Android Development with Kotlin

Asynchronous

Thread

A thread is a pipeline for sequential code execution. Multiple threads can be executed concurrently within an Application. This allows multiple tasks to be performed at the same time such as, fetching data from the server while running the screen animations at the same time. Android Applications consist of two types of threads, one that are tied to the Lifecycle of an Activity/Fragment and the ones that are not.

When an Android Application is launched, it creates the first thread of execution, known as the main or the UI thread. The main task of the UI thread is to draw the UI/layout onto the screen of the device. The devices that exist in the market at the present time have refresh rates as high as 60-120Hz, leaving only about 8-16ms for the main thread to redraw the UI, hence, to ensure high performance the main thread should only be tasked with drawing the UI. All the other tasks such as fetching data from the network, or modifying the database should be asynchronous. However, ***all the UI elements should only be accessed in the main thread as other threads are not bound to the UI and accessing them will result in an error.***

Tasks that aren't run on the same thread are known as asynchronous tasks, and the threads on which asynchronous tasks occur are knows as asynchronous threads. For example, if an Android Application needs to download an image from the network and a separate thread is created to accomplish this task, the network access thread is asynchronous to the main thread.

Lets consider the following code segment to fetch the HTML code from a URL and display it within a TextView.

```
download.setOnClicklistener{
      Thread(object: Runnable{
            override fun run(){
                  val data = getHTMLFromURL("www.google.com")
                  runOnUiThread {
                        textView.text = data
                  }
            }
      }).start()
}
```

The runnable object in the function call can also be replaced by a lambda function as the runnable interface is a single function interface. This can be done as follows:

```
download.setOnClicklistener{
      Thread {
            val data = getHTMLFromURL("www.google.com")
            runOnUiThread {
                  textView.text = data
            }
      }.start()
}
```

The constructor of the Thread class needs only one argument i.e. an object of the interface Runnable. The Runnable interface, has only one function known as run(). This function is

invoked when the thread is processed. Since the Runnable interface has only one function, ***the Runnable argument can be replaced with a lambda function during constructor invocation.***

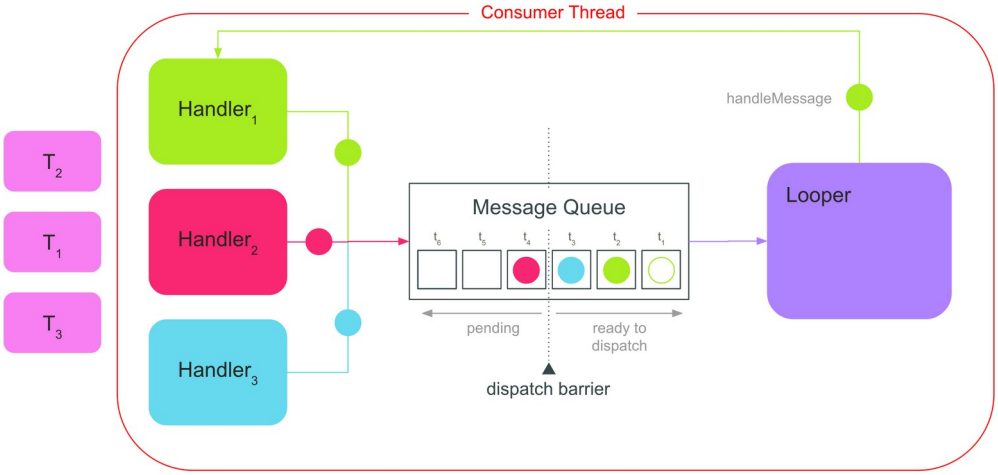
The runOnUiThread() function needs a runnable object as its arguments as well which is run on the UI thread as the name suggests. This can be used to establish a communication between the UI thread and the current thread.

The start() function when invoked starts running a thread. This function works only once and has no consequence on subsequent invocations as ***you can only start a thread once.***

If the network access does not occur in the background, the application gives a Run-time Exception.

HandlerThread

The main drawback of using a regular thread is that the thread runs only once and hence, multiple threads will be required to perform multiple functions. This can be rectified by using a thread that runs in a continuous loop, this can be achieved with the use of three additional components, a Looper, a MessageQueue and Handlers.



1. **Looper:** The Looper provides Thread an ability to keep on running in a loop and associates a MessageQueue for the thread on which this Looper object is created. When a Message/Runnable is posted, Looper executes those on its associated thread, and returns the results/handleMessage to the Handler associated with the message/runnable. Only one Looper can be associated with a thread.
2. **MessageQueue:** The MessageQueue maintains a queue of all the messages/runnable requests received via various handlers.
3. **Handler:** The Handler assigns jobs to the Looper. A Looper can be attached to multiple handlers at a time. Every time a Handler requests a job, it is added in the MessageQueue which is then processed by the Handler in the order of receiving the requests. The requests can be in the form of Runnables or Messages.

The Main Thread follows the same Lopper, MessageQueue, Handler architecture for its processing.

The Android System provides a HandlerThread Class that associates a Looper and MessageQueue with a thread so that the user developer doesn't have to implement this behaviour on their own. An object of the Handler class can be created as follows:

```
handlerThread = HandlerThread("TaskHandlerThread").also{ it.start() }
```

The constructor of the HandlerThread class requires the name of the Thread as its input. The thread needs to be started to inform the Looper to begin its loop. The thread must be stopped using the ".quit()" function in the onDestroy() function to quit the Thread processes once the activity is destroyed.

A Handler object needs to be associated with a looper object and has a few functions that can be overridden such as the handleMessage() function. The following code segments demonstrate the two ways of defining a handler, one without overriding the handleMessage() function and the other with it.

```
//without overriding the handleMessage Function
//usually used for Runnables
val runnableHandler = Handler(handlerThread.looper)


//overriding the handleMessage function
//used for Messages as well as Runnables
val messageHandler = object : Handler(handlerThread.looper) {
    override fun handleMessage(msg: Message) {
        if(msg.what == TOAST_MESSAGE) {
            runOnUiThread {
                Toast.makeText(this@MainActivity,
                  "Message to Handler", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

The "what" property for a message is an integer that is used to identify the type of message. The Handlers are used with Runnables when the only job is to run a code segment on a separate thread, if a complex processing is to be accomplished, Messages are a better fit.

To send a message to the handler, the following code can be used:

```
messageHandler.sendEmptyMessage(TOAST_MESSAGE)
```

To send a runnable, post() function is used instead. This can be replaced with the postDelayed() to perform the action after a delay of the specified milliseconds. "post()" takes in only one argument i.e. the runnable object whereas "postDelayed" requires two arguments, the runnable object as well as the number of milliseconds by which the post request is to be delayed. The following code segment illustrates the use of both of these functions.

```
runnableHandler.post{
    runOnUiThread {
        Toast.makeText(this, "Runnable to Handler",
            Toast.LENGTH_SHORT).show()
    }
}


runnableHandler.postDelayed({
    runOnUiThread {
        Toast.makeText(this, "Runnable to Handler: 1000ms Delay",
            Toast.LENGTH_SHORT).show()
    }
}, 1000)
```

The runOnUiThread function basically runs the runnable using a handler attached to the Looper of the Main Thread, the same results can be achieved by the following code segment.

```
val uiHandler = Handler(Looper.getMainLooper())
uiHandler.postDelayed({
    Toast.makeText(this@MainActivity, "Delayed to UI : 1000ms",
      Toast.LENGTH_SHORT).show()
}, 1000)
```

AsyncTask

Multi-threading is used to optimize the performance of an Android Application. The background threads are mainly used to modify or fetch data from databases and the UI thread is tasked with reflecting those changes. Hence, a communication between the two threads is necessary. AsyncTask is an Android Jetpack class that creates a new background thread for a task which can communicate with the main thread with ease.

An AsyncTask in a generic class with three data-types, namely, the input data-type, the progress data-type and the output data-type.

The following example will demonstrate how to fetch data from a website and publish its HTML code on the Text View.

1. Add the permission for internet access in the manifest file for your application, this can be done as follows:

```
<manifest  xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.test">
   <uses-permission android:name="android.permission.INTERNET"/>
   ...
</manifest>
```

2. Create an inner class within the Activity Class that extends the AsyncTask class and overrides the *doInBackground()*, *onProgressUpdate()* and *onPostExecute()* functions.

```
inner class Task : AsyncTask<String, Int, String>() {
    override fun doInBackground(vararg params: String?): String {
        publishProgress(0)
        val urlConnection = URL(params[0]).openConnection()
                                as HttpURLConnection
        publishProgress(10)
        try {
            publishProgress(100)
            return
                urlConnection.inputStream.bufferedReader().readText()
        } finally {
            urlConnection.disconnect()
        }
    }


override fun onProgressUpdate(vararg values: Int?) {
        super.onProgressUpdate(*values)
        printToast("Progress: ${values[0]}%")
    }

    override fun onPostExecute(result: String?) {
        super.onPostExecute(result)
        textView.text = result
    }
}
```

The *doInBackground* function runs on the background thread and is used to fetch data from the website. The *onProgressUpdate* function can be called from the doInBackground function by invoking "*publishProgress*", this allows us to update the UI to show progress. The o*nPostExecute* function is used to display the results from the website onto our UI.

The printToast function exists within the Activity Class and contains the following code segment

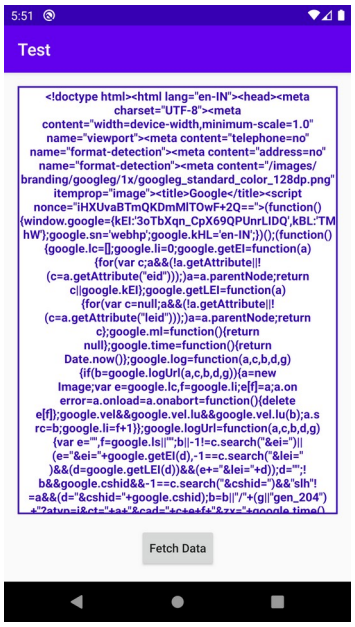
```
fun printToast(s: String) {
    Toast.makeText(this, s, Toast.LENGTH_SHORT).show()
}
```

3. Execute an object of the Task class within the MainActivity.

```
fetchTextButton.setOnClickListener {
    Task().execute("https://www.google.com/")
}
```

The task is executed on button click as per this example but this code segment can be used anywhere within the application.

The final result is

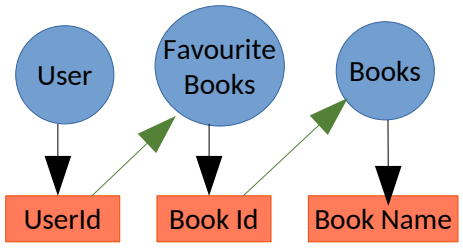


The class that extends the AsyncTask needs to be an inner class so that it can access the functions as well as UI elements of the Activity class itself. But if the Activity is recreated due to orientation change or any other reasons during the execution of the AsyncTask, the executing task still holds an instance of the older Activity. This is known as memory leak, and is one of the reasons why AsyncTask has now been deprecated.

CallBack Hell & Coroutines

Callback Hell is a phenomenon that occurs when multiple accesses to the database/network are required to fetch the information. We can better understand this with an example.

Suppose, you have a two entity database consisting of Users and Books, and a relationship between the two i.e. favourite books. This database can be represented in the form of a three table architecture with each table representing the Users, the Books and the favourite relationship. To access the favourite books of a user, you will first have to access the id of the user from the User table, find an entry corresponding to the user id in the favourites table, and then use the books id from the fetched entry to find the name of the book. This can be visualised as follows:



Fetch the name of the Favourite book for a user

In this case, the three tables cannot be accessed asynchronously and should be accessed in a sequential manner and is often referred to as a callback hell, because you will need multiple callbacks to implement the above mentioned task.

Another way of approaching such problems is the use of **Coroutines.**

Coroutines allow both sequential as well as asynchronous network/database access, and can be used in the Kotlin Application using a lifeCycleScope Library along with the "suspend" keyword.

Suspend Keyword

The suspend keyword when used within a function suspends the code segment followed by an async request until the request returns a response. This allows the code to run sequentially even when a few tasks are run on the background thread.

Network Access using Sequential Coroutines

The following example demonstrates how to access data from a network sequentially using Coroutines.

In this example, the application will sequentially access the HTML code from "www.google.com" and "www.facebook.com" and display it on two different textviews.

1. Create an Activity with two textViews, textView1 and textView2, and a button to begin the fetching mechanism.
2. Add the lifecycle scope dependency to your project by adding the following code segment to the build.gradle file

```
dependencies {
    implementation "androidx.lifecycle:lifecycle-runtime-ktx:2.2.0-alpha01"
}
```

3. Create a fetchData function with the suspend keyword.

```
private suspend fun fetchData() {
    val googleData = withContext(Dispatchers.IO) {
        fetchDataFromURL("https://www.google.com")
    }
    textView1.text = googleData
    val facebookData = withContext(Dispatchers.IO) {
        fetchDataFromURL("https://www.facebook.com")
    }
    textView2.text = facebookData
}
```

4. Call the function within the **lifecycleScope.launch{ }**. This ensures that the fetching process is terminated with the end of the lifecycle of the lifecycle owner. This is done as follows:

```
fetchTextButton.setOnClickListener {
    lifecycleScope.launch {
        fetchData()
    }
}
```

In the above example, Facebook Data and Google Data are independent of each other and async access can work as well. This will help save time and make the application more efficient.

This can be achieved by making the following changes to the fetchData function in the above mentioned code segment.

```
private suspend fun fetchData() {
    val googleData = lifecycleScope.async {
        withContext(Dispatchers.IO) {
            fetchDataFromURL("https://www.google.com")
        }
    }
    textView1.text = googleData.await()
    val facebookData = lifecycleScope.async {
        withContext(Dispatchers.IO) {
            fetchDataFromURL("https://www.facebook.com")
        }
    }
    textView2.text = facebookData.await()
}
```

Here, the function returns Deferred data-type rather than the actual value of the String. This Deferred data type is then used to extract the string using *.await()* function. This function is only called once the network returns the data and assigns value to the deferred type.

If an exception occurs during the sequential access, it arises at the point where the value is being assigned to the data type but in case of async access, the exception arises when the .await() function is called.