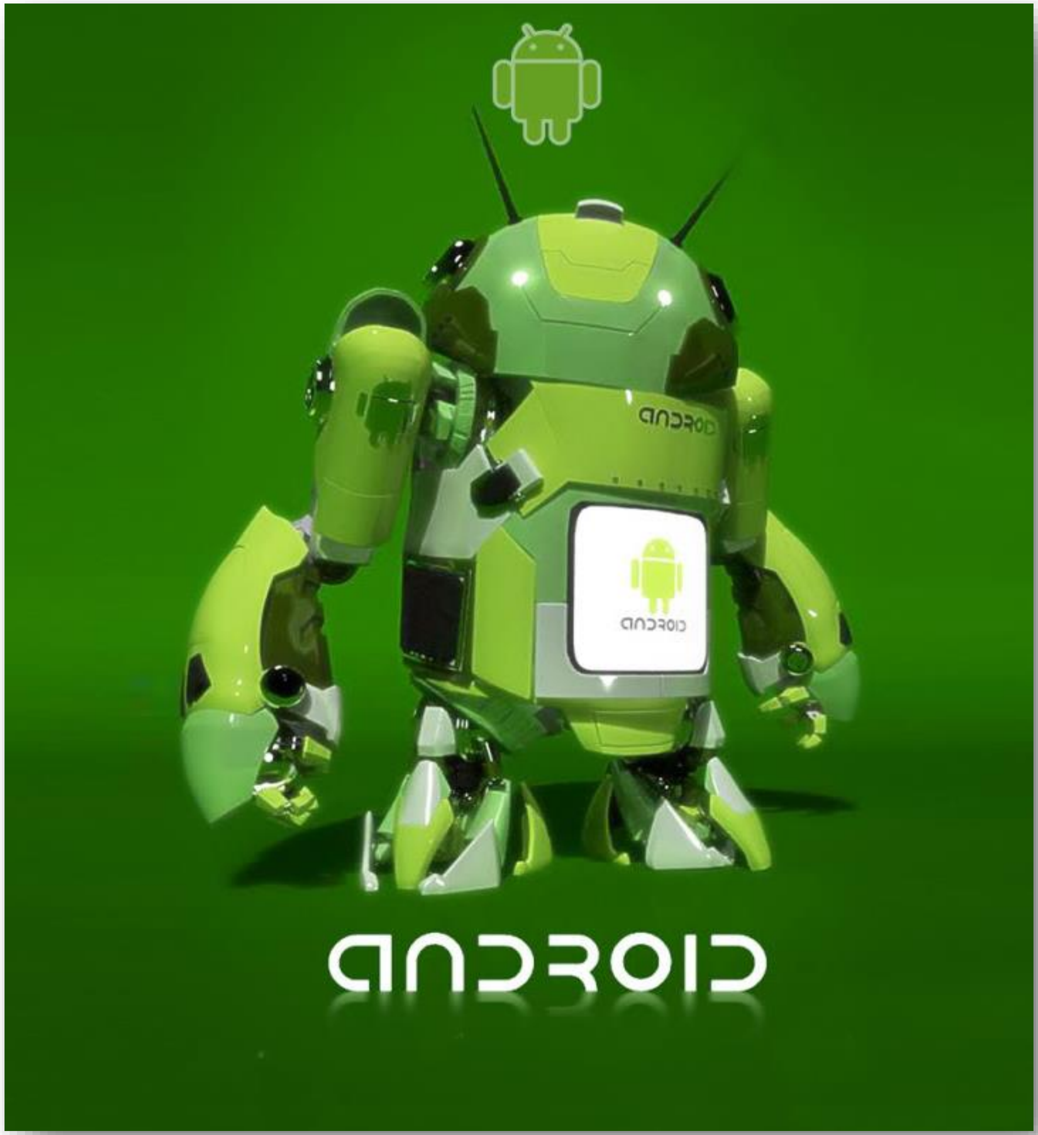
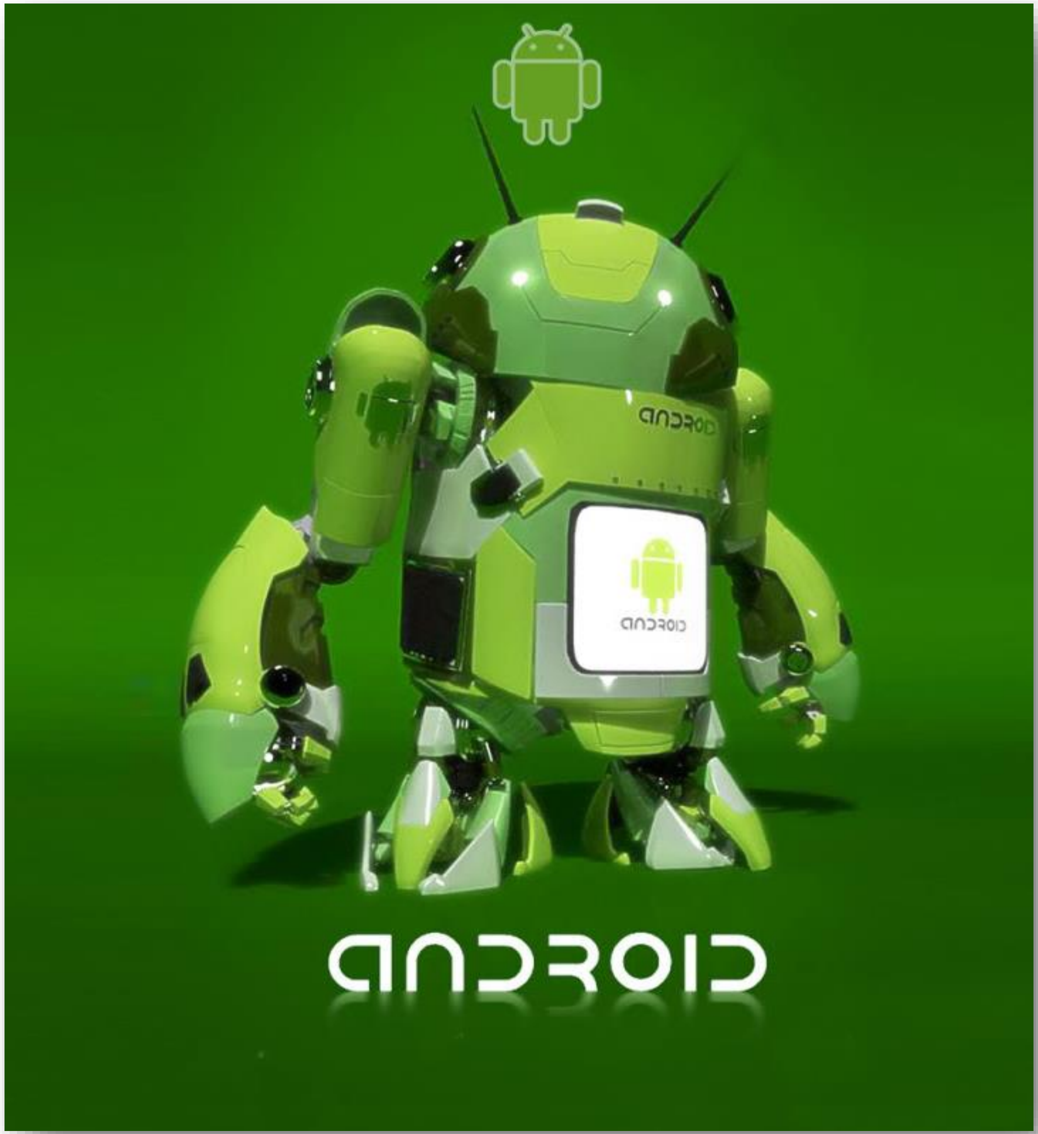
Week 3

FIT2081 – Mobile Application Development
Stephen Huxford
MONASH University
FIT, Clayton

- Mobile OSs
- Activity Lifecycles
- Activity Instance State
- Persistent Data
- Reference Summary
- SimpleActivityLifecycle Demo App
- API Documentation- How to Use



- Mobile OSs



Mobile OSs are Different

- "Regardless of the fanfare about how much memory and computing power resides in the mobile devices of today compared to the desktop systems of yesterday, it is important to keep in mind that these devices are still considered to be "resource constrained" by the standards of modern desktop and laptop based systems, particularly in terms of memory.
- As such, a key responsibility of the Android system is to ensure that these limited resources are managed effectively and that both the operating system and the applications running on it remain responsive to the user at all times.
- In order to achieve this, Android is given full control over the lifecycle and state of both the processes in which the applications run, and the individual components that comprise those applications.
  - [the Android app developer is not!]
  - See here for a heated exchange as developers struggle with aspects of this new OS paradigm and here for a calm analysis
- An important factor in developing Android applications, therefore, is to gain an understanding of both the application and activity lifecycle management models of Android, and the ways in which an application [i.e. the application's developer] can react to the state changes that are likely to be imposed upon it during its execution lifetime."

Source: Smyth, Neil. Android Studio Development Essentials - Android 7 Edition

Android is a Killer!

- "Each running Android application is viewed by the operating system as a separate process.
- If the system identifies that resources on the device are reaching capacity it will take steps to terminate processes to free up memory.
- When making a determination as to which process to terminate in order to free up memory, the system takes into consideration both the priority and state of all currently running processes, combining these factors to create what is referred to by Google as an importance hierarchy.
- Processes are then terminated starting with the lowest priority and working up the hierarchy until sufficient resources have been liberated for the system to function.
- Processes host applications and applications are made up of components. Within an Android system, the current state of a process is defined by the highest-ranking active component within the application that it hosts."

Source: Smyth, Neil. Android Studio Development Essentials - Android 7 Edition

Who Gets Killed?

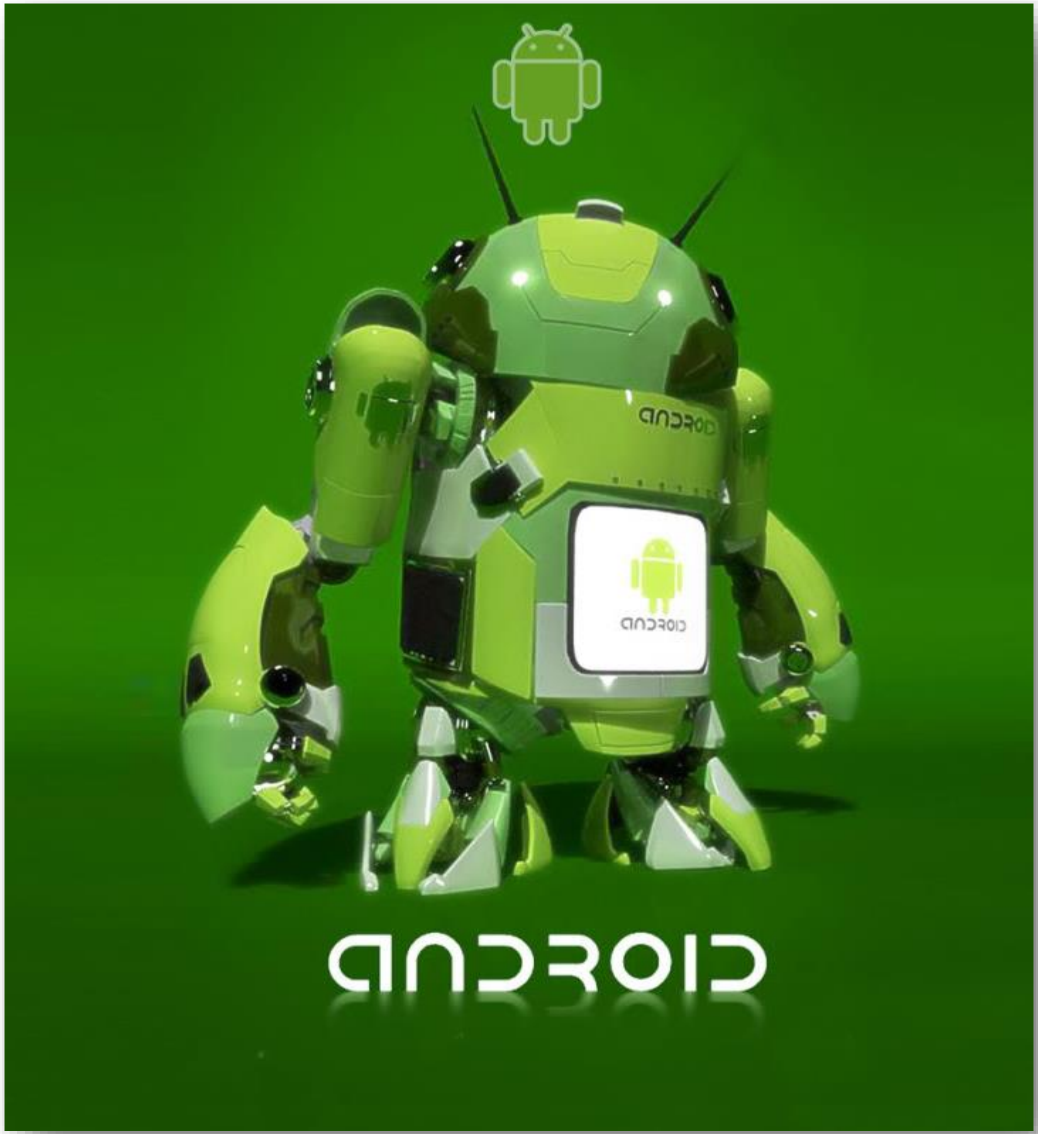
* e.g. an Activity can start a service in another thread running in a separate process

** We are basically ignoring Services here which can also play a significant role in ranking processes

- So Android will terminate/kill a process if it needs resources
  - Remember each process contains an ART VM running an app which consists of components (e.g. activities, services etc.)
- It will rank processes by their highest ranking active component
  - Processes will be terminated/killed if necessary, lowest ranking first
  - Processes can depend on each other *
    - A process can never be ranked lower than the process it's serving
- To keep things simple** we will now concentrate on the visible components of an app i.e. Activities
  - How are Activities ranked? By 3 of their 4 possible Lifecycle states!
    - 1. Activity is in the foreground (user is interacting/can interact with it)
    - 2. Activity's UI is partially hidden***
    - 3. Activity's UI is fully hidden
  - Only 1 process contains a foreground Activity and it cannot be killed
  - All other processes can be killed beginning with those that only have Activities with fully hidden Uis
    - Killing a process destroys all of its app's components of course

*** This is rare and does not include menu, toast, snackbar or dialog overlays

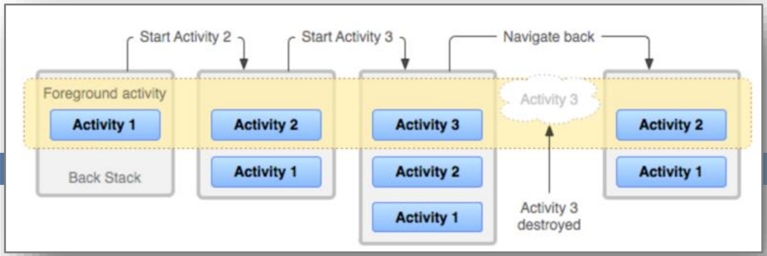
- Activity Lifecycles



How Do Activities Move from State-to-State?

- So the state of a process's Activities determine its vulnerability to an OS kill
- Activities have 4 possible Lifecycle states
  - Foreground
  - Partially hidden
  - Fully hidden
  - Destroyed
- They transit between these states as a result of:
  - User interaction
    - This involves the concept of Tasks and Activity Back Stacks
      - See next slide
  - OS process kills
    - When a process is killed all its app's components are destroyed in including its Activities

Back Stacks



- Stacks
  - “A task is a collection of activities that users interact with when performing a certain job. The activities are arranged in a stack (the back stack), in the order in which each activity is opened.
  - “When the current activity starts another, the new activity is pushed on the top of the stack and takes focus. The previous activity remains in the stack, but is stopped. When an activity stops, the system retains the current state of its user interface.
  - When the user presses the Back button, the current activity is popped from the top of the stack (the activity is destroyed) and the previous activity resumes (the previous state of its UI is restored). Activities in the stack are never rearranged, only pushed and popped from the stack—pushed onto the stack when started by the current activity and popped off when the user leaves it using the Back button.”
  - If the user continues to press Back, then each activity in the stack is popped off to reveal the previous one, until the user returns to the Home screen (or to whichever activity was running when the task began). When all activities are removed from the stack, the task no longer exists.
  - A task is a cohesive unit that can move to the "background" when users begin a new task or go[es] to the Home screen, via the Home button. While in the background, all the activities in the task are stopped, but the back stack for the task remains intact—the task has simply lost focus while another task takes [its] place […]. A task can then return to the "foreground" so users can pick up where they left off.”

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

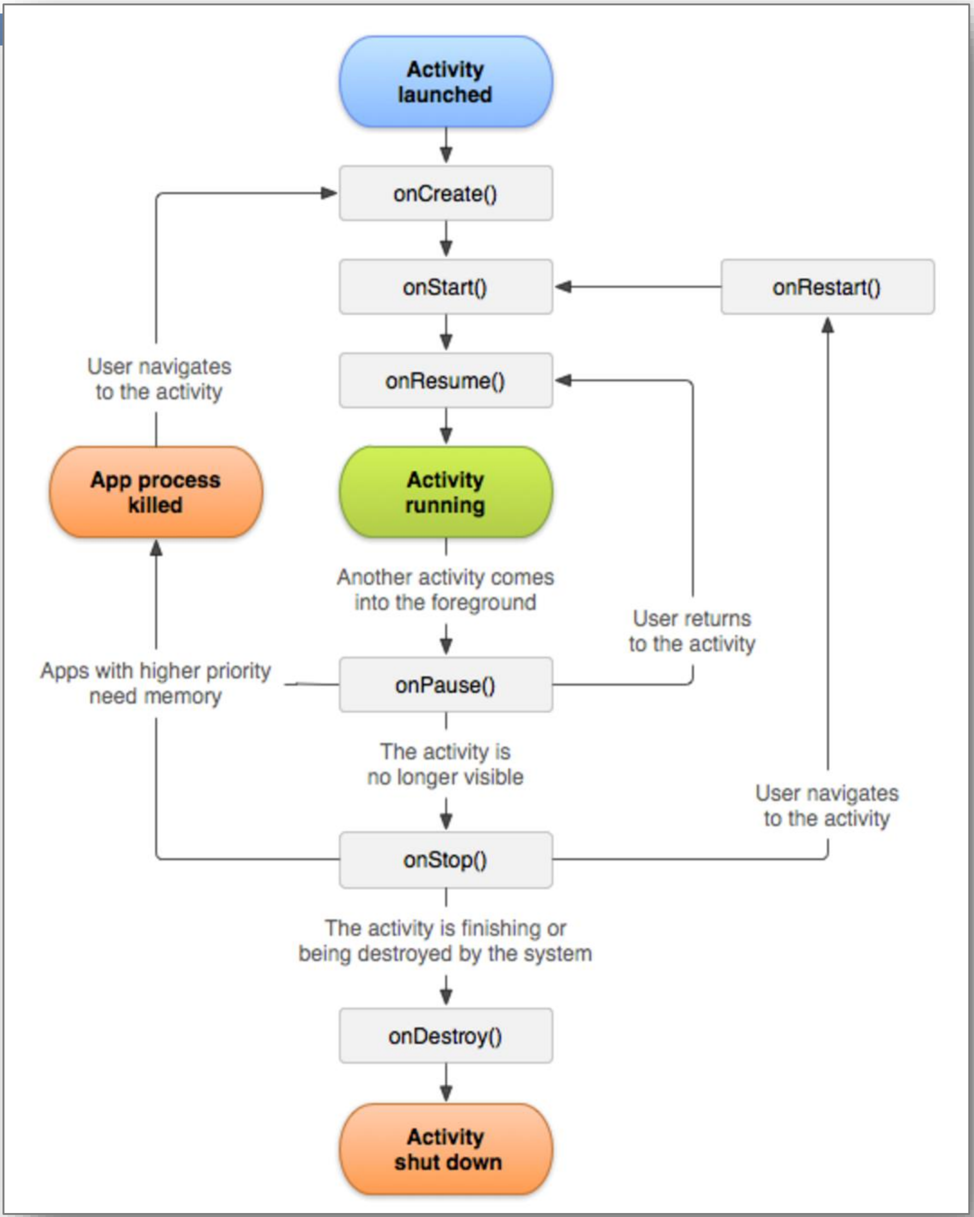
There More to It

- Complications
  - A task back stack can include Activities from several different apps
    - Remember an Activity can intent an Activity in another app
  - If the user performs an action that causes code to execute an Activity’s finish() method this is entirely equivalent to the user hitting the back button (i.e. its Activity is destroyed)
  - There can be multiple task back stacks in the background at once (if other tasks are started before the current task is finished)
    - This can even include multiple instances of the same Activity in different tasks
  - Putting your phone to sleep has the same effect as pushing a blank activity onto the current task back stack
    - Waking you phone pops the blank Activity
  - Fragments (more later on these)
    - These UI sub-activities have their own lifecycle albeit slaved to their parent Activity’s life cycle
    - Fragment transactions (such as display fragment or swap fragments) can be pushed onto the back stack of a task making them reversible

Activity Lifecycle

To keep things clear in your mind say Processes are killed by the OS but Activities are Destroyed by the OS or by User actions.

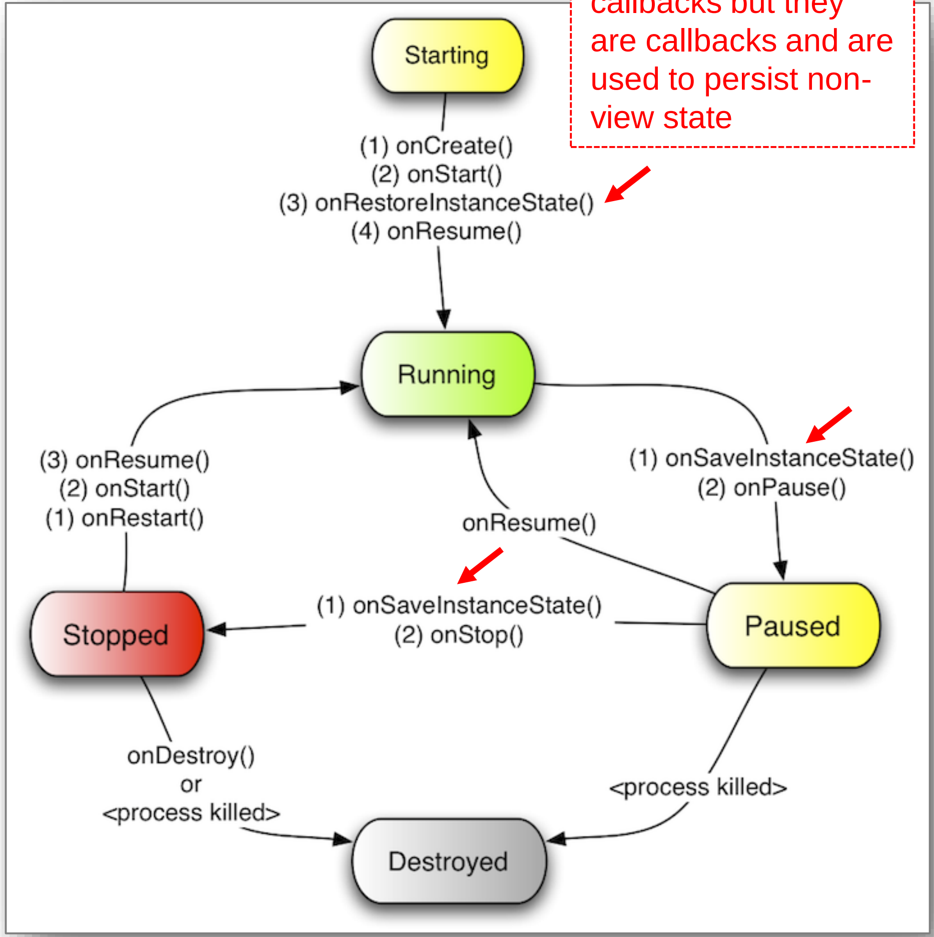
- Activity States
  - foreground, partially hidden, fully hidden, destroyed
- Lifecycle Callbacks
  - onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy()
- Lifecycle Loops
  - Partially hidden
    - Navigate back before destroyed (recents, home)
  - Fully hidden
    - Navigate back before destroyed (recents, home)
  - Process Destroyed
    - App relaunched
- “Back” destroys an Activity!


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

MONASH University

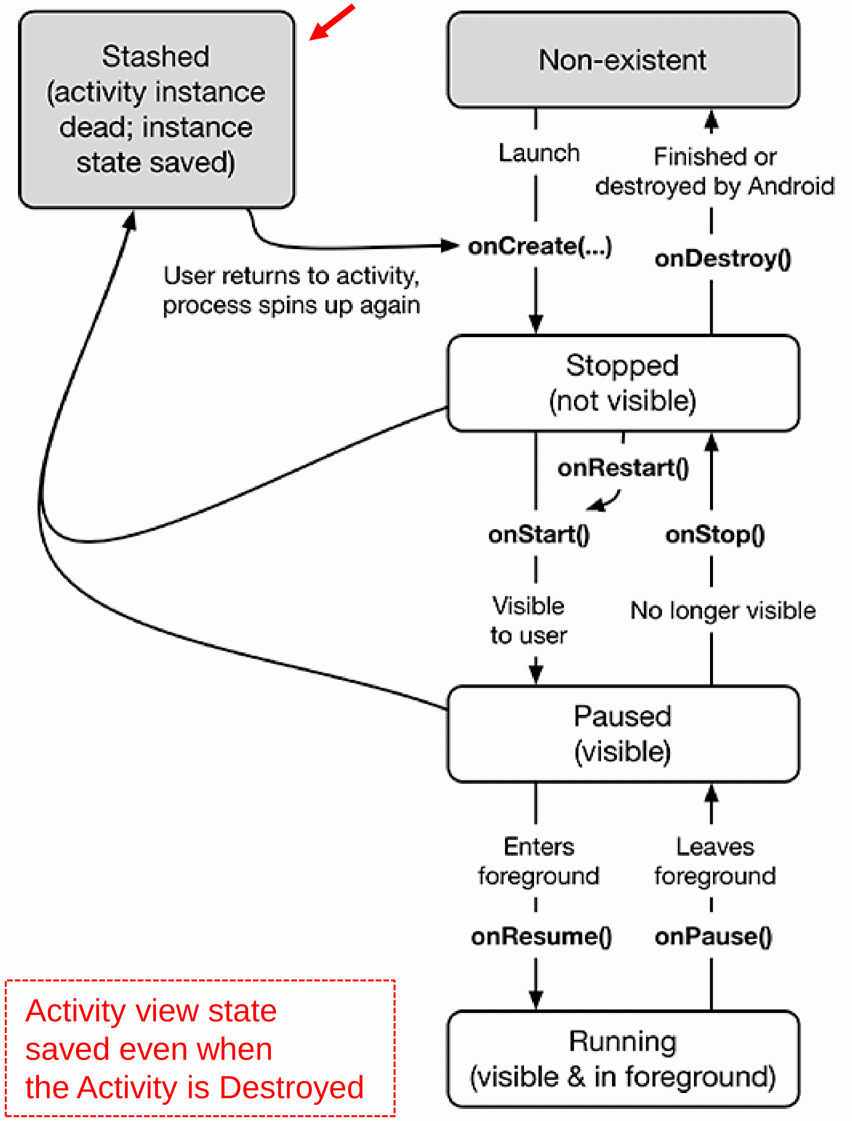
Activity Lifecycle - More

not lifecycle callbacks but they are callbacks and are used to persist non-view state



Source: http://ofps.oreilly.com/titles/9781449390501/Main_Building_Blocks.html

Source: Android Programming – The Big Nerd Ranch Guide (Phillips, B. et al)



Activity view state saved even when the Activity is Destroyed

Lifecycle Callbacks

- Out of Control?
  - User interaction and OS process kills drag our Activities around their lifecycle without our say so
    - As coders are we doomed to watch as helpless bystanders
- Not really!
  - “As a user navigates through, out of, and back to your app, the Activity instances in your app transition through different states in their lifecycle. The Activity class provides a number of callbacks that allow the activity to know that a state has changed: that the system is creating, stopping, resuming, or destroying an activity.”
    - Callbacks are methods Android calls but we can override and code our responses to an Activity’s lifecycle state changes

Source: https://developer.android.com/guide/components/activities/activity-lifecycle.html

Lifecycle Callbacks

- "Within the lifecycle callback methods, you can declare how your activity behaves when the user leaves and re-enters the activity.

Source: https://developer.android.com/guide/components/activities/activity-lifecycle.html

  - 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. When the user returns, you can reconnect to the network and allow the user to resume the video from the same spot.
  - In other words, each callback allows you to 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.
  - For example, good implementation of the lifecycle callbacks can help ensure that your app avoids:
    - 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."

Lifecycle Callbacks

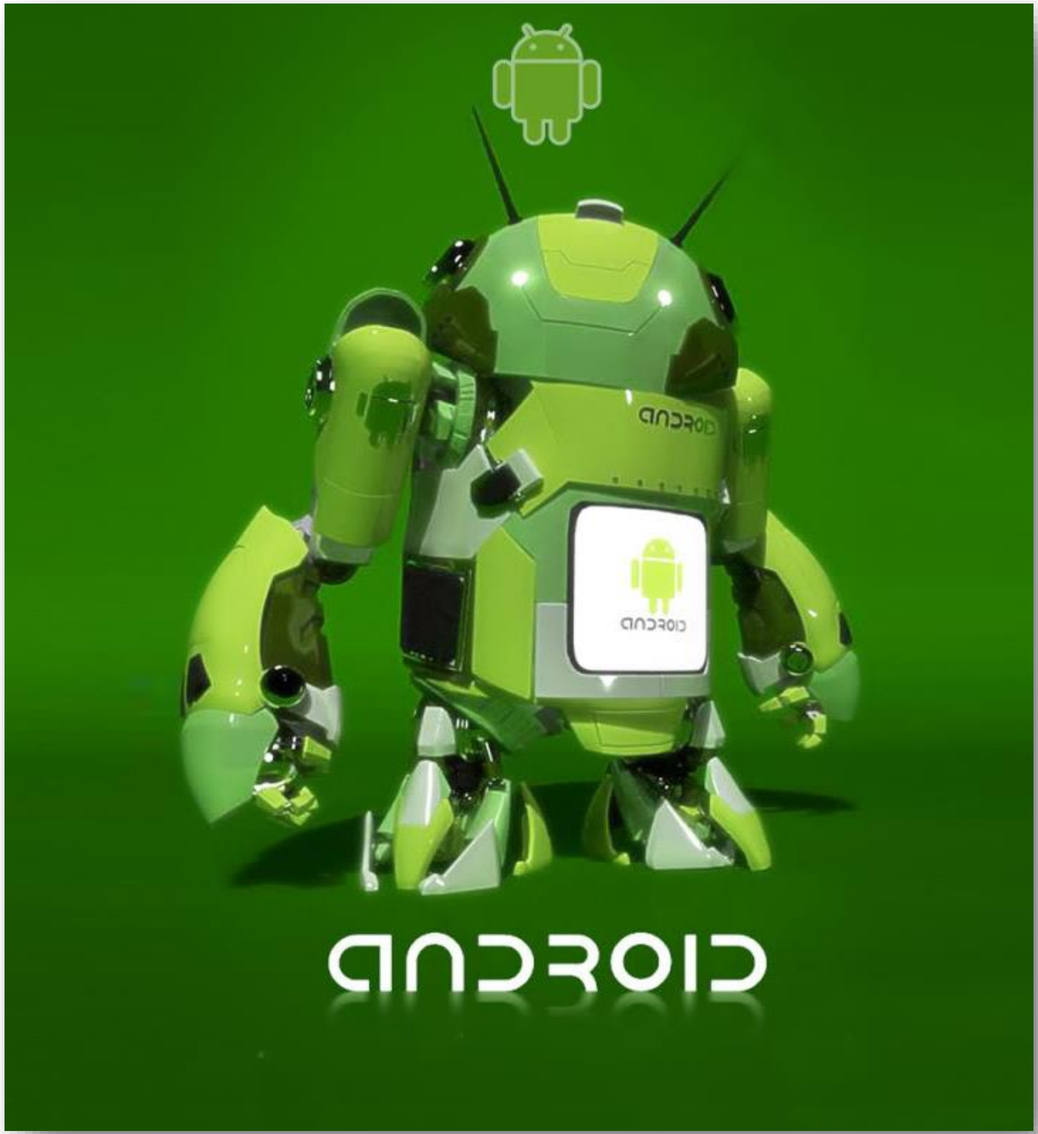
- Call super in all Lifecycle callbacks
  - Always begin lifecycle callback overrides with a call to their super
    - e.g. onCreate(…) override should begin with super.onCreate(…);
  - The supers perform a lot of standard processing required by any Activity
    - Our code just adds customised processing specific to our particular Activity

More Activity Callbacks

- onSaveInstanceState(…), onRestoreInstanceState(…)
  - Lifecycle callbacks are guaranteed to execute at the appropriate transitions between Activity states
  - These are 2 other important Activity callbacks that play a role in saving the state of an Activity (we will deal with this next)
    - These 2 callbacks will execute during specific transitions but only if certain conditions are met
    - As with the Lifecycle callbacks it's important to call the super of these callbacks
      - It turns out Android Auto-saves and restores a lot (but not all) of an Activity's instance state in these methods

- Activity Instance State

Do not confuse an Activity's Lifecycle State (foreground, partially hidden, fully hidden, destroyed) with its instance state i.e. the values of all its data (view and non-view).



Activity Data

- What data?
  - Activity Instance Data
    (i.e. data associated with a single use of the Activity)
    - Activity view data
      - i.e. the state of all Views in the Activity's layout
    - Activity non-view data
      - e.g. Activity instance variables
    - This involves writing to and reading from the device's volatile memory (this is a simplification)
  - Persistent data
    (i.e. data associated with multiple uses of the Activity separated by any amount of time i.e. it spans an Activity's instances)
    - Persistent across app uses implies across app destroys
      - Because uses are delineated by users indicating they are finished with an Activity which involves a destroy
    - This type of data can be made App private but not Activity private
      - i.e. its App data rather than Activity data but can often be used by just one Activity in which case it could be considered part of its state
    - This involves writing to and reading from the device's non-volatile memory

Activity Data & Persistence Across Events

- What events?
  - Activity is partially or fully hidden (+ No OS kill, + OS kill)
    - Activity is still in memory, no state data (view or non-view) is lost
    - Remember putting your phone to sleep is the same as completely hiding it with a blank Activity
    - Based on just this Activity its app's process is eligible for an OS kill. Why? What if this happens?
      - Surprisingly Android automatically saves the Activity's view-state away in a Bundle object (an object capable of storing key/value pairs) that is stored in an Activity record in the OS's own process since the Activity's process has been destroyed (how long for?)
      - Developers must code to save and restore non-view state in the same Bundle object
    - Persistent data remains in its last updated state (non-volatile memory)
  - Activity is destroyed using the back button
    - Or user interaction causes Activity code containing the Activity's finish() method to execute or user swipes Activity from recents
    - Android interprets such a user action to mean the Activity's view-state is no longer required so it destroys it along with the Activity
      - Developers should follow Android's lead and not save non-view state
    - Persistent data remains in its last updated state in non-volatile memory

Activity Data & Persistence Across Events

- What events?
  - Device configuration event such as rotating the device
    - Or pulling out a h/w keyboard etc
    - Surprisingly (or maybe not if you think about it) Android destroys and then relaunches the Activity (there is no app kill involved)
      - Think about, for instance, how different layouts may need to be inflated in such a rotation to see why Android starts from scratch again (e.g. standard and scientific calculator)
    - Android (correctly) DOES NOT interpret such a user action to mean the Activity's view-state is no longer required so it saves it in a Bundle object which it used on relaunch to restore the Activity's view-state
      - Developers must code to save and restore non-view state in the same Bundle object
      - Since the Activity's app process is not killed the Bundle object can be saved and retrieved from it (rather than to/from an Activity record in the OS's process)
  - App is uninstalled from the device
    - All of an app's data including any persistent data stored on the device's non-volatile memory is lost

Saving/Restoring Activity Instance State

- onSaveInstanceState(Bundle outState)
  - Not called if user indicates they are done with the Activity
  - Must call super to let Android save view-state
  - Must code to manually save non-view state in Bundle

not a syntax requirement

- onCreate(Bundle savedInstanceState)
  - Must call super to let Android restore view-state
  - Must code to manually restore previously saved non-view state from Bundle
    - Check if Bundle is non-null to see if there is an existing state that needs restoring else initialise the state with sensible default values

also a syntax requirement

- onRestoreInstanceState(Bundle savedInstanceState)
  - Only called if the Bundle is non-null
    - So no need to check for this condition
    - Unlike onCreate you cannot set default values here
  - Must call super to let Android restore view-state it previously saved
  - Must code to manually restore previously saved non-view state from Bundle

not a syntax requirement

Saving/Restoring Activity Instance State

- A sub class instance of Activity
  - Will execute its parent's (i.e. Activity's) lifecycle callbacks at the appropriate times if you don't override them
  - If you do override however you must manually call their super if you want them to execute in addition to their override's code
- The auto-saving/restoring of the view-state of an Activity is done in Activity's onCreate(…), onSaveInstanceState(…) and onRestoreInstanceState(…)
  - The parameter of these three callbacks is a shared Bundle object that Android uses to save/restore an Activity's view-state
  - Programmers can also use this bundle to manually save and restore an Activity's non-view state
- To auto-save/restore their state each View must have:
  - A unique ID (to uniquely tag their saved state in the bundle)
  - Their saveEnabled property set to true (the default)

Saving/Restoring Activity Instance State

- Android will auto-save the view-state of an Activity
  - If it destroys it and the user can reasonably expect its state will be maintained
    - i.e. the destruction was not instigated by the user (e.g. System kill for resources) or not the user's fault (e.g. device reorientation)
    - The view-state will be auto-saved/restored using the Bundle object that is a common parameter to an Activity sub class's inherited onSaveInstanceState, onCreate and onRestoreInstanceState methods
      - A programmer can override these methods and use the same Bundle to manually save/restore an Activity's non-view state
    - If these methods are overridden in an Activity subclass and these overrides do not call their super Android's auto-save/restore of view-state will not occur

Saving/Restoring an Activity's State

- Android will NOT save the view-state of an Activity
  - If the user by their actions indicates they are done with the Activity (i.e. require the Activity to be destroyed)
    - e.g. Back key
    - e.g. user actions causes code containing the Activity's finish method to execute
    - BUT NOT user instigated device configuration that causes an Activity destruction
  - Note
    - Under these circumstances Android does NOT even call onSaveInstanceState so any manual saving of non-view state will not occur

Summary – Saving/Restoring Instance State

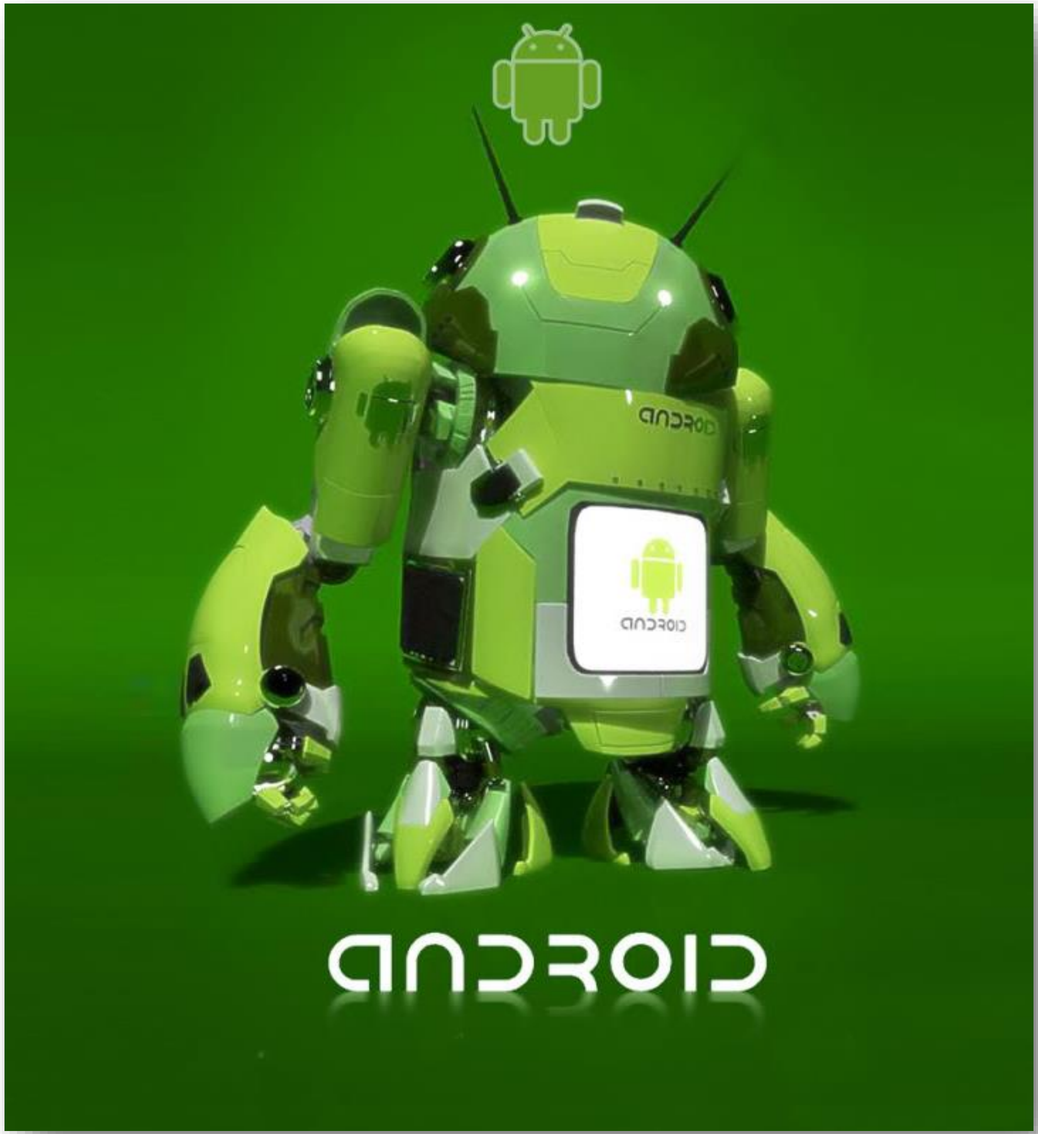
Activity loses focus	Activity State	Navigate back
Activity gets partially or fully hidden*. OS kill now possible! No OS kill!	Activity is not destroyed Activity stays in memory with instance state intact	Nothing to do State is intact
Activity gets partially or fully hidden. OS kill now possible! OS kill occurs!	Activity is destroyed onSaveInstanceState runs Android auto-saves view-state in bundle** We should follow Android's lead and save any non-view state in bundle	Relaunch, onCreate executes*** Android uses bundle to auto-restore previously saved view-state We should do the same for previously saved non-view state
Back button or swipe from recents or code executes Activity's finish() method as result of user interaction	Activity is destroyed onSaveInstanceState does NOT run Android cannot auto-save view-state Any code we write in overridden onSaveInstanceState to save non-view state will not execute	Relaunch but nothing to restore State lost Which is appropriate Default state can be set in onCreate
Device configuration change	Activity is destroyed and relaunched by Android onSaveInstanceState runs Android auto-saves view-state in bundle We should follow Android's lead and save any non-view state in bundle	Relaunch, onCreate executes Android uses bundle to restore view-state We should do the same for previously saved non-View state

* Includes sleep

** Which is stored in an activity record which is an OS maintained data store that will survive even a kill of the Activity's App process. The bundle in this activity record will be fed to the Activity's onCreate and onRestoreInstance state should the Activity be revisited

*** and onRestoreInstanceState if bundle is non-null

- Persistent Data



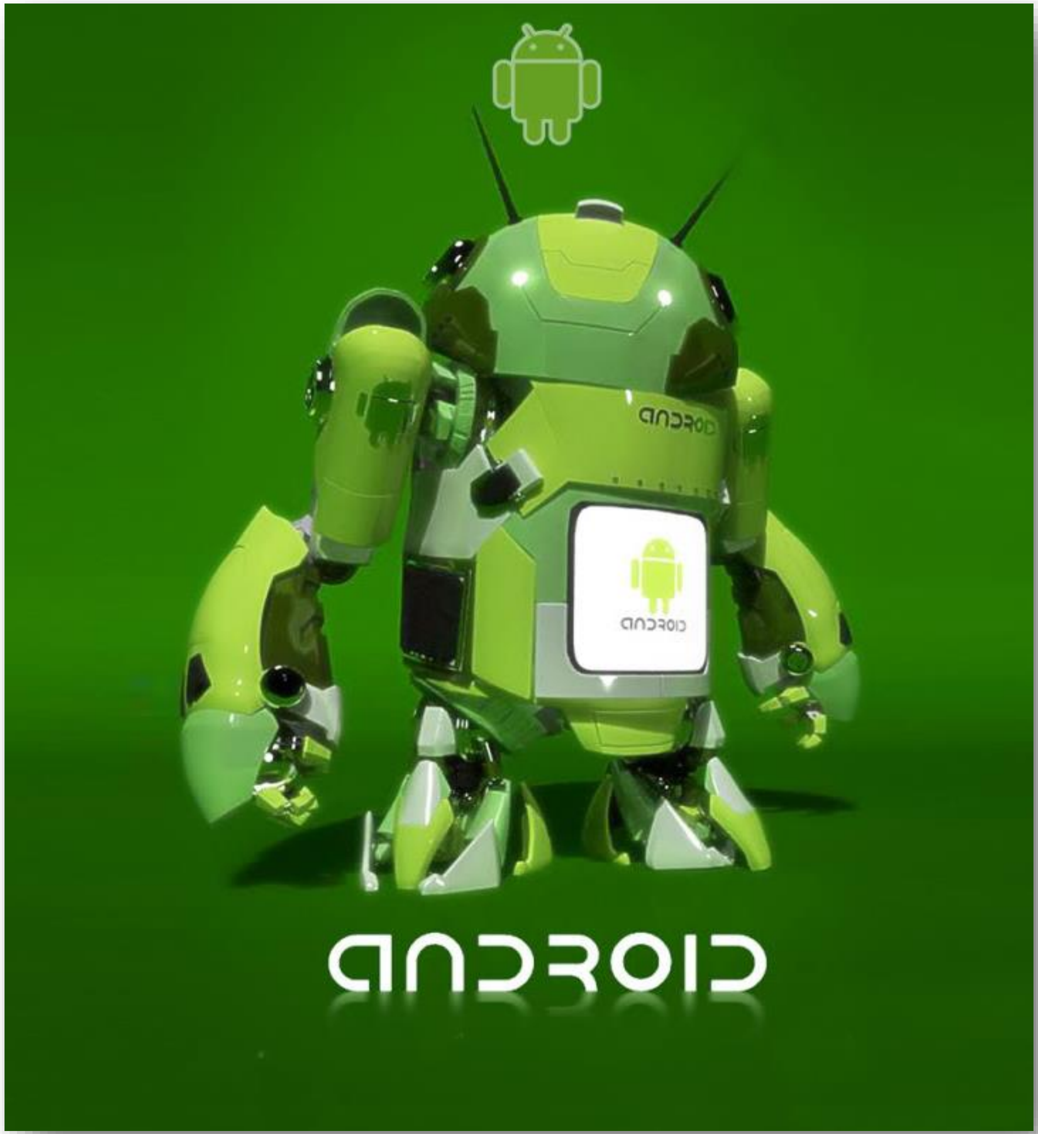
Saving/Restoring Persistent Activity Data

- Options for saving data persistently (across app uses i.e. Activity destroys)
  - Saving Key-Value Sets
    - Using a shared preferences file for storing small amounts of information in key-value pairs. We cover this next.
  - Saving Files
    - Using a basic file, such as to store long sequences of data that are generally read in order. Not covered in this unit.
  - Saving Data in SQL Databases
    - Learn to use a SQLite database to read and write structured data.
    - We will cover this in later weeks
- All these options
  - Write to and read from a device’s non-volatile memory

Saving/Restoring Persistent Activity Data

- How?
  - It involves an editor and a commit/apply action
  - See
    - See : https://developer.android.com/training/basics/data-storage/shared-preferences.html
    - See : https://developer.android.com/reference/android/content/SharedPreferences.html
- When?
  - Restore current persistent values from a SharedPreferences file to instance variables in onCreate or if nothing to restore set default values
    - This is efficient, we do not want to keep accessing non-volatile memory for these values
  - Update these instance variables holding current persistent values as required during Activity operation
  - Save the values of these instance variables holding current persistent values to their SharedPreferences file in onPause (not in onSaveInstanceState)
    - Remember onSaveInstanceState will not execute in a User kill
    - While a user may expect Activity instance state to be lost during a User kill they probably expect the latest values of any "persistent" data to be saved
      - Think current game score (Activity instance state) and highest score (persistent data) in a game app

- Reference Summary



References

- Android UI – User/Developer Perspectives

  See: https://developer.android.com/design/patterns/navigation.html

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

- Activity Lifecycles and Instance State

  See: https://developer.android.com/guide/components/activities/activity-lifecycle.html

  See : https://developer.android.com/reference/android/app/Activity.html (up to summary)

  See: https://developer.android.com/training/basics/activity-lifecycle/recreating.html?hl=es (translate page)

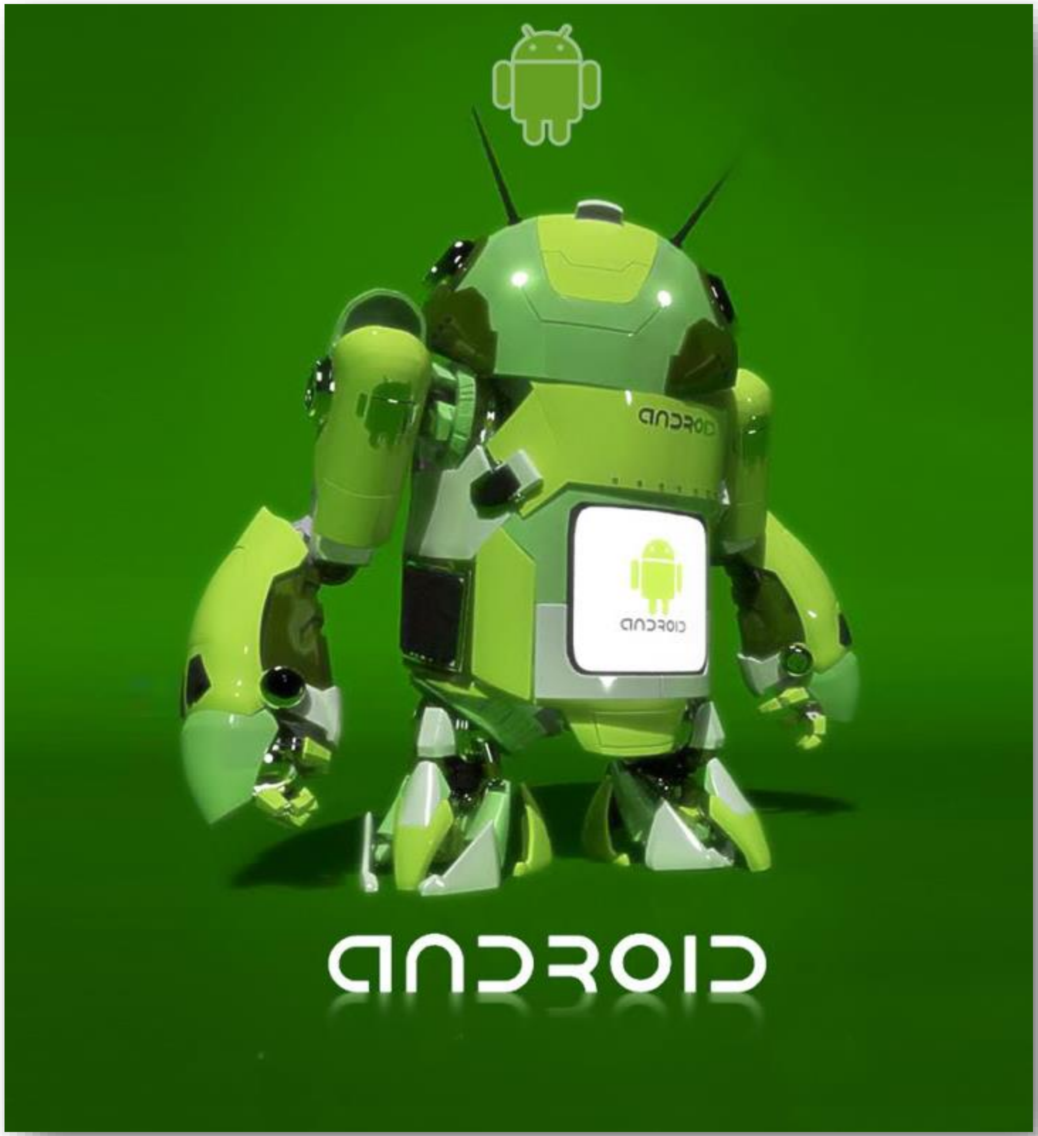
- Persistent Data Storage – Shared Preferences

  See : https://developer.android.com/training/basics/data-storage/index.html

  See : https://developer.android.com/training/basics/data-storage/shared-preferences.html

  See : https://developer.android.com/reference/android/content/SharedPreferences.html

- SimpleActivityLifecycle Demo App



What You Should Understand

USE ANDROID DOCO IF YOU DON'T UNDERSTAND

- Manifest
  - "." is shorthand for the package name
  - 3 Activities. Which one is the launch Activity
  - android:theme="@android:style/Theme.Dialog"
    - Ensures TestPartialActivity will not fill the devices screen
- Layouts more on layouts next week – but we can get a start now
  - Main
    - FrameLayout (simplest layout) only exists to contain a ScrollView widget
      - This allows access by the user to the entire layout when the device is landscape oriented since an alternative layout for landscape orientation has not been designed in this app (we will see how to do this later)
    - TableLayout, TableRow, use of android_layout_column and android_layout_span, android:layout_weight etc.
    - Difference between android:*someproperty* and android:layout_*someproperty*
      - e.g. android:gravity and android:layout_gravity
    - Use of onClick (not recommended but its quick and dirty)
    - dp and sp

What You Should Understand

USE ANDROID DOCO IF YOU DON’T UNDERSTAND

- Activity Classes
  - Lifecycles
    - Use of Log.x to send trace statement to Logcat
      - Types of Log messages
      - Use of the Logcat filtering functions
        - » Including creating your own filter
    - Lifecycle callbacks and their call to their super
    - Other callbacks and their call to super
    - Super calls involvement with saving/restoring Activity instance state (specifically what happens if you forget to call super)
      - Try commenting some out!
    - Activity instance variables
    - Why are the class level constants public?

What You Should Understand

USE ANDROID DOCO IF YOU DON'T UNDERSTAND

- Activity Classes
  - Lifecycles
    - With respect to saving Activity instance state and persistent data
      - What's going on in onCreate?
      - What going on in onSaveInstanceState, onRestoreInstanceState
    - With respect to saving Activity instance state and persistent data
      - What's going on in all the click event handlers
        - » Non-view state, instant variables and Bundle
        - » Persistent data, instant variables and SharedPreferences
    - What going on in onCreate in addition to the above
    - Launching other Activities using intents
      - In doPartialActivity and doFullActivity methods
      - Packing of data in intent to launch TestPartialActivity
    - The use of finish() in the doFinish method to allow an Activity to destroy itself

What You Should Understand

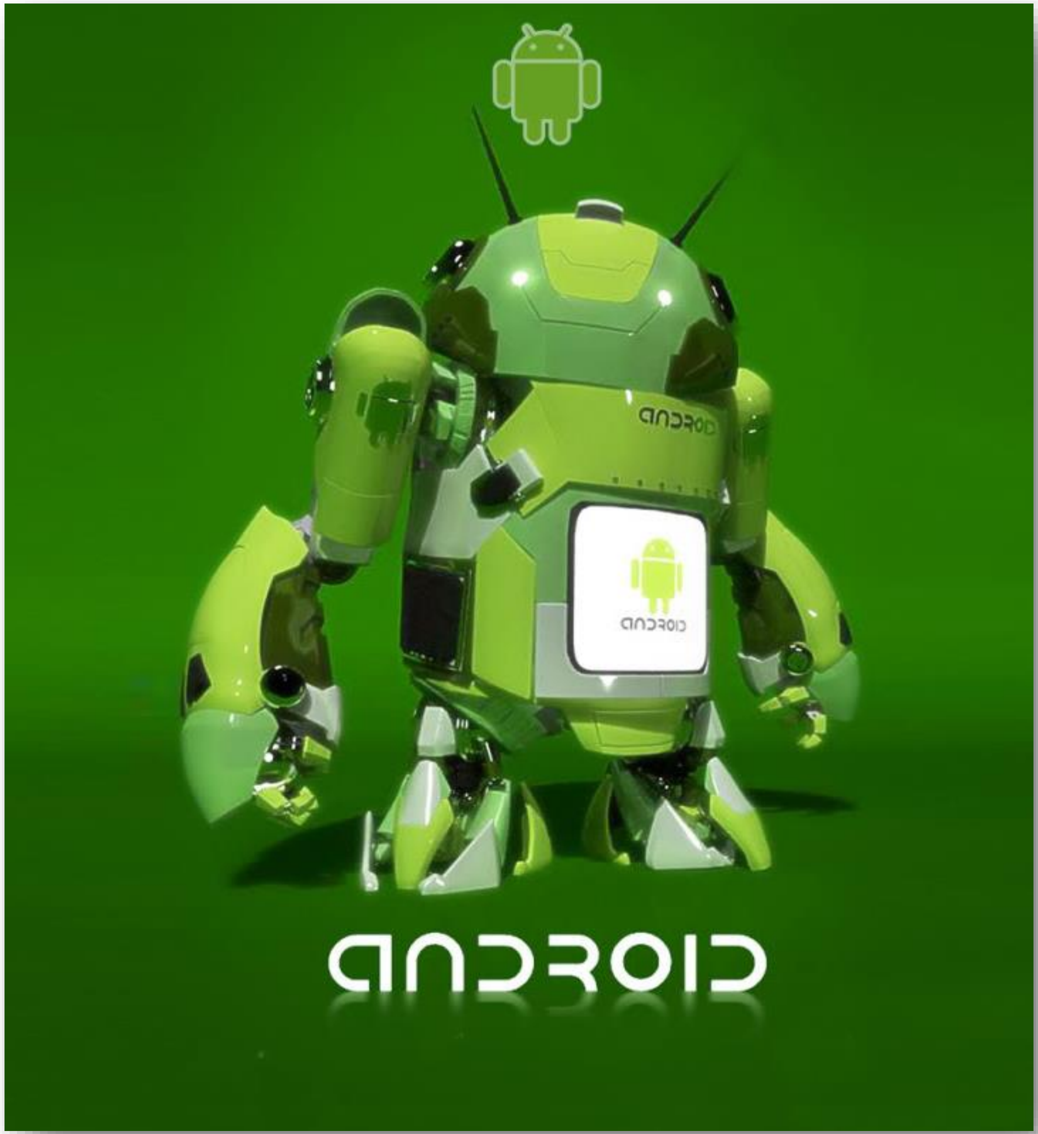
USE ANDROID DOCO IF YOU DON'T UNDERSTAND

- TestPartialActivity
  - Unpacking data passed in the intent that launched this Activity
- TestFullActivity
  - Proof SharedPreferences data is available to all Activities in an app

What You Can Ignore for Now

- Option Menu Implementation
  - Layout
    - app/res/menu/test_menu.xml
  - Methods (in Lifecycles.java)
    - onCreateOptionsMenu, onOptionsItemSelected
- Toast creation and display
  - In doToast method in Lifecycles.java
- AlertDialog creation and display
  - In doDialog method in Lifecycles.java

- API Documentation- How to Use



API Doco - Example

- Example
  - SharedPreferences settings = getSharedPreferences(SP_FILE_NAME, 0); settings.getString("persistentDataKey", "");
  - So settings is a SharedPreferences object and we use its getString method which apparently has 2 parameters. What are the semantics and allowed syntax of these parameters?
  - getSharedPreferences is a method of what? Since its not invoked on any object explicitly it must be this.getSharedPreferences(…). In this case the call is made in an Activity's code so this must be an Activity. What are the semantics and allowed syntax of these parameters?
- If you Android Studio has been set up properly
  - Highlight objects and method names and hit ctrl + q to get quick documentation including documentation of the syntax and semantics of method parameters

API Doco - Example

- Go to https://developer.android.com/index.html
  - In the search box type SharedPreferences
  - When selecting a link form the list provided look for any of the following in the link's URL
    - Reference
      - For detailed syntax and semantics
    - Guide or Training or basics
      - For "how to use" information if available
  - Choosing the reference link for SharedPreferences
    - Now use the browser's find facility (ctrl + f) to locate "getString"
    - Usually you will locate the method in a summary table
      - This will tell you the required type of the parameters and give you a brief description of what the method does
    - Click the methods name to go to more detailed documentation of the methods parameters and purpose lower on the same page
- Now go do the same for
    - getSharedPreferences