Mobile Application Development
Week 11: 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 Halloween 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, 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 **setContentView(View)** to set the layout used for the activity's user interface
    - **setContentView(R.layout.activity_main)** is how the activity knows which layout to load, or inflate
    - Layout inflation takes the views from the layout file and creates View objects that are stored in Java memory.
    - Once the View objects are in memory they are drawn to the screen and can be accessed and modified programmatically.
    - Can't access layout views until this method has been called and the layout has been inflated.
  - 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 View in the layout.

Destroying and Recreating

- An activity is destroyed 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
- A change in device configuration such as rotation or switching into multi-window mode results in the activity being destroyed and then recreated so the new device configuration can be loaded
  - Device configuration includes screen size, screen orientation, whether there's a keyboard attached, and also configuration options specified by the user (such as the locale).
- 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 View object 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
- As an activity begins to stop, the system the calls the **onSaveInstanceState(Bundle)** method to save the current state of the activity https://developer.android.com/guide/components/activities/activity-lifecycle.html#instance-state
  - By default the Bundle instance saves information about each View object in your activity layout (such as the text in an EditText)
  - To save additional simple UI data use the **savedInstanceState.putxxx()** methods to save key/value pairs
    - putInt()
    - putBoolean()
    - putLong()
    - etc
  - A Bundle object isn't appropriate for preserving more than a trivial amount of data because it requires serialization on the main thread and consumes memory.
- To restore activity UI state you can recover your saved instance state from the Bundle that the system passes to your activity. The **onRestoreInstanceState()** method is called after the **onStart()** method and is passed the Bundle
  - To restore the saved UI data use the corresponding **savedInstanceState.getxxx()** methods

Halloween

Let’s update Halloween so the state is saved when the device is rotated and the activity is destroyed.
(Halloween theme constraints state)

Let’s update the app to handle device configuration changes by saving our UI state. The text in the EditText is automatically saved so we’ll save the message and the image id. Although this project has only one image we’ll set it up so it would work for multiple images as well.

In MainActivity.kt we had everything in our sayBoo() method. But now we’ll need access to the EditText, TextView, and ImageView throughout the class so we’ll define them at the class level. But you can’t call findViewById() until after the layout file has been set as the content view in onCreate(Bundle). So you’ll need the keyword lateinit before their definitions since they will be initialized later.

```
lateinit var editName: EditText
lateinit var booText: TextView
lateinit var imageGhost: ImageView
```

A good place to initialize them is in onCreate() after setContentView().

```
editName = findViewById<EditText>(R.id.editTextName)
booText = findViewById<TextView>(R.id.textMessage)
imageGhost = findViewById<ImageView>(R.id.imageView)
```

We’ll also need our message and image in multiple methods, so we’ll make them variables at the class level. I made imageId nullable so we could practice working with that type in Kotlin.

```
var message: String = ""
var imageId: Int? = null
```

Then we’ll reorganize our sayBoo() method by separating out where we update our UI so we can reuse that method when we’re restoring our UI state.

```
fun sayBoo(view: View) {
    //EditText
    val name = editTextName.text
    //message
    message = "Happy Halloween " + name + "!"
    //image
    imageId = R.drawable.ghost
    updateUI()
}

fun updateUI(){
    //TextView
    booText.text = message
    //ImageView
    imageId?.let { imageGhost.setImageResource(it) }
}
```

Note the use of the let() method which will only execute the action if imageId is not null. Do you remember why the instructionis in {}? What is "it"?
It's a lambda expression and "it" is the default name for a single parameter.

Now let's implement the methods to save and restore instance state.

If you just start typing onsaveinstance… autocomplete will do the rest for you.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
}
```

We'll use key/value pairs to put our message String and imageId into the Bundle instance outState. We'll only add the imageId if it's not null.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putString("message", message)
    imageId?.let { outState.putInt("image", it) }
}
```

onRestoreInstanceState() is called after the onStart() method only if there is a saved state to restore. In that method you can retrieve any data you saved using the same keys so you can restore the saved state which we do by calling updateUI() to update our TextView and ImageView.

```
override fun onRestoreInstanceState(savedInstanceState: Bundle)
{
    super.onRestoreInstanceState(savedInstanceState)
    message = savedInstanceState.getString("message", "")
    imageId = savedInstanceState.getInt("image")
    updateUI()
}
```

Test the app and rotation. Test it with data and without to test the nullable variable properly.