Kotlin Coroutines(Part — 1): The Basics

Aditi Katiyar · Follow

Published in DeHaat · 4 min read · Jul 30, 2021

Getting started with Coroutines in Kotlin for Android Apps.

Photo by Pixabay from Pexels

Asynchronous programming is inevitable while developing Android apps. We do some tasks on a background thread and publish the results on the UI thread. For this, we usually implement 'AsyncTask'.

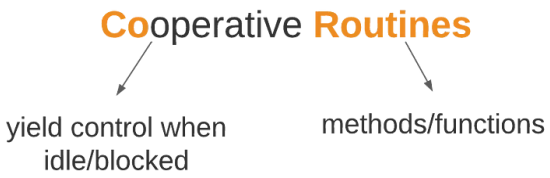
With Kotlin Coroutines, we no longer need to implement the hefty callbacks. Coroutines help us to write async code in a sequential and imperative fashion, hence it becomes easier to write tests and debug. They also optimize app performance by spawning minimum possible threads(Multiple coroutines can run on a single thread!).

What are coroutines?

Wikipedia says -

> “Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.”

Here, **subroutine** means a block of code(function or method) that performs a task, and **non-preemptive** means OS does not initiate context switch. The process itself yields control periodically or when idle or logically blocked. (This is known as **cooperation.**)

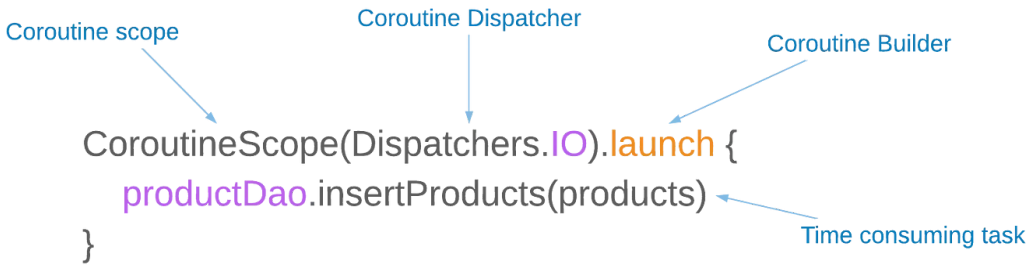


Coroutines vs Threads

1. Coroutines are cooperatively multitasked. While threads are preemptively multitasked.
2. Coroutines provide concurrency but no parallelism — only one is executing at a time on a thread(just like subroutines/functions — they just pass the baton among each other). On the other hand, multiple threads can run simultaneously.
3. In the case of coroutines, there are no system calls, no need for mutexes & semaphores, and no critical-section problems. Whereas for threads, the OS does context switching among them, and it needs to deal with critical section problems using mutexes & semaphores.

Building Blocks of a Coroutine

Let’s look at an example coroutine...



This is a coroutine to run a Room database query on a background thread(viz, *Dispatcher.IO*)

Coroutine Dispatcher

It defines what thread to use to run the coroutine. e.g.

- Main — main thread (run UI operations, update LiveData)
- IO — background thread (disk read/write, networking)
- Default — a shared pool of background threads(for CPU intensive tasks)

Coroutine Builder

It creates and starts a coroutine.

- *launch* — simply creates a coroutine and starts it. It does not return any result on completion.
- *async* — it starts a coroutine and returns a result after completion.
- *runBlocking* — it blocks the underlying thread. Generally used for writing unit tests.

Coroutine Scope

It defines the lifecycle/lifetime of a coroutine. e.g.

- CoroutineScope — helps to create a custom scope. Coroutines started in this scope need to be canceled manually.
- *viewModelScope* — bound to the lifetime of a ViewModel. When the ViewModel clears, the coroutines in this scope will also cancel.
- *lifecycleScope* — bound to the lifetime of a LifecycleOwner (Fragment/Activity). When the Fragment/Activity destroys, the coroutines in this scope will also cancel.

Illustrative Examples

1. Do some tasks in the background

```
class DoggoRepository @Inject constructor(
    private val dao: DogsDao
) {

    fun saveInDB(list: List<Dog>) {
        CoroutineScope(Dispatchers.IO).launch {
            dao.saveDoggos(dogs)
        }
    }
}
```

Here, we are saving a list in the database. We created a coroutine using the *CoroutineScope*. We dispatched it on the background thread(*Dispatchers.IO*). Also, note that we have used *launch* because we don't want to return any result after completing this operation.

2. Load data from a database

```
class DoggoViewModel @Inject constructor(
    private val dao: DogsDao
) : ViewModel() {
    private val doggos = MutableLiveData<List<Dog>>()

    fun loadDoggos() {
        viewModelScope.launch {
            doggos.value = dao.getDoggos()
        }
    }

    fun getDoggos(): LiveData<List<Dog>> = doggos
}
```

Here, the coroutine is bound to *viewModelScope*'s lifetime. The coroutine will be canceled(if it is running) when the ViewModel clears. Since we do not wish to return anything after the execution, we used *launch* coroutine builder. Also, we didn't pass any dispatcher because *viewModelScope* implicitly starts a coroutine on the Main thread.

Note that the function '*getDoggos()*' is a suspending function. A **suspend** function executes a long-running task and forces the coroutine to wait for its completion. We'll learn more about it in the future.

3. Do expensive calculations and return a result

```
class DoggoViewModel @Inject constructor() : ViewModel() {
    private val fibonacciNumber = MutableLiveData<BigInteger>()

    fun loadNthFibonacciNumber(n: Int) {
        val fibonacciDeferred: Deferred<BigInteger> =
            CoroutineScope(Dispatchers.Default).async {
                var a = BigInteger.ONE
                var b = BigInteger.ONE
                repeat(n) {
                    val sum = a + b
                    a = b
                    b = sum
                }
                return@async b
            }
        viewModelScope.launch {
            fibonacciNumber.value = fibonacciDeferred.await()
        }
    }
```

```
    fun getFibonacciNumber(): LiveData<BigInteger> = fibonacciNumber
}
```

Let's dig into the concepts one by one. We are doing a time-taking

tasks(*Dispatchers.Default*).

Since we have to return the result after calculations, we used the *async* coroutine builder(*return@async b*). The *async* returns a *Deferred<T>* object. A *Deferred* encapsulates an operation that will be finished at some point in the future after its initialization.

After creating the *Deferred* object, we are calling *await()* on it. It means that we are waiting for its result and assigning it to the LiveData — *fibonacciNumber*. We can call *await()* only from a coroutine scope or a 'suspend' function. Thus, we are doing this operation in a coroutine started in *viewModelScope*.

To learn more about coroutines, check these out:

- Kotlin Coroutines(Part — 2): The 'Suspend' Function
- Kotlin Coroutines(Part — 3): Coroutine Context
- Kotlin Coroutines(Part — 4): Cancellation
- Kotlin Coroutines(Part — 5): Exception Handling

Thanks for reading! If you liked it, do hit the clap button! Keep learning!

Kotlin Coroutines | Android | Android App Development | AndroidDev | Multitasking