Android
Application Activity Lifecycle

MVC (Model View Controller) Architecture Pattern in Android

- Developing an android application by applying a software architecture pattern.
- An architecture pattern gives modularity to the project files and assures that all the codes get covered in Unit testing.
- Easy for developers to maintain the software and to expand the features of the application in the future.

MVC (Model View Controller) Architecture Pattern in Android

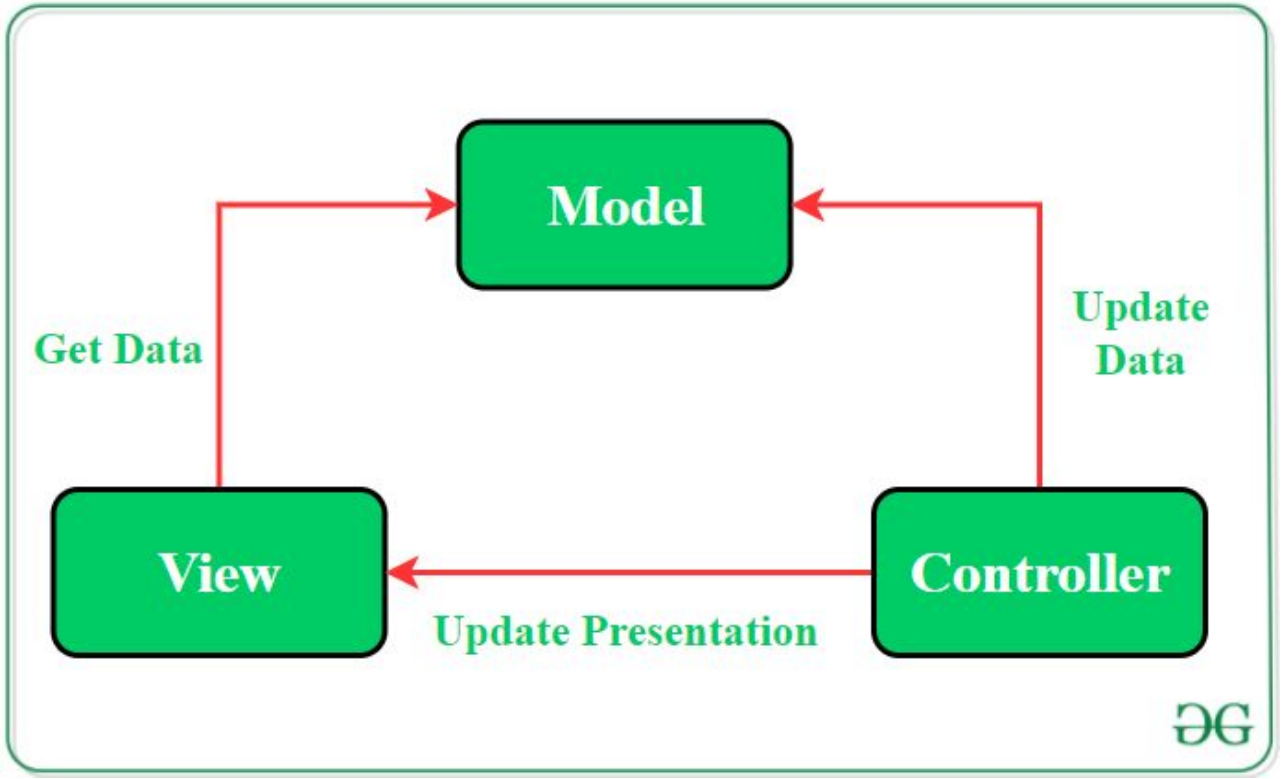
- Model—View—Controller(MVC) Pattern.
- The MVC pattern suggests splitting the code into 3 components.
  - Model: This component stores the application data. It has no knowledge about the interface. The model is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

MVC (Model View Controller) Architecture Pattern in Android

- View: It is the UI(User Interface) layer that holds components that are visible on the screen. Moreover, it provides the visualization of the data stored in the Model and offers interaction to the user.

- Controller: This component establishes the relationship between the View and the Model. It contains the core application logic and gets informed of the user's behavior and updates the Model as per the need.

MVC (Model View Controller) Architecture Pattern in Android



The Activity Lifecycle

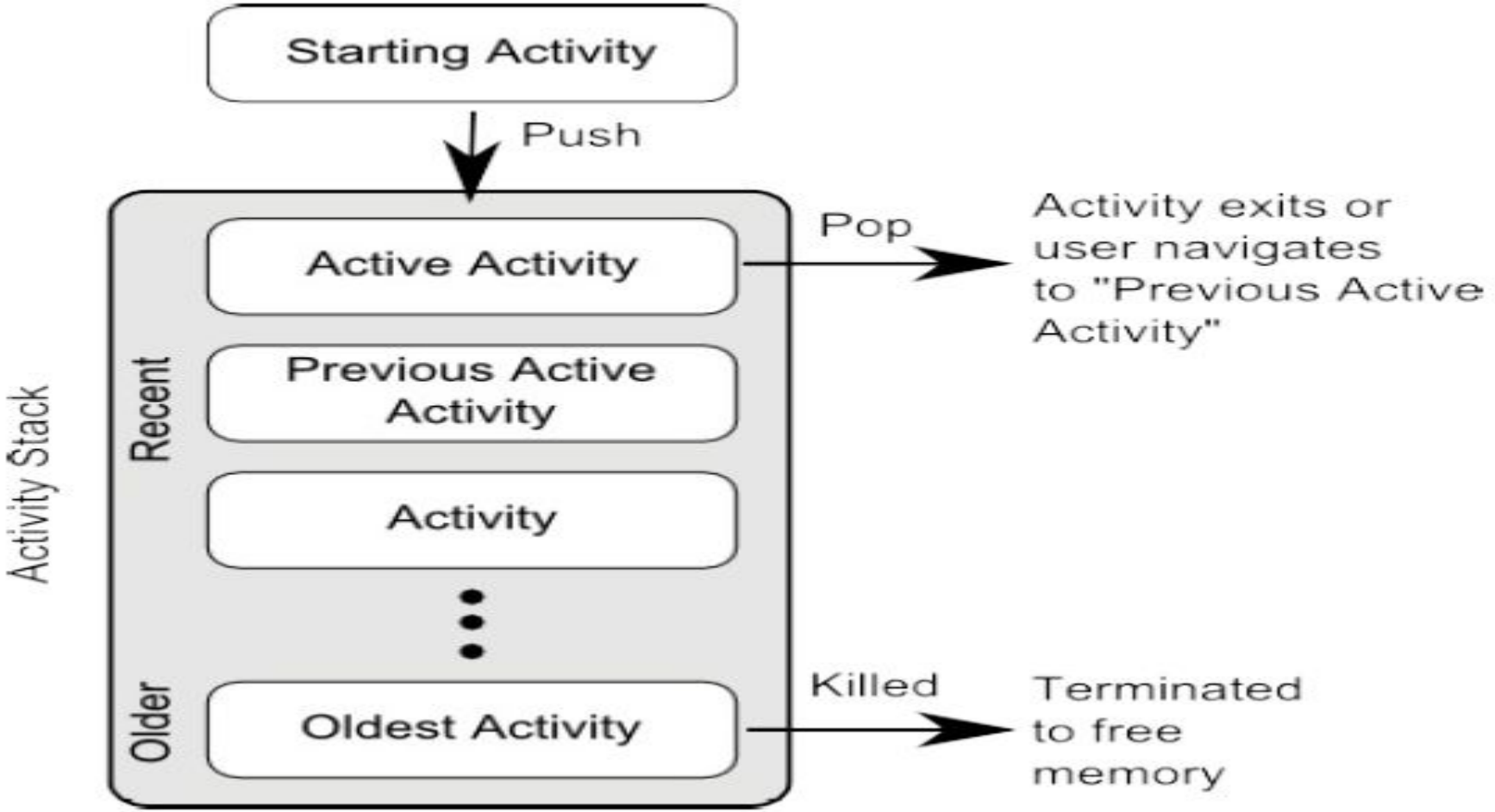
- Activities transition through different states during the execution lifetime of an application.
- The current state of an activity is determined, in part, by its position in something called the *Activity Stack*.

The Activity Stack

- For each application that is running on an Android device, the runtime system maintains an *Activity Stack*.
- When an application is launched, the first of the application's activities to be started is placed onto the stack.
- When a second activity is started,
  - It is placed on the top of the stack
  - Previous activity is pushed down.
- The activity at the top of the stack is referred to as the *active (or running)* activity.

The Activity Stack

- When the active activity exits,
  - It is popped off the stack by the runtime
  - Activity located immediately beneath it in the stack becomes the current active activity.
- New activities are pushed on to the top of the stack when they are started.
- The current active activity is located at the top of the stack until it is either
  - Pushed down the stack by a new activity
  - Or popped off the stack when it exits or the user navigates to the previous activity.
- In the event that resources become constrained, the runtime will kill activities, starting with those at the bottom of the stack.


Starting Activity
Push
Activity Stack
Recent
Active Activity
Previous Active Activity
Activity
Older
Oldest Activity
Pop
Activity exits or user navigates to "Previous Active Activity"
Killed
Terminated to free memory

Activity States

An activity can be in one of a number of different states during the course of its execution within an application:

- **Active / Running**
  - The activity is at the top of the Activity Stack
  - Foreground task visible on the device screen, has focus and is currently interacting with the user.
  - Least likely activity to be terminated in the event of a resource shortage.

Activity States

- **Paused**
  - The activity is visible to the user but does not currently have focus
    - Because this activity is partially obscured by the current active activity
  - Paused activities are held in memory
  - Remain attached to the window manager
  - Retain all state information
  - Can quickly be restored to active status when moved to the top of the Activity Stack.

Activity States

- **Stopped –** The activity is currently not visible to the user (in other words it is totally obscured on the device display by other activities).
  - It retains all state and member information
  - But is at higher risk of termination in low memory situations.
- **Killed** – The activity has been terminated by the runtime system in order to free up memory
  - No longer present on the Activity Stack.
  - Must be restarted if required by the application

Dynamic State vs. Persistent State

- *State of activity:* Mean the data that is currently being held within the activity and the appearance of the user interface.
- Eg: The activity might maintain a data model in memory that needs to be saved to a database, content provider or file.
- Such state information, because it persists from one invocation of the application to another, is referred to as the *persistent state*.

Dynamic State vs. Persistent State

- The appearance of the user interface (such as text entered into a text field but not yet committed to the application's internal data model) is referred to as the *dynamic state*
  - Since it is typically only retained during a single invocation of the application
  - Also referred to as user interface state or instance state
- The purpose of saving the persistent state is to avoid the loss of data that may result from an activity being killed by the runtime system while in the background.

Dynamic State vs. Persistent State

- Eg: An application contains an activity (Activity A) containing a text field and some radio buttons.
  - User enters some text into the text field and makes a selection from the radio buttons.
  - Before performing an action to save these changes, the user then switches to another activity
  - Causes Activity A to be pushed down the Activity Stack and placed into the background.

Dynamic State vs. Persistent State

- After some time, the runtime system ascertains that memory is low.
- Consequently kills Activity A to free up resources.
- As far as the user is concerned, Activity A was simply placed into the background and is ready to be moved to the foreground at any time.
- On returning Activity A to the foreground the user would expect the entered text and radio button selections to have been retained.
- However, a new instance of Activity A will have been created
- If the dynamic state was not saved and restored, the previous user input lost.

Dynamic State vs. Persistent State

- The main purpose of saving dynamic state is to give the perception of seamless switching between foreground and background activities.
  - Regardless of the fact that activities may actually have been killed and restarted without the user's knowledge.

The Android Activity Lifecycle Methods

- Activity class contains a number of lifecycle methods which act as event handlers when the state of an Activity changes.
- The primary methods supported by the Android Activity class are as follows:
  - **onCreate(Bundle savedInstanceState):**
    - Method that is called when the activity is first created
    - Ideal location for most initialization tasks to be performed.
    - The method is passed an argument in the form of a *Bundle* object that may contain dynamic state information (typically relating to the state of the user interface) from a prior invocation of the activity.

The Android Activity Lifecycle Methods

- **onRestart()** – Called when the activity is about to restart after having previously been stopped by the runtime system.
- **onStart()** – Always called immediately after the call to the onCreate() or onRestart() methods.
  - This method indicates to the activity that it is about to become visible to the user.
  - This call will be followed by a call to onResume() if the activity moves to the top of the activity stack, or onStop() in the event that it is pushed down the stack by another activity.

The Android Activity Lifecycle Methods

- **onResume() –** Indicates that the activity is now at the top of the activity stack and is the activity with which the user is currently interacting.
- **onPause()** – Indicates that a previous activity is about to become the foreground activity.
  - This call will be followed by a call to either the onResume() or onStop() method depending on whether the activity moves back to the foreground or becomes invisible to the user.
  - Steps may be taken within this method to store persistent state information not yet saved by the app.

The Android Activity Lifecycle Methods

- To avoid delays in switching between activities, time consuming operations such as storing data to a database or performing network operations should be avoided within this method.
- This method should also ensure that any CPU intensive tasks such as animation are stopped.

- **onStop()** – The activity is now no longer visible to the user.
  - The two possible scenarios that may follow this call are a call to onRestart() in the event that the activity moves to the foreground again, or onDestroy() if the activity is being terminated.

The Android Activity Lifecycle Methods

- **onDestroy()** – The activity is about to be destroyed
  - Either voluntarily because the activity has completed its tasks and has called the finish() method
  - Or because the runtime is terminating it either to release memory or due to a configuration change (such as the orientation of the device changing).
  - A call will not always be made to onDestroy() when an activity is terminated.

The Android Activity Lifecycle Methods

- **onConfigurationChanged()** – Called when a configuration change occurs for which the activity has indicated it is not to be restarted.
  - The method is passed a Configuration object outlining the new device configuration.
  - It is then the responsibility of the activity to react to the change.
- There are two methods intended specifically for saving and restoring the dynamic state of an activity:

The Android Activity Lifecycle Methods

- **onRestoreInstanceState(Bundle savedInstanceState):**
  - Called immediately after a call to the onStart() method in the event that the activity is restarting from a previous invocation in which state was saved.
  - As with onCreate(), this method is passed a Bundle object containing the previous state data.
  - Used in situations where it makes more sense to restore a previous state after the initialization of the activity has been performed in onCreate() and onStart().

The Android Activity Lifecycle Methods

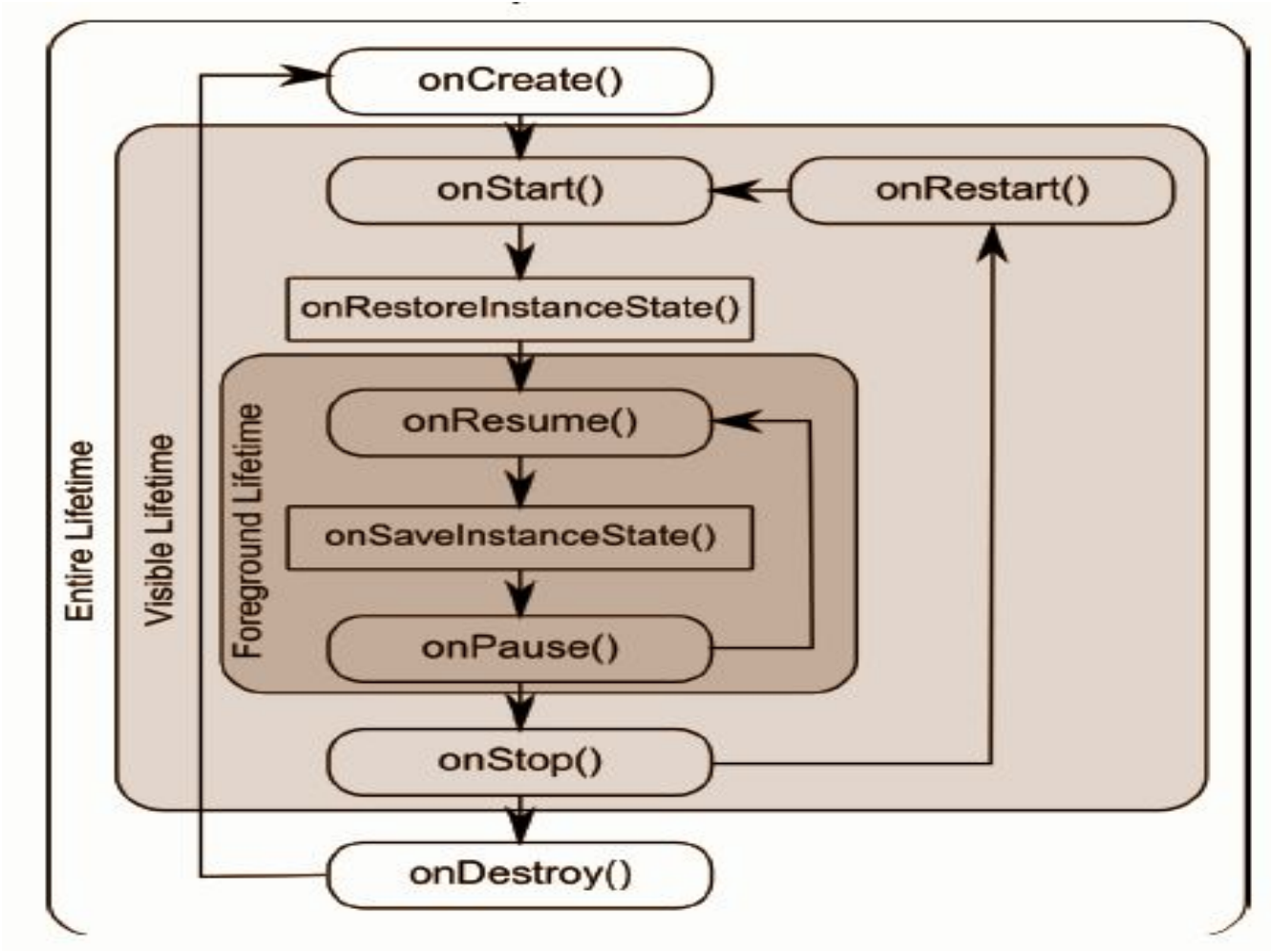
- **onSaveInstanceState(Bundle outState) –**
  - Called before an activity is destroyed so that the current dynamic state (usually relating to the user interface) can be saved.
  - The method is passed the Bundle object into which the state should be saved
  - Subsequently passed through to the onCreate() and onRestoreInstanceState() methods when the activity is restarted.
  - This method is only called in situations where the runtime ascertains that dynamic state needs to be saved.

Activity Lifetimes

- Lifetimes through which an activity will transition during execution:
  - **Entire Lifetime** – The term "entire lifetime" is used to describe everything that takes place within an activity between the initial call to the onCreate() method and the call to onDestroy() prior to the activity terminating.
  - **Visible Lifetime** – Covers the periods of execution of an activity between the call to onStart() and onStop().
    - During this period the activity is visible to the user though may not be the activity with which the user is currently interacting.
  - **Foreground Lifetime** – Refers to the periods of execution between calls to the onResume() and onPause() methods.

Activity Lifetimes

- An activity may pass through the *foreground* and *visible* lifetimes multiple times during the course of the *entire* lifetime.


onCreate()
onStart()
onRestart()
onRestoreInstanceState()
onResume()
onSaveInstanceState()
onPause()
onStop()
onDestroy()
Entire Lifetime
Visible Lifetime
Foreground Lifetime

Bundle Class

- Container for storing data
- Stores as key (any string) and value pair

Thank You..!!