Val user = fetchUserData() -> error NetworkOnMainThreadException also it wil block our

textView.text = user.name UI thread too

***Therefore then we will think of this solution***

Thread {

Val user = fetchUserData()

textView.text = user.name -> Now error will be that this can’t be done on the

background thread

}

***We can handle the above in threads too easily by using the Async class But what if there are 1 lakhs of threads on our system running simultaneously. It would make our OS extremely slow and our app may crash too. Therefore the better solution we will think of will be call backs in kotlin.***

Fun requestToken() : Token { fun requestTokenAsync(cb: (Token) -> Unit){

//makes a request for a token and wait 🡪 //makes a request for a token

Return token //invokes callback when done

} //return immediatel

}

***Fun loadUser(){***

***Api.fetchUser { user ->***

***Show(user)***

***}***

***}***

***Call backs are very complicated. Therefore instead of them we use corutines.***

***COROUTINES***

Coroutines are nothing but they make the task of Async programming very easy.

First of all to use this we need to add the libriary.

Therefore coroutines has only 2 main mechanisms -> one is suspend and the other is resume. When the function is in suspend state , the coroutines can be assumes to be paused. It is because the suspend function doesn’t occur on the main thread and as soon as it brings back the result . it gives it to the resume function

You can call suspend functions only from other suspend functions or by using a coroutine builder such as launch to [start a new coroutine](https://developer.android.com/kotlin/coroutines#start). Suspend pauses the execution of the current coroutine, saving all local variables. Resume continues execution of a suspended coroutine from the place where it was suspended. E.g -



MECHANISM OF ABOVE COROUTINE

//Here the default dispature in the mainThread. Here api.fetchUser is also a suspend function. Therefore when this suspend functions is executed here. The loadUser function comes to a pause state till the fetchUser function is completely executed. When this fetchUser Function is executed, though the loadUser function is in pause state but still the activities which are running on the UI thread(or the thread of the dispature which we have mentioned globally), Therefore without using another thread we can perform task asynchronously here. When the fetchUser is completed, the loadUser function gets Resumed and it comes back to the thread on which it earlier was. Therefore note when fetchUser function is executed (when loadUser is in the pause state) , since then too all UI is working if the default discpature is main thread. This doesn’t mean it will run the suspend function on another thread, it just suspends that function for a while

***NOTE -*** Kotlin coroutines use *dispatchers* to determine which threads are used for coroutine execution. In Kotlin, all coroutines must run in a dispatcher, even when they're running on the main thread. Coroutines can suspend(come is the pause state as explained avove) themselves, and the dispatcher is responsible for resuming them.

To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

* **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions(here too the suspend function comes in the pause state as explained) , running Android UI framework operations, and updating [LiveData](https://developer.android.com/topic/libraries/architecture/livedata) objects
* **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. **Examples include using the**[**Room component**](https://developer.android.com/topic/libraries/architecture/room)**, reading from or writing to files, and running any network operations.** Therefore use this dispatcher when you need to perform newtwork operations.
* **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work outside of the main thread**. Example use cases include sorting a list and parsing JSON.** Therefore use this dispature when you need to perform a CPU intensive work. Since it will not make your main Thread slow.

***Note – We can even mention the default Dispacture for a particular activity. i.e it would consider this dispature if no dispature is mentioned in the suspend function or while defining tha launcher.***

***One more e.g. –***

We can easily see that the fetchDocs occur on the mainThread. From this if we call another suspend function get() , then this fetchDocs function gets paused like explained above, but the other UI work happening on the mainThread can happen without getting stacked. Till then this getFunction executes on the backgroundThread as we have mentioned the dispature for that and on the completion of this function, fetchDocs() function gets resumed.

suspend fun fetchDocs() {                      // Dispatchers.Main  
    val result = get("developer.android.com")  // Dispatchers.Main  
    show(result)                               // Dispatchers.Main  
}  
  
suspend fun get(url: String) =                 // Dispatchers.Main  
    withContext(Dispatchers.IO) {              // Dispatchers.IO (main-safety block)  
        /\* perform network IO here \*/          // Dispatchers.IO (main-safety block)  
    }                                          // Dispatchers.Main  
}

***Start a coroutine***

You can start a coroutine in one of two ways:

* **launch** starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.
* **async** starts a new coroutine and allows you to return a result with a suspend function called await.

Typically, you should launch a new coroutine from a regular function, as a regular function cannot call await. Use async only when inside another coroutine or when inside a suspend function and performing [parallel decomposition](https://developer.android.com/kotlin/coroutines#parallel).

***Note- we should also finish the scope of coroutines after ther are used as they don’t get finish on their own.***

***Now I would show you How to fetch data from the Website using okHTTP and Gson (using coroutines) which is better way to do it than using callback which I have don in the project. The site from where we will fetch Data is -*** [**https://jsonplaceholder.typicode.com/users**](https://jsonplaceholder.typicode.com/users)

***Here too we will bring the data from the API when the button is clicked –***

//These are data class in whose objects we will store the data // we fetch from the net

**data class** User(  
 **val** id:Int,  
 **val** name:String,  
 **val** username:String,  
 **val** email:String,  
 **val** address:Address  
)

**data class** Address(  
 **val** city:String,  
 **val** street:String  
)

*//Therefore our default dispatcher is the main dispatcher*

**val** supervisor = SupervisorJob()  
**override val** coroutineContext: CoroutineContext  
 **get**() = Dispatchers.Main + supervisor

button.setOnClickListener **{***//getUsers("https://jsonplaceholder.typicode.com/users"). Here we can’t make an http //call as the network operations can’t happen on the mainThread.*

*//we can simply write this getUser here as we want to call another suspend function //getUsers which will happen on another thread. Since we can call another suspend function from either inside a coroutine or suspend function. Therefore we created a coroutine here.* launch **{**

**val** deferredUsers=getUsersSmart(**"https://jsonplaceholder.typicode.com/users"**)

*//when we will wait for the result the above function would get executed and*

*//then the writing the below statement the users would be converted to the*

*//list of type “User” objects*  
 **val** users = deferredUsers.await()

*// Now we can set the above list in the adapter we want*  
 **}**

**}**

**fun** getUsersSmart(url:String):Deferred<List<User>>  
{

*// as we have discussed above async is also used to start a suspend function. When*

*//we want to return from the suspend function we start the suspend function using*

*// keyword async and then wait for the*

**return** async(Dispatchers.IO) **{  
 val** client = OkHttpClient()  
 **val** request = Request.Builder()  
 .url(url)  
 .build()  
  
 **val** data = client.newCall(request).execute()  
 **val** jsondata = data.body()?.string()?:**""**

**val** users = parseJson(jsondata) *// this statement will pass the string data to*

*// the parseJson function*  
 users  
 **}**}

**fun** parseJson(data:String):List<User>{**val** gson = Gson()  
 **val** users = gson.fromJson(data,Array<User>::**class**.java)  
 **return** users.toList()  
 }

***Now you can refer to the retrofit project where I have used the coroutines too.***