Fresher Android

Threading Networking Basic

Lesson Objectives

#1. Asynchronous programming

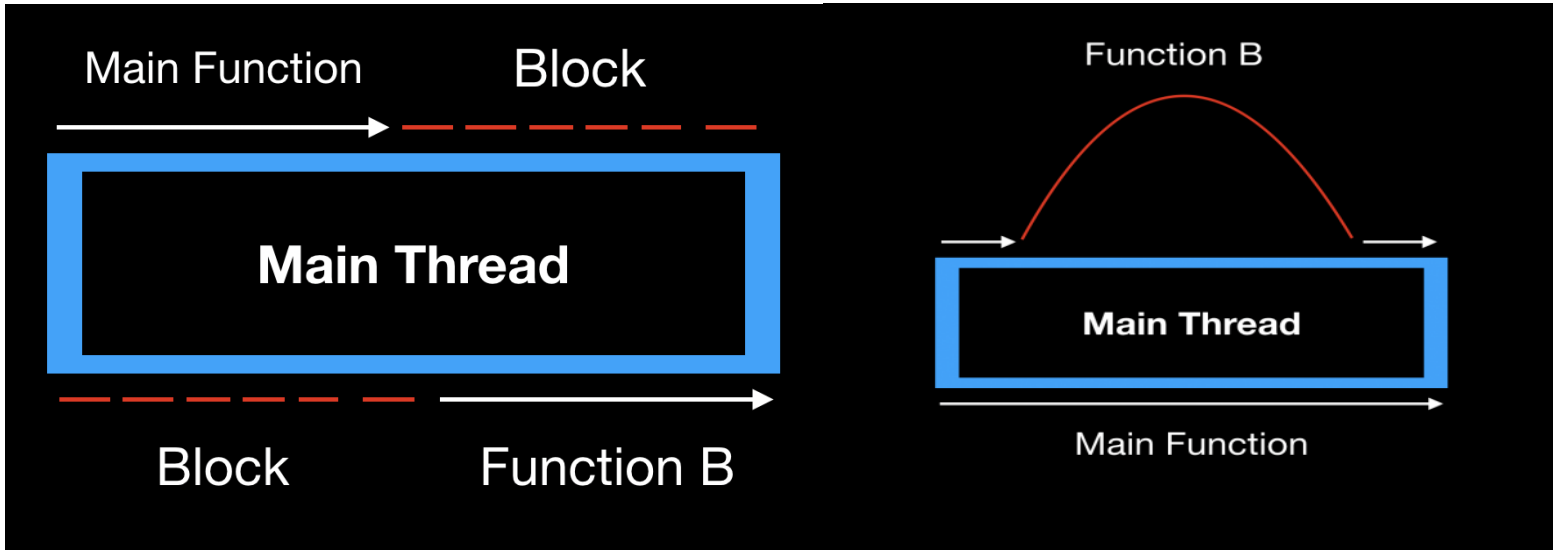
Asynchronous programming is very important for modern applications. Using it increases amount of work your app can perform in parallel. This in turn allows you to run heavy-duty tasks away from the **UI thread**, in the background. By doing so, you avoid UI freezes, and provide a fluid experience for your users.

#2. Why Use Coroutines?

- As you may know, Android developers today have many async tools at hand. These include **RxJava/Kotlin/Android**, **AsyncTasks**, **Jobs**, **Threads**, and more
- If you've worked with Rx, then you know it takes a lot of effort to get to know it enough, to be able to use it safely. On the other hand, AsyncTasks and Threads can easily introduce leaks and memory overhead. Finally, relying on all these APIs, which use callbacks, can introduce a ton of code. Not only that, but the code can become unreadable, as you introduce more callbacks.

#3. Introduction to Coroutines

- **Suspending vs. blocking**



#3. Introduction to Coroutines

- A blocking call to a function means that a call to any other function, from the same thread, will halt the parent's execution. Following up, this means that if you make a blocking call on the main thread's execution, you effectively freeze the UI. Until that blocking calls finishes, the user will see a static screen, which is not a good thing.
- On the other hand, suspending doesn't necessarily block your parent function's execution. If you call a suspending function in some thread, you can easily push that function to a different thread. In case it is a heavy operation, it won't block the main thread. Moreover, if you require a result from the function, you can bridge back to the main thread, without a lot of code. That way you can fetch data in a coroutine, from the main thread. All you have to do is launch the coroutine in a worker thread. This way you'll effectively call something from the main thread, switch to the background, and switch back once the data is ready.

#4. Coroutine Terminology

- Kotlin Coroutines give you an API to write your asynchronous code sequentially
- **Suspending functions**: This kind of function can be suspended without blocking the current thread. Instead of returning a simple value, it also knows in which context the caller suspended it. Using this, it can resume appropriately, when ready.
- **CoroutineBuilders**: These take a suspending lambda as an argument to create a coroutine. There are a bunch of coroutine builders provided by Kotlin Coroutines, including async(), launch(), runBlocking.

#4. Coroutine Terminology

- CoroutineScope: Helps to define the lifecycle of Kotlin Coroutines. It can be application-wide or bound to a component like the Android Activity. You have to use a scope to start a coroutine.
- CoroutineDispatcher: Defines thread pools to launch your Kotlin Coroutines in. This could be the background thread pool, main thread or even your custom thread pool. You'll use this to switch between, and return results from, threads

#5. Adding Kotlin Coroutines support

- implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.2'
  implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.2'

#6. Coroutine Builders

- **runBlocking**
  - ✓ This builder blocks the current thread until all the tasks inside that coroutine are finished.
- **launch**
  - ✓ This is the main builder. You'll use it a lot because it's **the simplest way to create coroutines**. As opposed to runBlocking, it won't block the current thread (if we use the proper dispatchers, of course).
- **async**
  - ✓ async allows **running several background tasks in parallel**

#7. Coroutine Jobs

- private val parentJob = Job()
- launch returns a Job, which is another class that implements CoroutineContext.
- **job.join :** With this function, you can **block the coroutine associated with that job until all the child jobs are finished**

```
val job = GlobalScope.launch(Dispatchers.Main) {
    doCoroutineTask()
    val res1 = suspendingTask1()
    val res2 = suspendingTask2()
    process(res1, res2)
}
job.join()
```

- **job.cancel:** This function will cancel all its associated child jobs. So if, for instance, the suspendingTask1() is running when cancel() is called, this won't return the value to res1 and suspendingTask2() will never be executed:

#8. async

- async allows **running several background tasks in parallel**. It's not a suspending function itself, so when we run async the background process starts, but it immediately continues running the next line. async always needs to be called inside another coroutine, and it returns a specialized job that is called Deferred.

```
GlobalScope.launch(Dispatchers.Main) {
    val user = withContext(Dispatchers.IO) { userService.doLogin(username,     password) }
    val currentFriends = withContext(Dispatchers.IO) {
    userService.requestCurrentFriends(user) }
    val suggestedFriends = withContext(Dispatchers.IO) {
    userService.requestSuggestedFriends(user) }
    val finalUser = user.copy(friends = currentFriends + suggestedFriends)
```

#8. async

```
GlobalScope.launch(Dispatchers.Main) {
    val user = withContext(Dispatchers.IO) { userService.doLogin(username, password) }
    val currentFriends = async(Dispatchers.IO) { userService.requestCurrentFriends(user) }
    val suggestedFriends = async(Dispatchers.IO) {
userService.requestSuggestedFriends(user) }
    val finalUser = user.copy(friends = currentFriends.await() + suggestedFriends.await())
}
```

#9. Using Dispatchers With Kotlin Coroutines

- You can use these dispatchers for the following use cases:
- **Dispatchers.Default**: CPU-intensive work, such as sorting large lists, doing complex calculations and similar. A shared pool of threads on the JVM backs it.
- **Dispatchers.IO**: networking or reading and writing from files. In short – any input and output, as the name states
- **Dispatchers.Main**: recommended dispatcher for performing UI-related events. For example, showing lists in a RecyclerView, updating Views and so on.

#10. Scoping Kotlin Coroutines

Global scope: It's a general scope that **can be used for any coroutines that are meant to continue executing while the App is running**

Implement CoroutineScope

- This set of elements can have information about:
- **Dispatchers**, which dispatch coroutines in a particular thread pool and executor.
- **CoroutineExceptionHandler**, which let you handle thrown exceptions.
- **Parent Job**, which you can use to cancel all Kotlin Coroutines within the scope.

#10. Scoping Kotlin Coroutines

```
class MainActivity : AppCompatActivity(), CoroutineScope {
override val coroutineContext: CoroutineContext
        get() = Dispatchers.Main + job
private lateinit var job: Job
}
```

- We create the job as lateinit so that we can later initialize it in onCreate. It will then be canceled in onDestroy.

#11. Internal Workings of Coroutines

- Internally, Kotlin Coroutines use the concept of **Continuation-Passing Style** programming, also known as CPS. This style of programming involves passing the control flow of the program as an argument to functions. This argument, in Kotlin Coroutines' world, is known as **Continuation**.

#12. Handling Exceptions

Exception handling in Kotlin Coroutines behaves differently depending on the **CoroutineBuilder** you are using. The exception may get propagated automatically or it may get deferred till the consumer consumes the result.

Here's how exceptions behave for the builders you used in your code and how to handle them:

- **launch**: The exception propagates to the parent and will fail your coroutine parent-child hierarchy. This will throw an exception in the coroutine thread immediately. You can avoid these exceptions with try/catch blocks, or a custom exception handler.
- **async**: You defer exceptions until you consume the result for the async block. That means if you forgot or did not consume the result of the async block, through await(), you may not get an exception at all! The coroutine will bury it, and your app will be fine. If you want to avoid exceptions from await(), use a try/catch block either on the await() call, or within async().

#13. Extra – Convert callbacks to coroutines

```
suspend fun suspendAsyncLogin(username: String, password: String): User =
suspendCancellableCoroutine { continuation ->
userService.doLoginAsync(username, password) { user ->
continuation.resume(user)
}
```

#14. Coroutine Flow & Channel

#15. Cleaning Up

```
override fun onDestroy() {
        super.onDestroy()
        parentJob.cancel()
}
```

Assignment

- Read bellow series in sequnce
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-1-gioi-thieu-kotlin-coroutine-va-ky-thuat-lap-trinh-bat-dong-bo-gGJ59xajlX2
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-2-build-first-coroutine-with-kotlin-Do7544Ee5M6
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-3-coroutine-context-va-dispatcher-Qbq5QkDzZD8
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-4-job-join-cancellation-and-timeouts-Do75463QZM6
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-5-async-await-naQZRxGm5vx
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-6-coroutine-scope-aWj536n8l6m
  - ✓ https://viblo.asia/p/cung-hoc-kotlin-coroutine-phan-7-xu-ly-exception-trong-coroutine-supervision-job-supervision-scope-naQZRDaG5vx
- Make sample app using coroutine, explain your function and concept use

Lesson Summary

- Asynchronous programming
- Why Use Coroutines?
- Introduction to Coroutines
- Introduction to Coroutines
- Coroutine Terminology
- Adding Kotlin Coroutines support
- Coroutine Builders
- Coroutine Jobs
- async
- Using Dispatchers With Kotlin Coroutines
- Scoping Kotlin Coroutines
- Internal Workings of Coroutines
- Handling Exceptions
- Extra – Convert callbacks to coroutines
- Coroutine Flow & Channel
- Cleaning Up

Thank you