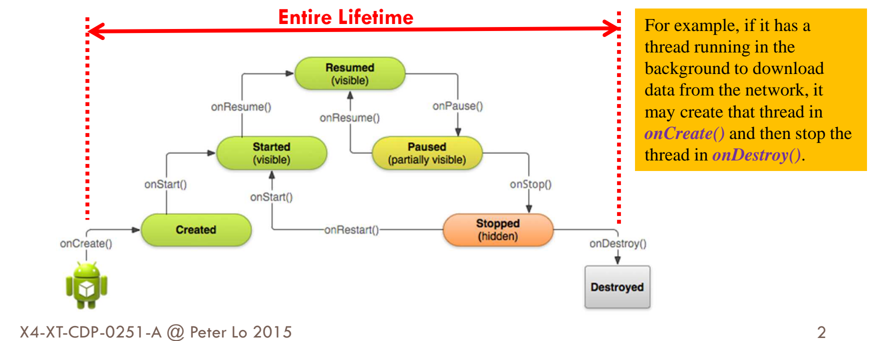
ANDROID APPS DEVELOPMENT FOR MOBILE AND TABLET DEVICE (LEVEL I)

Lecture 3: Android Life Cycle and Permission

Peter Lo

Entire Lifetime

- An activity begins its lifecycle when entering the ***onCreate()*** state
- If not interrupted or dismissed, the activity performs its job and finally terminates and releases its acquired resources when reaching the ***onDestroy()*** event.

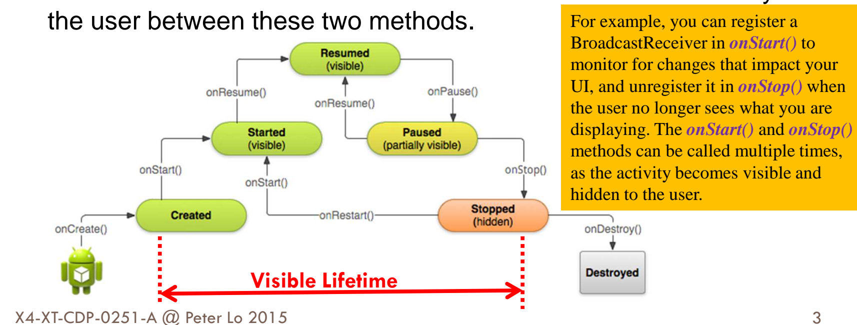


For example, if it has a thread running in the background to download data from the network, it may create that thread in *onCreate()* and then stop the thread in *onDestroy()*.

Visible Lifetime

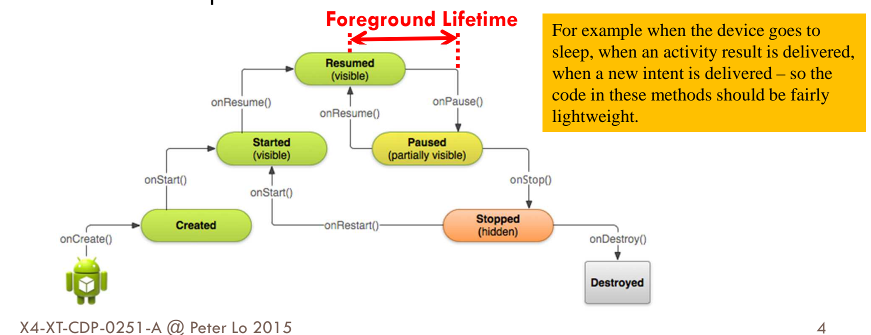
- It happens between a call to ***onStart()*** until a corresponding call to ***onStop()***.
- During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user.
- You can maintain resources that are needed to show the activity to the user between these two methods.

For example, you can register a BroadcastReceiver in *onStart()* to monitor for changes that impact your UI, and unregister it in *onStop()* when the user no longer sees what you are displaying. The *onStart()* and *onStop()* methods can be called multiple times, as the activity becomes visible and hidden to the user.



Foreground Lifetime

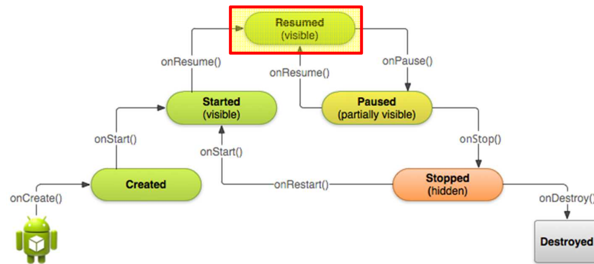
- It happens between a call to ***onResume()*** until a corresponding call to ***onPause()***.
- During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states



For example when the device goes to sleep, when an activity result is delivered, when a new intent is delivered – so the code in these methods should be fairly lightweight.

Life Cycle States – Resumed

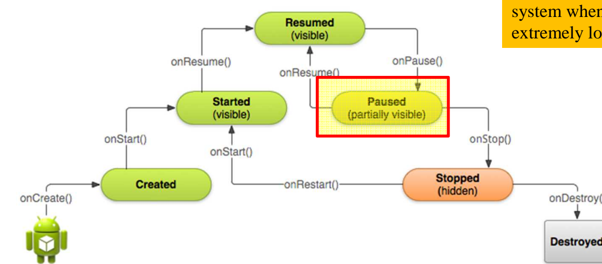
- It is active or running when it is in the foreground of the screen (at the top of the activity stack).
- This is the activity that has “focus” and its graphical interface is responsive to the user’s interactions.



Life Cycle States – Pause

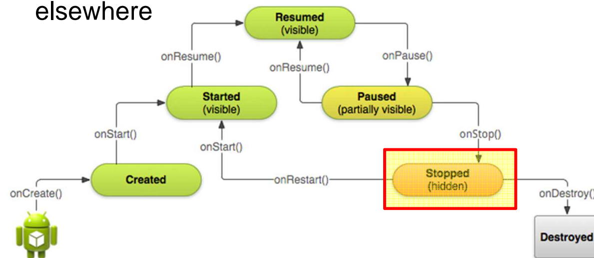
- Lost focus but is still visible to the user.
- Another activity lies on top of it and that new activity either is transparent or doesn't cover the full screen.
- A paused activity is alive, maintaining its state information and attachment to the window manager

Paused activities can be killed by the system when available memory becomes extremely low.



Life Cycle States – Stopped

- Completely obscured by another activity.
- Continues to retains all its state information.
- It is no longer visible to the user
  - Its window is hidden and its life cycle could be terminated at any point by the system if the resources that it holds are needed elsewhere



Life Cycle Callbacks

```
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 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.
   }
}
```

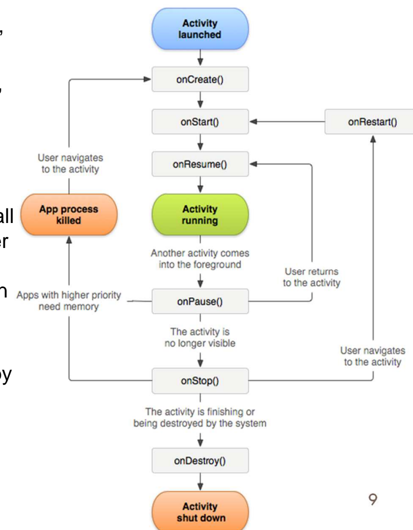
All activities must implement *onCreate()* to do the initial setup when the object is first instantiated

Activities should implement *onPause()* to commit data changes in anticipation to stop interacting with the user

Applications do not need to implement each of the transition methods, however there are mandatory and recommended states to consider

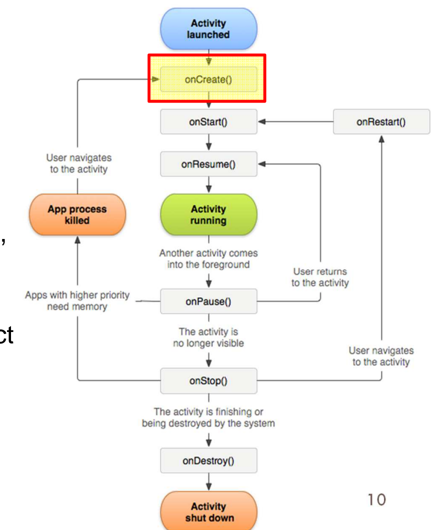
Activity Lifecycle

- If an activity in the foreground of the screen, it is active or running.
- If an activity has lost focus but is still visible, it is paused. A paused activity is completely alive, but can be killed by the system in extreme low memory situations.
- If an activity is completely obscured by another activity, it is stopped. It still retains all state and member information, but 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.



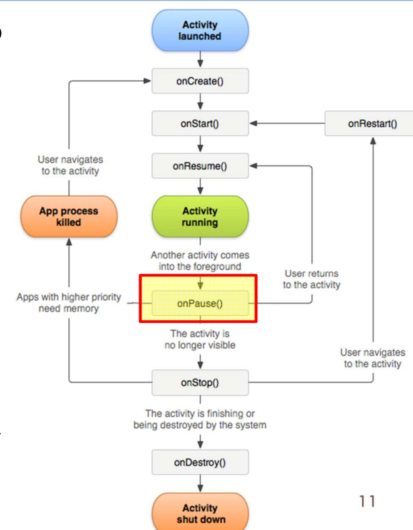
Life Cycle Methods – onCreate( )

- Called when the activity is first created.
- Most of your application's code is written here.
- Typically used to define listener's behavior, initialize data structures, wire-up UI view elements (buttons, text boxes, lists) with static Java controls, etc.
- It may receive a data Bundle object containing the activity's previous state (if any).
- Followed by ***onStart()***.



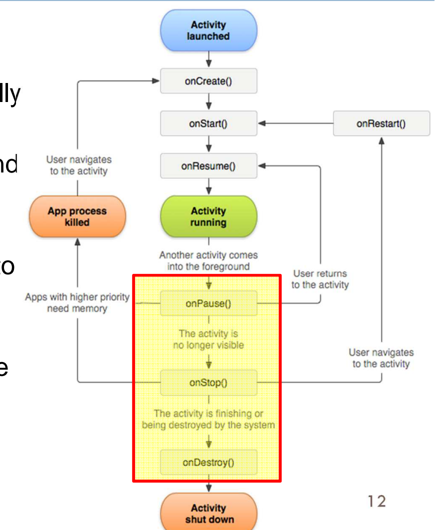
Life Cycle Methods – onPause( )

- Called when the system is about to transfer control to another activity.
- Gives you a chance to commit unsaved data, and stop work that may unnecessarily burden the system.
- The next activity waits until completion of this state.
- Followed either by ***onResume()*** if the activity returns back to the foreground, or by ***onStop()*** if it becomes invisible to the user.
- A paused activity could be killed by the system.



Killable States

- Activities on killable states can be terminated by the system when memory resources become critically low.
- Methods: ***onPause()***, ***onStop()*** and ***onDestroy()*** are killable.
- ***onPause()*** is the only state that is guaranteed to be given a chance to complete before the process is killed.
- You should use ***onPause()*** to write any pending persistent data.



Killable Priority

- If the Android system needs to terminate processes it follows the following priority system.

Process Status	Description	Priority
Foreground	An application in which the user is interacting with an activity, or which has an 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 a 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

Permission Concept in Android

- Android contains a permission system and predefines permissions for certain tasks. Every application can request required permissions and also define new permissions.
  - E.g. an application may declare that it requires access to the Internet.
- Permissions have different levels.
  - Some permissions are automatically granted by the Android system
  - Some are automatically rejected.
- In most cases the requested permissions are presented to the user before installing the application. The user needs to decide if these permissions shall be given to the application.
- An Android application declares the required permissions in its AndroidManifest.xml configuration file. It can also define additional permissions which it can use to restrict access to certain components.

Using Permissions

- A basic Android application has no permissions associated with it by default, meaning it cannot do anything that would adversely impact the user experience or any data on the device.
- To make use of protected features of the device, you must include in your AndroidManifest.xml one or more <uses-permission> tags declaring the permissions that your application needs

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

- The permissions provided by the Android system can be found at
  - http://developer.android.com/reference/android/Manifest.permission.html

Common Permission

Permission	Description
ACCESS_FINE_LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
BLUETOOTH	Allows applications to connect to paired Bluetooth devices
CAMERA	Required to be able to access the camera device.
INTERNET	Allows applications to open network sockets.
READ_CONTACTS	Allows an application to read the user's contacts data.
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.
NFC	Allows applications to perform I/O operations over NFC
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed.

Appendix: Event Handling Approaches

- Events are a useful way to collect data about a user's interaction with interactive components of your app, like button presses or screen touch etc.
- There are several approaches for event handling:
  - Handling Events by Using an Inner Class
  - Handling Events by Having Main Activity Implement Listener Interface
  - Handling Events by Specifying the Event Handler Method in Layout

Handling Events by Using an Inner Class

- Use an inner class that implements the Listener

Advantages

- Assuming that each class is applied to a single control only, same advantages as named inner classes, but shorter.
  - This approach is widely used in Swing, SWT, AWT, and GWT

Disadvantages

- If you applied the handler to more than one control, you would have to cut and paste the code for the handler.
  - This approach should be applied for a single control only
- If the code for the handler is long, it makes the code harder to read by putting it inline.
  - This approach is usually used only when handler code is short

Handling Events by Using an Inner Class

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Attach the listener to the button
        Button button1 = (Button) findViewById(R.id.button1);
        button1.setOnClickListener( new OnClickListener() {
                public void onClick(View arg0) {
                        Your Action …
                }
        } );
}
```

This defines the class and instantiates it all in one fell swoop. This is very analogous to anonymous functions (closures) that are widely used in functional programming languages.

MainActivity.java

Handling Events by Having Main Activity Implement Listener Interface

- Have the main Activity implement the Listener interface. Put the handler method in the main Activity. Call ***setOnClickListener(this)***.

Advantages

- Assuming that the app has only a single control of that Listener type, this is the shortest and simplest of the approaches

Disadvantages

- Scales poorly to multiple controls unless they have completely identical behavior.
  - If you assigned "this" as the handler for more than one control of the same Listener type, the ***onClick*** method would have to have cumbersome if statements to see which control was clicked
  - This approach should be applied when your app has only a single control of that Listener type
- Cannot pass arguments to the Listener.
  - Works poorly for multiple controls

Handling Events by Having Main Activity Implement Listener Interface

```
public class MainActivity extends Activity implements OnClickListener {
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                …
                Button button1 = (Button) findViewById(R.id.button1);
                button1.setOnClickListener(this);
        }

        @Override
        public void onClick(View arg0) {
                Your Action …
        }
}
```

MainActivity.java

Handling Events by Specifying the Event Handler Method in Layout

- Put the handler method in the main Activity. Do not implement a Listener interface or call ***setOnClickListener***. Have the layout file specify the handler method via the ***android:onClick*** attribute.

Advantages

- Assuming that the app has only a single control of that Listener type, mostly the same advantages (short/simple code) as the previous approach where the Activity implemented the interface.
- More consistent with the “do layout in XML” strategy
- You can supply different method names for different controls, so not nearly as limited as interface approach.

Disadvantages

- You cannot pass arguments to Listener.
- Less clear to the Java developer which method is the handler for which control
- Since no *@Override*, no warning until run time if method is spelled wrong or has wrong argument signature

Handling Events by Specifying the Event Handler Method in Layout

```
public class MainActivity extends Activity {
        …
        public void CustomMethod(View arg0) {
                Your Action …
        }
}
```

MainActivity.java

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   tools:context=".MainActivity" >

   <Button
      android:id="@+id/button1"
      android:onClick="CustomMethod"
      android:text="Button" />
…
</RelativeLayout>
```

This is the name of the event handler method in the main class.
This method must have a void return type and take a View as an argument. However, the method name is arbitrary, and the main class need not implement any particular interface.

activity_main.xml