Android Mobile Application Development
Week 6: Android Lifecycle

In Android an activity is a single, specific task a user can do. Most apps include several different activities that allow the user to perform different actions. Generally, one activity implements one screen in an app.

When the user taps the launcher icon on the Home screen, it starts the main activity in the AndroidManifest.xml file. By default this is the activity you defined when you created your Android project.
Every app must have an activity that is declared as the launcher activity. This is the main entry point to the app

- The main activity for your app must be declared in the AndroidMainfest.xml file
- Must have an <intent-filter> that includes the MAIN action and LAUNCHER category
- When you create a new Android project the default project files include an Activity class that's declared in the manifest with this filter.
- If either the MAIN action or LAUNCHER category are not declared for one of your activities, then your app icon will not appear in the Home screen's list of apps.

[show this in the Hello Android app]

Activities transition through different states throughout their life cycle that are important to understand.

Android Activity States (slide)

- Created
  - An app's main activity has been launched
  - Your activity does not reside in the Created state, it then enters the Started state.
- Started
  - Activity is becoming visible
  - As with the Created state, the activity does not stay resident in the Started state, it then enters the Resumed state.
- Resumed
  - App is visible in the foreground and the user can interact with it
  - This is the running state
  - This is the state in which the app interacts with the user. The app stays in this state until something happens to take focus away from the app.
- Paused
  - The activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode).
  - User is leaving the activity (though it does not always mean the activity is being destroyed)
  - Activity is partially visible, another activity is in the foreground
  - When paused it does not receive user input and doesn't execute any code
- Stopped
  - Activity is in the background and no longer visible

- Destroyed
  - All app processes have ended

Android has callback methods that correspond to specific stages of an activity's lifecycle. There is a sequence of callback methods that start up an activity and a sequence of callback methods that tear down an activity.

Android Lifecycle Methods
https://developer.android.com/guide/components/activities/activity-lifecycle.html
The lifecycle methods are all from the Activity class

- **onCreate() –** activity is first created
  - Good place to initialize the essential components of your activity and do setup
  - calls **setContent()** which triggers initial composition of composables, rendering the UI
  - Your activity does not reside in the Created state. After the **onCreate(Bundle)** method finishes execution, the activity enters the Started state, and the system calls the **onStart()** and **onResume()** methods in quick succession.
- **onStart()** – activity is becoming visible
  - **onStart()** contains the activity's final preparations for coming to the foreground and becoming interactive.
  - Followed by **onResume()** if the activity comes into the foreground
  - Followed by **onStop()** if the activity is made invisible
  - The **onStart()** method completes very quickly and, as with the Created state, the activity does not stay resident in the Started state. Once this callback finishes, the activity enters the Resumed state, and the system calls the **onResume()** method.
- **onResume()** – activity is in the foreground
  - the system calls **onResume()** every time your activity comes into the foreground, including when it's created for the first time and when the app goes from Paused to Resumed
  - This is where the lifecycle components can enable any functionality that needs to run while the component is visible and in the foreground.
  - When an interruptive event occurs, the activity enters the Paused state, and the system invokes the **onPause()** callback.
- **onPause() –** activity is no longer in the foreground, another activity is starting
  - The system calls this method as the first indication that the user is leaving your activity (though it does not always mean the activity is being destroyed); it indicates that the activity is no longer in the foreground (though it may still be visible if the user is in multi-window mode).
  - An activity in the Paused state may continue to update the UI if the user is expecting the UI to update (ie Google Maps)
  - Release or adjust any operations or release any resources not needed when paused
  - **onPause()** execution is very brief, and does not necessarily afford enough time to perform save operations. Instead, you should perform heavy-load shutdown operations during **onStop().**
  - Followed by **onResume()** if the activity returns to the foreground
  - Followed by **onStop()** if the activity becomes invisible

- **onStop() –** activity is no longer visible
  - Use **onStop()** to perform large, CPU intensive operations such as saving application or user data, make network calls, or execute database transactions.
  - Followed by **onRestart()** if the activity becomes visible again
  - Followed by **onDestroy()** if the activity is going to be destroyed
  - If the device is low on memory **onStop()** might not be called before the activity is destroyed
- **onRestart()** – activity was stopped and is about to restart
  - **onRestart()** doesn't get called the first time the activity is becoming visible so it's more common to use the **onStart()** method
- **onDestroy() –** activity is about to be destroyed
  - The system invokes this callback either because:
    - the activity is finishing (due to the user completely dismissing the activity or due to finish() being called on the activity)
    - the system is temporarily destroying the activity due to a configuration change (such as device rotation or multi-window mode)
  - The **onDestroy()** callback should release all resources that have not yet been released by earlier callbacks such as **onStop().**

When you override these methods to implement them make sure you call their super class method first.

Pausing and Resuming

- When an activity in the foreground becomes partially obscured it becomes paused
  - Stop ongoing actions
  - Commit unsaved changes (if expected)
  - Release system resources
- As long as an activity is partially visible but not in focus it remains paused
- When the user resumes the activity you should reinitialize anything you released when it paused

When your activity is paused, the Activity instance is kept in memory and is recalled when the activity resumes.

Stopping and Restarting

- If an activity is fully obstructed and not visible it becomes stopped
  - Switches to another app
  - Another activity is started
  - User gets a phone call
- The activity remains in memory while stopped
- If the user goes back to the app, or uses the back button to go back to the activity, it is restarted

When your activity is stopped, the Activity instance is kept in memory and is recalled when the activity resumes.

You don't need to re-initialize components that were created during any of the callback methods leading up to the Resumed state.
The system also keeps track of the current state for each UI component in the layout.

Destroying and Recreating

- An activity is temporarily destroyed and then recreated when there's a change in device configuration so the new device configuration can be loaded
  - Device configuration includes screen orientation, screen size, whether there's a keyboard attached, switching into multi-window mode, and also system-wide settings specified by the user such as the locale, font or dark/light mode.
- If the system destroys the activity due to system constraints, then although the actual activity instance is gone, the system saves some instance state data. This includes information about each UI component in your activity layout in a Bundle object
  - If the user navigates back to that activity, a new instance of the activity is recreated using the data saved in the Bundle object
- An activity is destroyed and finished when the system decides it's no longer needed
  - User presses the back button
  - Hasn't been used in a long time
  - Needs to recover memory

Configuration Changes
Although a change in device configuration destroys and recreates activities, the user expects the app to handle this seamlessly.
In Compose you can save state through configuration changes in two ways.

1. Use `rememberSaveable` instead of `remember`
   a. `rememberSaveable` retains state across activity recreation in addition to across recompositions.
2. Use ViewModel which is a lifecycle aware architecture component
   a. ViewModel survives configuration changes automatically as it will remain in memory until the activity completely finishes (activity is destroyed or in a single activity app, the app exits).
   b. Regardless of how many times an activity is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency.
   c. The ViewModel remains in memory until the activity is finished and onCleared() is called.

These approaches handle transient data during the app's lifecycle, but do not handle data persistence across app launches. We will look at data persistence approaches later in the semester.

HelloAndroid
Run HelloAndroid and rotate the device to see how the textfield user input and the resulting text in the Text composable are cleared. You see the same behavior if you change the dark mode setting. These device configuration changes are causing the activity to be destroyed and recreated. You can add a log message in onCreate() to see this.

```
Log.d("oncreate","restart")
```

Let's update HelloAndroid so the state is saved during device configuration changes. (HelloAndroid theme lifecycle)

To save our state across configuration changes we change `remember` to `rememberSaveable.`

```
var name by rememberSaveable { mutableStateOf("") }
var textFieldName by rememberSaveable { mutableStateOf("") }
```

Now when you run the app and trigger a configuration change by rotating the device or changing the dark mode the text in the TextField and in the Text composables are saved.

In my layout on the device I was using you couldn't see the Text composable in landscape orientation so in Greeting() I added scrolling for the column for the layout to work in landscape.

```
Column(
    horizontalAlignment = Alignment.CenterHorizontally,
    modifier = Modifier
        .verticalScroll(rememberScrollState(), enabled = true)
)
```

I also added bottom padding to the modifier for the box.

```
.padding(bottom = 10.dp)
```