NOTES

Bachelor of Science (Third Year)
CSC-108-Mobile Application Development
Unit : II - Activities and UI
Module Name : Activity Life Cycle, Understanding Exception Handler
Module Number : 11

An activity is a crucial component of an Android app. Activity serves as the entry point for an app's interaction with the user. When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. Activities in the system are managed as activity stacks.A new Activity is placed on top of current stack & becomes the running activity.

Let us look at the different States of App (or, the main App Activity). Starting from a user clicking on the App icon to launch the app, to the user exiting from the App, there are certain defined states that the App is in, let's see what they are.

Active State: When a user clicks on the App icon, the Main Activity gets started and it creates the App's User Interface using the layout XMLs. And the App or Activity starts running and it is said to be in active state. When an Activity is in active state, it means it is active and running. It is visible to the user and the user is able to interact with it.Android Runtime treats the Activity in this state with the highest priority and never tries to kill it.

Paused State: When any dialog box appears on the screen, like when you press exit on some apps, it shows a box confirming whether you want to exit or not. At that point of time, we are not able to interact with the App's UI until we deal with that dialog box/popup. In such a situation, the Activity is said to be in paused state. An activity being in this state means that the user can still see the Activity in the background such as behind a transparent window or a dialog box i.e it is partially visible. The user cannot interact with the Activity until he/she is done with the current view. Android Runtime usually does not kill an Activity in this state but may do so in an extreme case of resource crunch.

Stopped State: When we press the Home button while using the app, our app doesn't closes. It just get minimized. This state of the App is said to be stopped state. 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.

Distroyed State: When we finally destroy the App i.e when we completely close it, then it is said to be in destroyed state. When a user hits a Back key or Android Runtime decides to reclaim the memory allocated to an Activity i.e in the paused or stopped state, It goes into the Destroyed state. The Activity is out of the memory and it is invisible to the user.

Let us now come to the Life Cycle of an activity. The activity life cycle consists of five main functions: onCreate(), onStart(), onResume(), onPause(), onStop(), onRestart() and onDetroy(). Let us look at each one in detail.

onCreate()

- Called when the system creates your activity.
- Initialize the essential components of your activity ex. create views, bind data to lists etc.
- Define your layout resource i.e. XML file.
- Includes calling setContentView() to define the layout for the activity's user interface.
- When onCreate() finishes, the next *callback* is always onStart()

onStart()

- As onCreate() exits, the activity enters the Started state, and the activity becomes visible to the user.
- Contains activity's final preparations for coming to the foreground and becoming interactive.

onResume()

- Invoked just before the activity starts interacting with the user.
- The activity comes to the top of the activity stack, and captures all user input.
- App's core functionality is implemented in the onResume() method
- Followed by onPause()

onPause()

- Called when the activity loses focus and enters a Paused state. Ex. When user taps the Back or Recents button
- Activity is partially visible but user is leaving the activity
- *Activity* will soon enter the Stopped or Resumed state
- Activity may continue to update the UI. Ex. navigation map screen or a media player playing. User will expect the activity to keep updating.
- Should not be used to save application or make network calls
- The next callback is either onStop() or onResume() depending on what happens after the activity enters the Paused state

onStop()

- Called when activity is no longer visible to the user

- Occurs when the activity is being destroyed, a new activity is starting, or an existing activity is entering a Resumed state and is covering the stopped activity
- The stopped activity is no longer visible at all.
- The system can call onRestart(), if activity is coming back to interact with the user, or by onDestroy() if this activity is completely terminating

onRestart()

- Activity in the Stopped state is about to restart.
- Restores the state of the activity from the time that it was stopped.
- Followed by onStart()

onDestroy()

- onDistroy() is called before an activity is destroyed.
- Final callback that the activity receives
- Activity's resources are released.

The subtopic we learn here is the Exception Handler**.** The Java eception handling mechanism is controlled by firve keywords: *try, catch, throw, throws, and finally.*Program statements to monitor are contained within a **try** block.The **catch** block immediately follows the try block.There can be more than one catch blocks each catching different exceptions.The **throw** keyword in Java is used to explicitly throw an exception from a method or any block of code. The **throws** is a keyword in Java is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The f**inally** block will be executed after a try/catch block has completed and before the code following the try/catch block. Finally block will be executed whether or not an exception is thrown.The rest of the program continues to execute hereafter.

Let us look at the control flow in exception handling with the following three cases

Case 1: Exception occurs in try block and handled in catch block.

- If an exception occurs within the try block, rest of the try block doesn't execute,control is thrown to the catch block.
- The exception is caught by the corresponding catch block.
- Code within catch block catches the exception and handles it.
- After executing catch block, the control will be transferred to finally block(if present) and then rest program will be executed.

Case 2: Exception occurs in try-block is not handled in catch block

- If catch block is unable to catch an exception then default exception handling mechanism is invoked.
- If finally block is present, it will be executed followed by default handling mechanism.

Case 3: Exception does not occur in try-block

- Catch block never runs as they are only meant to be run when an exception occurs.
- If present, finally block will be executed followed by rest of the program

Here are the references for the content material for this presentation