Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Activities

Federico Montori
federico.montori2@unibo.it

Table of Contents

Activity

A mobile app experience differs from its desktop counterpart

- A user lands in the application nondeterministically
- You can open your emails and see the full inbox
- If you go there from a website you may land on the “compose message” screen instead

These different contexts are called **Activities**

Activity

We call **Activity** a **screen state**

- The entry point for a user interaction
  - Can be seen as a single screen
- Has methods to react to certain events
- An application can be composed of multiple activities
  - it is not seen as an atomic whole
- Android maintains a **stack** of activities

Activity

Declare them in the Manifest before running them (Usually done automatically)

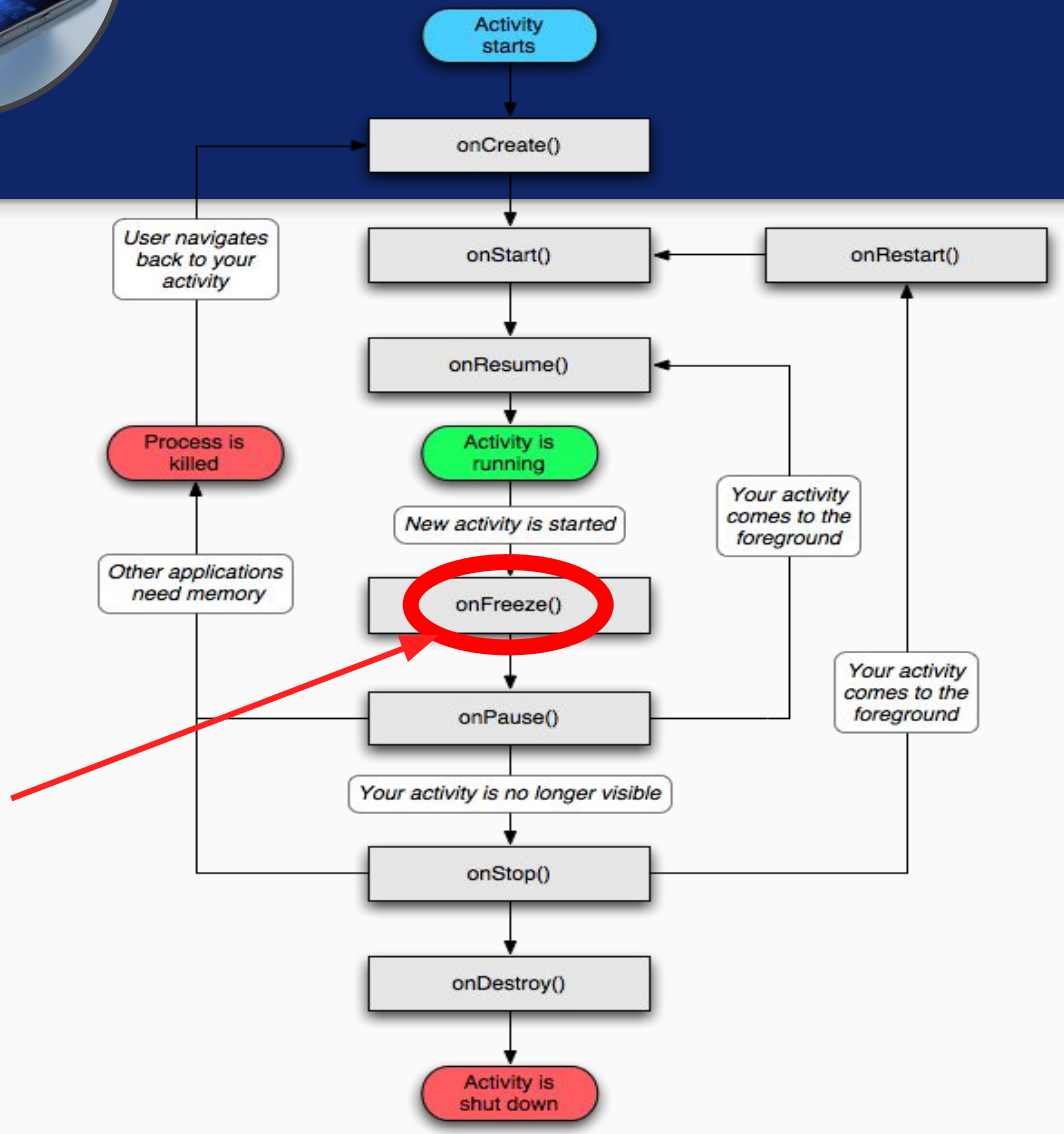
```
<application
  ... >
   <activity  android:name=".MainActivity" android:exported="true">
     <intent-filter>
       <action android:name="android.intent.action.MAIN" />
       <category android:name="android.intent.category.LAUNCHER" />
     </intent-filter>
   </activity>
</application>
```

Activity

The Android Manifest is what the operating system can read about your application.

- It tells which activities you have and how a user can access them.
- **MAIN** and **LAUNCHER** means that this activity is accessed via the app icon in the home screen.
  - We'll see more of these in the lecture about "Intents"

Activity Lifecycle

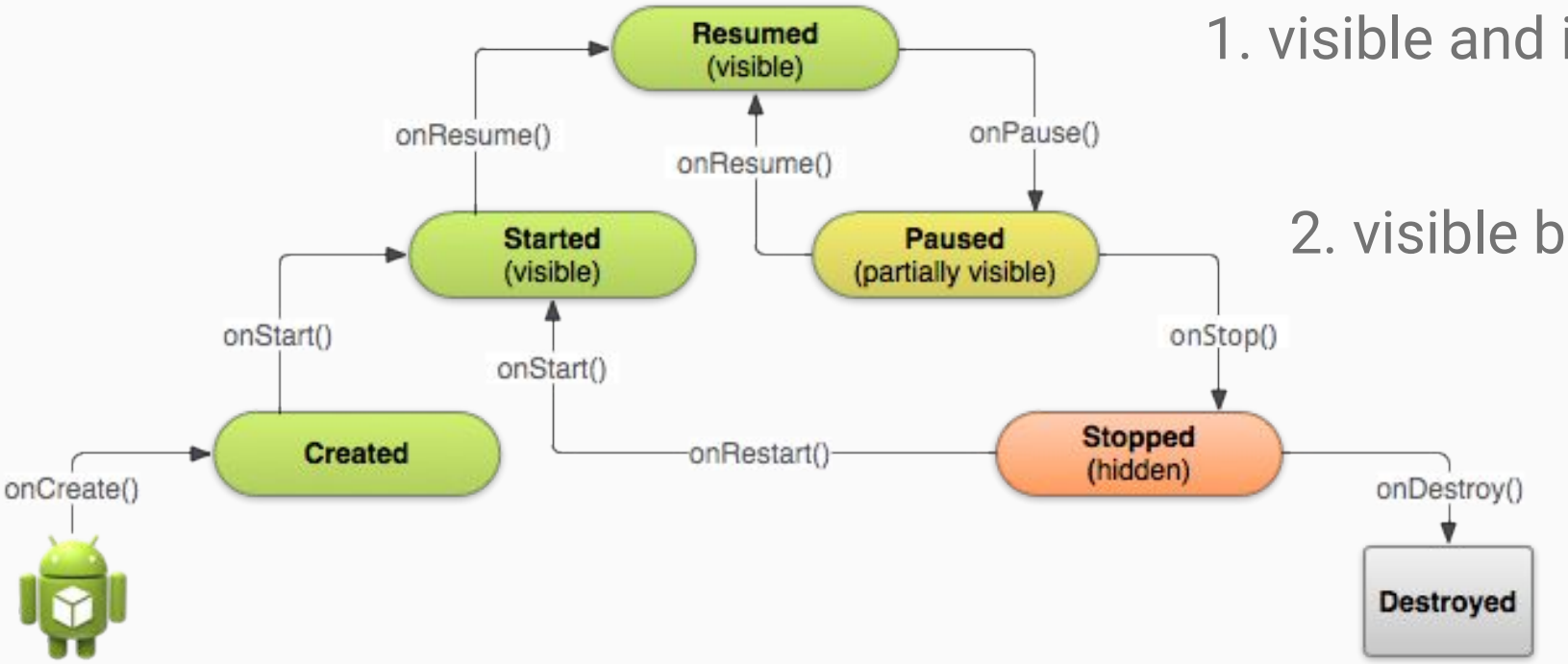


onFreeze() is not used anymore since 2008.

As the user navigates in and out the app, the Activity can go through several **states**.

We use reactive programming, since we put our code in **callbacks**, invoked when the activity transitions from one state to another.

Activity Lifecycle



1. visible and interactable

2. visible but not interactable

3. not visible

4. not in memory

This reflects the likelihood of killing the Activity's process if the system needs memory

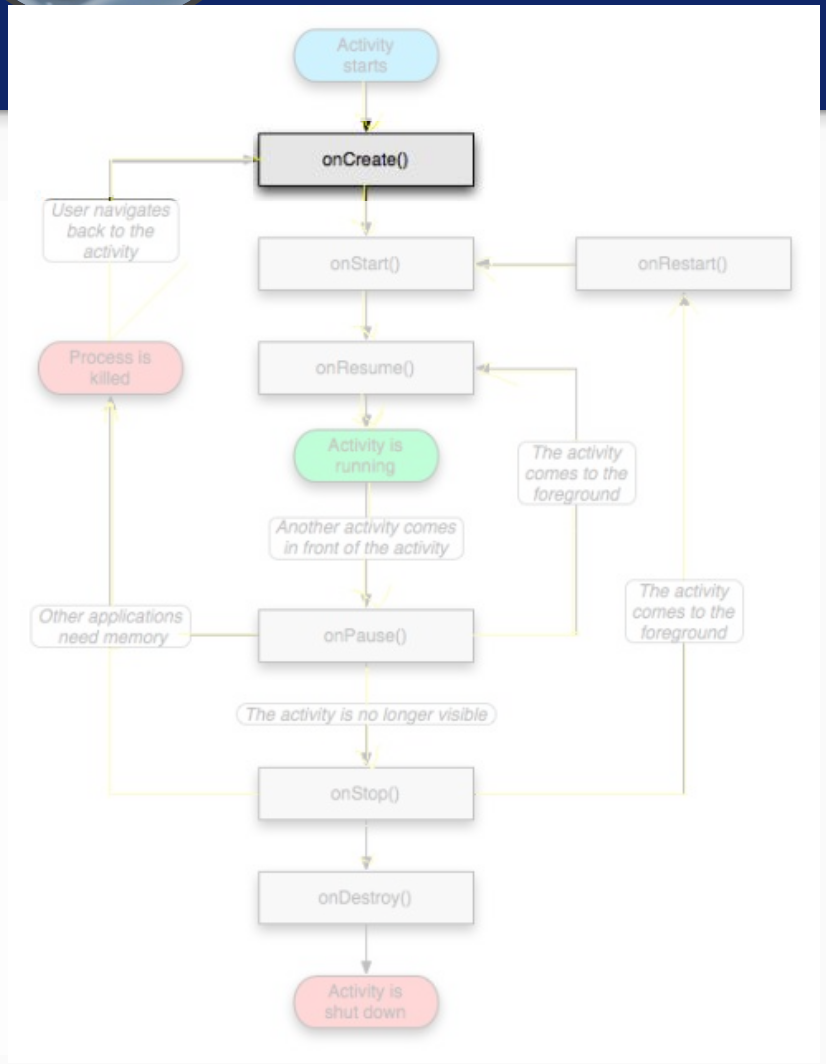
Activity Lifecycle

- Resumed
  - The activity is in the foreground, and the user can interact.
- Paused (but started...)
  - The activity is visible, maybe overlaid by another activity. Cannot execute any code nor receive direct inputs.
- Stopped (but created...)
  - Activity is hidden, in the background. It cannot execute any code.

Activity Lifecycle

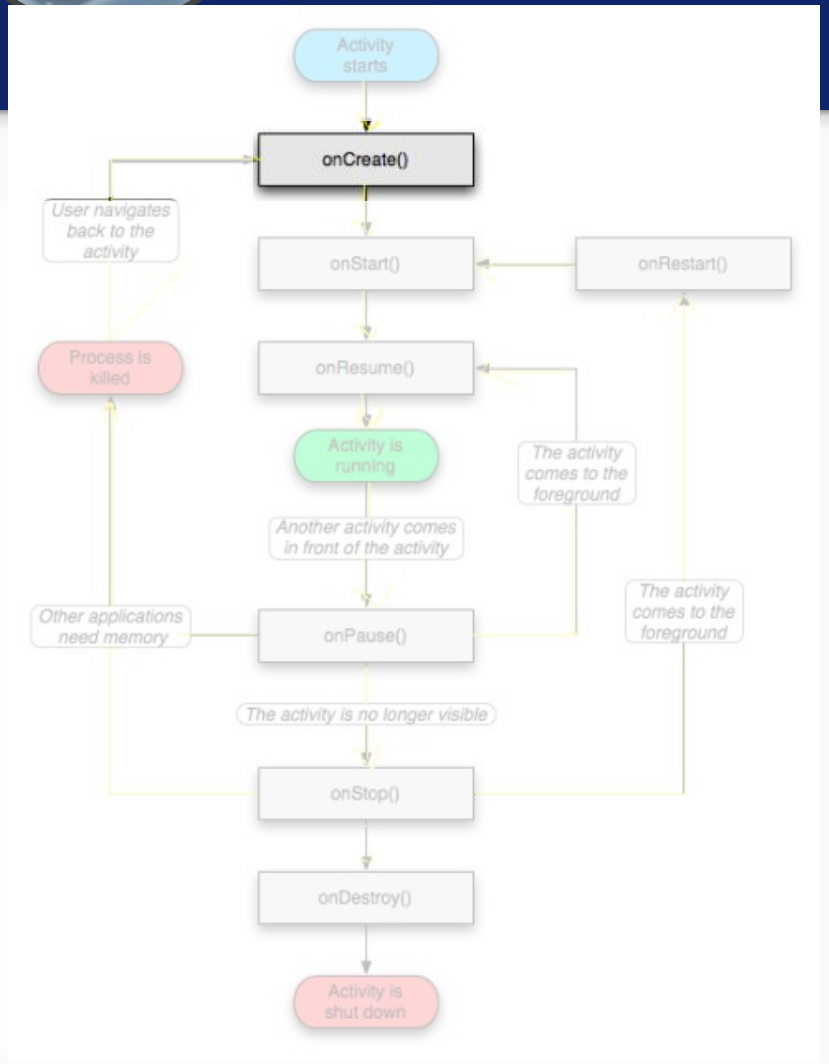
- Need to implement every single method? No!
  - It depends on the application complexity
- Why is it important to understand the activity lifecycle?
  - So your application does not crash (or do “funny” things) while the user is running something else on the smartphone
  - So your application does not consume unnecessary resources
  - So the user can safely stop your application and return to it later

Activity Lifecycle



- onCreate()
  - Called when the activity is created
  - Should contain the startup logic to be executed only once.
  - Has a **Bundle** parameter (a composite with saved data)
  - If **onCreate**() terminates, it calls **onStart**()

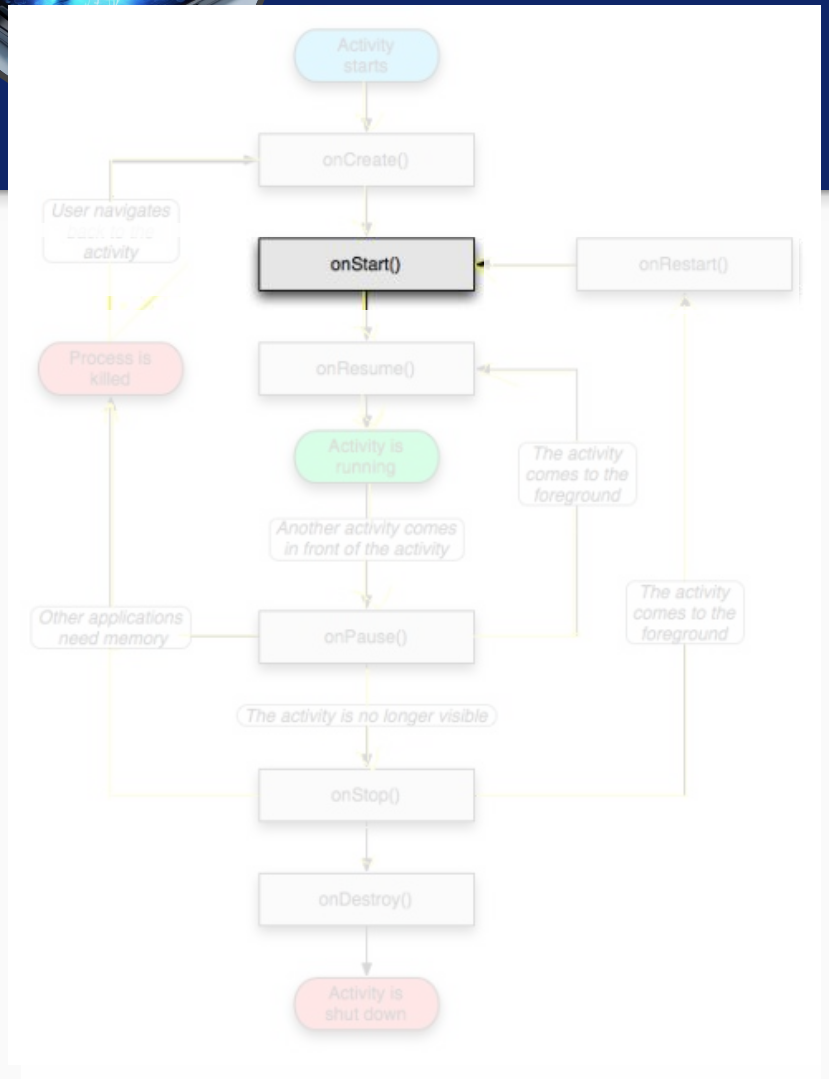
Activity Lifecycle



- onCreate()
  - Responsible for drawing the UI with **setContentView()**

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

```
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }
```



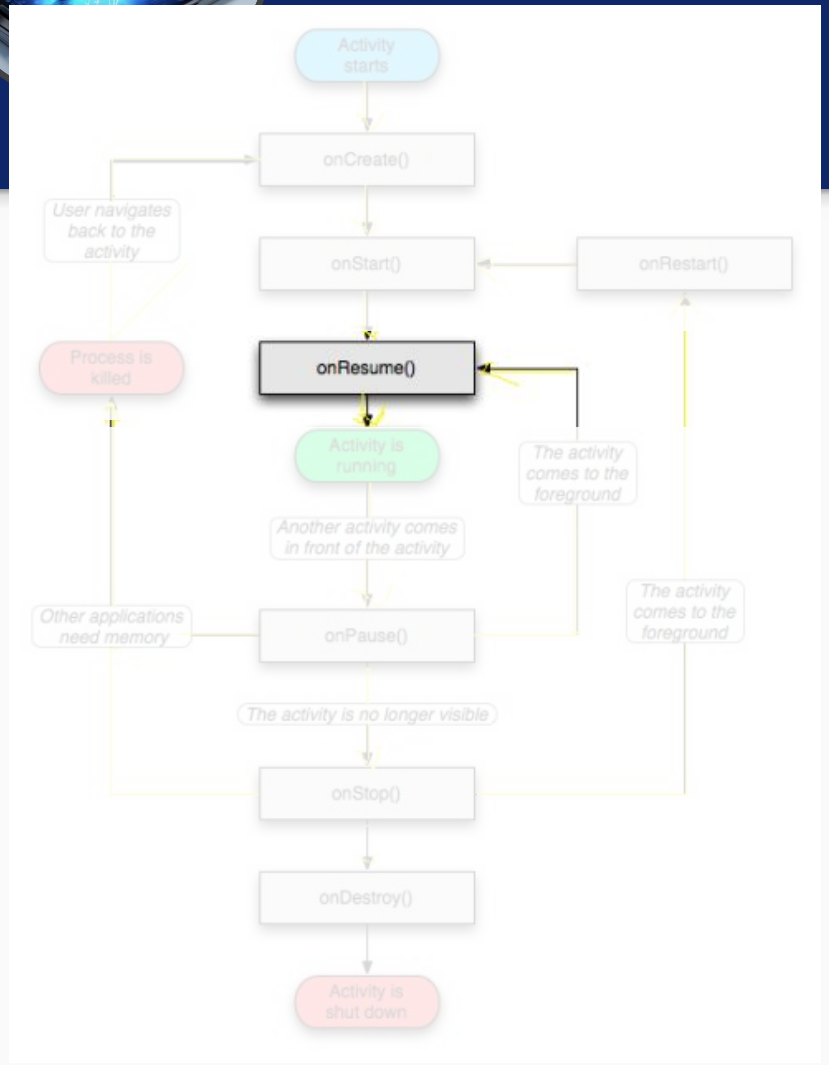
Activity Lifecycle

- OnStart()
  - Called right before it is visible to user (where the code that maintains the UI is initialized).

```
@Override
  protected void onStart() {
    super.onStart();
  }
```

```
  override fun onStart() {
    super.onStart()
  }
```

Activity Lifecycle

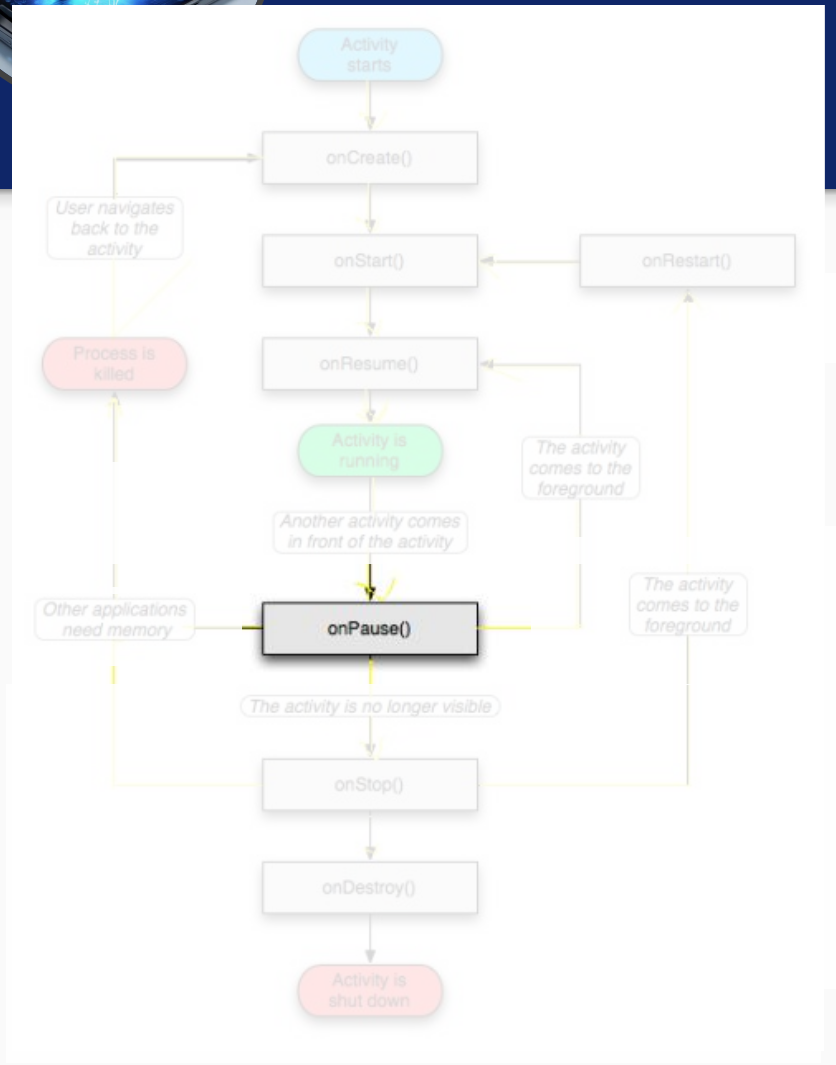


- OnResume()
  - If it successfully terminates, then the Activity is RUNNING
  - Restore components that were disposed in onPause()

```
@Override
  protected void onResume() {
    super.onResume();
  }
```

```
  override fun onResume() {
    super.onResume()
  }
```

Activity Lifecycle

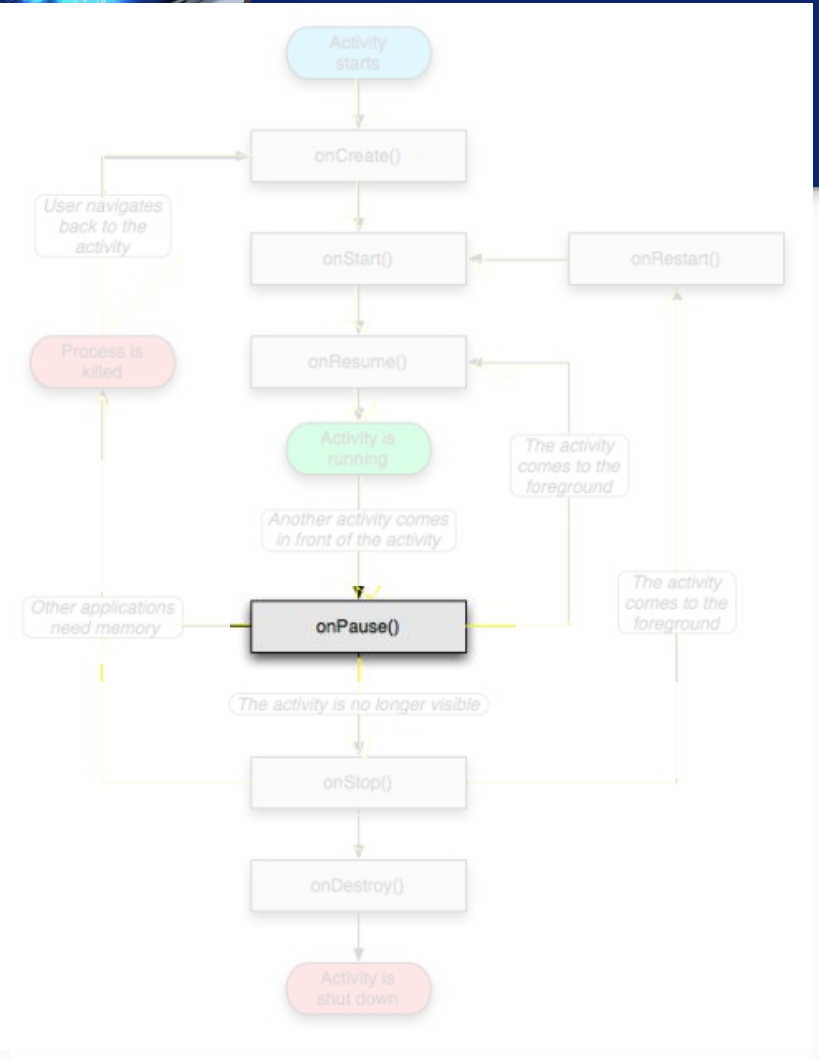


- OnPause()
  - Called when something interrupts the activity.
  - Do not save data, it is fast.
  - Stop cpu-consuming processes.

```
@Override
  protected void onPause() {
    super.onPause();
  }
```

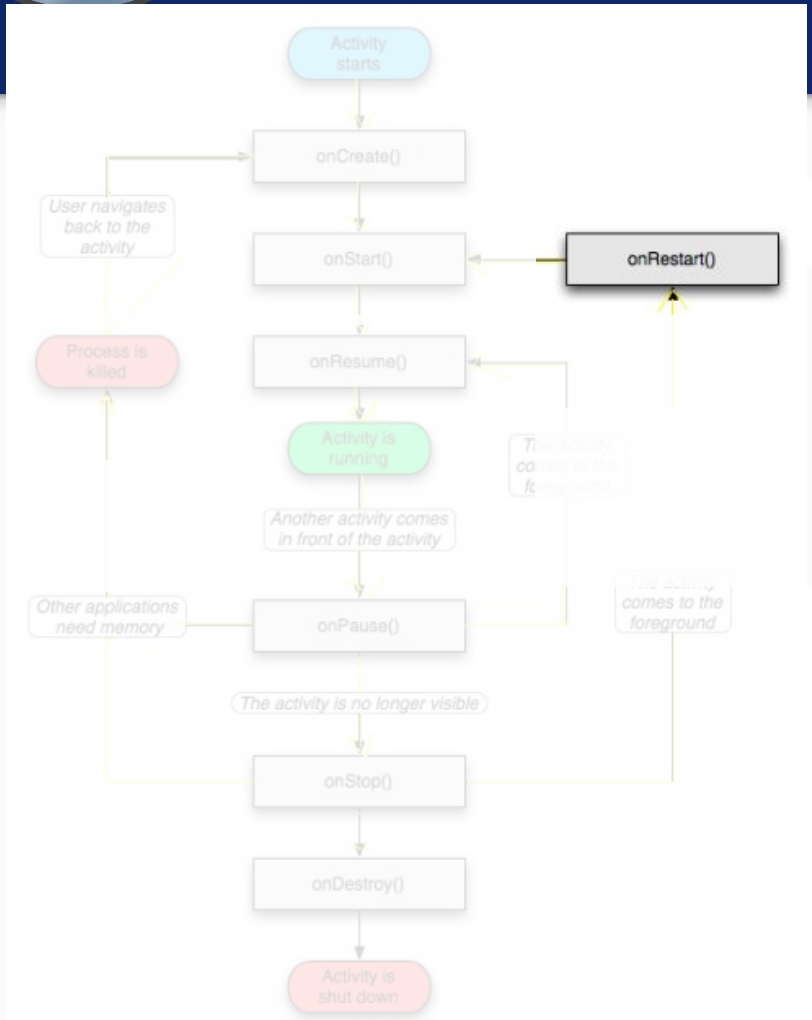
```
  override fun onPause() {
    super.onPause()
  }
```

Activity Lifecycle



- OnPause()
  - Another component (NOT FROM THE SAME ACTIVITY) requests the foreground
  - A component comes in the foreground partially hiding the activity (e.g. a dialog)
  - Another window in a multi-window application is tapped.
  - Any other event that will also imply the onStop()

Activity Lifecycle

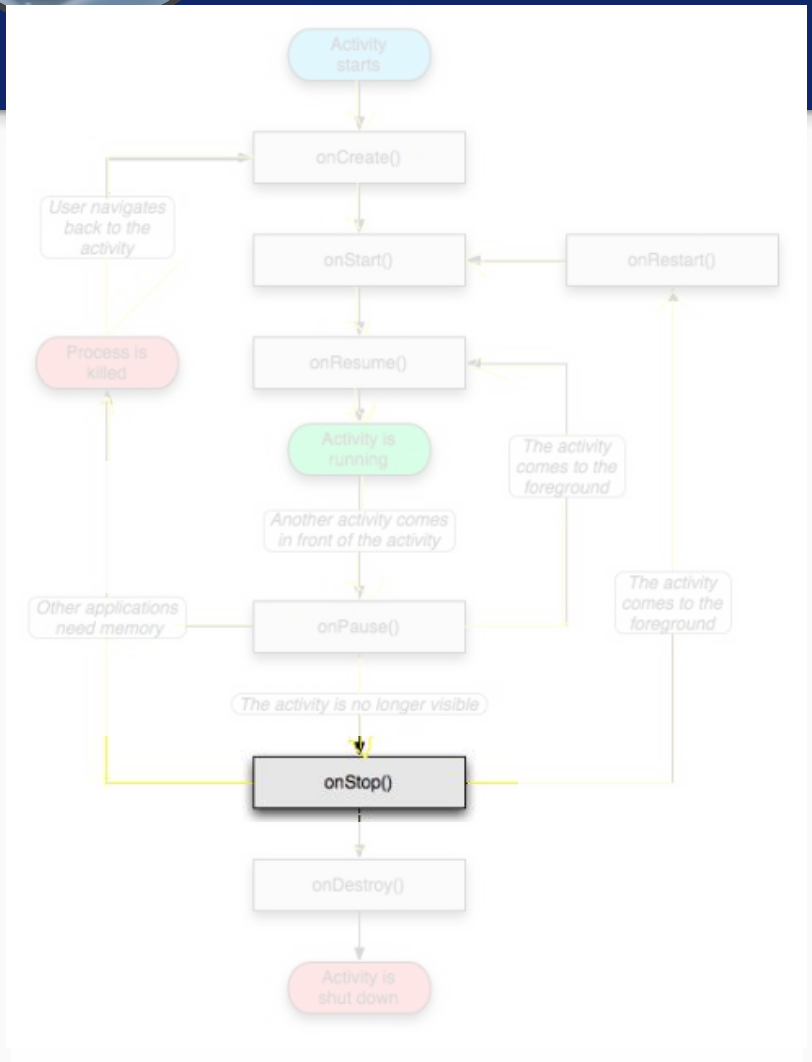


- OnRestart()
  - Only when the activity was previously stopped.

```
@Override
  protected void onRestart() {
    super.onRestart();
  }
```

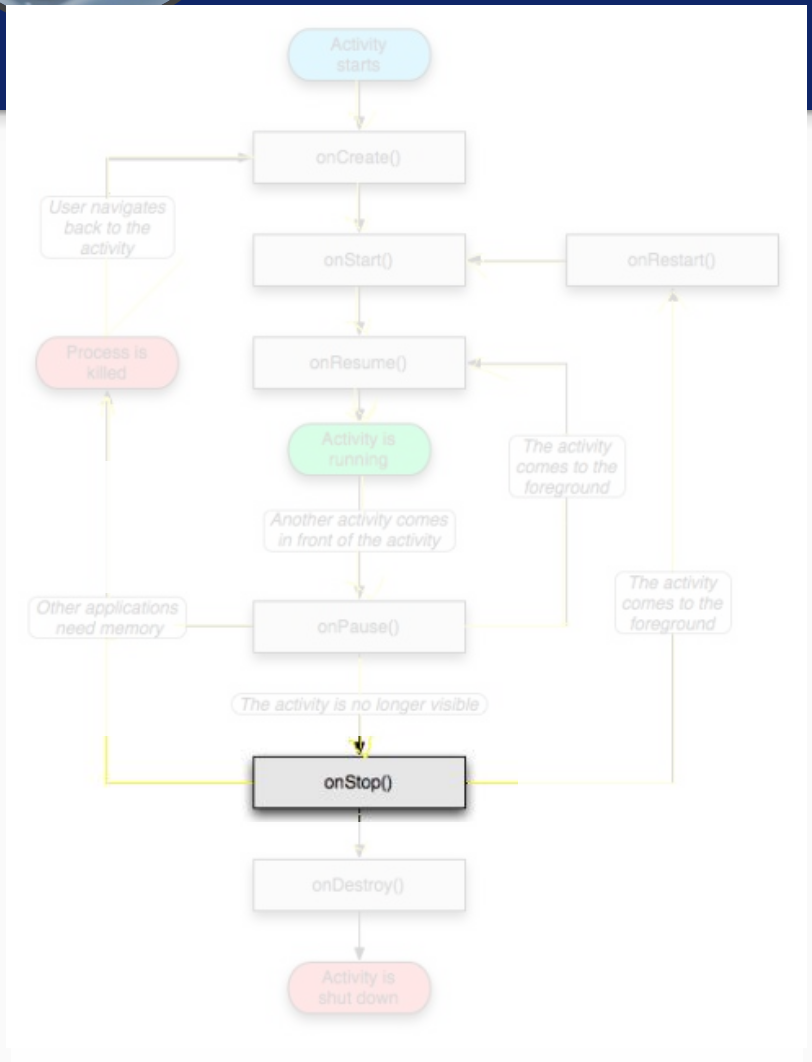
```
  override fun onRestart() {
    super.onRestart()
  }
```

Activity Lifecycle



- OnStop()
  - Activity is no longer visible to the user
  - Could be called because:
    - the activity is about to be destroyed
    - another activity comes to the foreground

Activity Lifecycle

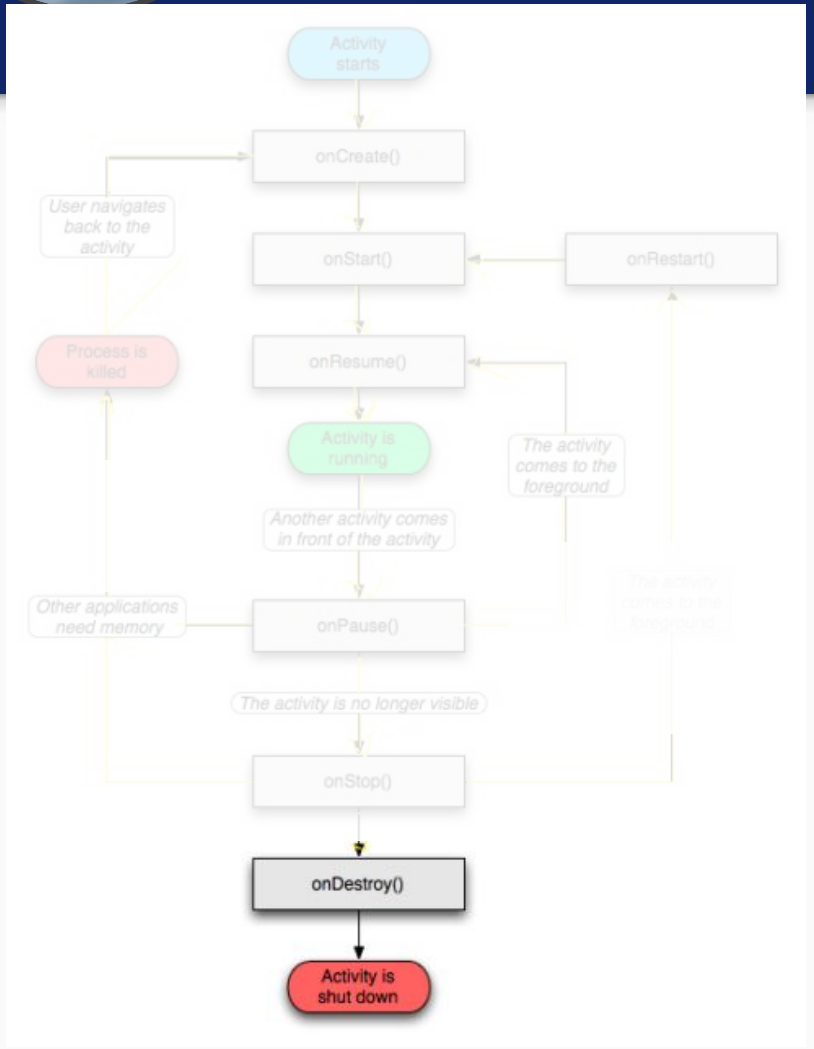


- OnStop()
  - Used to perform CPU-intense shutdown operations.

```
@Override
  protected void onStop() {
    super.onStop();
  }
```

```
  override fun onStop() {
    super.onStop()
  }
```

Activity Lifecycle

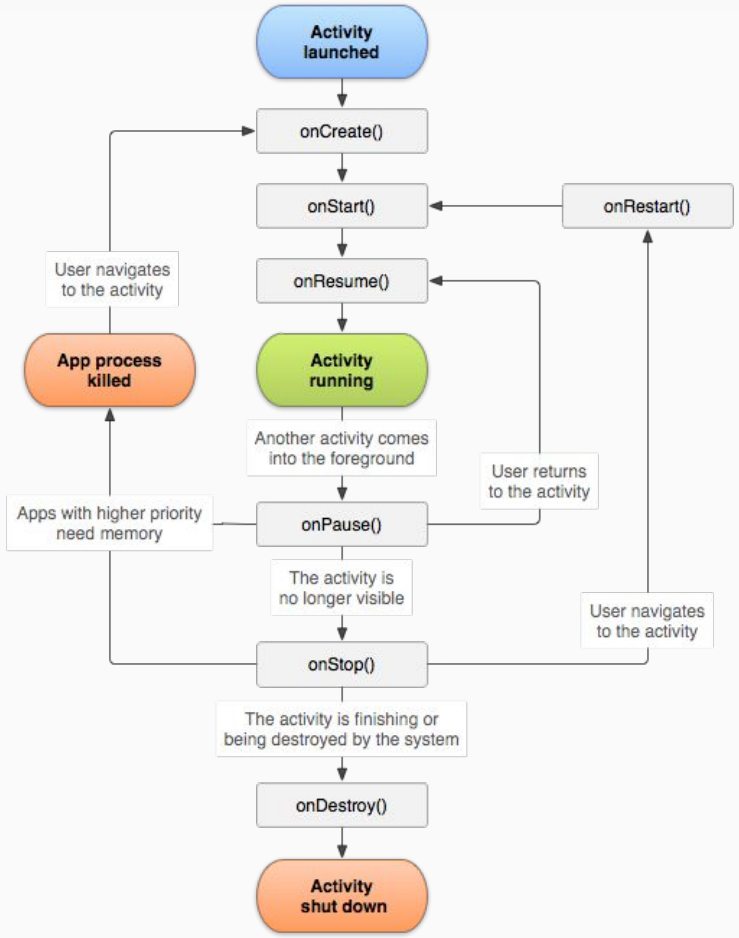


- OnDestroy()
  - The system needs some stack space
  - The screen changes the layout (portrait/landscape)
  - Someone called finish() method on this activity
    - check with isFinishing()

```
@Override
  protected void onDestroy() {
    super.onDestroy();
  }
```

```
  override fun onDestroy() {
    super.onDestroy()
  }
```

Activity Lifecycle



Three Loops:

- **Entire lifetime**
  - Between onCreate() and onDestroy().
  - Setup of global state in onCreate()
  - Release remaining resources in onDestroy()
- **Visible lifetime**
  - Between onStart() and onStop().
  - Maintain resources that have to be shown to the user.
- **Foreground lifetime**
  - Between onResume() and onPause().
  - Code should be light.

Logs with Logcat

The **Logcat** window in Android Studio helps you debug your app by displaying logs from your device in real time.

For example, messages that you added to your app with the **Log** class…

Class Log has several methods that define the importance of the log message.

They can be filtered by Logcat.

Logs with Logcat

LOWEST PRIORITY

```
Log.v ("LABEL", "message")          // VERBOSE
Log.d ("LABEL", "message")          // DEBUG
Log.i ("LABEL", "message")          // INFORMATION
Log.w ("LABEL", "message")          // WARNING
Log.e ("LABEL", "message")          // ERROR
Log.wtf ("LABEL", "message")        // SHOULD NEVER HAPPEN IN LIFE
```

HIGHEST PRIORITY

Recreating Activities

When an activity is destroyed and then navigated back, the system recreates a new instance. We typically want everything back as it was, which is saved to a Bundle called **Instance State**.

- Android keeps the state of each view
  - Remember to assign unique Ids to them
  - So, no explicit code is needed for the “basic” behavior
- What if I want to save more data?
  - Variables, states…

Recreating Activities

What if I want to save more data?

- Override **onSaveInstanceState**() and **onRestoreInstanceState**()
- Use a ViewModel (we will see that later on…)

onSaveInstanceState() called likely right before **onStop**()

```
static final String STATE_SCORE = "playerScore";
@Override
public void onSaveInstanceState
      (Bundle savedInstanceState) {

      super.onSaveInstanceState (savedInstanceState);
      savedInstanceState.putInt(
            STATE_SCORE, mCurrentScore
      );
}
```

```
override fun onSaveInstanceState(
            savedInstanceState : Bundle) {
            super.onSaveInstanceState
                  (savedInstanceState)
            outstate.putInt(
                  STATE_SCORE, mCurrentScore
            )
      }

companion object { val STATE_SCORE = "playerScore" }
```

Recreating Activities

onRestoreInstanceState() called likely right after **onStart**()

```
@Override
public void onRestoreInstanceState
        (Bundle savedInstanceState) {
            // Call the superclass to restore the views
        super.onRestoreInstanceState
            (savedInstanceState);
        mCurrentScore  =
            savedInstanceState.getInt(STATE_SCORE);
}
```

```
override fun onRestoreInstanceState
        (savedInstanceState: Bundle) {
        // Call the superclass to restore the views
        super.onRestoreInstanceState
            (savedInstanceState)
        mCurrentScore  =
            savedInstanceState.getInt(STATE_SCORE)
}
```

As an alternative, you can do so in the **onCreate**() method, as the bundle (possibly null) is passed in as a parameter.

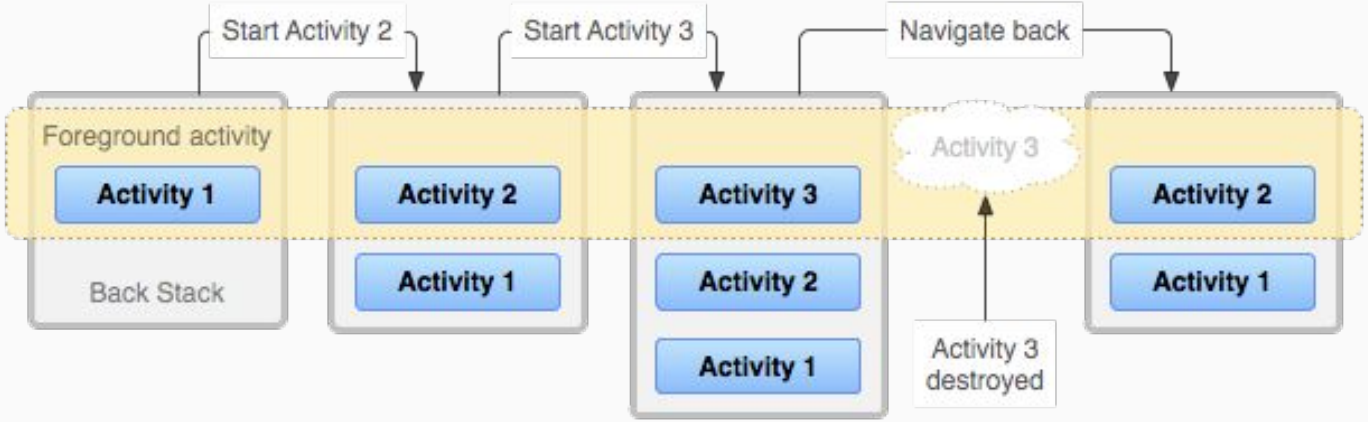
Activities: Summary

- Activities should be declared in the Manifest
- Extend the Activity class
- Code wisely
  - Put your code in the right place
  - Optimize it
  - Test even on low-end devices
  - Watch out, configuration changes (rotating screens) destroy the activity

Tasks and BackStack

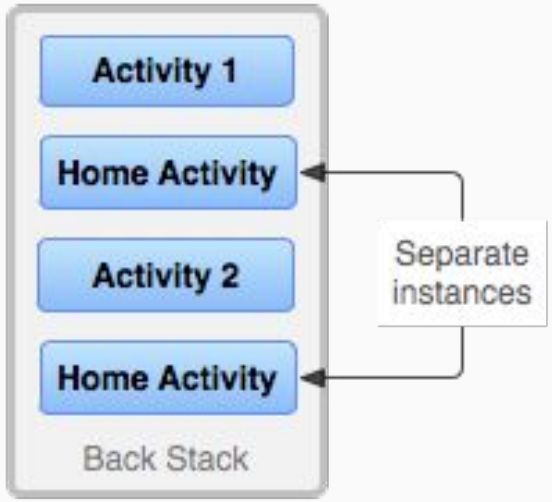
Activities in the same app can occur on top of each other, in such case the previous activity (stopped) stays saved in the BackStack.

- By navigating back the user pops the current activity from the BackStack and destroys it, restoring the one on top.



Tasks and BackStack

Launching the same Activity in two different phases of the same storyline causes the creation of two separate instances by default.

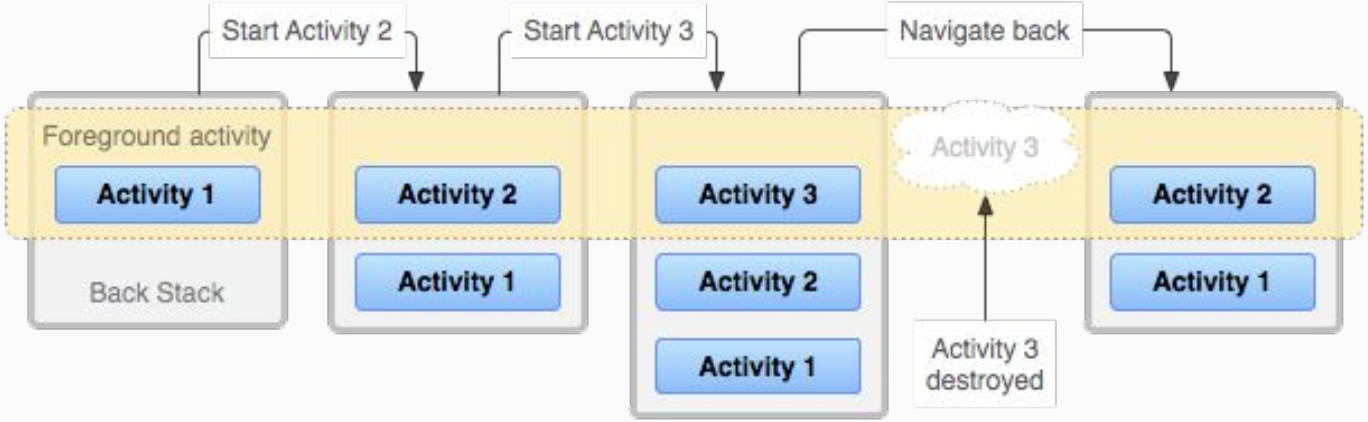


This can be avoided…

- *use Flags in the calling Intent… more on this later*

Tasks and BackStack

- Navigating back on the root activity causes the app to terminate (Android 11 and previous) or brings the current **task** in background (Android 12 and later).
  - Navigating through activities requires Intents (we'll see them).



Tasks and BackStack

What is a **Task**?

It is a cohesive unit that contains a storyline (a **BackStack**) and can be in the foreground (if the top Activity is running) or in the background (if all Activities are stopped).

A task in the background can be seen in the “Recent Activities” UI.

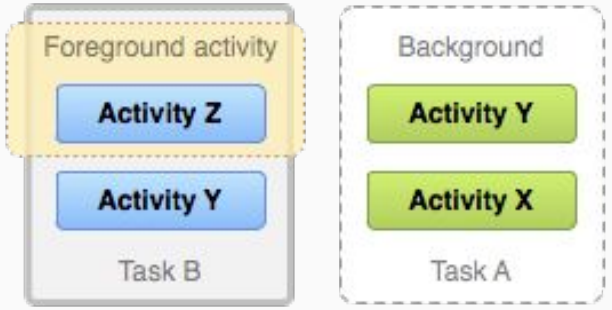
An app can be made of multiple tasks (thus, multiple BackStacks).

Tasks and BackStack

Launching an app from the Home screen, by default, lands always in the same Task.

For each activity we can choose if starting a new task and customize several parameters:

- In the manifest
- In the flags in the launching Intent



More details on
https://developer.android.com/guide/components/activities/tasks-and-back-stack#ManagingTasks
and in the next lectures…

The Main Thread

Normally, each application runs on its own Linux process, called the ***Main Thread***

“An unusual and fundamental feature of Android is that an application process' lifetime isn't directly controlled by the application itself. Instead, it is determined by the system through a combination of the parts of the application that the system knows are running, how important these things are to the user, and how much overall memory is available in the system.”

Activities are running and keeping alive the Main Thread, but other components may influence it so we should keep an eye out.

Contexts

Class **Activity** or **AppCompatActivity** (like others) implement the abstract class **Context**.

Context is a handle to the system, providing environment references and used for, e.g. :

- Loading a resource.
- Launching a new activity.
- Creating views.
- Obtaining system services.

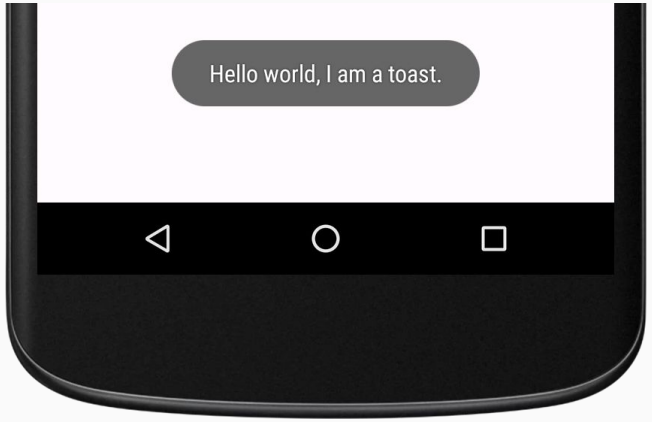
etc…

Making a Toast

- Tiny messages over the Activity (takes the Context as an input)
- Used to signal to the user confirmation, little errors
- You can control the duration of the Toast

As simple as:

```
Toast.makeText(this, “Hello world, I am a toast.”, Toast.LENGTH_SHORT).show()
```



Questions?

federico.montori2@unibo.it