MOBILE DEVELOPMENT

APPLICATION’S LIFE CYCLE

CONTENTS

ANATOMY OF ANDROID APPLICATIONS

Core components are the primordial classes or building blocks from which apps are made.

An application consists of one or more core component objects, working in a cooperative mode, each contributing somehow to the completion of the tasks

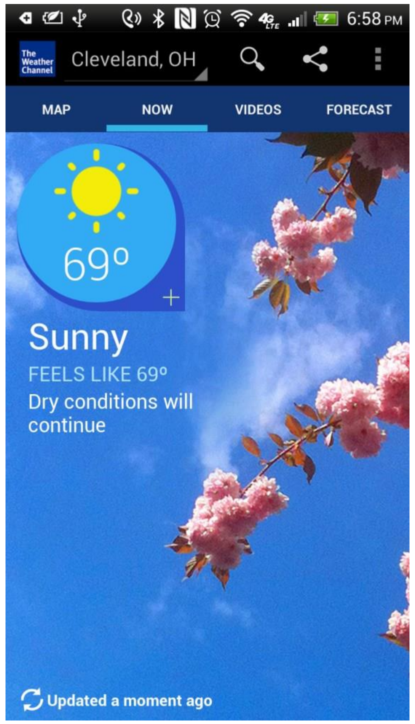
Each core component provides a particular type of functionality and has a distinct lifecycle.

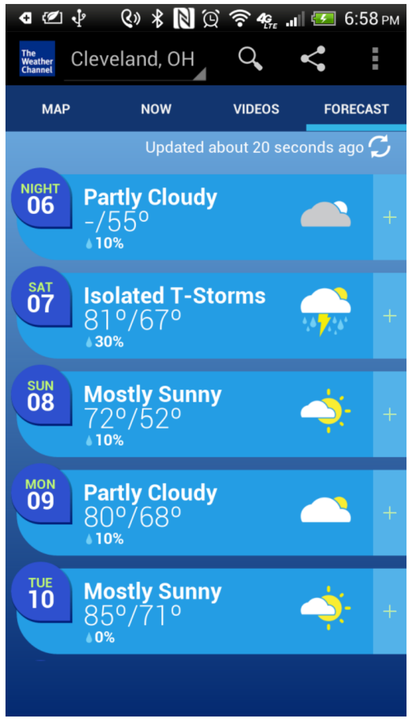
A lifecycle defines how the component is created, transitioned, and destroyed.

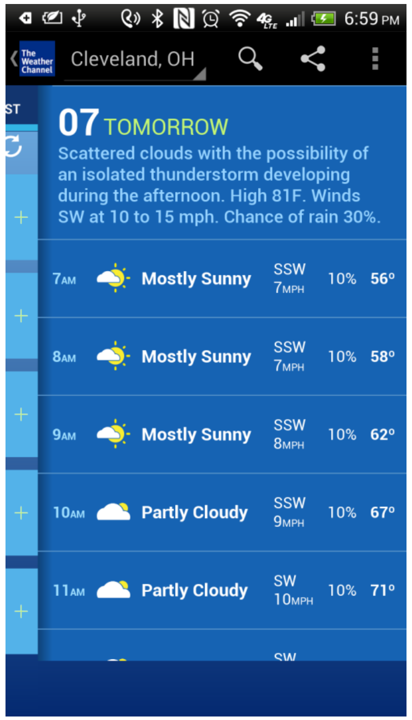
There are four type of core components

- Activities
- Services
- Broadcast Receiver
- Content Provider

ANATOMY OF ANDROID APPLICATIONS (Android's core components – activity)




6:58 PM
Cleveland, OH
MAP
NOW
VIDEOS
FORECAST
Updated about 20 seconds ago
NIGHT 06 Partly Cloudy -/55º 10%
SAT 07 Isolated T-Storms 81º/67º 30%
SUN 08 Mostly Sunny 72º/52º 10%
MON 09 Partly Cloudy 80º/68º 10%
TUE 10 Mostly Sunny 85º/71º 0%


6:59 PM
Cleveland, OH
07 TOMORROW
Scattered clouds with the possibility of an isolated thunderstorm developing during the afternoon. High 81F. Winds SW at 10 to 15 mph. Chance of rain 30%.
7AM Mostly Sunny SSW 7MPH 10% 56º
8AM Mostly Sunny SSW 7MPH 10% 58º
9AM Mostly Sunny SSW 8MPH 10% 62º
10AM Partly Cloudy SSW 9MPH 10% 67º
11AM Partly Cloudy SW 10MPH 10% 71º

An activity usually presents a single graphical visual interface (GUI), in addition to displaying/collecting of data, provides some kind of 'code-behind' functionality.

A typical Android application contains one or more Activity objects.

Applications must designate one activity as their main task or entry point. That activty is the first to be executed when the app is launched.

An activity may transfer control and data to another activity through an interprocess communication protocol called intents

For example, a login activity may show a screen to enter username and password. After clicking a button some authentication process is applied on the data, and before the login activity ends some other activity is called.

ANATOMY OF ANDROID APPLICATIONS (Android’s core components – service)

Services are a special type of activity that do not have a visual user interface. A service object may be active without the user noticing its presence.

Services are analogous to secondary threads, usually running some kind of background ‘busy-work‘ for an indefinite period of time.

Applications start their own services or connect to services already active.

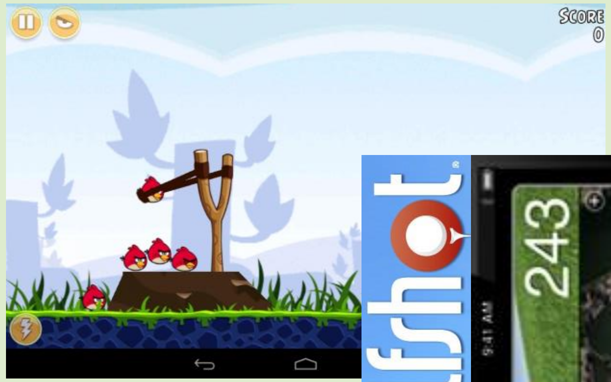
Examples:

- Your background GPS service could be set to quietly run in the background detecting location information from satellites, phone towers or wi-fi routers.
- The service could periodically broadcast location coordinates to any app listening for that kind of data.
- An application may opt for binding to the running GPS service and use the data that it supplies.

ANATOMY OF ANDROID APPLICATIONS (Android's core components – service)

In this example a music service (say Pandora Radio) and GPS location run in the background.

The selected music station is heard while other GUIs are show on the device's screen. For instance, our user –an avid golfer- may switch between occasional golf course data reading (using the GolfShot app) and "Angry Birds"

Background	Foreground
	

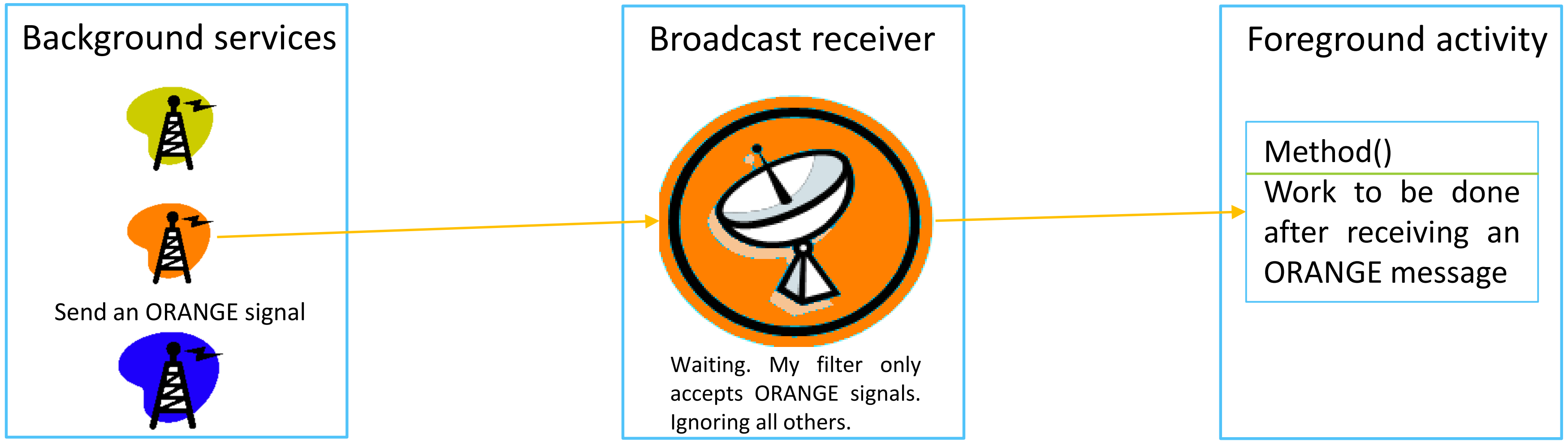
ANATOMY OF ANDROID APPLICATIONS (Android’s core components – broadcast receiver)

A broadcast receiver is a dedicated listener that waits for a triggering system-wide message to do some work. The message could be something like low-battery, wi-fi connection available, earth-quakes in California, speed-camera nearby.

Broadcast receivers do not display a user interface.

They typically register with the system by means of a filter acting as a key. When the broadcasted message matches the key the receiver is activated.

A broadcast receiver could respond by either executing a specific activity or use the notification mechanism to request the user‘s attention.


Background services
Send an ORANGE signal
Broadcast receiver
Waiting. My filter only accepts ORANGE signals. Ignoring all others.
Foreground activity
Method()
Work to be done after receiving an ORANGE message

ANATOMY OF ANDROID APPLICATIONS (Android's core components – content provider)

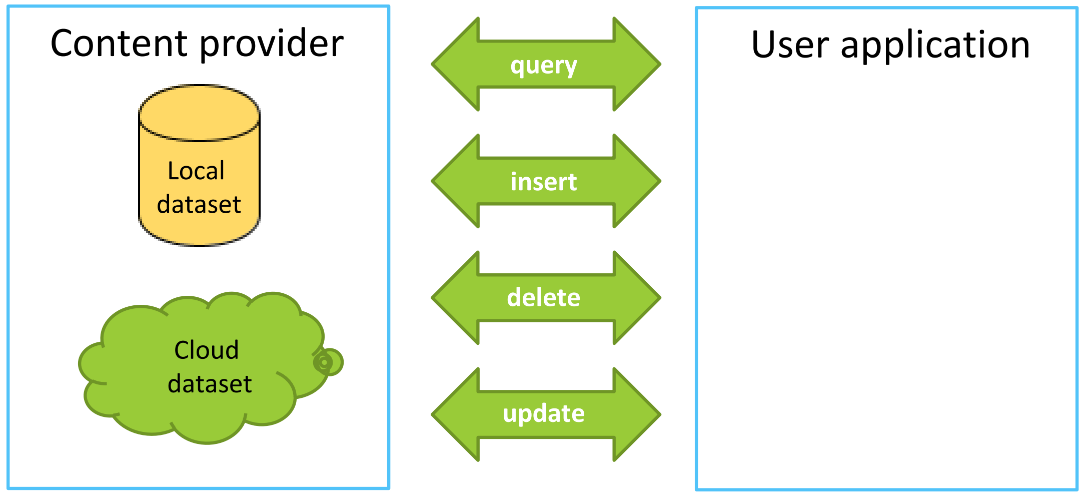
A content provider is a data-centric service making persistent datasets available to any applications.

Common global datasets include contacts, pictures, messages, audio files, emails.

The global datasets are stored in an SQLite database (however developer doesn't have to be an SQL expert)

The content provider class offers a standard set of parametric methods to enable other applications to retrieve, delete, update, and insert data items.

Content provider is a wrapper hiding the actual physical data. Users interact with their data through a common object interface.



COMPONENT’S LIFE CYCLE

Each Android application runs inside its own instance of a Virtual Machine (VM)

At any point in time several parallel VM instances could be active (real parallelism as opposed to task-switching)

Unlike a common Windows or Unix process, an Android application does not completely control the completion of its lifecycle.

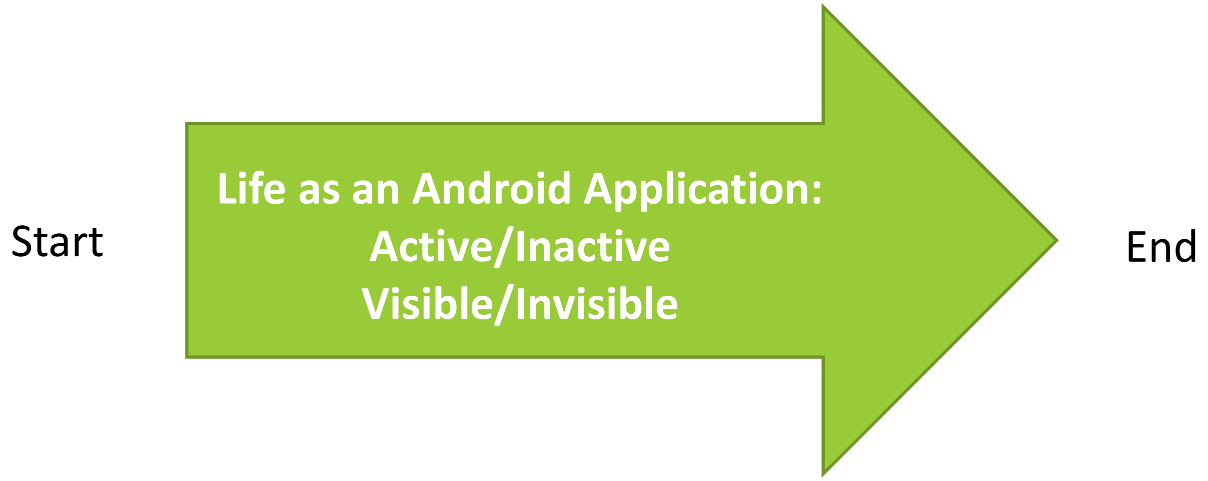
Occasionally hardware resources may become critically low and the OS could order early termination of any process. The decision considers factors such as:

- Number and age of the application’s components currently running,
- Relative importance of those components to the user, and
- How much free memory is available in the system.

COMPONENT'S LIFE CYCLE (Life and death in Android)

All components execute according to a master plan that consists of:

- A beginning - responding to a request to instantiate them
- An end - when the instances are destroyed.
- A sequence of in-between states – components sometimes are active or inactive, or in the case of activities - visible or invisible.



COMPONENT’S LIFE CYCLE (Activity stack)

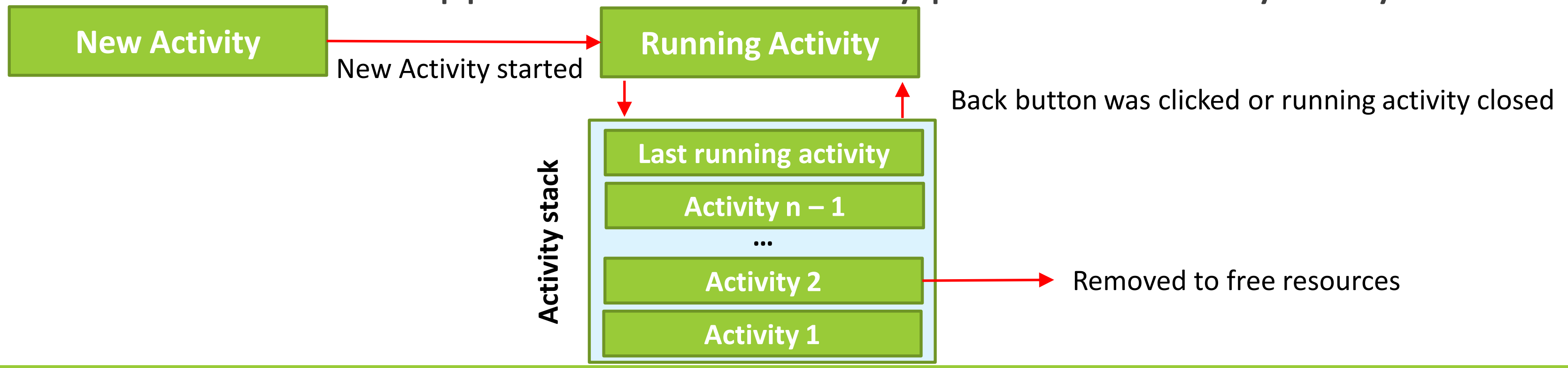
Activities in the system are scheduled using an activity stack.

When a new activity is started, it is placed on top of the stack to become the running activity

Previous activity is pushed-down one level in stack and may come back to foreground once new activity finishes.

If the user presses the Back Button the current activity is terminated and previous activity on stack moves up to become active.

Android 4.0 introduced the ‘Recent app’ button to arbitrarily pick as ‘next’ any entry currently in stack



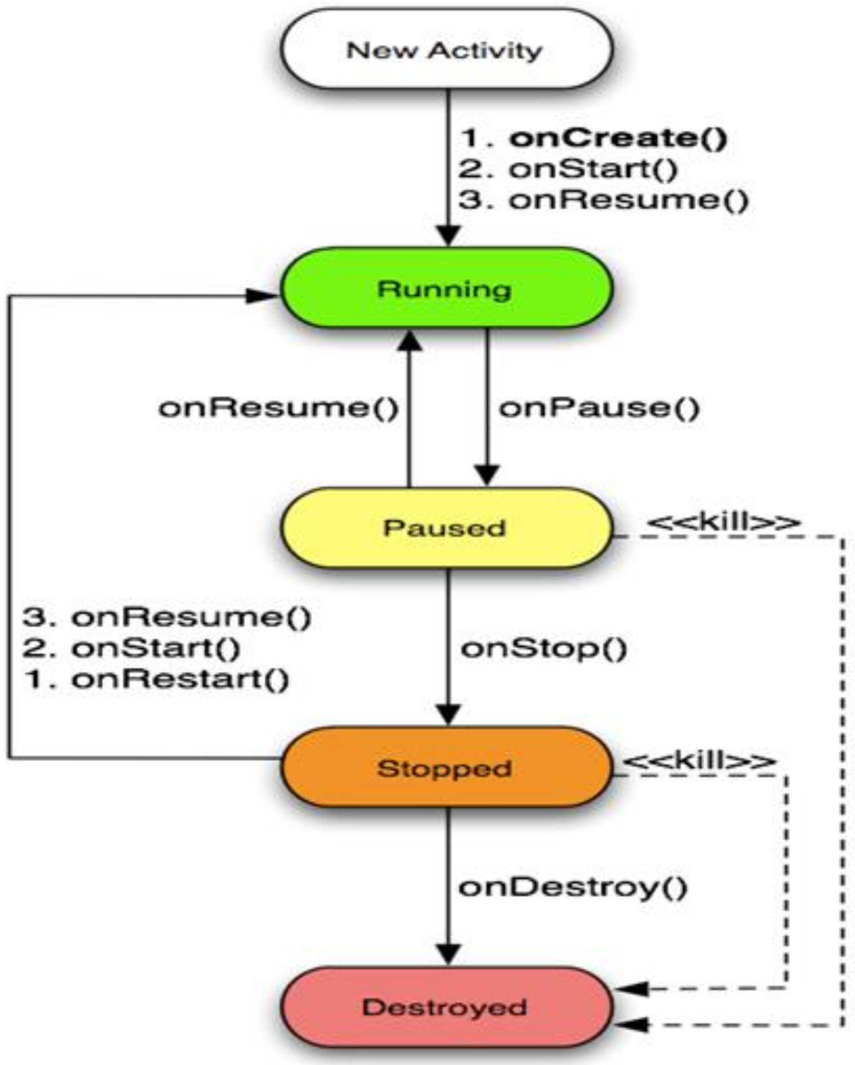
COMPONENT’S LIFE CYCLE (Life cycle callbacks)

When progressing from one state to the other, OS notifies application of the changes by issuing calls to the following protected transition methods:

- void onCreate( ): The activity is being created.
- void onStart( ): The activity is about to become visible.
- void onResume( ): The activity has become visible (it is now “resumed”)
- void onPause( ): Another activity is taking focus (this activity is about to be “paused”)
- void onStop( ): The activity is no longer visible (it is now “stopped”)
- void onDestroy( ): The activity is about to be destroyed

Code:

```
public class ExampleActivity extends Activity {
  @Override
  public void onCreate (Bundle savedInstanceState) { super.onCreate(savedInstanceState); …}
  @Override
  protected void onStart() { super.onStart(); …}
  @Override
  protected void onResume() { super.onResume(); …}
  @Override
  protected void onPause() { super.onPause(); …}
  @Override
  protected void onStop() {super.onStop(); …}
  @Override
  protected void onDestroy() { super.onDestroy(); …}
}
```



COMPONENT'S LIFE CYCLE (Activity states and callback methods)

An activity has essentially three phases:

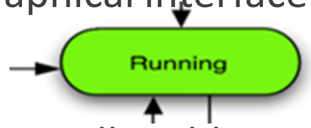
- 1. It is active or running
- 2. It is paused or
- 3. It is stopped .

Moving from one state to the other is accomplished by means of the callback methods listed on the edges of the diagram

COMPONENT'S LIFE CYCLE (Activity state: running & paused & stopped)

Your activity is active or running when it is in foreground of screen (seating on top of the activity stack)

This is the activity that has "focus" and its graphical interface is responsive to the user's interactions.

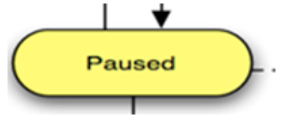


Your Activity is paused if it has lost focus but is still visible to the user.

That is, another activity seats on top and new activity either is transparent or doesn't cover 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.



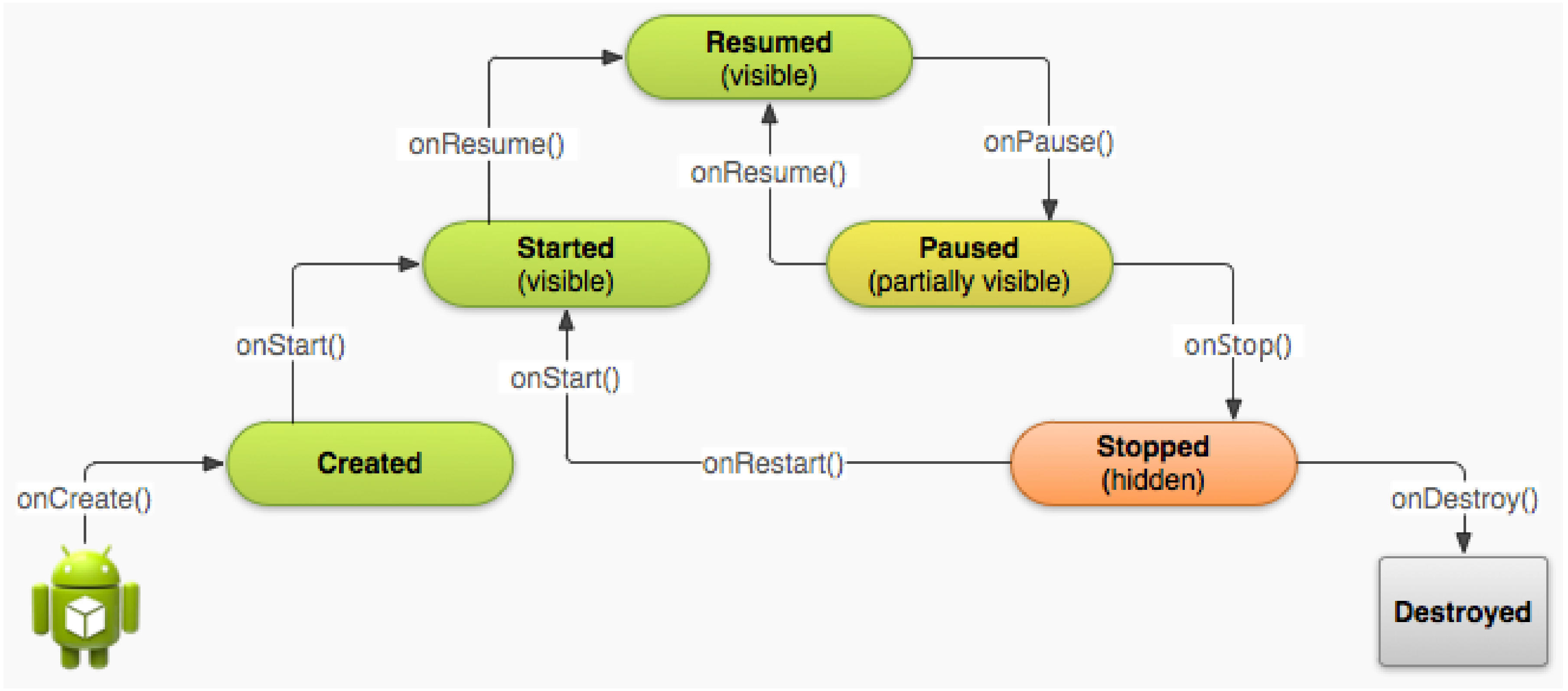
Your Activity is stopped if it is completely obscured by another activity.

Although stopped, it continues to retain all its state information.

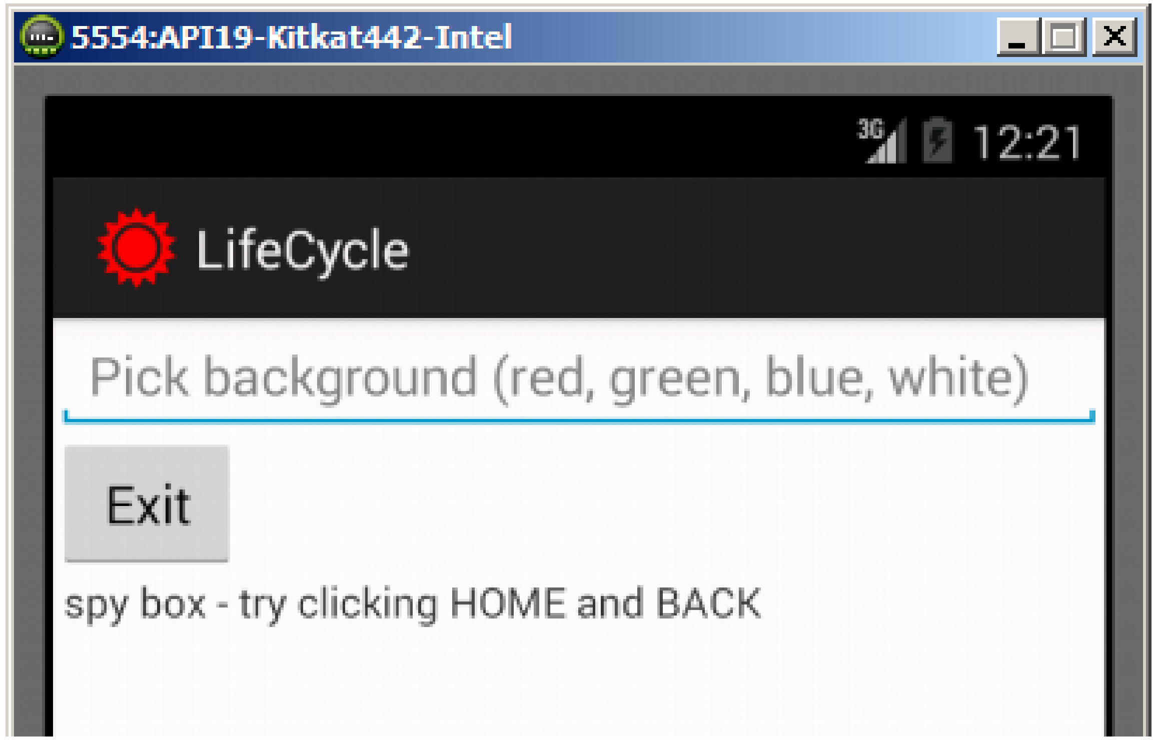
It is no longer visible to the user (its window is hidden, and its life cycle could be terminated at any point by the system if the resources that it holds are needed elsewhere).

COMPONENT'S LIFE CYCLE (Activity life cycle)

Another view of activity life cycle



http://developer.android.com/training/basics/activity-lifecycle/starting.html


5554:API19-Kitkat442-Intel

12:21

LifeCycle

Pick background (red, green, blue, white)

Exit

spy box - try clicking HOME and BACK

COMPONENT’S LIFE CYCLE (Lab1: transitioning one state at the time)

1. Create an Android app (LifeCycle) to show the different states traversed by an application

2. The activity_main.xml layout should include an EditText box (txtMsg), a button (btnExit), and a TextView (txtSpy). Add to the EditText box the hint depicted in the figure on the right.

3. Use the onCreate method to connect the button and textbox to the program. Add the following line of code: Toast.makeText(this, "onCreate", Toast.LENGTH_SHORT).show();

4. The onClick method has only one command: finish(); called to terminate the application.

5. Add a Toast-command (as the one above) to each of the remaining six main events.

6. Save your code

7. Compile and execute application (write messages from here)

8. Press the EXIT button. Observe the sequence of states displayed.

9. Re-execute the application (click green triangle button in IDE)

10. Press emulator’s HOME button. What happens?

11. Click on launch pad, look for the app’s icon and return to the app. What sequence of messages is displayed?

12. Click on the BACK button to return to the application

COMPONENT'S LIFE CYCLE (Lab2-3: call & text emulator-to-emulator with data persistence)

Lab 2:

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

Lab 3:

- 16. 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();
```

- 17. Use the onResume method to add the following fragment

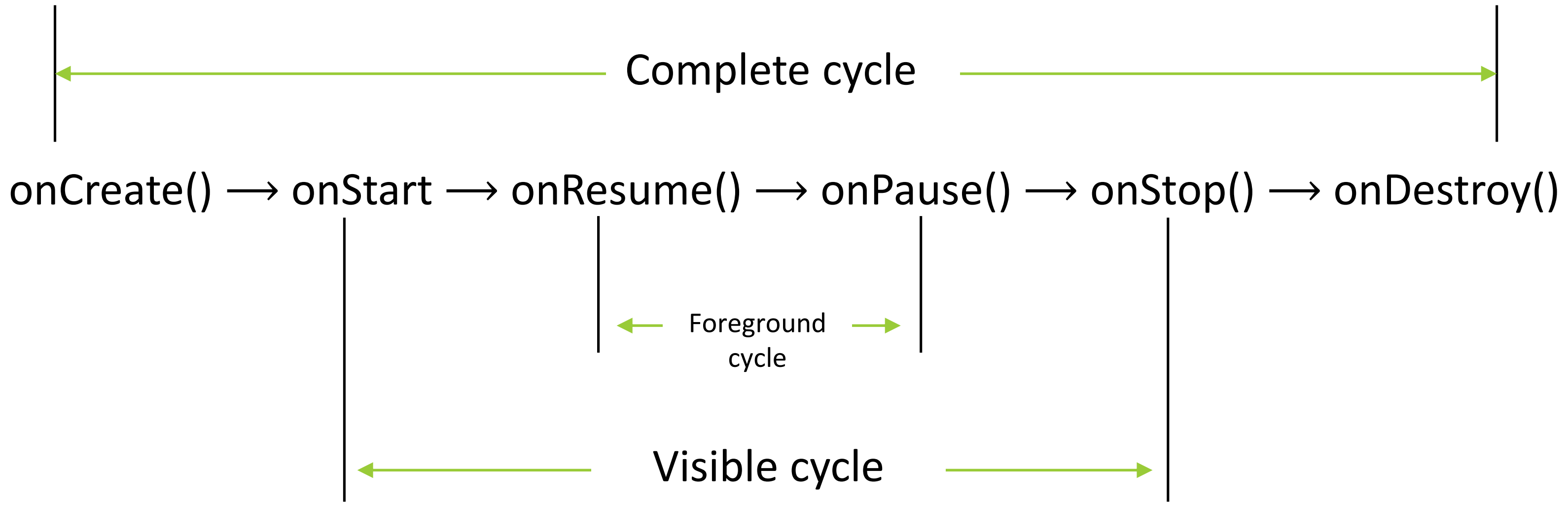
```
SharedPreferences myFile = getSharedPreferences("myFile1", Activity.MODE_PRIVATE);
if ( (myFile != null) && (myFile.contains("mydata")) ) {
  String temp = myFile.getString("mydata", "***");
  txtMsg.setText(temp);
}
```

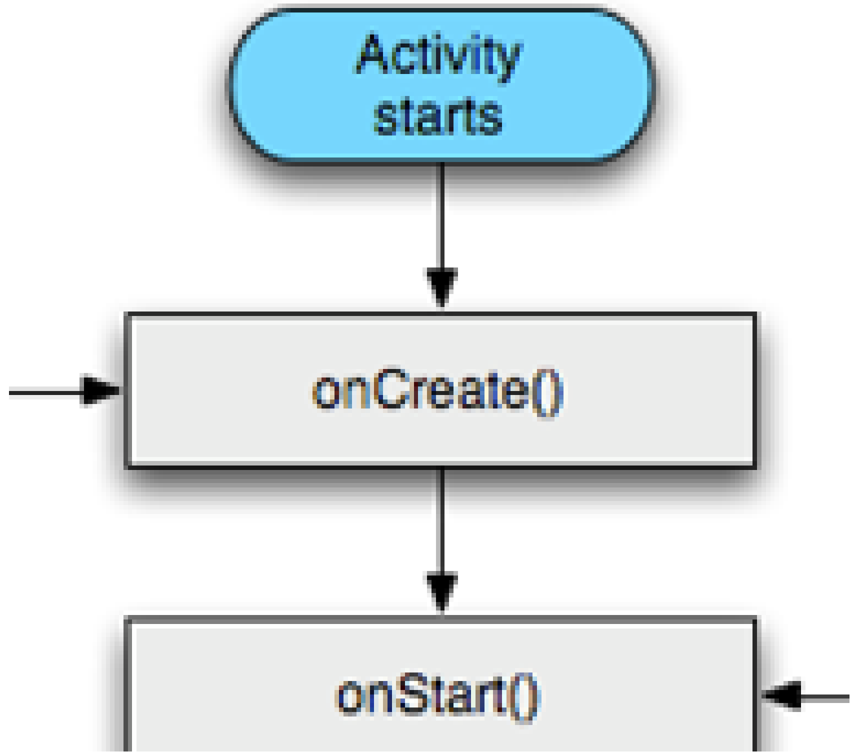
- 18. What happens now with the data previously entered in the text box?

APPLICATION'S LIFE CYCLE (Foreground lifetime)

An activity begins its lifecycle when it enters the onCreate() state.

If it is not interrupted or dismissed, the activity performs its job and finally terminates and releases resources when reaching the onDestroy() event.


Complete cycle
onCreate() ⟶ onStart ⟶ onResume() ⟶ onPause() ⟶ onStop() ⟶ onDestroy()
Foreground cycle
Visible cycle

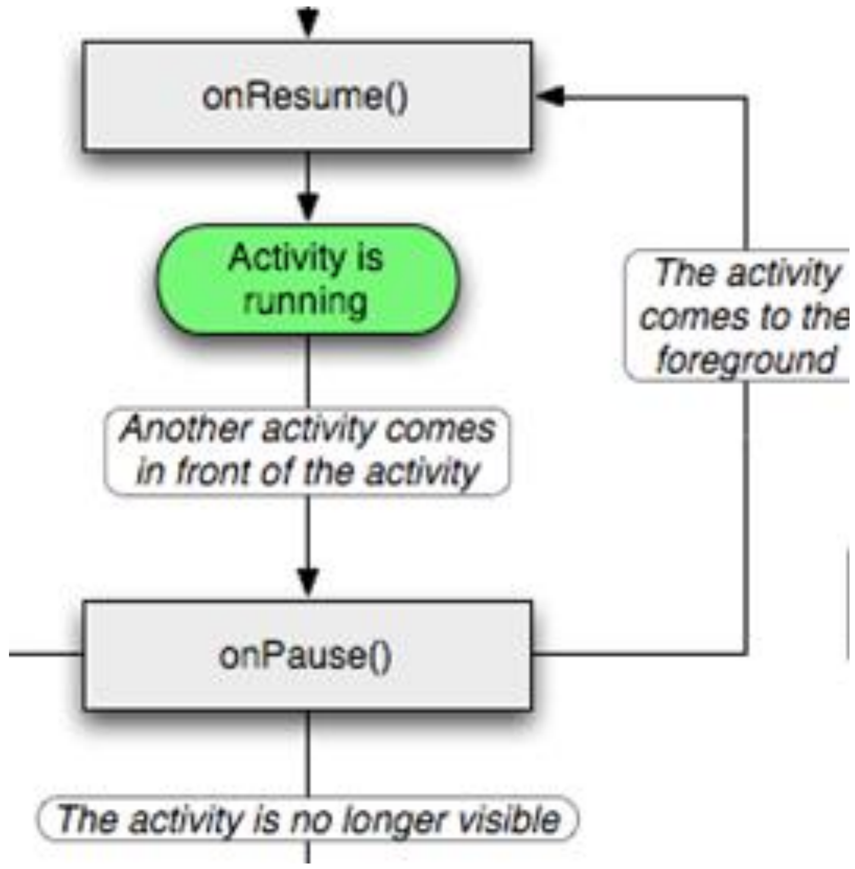


APPLICATION'S LIFE CYCLE (Associating lifecycle events with application's code)

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

- Mandatory
- onCreate() must be implemented by each activity to do its initial setup. The method is executed only once on the activity's lifetime.
- This is the first callback method to be executed when an activity is created.
- Most of your application's code is written here.
- Typically used to initialize the application's data structures, wire-up UI view elements (buttons, text boxes, lists) with local Java controls, define listeners' behavior, etc.
- It may receive a data Bundle object containing the activity's previous state (if any).
- Followed by onStart() ⟶ onResume() ….

APPLICATION'S LIFE CYCLE (Associating lifecycle events with application's code)



onResume()
Activity is running
The activity comes to the foreground
Another activity comes in front of the activity
onPause()
The activity is no longer visible

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

- Highly Recommended
- onPause() should be implemented whenever the application has some important data to be committed so it could be reused.
- Called when the system is about to transfer control to another activity. It should be used to safely write uncommitted data and stop any work in progress.
- 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.

APPLICATION’S LIFE CYCLE (Killable states)

Android OS may terminate a killable app whenever the resources needed to run other operation of higher importance are critically low.

When an activity reaches the methods: onPause(), onStop(), and onDestroy() it becomes killable.

onPause() is the only state that is guaranteed to be given a chance to complete before the process is terminated.

You should use onPause()to write any pending persistent data.

APPLICATION'S LIFE CYCLE (Data persistence using SharedPreferences class)

SharedPreferences is a simple persistence mechanism used to store and retrieve <key,value> pairs, where key is a string and value is a primitive data type

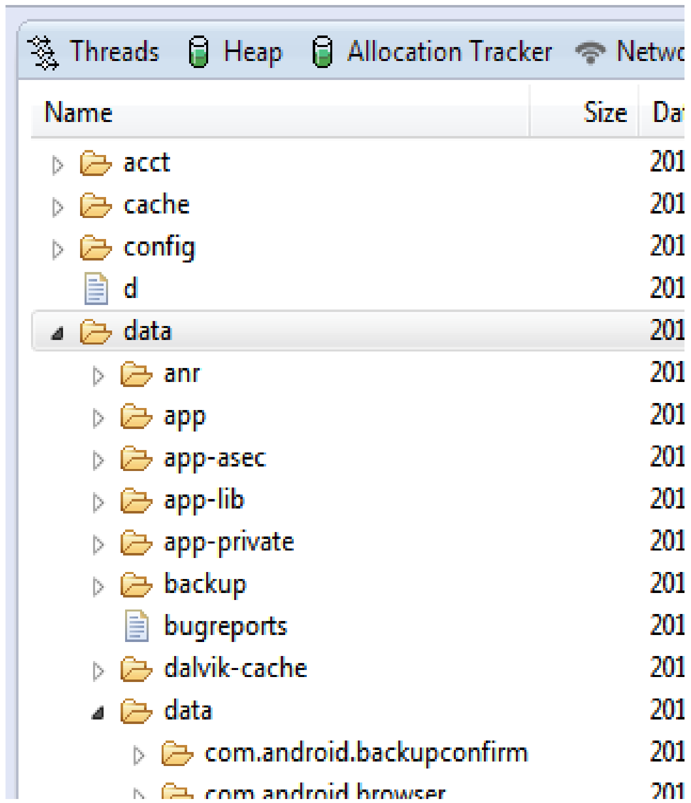
This container class reproduces the structure and behavior of a Java HashMap, however; unlike HashMaps it is persistent.

Appropriate for storing small amounts of state data across sessions.

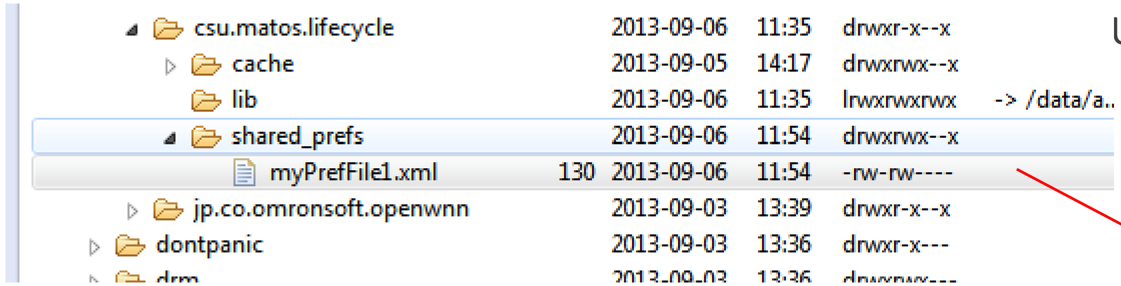
SharedPreferences myPrefSettings = getSharedPreferences(MyPreferrenceFile, actMode);

SharedPreference files are permanently stored in the application's process space.

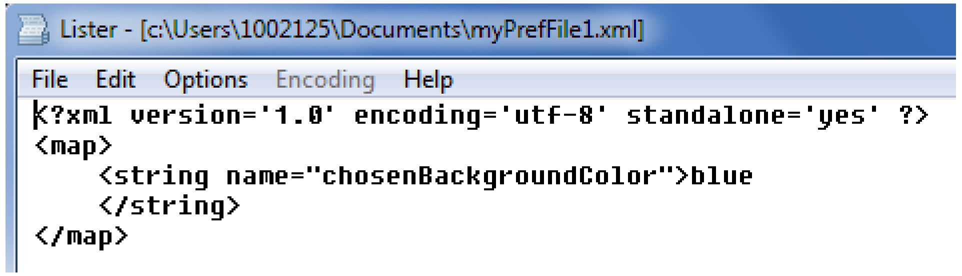
Use file explorer to locate the entry: data/data/your-package-name/shared-prefs



```
Name                                 Size  Da
▷ acct                                     201
▷ cache                                    201
▷ config                                   201
  d                                        201
◢ data                                     201
   ▷ anr                                   201
   ▷ app                                   201
   ▷ app-asec                              201
   ▷ app-lib                               201
   ▷ app-private                           201
   ▷ backup                                201
     bugreports                            201
   ▷ dalvik-cache                          201
   ◢ data                                  201
      ▷ com.android.backupconfirm          201
      ▷ com.android.browser                201
```



```
   ◢ csu.matos.lifecycle          2013-09-06  11:35  drwxr-x--x
      ▷ cache                     2013-09-05  14:17  drwxrwx--x
        lib                       2013-09-06  11:35  lrwxrwxrwx  -> /data/a..
      ◢ shared_prefs              2013-09-06  11:54  drwxrwx--x
           myPrefFile1.xml   130  2013-09-06  11:54  -rw-rw----
   ▷ jp.co.omronsoft.openwnn      2013-09-03  13:39  drwxr-x--x
▷ dontpanic                       2013-09-03  13:36  drwxr-x---
▷ drm                             2013-09-03  13:36  drwxrwx---
```

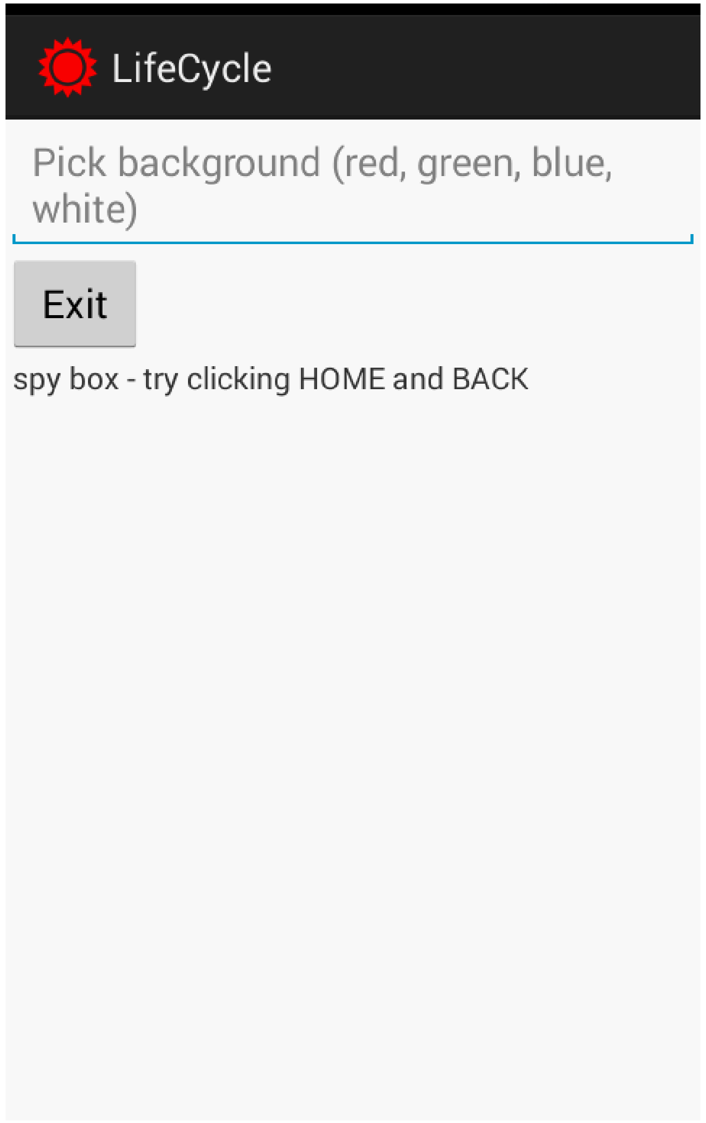


```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="chosenBackgroundColor">blue
    </string>
</map>
```

EXAMPLE

The following application demonstrates the transitioning of a simple activity through the Android’s sequence of Life-Cycle states.

1. A Toast-msg will be displayed showing the current event’s name.
2. An EditText box is provided for the user to indicate a background color.
3. When the activity is paused the selected background color value is saved to a SharedPreferences container.
4. When the application is re-executed the last choice of background color should be applied.
5. An EXIT button should be provideD to terminate the app.
6. You are asked to observe the sequence of messages displayed when the application:
   - 6.1. Loads for the first time
   - 6.2. Is paused after clicking HOME button
   - 6.3. Is re-executed from launch-pad
   - 6.4. Is terminated by pressing BACK and its own EXIT button
   - 6.5. Re-executed after a background color is set


LifeCycle
Pick background (red, green, blue, white)
Exit
spy box - try clicking HOME and BACK

EXAMPLE (The LifeCycle app – layout)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:tools="http://schemas.android.com/tools" android:id="@+id/myScreen1"
              android:layout_width="fill_parent" android:layout_height="fill_parent"
              android:orientation="vertical"
              tools:context=".MainActivity">
 <EditText android:id="@+id/editText1"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:hint="Pick background (red, green, blue, white)"
           android:ems="10" >
  <requestFocus />
 </EditText>
 <Button android:id="@+id/button1"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="Exit" />
 <TextView android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="spy box - try clicking HOME and BACK" />
</LinearLayout>
```

EXAMPLE
(The LifeCycle app – code: MainActivity.java)

```java
package csu.matos.lifecycle;
import java.util.Locale;
. . . //other libraries omitted for brevity
public class MainActivity extends Activity {
 //class variables
 private Context context;
 private int duration = Toast.LENGTH_SHORT;
 //PLUMBING: Pairing GUI controls with Java objects
 private Button btnExit;
 private EditText txtColorSelected;
 private TextView txtSpyBox;
 private LinearLayout myScreen;
 private String PREFNAME = “myPrefFile1”;
 @Override
 protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  //display the main screen
  setContentView(R.layout.activity_main);
  //wiring GUI controls and matching Java objects
  txtColorSelected = (EditText)findViewById(R.id.editText1);
  btnExit = (Button) findViewById(R.id.button1);
  txtSpyBox = (TextView)findViewById(R.id.textView1);
  myScreen = (LinearLayout)findViewById(R.id.myScreen1);
```

```java
  //set GUI listeners, watchers,...
  btnExit.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View v) { finish();}
  });
  //observe (text) changes made to EditText box (color selection)
  txtColorSelected.addTextChangedListener(new TextWatcher() {
   @Override
   public void onTextChanged(CharSequence s, int start, int before, int count) { /* nothing TODO, needed by interface */ }
   @Override
   public void beforeTextChanged(CharSequence s, int start, int count, int after) { /* nothing TODO, needed by interface */ }
   @Override
   public void afterTextChanged(Editable s) {
    //set background to selected color
    String chosenColor = s.toString().toLowerCase(Locale.US);
    txtSpyBox.setText(chosenColor);
    setBackgroundColor(chosenColor, myScreen);
   }
  });
  //show the current state's name
  context = getApplicationContext();
  Toast.makeText(context, "onCreate", duration).show();
 } //onCreate
```

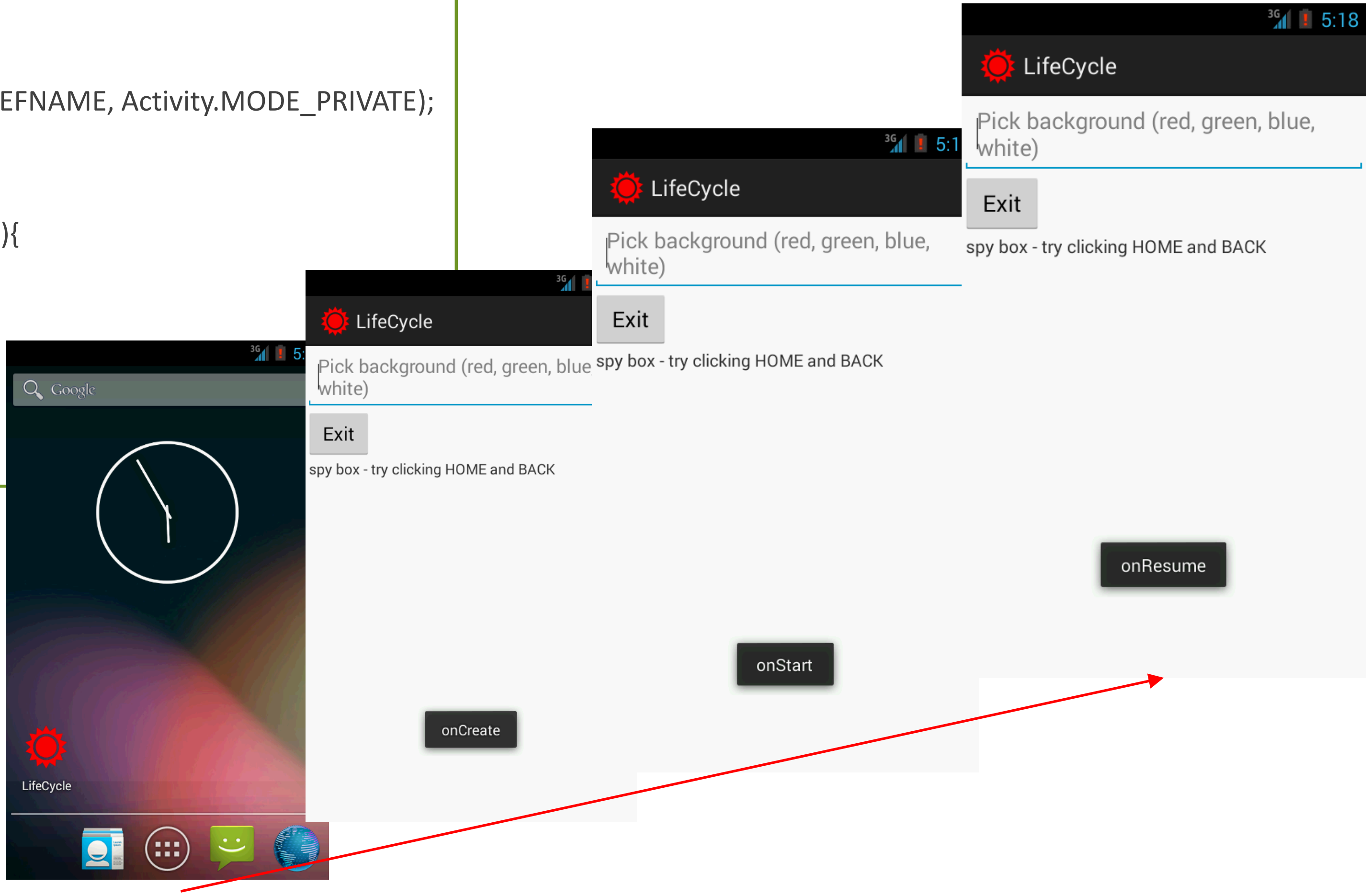
EXAMPLE
(The LifeCycle app – code: MainActivity.java)

```
@Override
protected void onDestroy() {
  super.onDestroy();
  Toast.makeText(context, “onDestroy”, duration).show();
}
@Override
protected void onPause() {
  super.onPause();
  //save state data (background color) for future use
  String chosenColor = txtSpyBox.getText().toString();
  saveStateData(chosenColor);
  Toast.makeText(context, “onPause”, duration).show();
}
@Override
protected void onRestart() {
  super.onRestart();
  Toast.makeText(context, “onRestart”, duration).show();
}
@Override
protected void onResume() {
  super.onResume();
  Toast.makeText(context, “onResume”, duration).show();
}
```

```
@Override
protected void onStart() { super.onStart();
  //if appropriate, change background color to chosen value
  updateMeUsingSavedStateData();
  Toast.makeText(context, “onStart”, duration).show();
}
@Override
protected void onStop() { super.onStop(); Toast.makeText(context, “onStop”, duration).show(); }
private void setBackgroundColor(String chosenColor, LinearLayout myScreen) {
  //hex color codes: 0xAARRGGBB AA:transp, RR red, GG green, BB blue
  if (chosenColor.contains(“red”)) myScreen.setBackgroundColor(0xffff0000); //Color.RED
  if (chosenColor.contains(“green”)) myScreen.setBackgroundColor(0xff00ff00); //Color.GREEN
  if (chosenColor.contains(“blue”)) myScreen.setBackgroundColor(0xff0000ff); //Color.BLUE
  if (chosenColor.contains(“white”)) myScreen.setBackgroundColor(0xffffffff); //Color.WHITE
} //setBackgroundColor
private void saveStateData(String chosenColor) {
  //this is a little <key,value> table permanently kept in memory
  SharedPreferences myPrefContainer = getSharedPreferences(PREFNAME, Activity.MODE_PRIVATE);
  //pair <key,value> to be stored represents our 'important' data
  SharedPreferences.Editor myPrefEditor = myPrefContainer.edit();
  String key = “chosenBackgroundColor”, value = txtSpyBox.getText().toString();
  myPrefEditor.putString(key, value);
  myPrefEditor.commit();
}//saveStateData
```

EXAMPLE
(The LifeCycle app – code: MainActivity.java)

```
private void updateMeUsingSavedStateData() {
  // (in case it exists) use saved data telling backg color
  SharedPreferences myPrefContainer = getSharedPreferences(PREFNAME, Activity.MODE_PRIVATE);
  String key = "chosenBackgroundColor";
  String defaultValue = "white";
  if (( myPrefContainer != null ) && myPrefContainer.contains(key)){
    String color = myPrefContainer.getString(key, defaultValue);
   setBackgroundColor(color, myScreen);
  }
}//updateMeUsingSavedStateData
} //Activity
```

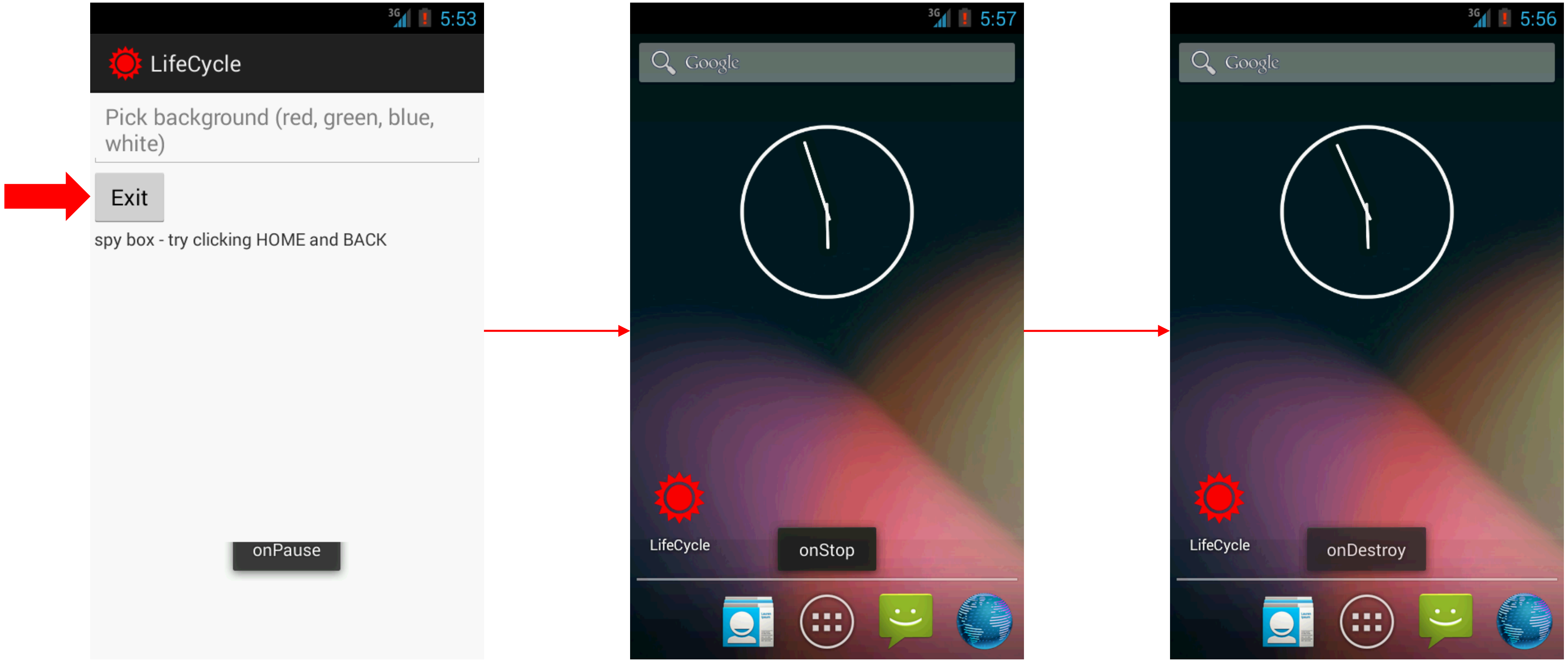

Google
LifeCycle
LifeCycle
Pick background (red, green, blue white)
Exit
onCreate
5:1
LifeCycle
Pick background (red, green, blue, white)
Exit
onStart
5:18
LifeCycle
Pick background (red, green, blue, white)
Exit
onResume

spy box - try clicking HOME and BACK

spy box - try clicking HOME and BACK

spy box - try clicking HOME and BACK

EXAMPLE
(The LifeCycle app – code: MainActivity.java)



Pick background (red, green, blue, white)

spy box - try clicking HOME and BACK

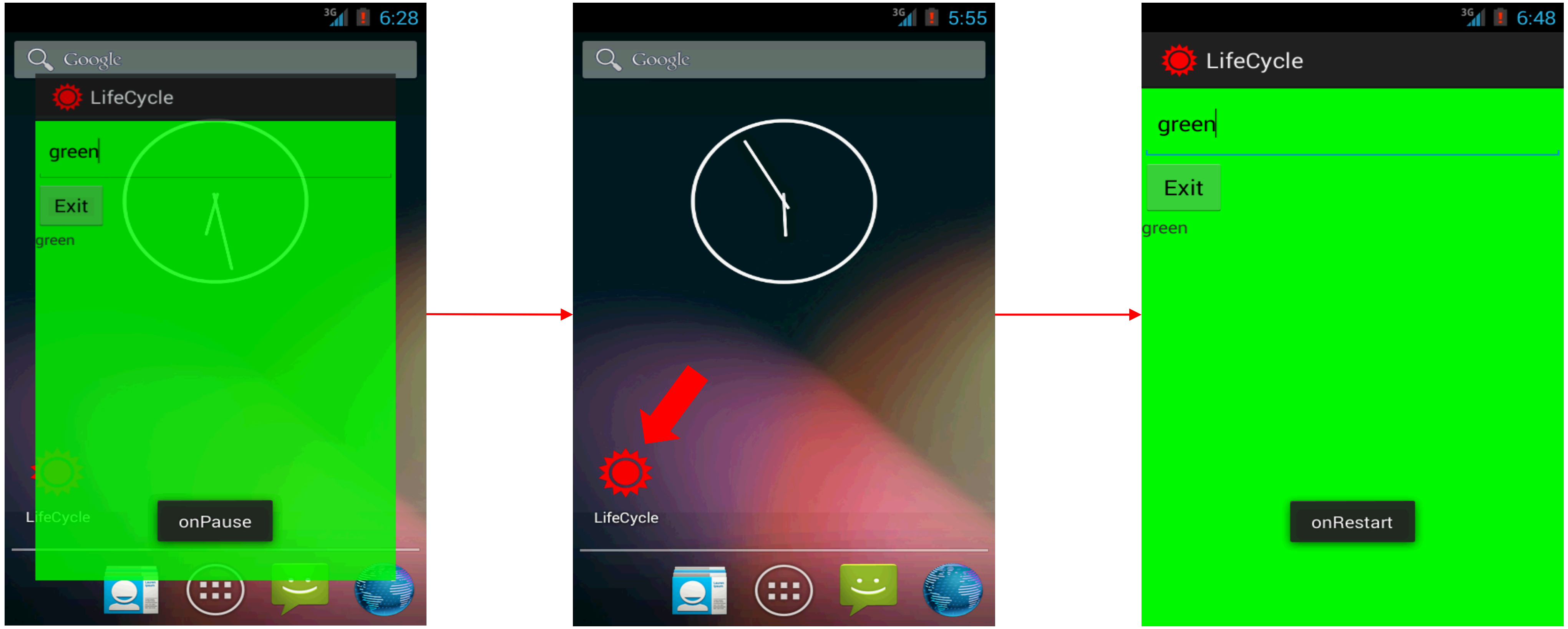
onPause

onStop

onDestroy

EXAMPLE
(The LifeCycle app – code: MainActivity.java)

User selects a green background and clicks the HOME key. When the app is paused the user's selection is saved, the app is still active, but it is not visible.



APPENDIX

Using Bundles to save/restore State Values

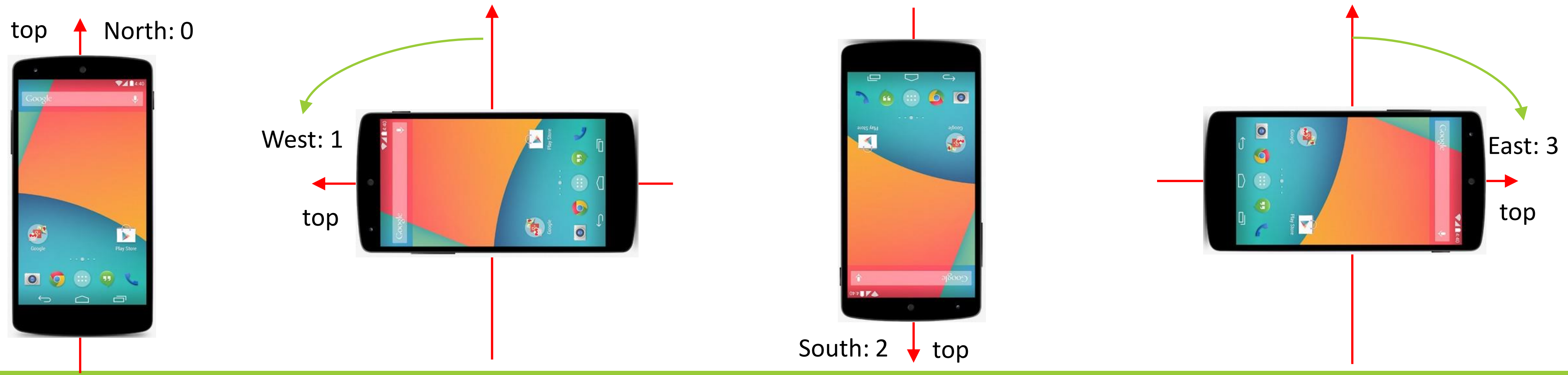
```
@Override
public void onCreate(Bundle savedInstanceState) {
 ...
 if ( savedInstanceState != null )
   String someStrValue = savedInstanceState.getString(“STR_KEY”, “Default”);
   ...
 }
 @Override
 public void onSaveInstanceState(Bundle outState) {
   ...
   outState.putString(“STR_KEY”, “blah blah blah”);
   super.onSaveInstanceState(myBundle);
   ...
}
```

Note: This approach works well when Android kills the app (like in a device-rotation event), however; it will not create the state bundle when the user kills the app (eg. pressing BackButton). Hint: It is a better practice to save state using SharedPreferences in the onPause( ) method.

APPENDIX

The function below allows to obtain the current ORIENTATION of the device as NORTH(0), WEST(1), SOUTH(2) and EAST(3).

```
private int getOrientation(){
  // the TOP of the device points to [0:North, 1:West, 2:South, 3:East]
  Display display = ((WindowManager) getApplication().getSystemService(Context.WINDOW_SERVICE)).getDefaultDisplay();
  display.getRotation();
  return display.getRotation();
}
```



APPENDIX

Use the onCreate method to initialize a control variable with the original device's orientation. During onPause compare the current orientation with its original value; if they are not the same then the device was rotated.

```
int originalOrientation; //used to detect orientation change
@Override
protected void onCreate(Bundle savedInstanceState) {
  ...
  setContentView(R.layout.activity_main);
  originalOrientation = getOrientation();
  ...
  }
  @Override
  protected void onPause() {
   super.onPause();
   if( getOrientation() != originalOrientation ){
     // Orientation changed - phone was rotated
     // put a flag in outBundle, call onSaveInstanceState(…)
   }else {
     // no orientation change detected in the session
   }
}
```