Activity Lifecycle and Task Management

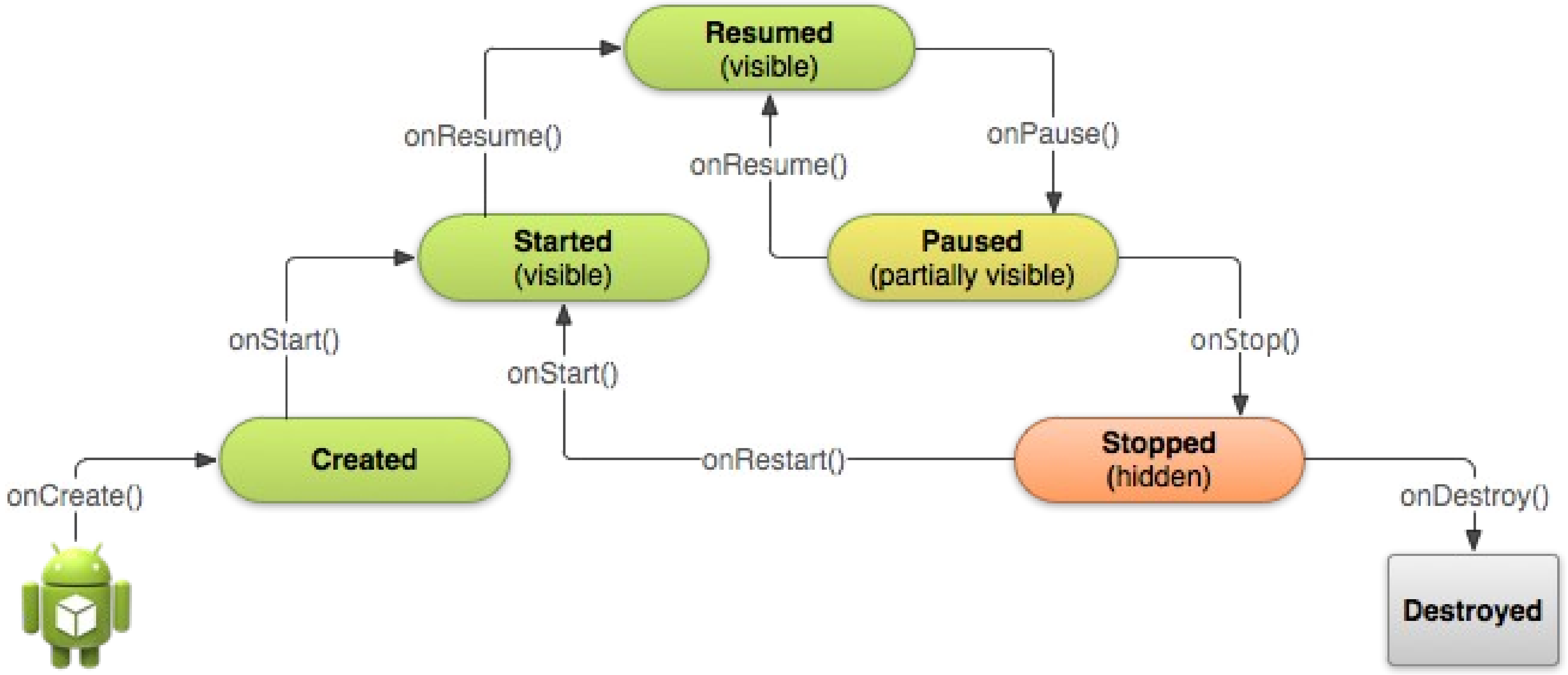
Activity Lifecycle and Task Management

- When you start android development you may be wondering what the point of having default methods such as onCreate() is
  - As you will discover in this lecture it is mainly to help preserve the scarce resources of an Android device
  - To maximise foreground application performance
  - Android forces you to write activities to its model.

You lose control of execution

- The primary reason for these methods is to give Android OS full control of all program execution.
  - When it requires the state of one of your activities to change it will call the appropriate method for you
  - Enables the OS to restrict processing to only a few activities
  - A big difference to what you were used to when you had full control from a main() method (a method you no longer have)

Life Cycle Diagram



Activity States

- As you can see from the life cycle diagram there are a number of states that your activity can be in at any given time
  - Also shows the handlers that are called when transitioning from one state to the next
  - We will ignore the created and destroyed states as they are pretty basic
    - Created state is just after object creation, destroyed is just after object destruction

Resumed State

- An activity that is in this state has been moved to the foreground ahead of all other activities and is visible to the user
  - Is considered to be running and has access to full processing power that is not consumed by background tasks or services
  - Can reach this point when the activity has been created or when it returns from a paused state
  - Generally you will have all of your helper threads enabled here if you have any in your activity

Paused State

- An activity gets into this state whenever it is partially obscured by any form of dialog.
  - As the user is being forced to make a decision it is recommended that all threads be temporarily suspended while the dialog is active
  - This state can only be reached from the resume state
  - If the user dismisses the dialog it will return to the resumed state or if the user switches to another activity or task it will go to the stopped state

Stopped State

- Represents an activity that has been moved to the background i.e. fully obscured by another activity or task
  - The activity will have ceased all processing in favour of the activities that have obscured it
  - Should a user navigate back to this activity it will move to the started state and will start processing again
  - From here it is also possible that the user can destroy the activity. Also possible that android can (temporarily) destroy it too

Started State

- Indicates that an activity is ready to be moved to the foreground and to start processing
  - An activity is in this state after it has transistioned out of the created state or has been restarted from a stopped state
  - Used to setup memory structures and threads ready for execution in the resumed state
  - Can only transistion to the resumed state from here.

Purpose of Handlers for Android

- The reason for the handlers in the case of Android is to give the OS complete control over which activities are active/inactive
  - To maximise use of limited cpu power and to conserve battery power.
  - And also to manage multitasking as activities can be kept in a list and android will ensure that only the foreground activity of the current task will be active

Purpose of Handlers for You

- These handlers give you a chance to react before android changes the state of your activity
  - Main reason for this is to relinquish resources that your activity will not need if it is being paused or stopped
  - Or to acquire resources should your activity be started or resumed
  - Also to do a full clean up of resources before your activity is moved to the destroyed state.

onCreate()

- Will be called for you after Android has generated an instance of your activity.
  - The main purpose of this handler is to generate the layout for your activity and to setup any event handlers that are required
  - Also used to get access to databases or network connections that are required by the activity
  - Called only once in the lifetime of an activity

onDestroy()

- The counterpart to the onCreate() method. This will be called before the activity is destroyed by the Android OS
  - Gives you a final chance to deallocate resources (databases, network connections) before the activity is destroyed
  - Also used to persist data from the activity for the next time it is started.
    - This includes settings and important data

onPause()

- Will be called whenever the activity has become partially obscured by a dialog
  - Generally used to pause any resources that are related to the activity while the user makes a decision
  - Whatever implementation of this method you provide it must be fast and efficient
    - As the transition between the paused and resumed states must be quick as this transition can happen multiple times a second

onResume()

- Called whenever the activity has returned from a paused state or the activity is transitioning from a started state
  - The counterpoint to the onPause() method. It is designed to reenable all resources that were previously paused by the onPause() method
    - Or those resources that have just been setup by the onStart() method.
  - Like the onPause() method the implementation here must be fast and efficient as it may be called many times.

onStop()

- Called whenever an activity is moving to the stopped state
  - Here any resources that you required for the resumed state of the application (such as threads or temporary memory structures) should be deallocated and handed back to the system
    - As it is likely that there are other applications that require the use of those resources
  - Also used to deregister interest in GPS and sensors while the activity is not being used.

onStart()

- Called after your activity has been created or after your activity has been restarted
  - Generally used to register interest in sensors and GPS while the activity is going to be active to users
  - Also used to setup threads that will be need by the activity when it becomes visible
  - The onResume() and onPause() methods should be responsible for managing the exectuion state of those threads

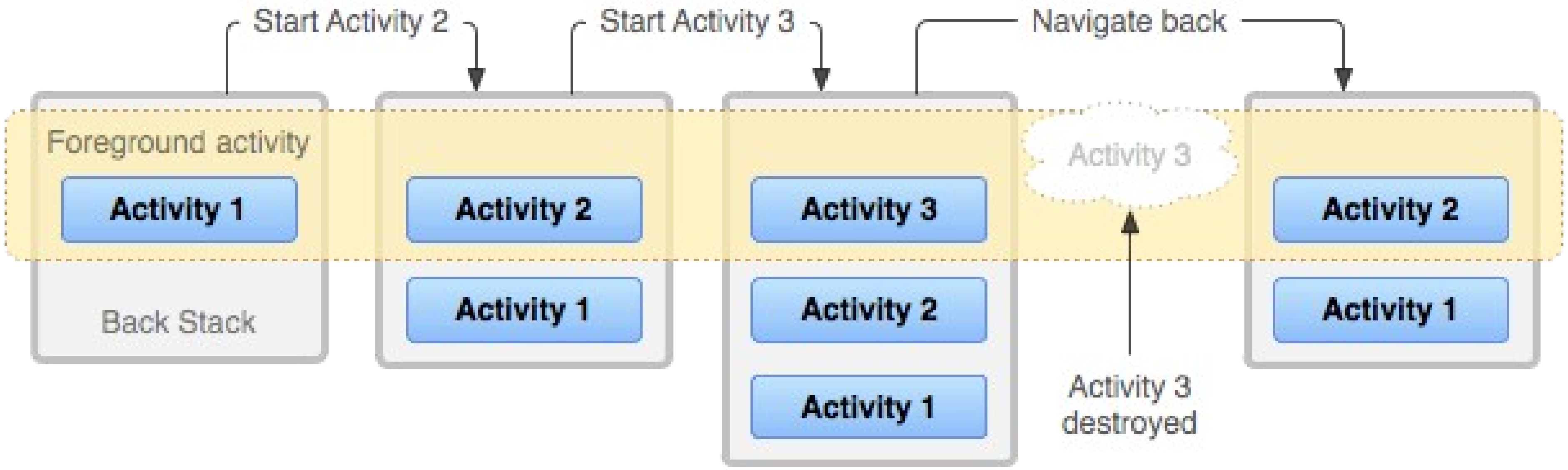
onRestart()

- Generally called after an activity is being transitioned from the stopped state into the started state
  - Most developers will avoid implementing this method as the onStart method will be called immediately after this method anyway
  - Feel free to ignore this method

Android Task List

- Everything you have seen previously concerns the management of a single activity.
  - However, an application can have more than one activity and there can also be more than one application at any given time.
  - Activities that are part of the same application are stored in tasks and are ordered in the form of a back stack
  - The collective list of back stacks produces the task list which is responsible for helping android manage all applications

Back Stack structure



How Activities within a Back Stack are Linked

- At the bottom of the stack is the root activity. The first activity which was started as part of this application
  - When a second activity is started it is pushed onto the top of the back stack. The item on top of the back stack is resumed while the previous activity is stopped
  - When the user finishes that activity or presses the back key the activity on top will be destroyed and the activity that is now top of the stack will be resumed
  - If there is no more activities on the stack the task is then deleted and removed.

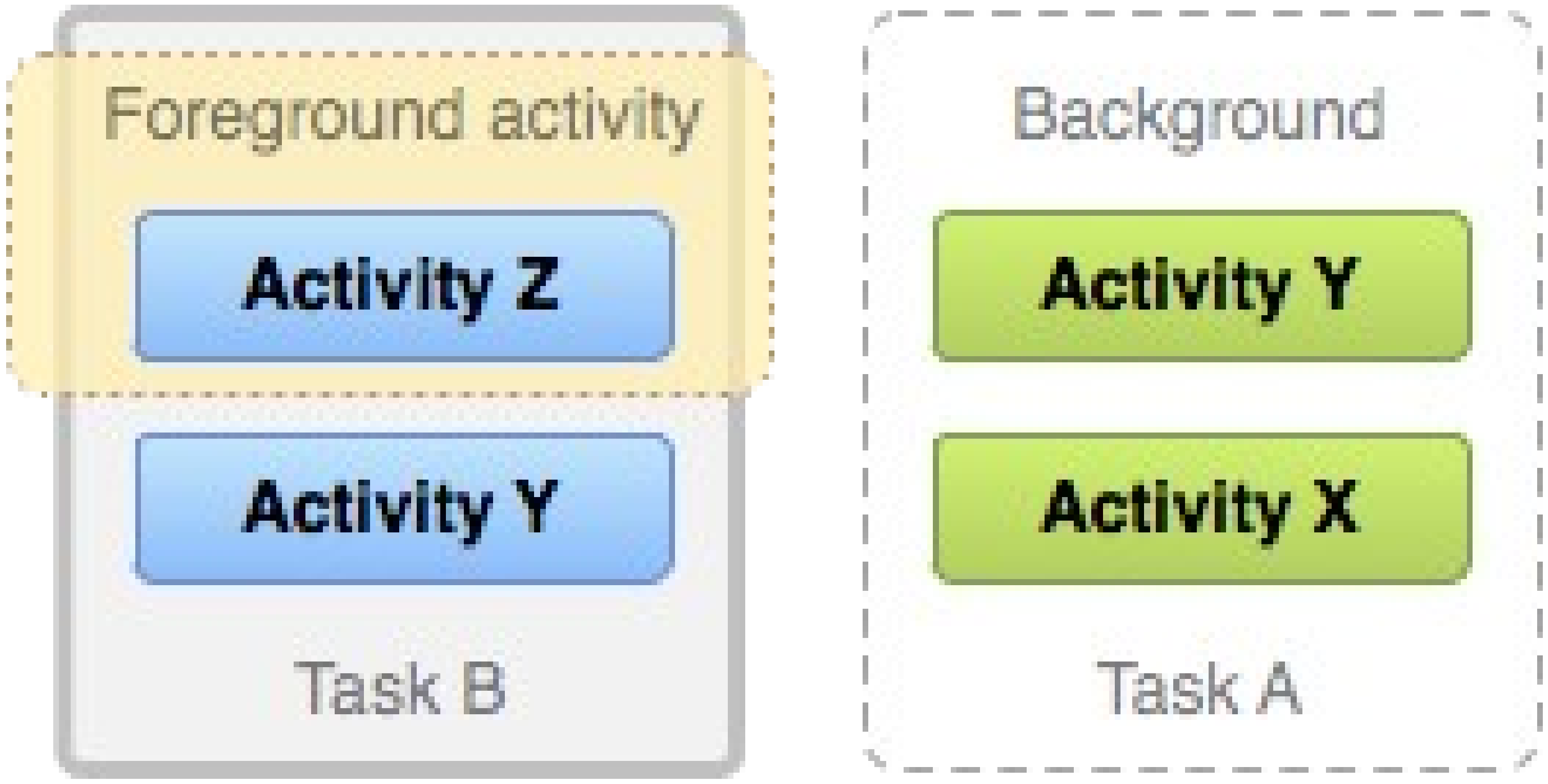
Why Multiple Instances of an Activity can be Present

- It is possible to have multiple instances of the same activity present on a back stack
  - While this may seem like a waste of memory enabling this behaviour will cause your application to become unintuitive and some what ambigious in it's actions
  - For example if we have an activity stack of A-B-C-D and we want to have activity B on the top the default behaviour of android is to generate a new instance yielding a stack of A-B-C-D-B
  - While leaving the original instance present

Problems of only using a Single Instance

- While it is possible to use a single instance it becomes unintuitive
  - Imagine the same situation as before but this time we take the instance of B and move it to the top of the stack. We then get A-C-D-B
  - However, when B is finalised or destroyed there are now two legitimate activities that can be returned to
  - D because that is next on the stack. But also A which was the initiatior of B. Initialising a new instance removes this ambiguity.

Task List Diagram



What happens when Switching Tasks

- When the user switches task the activity that is top of the back stack will be paused. The task that was selected is then removed from the list and placed at the front
  - The activity on top of the back stack is then switched from stopped to resumed.
  - As more tasks get switched in front of the original task it slows moves down to the end of the list
  - Unless the user switches it to the foreground again

Back Stack and Memory Management

- Where android uses this is for memory management, particularly when memory needs to be freed.
  - Activities that are towards the end of the back stack will be assumed to have been abandoned by the user
  - As they take a length of time to reach the end of the back stack
  - When memory needs to be freed the first action taken by android is to eliminate every activity except the root activity of the forgotten tasks

Back Stack and Memory Management

- This should free up memory for the task at the front that is requesting memory
  - However, it may not be enough. In this case android will then take the tasks at the end of the task list and destroy them completely
  - To free up more memory
  - How often this process occurs is entirely dependant on the memory requirements of the applications and also the amount of memory installed on the device itself

Back Stack and Memory Management

- This clearing process will happen less as main memory size increases
  - Less likely that a larger memory will be exhausted as often as a smaller memory
  - The other factor that influences this is the memory footprint of applications that are in regular use.
    - The closer this footprint is to the overall memory size the greater the probability of new apps requring the automatic back stack clearing process

Automatic Clearing of Back Stack

- Even if memory has not been maxed out or the task list isn't big enough to warrant activity destruction
  - The back stack will be periodically cleaned from time to time.
  - This is where the lifecycle handlers particularly onDestroy() become useful as you never know when your activity or application will die
  - Will give you a chance to save important data before your activity or application is killed.

How you can Affect Clearing with the Manifest

- It is possible to affect the automatic clearing process by setting parameters for activities in the manifest file
  - It is possible to have a single instance of an activity that will be moved to the top every time it is requested
  - Also possible to maintain your entire back stack throughout the automatic cleaning process
  - However, it is not recommended that you do this unless you have a very good reason for doing so.