Activities and Intents

C.-Z. Yang
http://syslab.cse.yzu.edu.tw/~czyang

3.1 Activity Lifecycle

- All applications are composed of at least one Activity class.
- In most cases, applications will require the use of several activities.
  - Activities in an application are often loosely connected to each other.
  - Information can be passed from one activity to another, but they remain distinct and separate in every other way.
  - Every application has one activity class that serves as the main activity.

A Chess Game example

- A Chess Game App Showing Two Activities: Game Play and Chatting

Activities

- Activities must be declared in AndroidManifest.xml
- Activities are defined using the <activity> tag.
  - An <activity> must be added as a child to the <application> element

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.cornez.autopurchase" >

   <application
      android:allowBackup="true"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.Light.NoTitleBar.Fullscreen" >

      <!-- MY ACTIVITY CLASS: USED TO INPUT THE CAR PURCHASE INFORMATION -->
      <activity
         android:name=".PurchaseActivity"
         android:label="@string/app_name" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <activity
         android:name=".LoanSummaryActivity"
         android:label="@string/app_name"
         android:parentActivityName=".PurchaseActivity" >
         <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value=".PurchaseActivity" />
      </activity>
   </application>

</manifest>
```

Activity class

- The activities in an application are implemented as a subclass of Activity.
- The Activity class is an important part of every application's overall lifecycle.
  - When an application is first loaded, its main activity, specified within the AndroidManifest, is immediately created.
  - Once the main activity is started, it is given a window in which to draw its layout.

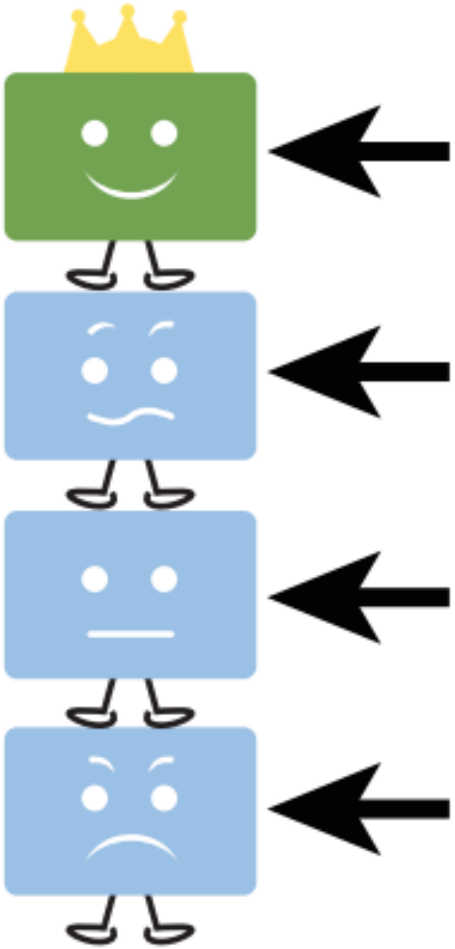
Activity Stack

- The Activities in an application are managed by an Activity stack.
- New activities are pushed onto the top of the stack and become the running activity.
- Each time a new activity starts, the previous activity is paused and its status is preserved by the Android system.

Tasks and the back stack

- A *task* is a collection of activities that users interact with when trying to do something in your app.
- These activities are arranged in a stack—the *back stack*—in the order in which each activity is opened.

The Activity stack



I am the top Activity.
User can see and interact with me!

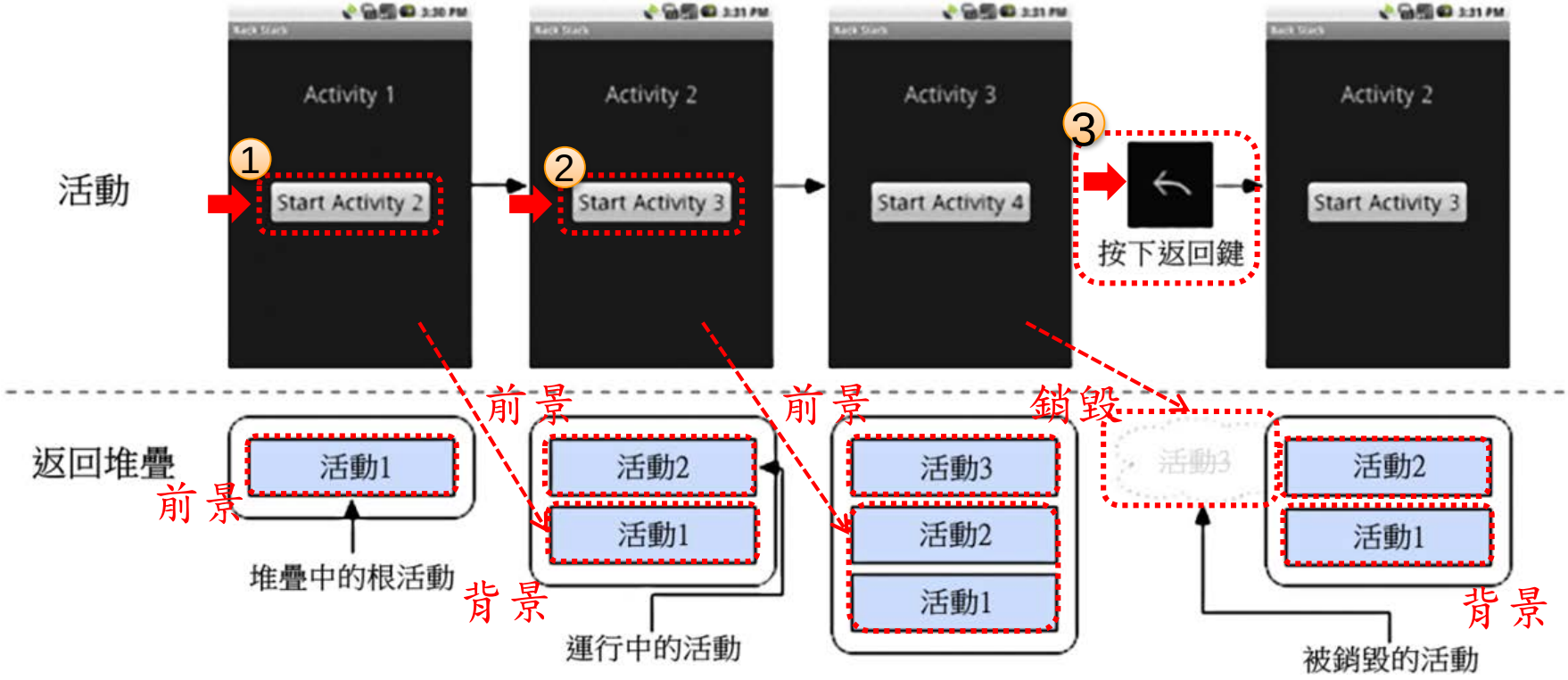
I am the second Activity in the stack.
If the user hits Back or the top Activity is destroyed, the user can see and interact with me again!

I am an Activity in the middle of the stack.
Users cannot see and interact with me until everyone above me is destroyed.

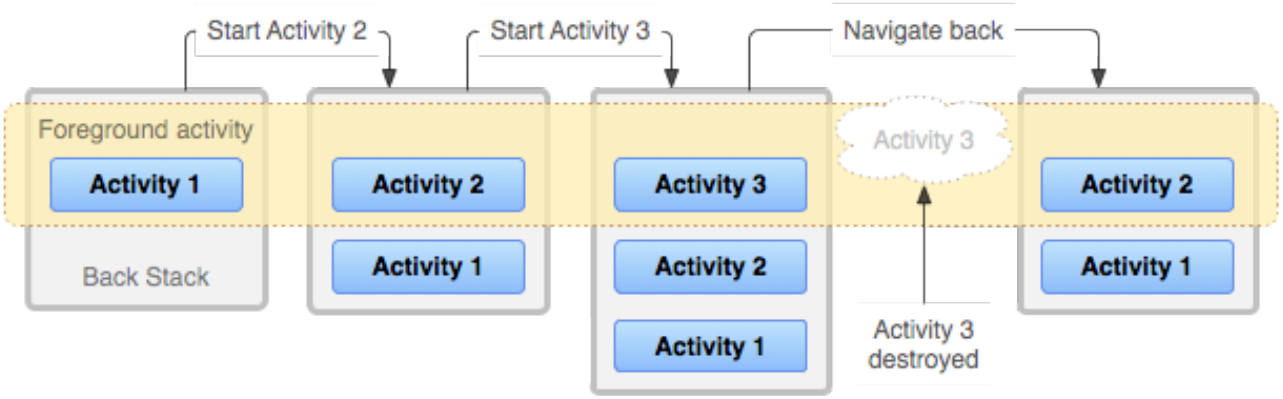
I am an Activity at the bottom of the stack.
If those Activities above me use too many resources, I will be destroyed!

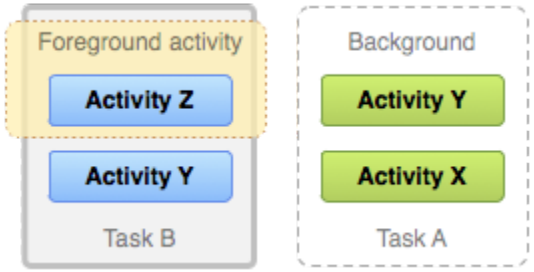
Activity stack

- The activity changing



Back stack





https://developer.android.com/guide/components/activities/tasks-and-back-stack

Back press behavior

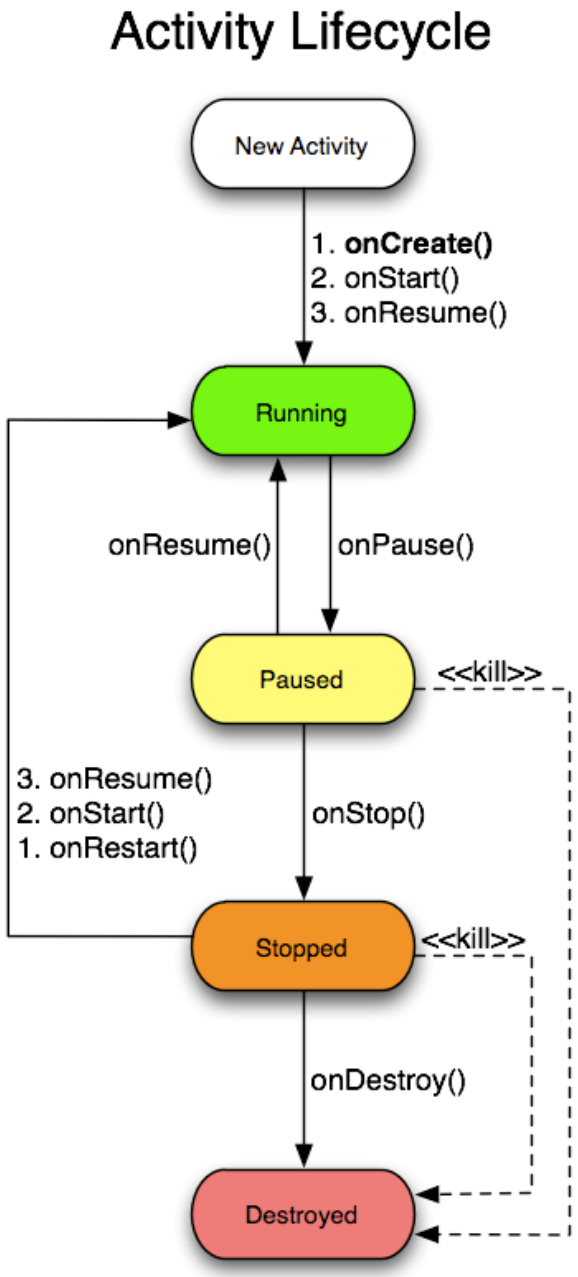
- When a user presses or gestures Back from a root launcher activity, the system handles the event differently depending on the version of Android that the device is running.
- System behavior on Android 11 and lower
  - The system finishes the activity.
- System behavior on Android 12 and higher
  - The system moves the activity and its task to the background instead of finishing the activity.

Activity Lifecycle

- Android applications are responsible for managing their state and their memory, resources, and data.
- Understanding the different states within the Activity lifecycle is the first step to designing and developing robust Android applications.

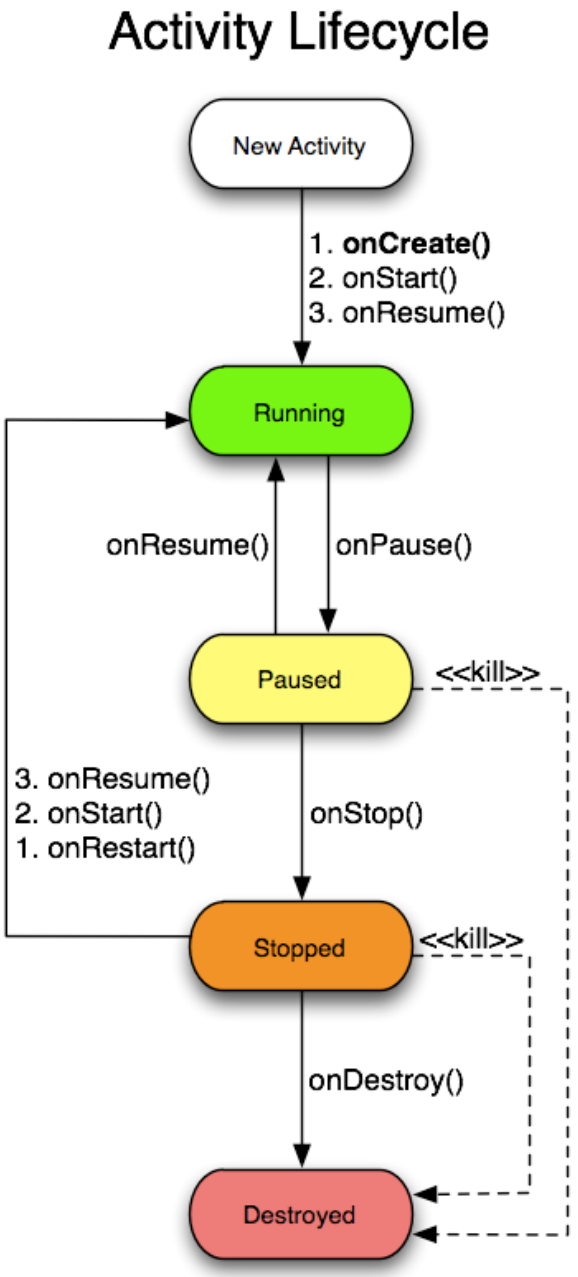
Activity states (1)

- Running
  - If an activity in the foreground of the screen (at the top of the stack), it is active or running.



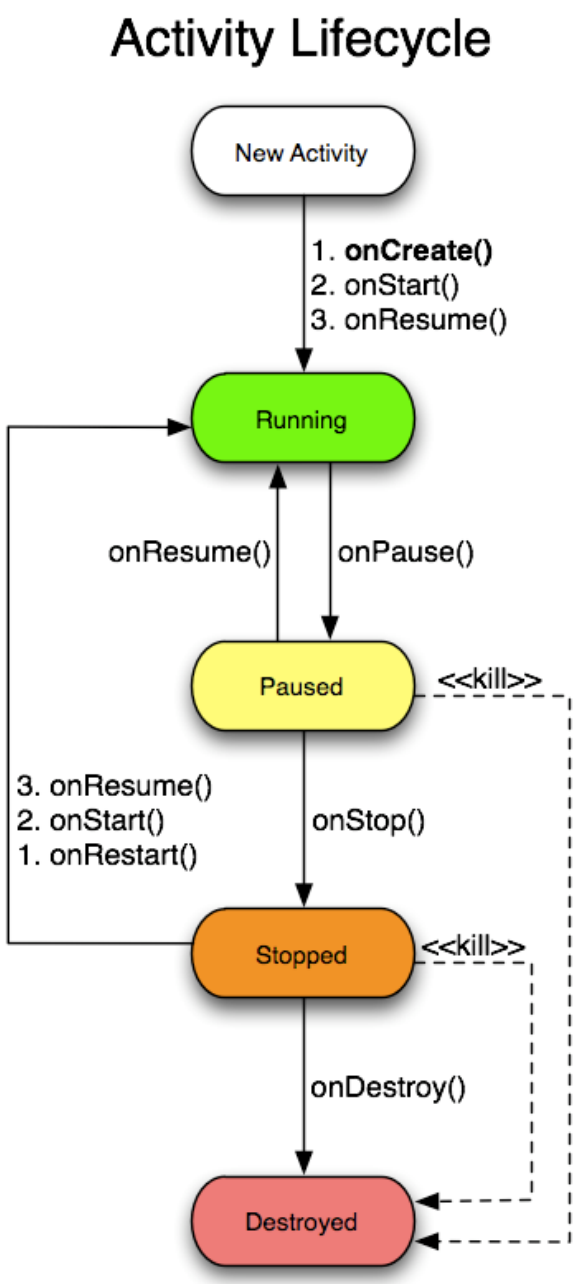
Activity states (2)

- Paused
  - If an activity has lost focus but is still visible (that is, a new non-full-sized or transparent activity has focus on top of your activity), it is *paused*.
  - 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.



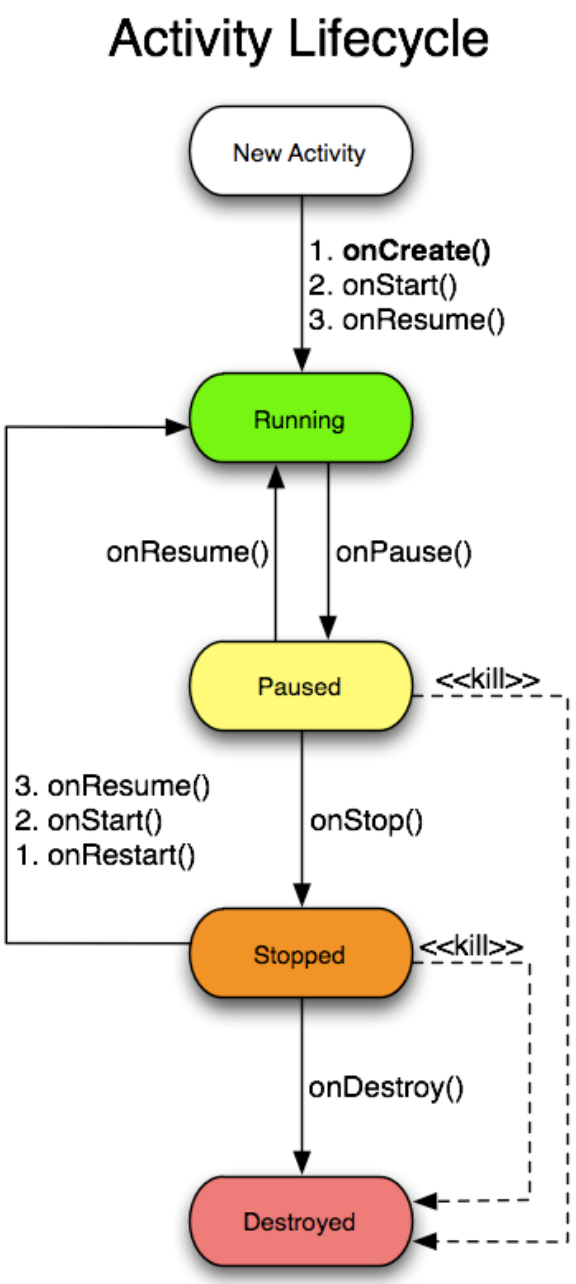
Activity states (3)

- Stopped
  - If an activity is completely obscured by another activity, it is *stopped*.
  - 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.



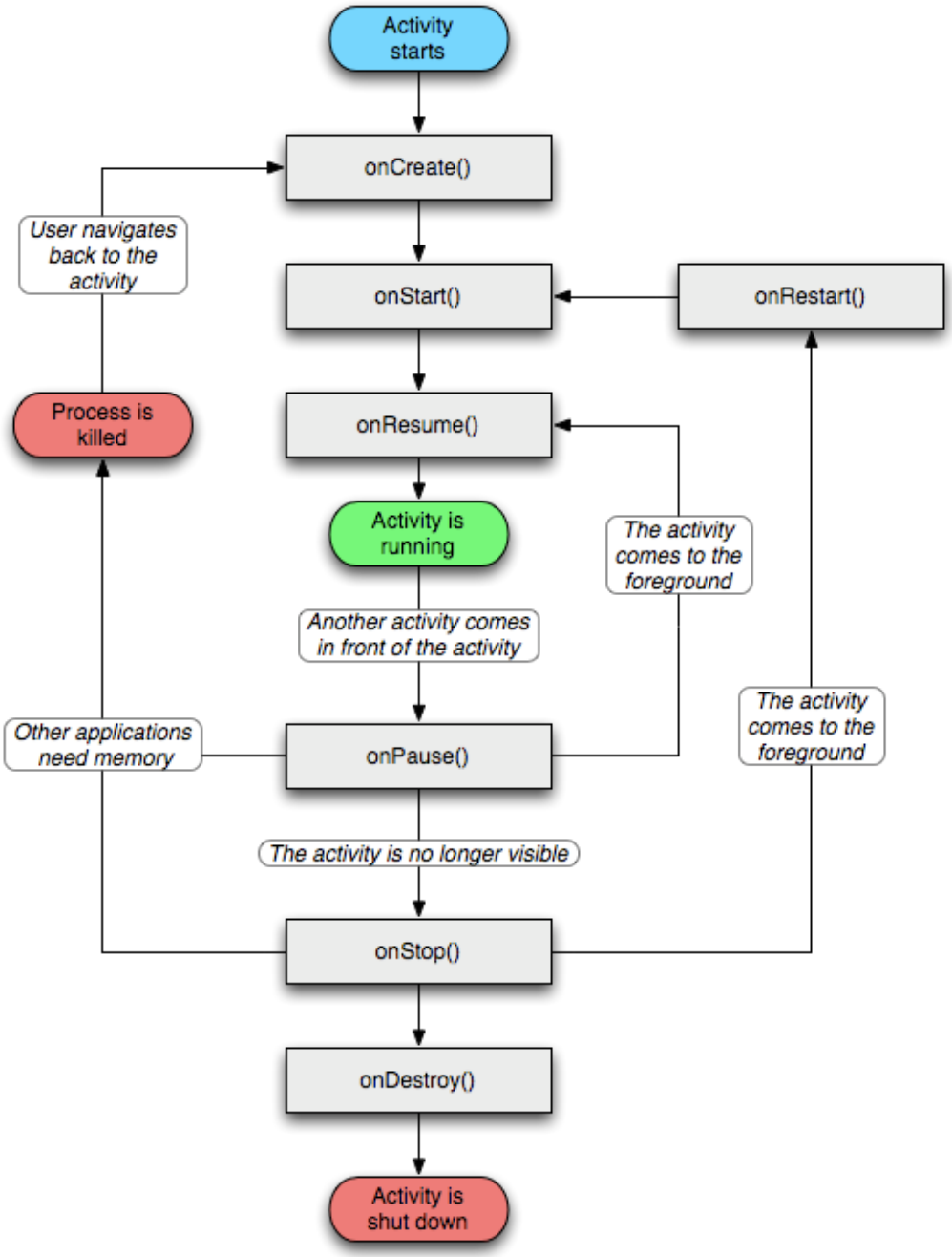
Activity states (4)

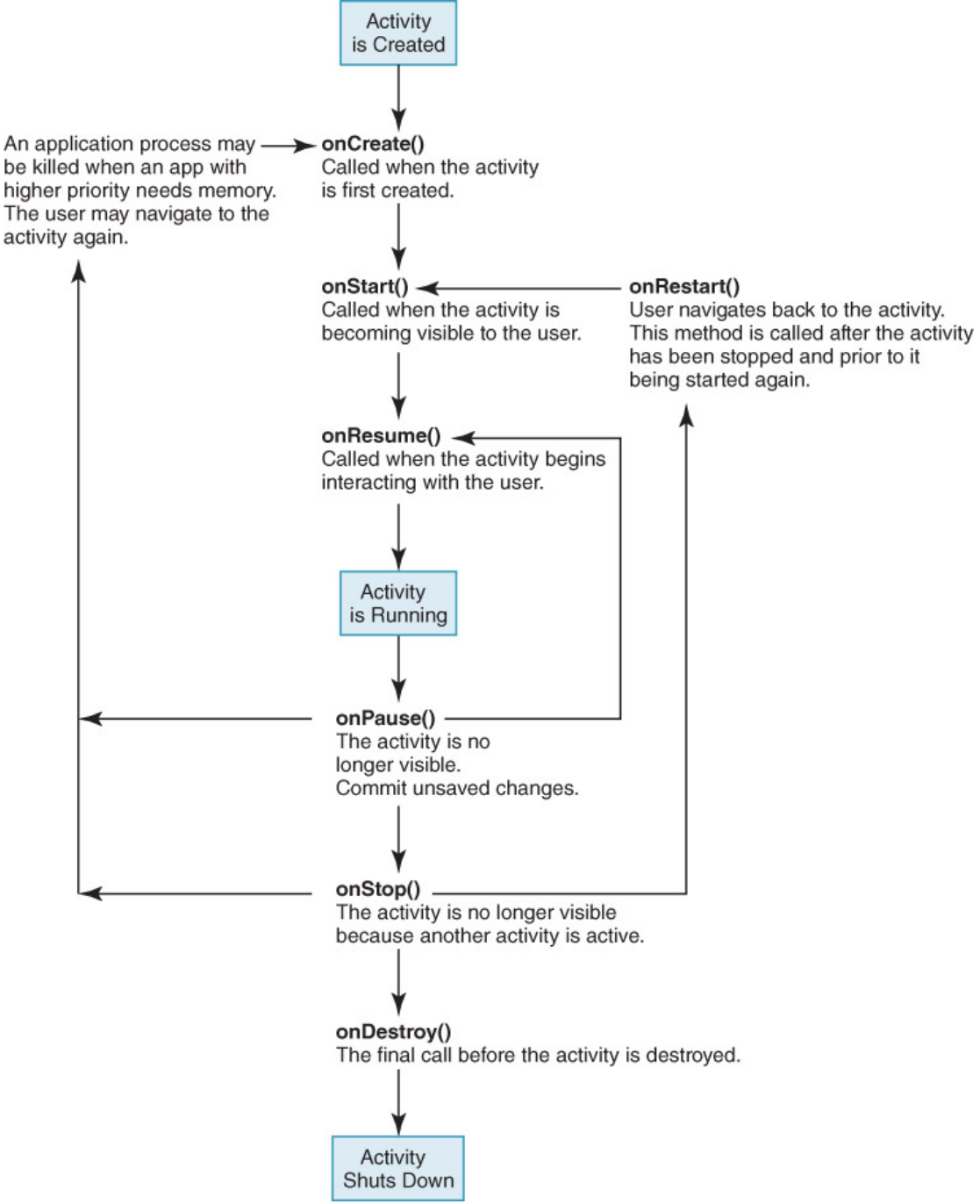
- Destroyed
  - If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish, or simply killing its process.
  - When it is displayed again to the user, it must be completely restarted and restored to its previous state.



Callbacks

- The Activity class defines the following 7 callback methods:
  - onCreate()
  - onStart()
  - onResume()
  - onPause()
  - onStop()
  - onRestart()
  - onDestroy()




Activity is Created
An application process may be killed when an app with higher priority needs memory. The user may navigate to the activity again.
onCreate()
Called when the activity is first created.
onStart()
Called when the activity is becoming visible to the user.
onRestart()
User navigates back to the activity. This method is called after the activity has been stopped and prior to it being started again.
onResume()
Called when the activity begins interacting with the user.
Activity is Running
onPause()
The activity is no longer visible.
Commit unsaved changes.
onStop()
The activity is no longer visible because another activity is active.
onDestroy()
The final call before the activity is destroyed.
Activity Shuts Down

Callbacks for creation and destroy

- The onCreat() method is automatically called when an activity is first created.
  - It can be used to initialize the activity, such as inflating the layout for the user interface of the activity.
- The onDestroy() method is called when a running activity exists and need to be destroyed by the system.

onStart() and onResume()

- When an activity has been created and become visible to the user, the onStart() method is called.
- For the user to begin interacting with the application activity, onResume() is called.
  - Both onStart() and onResume() are performed in sequence automatically after onCreate().
  - Actions such as transition animations and access to required devices, such as the camera, are often initiated with the onResume() method.

onResume()

- When the Activity reaches the top of the activity stack and becomes the foreground process, the onResume() method is called.
- Although the Activity might not be visible yet to the user, this is the most appropriate place to retrieve any instances of resources (exclusive or otherwise) that the Activity needs to run.
  - The onResume() method is the appropriate place to start audio, video, and animations.

onPause()

- The onPause() method is most often used for saving a persistent state the activity might be editing.
- Here, the Activity should stop any audio, video, and animations it started in the onResume() method.
- This is also where you must deactivate resources such as database Cursor objects if you have opted to manage them manually, as opposed to having them managed automatically.

onPause()

- The onPause() method can also be the last chance for the Activity to clean up and release any resources it does not need while in the background.
- You need to save any uncommitted data here, in case your application does not resume.
- NOTICE: The Activity needs to perform anything in the onPause() method quickly.
  - The new foreground Activity is not started until the onPause() method returns.

onStop() and onRestart()

- Once an activity has been paused and a new activity comes into focus, the onStop() method is called.
  - The activity is no longer visible to the user.
- onRestart() is triggered when an activity has been stopped and then restarted when the user navigates back to the activity.

onDestroy()

- onDestroy() performs final cleanup prior to an activity's destruction.
- It can be triggered in two ways
  - When the app has ended the activity naturally
  - When the system is temporarily destroying this instance of the activity to save space

Temporarily destroying activities

- Android devices usually have limited amount of memory.
- In some situations, the system may require more memory while an application is running.
- In this situation, the system may kill processes that are on pause in order to reclaim resources.

Low memory space available

- Under low-memory conditions, the Android operating system can kill the process for any Activity that has been paused, stopped, or destroyed.
- This essentially means that any Activity not in the foreground is subject to a possible shutdown.

- If the Activity is killed after onPause(), the onStop() and onDestroy() methods might not be called.

The killing operation

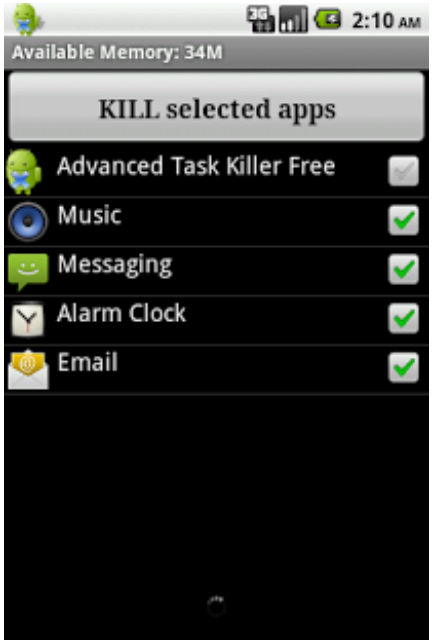
- The act of killing an Activity does not remove it from the activity stack.
- Instead, the Activity state is saved into a Bundle object, assuming the Activity implements and uses onSaveInstanceState() for custom data, though some View data is automatically saved.
- When the user returns to the Activity later, the onCreate() method is called again, this time with a valid Bundle object as the parameter.

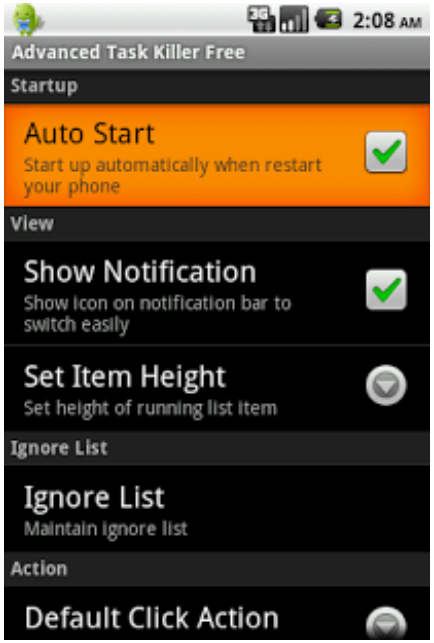
Avoiding Activity Objects Being Killed

- The more resources released by an Activity in the onPause() method, the less likely the Activity is to be killed while in the background.
- Some tools can be used to periodically sweep the stopped activities away.

Advanced Task Killer

- Three destroy levels
  - **Safe**: killing applications that are not running but have allocated memory
  - **Aggressive**: additionally killing the applications running background besides applications that are not running
  - **Crazy**: killing all applications except the foreground application



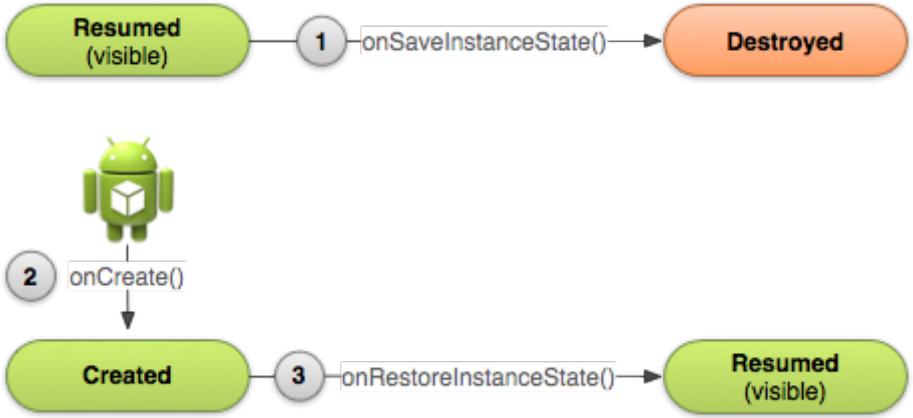


3.2 Starting, Saving, and Restoring an Activity

- When an activity is paused or stopped, the state of the activity is retained.
  - This means that an Activity object is still held in memory, along with all of its information about its data objects.
- When the system destroys an activity in order to recover memory, the memory for that activity object is also destroyed.
  - This can be a problem because once this memory has been erased, the system cannot simply resume the activity.

Bundle

- If an Activity is vulnerable to being killed by the Android operating system due to low memory, the Activity can save state information to a Bundle object using the onSaveInstanceState() callback method.
- A Bundle object is a container for the activity state information that can be saved.



https://developer.android.com/images/training/basics/basic-lifecycle-savestate.png?hl=zh-tw

Bundle preparation

- A collection of put methods can be used to insert data into the Bundle.
  - putChar()
  - putString()
  - putBoolean()
  - …

onSaveInstanceState()

- You can add the values with their indexes to the budle in onSaveInstanceState().

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
  // Save the user's current game state
  savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
  savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

  // Always call the superclass so it can save the view hierarchy state
  super.onSaveInstanceState(savedInstanceState);
}
```

https://developer.android.com/training/basics/activity-lifecycle/recreating?hl=zh-tw

Read Bundle back

- When this Activity is returned to later, this Bundle is passed into the onCreate() method, allowing the Activity to return to the exact state it was in when the Activity paused.
- You can also read Bundle information after the onStart() callback method using the onRestoreInstanceState() callback.
- If there is no state information to restore, the Bundle will contain a null value.

Code example

- The following code shows that the app keeps track of the user's playing level in a game.

```
static final String CHESS_LEVEL = "playerLevel";
private int mPlayerLevel;
…
@Override
public void onSaveInstanceState(Bundle savedInstanceState){
   savedInstanceState.putInt(CHESS_LEVEL, mPlayerLevel);
   super.onSaveInstanceState(savedInstanceState);
}
```

Code example

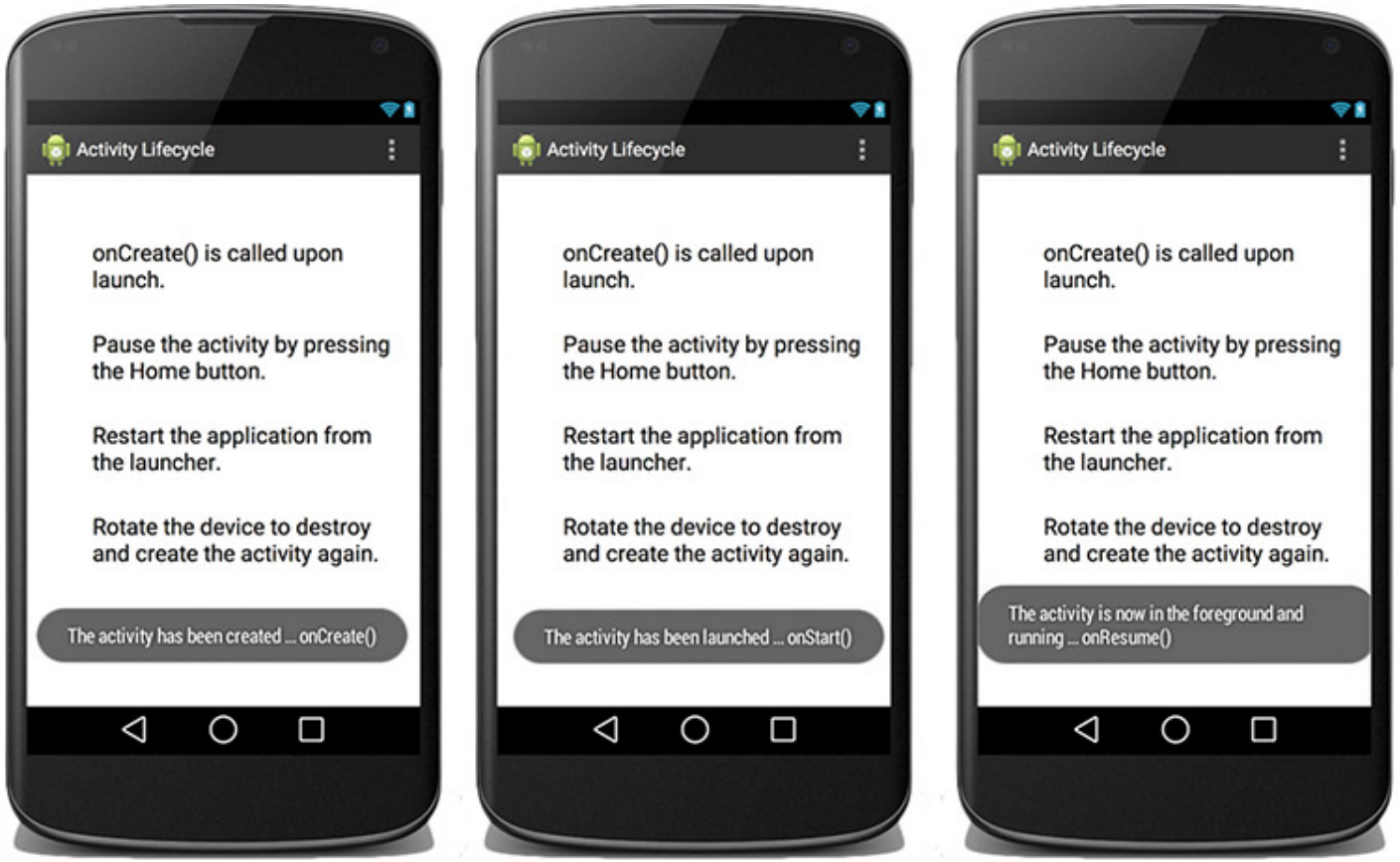
- The following code shows that the app restore the value of mPlayerLevel.

```
@Override
protected void onCreate(Bundle savedInstanceState){
   super.onCreate(savedInstanceState);

   if (savedInstanceState != null) {
       mPlayerLevel = savedInstanceState.getInt(CHESS_LEVEL);
   }
}
```

Lab example 3-1: Activity Lifecycle Exploration App

- This lab example is used for the purpose of exploring Android activities.
- The main objective of the app for this example is to experience callback methods.
- Four experiments
  1. The system creation of an Activity
  2. The system launching of an Activity
  3. An Activity running in the foreground
  4. And activity paused by another Activity running in front of it
  5. The Activity is no longer visible on the screen
  6. An Activity is restarted

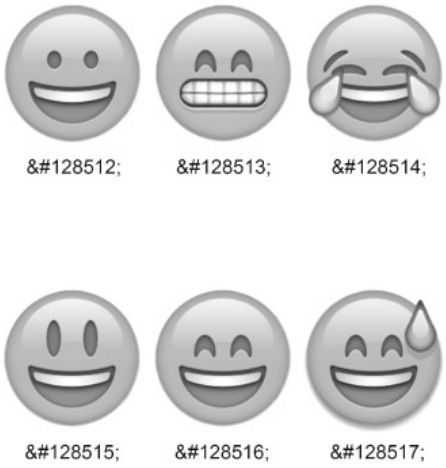
Lifecycle Experiments



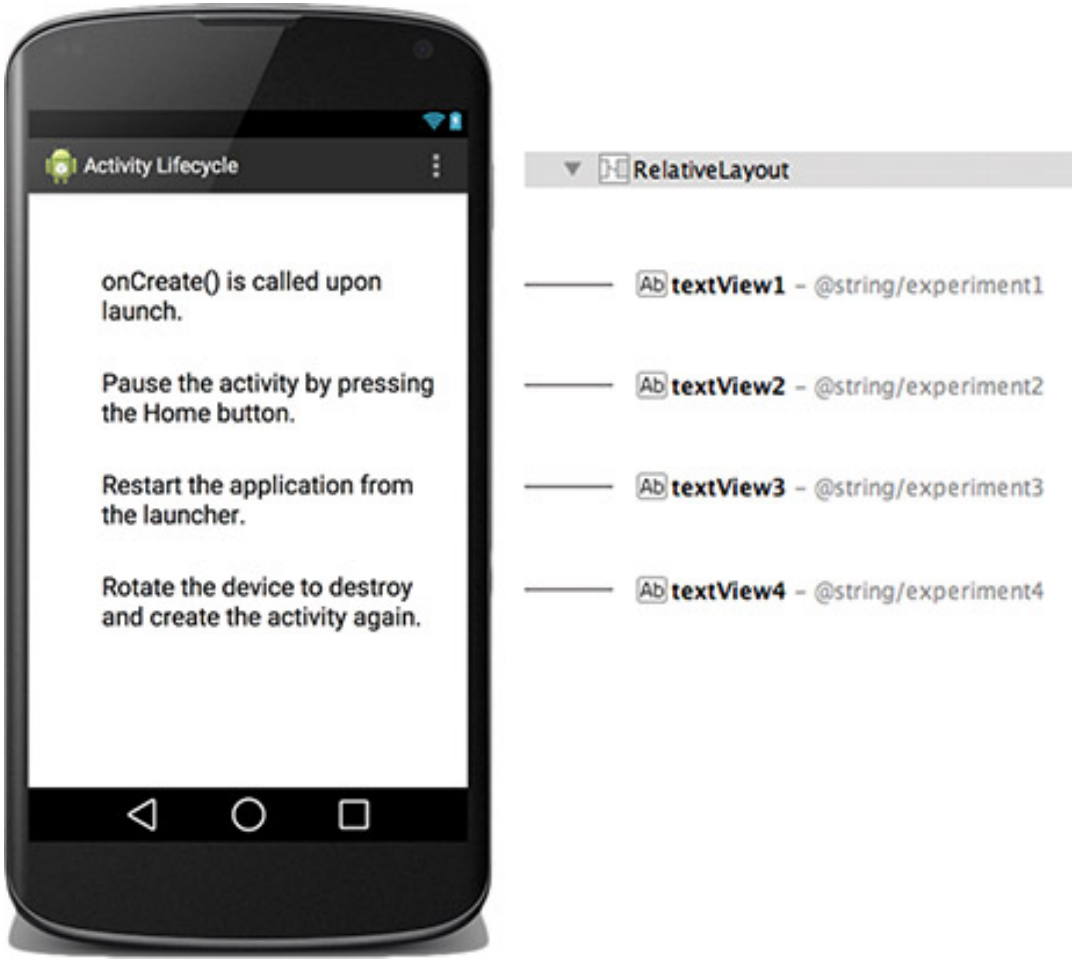
External Value Resources

strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">Activity Lifecycle</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>

    <string name="experiment1">onCreate() is called upon launch.</string>
    <string name="experiment2">Pause the activity by pressing the Home button.</string>
    <string name="experiment3">Restart the application from the launcher.</string>
    <string name="experiment4">Rotate the device to destroy and create the activity again.</string>

    <!-- MESSAGES DISPLAYED FOR ACTIVITY EVENTS -->
    <string name="create_message">The activity has been
        created  &#8230; onCreate()</string>
    <string name="start_message">The activity has been
        launched &#8230; onStart()</string>
    <string name="resume_message">The activity is now
        in the foreground and running &#8230; onResume()</string>
    <string name="pause_message">Another activity is
        in front of this activity &#8230; onPause()</string>
    <string name="stop_message">The activity is no
        longer visible on the screen &#8230; onStop()</string>
    <string name="restart_message">The activity is
        restarting after being stopped &#8230; onRestart()</string>
</resources>
```



The Layout


Activity Lifecycle
onCreate() is called upon launch.
Pause the activity by pressing the Home button.
Restart the application from the launcher.
Rotate the device to destroy and create the activity again.
RelativeLayout
textView1 - @string/experiment1
textView2 - @string/experiment2
textView3 - @string/experiment3
textView4 - @string/experiment4

```
package com.cornez.activitylifecycle;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.widget.Toast;

public class MyActivity extends Activity {
   private String createMsg;
   private String startMsg;
   private String resumeMsg;
   private String pauseMsg;
   private String stopMsg;
   private String restartMsg;

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

      initializeMessages();

      Toast.makeText(this, createMsg, Toast.LENGTH_LONG).show();
   }

   private void initializeMessages(){
      createMsg = (String) getResources().getText(R.string.create_message);
      startMsg = (String) getResources().getText(R.string.start_message);
      resumeMsg = (String) getResources().getText(R.string.resume_message);
      pauseMsg = (String) getResources().getText(R.string.pause_message);
      stopMsg = (String) getResources().getText(R.string.stop_message);
      restartMsg = (String) getResources().getText(R.string.restart_message);
   }
```

```
@Override
  protected void onStart(){
     super.onStart();
     Toast.makeText(this, startMsg, Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onResume(){
     super.onResume();
     Toast.makeText(this, resumeMsg, Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onPause(){
     super.onPause();
     Toast.makeText(this, pauseMsg, Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onStop(){
     super.onStop();
     Toast.makeText(this, stopMsg, Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onRestart(){
     super.onRestart();
     Toast.makeText(this, restartMsg, Toast.LENGTH_LONG).show();
  }
```

```
@Override
  public boolean onCreateOptionsMenu(Menu menu) {
     // Inflate the menu; this adds items to the action bar if it is present.
     getMenuInflater().inflate(R.menu.my, menu);
     return true;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
     // Handle action bar item clicks here. The action bar will
     // automatically handle clicks on the Home/Up button, so long
     // as you specify a parent activity in AndroidManifest.xml.
     int id = item.getItemId();
     if (id == R.id.action_settings) {
        return true;
     }
     return super.onOptionsItemSelected(item);
  }
}
```

Experiments

1. Launch the app. The callbeck methods are onCreate(), onStart() and onResume().
2. Pause the activity by pressing the Home button. The callback methods you should see are onPause() and onStop().
3. Restart the appilication from the launcher. The callback methods are onRestart(), onStart(), and onResume().
4. This must be done using a physical device. An activity is destroyed when the screen of a device is rotated. Once the activity is destroyed, it is created again. You will see onPause(), onStop(), onCreate(), onStart(), and onResume().

3.3 Multiple Activities and the Intent Class

- Applications that are built with multiple activities need to utilize the Intent class.
- An Intent object is a message from one component to another component, either within the application or outside the application.
  - This class provides the framework for the navigation from one screen to another.

Intents

- Intents are designed to communicate messages between three application core components:
  - Activities
  - Broadcast receivers
  - Services
- An Intent is an asynchronous message mechanism.
- An Intent is also an abstract description of an operation to be performed.

Main parts in an Intent

- Action
  - This component describes the action to take upon receiving the Intent.
- Data
  - This component identifies the data resource. More specifically, it contains the URI of the data.
- Category
  - The category of an Intent is not required, but it can be sometimes useful.
  - A category is specified when it is necessary to define a particular component that will handle the Intent.

Intent Resolution

- Intents are broadly grouped into two categories: explicit and implicit.
  - Often the explicit Intents will not include any other information, simply being a way for an application to launch various internal activities it has as the user interacts with the application.
  - The implicit Intents must include enough information for the system to determine which of the available components is best to run for that intent.

Explicit Intents

- Explicit Intents use a specific name when starting a component
- This name will be the full Java class name of the activity or service
- The most common use of an explicit Intent is the launching of a target component with a known name within the currently running application.

Implicit Intents

- Unlike an explicit Intent, an implicit Intent does not name a specific component.
- it declares a general action to perform, which allows a component from another app to handle it.
- When an implicit Intent is created, the system locates the appropriate target component by comparing the contents of the Intent to an Intent filter.
- Intent filters are declared in the manifest file of other apps located on a given device.
- When the Intent has found a match with the Intent filter, the system starts that component and delivers it to the Intent object.

Code example

- Explicit Intent

```
public void launchTargetActivity (View view) {
   Intent intent = new Intent(getApplicationContext(), TargetActivity.class);

   startActivity(intent);
}
```

- Implicit Intent

```
Intent intent = new Intent( Intent.ACTION_VIEW,
   Uri.parse("http://www.example.com"));

startActivity(intent);
```

Code example: ACTION_DIAL

- Create a simple Intent with a predefined action (ACTION_DIAL) to launch the Phone Dialer with a specific phone number to dial in the form of a simple Uri object.

```
Uri number = Uri.parse(tel:4638800);
Intent dial = new Intent(Intent.ACTION_DIAL, number);
startActivity(dial);
```

More examples

- View a map

```
// Map point based on address
Uri location = Uri.parse("geo:0,0?q=1600+Amphitheatre+Parkway,+Mountain+View,+California");
// Or map point based on latitude/longitude
// Uri location = Uri.parse("geo:37.422219,-122.08364?z=14"); // z param is zoom level
Intent mapIntent = new Intent(Intent.ACTION_VIEW, location);
```

- Send an email with an attachment

```
Intent emailIntent = new Intent(Intent.ACTION_SEND);
// The intent does not have a URI, so declare the "text/plain" MIME type
emailIntent.setType(HTTP.PLAIN_TEXT_TYPE);
emailIntent.putExtra(Intent.EXTRA_EMAIL, new String[] {"jon@example.com"}); // recipients
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Email subject");
emailIntent.putExtra(Intent.EXTRA_TEXT, "Email message text");
emailIntent.putExtra(Intent.EXTRA_STREAM, Uri.parse("content://path/to/email/attachment"));
// You can also attach multiple items by passing an ArrayList of Uris
```

3.5 Passing Data between Activities

- The Android framework provides a simple and flexible approach to working with multiple activities.
- Android also offers an efficient model for passing information between various activities.

Data passing

- Data can be passed as a message object to an activity implemented within the application or an activity outside of the applications.
- When an Intent object is constructed, its action is specified.
  - This represents the action we want the Intent to trigger.
  - This action will also send data across process boundaries.

Passing Additional Information using Intents

- You can also include additional data in an Intent.
- The Extras property of an Intent is stored in a Bundle object.
  - putExtra(): a name for the data and a String data value.
  - The data name must include a package prefix.
    - com.android.contacts.ShowAll
- The Intent class also has a number of helper methods for getting and setting name/value pairs for many common datatypes.

Code example

- The following Intent shows a simple example of sending an intent to another activity.

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_TEXT, "Hello World");
startActivity(intent);
```

Code example

- The following Intent includes two extra pieces of information—a string value and a Boolean :

```
Intent intent = new Intent(this, MyActivity.class);
intent.putExtra("SomeStringData","Foo");
intent.putExtra("SomeBooleanData",false);
```

Code example

- The activity sending an intent to another activity outside the application

```
Intent intent = new Intent();
intent.setAction(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_SUBJECT, "Sharing URL");
intent.putExtra(Intent.EXTRA_TEXT, "http://www.url.com");
startActivity(Intent.createChooser(intent, "Share URL"));
```

- Using <Intent-filter>

```
<activity android:name=".MyActivity">
<intent-filter>
<action android:name="android.intent.action.SEND" />
<category android:name="android.intent.category.DEFAULT" />
</intent-filter>
</activity>
```

Code example

- Intent filters are defined using the <intent-filter> tag and must contain at least one <action> tag but can also contain other information.

```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.BROWSABLE" />
  <category android:name="android.intent.category.DEFAULT" />
  <data android:scheme="geoname"/>
</intent-filter>
```

Concluding Remarks