Lesson 3

Application's Life Cycle

Victor Matos
Cleveland State University

Anatomy of Android Applications

Core Components

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

An Android application consists of one or more *core component* objects. Components work in a *cooperative mode*, each contributing somehow to the completion of the tasks undertaken by the app.

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

Android's Core Components

1. Activity Class

- An **Activity** object is similar to a WindowsForm. It usually presents a single graphical visual interface (GUI) which in addition to the 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 user name 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.
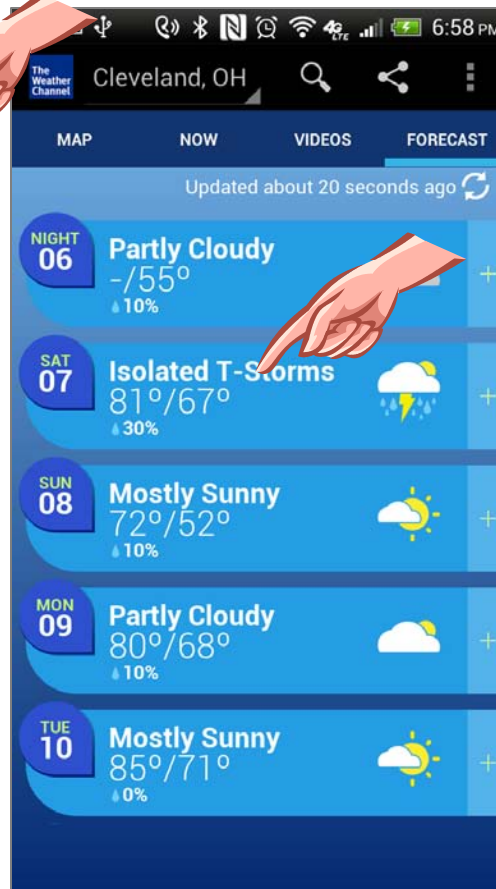
Android's Core Components

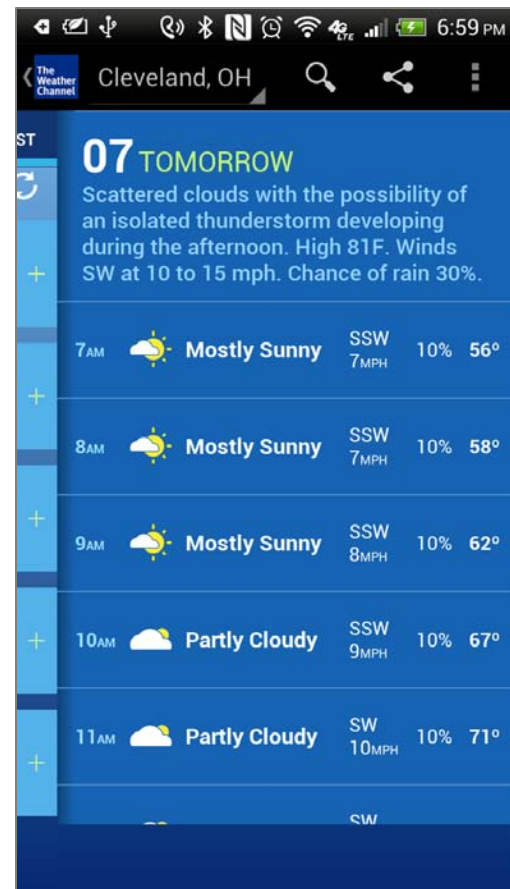## Example of an app containing multiple Activities

Weather Channel app
GUI-1- Activity 1

Weather Channel app
GUI-2- Activity 2

Weather Channel app
GUI-3- Activity 3

3 - 4

Android's Core Components

2. Service Class

- 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.
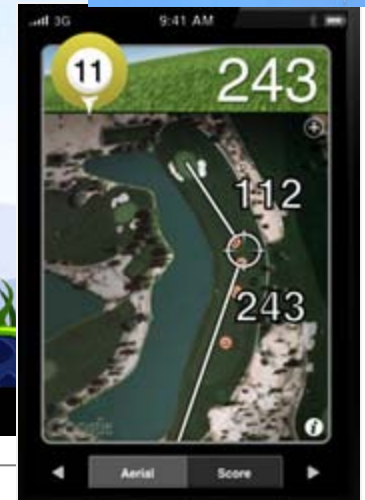
Android's Core Components
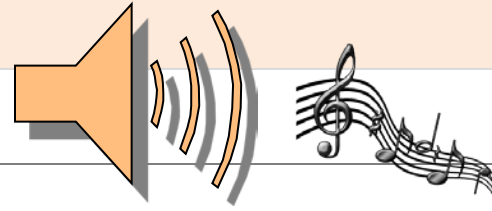
2. Service

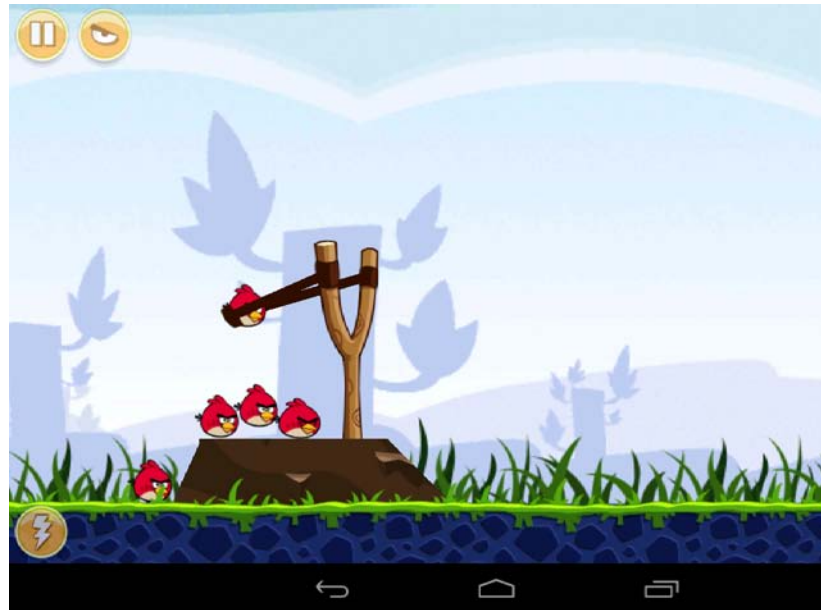| Background	Foreground
PANDORA GPS	

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" (perhaps some of his playing partners are very slow).

Android's Core Components

3. Broadcast Receiver Class

- A **BroadcastReceiver** 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 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

Background Services	Broadcast Receiver	Foreground Activity
		Method()
		Work to be done after receiving an ORANGE message
Send an ORANGE signal	**Waiting**. My *filter* only accepts ORANGE signals. Ignoring all others.	

Android's Core Components

4. Content Provider Class

- 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 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.

4. Content Provider Class

Content Provider

Local Data Sets

Cloud Data Sets

query(...)

insert(...)

delete(...)

update(...)
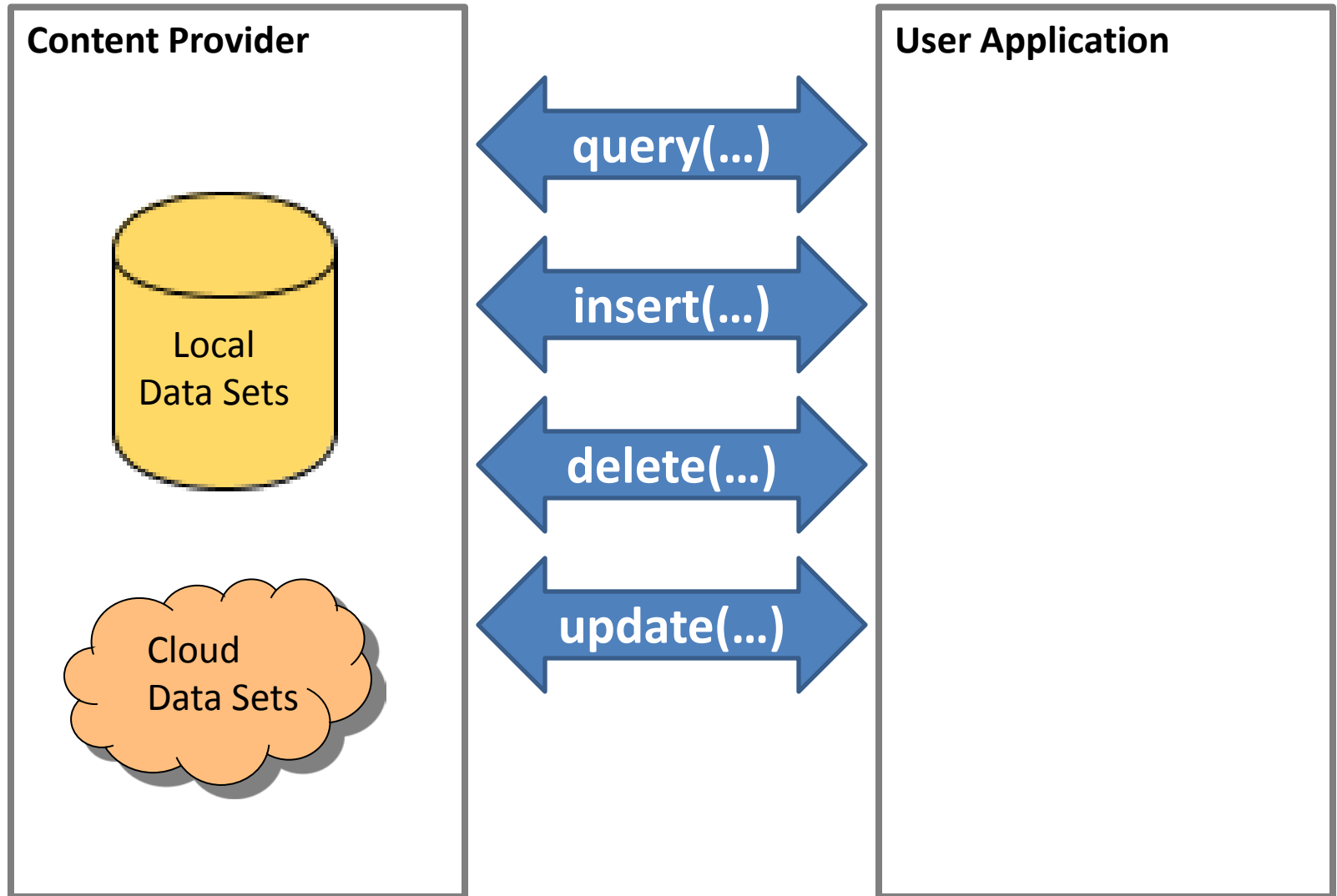
User Application

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

Component's Life Cycle

Life and Death in Android

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:

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's Life Cycle

Life and Death in Android

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*.

Start

**Life as an Android Application:
Active / Inactive
Visible / Invisible**

End

Component's Life Cycle
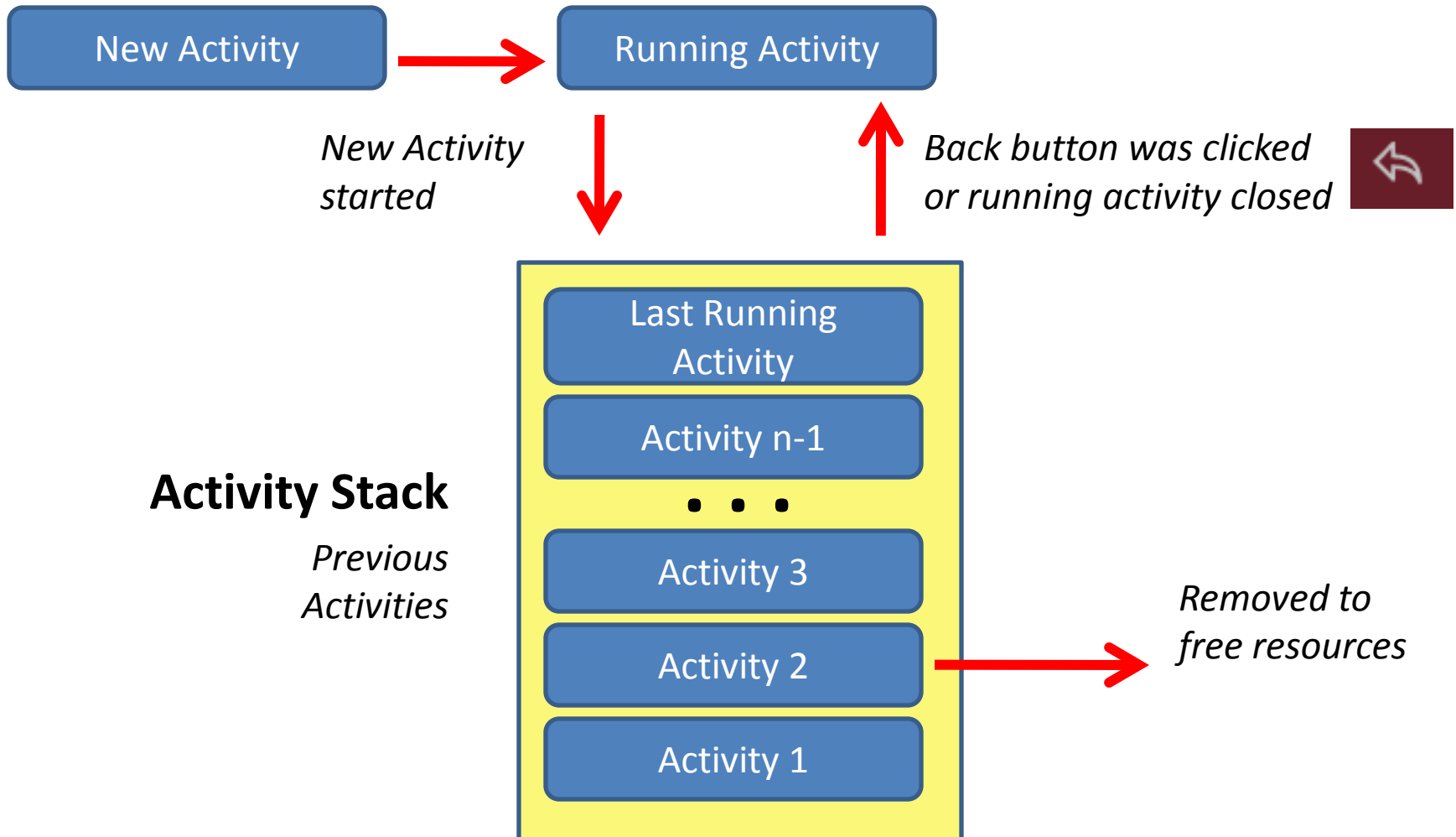
The Activity Stack

- **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

- 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 previous activity on the stack moves up to become active.

- Android 4.0 introduced the 'Recent app' [▭] button to arbitrarily pick as 'next' any entry currently in the stack (more on this issue later)

Virtual buttons (Android 4.x and 5.x): Back, Home, Recent apps

Component's Life Cycle

New Activity → Running Activity

New Activity started

Back button was clicked or running activity closed

Activity Stack

Previous Activities

Last Running Activity

Activity n-1

• • •

Activity 3

Activity 2

Activity 1

Removed to free resources

3 - 14

Component's Life Cycle

Life Cycle Callbacks

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( ) **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

```java
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.
    }
}
```

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.
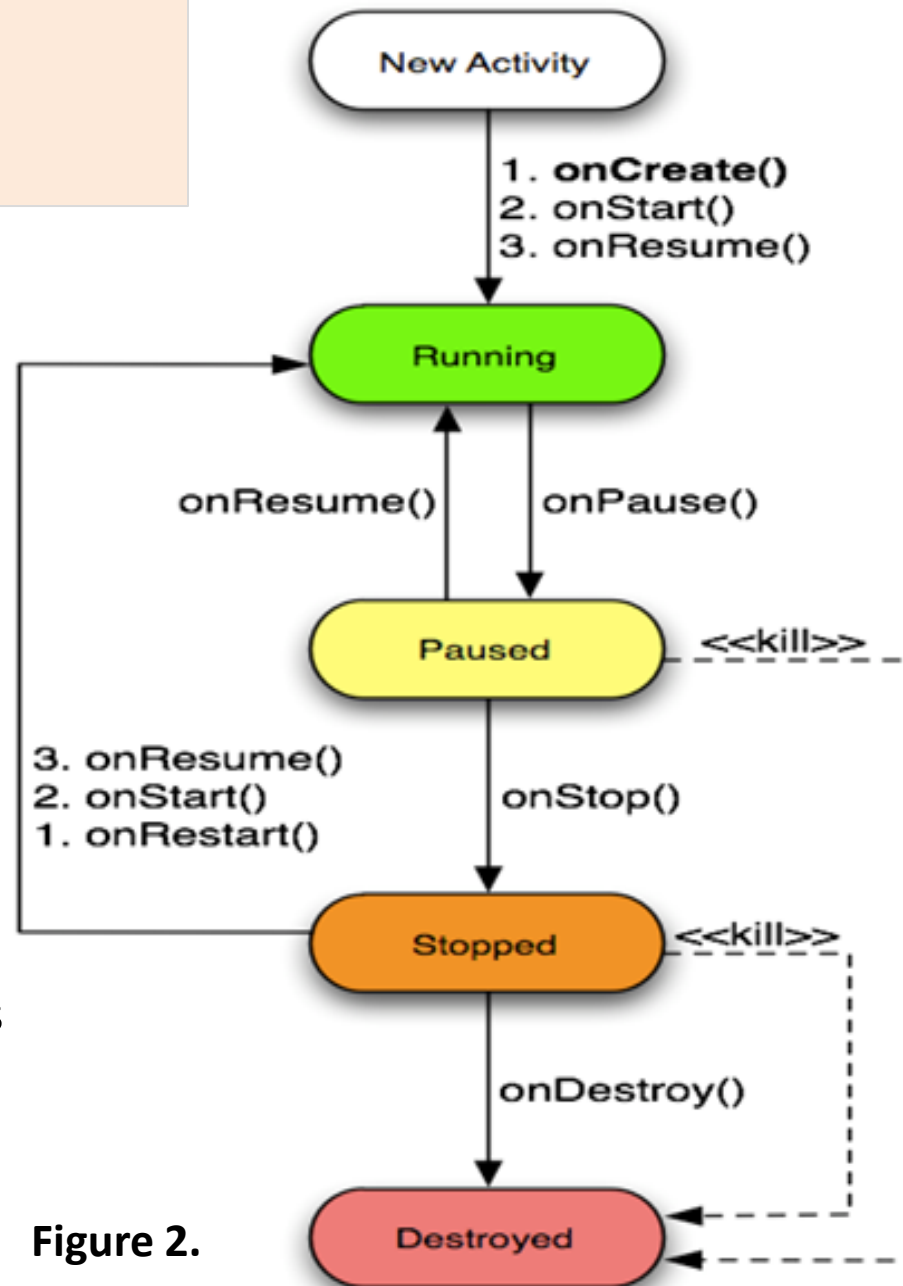
New Activity

1. **onCreate()**
2. onStart()
3. onResume()

Running

onResume()   onPause()

Paused   <<kill>>

3. onResume()
2. onStart()
1. onRestart()

onStop()

Stopped   <<kill>>

onDestroy()

Destroyed

Figure 2.

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

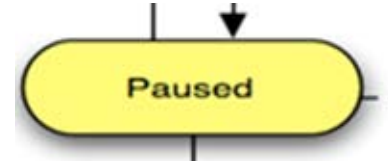# Component's Life Cycle

Activity State: RUNNING

1. Your activity is *active* **or** *running* when it is in the *foreground* of the 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.

Component's Life Cycle

Activity State: PAUSED

2. Your Activity 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.

Component's Life Cycle

3. 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).

Activity Life Cycle

Reference:

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

Component's Life Cycle
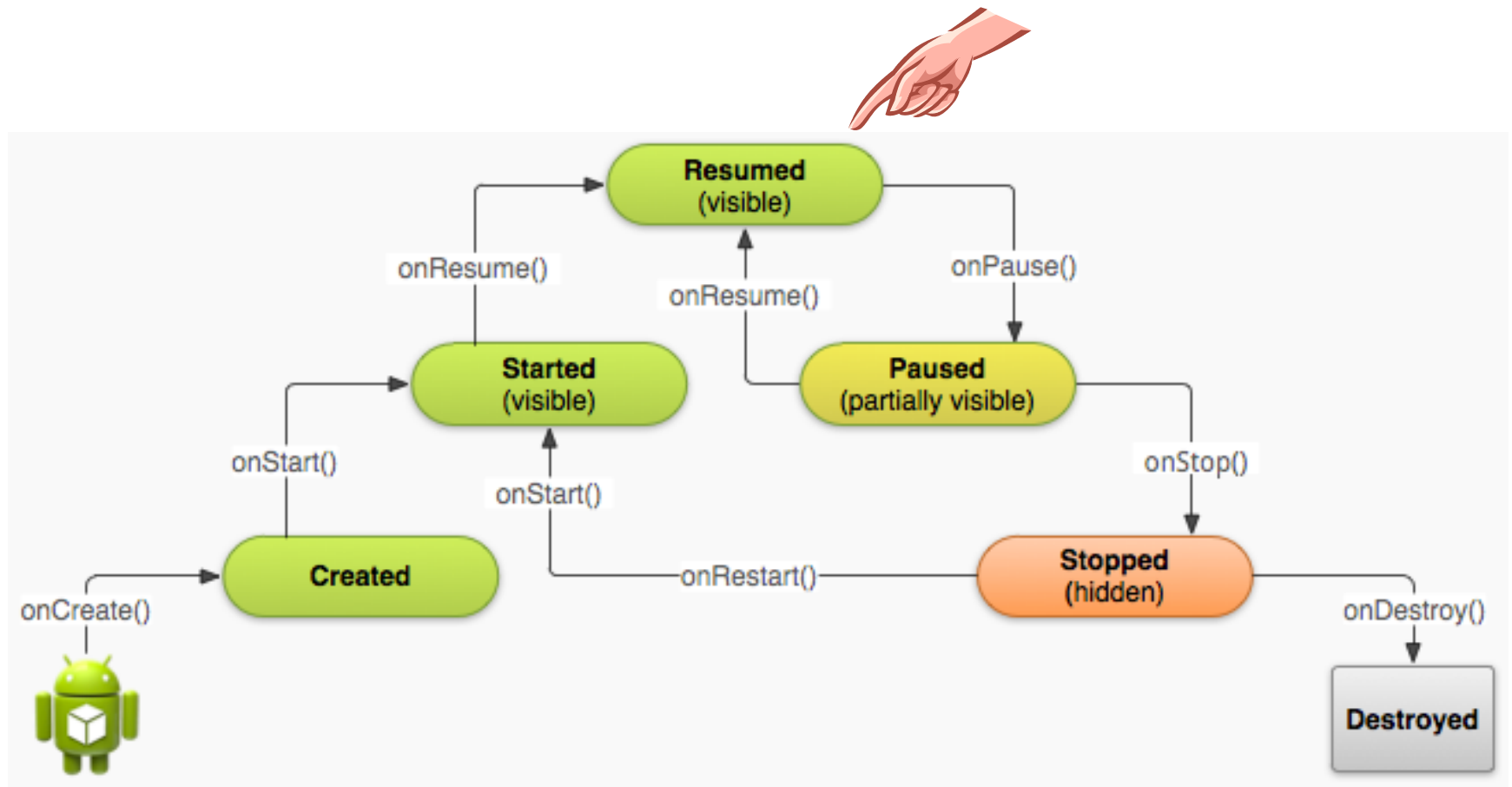
Your turn! Lab Experience 1.

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 - 22

Component's Life Cycle
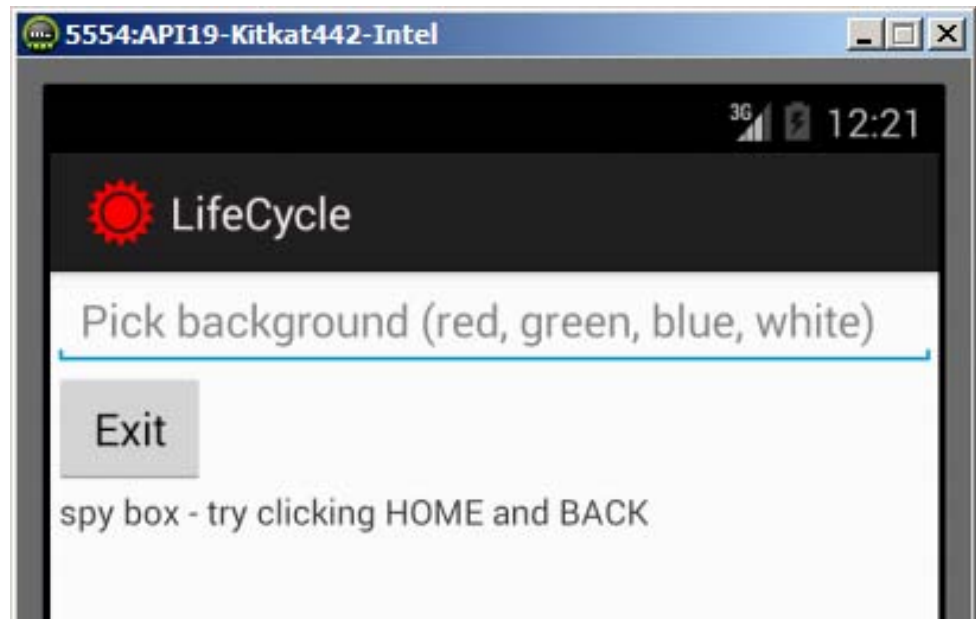
Your turn! Lab Experience 1.

Transitioning: One State at the Time

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. To simplify your job use Eclipse's top menu:
    Source > Override/Implement Methods… (look for callback methods)
    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*.

Component's Life Cycle

Your turn! Lab Experience 1 (cont).

7. Compile and execute the application.
8. Write down the sequence of messages displayed using the Toast-commands.
9. Press the EXIT button. Observe the sequence of states displayed.
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?

Component's Life Cycle

Your turn! Lab Experience 2.

Calling & Texting Emulator-to-Emulator

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?

Teaching notes

Your turn! Lab Experience 3.

Provide data persistence.

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 frament

```
SharedPreferences myFile = getSharedPreferences("myFile1",
                                        Activity.MODE_PRIVATE);
if ( (myFile != null) && (myFile.contains("mydata")) ) {
        String temp = myFile.getString("mydata", "***");
        txtMsg.setText(temp);
}
```
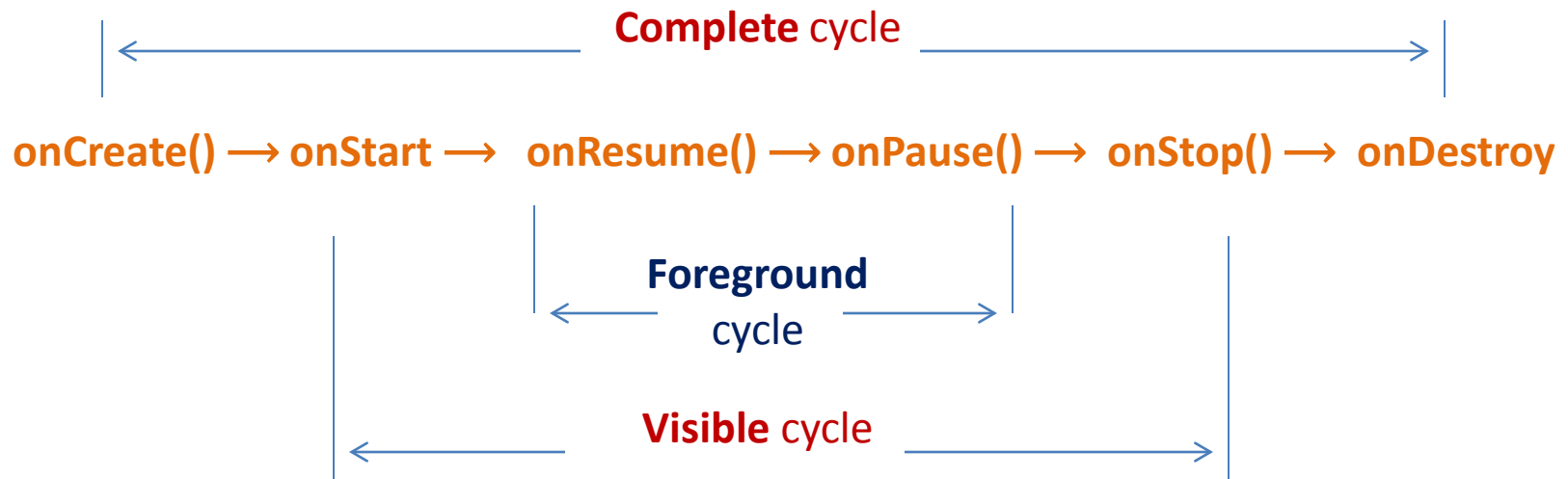
18. What happens now with the data previously entered in the text box?

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.

(**Highly Recommended**)
onPause() should be implemented whenever the application has some important data to be committed so it could be reused.

Application's Life Cycle

Associating Lifecycle Events with Application's Code

onCreate()

- 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*() ….
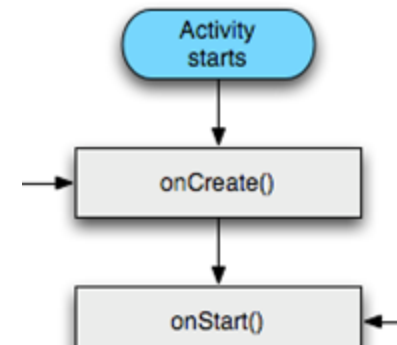
Application's Life Cycle

Associating Lifecycle Events with Application's Code

onPause()

1. 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.

2. The next activity waits until completion of this state.

3. Followed either by *onResume*() if the activity returns back to the foreground, or by *onStop*() if it becomes invisible to the user.

4. A paused activity could be *killed* by the system.
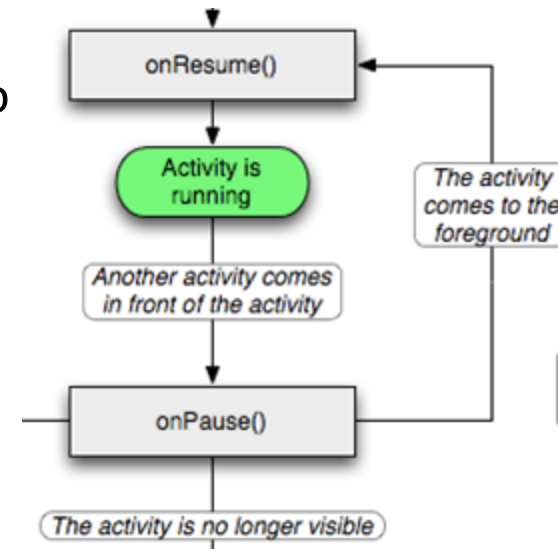
Application's Life Cycle

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 Android SharedPreferences Class

- **SharedPreferences** 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 (int, float, string...).
- 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);
```

Persistence is an important concept in Android, and it is discussed in more detail latter.

Application's Life Cycle

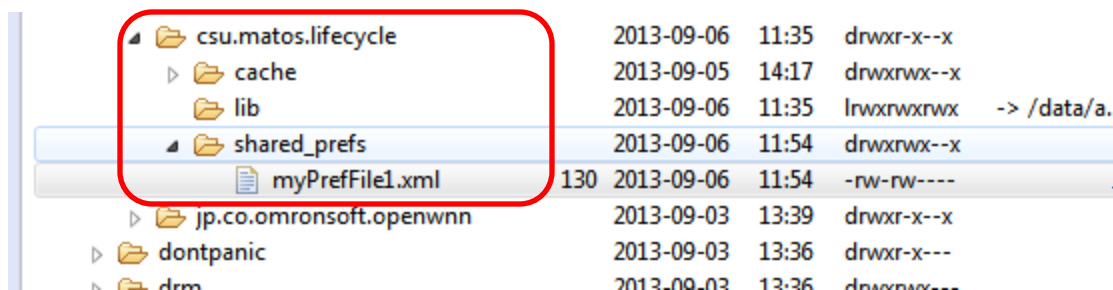## Data Persistence using Android SharedPreferences Class

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


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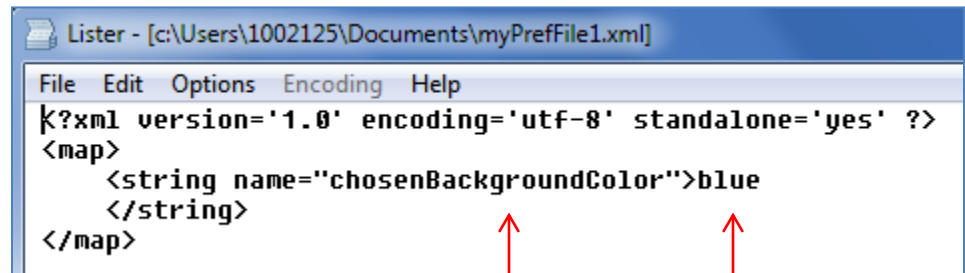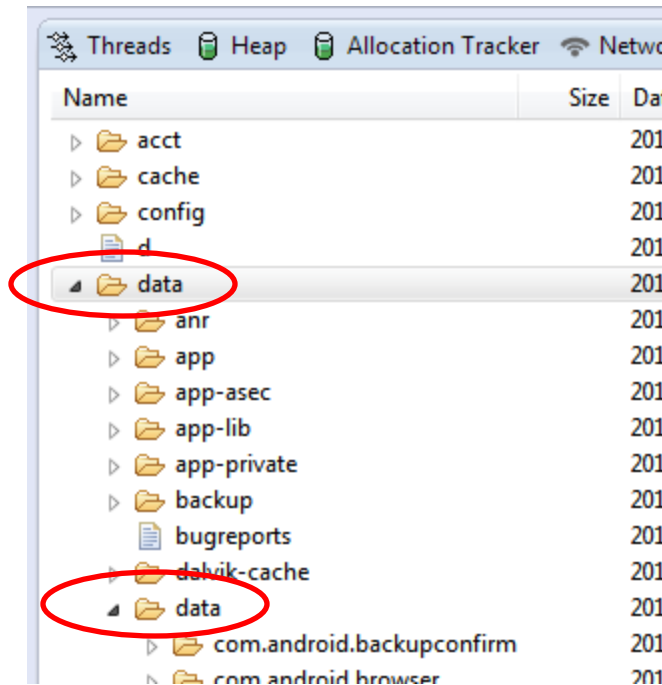
Key   Value

Application's Life Cycle

A complete Example: The 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 displayed 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

Application's Life Cycle

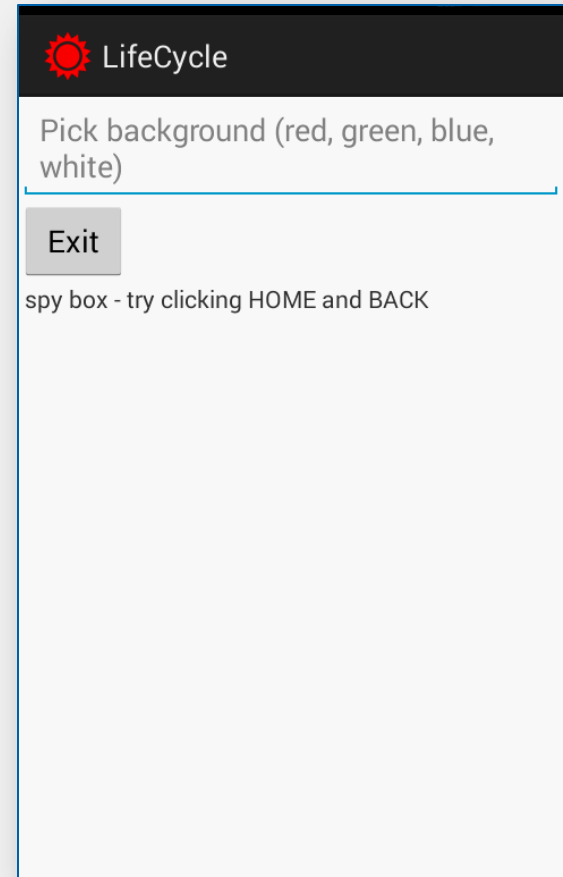## Example: The LifeCycle App – Code: MainActivity.java   pp.1

```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>
```

Example: The LifeCycle App – Code: MainActivity.java   pp.2

```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);
```

Example: The LifeCycle App – Code: MainActivity.java   pp.3

```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);
            }
        });
```

Example: The LifeCycle App – Code: MainActivity.java   pp.4

```java
    //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();
}
```

Application's Life Cycle

Example: The LifeCycle App – Code: MainActivity.java   pp.5

```java
@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();
}
```

Application's Life Cycle

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

Example: The LifeCycle App – Code: MainActivity.java   pp.7

```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
```
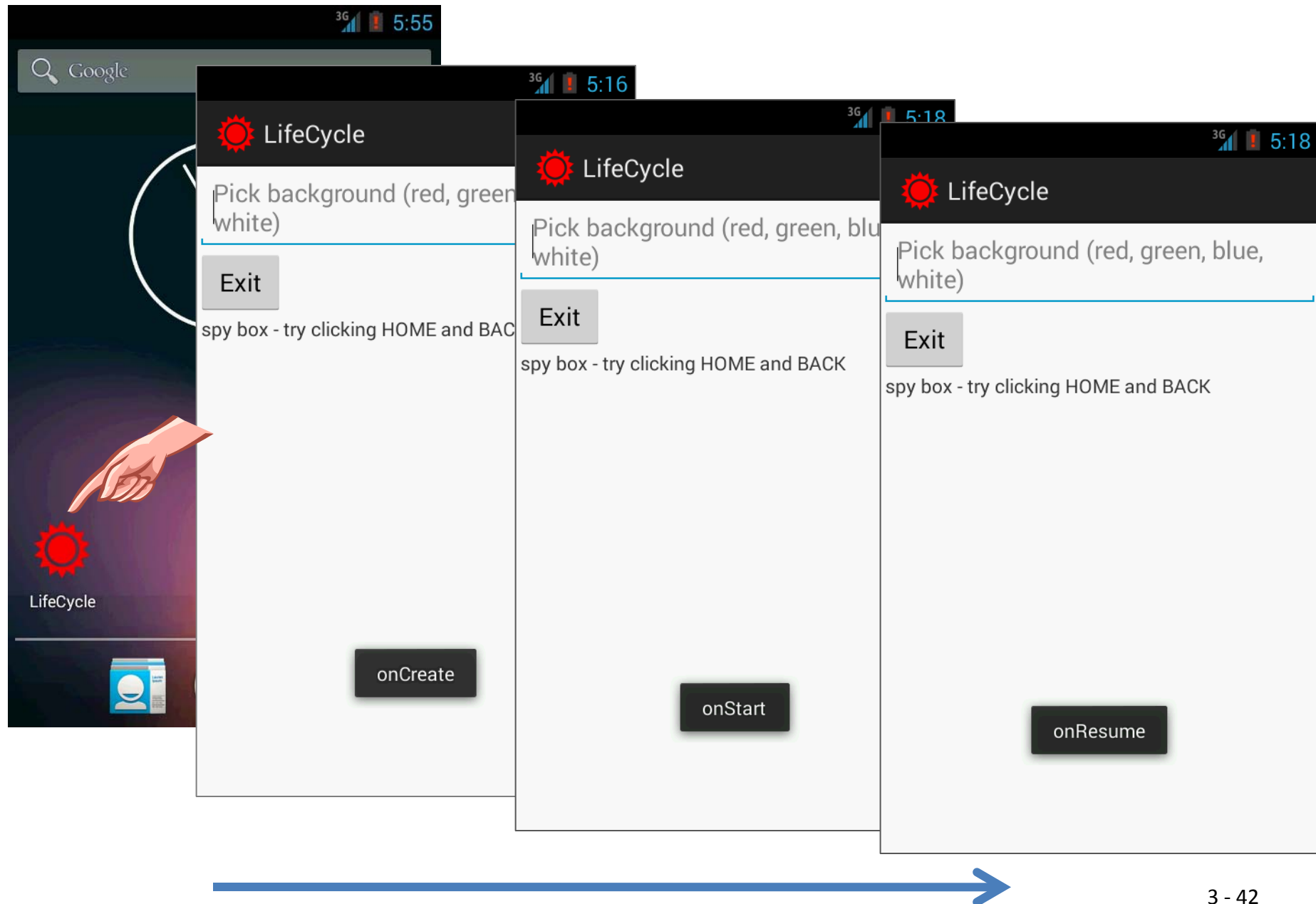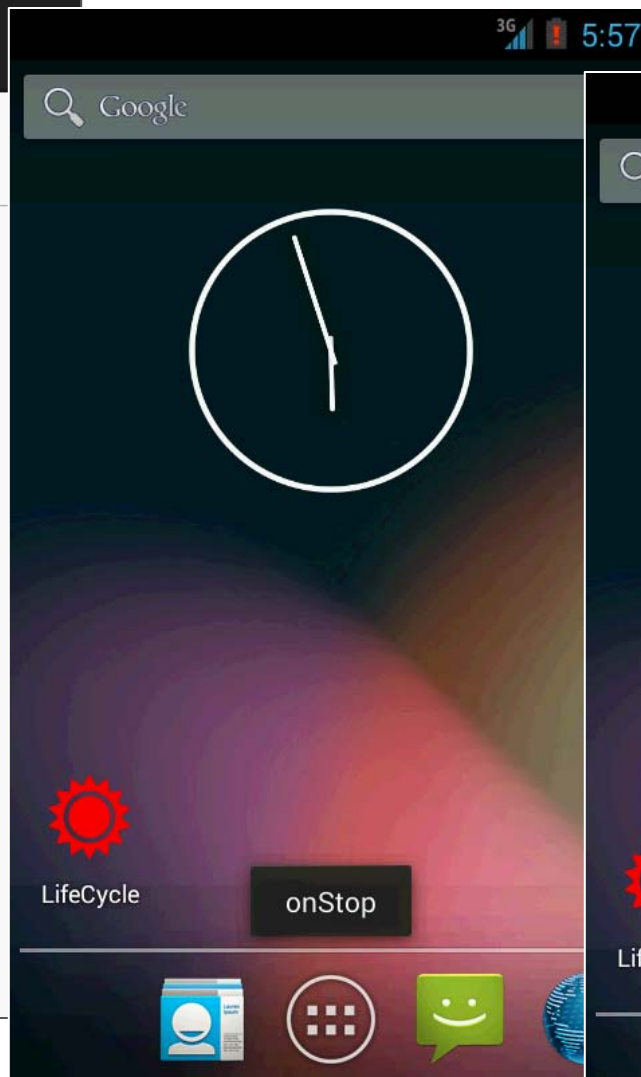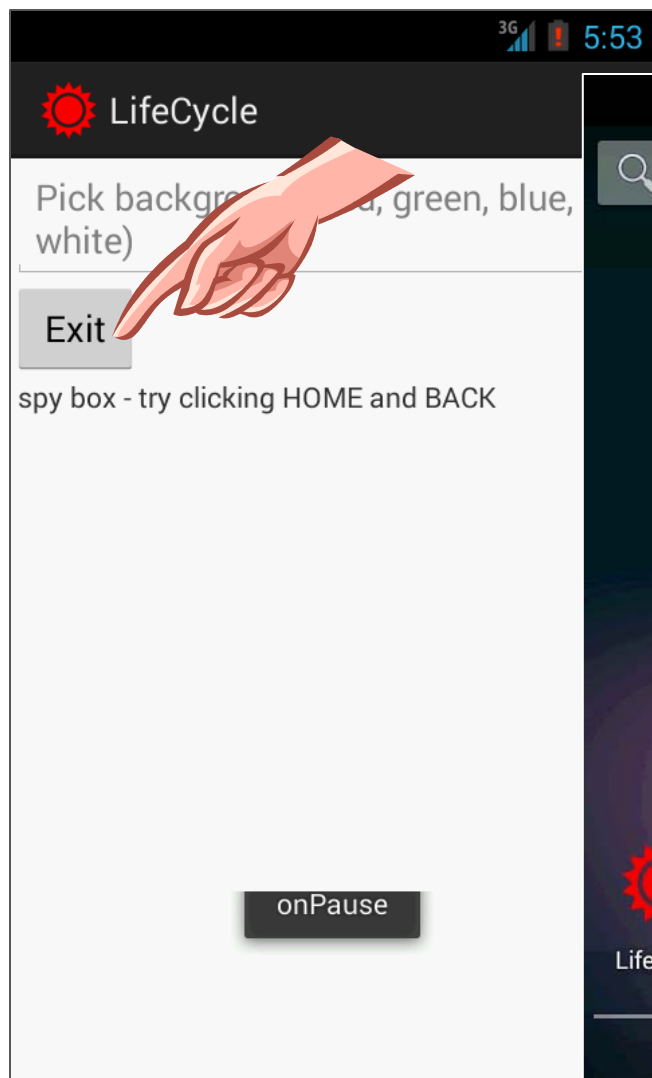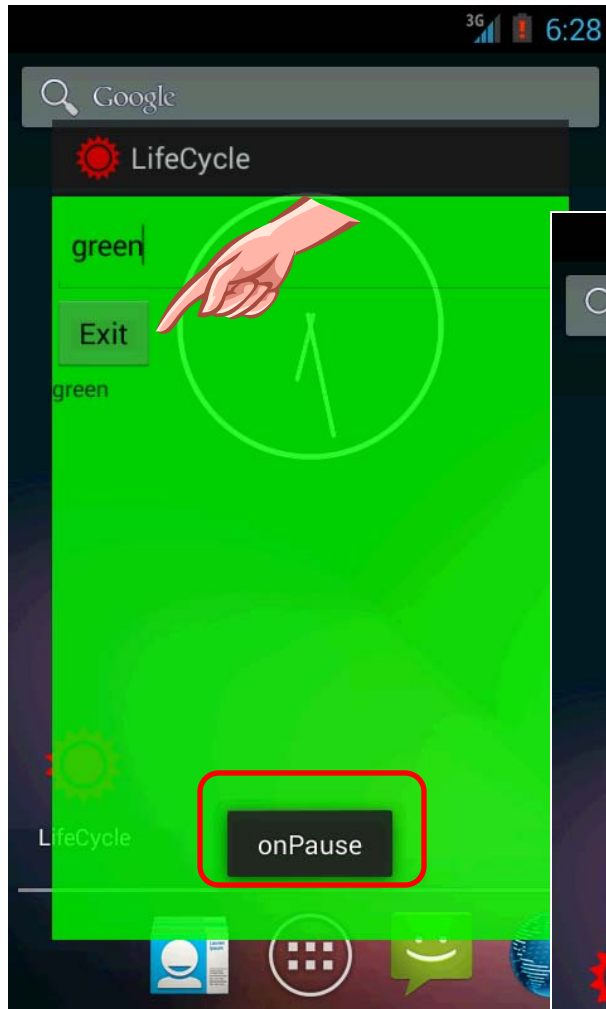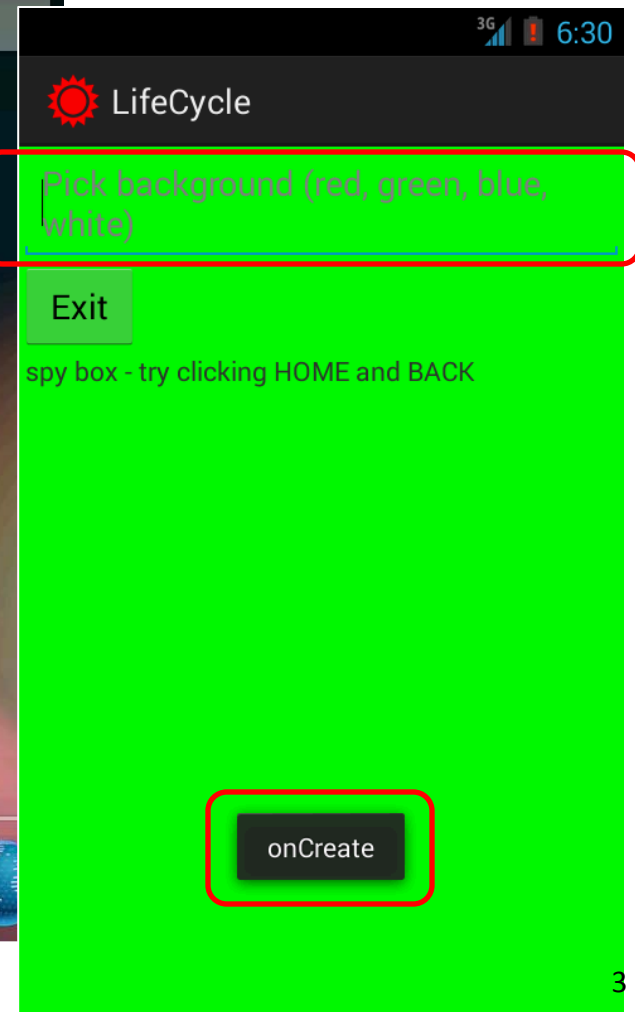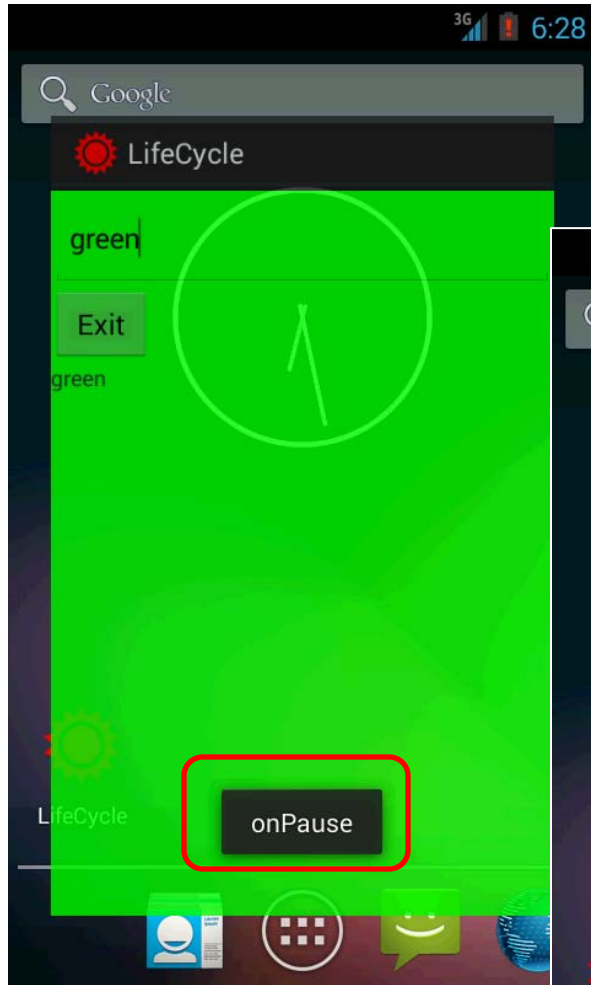
The app is re-executed
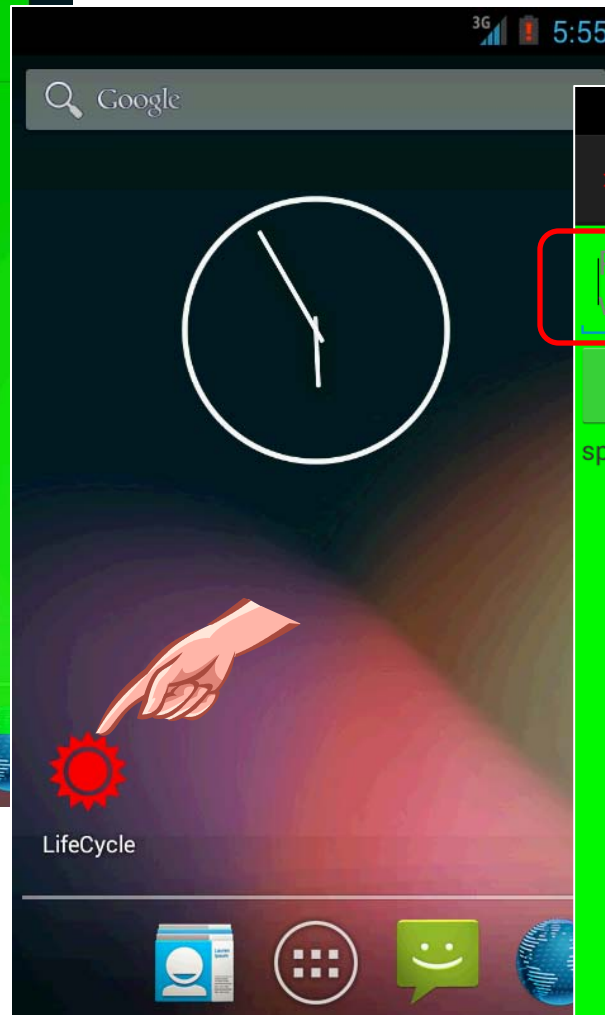
Saved state information defining background color is reused by the new app's instance. Life cycle begins on the onCreate state
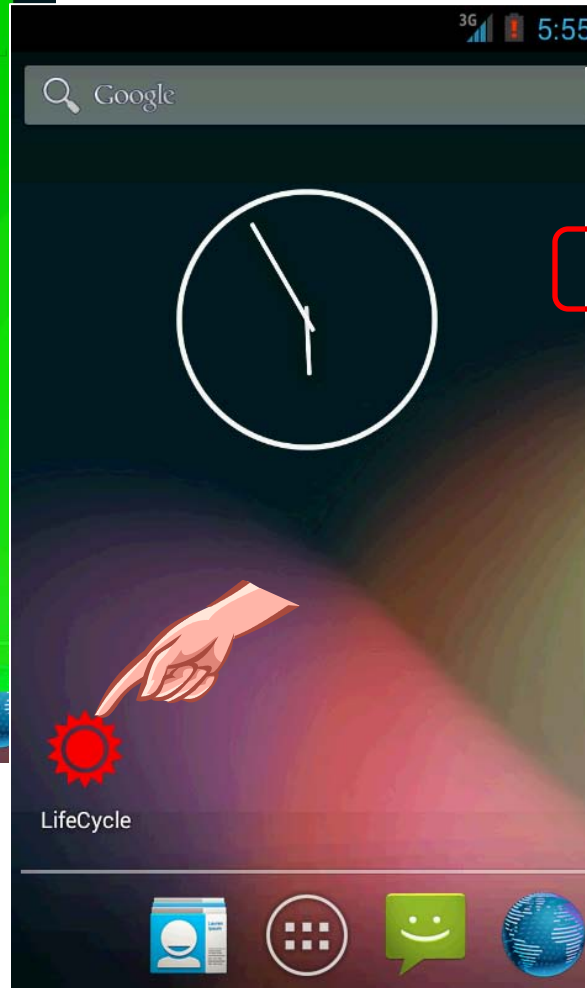
User selects a **green** background and clicks **Exit**. When the app is paused the user's selection is saved and the app finally terminates.

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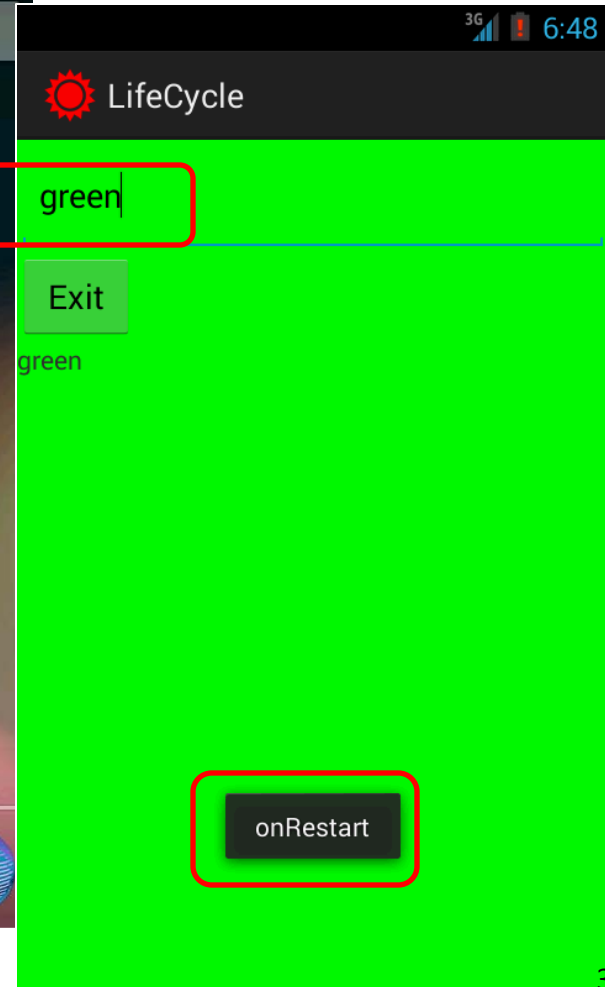
User selects a **green** background and clicks the **HOME** key. When the app is paused the user's selection is saved, the app is still active but it is not visible.

Application's Life Cycle

Questions?

Appendix A: 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) {
    ...
    myBundle.putString("STR_KEY", "blah blah blah");
    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.   3 - 46

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

```java
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();
}
```


North: 0

top

West: 1

top

East: 3

top

South: 2

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.

```java
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
    }
}
```