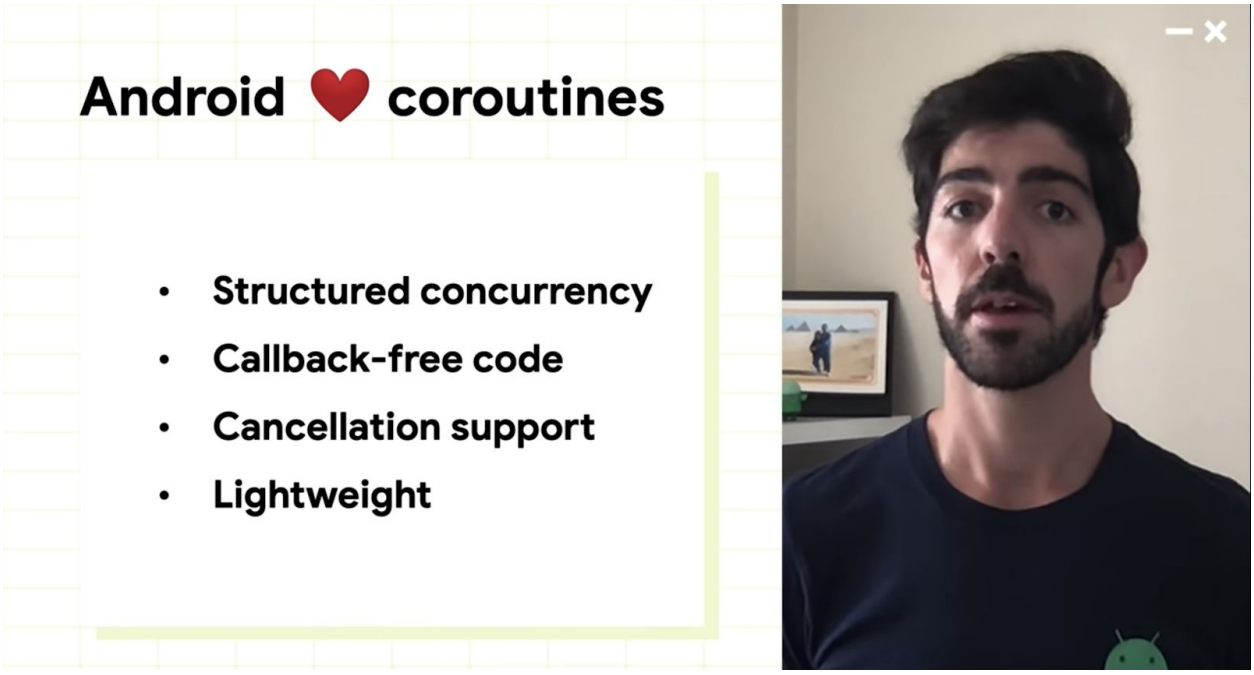
1. The alternative of AsyncTask
    a. Finally deprecated in Android 11!

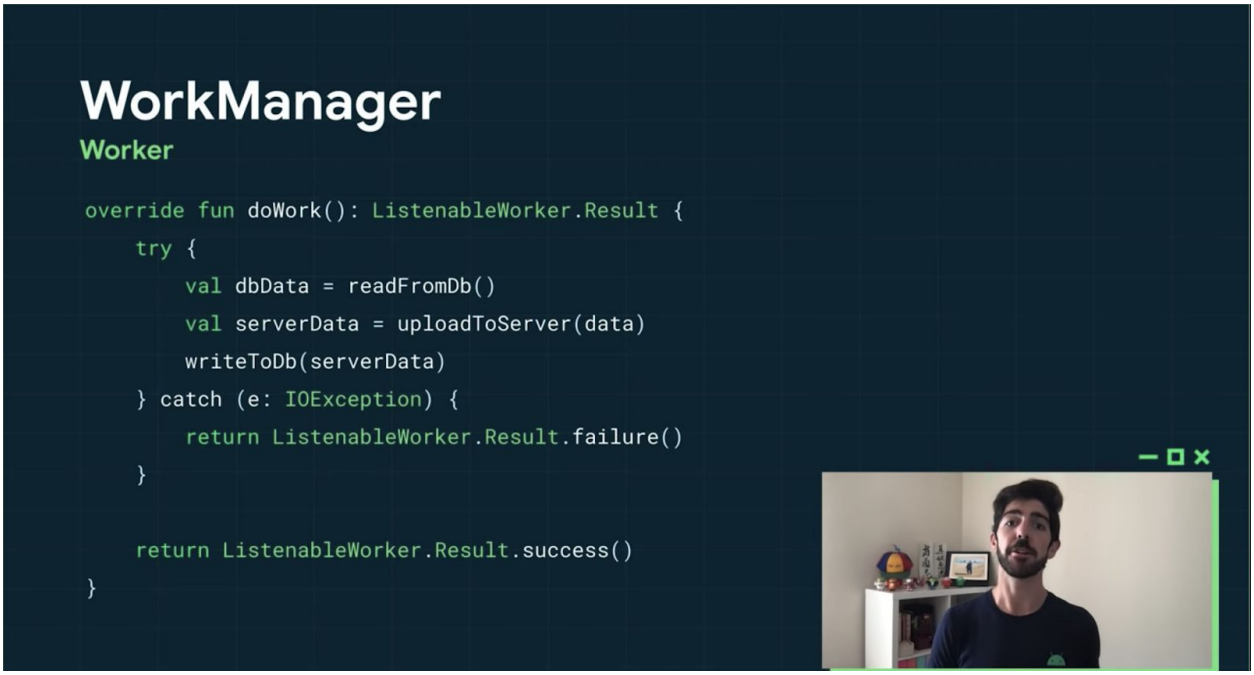
    b. Android official document says to use Concurrent Class of Java or Kotlin concurrency utilities. (Kotlin Coroutines)



    c. Why coroutines?

i. The unique properties of coroutines -> make it apps well-suited
ii. Structured Concurrency
    - Helps developers scope their works to application's component and prevent memory leaks
iii. Callback-free code
    - Higher readability and understandability
iv. Cancellation Support and natural exception handling
v. Lightweight
    - Many coroutines run on the single thread, suspending their work instead of blocking
    - Making them fast and lightweight

2. Practical Examples



a. WorkManager with synchronous job
    - But we can't stop the work at any moment.

```
override fun doWork(): ListenableWorker.Result {
    try {
        val dbData = readFromDb()
        if (isStopped()) return ListenableWorker.Result.retry()
        val serverData = uploadToServer(data)
        if (isStopped()) return ListenableWorker.Result.retry()
        writeToDb(serverData)
    } catch (e: IOException) {
        return ListenableWorker.Result.failure()
    }

    return ListenableWorker.Result.success()
}
```

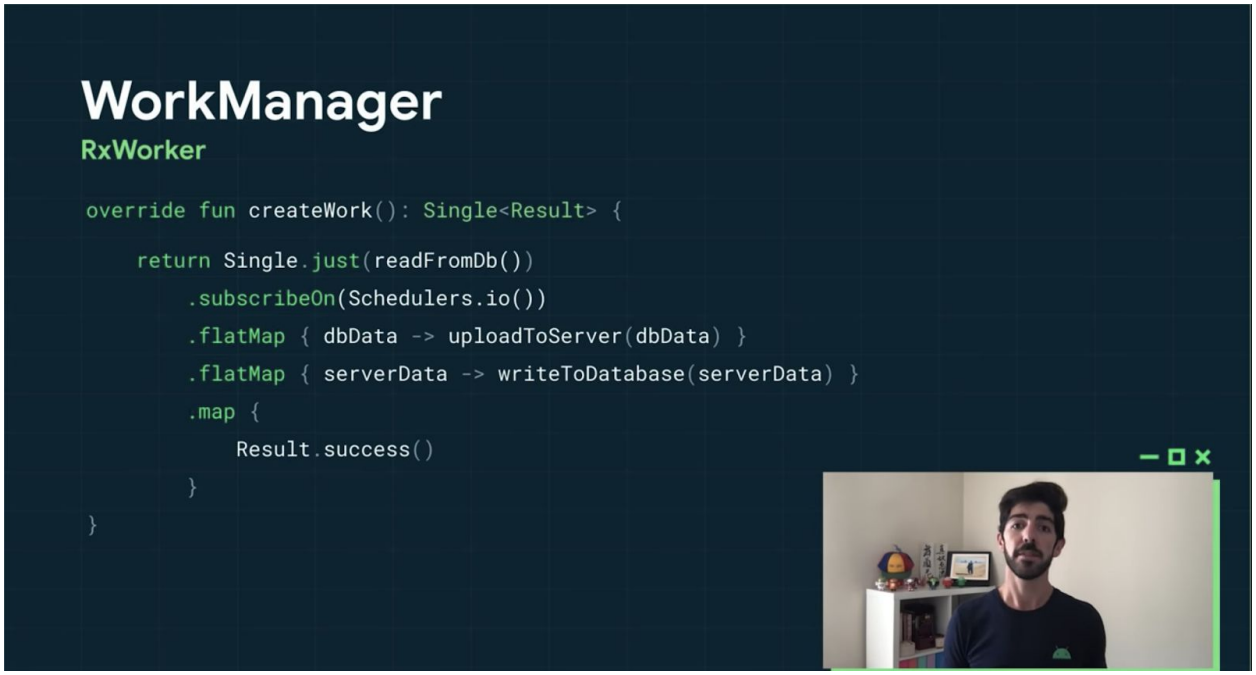
b. WorkManager with code handling ‘stopped case’
   i. But how to pass the signal to stop?

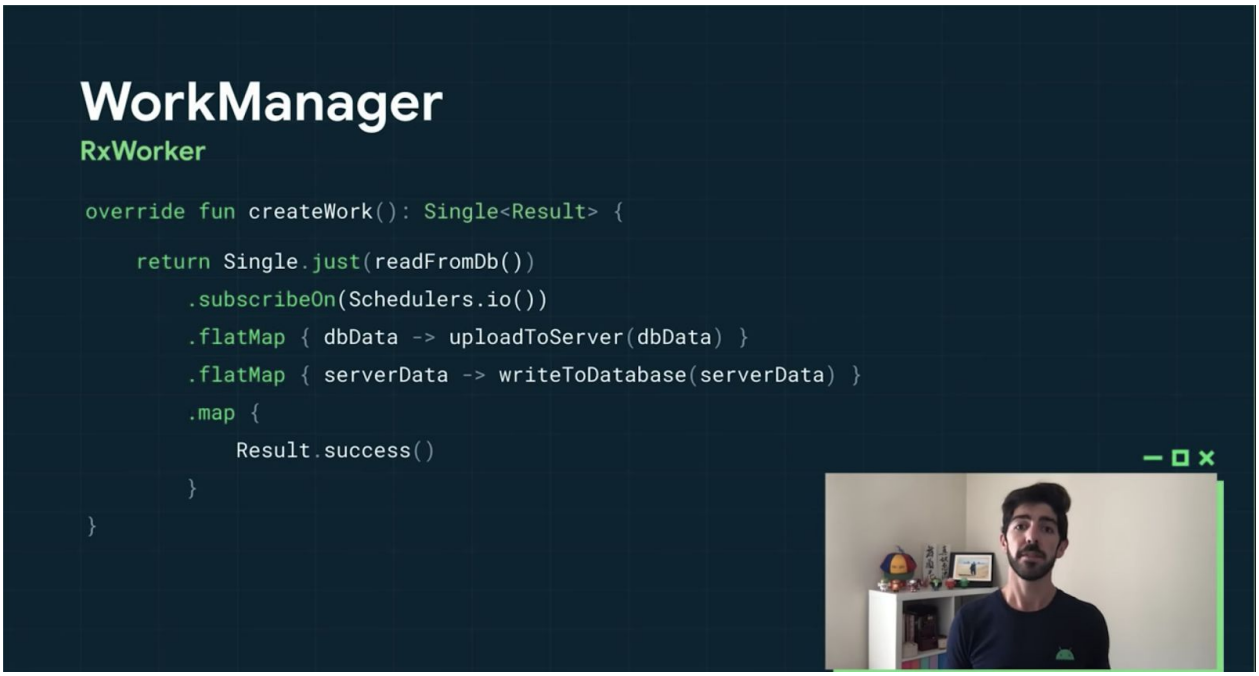
WorkManager

ListenableWorker

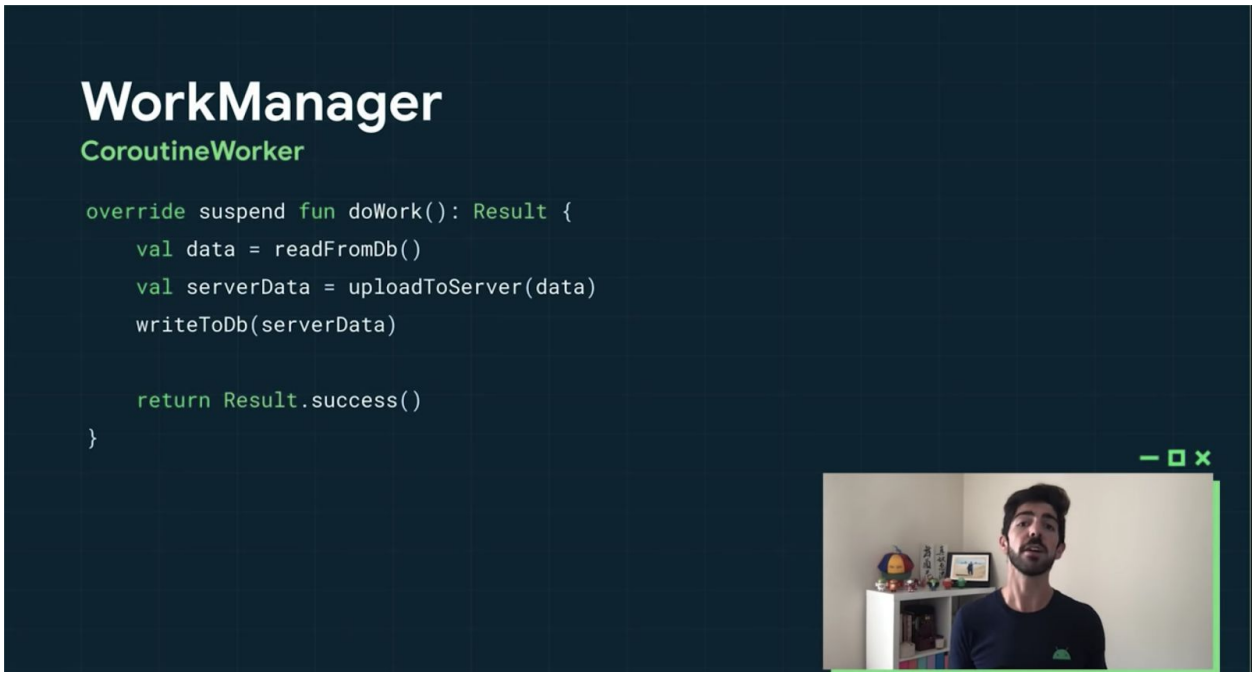
```
override fun startWork(): ListenableFuture<Result> {
    val futureDb: ListenableFuture<Data> = executor.submit{ readFromDb() }
    val futureSrv = Futures.transformAsync(futureDb,{ uploadToServer(it) }, executor)
    return Futures.transformAsync(
        futureSrv, {
            writeToDb(it)
            ListenableWorker.Result.success()
        },
        executor
    )
}
```

c. The other way to do asynchronous request using ListeneableFuture
   i. Cancellation, and error propagation
   ii. But it's hard to find where the actual work is being done.

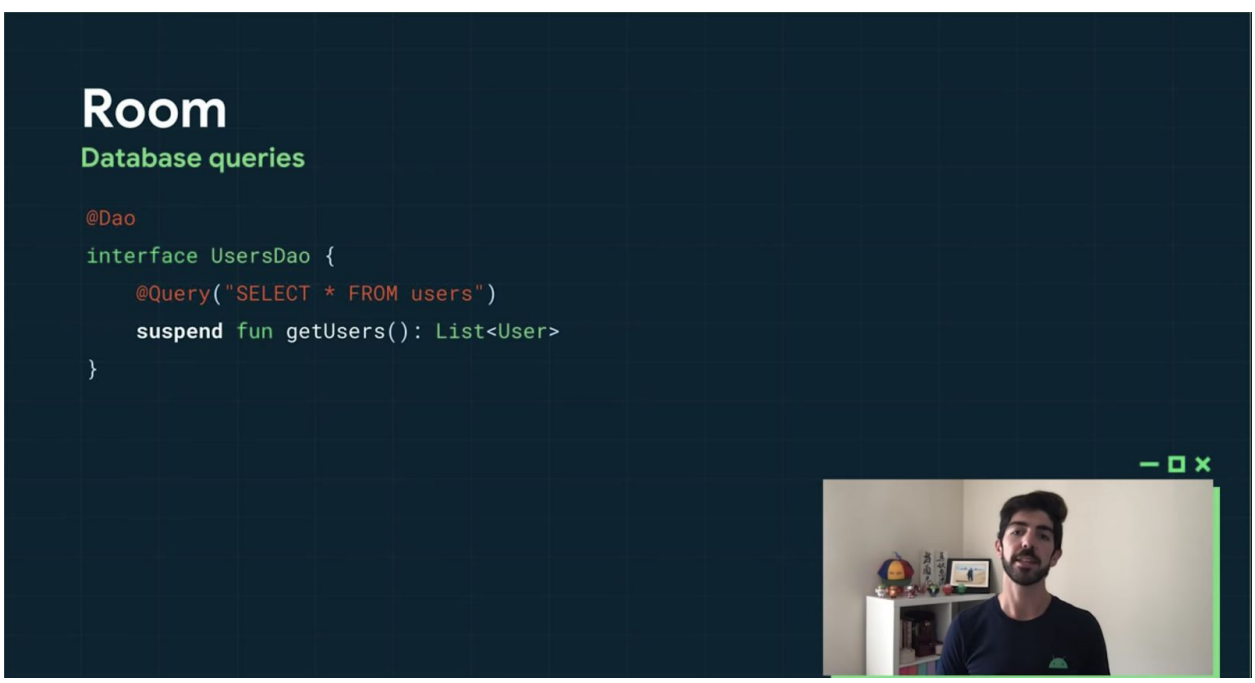




d. Still need a Java way, chaining calls.
e. Have to understand the operators of RxJava
f. Less readability

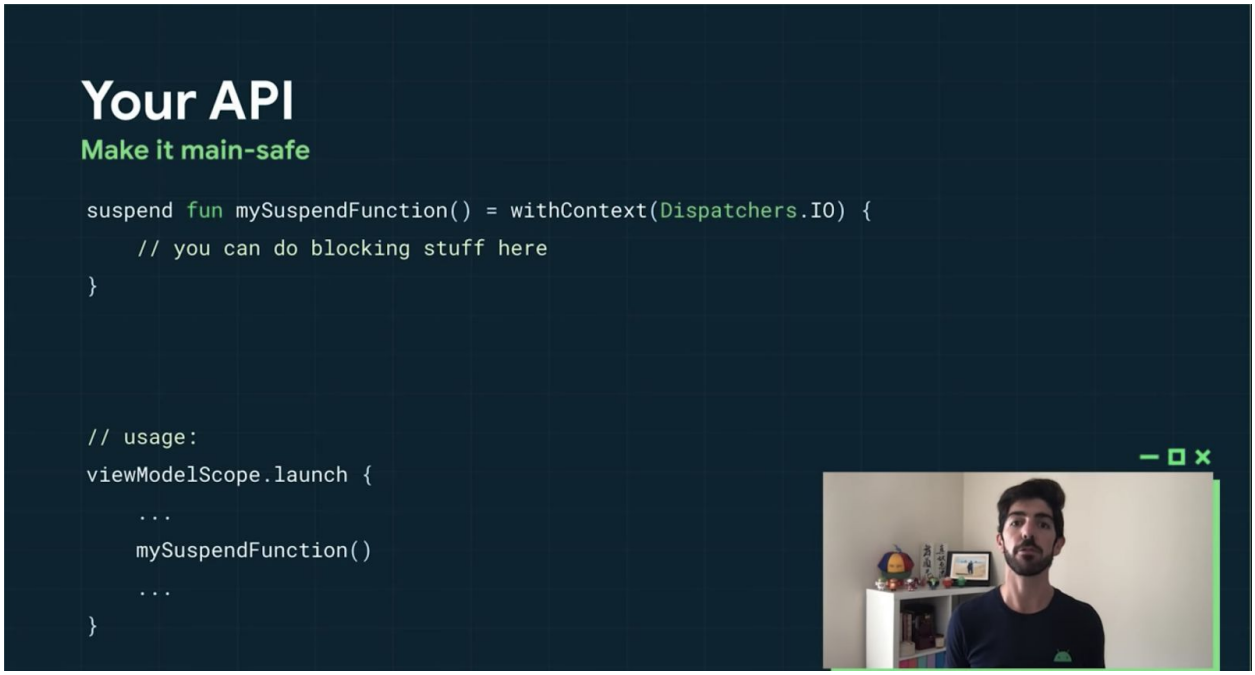


g. Finally we use Coroutines!
   i. Everything looks sequential and is readable
   ii. Error propagation is more natural
   iii. Try-catch if needed

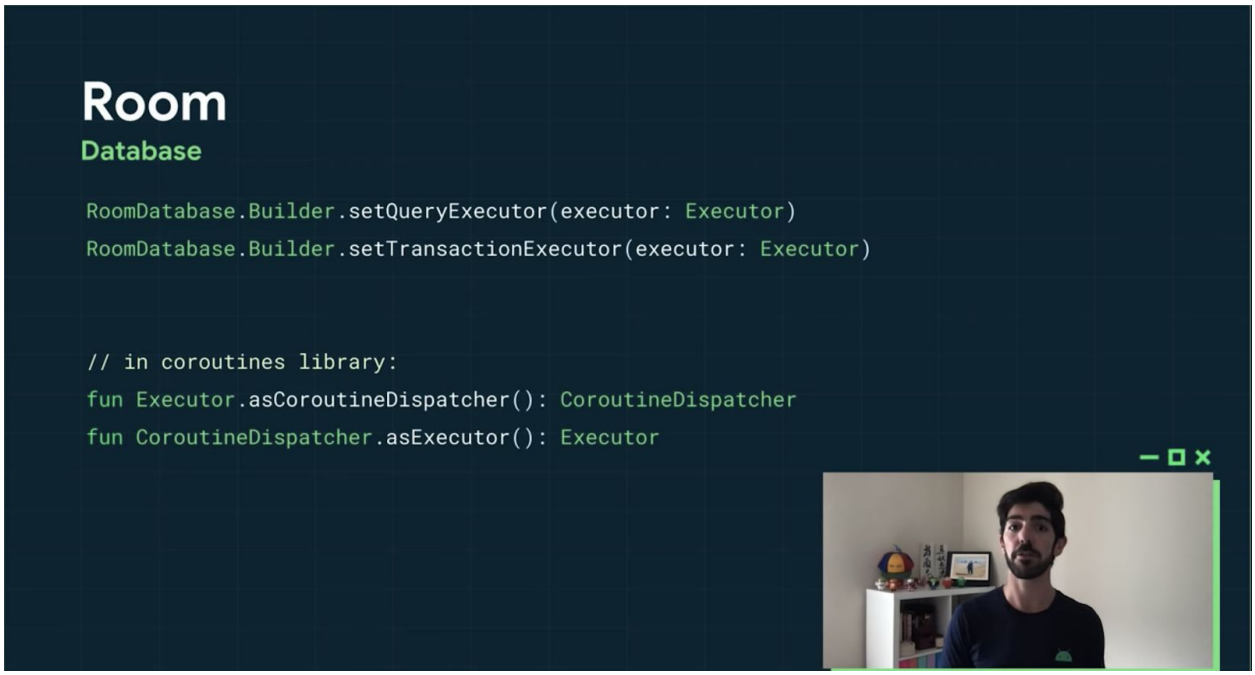


h. Assuming that these operations shouldn't be run in the main thread.

i. Adopting the same approaches in Room and other Jetpack Libraries, and 3rd party libraries such as Retrofit.

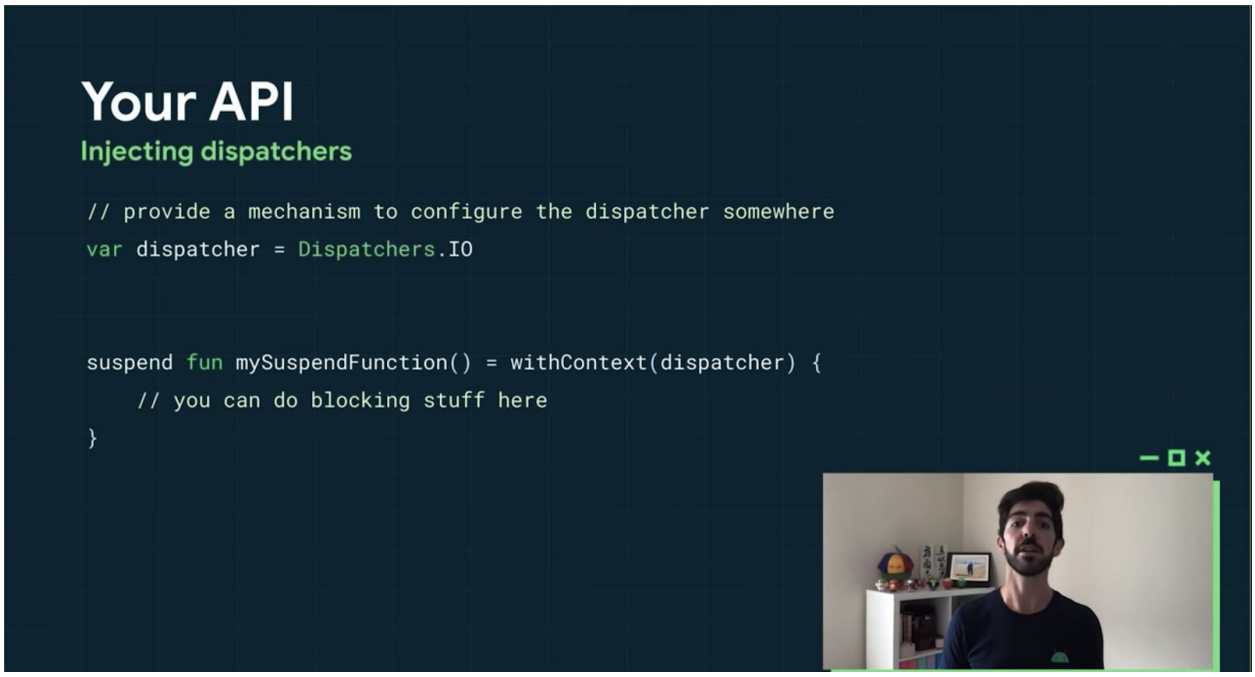


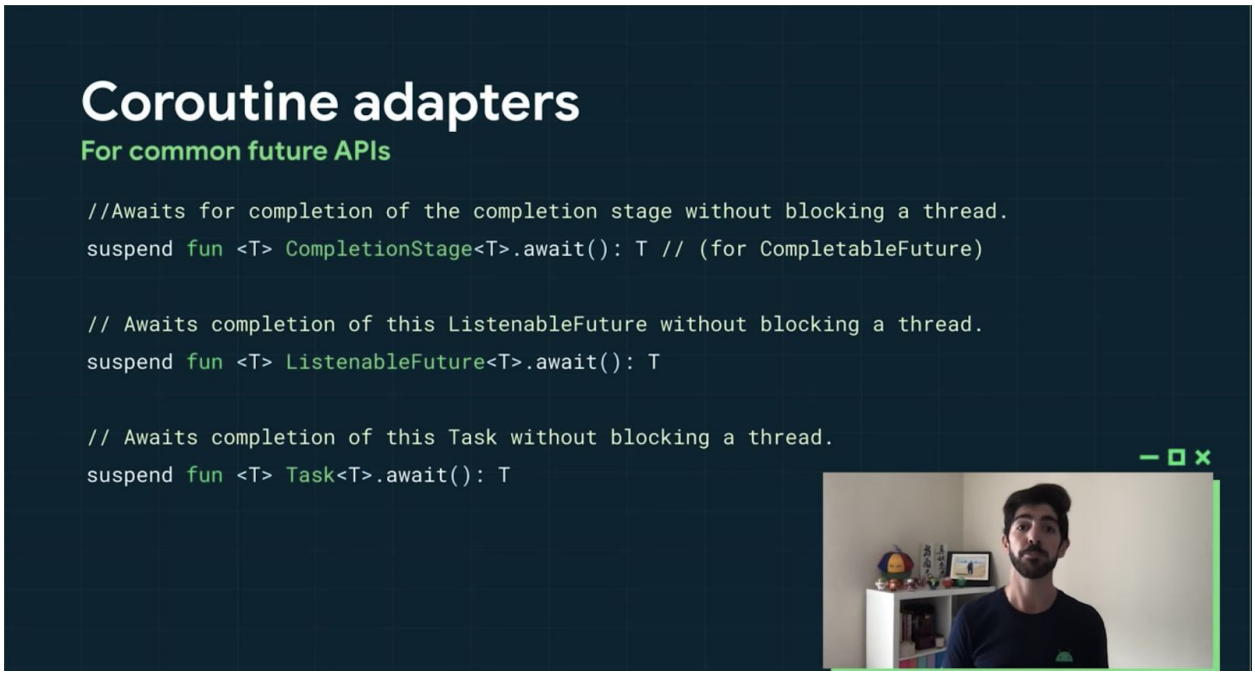
j. Suspend function is main-safe
   i. safe to call from the main thread or the main dispatcher
   ii. Non-blocking the thread



k. Executor’s rule is almost identical to the coroutine Dispatchers.

I. There are methods to translate between them.





```
suspend fun <T> Task<T>.await(): T =
    suspendCancellableCoroutine { continuation ->
        addOnSuccessListener { result ->
            continuation.resume(result)
        }
        addOnFailureListener { exception ->
            continuation.resumeWithException(exception)
        }
    }
```

```
suspend fun <T> Task<T>.await(): T {
    if (isComplete) { // eagerly resume coroutine if Task is ready
      if (isSuccessful) {
        return result
      } else {
        throw exception!!
      }
    }
    return suspendCancellableCoroutine { continuation ->
        addOnSuccessListener { result ->
            continuation.resume(result)
        }
        addOnFailureListener { exception ->
            continuation.resumeWithException(exception)
        }
    }
 }
```

m. This suspendCancellableCoroutine is **to wrap** the callback-based libraries with **suspend functions**. It transforms callback functions to coroutine corresponding.