Android Activities

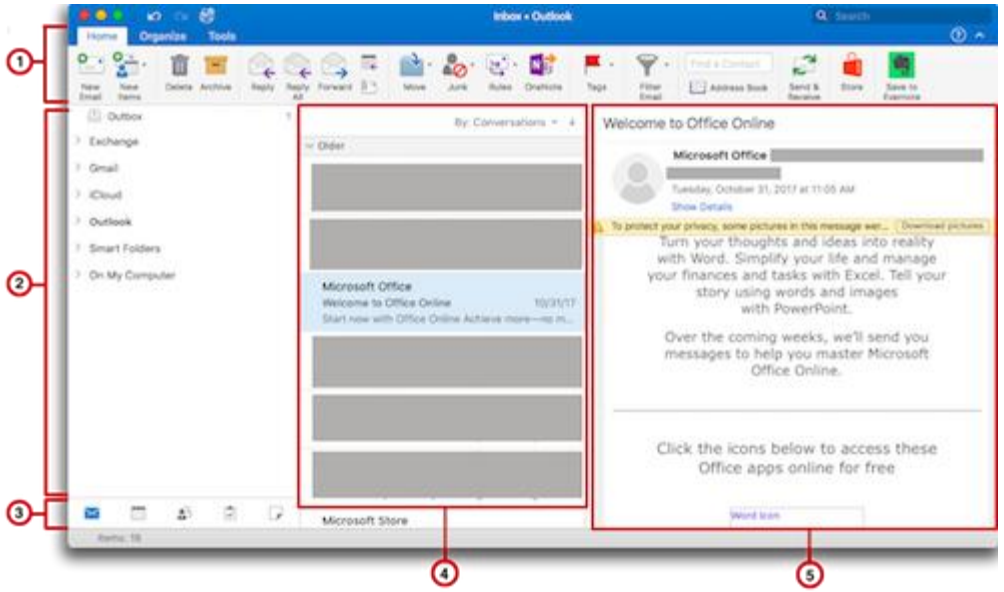
Hua Huang

Reading Materials

- Chapter 3,4

Desktop Program vs. Mobile Apps

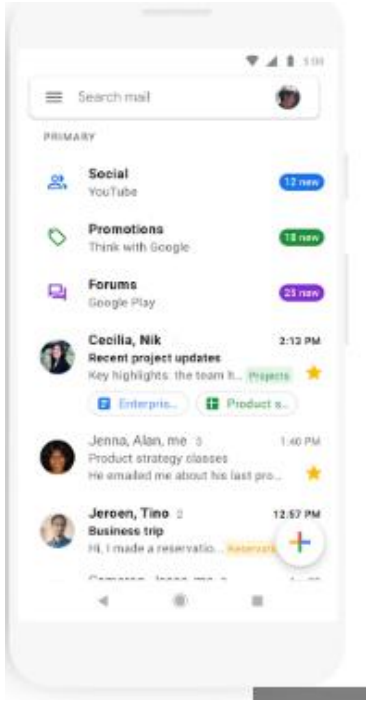
- A destkop program has a single entry point: main()



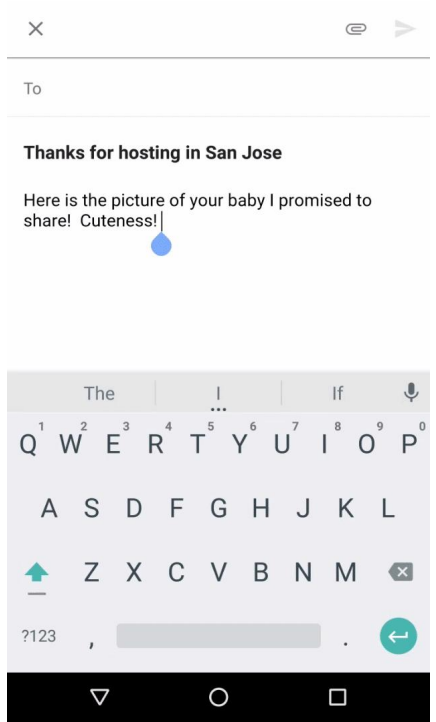
- A mobile app can start from different screens (non-determinstic)

Challenge: the starting point of an app is non deterministic

- e.g. open an email app from your home screen, you might see a list of emails.

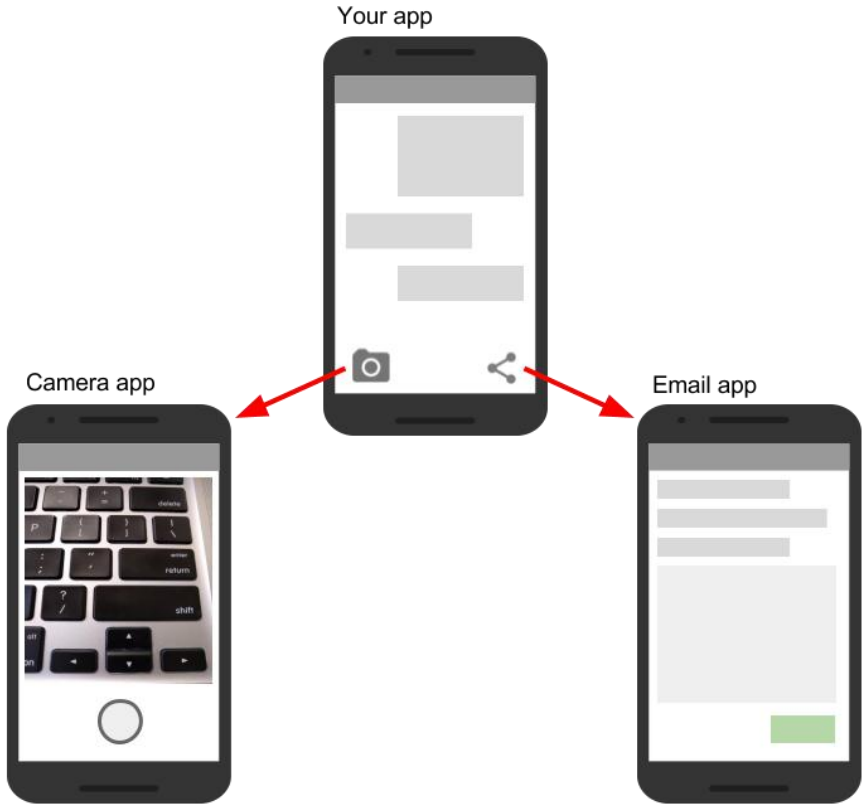


- By contrast, use a social media app that then launches email app, you might go directly to the email app's screen for composing an email.



Activities

- The Activity class is designed to facilitate this paradigm.
- When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole.
- In this way, the activity serves as the entry point for an app's interaction with the user.
- You implement an activity as a subclass of the Activity class.



Activity vs Android Application

- An activity provides the window in which the app draws its UI.
  - This window typically fills the screen, but may be smaller than the screen and float on top of other windows.
- Generally, one activity implements one screen in an app.
  - e.g., one of an app's activities may implement a Preferences screen, while another activity implements a Select Photo screen.

- Android Application
  - includes activities, services, intents,data... etc
  - the manifest file itemize them

Activities

- Most apps contain multiple screens, which means they comprise multiple activities.
- Typically, one activity the main activity, which is the first screen to appear when the user launches the app.
- Each activity can start another activity to perform different actions.
  - e.g., the main activity in a simple e-mail app may provide the screen that shows an e-mail inbox.
  - From there, the main activity might launch other activities that provide screens for tasks like writing e-mails and opening individual e-mails.

Activities

- Each activity is only loosely bound to the other activities in the app;
  - there are usually minimal dependencies among the activities in an app.
  - In fact, activities often start up activities belonging to other apps.
  - e.g., a browser app might launch the Share activity of a social-media app.

Inside “Hello World” AndroidManifest.xml

This file is written using xml namespace and tags and rules for android

package name

Android Version

Activity List

```
<?xml version="1.0"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.skeleton"
  android:versionCode="1"
  android:versionName="1.0">

  <application>
    <activity
      android:name="Now"
      android:label="Now">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

One activity (screen) designated LAUNCHER.
The app starts running here

Activity Cycles

Activity Lifecycle

- Activities keep evolving in the Apps.
  - Example: click to open the main activity. Click a button to open on child activity
  - What happens under the hood?
  - How to make an activity light up?
  - How to make it go away?
  - How to maintain the previous context?

Activity Lifecycle

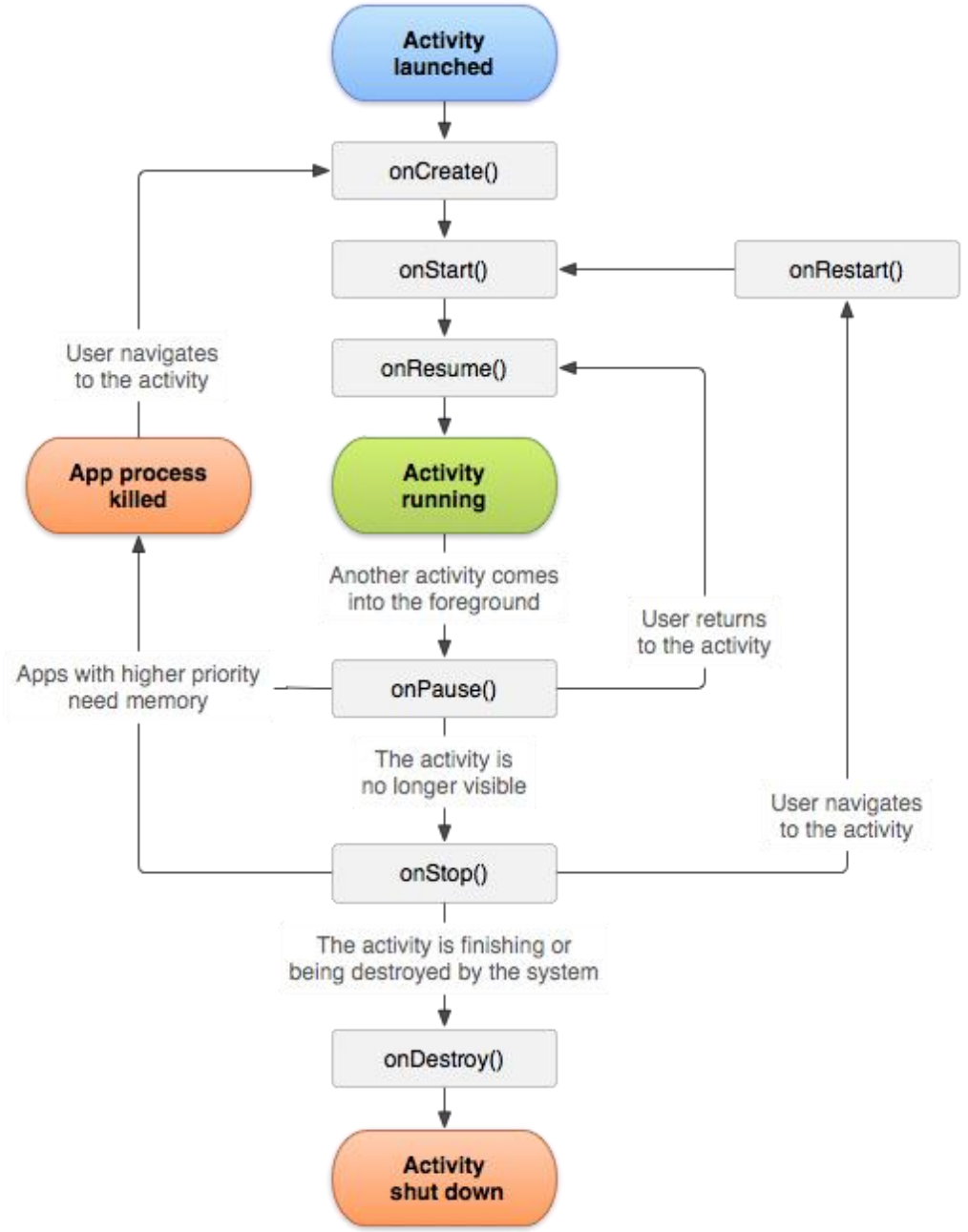
- What happens under the hood?
  - How to make old screen go away?
    - save states
    - stop sensors or GPS
    - release RAM
    - ...
  - How to make an activity light up?
    - draw the UIs
    - initiate senosrs

Activity

- The Activity class provides a number of callbacks to manage the transitions
- When the system is creating, stopping, or resuming an activity, we can define the the codes to run

A Simplified Activity Lifecycle

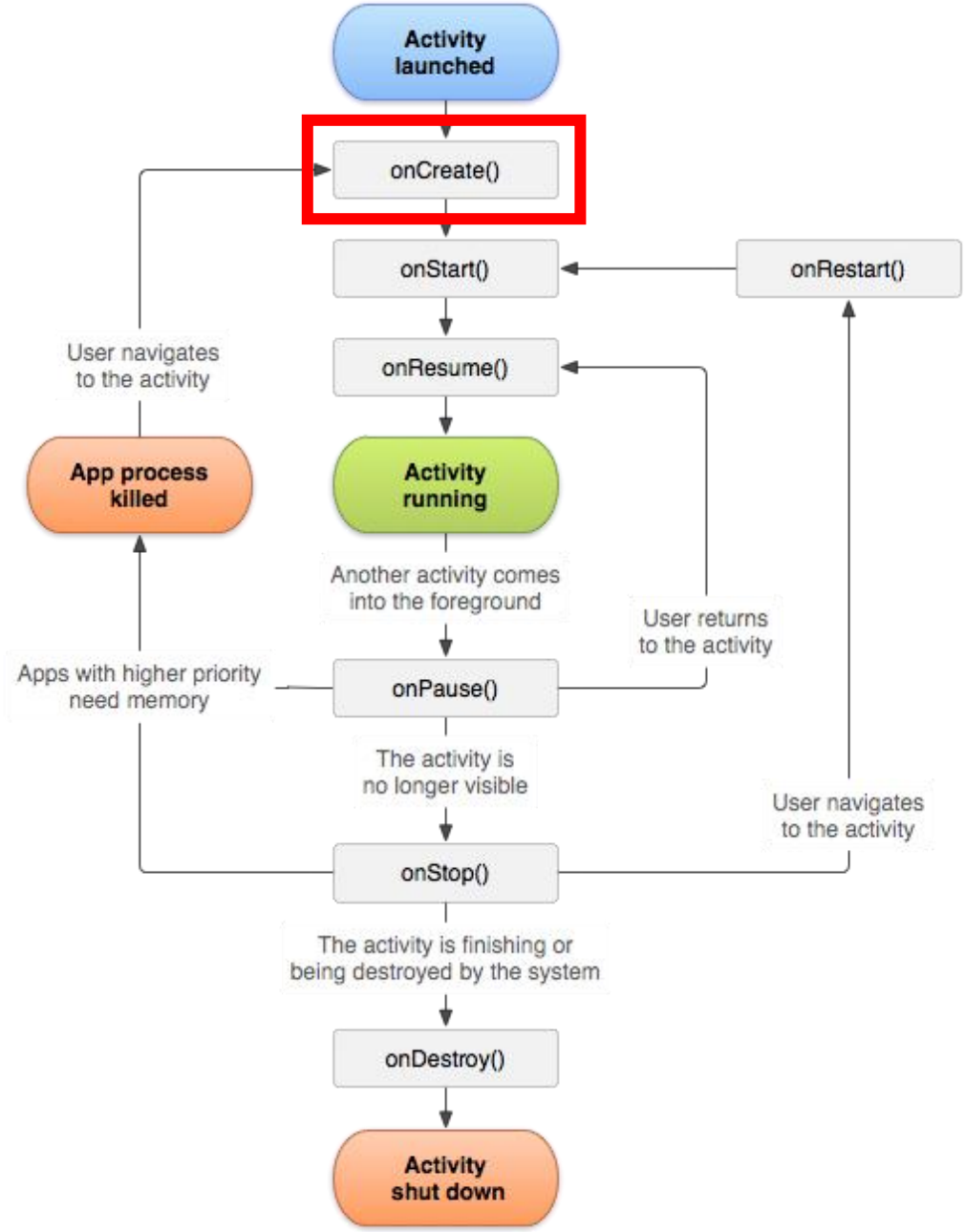
Think: Why are they necessary?



onCreate()

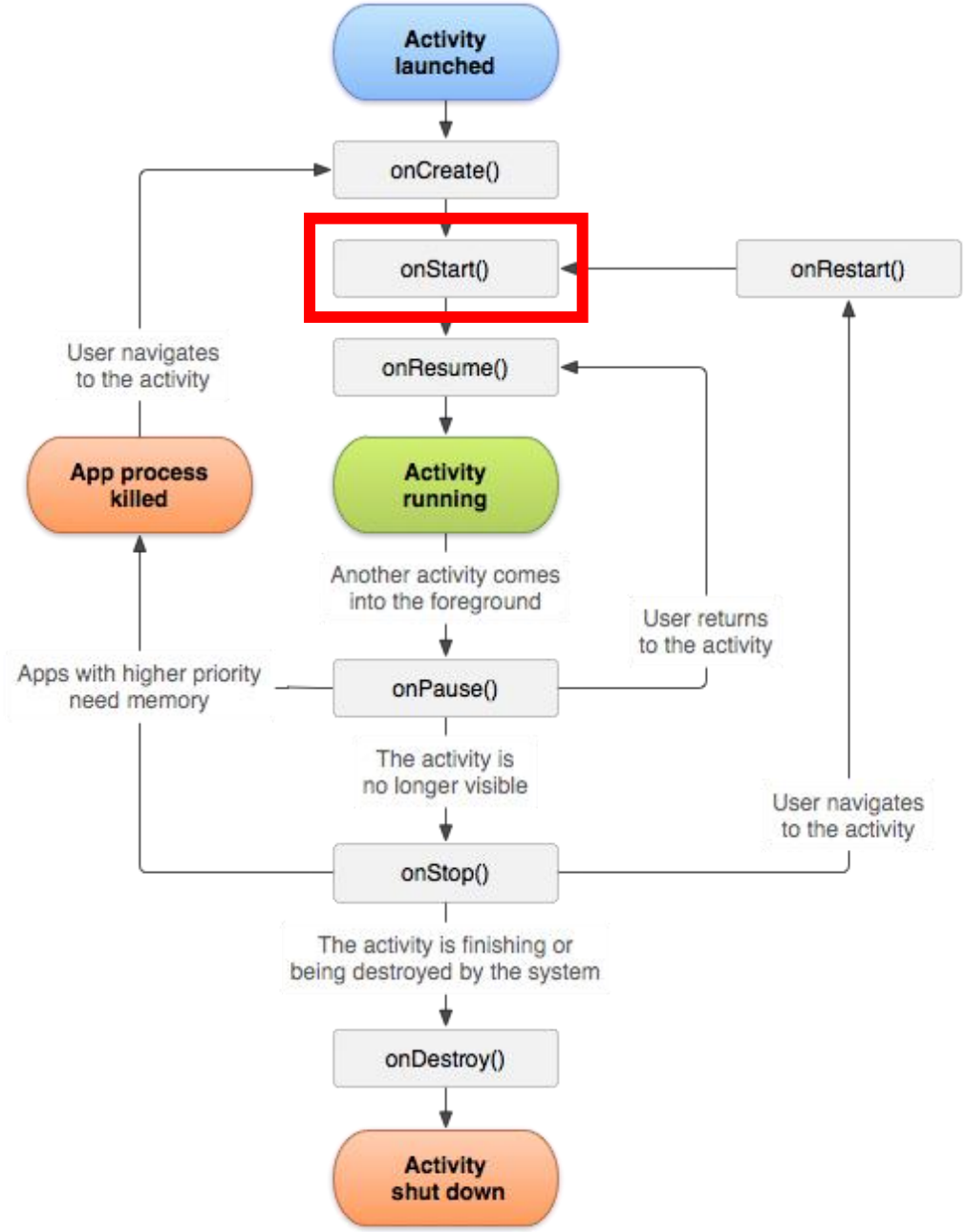
- You must implement this callback, which fires when the system first creates the activity.
- Perform basic application startup logic that should happen only once for the entire life of the activity.
  - E.g., bind data to lists, associate the activity with a ViewModel, and instantiate some class-scope variables

- Enters **Created** state after execution



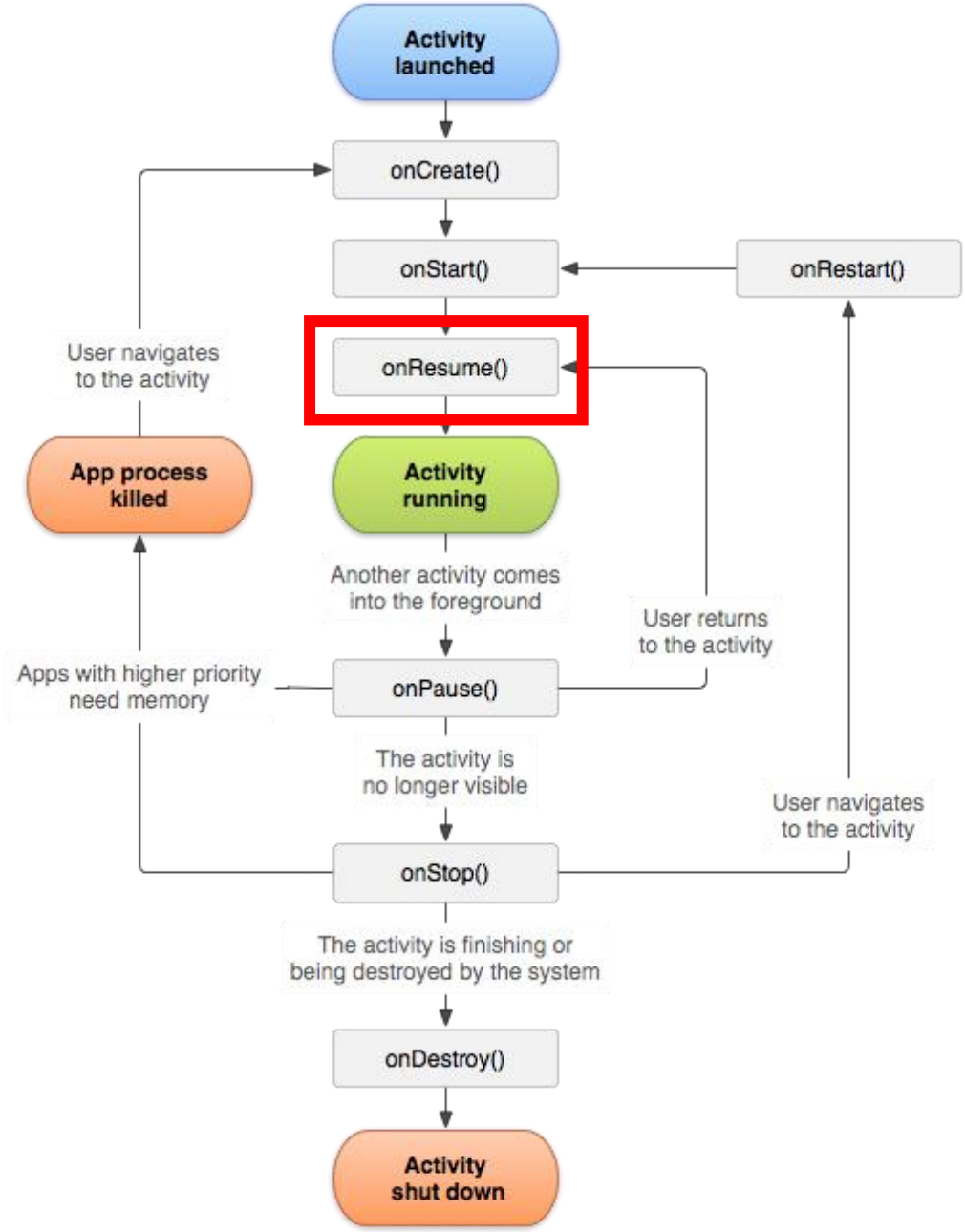
onStart()

- Once in **Created** state, automatically calls this function.
- Makes the activity visible to the user, as the app prepares for the activity to enter the foreground and become interactive.
- The onStart() method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state.
    - Once this callback finishes, the activity enters the **Started** state, and the system invokes the onResume() method.

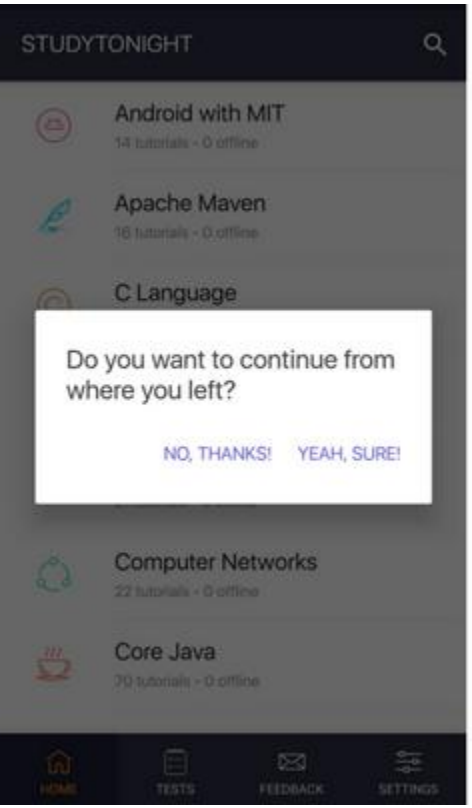


onResume()

- When the activity enters the **Started** state, it comes to the foreground, and then the system invokes the onResume() callback.
- Resumed is the state in which the app interacts with the user.
- The app stays in this state until something happens to take focus away from the app.
  - E.g., receiving a phone call, the user's navigating to another activity, or the device screen's turning off.
- When an interruptive event occurs, the activity enters the **Paused** state, and the system invokes the onPause() callback.
- If the activity returns to the Resumed state from the Paused state, the system once again calls onResume() method.



The Paused State



Activity State:

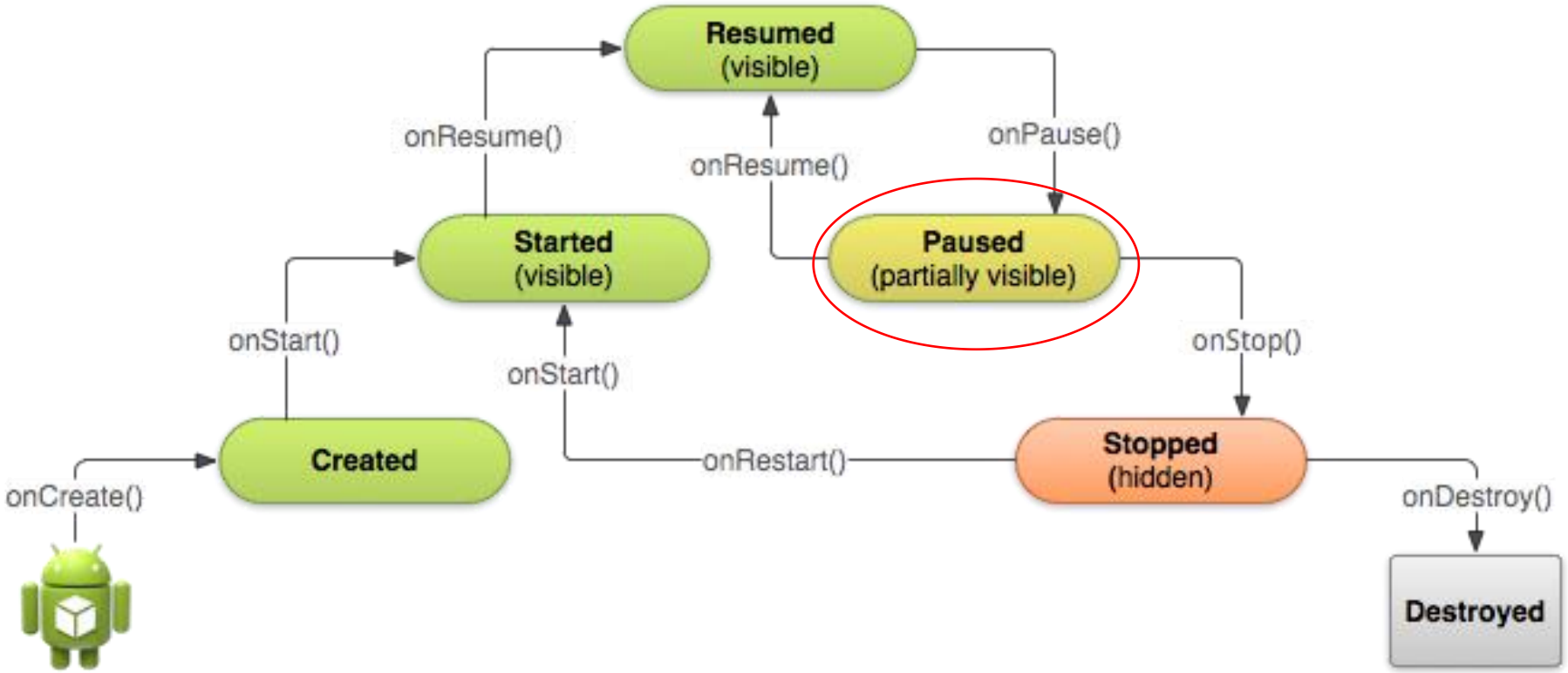
Paused

Process state is Background(lost focus).

And the likelihood of killing the app is More.

- The app stays in **Resumed** state until something happens to take focus away from the app.
- The system calls this method as the first indication that the user is leaving your activity
  - It does not always mean the activity is being destroyed
- It indicates that the activity is no longer in the foreground

The Paused State



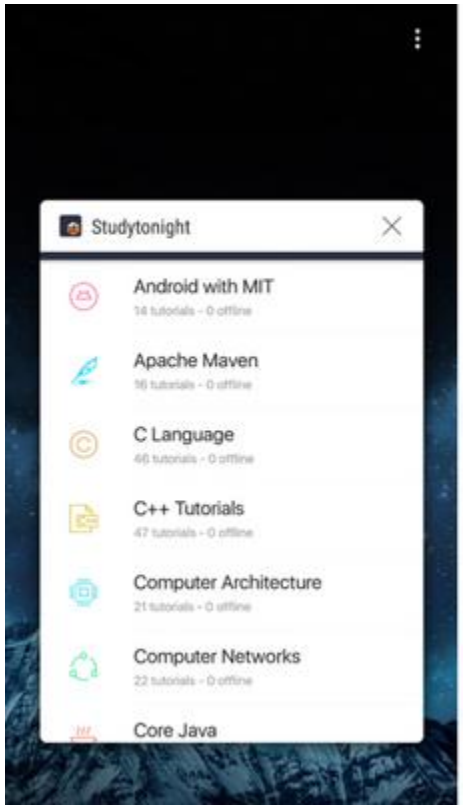
onPause()

- Scenarios that trigger onPause():
  - Some event interrupts app execution, as described in the onResume() section. This is the most common case.
  - A new, semi-transparent activity (such as a dialog) opens. As long as the activity is still partially visible but not in focus, it remains paused.
  - In Android 7.0 (API level 24) or higher, multiple apps run in multi-window mode. Because only one of the apps (windows) has focus at any time, the system pauses all of the other apps.

onPause()

- We can stop any functionality that does not need to run while the component is not in the foreground
  - E.g., stop the camera preview
  - release system resources, handles to sensors (like GPS), or any resources that may affect battery life while your activity is paused and the user does not need them

Stopped State



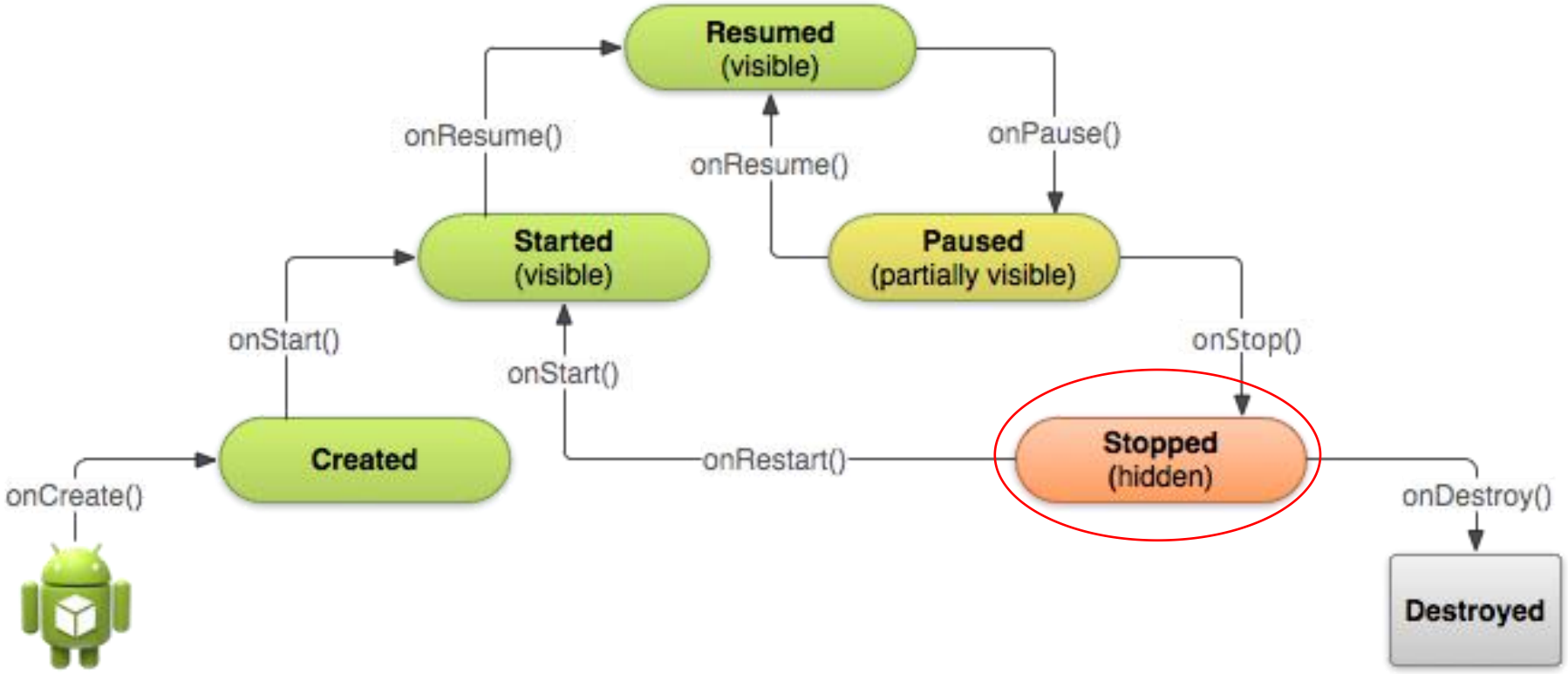
Activity State:

Stopped

Process state is Background(not visible).

And the likelihood of killing the app is Most.

- When a new Activity is started on top of the current one or when a user hits the Home key, the activity is brought to Stopped state.
- The activity in this state is invisible, but it is not destroyed.
- Android Runtime may kill such an Activity in case of resource crunch.


Resumed
(visible)
onResume()
onResume()
onPause()
Started
(visible)
Paused
(partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped
(hidden)
onCreate()
onDestroy()
Destroyed

onStop()

- When your activity is no longer visible to the user, it has entered the Stopped state, and the system invokes the onStop() callback.
- Should release or adjust resources that are not needed while the app is not visible to the user here.
  - E.g., pause animations
  - switch from fine-grained to coarse-grained location updates
- Also use onStop() to perform relatively CPU-intensive shutdown operations.
  - E.g., save information to a database

Question

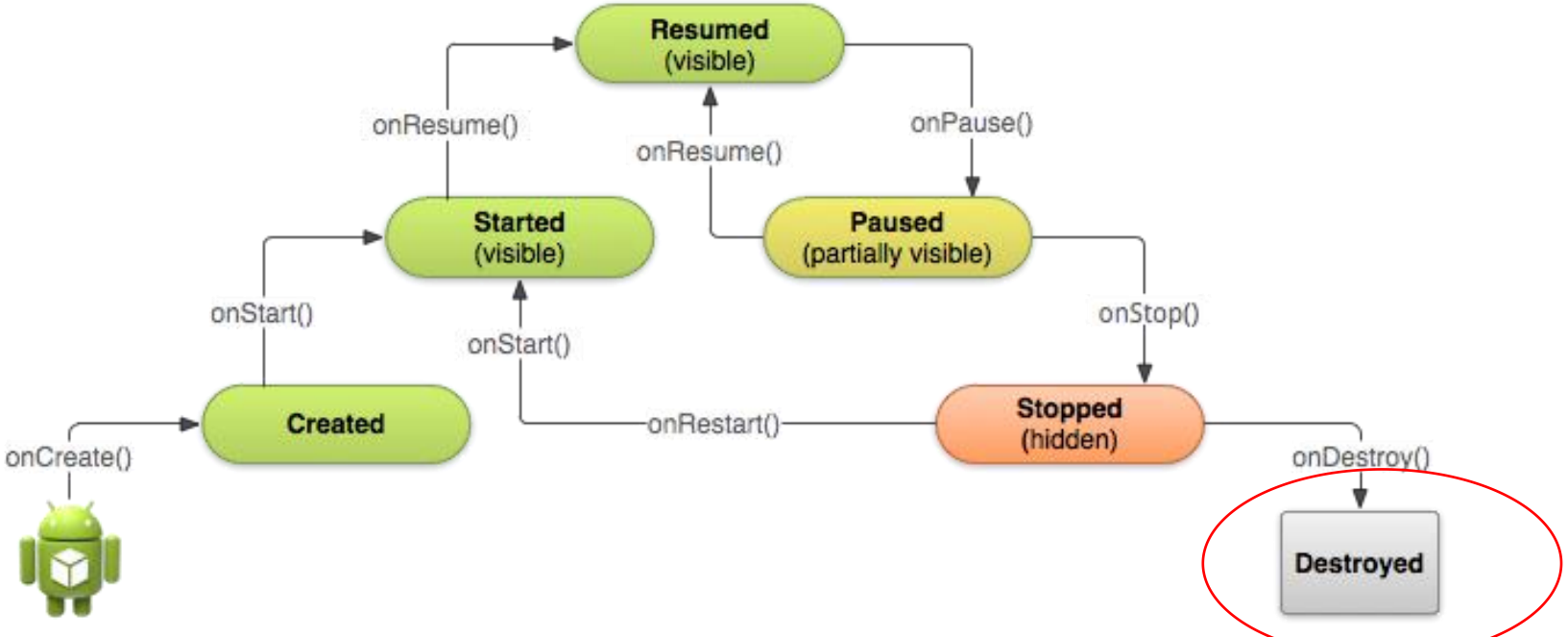
- Should we release and adjust UI components in onPause() instead? What are the pros and cons?
- Answer: a Paused activity may still be fully visible if in multi-window mode. As such, you should consider using onStop() instead of onPause() to fully release

Question

- Should we perform relatively CPU-intensive shutdown operations in onPause() instead?
- Answer: No, because on onPause() is supposed to be brief.

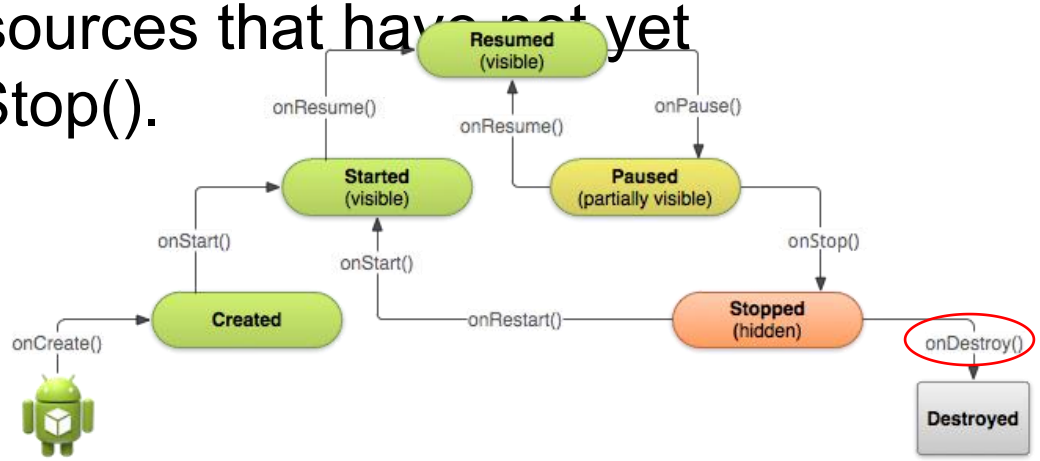
The Destroyed State

- When a user hits a Back key or Android Runtime decides to reclaim the memory allocated to an Activity
- The Activity is out of the memory and it is invisible to the user.



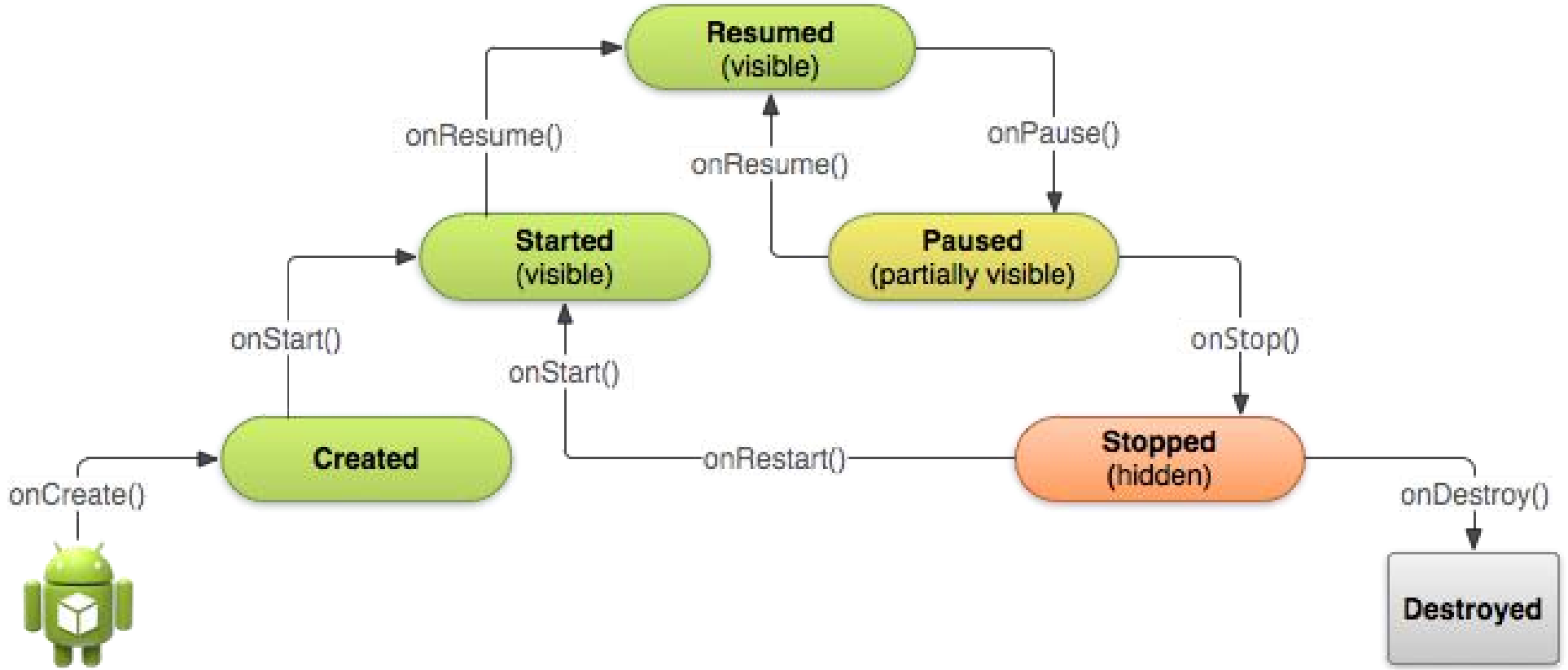
onDestroy()

- onDestroy() is called before the activity is destroyed. The function is called either because:
  - The activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity), or
  - The system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)
- The onDestroy() callback should release all resources that have not yet been released by earlier callbacks such as onStop().



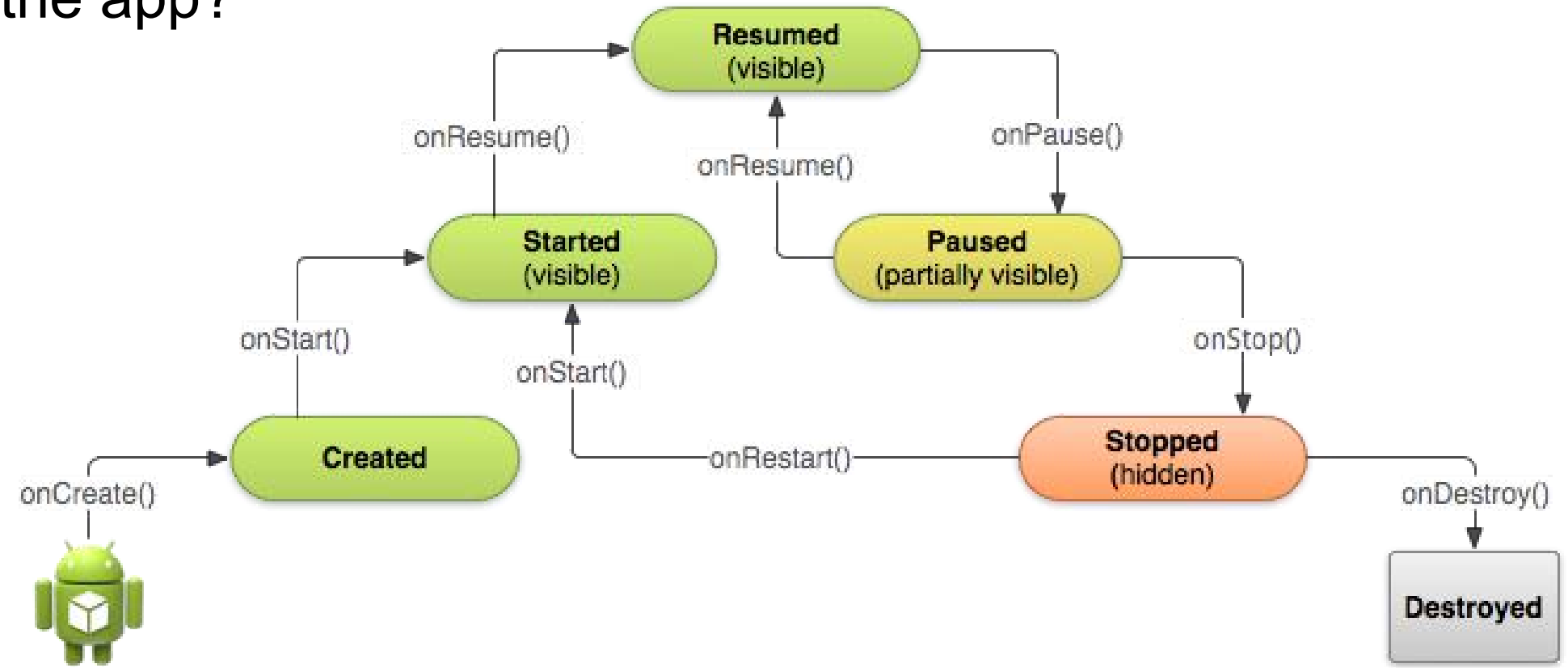
Discussions

- What functions are called when open the app?



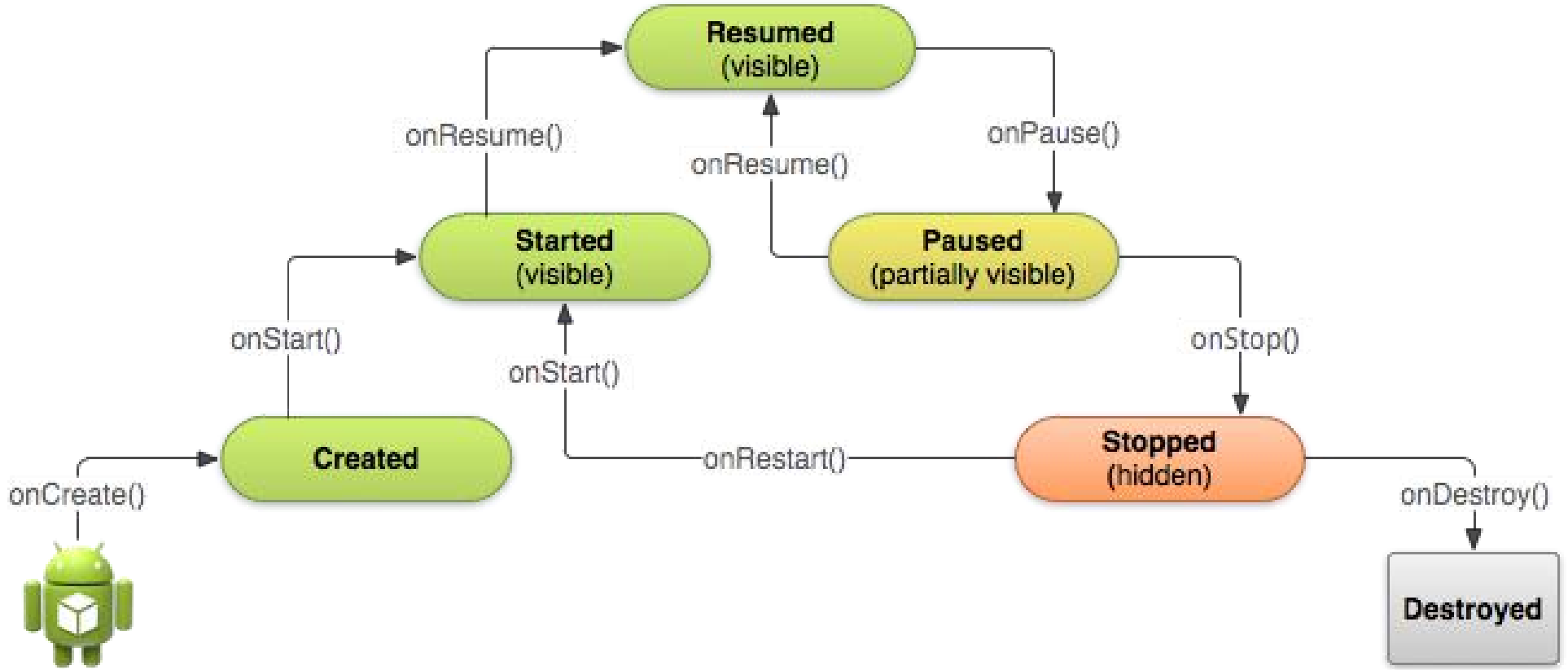
Discussions

- What functions are called when back button pressed and exit the app?



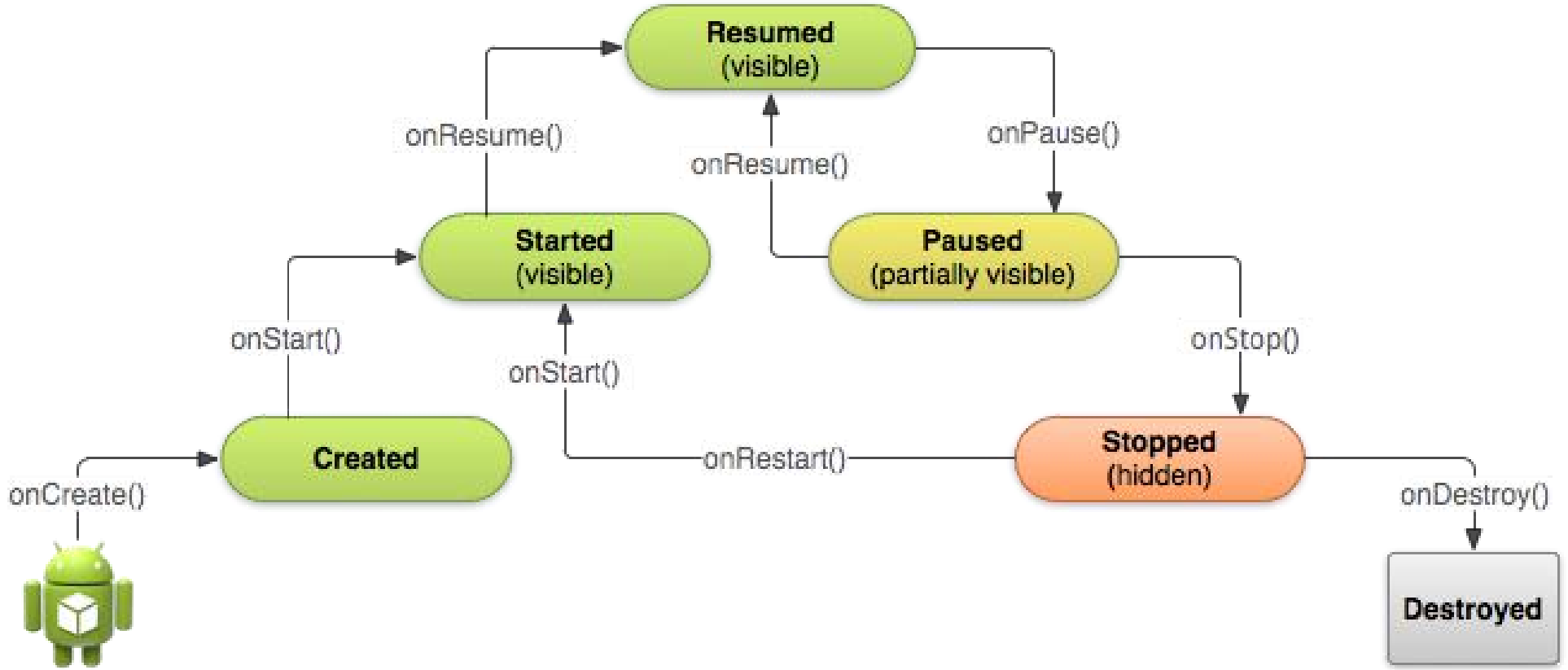
Discussions

- What functions are called when home button pressed?



Discussions

- What functions are called when open app from recent task list?



- Good implementation of the lifecycle callbacks can help your app avoid the following:
  - Crashing if the user receives a phone call or switches to another app while using your app.
  - Consuming valuable system resources when the user is not actively using it.
  - Losing the user's progress if they leave your app and return to it at a later time.
  - Crashing or losing the user's progress when the screen rotates between landscape and portrait orientation.

Take Away Message

- Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.
  - For example, if you're building a streaming video player, you might pause the video and terminate the network connection when the user switches to another app.
- Each callback lets you perform specific work that's appropriate to a given change of state.
- Doing the right work at the right time and handling transitions properly make your app more robust and performant.