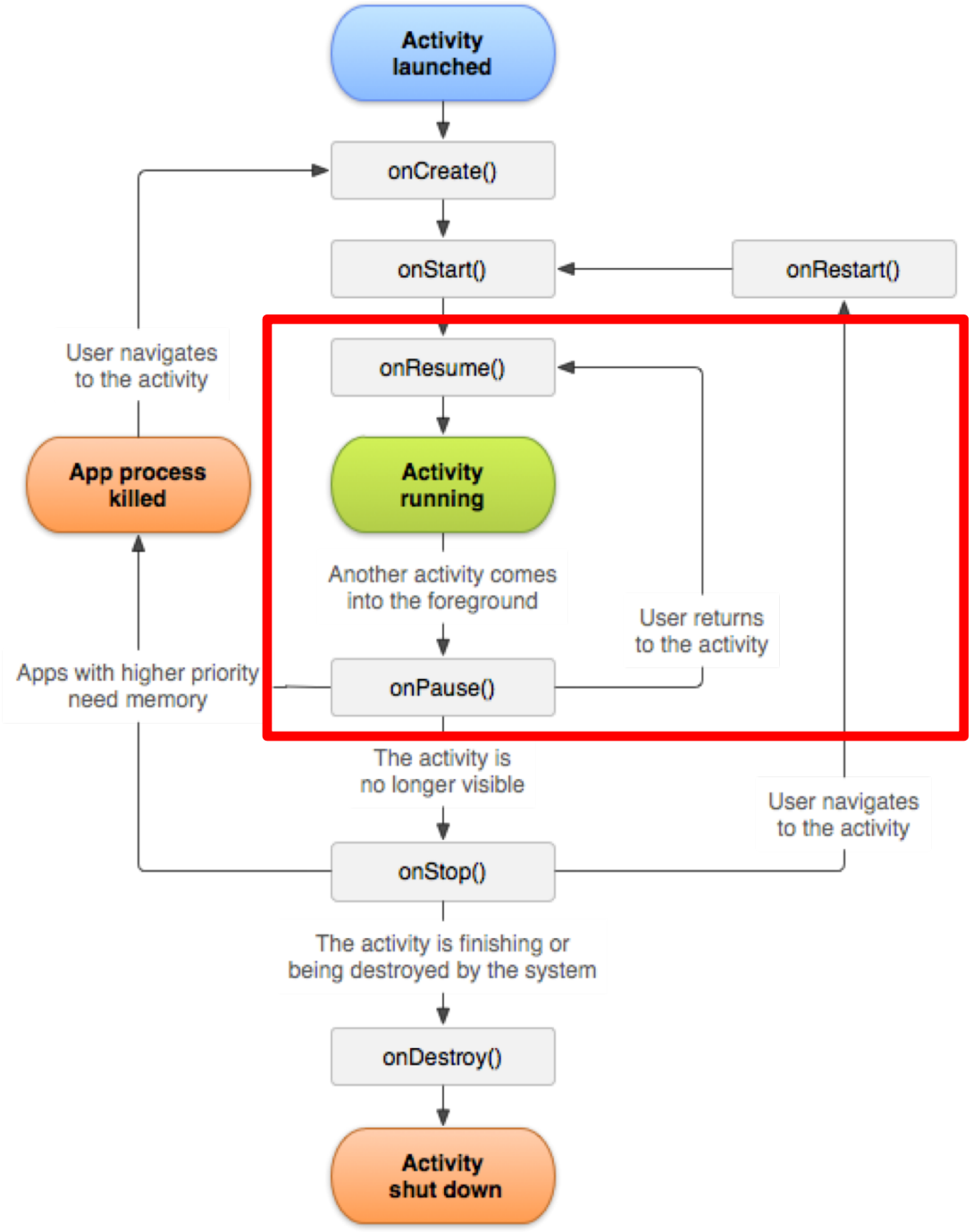
http://developer.android.com/guide/components/activities.html



Taken together, these methods define the entire lifecycle of an activity.

By implementing these methods, you can monitor three nested loops in the activity lifecycle:

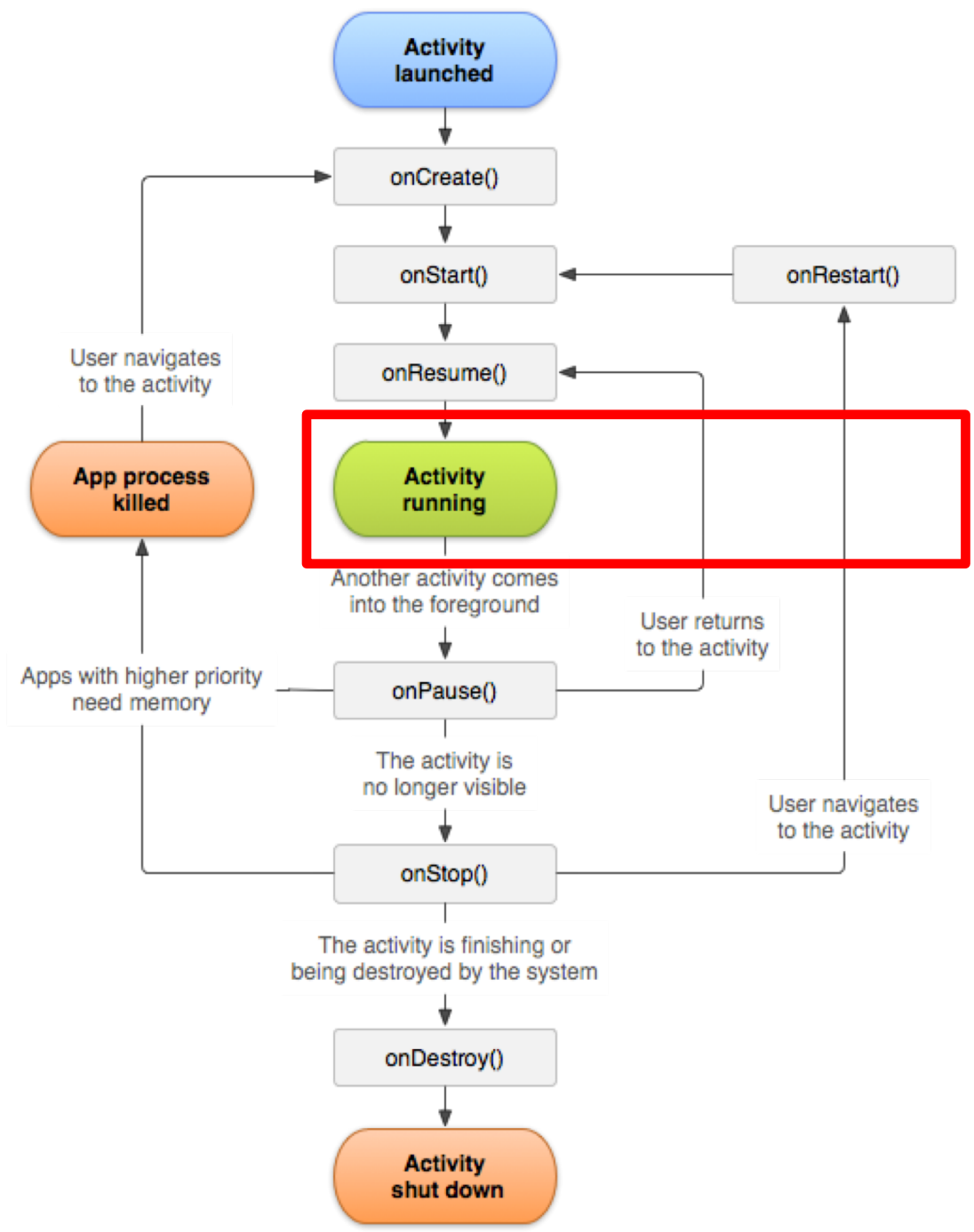
1. **The entire lifetime**
1. **The visible lifetime**
2. **The foreground lifetime**



Taken together, these methods define the entire lifecycle of an activity.

By implementing these methods, you can monitor three nested loops in the activity lifecycle:

1. **The entire lifetime**
1. **The visible lifetime**
2. **The foreground lifetime**



Taken together, these methods define the entire lifecycle of an activity.

By implementing these methods, you can monitor three nested loops in the activity lifecycle:

1. **The entire lifetime**

1. **The visible lifetime**

2. **The foreground lifetime**

The activity is in front of all other activities on screen and has user input focus.

onResume() and onPause() should be fairly lightweight in order to avoid slow transitions that make the user wait.

ACTIVITIES CONTD.

An activity can exist in essentially three states:

Resumed
The activity is in the foreground of the screen and has user focus. (This state is also sometimes referred to as "running" or "active".)

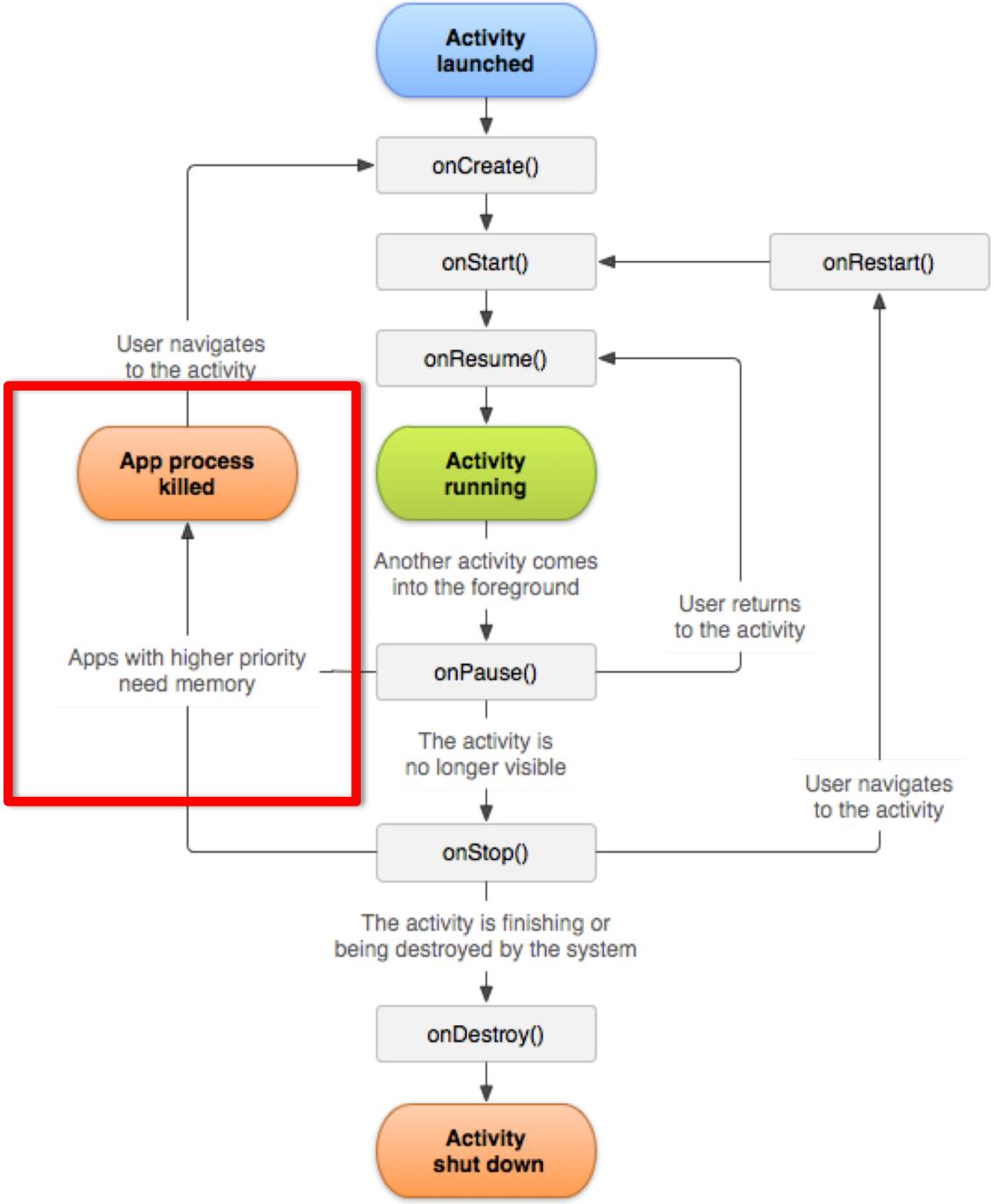
Paused
Another activity is in the foreground and has focus, but this one is still visible. E.g. another activity is visible on top of this one and that activity is partially transparent or doesn't cover the entire screen.

Stopped
The activity is completely obscured by another activity (the activity is now in the "background"). A stopped activity is also still alive, however, it is no longer visible to the user and it can be killed by the system when memory is needed elsewhere.

Note:

The Android system reserves the right to kill your app.

Your activity should perform most cleanup during onPause() and onStop()



STARTING AN ACTIVITY

You can start an activity by passing an Intent to

startActivity( )

or startActivityForResult( ) (when you want the activity to return a result).

```
Intent intent = new Intent(this, SignInActivity.class);
startActivity(intent);
```

The intent specifies either the exact activity you want to start

or can describes the type of action you want to perform, and leverage activities provide by other applications.

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

STARTING AN ACTIVITY FOR A RESULT

Sometimes, you might want to receive a result from the activity that you start.

In that case, start the activity by calling startActivityForResult()

```
private void pickContact() {
    // Create an intent to "pick" a contact, as defined by the content provider URI
    Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
    startActivityForResult(intent, PICK_CONTACT_REQUEST);
}
```

STARTING AN ACTIVITY FOR A RESULT

To then receive the result from the subsequent activity, implement the onActivityResult() callback method.

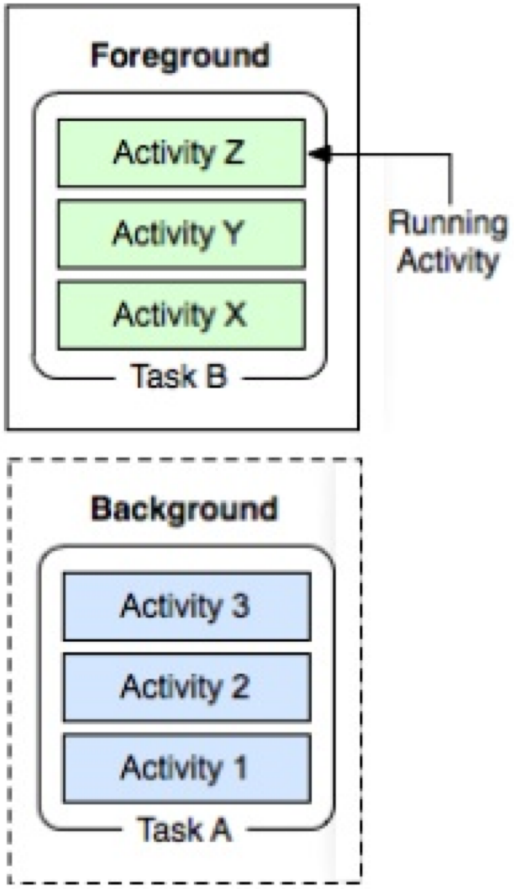
When the subsequent activity is done, it returns a result in an Intent to your onActivityResult() method.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    // If the request went well (OK) and the request was PICK_CONTACT_REQUEST
    if (resultCode == Activity.RESULT_OK && requestCode == PICK_CONTACT_REQUEST) {
        // Perform a query to the contact's content provider for the contact's name
        Cursor cursor = getContentResolver().query(data.getData(),
        new String[] {Contacts.DISPLAY_NAME}, null, null, null);
        if (cursor.moveToFirst()) { // True if the cursor is not empty
            int columnIndex = cursor.getColumnIndex(Contacts.DISPLAY_NAME);
            String name = cursor.getString(columnIndex);
            // Do something with the selected contact's name...
        }
    }
}
```

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.

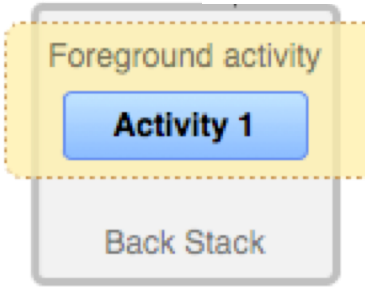


TASKS AND BACK STACK

The device Home screen is the starting place for most tasks.

When the user touches an icon in the application launcher, that application's task comes to the foreground.

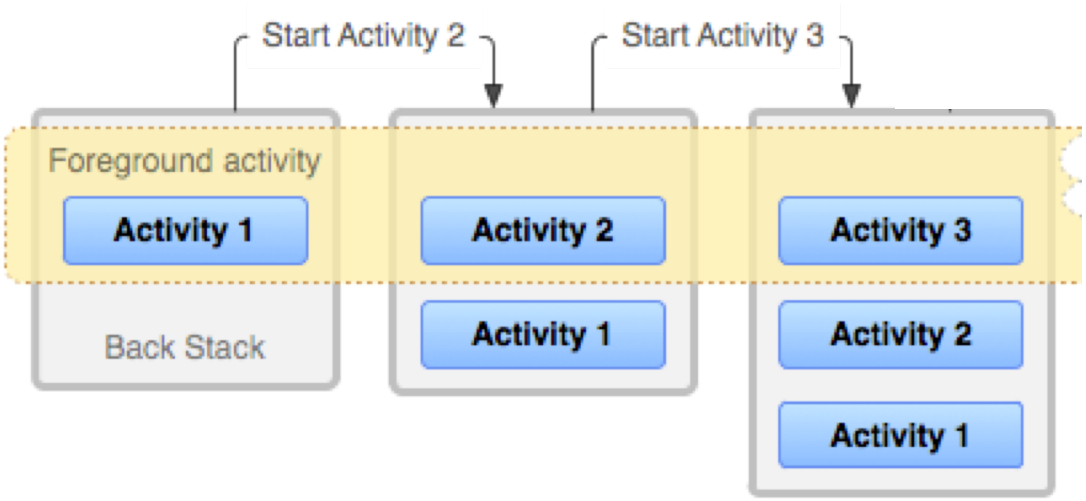
If no task exists for the application (the application has not been used recently), then a new task is created and the "main" activity for that application opens as the root activity in the stack.



TASKS AND BACK STACK

When the current activity starts another, the new activity is pushed on the top of the stack and takes focus.

The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface.

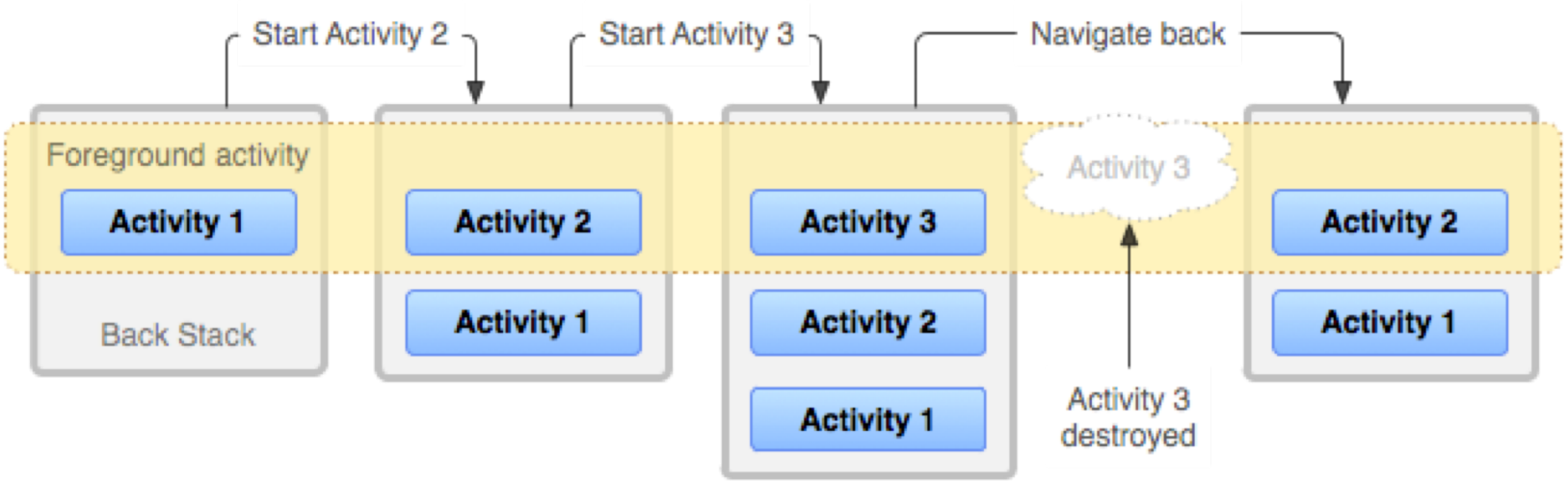


TASKS AND BACK STACK

When the current activity starts another, the new activity is pushed on the top of the stack and takes focus.

The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface.

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).



MULTITASKING

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 below.

A task can then return to the "foreground" so users can pick up where they left off.

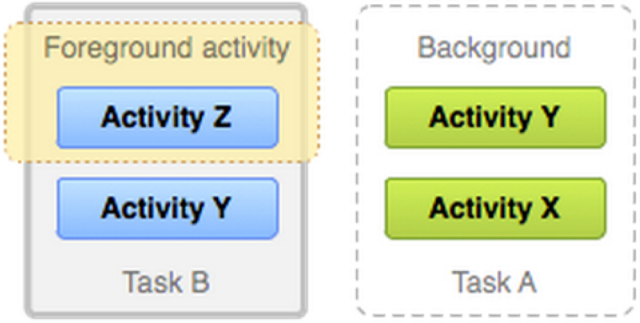


Figure 2. Two tasks: Task B receives user interaction in the foreground, while Task A is in the background, waiting to be resumed.

MULTITASKING

Note: 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.

It is important to proactively retain the state of your activities using callback methods, in case the activity is destroyed and must be recreated.

You do this by implementing the onSaveInstanceState() callback methods in your activity.

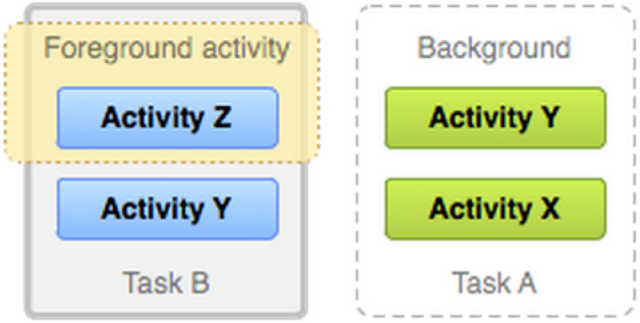


Figure 2. Two tasks: Task B receives user interaction in the foreground, while Task A is in the background, waiting to be resumed.

MANAGING TASKS

Generally you shouldn't have to worry about how your activities are associated with tasks or how they exist in the back stack.

However, you might decide that you want to interrupt the normal behavior.

You can do this with attributes in the <activity> manifest element and with flags in the intent that you pass to startActivity().

The principal <activity> attributes and intent flags you can use are listed on the right.

Attributes

- `taskAffinity`
- `launchMode`
- `allowTaskReparenting`
- `clearTaskOnLaunch`
- `alwaysRetainTaskState`
- `finishOnTaskLaunch`

Flags

- `FLAG_ACTIVITY_NEW_TASK`
- `FLAG_ACTIVITY_CLEAR_TOP`
- `FLAG_ACTIVITY_SINGLE_TOP`

https://developer.android.com/guide/components/tasks-and-back-stack.html#ManagingTasks

RECREATING AN ACTIVITY

Activities can be destroyed deliberately (e.g. by pressing the back button). In this case the system's concept is gone forever. But this is not always the case.

System may also destroy the activity. E.g. to recover resources, the screen is rotated. If the system destroys an activity, it saves important details so that it can recreate a new instance if required.

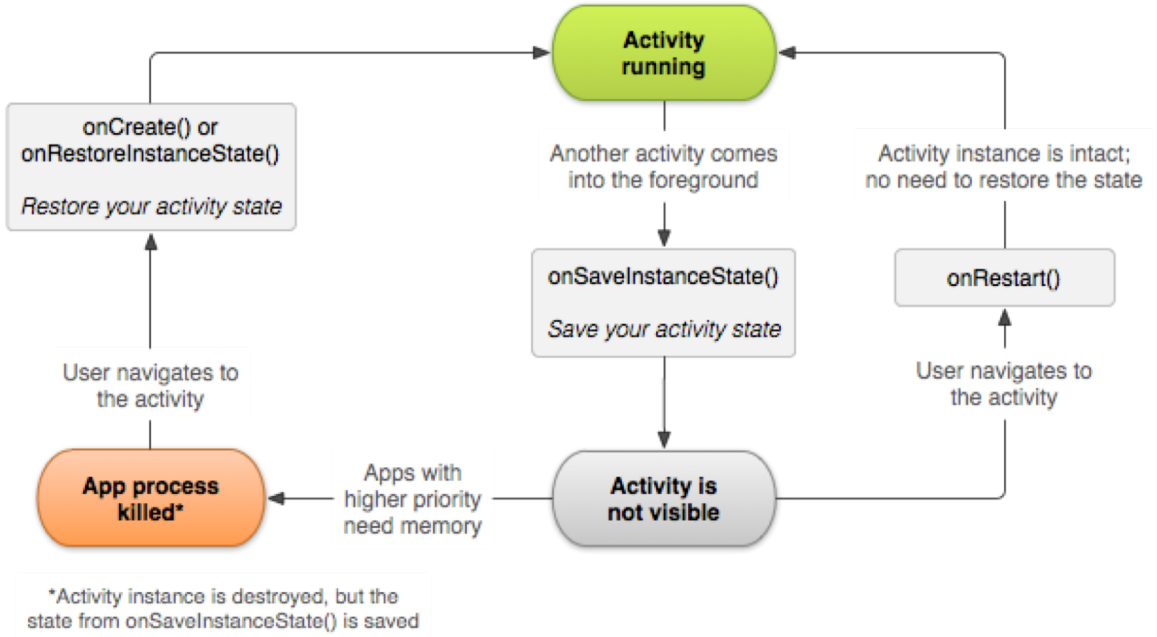
The saved data used to restore the previous state is called the "instance state" and is a collection of key-value pairs stored in a Bundle object.

- By default, the system uses the Bundle instance state to save information about each View object in your activity layout.
- You can save state information about the activity as name-value pairs, using methods such as putString() and putInt().

RECREATING AN ACTIVITY

To save additional data about the activity state, you must override the onSaveInstanceState() callback method.

The system calls this method when the user is leaving your activity and passes it the Bundle object that is saved in the event that your activity is destroyed unexpectedly.



If the system must recreate the activity instance later, it passes the same Bundle object to both the onRestoreInstanceState() and onCreate() methods.

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

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

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

Don’t forget to do this

```
@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
    }
    ...
}
```

PRACTICAL

Download and install Android Studio and add recommended SDK packages

https://developer.android.com/sdk/index.html

Read the introduction to Android Studio

https://developer.android.com/tools/studio/index.html

Complete the tutorial “Building Your First App”

https://developer.android.com/training/basics/firstapp/index.html

QUESTIONS?

Contact:

d.coyle@ucd.ie

Please ask in the Discussion Forum.

Next class:

Android Studio Demo