Part 3

Android Application's Life Cycle

Victor Matos
Cleveland State University

Notes are based on:

Unlocking Android
by Frank Ableson, Charlie Collins, and Robi Sen.
ISBN 978-1-933988-67-2
Manning Publications, 2009.

Android Developers
http://developer.android.com/index.html

Android Applications

An application consists of one or more *components* that are defined in the application's manifest file. A component can be one of the following:

1. ***An Activity***
2. ***A Service***
3. ***A broadcast receiver***
4. ***A content provider***

Android Applications

1. Activity

An *activity* usually presents a *single visual user interface* from which a number of actions could be performed.

Altough activities work together to form a cohesive user interface, *each activity is independent of the others*.

Typically, *one of the activities is marked as the first* one that should be presented to the user when the application is launched.

Moving from one activity to another is accomplished by having the current activity start the next one through so called ***intents***.

Reference: Friedger Müffke (friedger@openintents.org)

Android Applications

2. Service

A service doesn't have a visual user interface, but rather runs in the background for an indefinite period of time.

It's possible to connect to (*bind to*) an ongoing service (and start the service if it's not already running).

While connected, you can communicate with the service through an interface that the service exposes.

Reference: Friedger Müffke (friedger@openintents.org)

Android Applications

3. Broadcast receiver

A *broadcast receiver* is a component that does nothing but *receive and react to broadcast announcements.*

Many broadcasts originate in system code (eg. “you got mail“) but any other applications can also initiate broadcasts.

Broadcast receivers do not display a user interface. However, they may start an activity in response to the information they receive, or - as services do - they may use the notification manager to alert the user.

Reference: Friedger Müffke (friedger@openintents.org)

Android Applications

4. Content provider

A *content provider* makes a specific set of the application's data available to other applications.

The data usually is stored in the file system, or in an SQLite database.

The content provider implements a standard set of methods that enable other applications to retrieve and store data of the type it controls.

However, applications do not call these methods directly. Rather they use a *content resolver* object and call its methods instead. A content resolver can talk to any content provider; it cooperates with the provider to manage any interprocess communication that's involved.

Reference: Friedger Müffke (friedger@openintents.org)

Android Applications

Every Android application runs in its own process

(*with its own instance of the Dalvik virtual machine*).

Whenever there's a request that should be handled by a particular component,

- Android makes sure that the application process of the component is running,
- starting it if necessary, and
- that an appropriate instance of the component is available, creating the instance if necessary.

Application's Life Cycle

A Linux process encapsulating an Android application is created for the application when some of its code needs to be run, and will remain running until

1. it is no longer needed, **OR**
2. the system needs to reclaim its memory for use by other applications.

Application's Life Cycle

An unusual and fundamental feature of Android is that an application process's lifetime is not directly controlled by the application itself.

Instead, it is determined by the system through a combination of

1. the parts of the application that the system knows are running,
2. how important these things are to the user, and
3. how much overall memory is available in the system.

Component Lifecycles

Application components have a **lifecycle**

1. A **beginning** when Android instantiates them to respond to intents
2. An **end** when the instances are destroyed.
3. In **between**, they may sometimes be *active* or *inactive*, or -in the case of activities- *visible* to the user or *invisible*.

Activty Stack

- 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.
- If the user presses the *Back Button* the next activity on the stack moves up and becomes active. 

Activity Stack

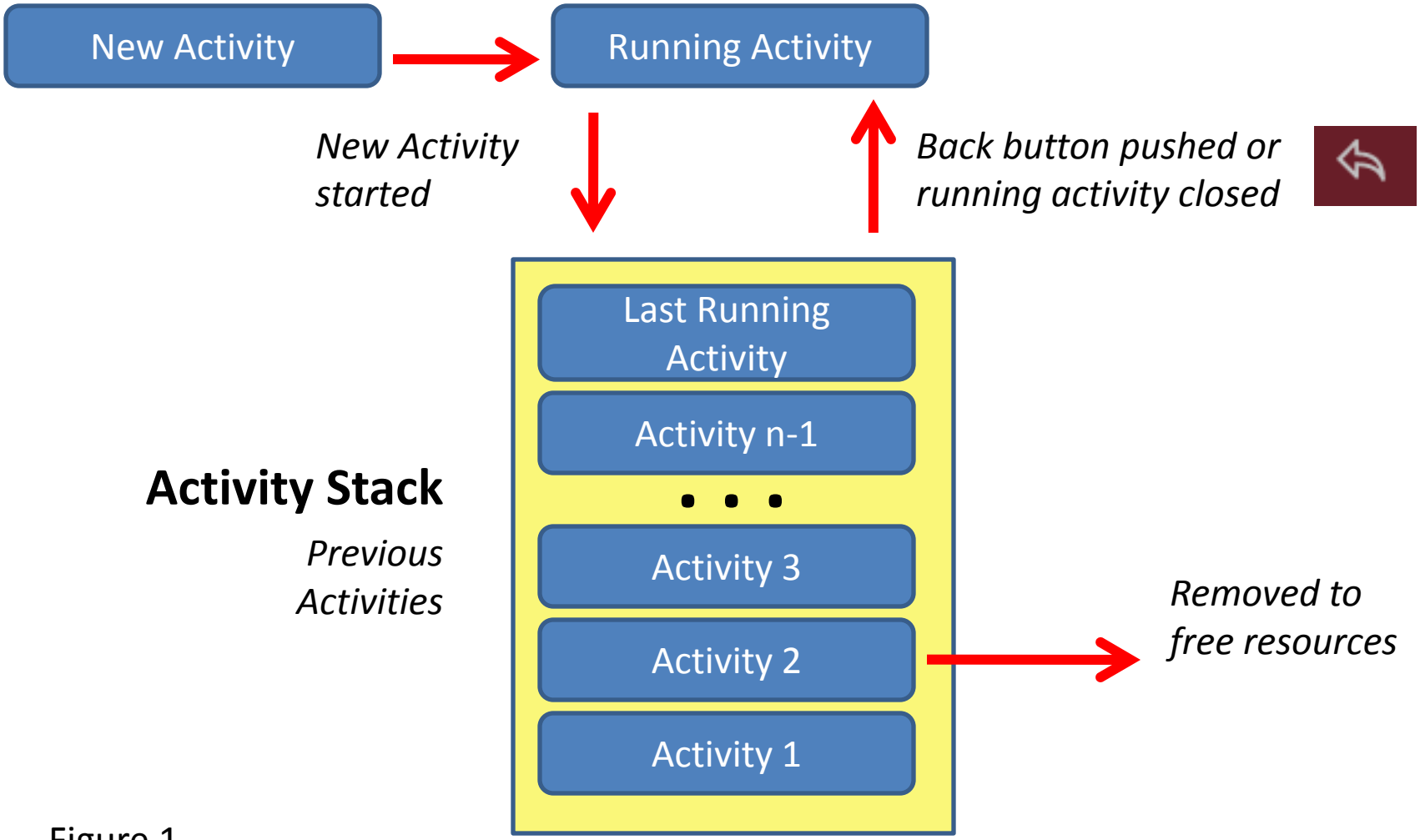


Figure 1.

Life Cycle States

An activity has essentially three states:

1. It is ***active*** or ***running***
2. It is ***paused*** or
3. It is ***stopped*** .

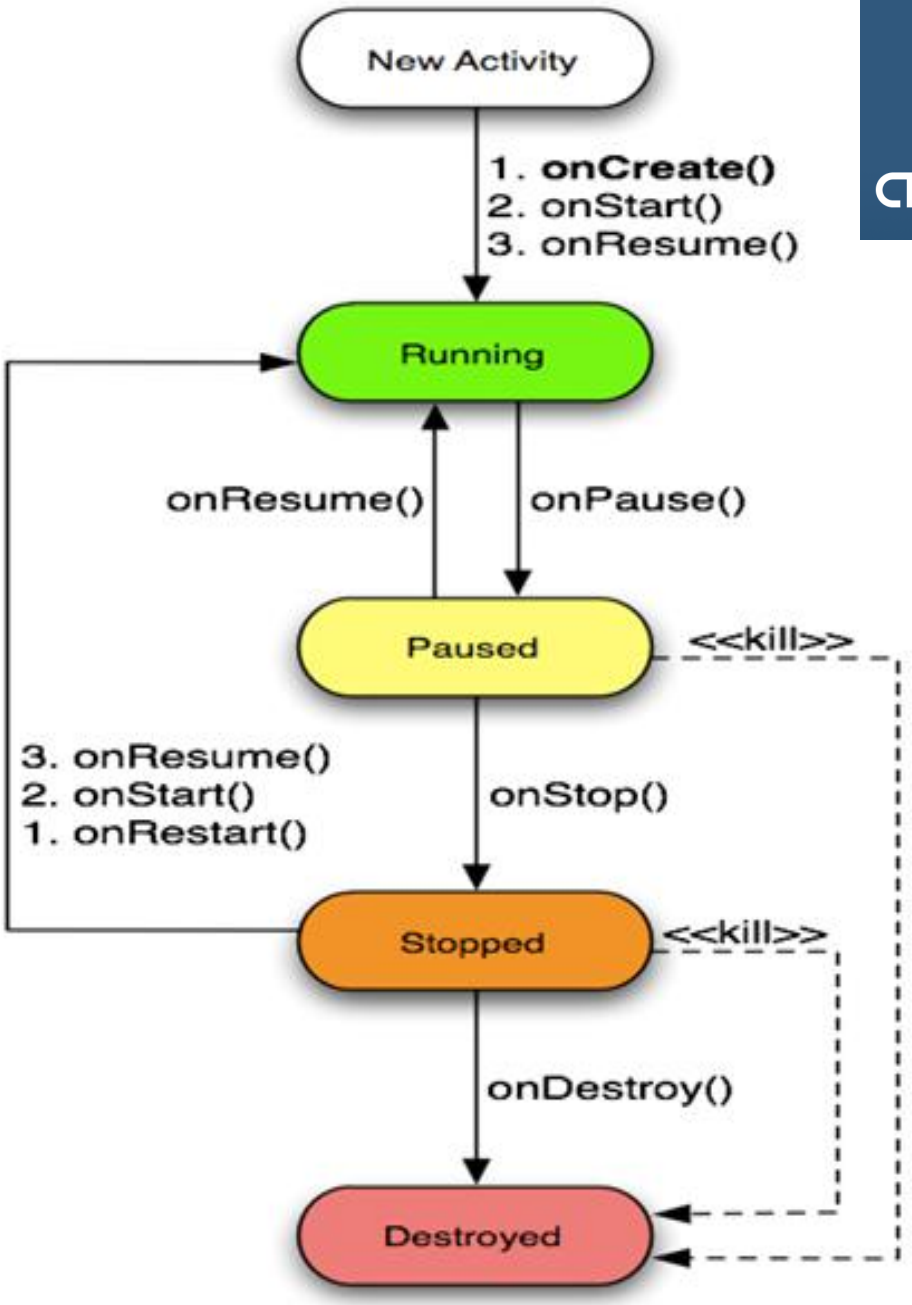


Figure 2.

Life Cycle States

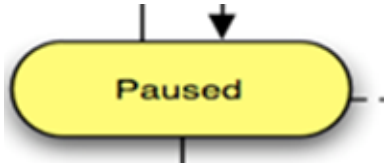
An activity has essentially three states:

1. It is ***active* or *running*** when it is in the *foreground* of the screen (at the top of the *activity stack* for the current task).

   This is the activity that is the focus for the user's actions.

Life Cycle States

An *activity has essentially three states (cont.) :*



2. It is ***paused*** if it has lost focus but is still visible to the user.

   That is, 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 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.

Life Cycle States

An activity has essentially three states (cont.):

3. It is ***stopped*** if it is completely *obscured* by another activity.

   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.

Application's Life Cycle

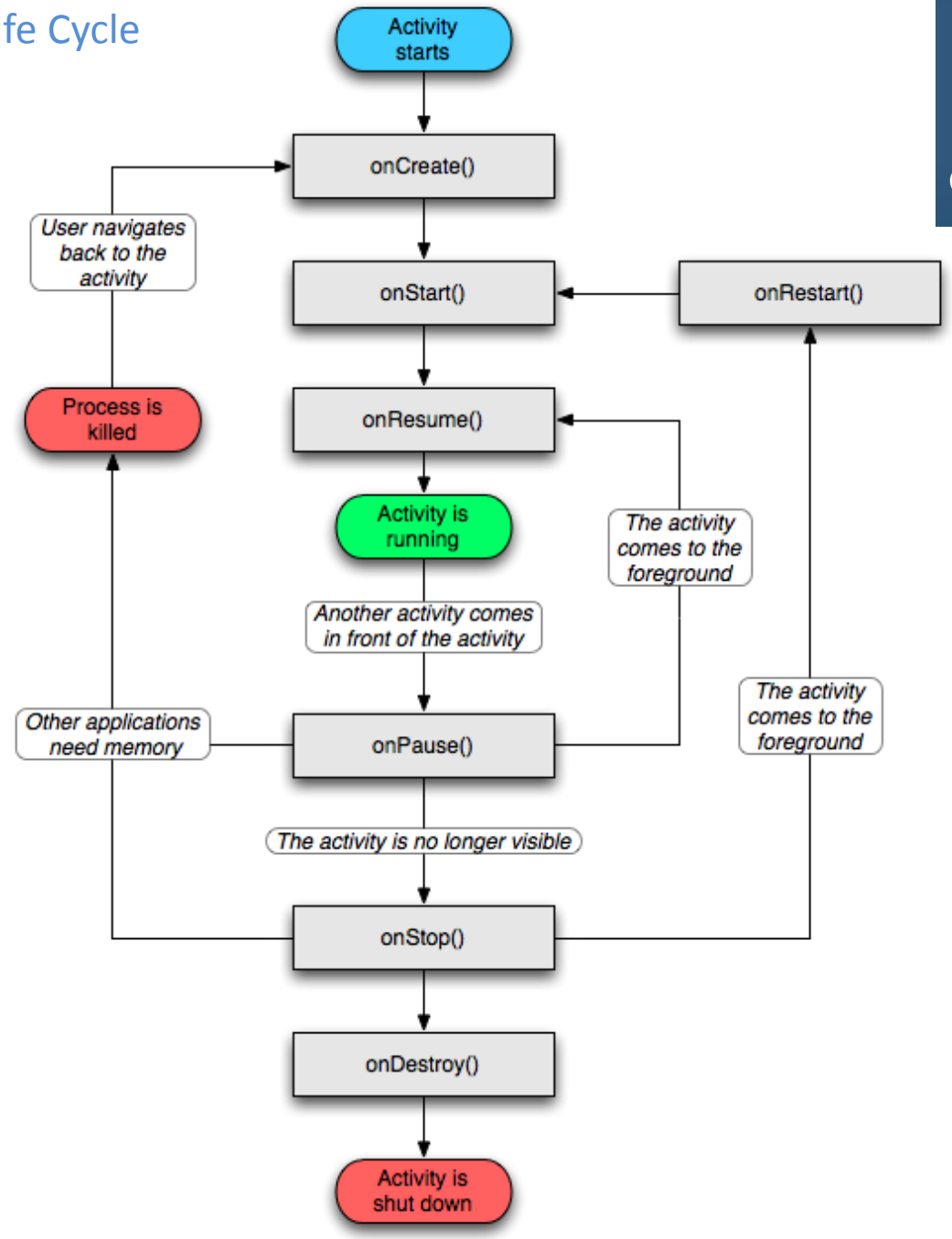


Figure 3.

Application's Life Cycle

Teaching notes

Your turn!
EXPERIMENT 1.

1. Write an Android app. ("PuraVida") to show the different cycles followed by an application.
2. The **main.xml** layout should include a Button (text: "Finish", id: btnFinish) and an EditText container (txt: "" and id txtMsg).
3. Use the onCreate method to connect the button and textbox to the program. Add the following line of code:

```
Toast.makeText(this, "onCreate", 1).show();
```

4. The click method has only one command: **finish();** called to terminate the application. Add a Toast-command (as the one above) to each of the remaining six main events. To simplify your job use the Eclipse's top menu: Source > Override/Implement Methods…
5. On the option window check mark each of the following events: onStart, onResume, onPause, onStop, onDestry, onRestart
   (notice how many *onEvent…* methods are there!!!)
6. Save your code.

Application's Life Cycle

Teaching notes

Your turn!
EXPERIMENT 1 (cont.)

7. Compile and execute application.
8. Write down the sequence of messages displayed by the Toast-commands.
9. Press the FINISH button. Observe the sequence of states.
10. Re-execute the application
11. Press emulator's HOME button. What happens?
12. Click on launch pad, look for icon and return to the "PuraVida" app. What sequence of messages is displayed?
13. Click on the emulator's CALL (Green phone). Is the app paused or stopped?
14. Click on the BACK button to return to the application.
15. Long-tap on the emulator's HANG-UP button. What happens?

Application's Life Cycle

Teaching notes

Your turn!
EXPERIMENT 2

7. Run a second emulator.
    1. Make a voice-call to the first emulator that is still showing our app. What happens on this case? (real-time synchronous request)
    2. Send a text-message to first emulator (asynchronous attention request)

8. Write a phrase in the EditText box ("these are the best moments of my life….").

9. Re-execute the app. What happened to the text?

Application's Life Cycle

Your turn!
EXPERIMENT 3

Teaching notes

Provide data persistency.

18. Use the **onPause** method to add the following fragment

```
SharedPreferences myFile1 = getSharedPreferences("myFile1",
                                                 Activity.MODE_PRIVATE);
SharedPreferences.Editor myEditor = myFile1.edit();
String temp = txtMsg.getText().toString();
myEditor.putString("mydata", temp);
myEditor.commit();
```

18. Use the **onResume** method to add the following frament

```
SharedPreferences myFile = getSharedPreferences("myFile1",
                                                Activity.MODE_PRIVATE);
if ( (myFile != null) && (myFile.contains("mydata")) ) {
        String temp = myFile.getString("mydata", "***");
        txtMsg.setText(temp);
}
```

19. What happens now with the data previously entered in the text box?

Life Cycle Events

Summary: APP MILESTONES

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its **finish**() method), or simply killing its process.

When it is displayed again to the user, it must be completely restarted and restored to its previous state.

As an activity transitions from state to state, it is notified of the change by calls to the following protected *transition* methods:

void onCreate(Bundle *savedInstanceState*) **void onStart()** **void onRestart()** **void onResume()**	**void onPause()** **void onStop()** **void onDestroy()**

Life Cycle Events

All of these methods are **hooks** that you can override to do appropriate work when the state changes.

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

(**Highly Recommended**)
Many activities will also implement **onPause()** to commit data changes and otherwise prepare to stop interacting with the user.

Application's Lifetime

Entire Lifetime

The seven transition methods (Figure 3) define the entire lifecycle of an activity.

- The **entire lifetime** of an activity happens between the first call to **onCreate()** through to a single final call to **onDestroy()**.
- An activity does all its initial setup of "global" state in onCreate(), and releases all remaining resources in onDestroy().

Visible Lifetime

Visible Lifetime

The **visible lifetime** of an activity 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.*

- The onStart() and onStop() methods can be called multiple times, as the activity alternates between being visible and hidden to the user.

- Between these two methods, you can maintain resources that are needed to show the activity to the user.

Foreground Lifetime

Foreground Lifetime

The **foreground lifetime** of an activity happens between a call to onResume() until a corresponding call to onPause().

> *During this time, the activity is in front of all other activities on screen and is interacting with the user.*

An activity can frequently transition between the *resumed* and *paused* states — for example,

- *onPause*() is called when the device goes to sleep or when a new activity is started,
- *onResume*() is called when an activity result or a new intent is delivered.

Life Cycle Methods

Method: **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, and so on.
- This method is passed a *Bundle* object containing the activity's previous state, if that state was captured.
- Always followed by *onStart*()

Life Cycle Methods

Method: **onRestart()**

- Called after the activity has been stopped, just prior to it being started again.
- Always followed by *onStart*()

Method: **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.

Life Cycle Methods

Method: **onResume()**

1. Called just before the activity starts interacting with the user.
2. At this point the activity is at the top of the activity stack, with user input going to it.
3. Always followed by *onPause*().

Life Cycle Methods

Method: **onPause()**

1. Called when the system is about to start resuming another activity.
2. This method is typically used to *commit* unsaved changes to persistent data, stop animations and other things that may be consuming CPU, and so on.
3. It should do whatever it does very quickly, because the next activity will not be resumed until it returns.
4. Followed either by *onResume*() if the activity returns back to the front, or by *onStop*() if it becomes invisible to the user.
5. The activity in this state is *killable* by the system.

Life Cycle Methods

Method: **onStop()**

1. Called when the activity is no longer visible to the user.
2. This may happen because it is being destroyed, or because another activity (either an existing one or a new one) has been resumed and is covering it.
3. Followed either by *onRestart*() if the activity is coming back to interact with the user, or by *onDestroy*() if this activity is going away.
4. The activity in this state is *killable* by the system.

Life Cycle Methods

Method: **onDestroy()**

1. Called before the activity is destroyed.
2. This is the final call that the activity will receive.
3. It could be called 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.
4. You can distinguish between these two scenarios with the *isFinishing()* method.
5. The activity in this state is *killable* by the system.

Life Cycle Methods

Killable States

- Activities on killable states can be terminated by the system *at any time after the method returns, without executing another line of the activity's code*.
- Three methods (*onPause*(), *onStop*(), and *onDestroy*()) are *killable.*
- ***onPause***() is the only one that is guaranteed to be called before the process is killed — *onStop*() and *onDestroy*() may not be.
- Therefore, you should use *onPause*() to write any persistent data (such as user edits) to storage.

Life Cycle Methods

As an aside…

Android Preferences

Preferences is a lightweight mechanism to store and retrieve *key-value* pairs of primitive data types. It is typically used to store application preferences, such as a default greeting or a text font to be loaded whenever the application is started.

Call **Context.getSharedPreferences()** to read and write values.

Assign a name to your set of preferences if you want to share them with other components in the same application, or use **Activity.getPreferences()** with no name to keep them private to the calling activity.

You cannot share preferences across applications (except by using a content provider).

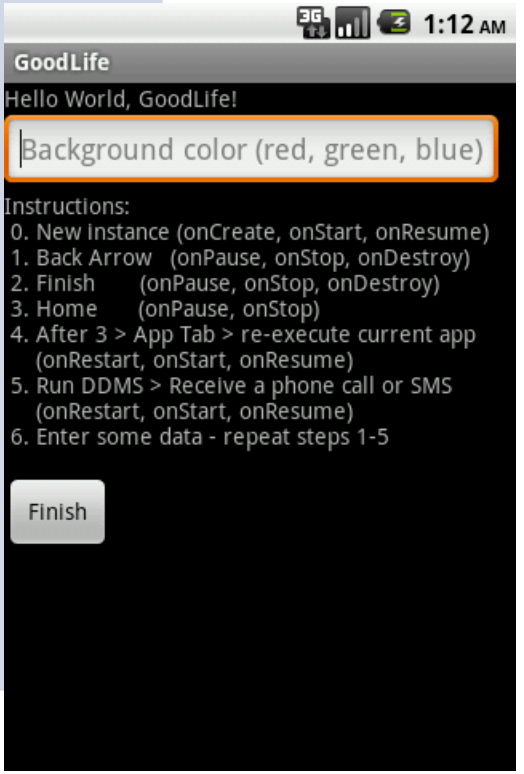
Example Life Cycle

Example

The following application demonstrates some of the state transitioning situations experienced in the life-cycle of a typical Android activity.

LAYOUT

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/myScreen"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:background="#ff000000"
   >
<TextView
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:text="@string/hello"
   />
<EditText
android:id="@+id/txtColorSelect"
android:hint="Background color (red, green, blue)"
android:layout_width="wrap_content"
android:layout_height="wrap_content">
</EditText>
<TextView
android:id="@+id/txtToDo"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:background="#00000000">
<!-- transparent  -->
</TextView>
<Button
android:text=" Finish "
android:id="@+id/btnFinish"
android:layout_width="wrap_content"
android:layout_height="wrap_content">
</Button>
</LinearLayout>
```



Example: Life Cycle

Code: Life Cycle Demo. Part 1

```
Package cis493.lifecycle

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.widget.*;

//GOAL: show the following life-cycle events in action

//protected void onCreate(Bundle savedInstanceState);
//protected void onStart();
//protected void onRestart();
//protected void onResume();
//protected void onPause();
//protected void onStop();
//protected void onDestroy();
```

Example: Life Cycle

Code: Life Cycle Demo. Part 2

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    myScreen = (LinearLayout) findViewById(R.id.myScreen);

    txtToDo = (TextView) findViewById(R.id.txtToDo);
     String msg = "Instructions:                                    \n "
         + "0. New instance (onCreate, onStart, onResume)    \n "
         + "1. Back Arrow   (onPause, onStop, onDestroy)     \n "
         + "2. Finish       (onPause, onStop, onDestroy)     \n "
         + "3. Home (onPause, onStop)                        \n "
         + "4. After 3 > App Tab > re-execute current app    \n "
         + "    (onRestart, onStart, onResume)               \n "
         + "5. Run DDMS > Receive a phone call or SMS        \n "
         + "    (onRestart, onStart, onResume)               \n "
         + "6. Enter some data - repeat steps 1-5            \n ";

    txtToDo.setText(msg);
```

Example: Life Cycle

Code: Life Cycle Demo. Part 2

```
  txtColorSelect = (EditText) findViewById(R.id.txtColorSelect);
   // you may want to skip discussing the listener until later
   txtColorSelect.addTextChangedListener(new TextWatcher(){
   public void onTextChanged(CharSequence s, int start, int before, int count) {
     // TODO Auto-generated method stub
   }
   public void beforeTextChanged(CharSequence s, int start, int count,int after) {
     // TODO Auto-generated method stub
   }
   public void afterTextChanged(Editable s) {
     changeBackgroundColor(s.toString());
   }
  });

  btnFinish = (Button) findViewById(R.id.btnFinish);
  btnFinish.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
         finish();
      }
  });
  Toast.makeText(getApplicationContext(), "onCreate", 1).show();
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 3

```
@Override
protected void onPause() {
    super.onPause();
    saveDataFromCurrentState();
    Toast.makeText(this, "onPause", 1).show();
}

@Override
protected void onRestart() {
    super.onRestart();
    Toast.makeText(this, "onRestart", 1).show();
}

@Override
protected void onResume() {
    super.onResume();
    Toast.makeText(this, "onResume", 1).show();
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 4

```
@Override
protected void onStart() {
    // TODO Auto-generated method stub
    super.onStart();
    updateFromSavedState();
    Toast.makeText(this, "onStart", 1).show();
}

@Override
protected void onDestroy() {
    // TODO Auto-generated method stub
    super.onDestroy();
    Toast.makeText(this, "onDestroy", 1).show();
}

@Override
    protected void onStop() {
    // TODO Auto-generated method stub
    super.onStop();
    Toast.makeText(this, "onStop", 1).show();
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 5

```
protected void saveDataFromCurrentState() {
   SharedPreferences myPrefs = getSharedPreferences(MYPREFSID, actMode);
   SharedPreferences.Editor myEditor = myPrefs.edit();
   myEditor.putString("myBkColor", txtColorSelect.getText().toString());
   myEditor.commit();
}// saveDataFromCurrentState

protected void updateFromSavedState() {
   SharedPreferences myPrefs = getSharedPreferences(MYPREFSID, actMode);

   if ((myPrefs != null) && (myPrefs.contains("myBkColor"))) {
      String theChosenColor = myPrefs.getString("myBkColor", "");
      txtColorSelect.setText(theChosenColor);
      changeBackgroundColor(theChosenColor);
   }
}// UpdateFromSavedState

protected void clearMyPreferences() {
   SharedPreferences myPrefs = getSharedPreferences(MYPREFSID, actMode);
   SharedPreferences.Editor myEditor = myPrefs.edit();
   myEditor.clear();
   myEditor.commit();
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 6

```
private void changeBackgroundColor (String theChosenColor){
    // change background color
    if (theChosenColor.contains("red"))
        myScreen.setBackgroundColor(0xffff0000);
    else if (theChosenColor.contains("green"))
        myScreen.setBackgroundColor(0xff00ff00);
    else if (theChosenColor.contains("blue"))
        myScreen.setBackgroundColor(0xff0000ff);
    else {
        //reseting user preferences
        clearMyPreferences();
        myScreen.setBackgroundColor(0xff000000);
    }
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 8

```
 /*
 protected void onRestoreInstanceState(Bundle savedInstanceState)
 This method is called after onStart() when the activity is being re-initialized
 from a previously saved state.
 The default implementation of this method performs a restore of any view state
 that had previously been frozen by onSaveInstanceState(Bundle).
 */
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
  super.onRestoreInstanceState(savedInstanceState);
  Toast.makeText(getBaseContext(),
                 "onRestoreInstanceState ...BUNDLING",
                 Toast.LENGTH_LONG).show();
}
```

Example: Life Cycle

Code: Life Cycle Demo. Part 9

```
/*
protected void onSaveInstanceState(Bundle outState)

Called to retrieve per-instance state from an activity before being killed
so that the state can be restored in
   onCreate(Bundle) or
   onRestoreInstanceState(Bundle)  (the Bundle populated by this method
                                    will be passed to both).
This method is called before an activity may be killed so that when it comes
back some time in the future it can restore its state. For example, if activity B
is launched in front of activity A, and at some point activity A is killed to
reclaim resources, activity A will have a chance to save the current state of
its user interface via this method so that when the user returns to activity A,
the state of the user interface can be restored via:
onCreate(Bundle) or onRestoreInstanceState(Bundle).
*/
```

Example: Life Cycle

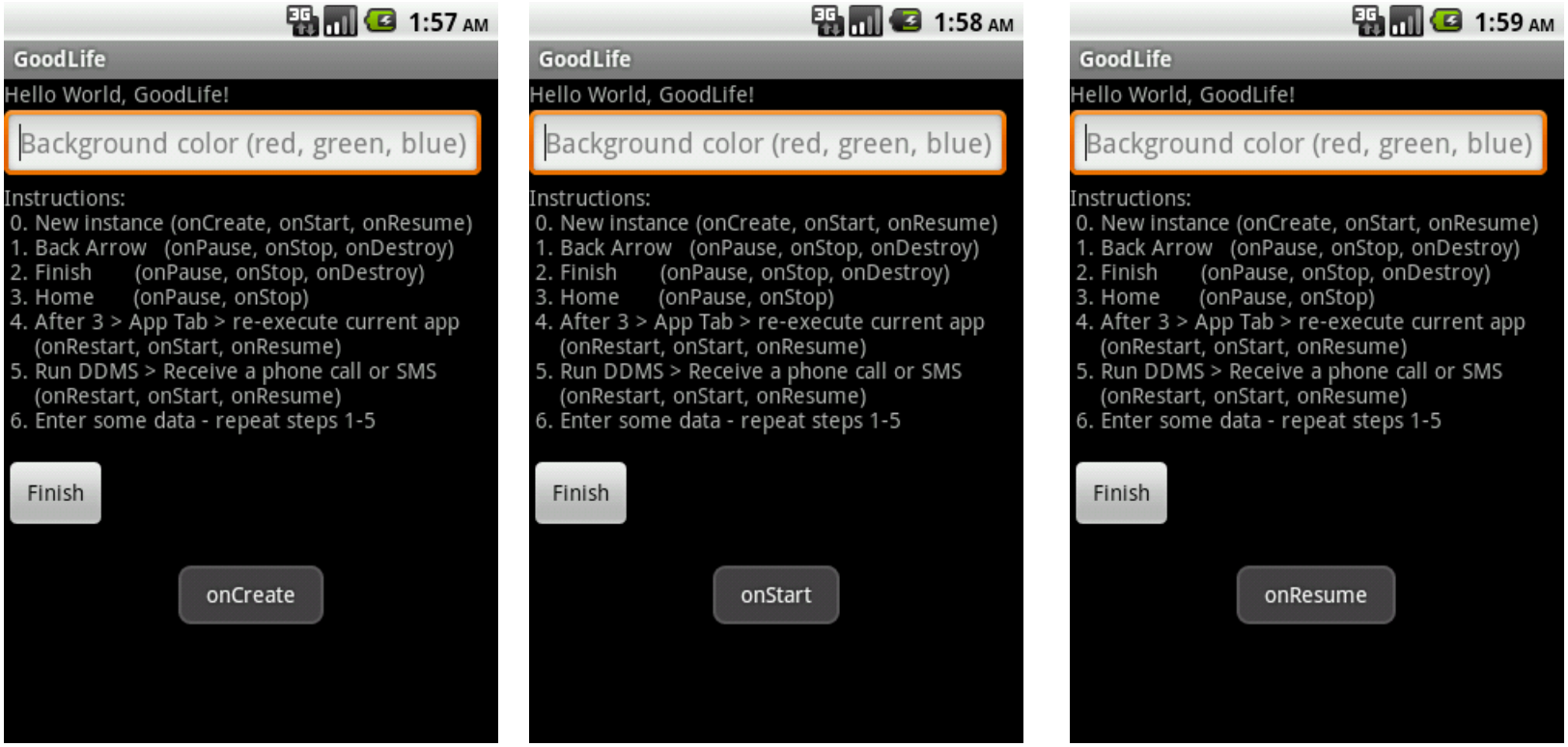
Code: Life Cycle Demo. Part 10

```
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        Toast.makeText(getBaseContext(),
                       "onSaveInstanceState   ...BUNDLING",
                       Toast.LENGTH_LONG).show();
    } // onSaveInstanceState

}//LyfeCicleDemo
```

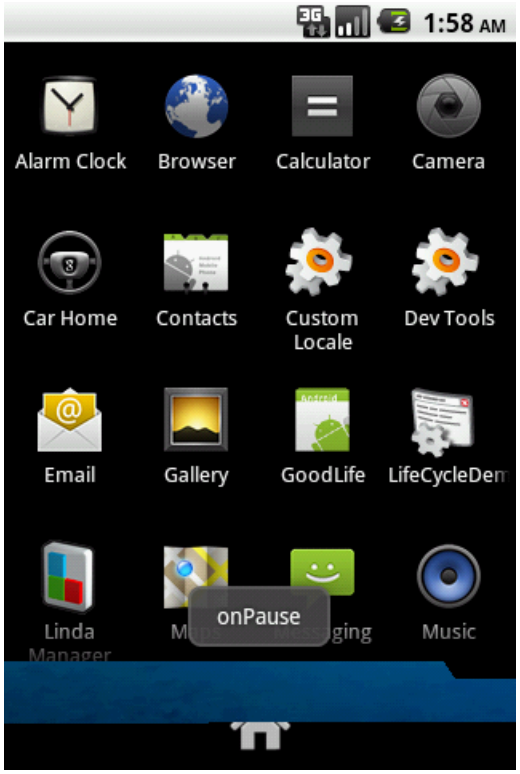
Example: Life Cycle

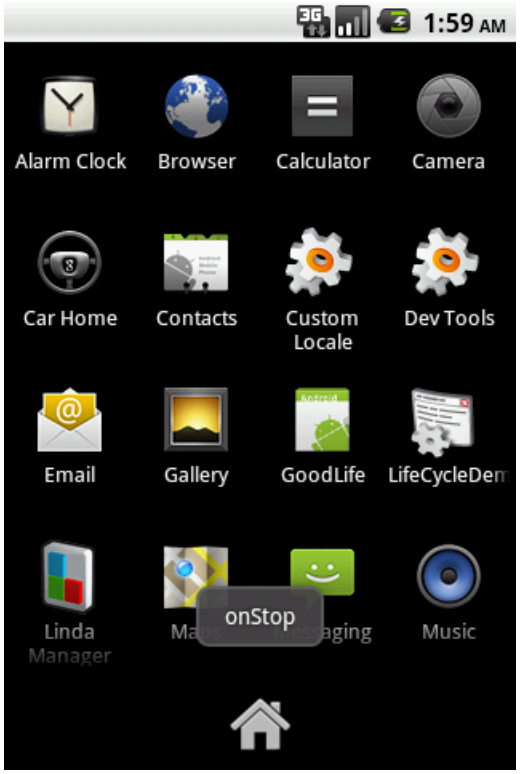
onCreate… **onStart…** **onResume…**

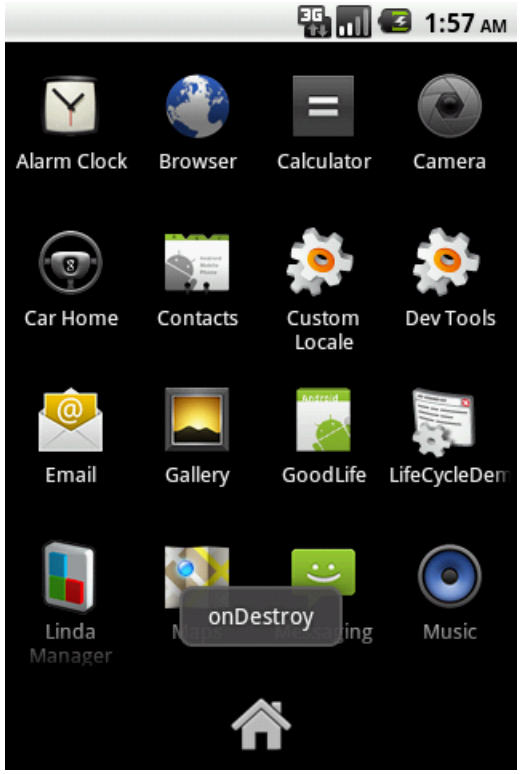


Example: Life Cycle

onPause… **onStop…** **onDestroy…**



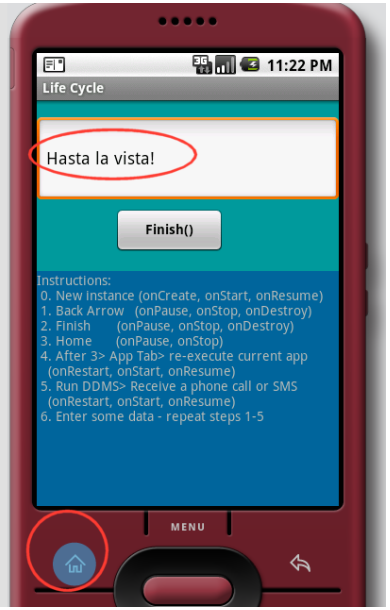




After pressing "Back Arrow"

Example: Life Cycle

After pressing “Home”	After re-executing AndLife2	After “Back Arrow” or Finish
onSavedInstanceState >	onRestart >	onPause >
onPause >	onStart >	onStop >
onStop >	onResume >	onDestroy >



Preserving State Information

1. Enter data: “Hasta la vista!”
2. Click Home button
3. onSavedInstance > onPause > onStop
4. Read your SMS
5. Execute an instance of the application
6. onRestart > onStart > onResume
7. You see the data entered in step 1

End of Example

Application's Life Cycle

Questions ?

Application's Life Cycle

Appendix

Saving State

```
@Override
public void onCreate(Bundle savedInstanceState) {
        ... somevalue = savedInstanceState.getString(SOME_KEY);
        ...
}
...
@Override protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putString(SOME_KEY, "blah blah blah");
}
```