Introduction to Activity Lifecycle

Activity

Activity is the entry point of the android application and basically, it is the screen where users interact with the android application and this is the place where we define how the screen should behave in which situations. In the activity, we can call all the UI-related functions and do whatever we want with them. Also, we can initialize and modify the view component here. Therefore, while the user is interacting with the application, the activity needs to know what the user is currently doing, such as when the user goes to another activity, or the user receives a phone call, or even if the user's phone is out of battery, and for that cases, the activity needs to take some action. For example, the activity needs to take action such as stopping the processes it is doing in the background. That's why we have ***activity lifecycle***.

The Activity lifecycle

In addition to the above, the activity needs to know if the user is using the app or not, using the activity lifecycles callbacks. Callbacks call functions when the activity state changes. Let's consider a scenario where the user has to fill in the payment information and gets the OTP, and the user exits the app for the OTP. And when the user returns to the application, the payment informations disappears and the user has to fill in the payment information again. We can handle these issues using activity lifecycle callbacks. In later section, we see activity lifecycle callbacks in detail.

There are mainly six lifecycle callbacks that when the activity state changes: *onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy()*

onCreate() - It's similar to the main method that we have in Java. Every activity must have an onCreate() method. It calls this method when the activity starts and sets the XML layout with setContentView. And all UI-related functions are called in this method. This method has a parameter called savedInstanceState, which is the Bundle type that holds the data key-value pair. If the activity called for the first time savedInstanceState is empty. Otherwise, it is not null. Therefore, this callback creates activity. The activity doesn't stay in the onCreate() method. The state changed to the next callback called *onStart().*

onStart() - So, the activity is created and the state is changed to onStart() . In this callback, the application comes to foreground and the UI is visible to the user. This callback finishes very quickly, the state changes to the next callback called onResume().

onResume() - After the application is created, the activity state resides in this callback. If the user clicks the home button or presses back etc., the activity state is changed to onPause(). After the user returns to the app, the activity state again changes to onResume() and the state stay there unless the user takes an action that changes the activity state, such as clicking the home button. For example, we have a chat application, we want to know if the user is online or not, and we can say that the user is online if the activity state is in onResume().

onPause() - This is state when the app loses the foreground and goes into the background. It is scenario like we open a window on the PC, then open another window and the first window is not visible to us, but it is running in the background. Thus, it is a situation where the activity is not visible to the user or partially visible to the user (for example, in multi-window mode).

onStop() - The state is changed to onStop() if the activity is not visible or new activity is started. If the operating system needs memory, it releases the activity. If the activity comes to the foreground, the state is changed to onCreate() directly, unlike onPause(). Because in onPause it goes to onResume() and activity starts where it left off. If we think that we need to save

information(such as a draft in Gmail) before the activity ends, we can make the network request here. Also, note that we shouldn't be doing network calls in onPause(). Because onStop() lifetime is greater than onPause().

onDestroy() - This state is called when the activity is closed. Also, this state is called when the configuration state changes, such as a rotating device. If we come back to this activity, the application creates new activity and onCreate is called with a null Bundle value. And in this case, we can't do the CPU-intensive process because the OS can kill that process before the onDestroy()'s code is executed.

Handling Lifecycle
Handling the life cycle is an important issue in Android OS. We can handle the life cycle with the methods described above. Consider the scenario described below:

```
class MusicActivity : AppCompatActivity() {
    private lateinit var isPlayMusic: Boolean
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_music)
      (...)
      isPlayMusic = true
    }

    override fun onResume() {
      super.onResume()
      if(isPlayMusic == true) {
        playMusic()
      }
    }

    override fun onPause() {
      super.onPause()
      if(isPlayMusic == true) {
        pauseMusic()
      }
    }

    override fun onDestroy() {
      isPlayMusic = false
      super.onDestroy()
    }

  }
}
```

Let's say we have an app, we play music while the app is in the foreground, and we handle its lifecycle from scratch. And it is difficult to handle the whole lifecycle using lifecycle methods. Also, in other scenarios, this causes memory leaks and poor organization of the code. So, this is where **lifecycle-aware components** come into play. Using lifecycle-aware components, it automatically checks the activity state based on the activity's current state. We can use lifecycle-aware components using androidx.lifecycle package. The best practice is to use the ViewModel for acquire the data and use the observable data holders to reflect the changes in the UI.

Let's recall the problem in the example above, which are lifecycles handled using callback methods. Now let's consider this block of code using the ViewModel. First of all, we obtain all the songs locally using the repository pattern. And, we implement the playMusic() and pauseMusic() methods and observe them using observable data holder. As you have noticed, we have not implemented any lifecycle callback methods. Because we use ViewModel, we don't have to use

any of these methods. ViewModel has its own scope() and onCleared() method to handle lifecycle. Note that these code blocks are just pseudocode for understanding the concept.

```
@HiltViewModel
class MusicsViewModel @Inject constructor(
     private val musicsRepository: MusicRepository
) : ViewModel() {

    fun fetchAllSongsFromDevice() : MutableLiveData<Songs> {
           return musicRepository.loadMusics();
    }

    fun playMusic() : MutableLiveData<Songs> {
        return musicRepository.play();
    }

    fun pauseMusic() : MutableLiveData<Songs> {
        return musicRepository.pause();
    }


}
```