Android Development with Kotlin

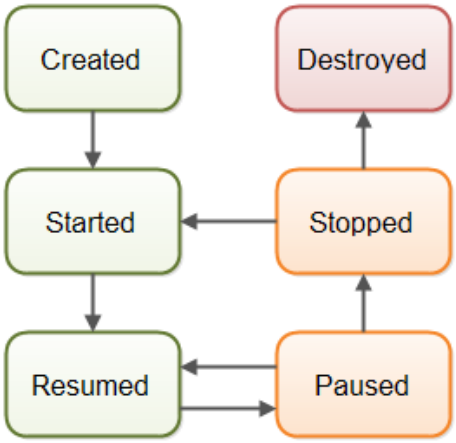
Activity in Android

What is an Activity?

An Android activity is one screen of the Android app's user interface. In that way an Android activity is very similar to windows in a desktop application. An Android app may contain one or more activities, meaning one or more screens. The Android app starts by showing the launcher activity, and from there the app may make it possible to open additional activities.

Activity Lifecycle

When an Android app is first started the main activity is created. The activity then goes through 3 states before it is ready to serve the user: Created, started and resumed.



Lifecycle of an Android Activity

If the current activity opens any other activities (screens), these activities will go through the same 3 states when they are opened. For example, if an activity A opens another activity B, then activity A will be paused. This means that activity A goes into the paused state. When the user clicks the back button and returns to activity A, activity A returns to the resumed state.
If the user returns to the home screen of the Android device, all activities will be paused and then stopped. If the user then returns to the app, the activities will go through the started and then resumed states. If the Android device needs the memory that the Android app occupies in the device's memory, then it may completely destroy the app, meaning the Android activities go into the destroyed state.
The state of the activity always follows the arrows in the diagram**. An activity cannot jump from created to resumed directly.** An activity will always go through created, started and resumed in that sequence. An activity may change from resumed to paused and back, and from paused to stopped and back to started, but never from stopped directly to resumed.
We will study about the Activity as well as the Application lifecycle in detail in the upcoming lectures.

Creating an Activity

The Activity class inherits the “AppCompatActivity” base class. The “Compat” in its name stands for Compatibility, which means that the class supports its older features in the newer versions as well. The activity class does not have any constructors or an init() block of its own and hence, we cannot create an instance of the activity class by ourselves, this is handled by

the system itself and the programmer can only modify the contents of the activity instance with the help of callbakcs.
An Activity class can be created using the following steps:

1. Firstly, click on app > res > layout > Right Click on layout. After that Select New > Activity and choose your Activity as per requirement.
2. After that Customize the Activity in Android Studio. Enter the "Activity Name" and "Package name" in the Text box, check the checkbox to create a layout for the activity and Click on Finish button.

A new activity with the chose name will be created with the following code

```
class MainActivity : AppCompatActivity() {
      override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
      }
}
```

The function onCreate() is called when the Activity is created. **setContentView()** is used to set the UI layout to the Activity.
R is the resource class in Android which is used to access the resources within the code segment.

What is Context?

A Context is a handle to the system; it provides services like resolving resources, obtaining access to databases and preferences, and so on. An Android app has activities. Context is like a handle to the environment your application is currently running in. The activity object inherits the Context object.

Debugging

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.
Android Studio provides us with a Logcat and a Debugger to debug out code segments. The logcat contains the information about the current processing of the application and contains all the system generated logs. The Android Developer can also add custom logs based on their requirements to identify the errors or bugs within their code.

Log Class

The Log Class allows you to log messages categorized based severity; each type of logging message has its own message. Here is a listing of the message types, and their respective method calls, ordered from lowest to highest priority:

- The Log.v() method is used to log verbose messages.
- The Log.d() method is used to log debug messages.
- The Log.i() method is used to log informational messages.
- The Log.w() method is used to log warnings.
- The Log.e() method is used to log errors.
- The Log.wtf() method is used to log events that should never happen ("wtf" being an abbreviation for "What a Terrible Failure", of course). You can think of this method as the equivalent of Java's assert method.

One should always consider a message's type when assigning log messages to one of the six method calls, as this will allow you to filter your logcat output when appropriate. It is also important to understand when it is acceptable to compile log messages into your application:

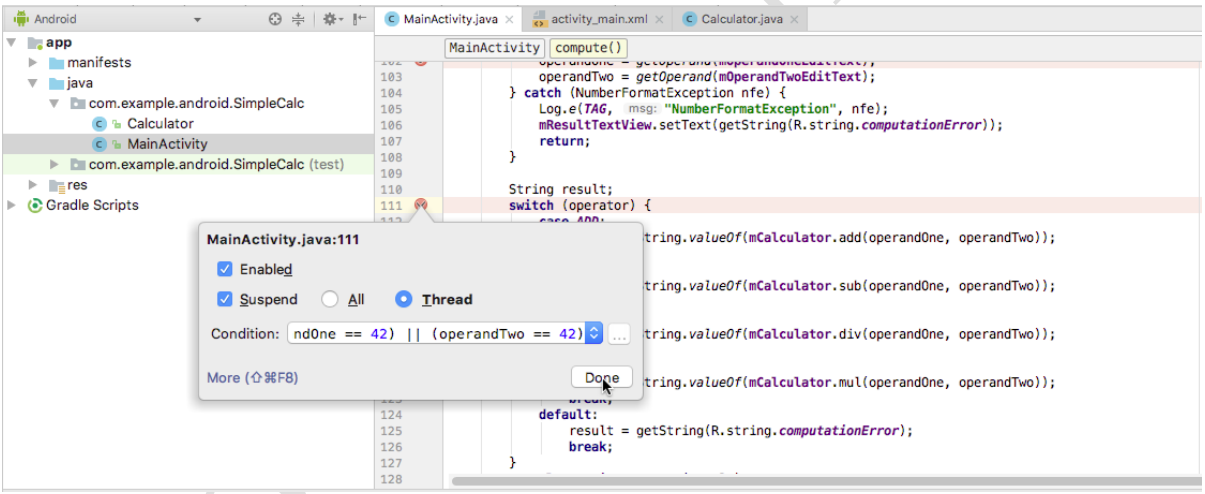
- **Verbose** logs should never be compiled into an application except during development. When development is complete and you are ready to release your application to the world, you should remove all verbose method calls either by commenting them out, or using ProGuard to remove any verbose log statements directly from the bytecode of your compiled JAR executable, as described in Christopher's answer in this StackOverflow post.
- **Debug** logs are compiled in but are ignored at runtime.
- **Error**, warning, and informational logs are always kept.

The following code segment explains the use of the Log class

```
Log.e("TAG","Content")
//Prints an Error Log with the tag "TAG" and log message "Content"
```

Debug Break Point

Android lets the users add a debug break point by placing the cursor on where the want to insert the break point and then selecting Run > Toggle Line Breakpoint. A breakpoint can be removed in the similar fashion. A red dot appears in front of a statement which contains a debug break point.



Debug Break Point

These breakpoints do not interfere when the app is run but when the app is in a debugging mode, the application halts its execution every time a breakpoint is encountered. A debugging window is then opened which allows the user to evaluate the values of all the variables at that point. The user is then provided with two options, Step Over and Step Under.

- Step over – An action to take in the debugger that will step over a given line. If the line contains a function the function will be executed and the result returned without debugging each line.
- Step into – An action to take in the debugger. If the line does not contain a function it behaves the same as "step over" but if it does the debugger will enter the called function and continue line-by-line debugging there.

Event Handling

An event in an Android Application can be defined as a user interaction, such as a click, a long click, a swipe, or a touch event. The events are handled by the system interface. The originate at the UI where an event is identified and the appropriate callback is invoked based on the type of the event. The following code segment can be used to set a callback for a click event on a button.

```
val clickListener = View.OnClickListener {view ->
      Toast.makeText(this@MainActivity,
            "Click event encountered",
            Toast.LENGHT_SHORT)
            .show()
}
val button = findViewByIf(R.id.button)
button.setOnClickListener(clickListener)
```

The following code displays a Toast (i.e. a small message) whenever the button is clicked. The makeToast function takesin three arguments, the context (the activity itself), the Text ("Click event encountered", and the time interval for which the toast is displayed.
The val "clickListener" can be replaced with a Lambda expression making the code simple and compact. This can be done as follows.

```
button.setOnClickListener{view ->
      Toast.makeText(this@MainActivity,
            "Click event encountered",
            Toast.LENGHT_SHORT)
            .show()
}
```

Intents

An Intent is a messaging object you can use to request an action from another app component. Although intents facilitate communication between components in several ways, there are three fundamental use cases such as, starting an activity, starting a service or delivering a broadcast. There are two types of intents:

- **Explicit intents** specify which application will satisfy the intent, by supplying either the target app's package name or a fully-qualified component class name. An explicit intent is typically used to start a component in the same app, because the class name of the activity or service is already known.

  ```
  val intent = Intent(this@MainActivity,
        SecondActivity::class.java)
  startActivity(intent)
  ```

  The intent instance for explicit intents has two parameters, the context of the application (in this case the current activity or the MainActivity) and the destination activity. Data can also be attached to the intent using the putExtra() function, this data can then be accessed in the destination activity (SecondActivity in this case). The first statement only creates the intent and the startActivity() function launches the intent and opens the next activity. The data can be attached as following:

```
val intent = Intent(this@MainActivity,
      SecondActivity::class.java).apply {
            putExtra("NAME_KEY","John Doe")
      }
startActivity(intent)
```

The function putExtra() has two parameters, the key and the content. The key-value pairs can than be accessed in the destination activity using the following code.

```
val name: String? = intent.getStringExtra("NAME_KEY")
```

- **Implicit intents** do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it.

```
val intent = Intent(Intent.ACTION_VIEW,
      Uri.parse("http://www.google.com"))
startActivity(intent)
```

The intent instance for implicit intents has two parameters, the action it needs to perform and the data the action requires. The data can also be set otherwise using the intent.setData() function. The first statement only creates the intent and the startActivity() function launches the intent i.e. opens the webpage with the specified link. If more than one application can perform the task, the user is asked to choose the activity.
The types of functions that an Activity can perform is specified within "Manifest.xml" as an intent filter. To read more about intent filters go to the following link:
https://developer.android.com/guide/components/intents-filters

Permissions

Android allows the applications to access the system resources as well but the application needs to gain permission from the Android user. Up until the API 23 version, the application would require the user to grant all its permissions at the time of installation, but this wasn't a secure practice as the user had no idea as to why the application needed those permissions. Post API 23, the applications use runtime permissions wherein they ask for permissions when the activity is up and running and the user can grant or deny those permissions as per their understanding. The following steps are to be followed when asking for user permissions:

1. Define the permission in the manifest. Some permissions such as internet permission need not be sked from the user and is given directly if defined in the manifest. This can be done as follows:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

2. Check for permissions. If the permission is granted, proceed with the task, else proceed with step 3. This can be done as follows:

```
if (ContextCompat.checkSelfPermission(thisActivity,
Manifest.permission.WRITE_CALENDAR)
      != PackageManager.PERMISSION_GRANTED) {
            // Permission is not granted
}
```

3. Define a Permission Rationale. This appears after the user has declined the permission once and the system asks for the same permission again, it is helpful to make the user understand of the permission requirements. Request for permission using the following code:

```
if(ActivityCompat.shouldShowRequestPermissionRationale
      (thisActivity, Manifest.permission.READ_CONTACTS)) {
   //Show a snackbar, toast or message explaining the use
   //of the permissions
} else {
    ActivityCompat.requestPermissions(thisActivity,
            arrayOf(Manifest.permission.READ_CONTACTS),
            MY_PERMISSIONS_REQUEST_READ_CONTACTS)
}
```

4. The granted permissions are the visible in the onPermissionRequestResult() callback. If the user has granted the permission, the tasks can be performed or a message can be displayed explaining the inability to perform the task. This can be done as follows:

```
override fun onRequestPermissionsResult(requestCode: Int,
    permissions: Array<String>, grantResults: IntArray) {
     when (requestCode) {
            MY_PERMISSIONS_REQUEST_READ_CONTACTS -> {
                    //If request is cancelled, the result arrays
                    //are empty.
                if((grantResults.isNotEmpty() && grantResults[0]
                    == PackageManager.PERMISSION_GRANTED)) {
                    //Permission granted, proceed with the tasks
                } else {
                    //Permission denied, Disable the
                    //functionality that depends on
                    //this permission.
                }
                return
            }
            else -> {
                    // Ignore all other requests.
            }
     }
}
```