Lesson 3

Application's Life Cycle

Victor Matos
Cleveland State University

Anatomy of Android Applications

An Android application consists of one or more *core components*.

In the case of apps made of multiple parts, collaboration among the independent core components is required for the success of the application.

A core component can be:

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

Anatomy of Android Applications

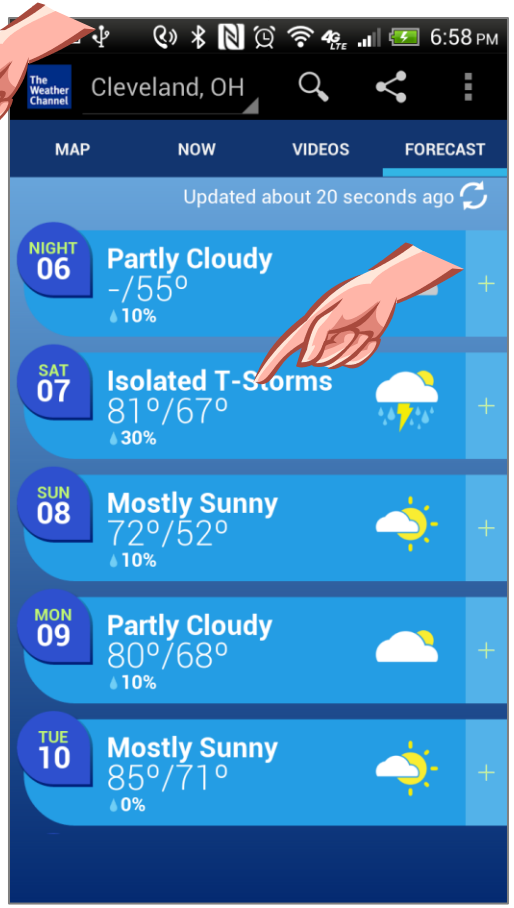
1. Activity

- A typical Android **application** consists of *one or more* **activities**.
- An activity is roughly equivalent to a Windows-Form .
- An *activity* usually shows a *single visual user interface* (GUI).
- Only one activity (known as *main*) is chosen to be executed first when the application is launched.
- An activity may transfer control and data to another activity through an interprocess communication protocol called ***intents***.

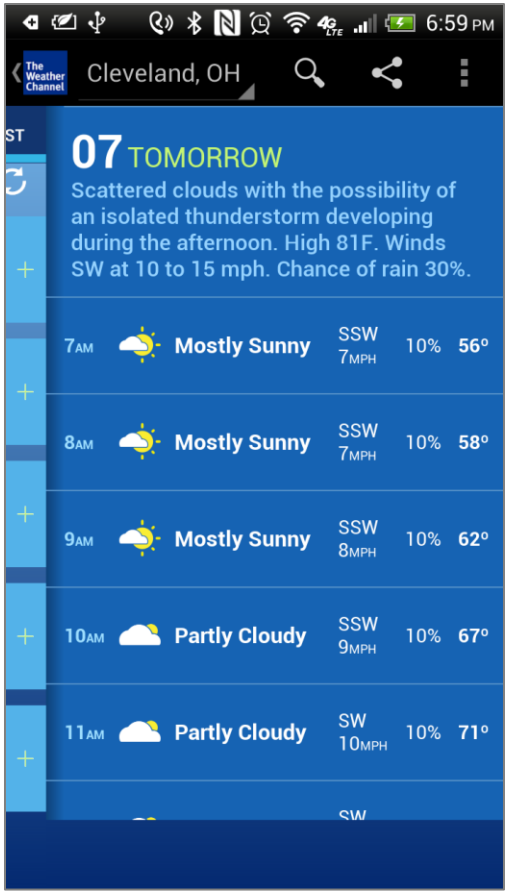
Anatomy of Android Applications

Weather Channel app
GUI-1- Activity 1

Weather Channel app
GUI-2- Activity 2



Weather Channel app
GUI-3- Activity 3

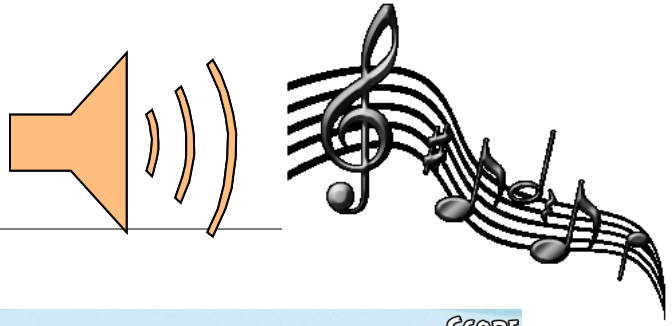


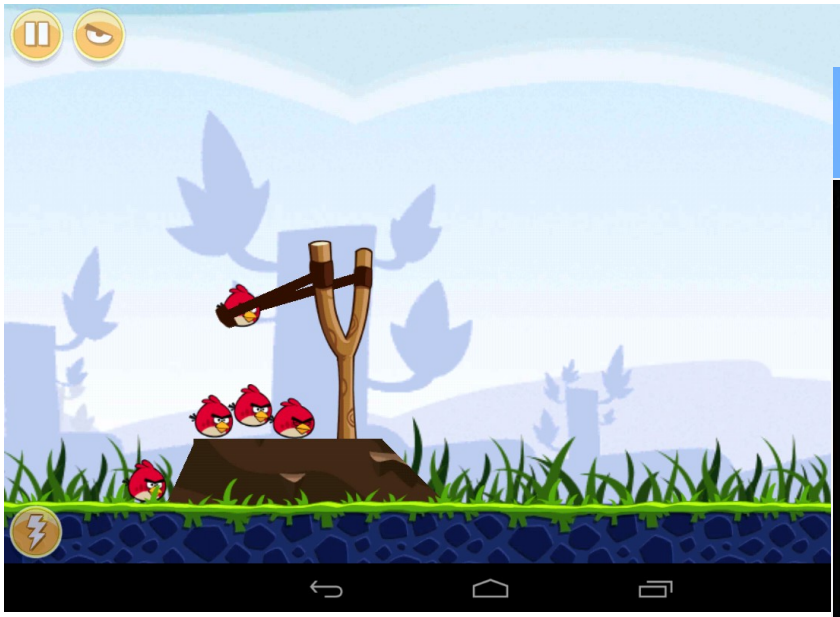
Anatomy of Android Applications

2. Service

- Services are a special type of activity that *do not have a visual user interface.*
- Services usually run in the background 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 inconspicuosly run in the backgroud detecting satellites, phone towers or wi-fi routers location information. The service periodically broadcast location coordinates to any application listening for that kind of data. An application may opt for binding to the running GPS service.

2. Example: Service



Background	Foreground
PANDORA GPS	 

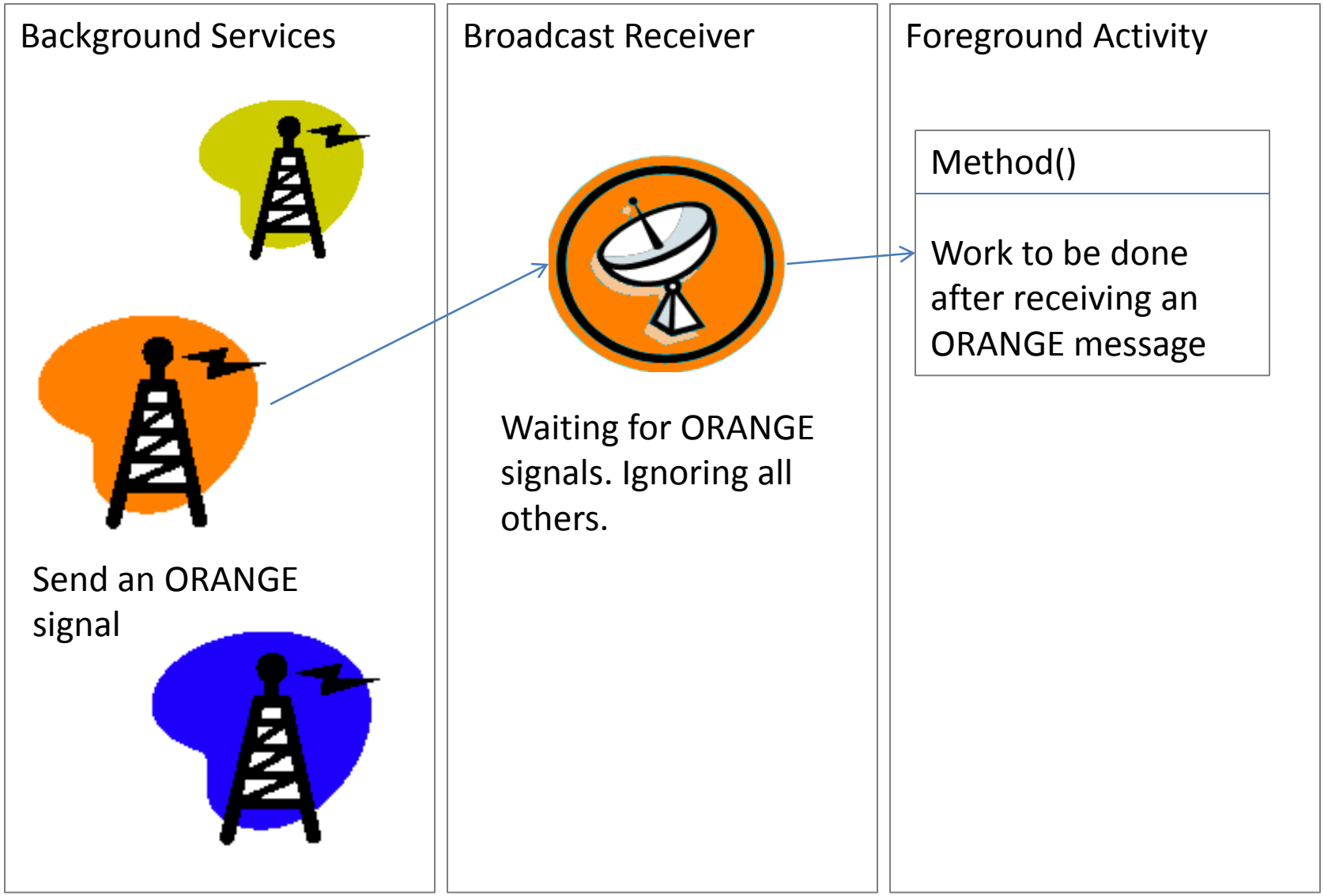
In this example a music service (Pandora Radio) and GPS location run in the background. The selected music station is heard while other GUIs are show on the device's screen. For instance, our user –an avid golfer- may switch between occasional golf course reading (using the GolfShot app) and "Angry Birds" (some of his playing partners could be very slow).

Anatomy of Android Applications

3. Broadcast receiver

- A **BroadcastReceiver** is a dedicated listener that waits for system-wide or locally transmitted messages.
- *Broadcast receivers do not display a user interface*.
- They tipically 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.

3. Broadcast receiver

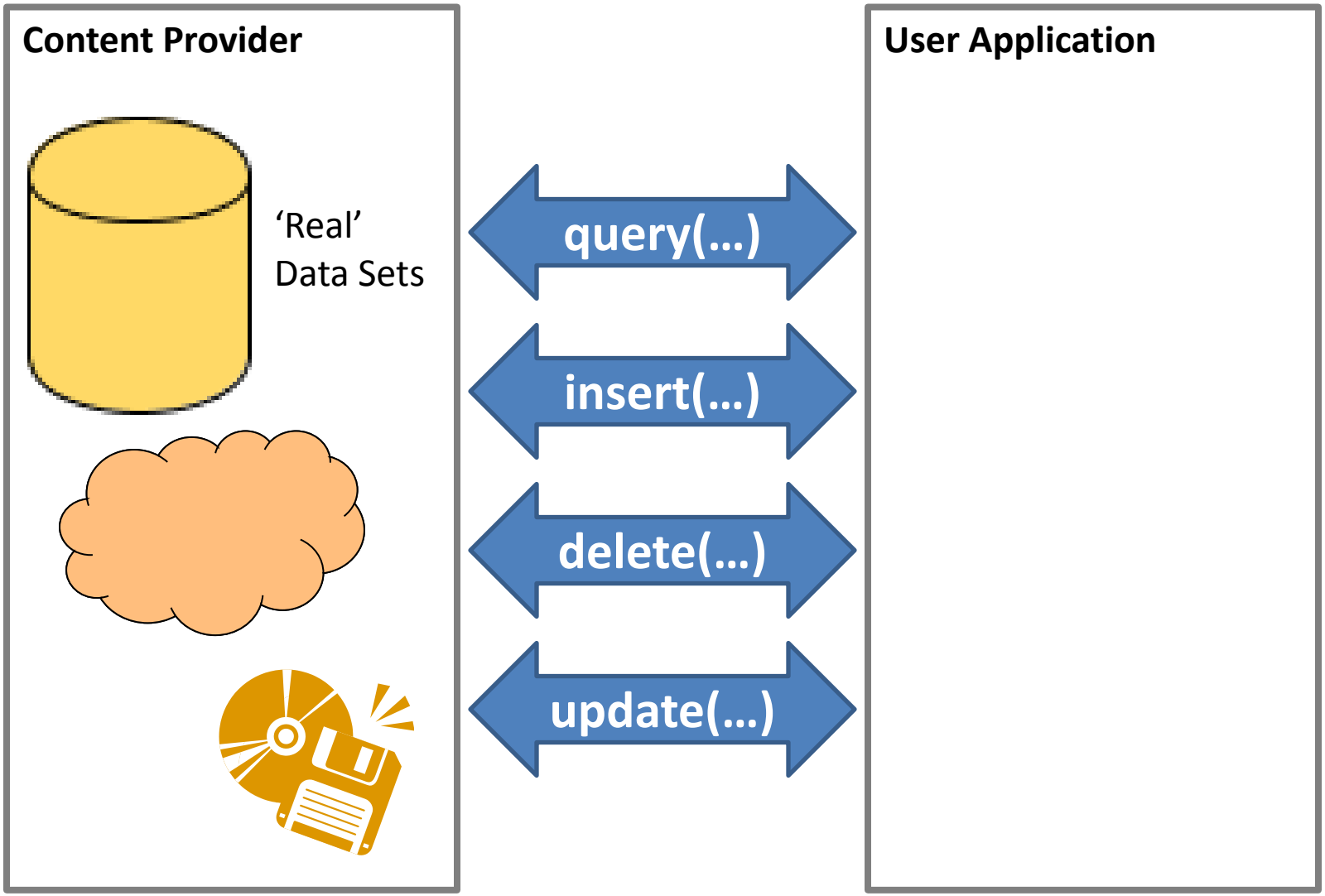


Anatomy of Android Applications

4. Content provider

- A *content provider* is a data-centric service that makes persistent datasets available to any number of applications.
- Common global datasets include: contacts, pictures, messages, audio files, emails.
- The global datasets are usually stored in a SQLite database (however the developer does not need to be an SQLexpert)
- The content provider class offers a standard set of “database-like“ methods to enable other applications to retrieve, delete, update, and insert data items.

4. Content provider



A Content Provider is a wrapper that hides the actual physical data. Users interact with their data through a common object interface.

Application's Life Cycle

Each Android application runs inside its own instance of a Dalvik Virtual Machine (DVM).

At any point in time several parallel DVM instances could be active.

Unlike a common Windows or Unix process, an Android application does not *completely* controls 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:

1. Number and age of the application's components currently running,
2. relative importance of those components to the user, and
3. how much free memory is available in the system.

Component Lifecycles

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

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

Activty 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
- The previous activity is pushed-down one level in the stack, and may come back to the foreground once the new activity finishes.
- If the user presses the *Back Button*  the current activity is terminated and the next activity on the stack moves up to become active.

Activity Stack

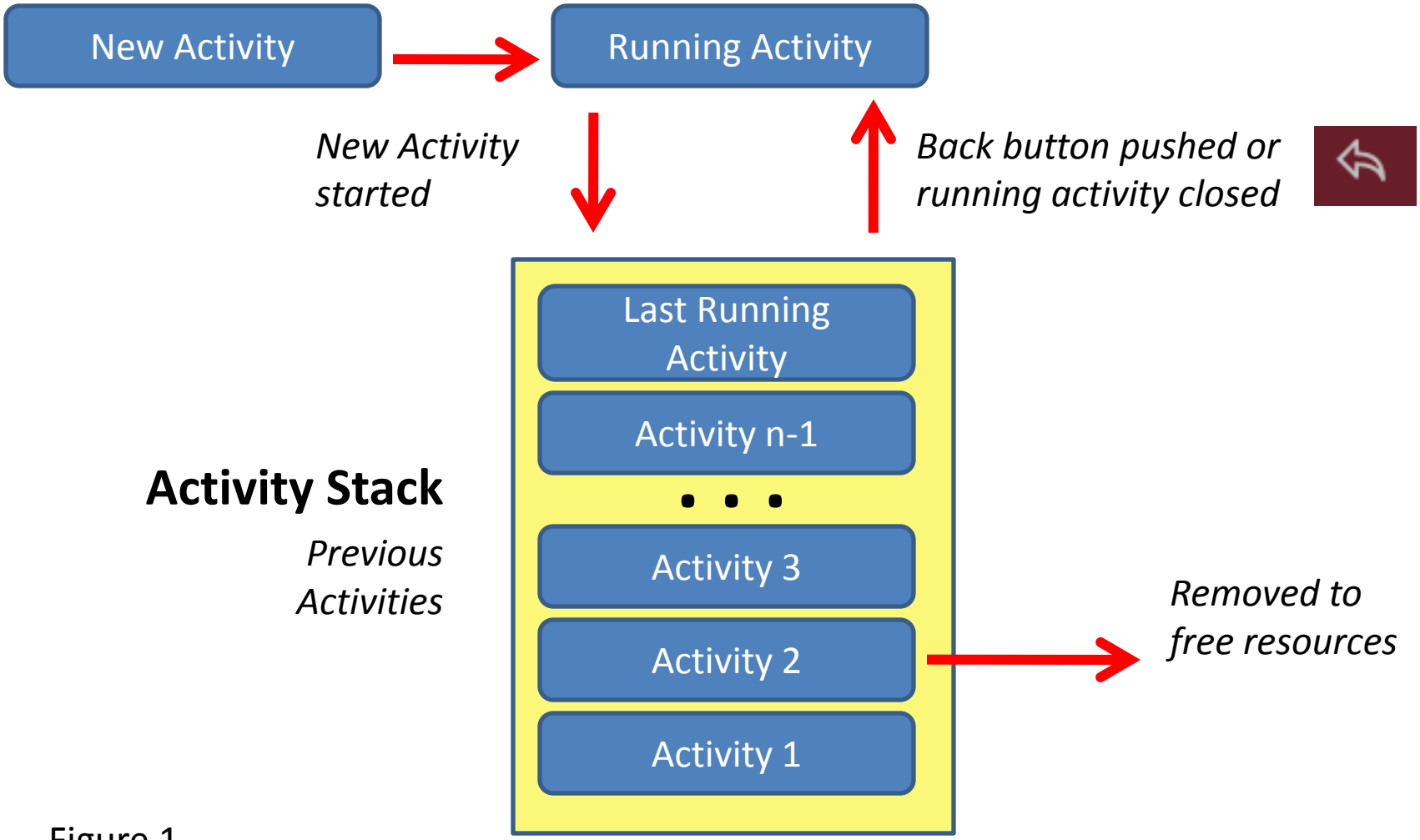


Figure 1.

Life Cycle Events

Life Cycle States

When progressing from one state to the other, the OS notifies the application of the changes by issuing 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 Callbacks

Most of your code goes here

Save your important data here

```
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.
    }
}
```

Reference:
http://developer.android.com/reference/android/app/Activity.html

Life Cycle States

An activity has essentially three states:

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

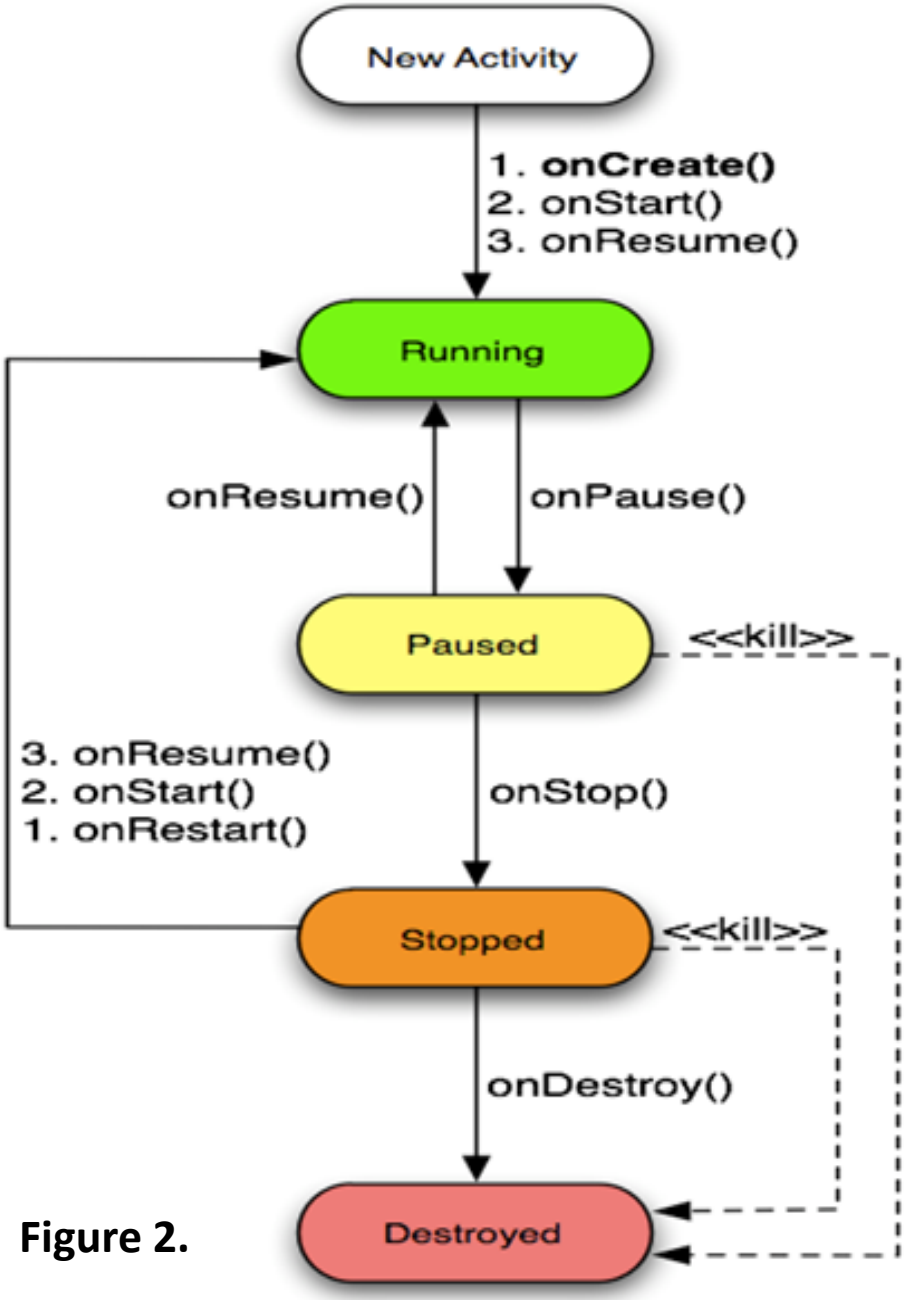


Figure 2.

Image from: http://ganiworldofandroid.blogspot.com/2011/07/complete-understanding-of-activity-life.html

Life Cycle States

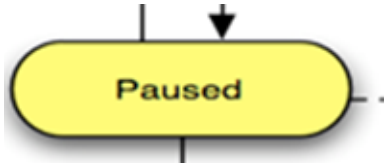
An activity has essentially three states:

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

   This is the activity that has “focus” and its graphical interface is responsive to the user’s interactions.

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 seats 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

An activity has essentially three states (cont.):

3. It is ***stopped*** if it is 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).

Application's Life Cycle

Teaching notes

Your turn!
EXPERIMENT 1.

1. Write an Android app 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 (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.
5. 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…
6. 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 using 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 the app's icon and return to the 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

Teaching notes

Your turn!
EXPERIMENT 3

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();
```

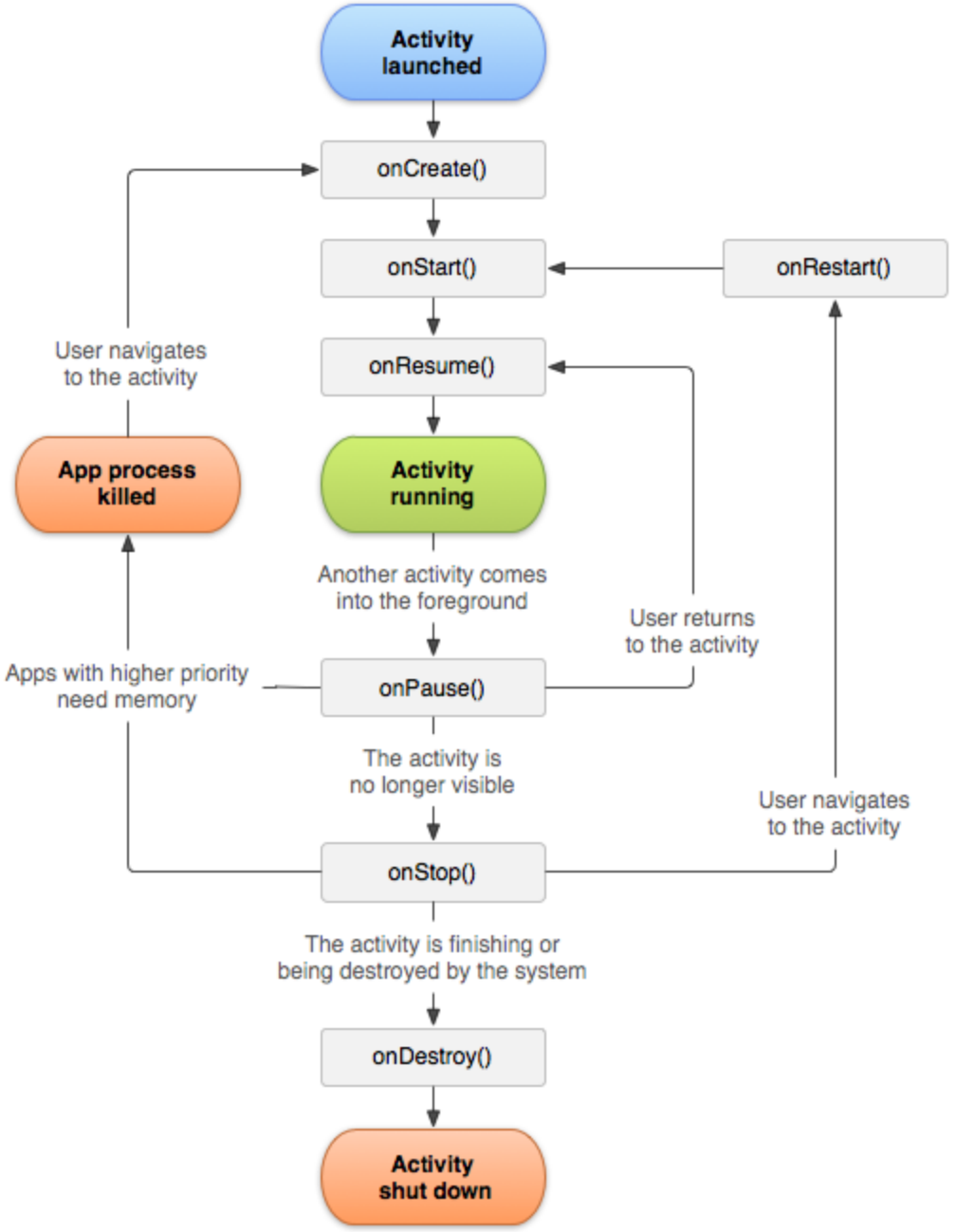
18. Use the **onResume** method to add the following frament

```
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?

Application's Life Cycle

Figure 3.

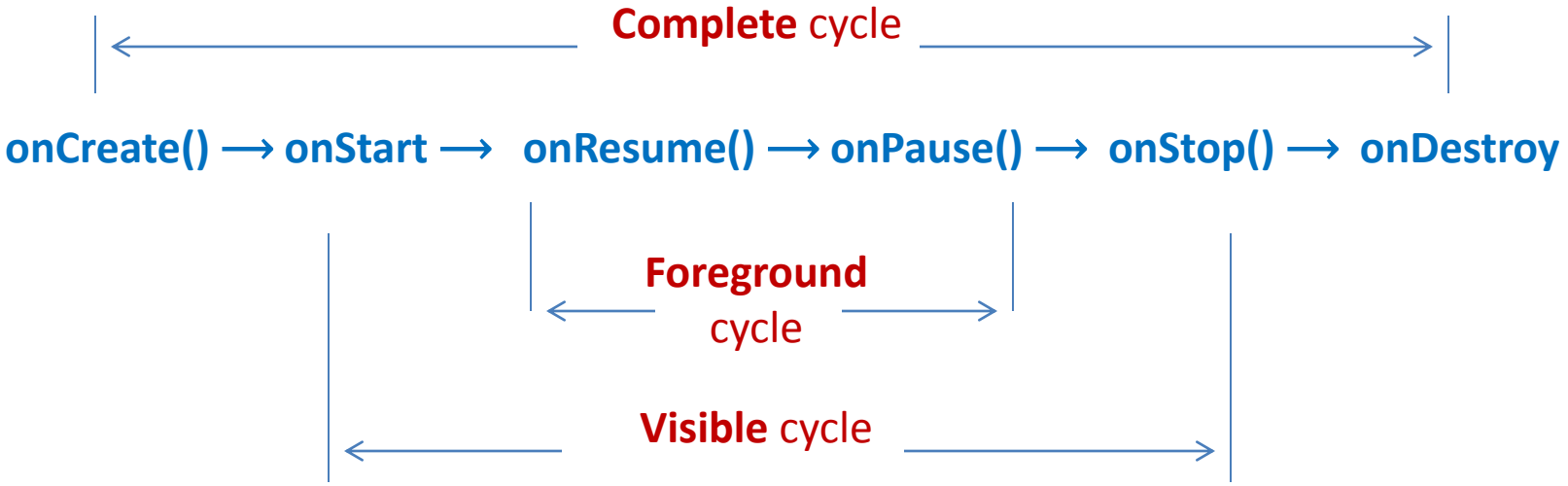


Reference:
http://developer.android.com/reference/android/app/Activity.html

Application's Lifetime

Complete / Visible / Foreground 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.



Life Cycle Events

Associating Lifecycle Events with Application's Code

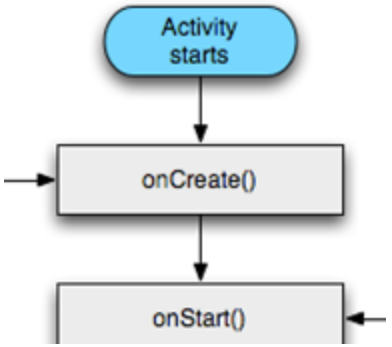
Applications do not need to implement each of the transition methods, however there are mandatory and recommended states to consider

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

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

Life Cycle Methods

Method: **onCreate()**

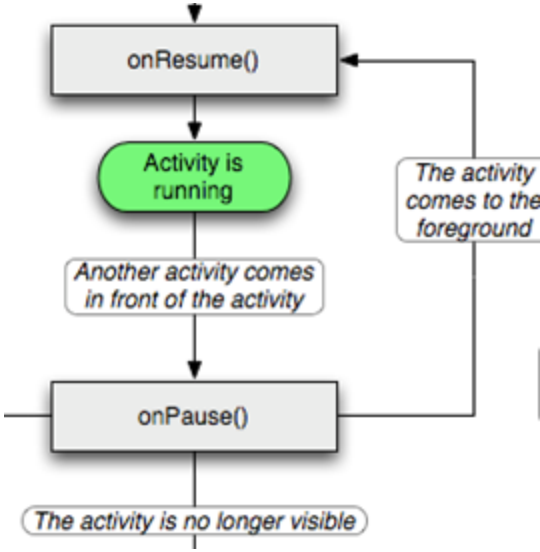


- Called when the activity is first created.
- Most of your application's code is written here.
- Typically used to define listener's behavior, initialize data structures, wire-up UI view elements (buttons, text boxes, lists) with local Java controls, etc.
- It may receive a data *Bundle* object containing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

Life Cycle Methods

Method: **onPause()**

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



Life Cycle Methods

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.

Life Cycle Methods

As an aside…

Android Preferences

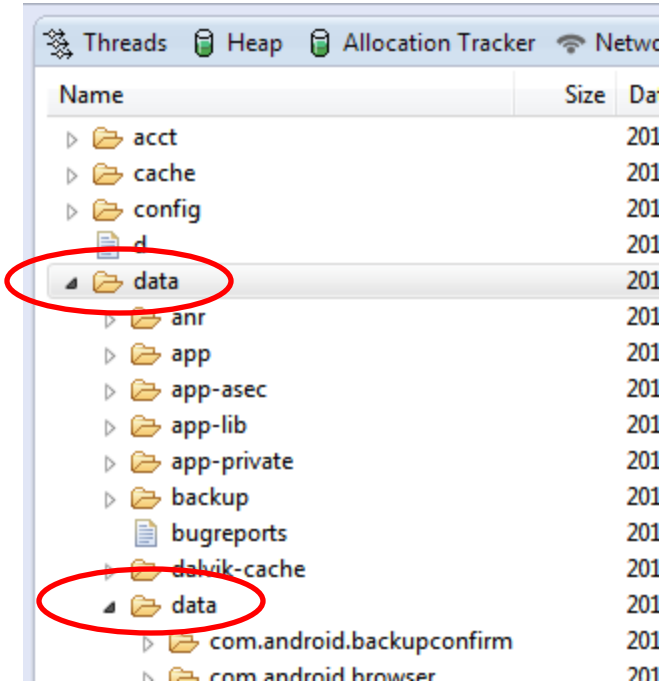
Preferences is a simple Android *persistence mechanism* used to store and retrieve **<key,value>** pairs, where **key** is a string and **value** is a primitive data type. Similar to a Java HashMap. Appropriate for storing small amounts of state data.

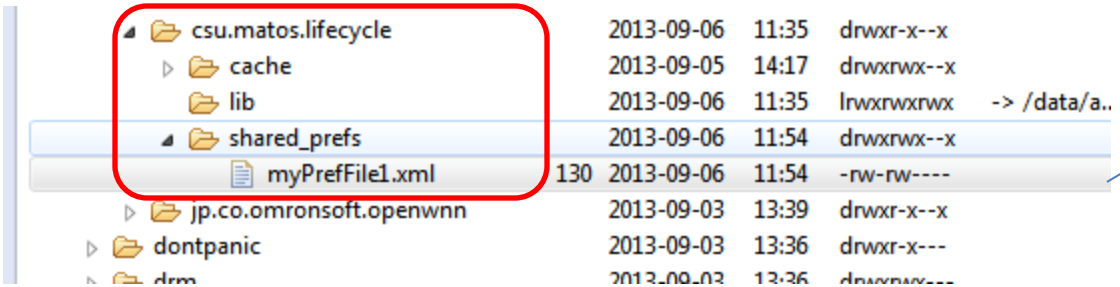
```
SharedPreferences myPrefSettings =
        getSharedPreferences(MyPreferrenceFile, actMode);
```

- A named *preferences file* could be shared with other components in the *same* application.
- actMode set to `Activity.MODE_PRIVATE` indicates that you cannot share the file across applications.

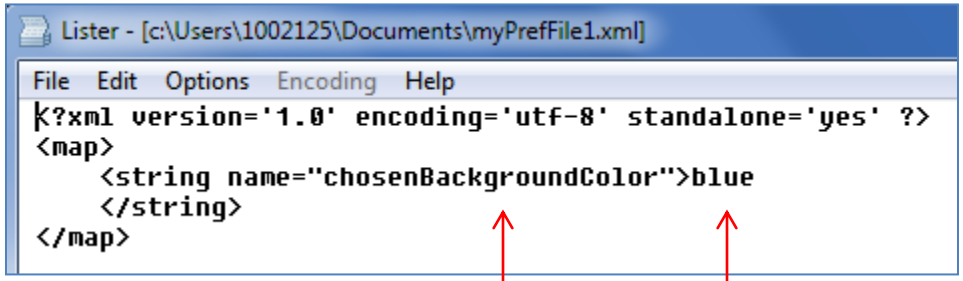
As an aside…

Android Preferences





SharedPreference files are permanently stored in the application's process space.
Use DDMS file explorer to locate the entry:
data/data/your-package-name/shared-prefs



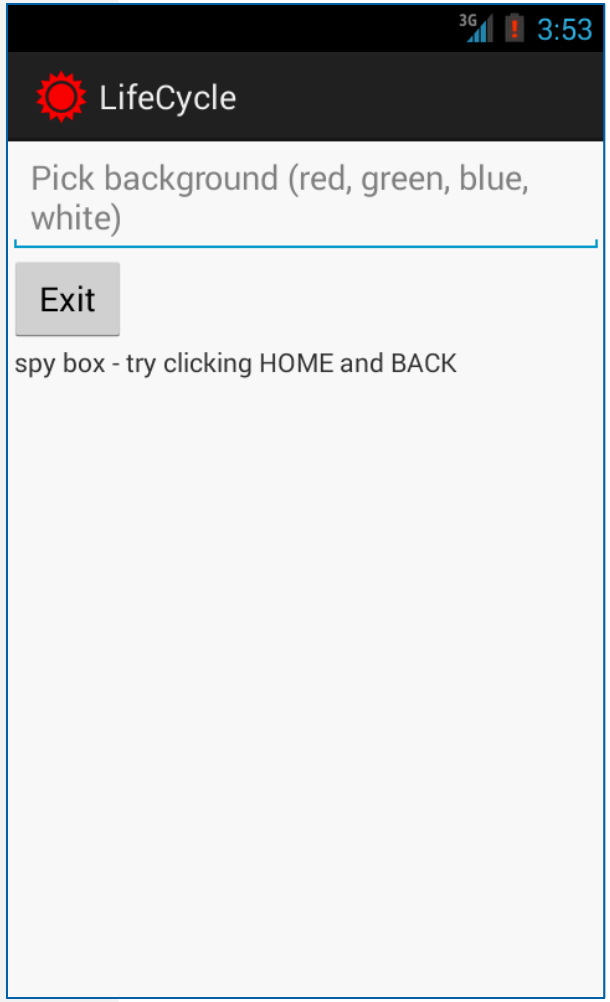
EXAMPLE: LifeCycle app

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 provide to terminate the app.
6. You are asked to observe the sequence of messages when the application:
    1. Loads for the first time
    2. Is paused after clicking HOME button
    3. Is re-executed from launch-pad
    4. Is terminated by pressing BACK and its own EXIT button
    5. Re-executed after a background color is set

Layout: atcivity_main.xml

```
<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>
```



```
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;
   //Matching GUI controls to 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);
```

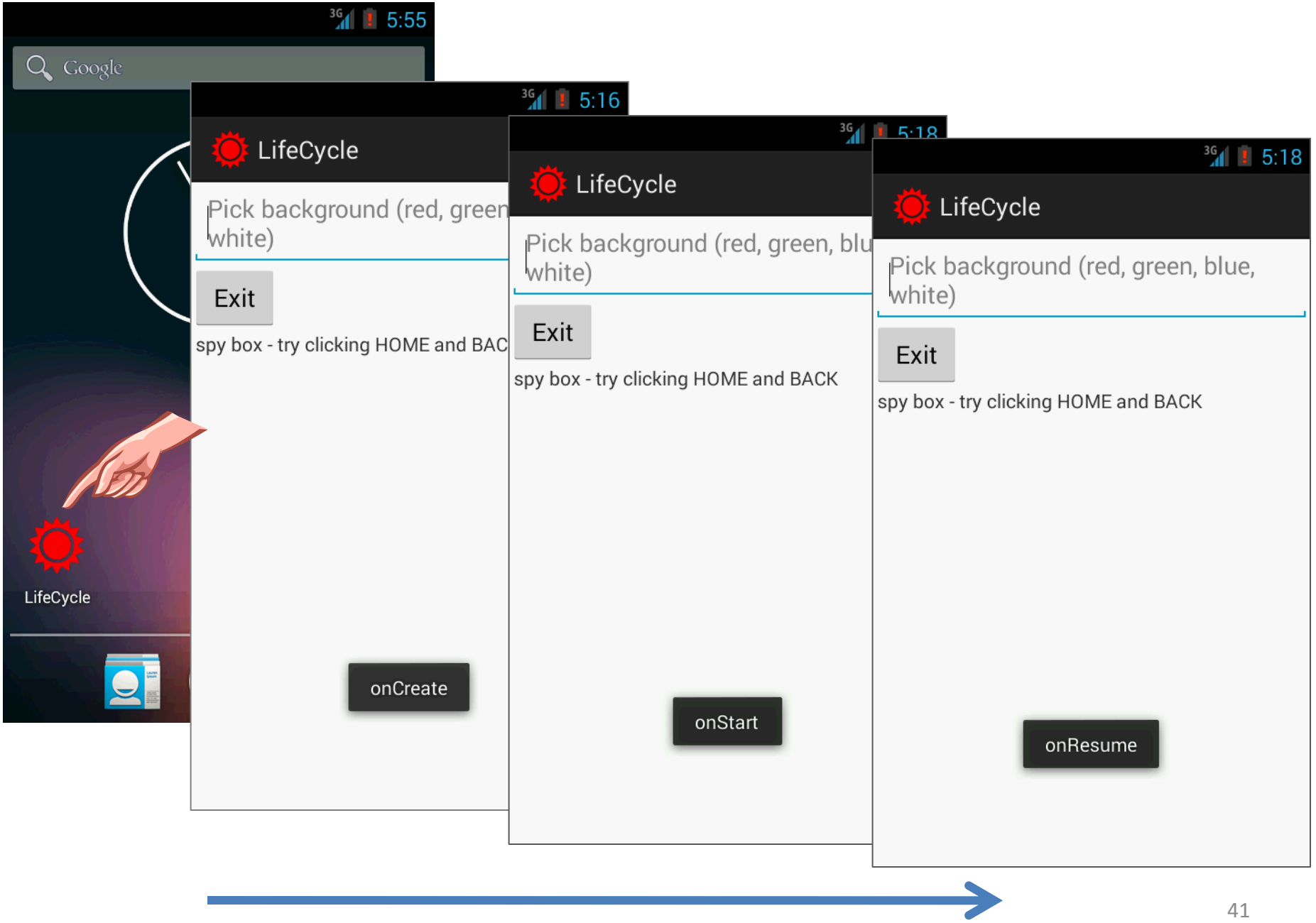
```
//set GUI listeners, watchers,...
btnExit.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View v) {
      finish();
   }
});

//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);
   }
});
```

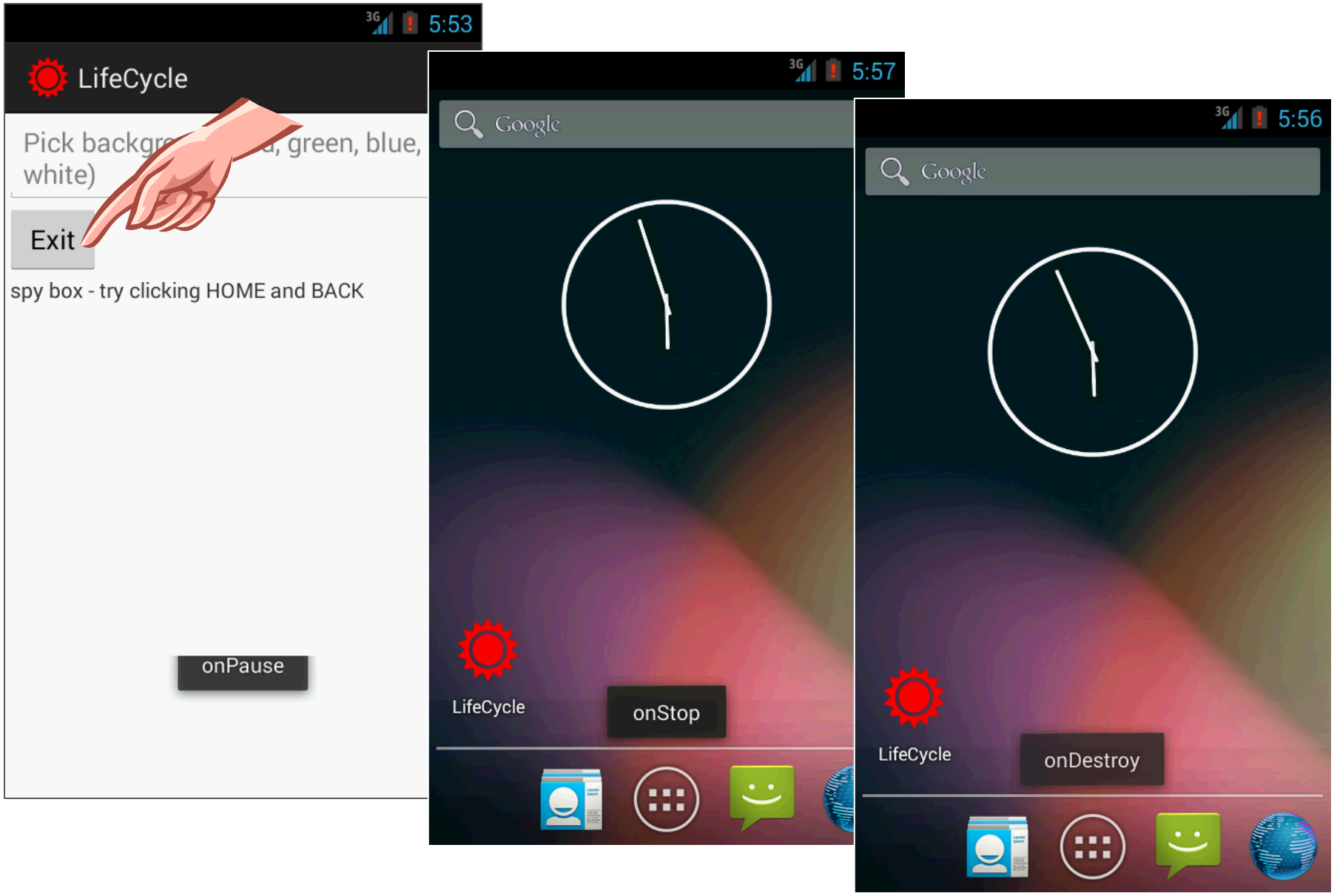
```
    //show the current state's name
    context = getApplicationContext();
    Toast.makeText(context, "onCreate", duration).show();
} //onCreate

@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();
}
```

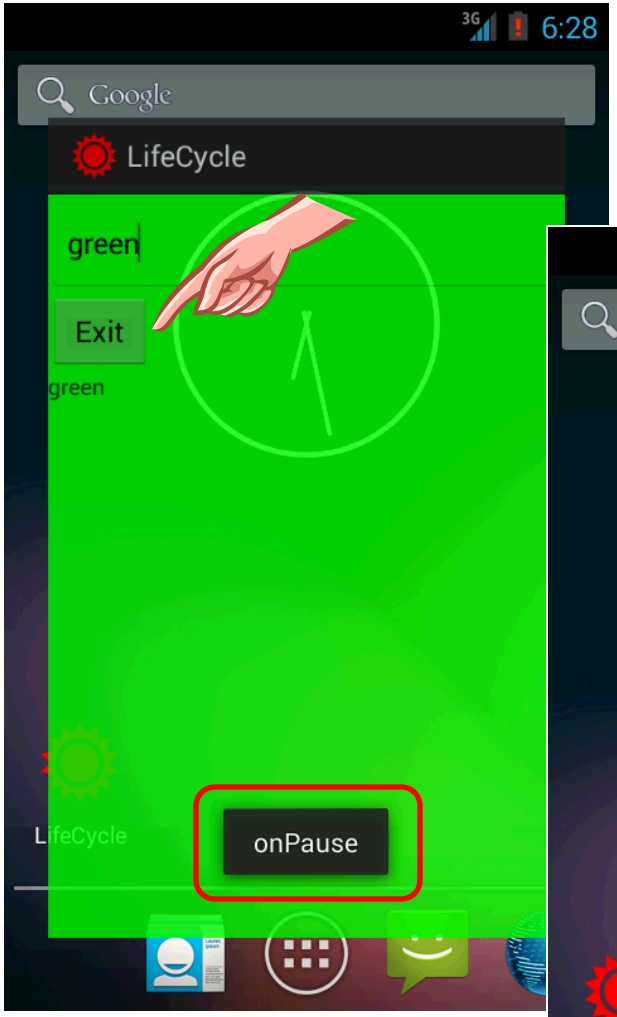
```
@Override
protected void onResume() {
    super.onResume();
    Toast.makeText(context, "onResume", duration).show();
}

@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.BLUE
} //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";
   String value = txtSpyBox.getText().toString();
   myPrefEditor.putString(key, value);
   myPrefEditor.commit();
}//saveStateData
```

```
   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
```

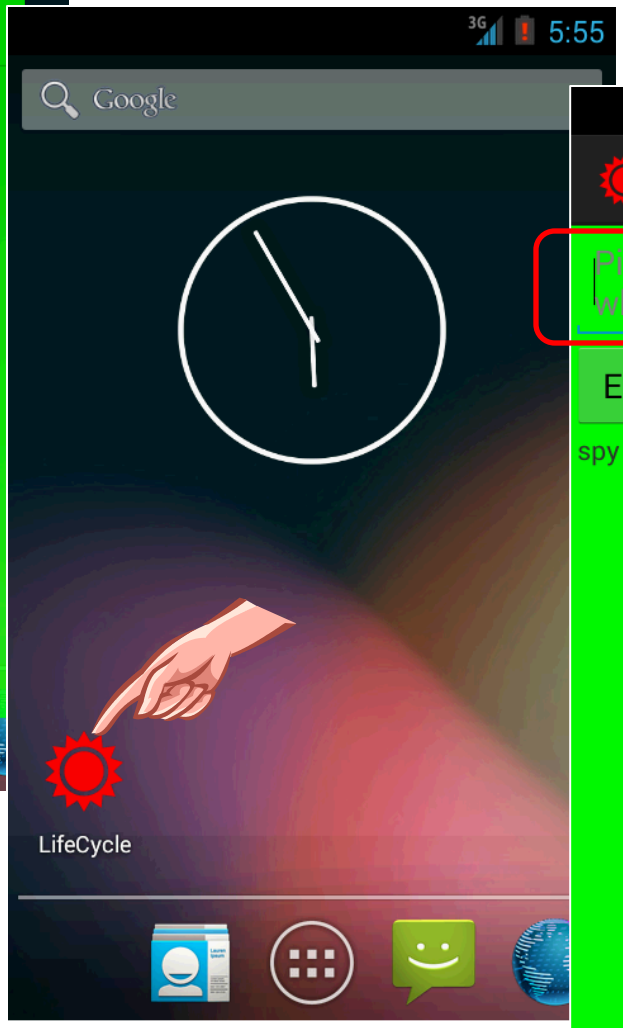

5:55
Google
LifeCycle
5:16
LifeCycle
Pick background (red, green
white)
Exit
spy box - try clicking HOME and BAC
onCreate
5:18
LifeCycle
Pick background (red, green, blu
white)
Exit
spy box - try clicking HOME and BACK
onStart
5:18
LifeCycle
Pick background (red, green, blue,
white)
Exit
spy box - try clicking HOME and BACK
onResume


5:53
LifeCycle
Pick backgr d, green, blue, white)
Exit
spy box - try clicking HOME and BACK
onPause
5:57
Google
LifeCycle
onStop
5:56
Google
LifeCycle
onDestroy

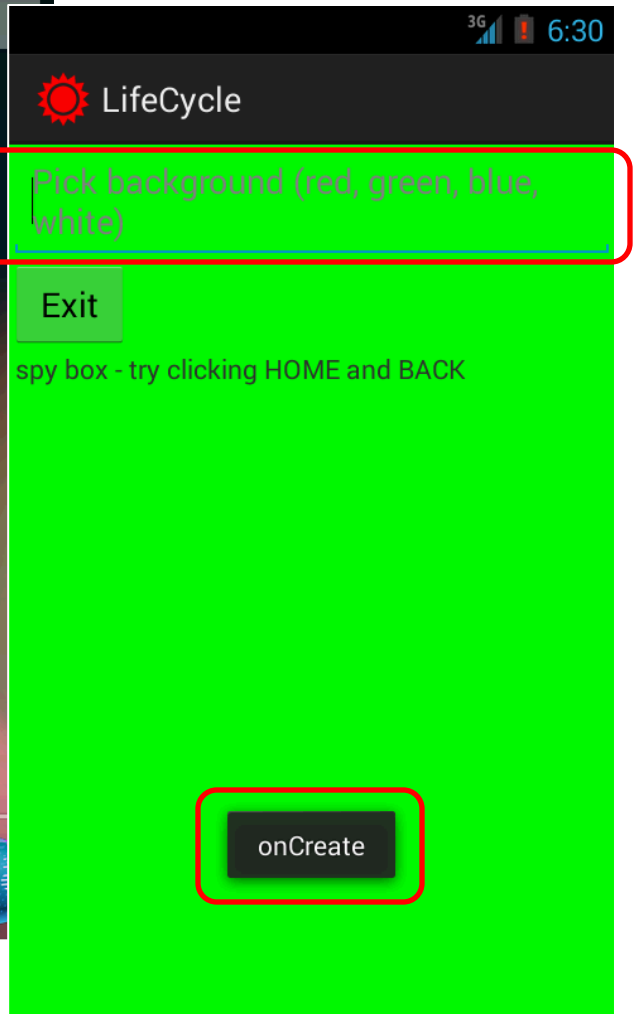


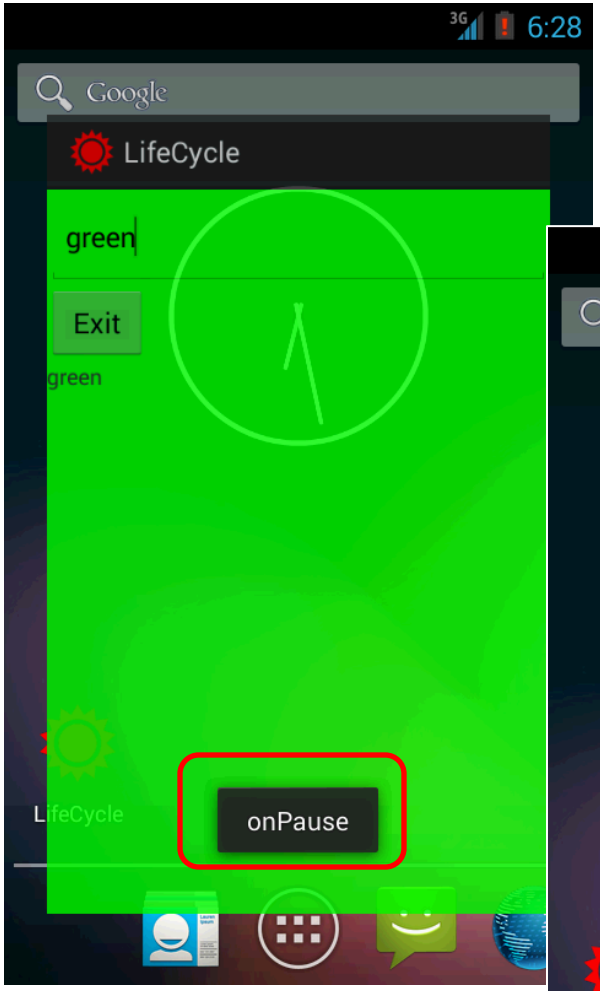
User selects a **green** background and clicks **Exit**. When the app is paused the user's selection is saved and the app finally terminates.

The app is re-executed



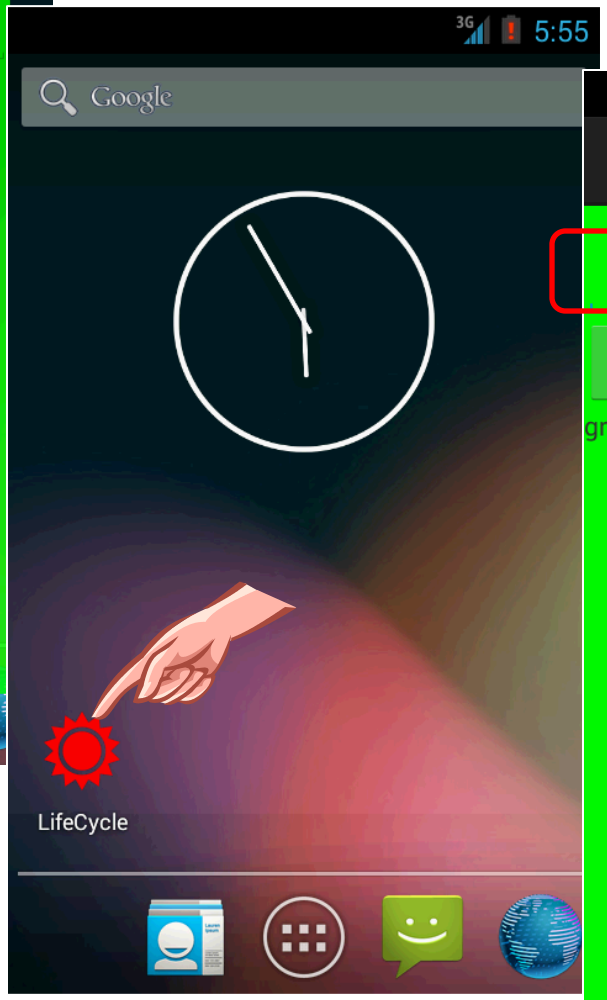
Saved state information defining background color is reused by the new app's instance. Life cycle begins on the onCreate state



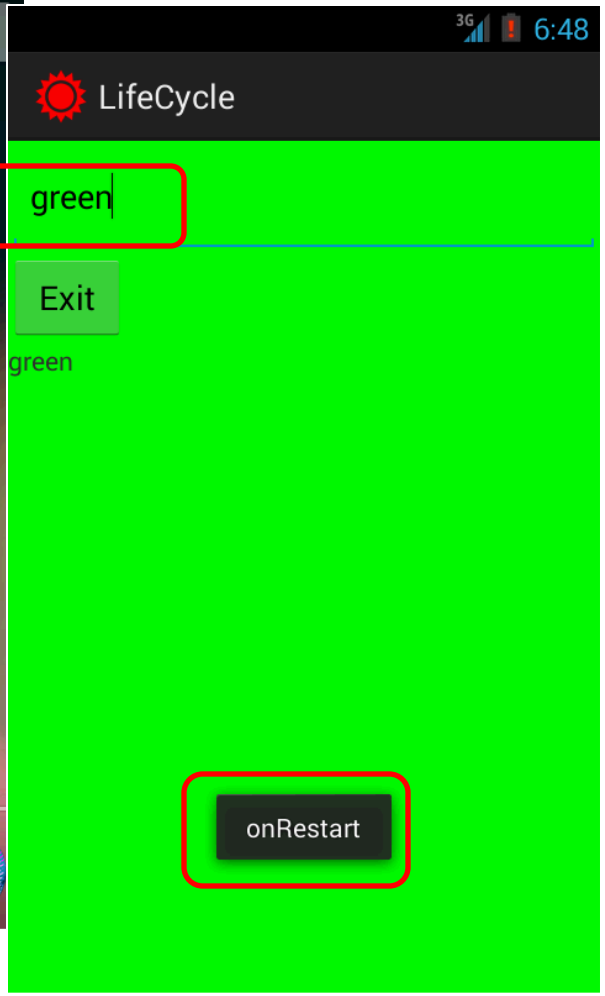


User selects a **green** background and clicks the **HOME** key. When the app is paused the user's selection is saved, the app is still active but it is not visible.

The app is re-executed



The app is re-started and becomes visible again, showing all the state values previously set by the user (see the text boxes)



Life Cycle – **QUESTIONS ?**

Appendix

Using Bundles to Save 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");
}
```