Kotlin coroutines y Kotlin flow

Kotlin coroutines

https://www.youtube.com/watch?v=ShNhJ3wMpvQ&list=PLQkwcJG4YTCQcFEPuYGuv54nYai_lwil_&ab_channel=PhilippLackner

A **function** is a sequence of intructions that takes inputs and gives us outputs.

A **thread** describes in which context this sequence of instructions should be executed.

If you put all your instructions in your main thread it means that if you use a function that takes a long time to execute you will freeze the UI because the UI also updates in the main thread.

Coroutines can do all the stuff that threads can and they are even more efficient. Coroutines are executed within a thread, are suspendable and they can switch their context.

To implement a coroutine first you have to **import** the dependencies.

```
GlobalScope.launch{
    Log.d("MainActivity", "Coroutine says hello from thread ${Thread.curr
}
Log.d("MainActivity", "Hello from thread ${Thread.currentThread().name}")
```

Every coroutine needs to be started in a coroutine scope. GlobalScope means the coroutine will live as long as our application does (Global Scope is not the best option).

A **suspend function** can only be executed inside another suspend function or inside a coroutine.

If two suspend functions are called in the same scope the duration of the first one will affect the second one.

We can also start a coroutine by passing a **Dispatcher** to the launch function.

```
GlobalScope.launch(Dispatchers.Main) {

}
```

Dispatchers.Main → It will start a coroutine on the main thread. It is useful if you have to do ui operations because you can only change the ui from the main thread.

Dispatchers.IO → Useful for all kinds of data operations.

Dispatchers.Default → Complex and long running calculations that will block the main thread.

Dispatchers.Unconfined → It will stay in the thread that the suspend function resumed.

newSingleThreadContext(”MyThread”)

You can easily switch between contexts. This is useful if for example we have to do a data operation and then change the UI we can switch from IO dispatcher to Main dispatcher

```
GlobalScope.launch(Dispatchers.IO) {
    val answer = doNetworkCall()
    withContext(Dispatchers.Main) {
        tvDummy.text = answer
    }
}
supsend fun doNetworkCall(): String {
    delay(3000L)
    return "This is the answer"
}
```

runBlocking will start a new coroutine and will block the main thread. This can be useful if you don’t really want a coroutine behavior but need to call a suspend function. Another use is to access suspend function from jUnit testing.

```
runBlcoking {
    delay(100L)
}
```

When we launch a coroutine it returns a **Job**, we can save it in a variable and, for example, wait it for finishing.

```
val job = GlobalScope.launch(Dispatcher.Default) {
    repeat(5) {
        Log.d("Main Activity", "Coroutine is still working...")
        delay(1000L)
    }
}

runBlocking {
    job.join() // Wait for the job
    Long.d("MainActivity", "Main Thread is continuing...")
}
```

job.cancel() → cancel a job.

If a coroutine does a long enough or complex calculation we have to manually check if it needs to be canceled with ‘if (isActive)’.

You can also wrap a long calculation function with ‘withTimeout(3000L)’ to end the job after 3 seconds.

If we want to do two calls in the same time and not one after the another we can execute them asynchronously. **Async** call instead of returning a job it will return a Deferred.

```
GlobalScope.launch(Dispatcher.IO) {
    val time = measureTimeMillis {
        val answer1 = async { doNetworkCall1() }
        val answer2 = async { doNetworkCall2() }

        Log.d("MainActivity", "Answer 1 is ${answer1.await()}")
        Log.d("MainActivity", "Answer 2 is ${answer2.await()}")
    }
}

supsend fun doNetworkCall1(): String {
    delay(3000L)
    return "Answer 1"
}
supsend fun doNetworkCall2(): String {
    delay(3000L)
    return "Answer 2"
}
```

To use the lifecycleScope and the viewModelScope we have to import some lifecycle dependencies first.

If we run a function inside GlobalScope the function won’t be destroyed until the app finishes. Instead if we run it inside **lifecycleScope** it will stick the coroutine to the lifecycle of the activity so the coroutine finishes when the activity is destroyed (this will also work with the fragment’s lifecycle). **viewModelScope** is used for the for the viewModel lifecycle.

In Firebase calls instead of nesting callbacks we can run several coroutines and wait for each one before executing the next one that depends on the first one.

```
val tutorialDocument = Firebase.firestore.collection( collectionPath: "coroutines")
    .document( documentPath: "tutorial")
val peter = Person( name: "Peter",  age: 25)
GlobalScope.launch(Dispatchers.IO) { this: CoroutineScope
    delay( timeMillis: 3000L)
    tutorialDocument.set(peter).await()
    val person = tutorialDocument.get().await().toObject(Person::class.java)
    withContext(Dispatchers.Main) { this: CoroutineScope
        tvData.text = person.toString()
    }
}
```

We can also use the lifecycleScope so the coroutines finish if the activity is destroyed.

In depth guide to cancellation & exception handling

You can't use a try-catch block inside a coroutine, it won't work.

It's not the same to handle exceptions in async than in launch because with async we expect a result.

With async you can wrap the await with a try-catch to handle the exceptions, but doing that you are doing something wrong.

We can use a **coroutineExceptionHandler**. This will catch exceptions that would propagate up (only uncaught exceptions).

```
val handler = CoroutineExceptionHandler { _, throwable ->
    println("Caught exception: $throwable")
}
lifecycleScope.launch(handler) {  this: CoroutineScope
    throw Exception("Error")
}
```

Using a CoroutineScope means that is one of the coroutines throw an exception the whole scope will stop, incluiding all of its coroutines that would succeed. In the following example the print will never get printed.

```
val handler = CoroutineExceptionHandler { _, throwable ->
    println("Caught exception: $throwable")
}
CoroutineScope( context: Dispatchers.Main + handler).launch {  this: CoroutineScope
    launch {  this: CoroutineScope
        delay( timeMillis: 300L)
        throw Exception("Coroutine 1 failed")
    }
    launch {  this: CoroutineScope
        delay( timeMillis: 400L)
        println("Coroutine 2 finished")
    }
}
```

Using a **supervisorScope** would make it work (if one coroutine throws an exception the other will continue).

Kotlin Flow

What is Kotlin Flow

Flow is reactive programming, being notified with changes in your code and doing something with this changes. A Flow is a coroutine that can emit multiple values over a period of time. It is like an assembly line.

You can create a Flow with:

```
val varFlow = flow<TypeReturnedOverType> {}
```

For example, the following function emits a value each second (it is a countdown from 10 to 0):

```
val countDownFlow = flow<Int> {  this: FlowCollector<Int>
    val startingValue = 10
    var currentValue = startingValue
    emit(startingValue)
    while(currentValue > 0) {
        delay( timeMillis: 1000L)
        currentValue--
        emit(currentValue)
    }
}
```

Providing that the code shown before is in a ViewModel and you imported the necessary dependencies, to show that value in the UI, using Compose:

```
setContent {
    KotlinFlowsTheme {
        val viewModel = viewModel<MainViewModel>()
        val time = viewModel.countDownFlow.collectAsState(initial = 10)
        Box(modifier = Modifier.fillMaxSize()) { this: BoxScope
            Text(
                text = time.value.toString(),
                fontSize = 30.sp,
                modifier = Modifier.align(Alignment.Center)
            )
        }
    }
}
```

Instead of using Compose you can create a function in the ViewModel, where you created the Flow, to notify changes:

```
init {
    collectFlow()
}

private fun collectFlow() {
    viewModelScope.launch { this: CoroutineScope
        countDownFlow.collect { time ->
            println("The current time is $time")
        }
    }
}
```

The difference between collect and collectLatest is that in collect every time a value is emitted it is executed and in collectLatest it will be canceled if a new value is emitted and the code inside the collect didn't finish processing the last value (in the example done before, if each print had a delay of 1.5 seconds, it would cancel all the values and only show the value 0).

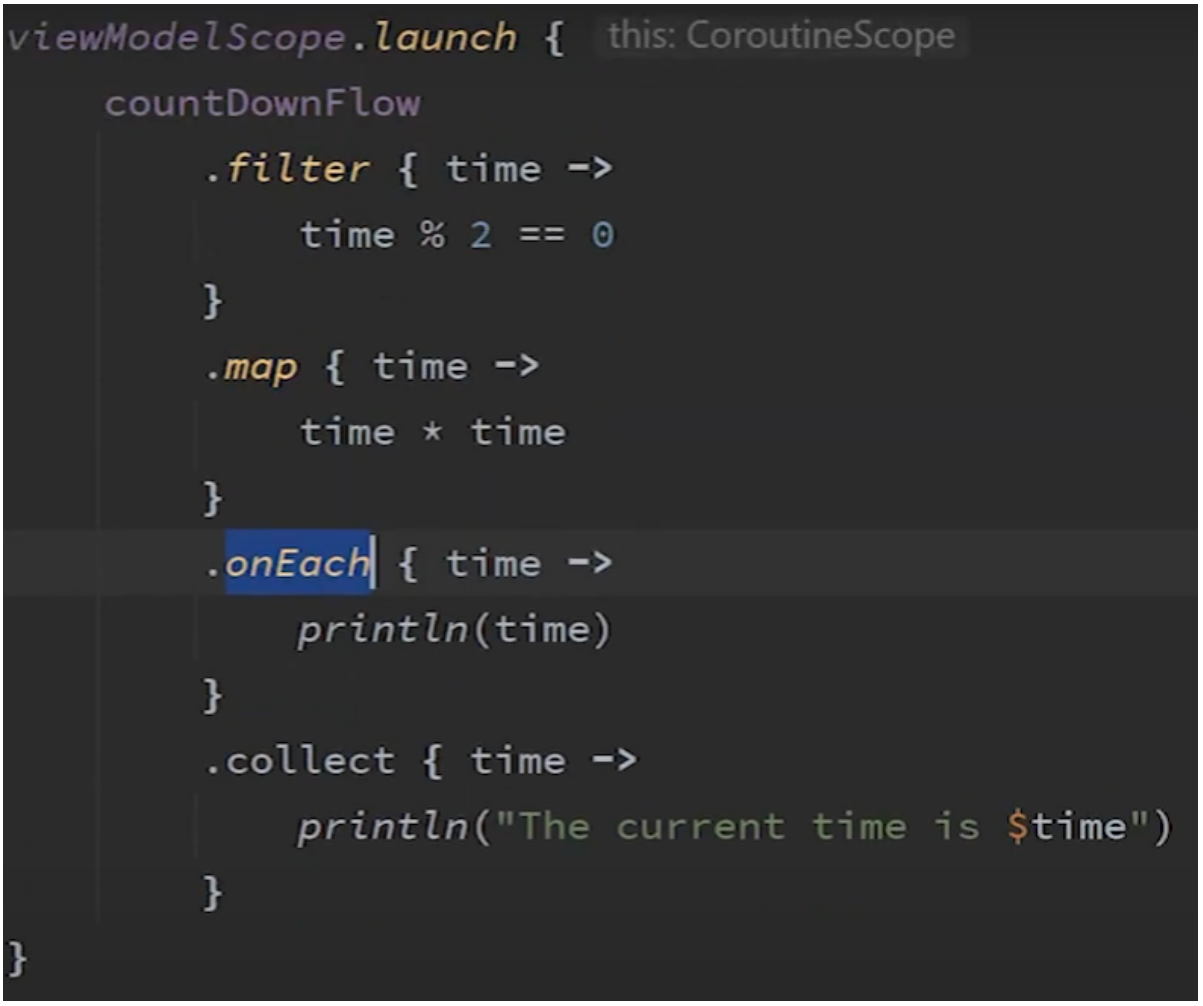
Flow Operators

A Flow operator is something that decides what happens with the emission of a Flow.

An example is the filter operator which would filter the emitted values before collecting them:

The map operator maps a value emitted to another value.

The onEach operator does something with each value emitted.



Terminal flow operators

They are called like that because they terminate the flow, they take all the emissions together and they do something with them.

For example the count operator counts the elements emitted:

```
private fun collectFlow() {
    viewModelScope.launch {  this: CoroutineScope
        val count = countDownFlow
            .count {  it: Int
                it % 2 == 0
            }
        println("The count is $count")
    }
}
```

Reduce has an accumulator and value it would continue to add value to the accumulator:

```
private fun collectFlow() {
    viewModelScope.launch {  this: CoroutineScope
        val reduceResult = countDownFlow
            .reduce { accumulator, value ->
                accumulator + value
            }
        println("The count is $reduceResult")
    }
}
```

Fold does the same but adding an initial value.

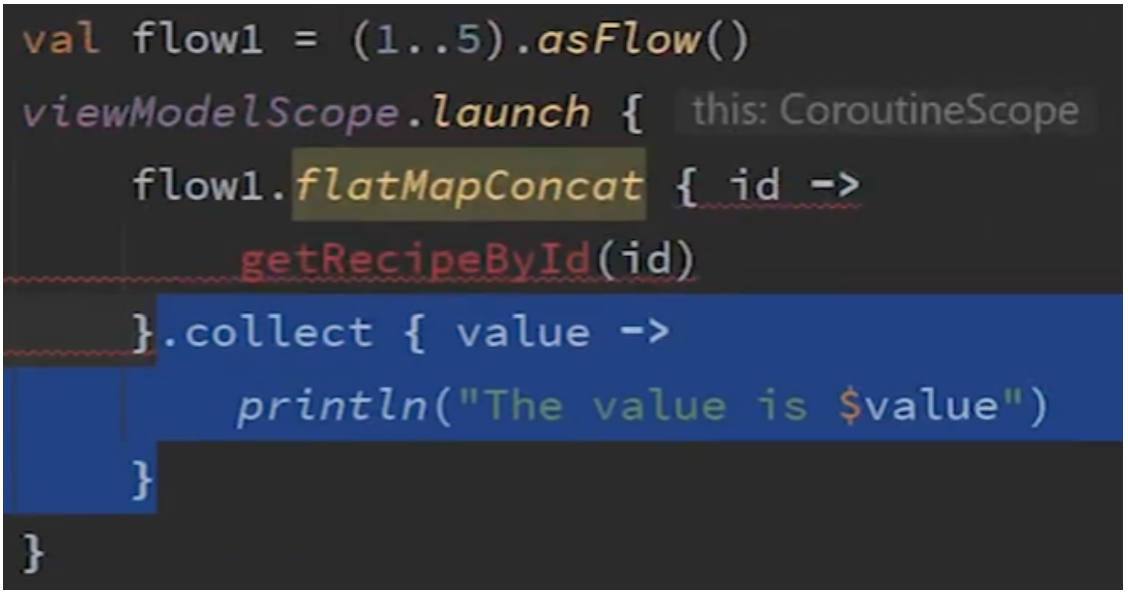
Flattening operators

It is like in lists where you have lists inside lists and you want a single list, removing the inner lists. With Flows you can get two Flows and emit the values of both Flows in a single Flow.

A practical use case would be for example if you have a recipe app. For retrieving a recipe you could have them stored in your local database for performance and return it in a Flow, at the same time you make the call to the API to get the recipe so the Flow emits the value updated when the request finishes updating the value of the recipe. In that case, if you requested

several recipes at the same time you could use the flatMapConcat operator to process them in a single flow.

In the following example you collect 5 recipe ids and for each one of them you collect the recipe:



flatMapLatest is the same logic than collectLatest with with flatMapConcat.

Operators to change how are emissions forwarded

Buffer makes sure that the collect part runs in a different coroutine than the flow emission part so the flow and the collect works independently.

With Conflate if there are more than 1 emissions that we have to collect at the same time instead of getting one after another it would collect only the latest one.

CollectLatest only gets the last emission (if a new emission comes in it cancels the current emission process).

StateFlow & SharedFlow

StateFlow

StateFlow is used to keep state. It will notify all the collectors when there is a new value and only holds a single value (not like the Flows implemented before that continually emit values). The countDownFlow implemented before is a cold flow, if there are no collectors it won't do anything, however a hot flow will do something if there are no collectors. Example:

```
private val _stateFlow = MutableStateFlow(0)
val stateFlow = _stateFlow.asStateFlow()

fun incrementCounter() {
```

```
    _stateFlow.value+=1
}
```

To use that function:

SharedFlow

SharedFlow is used to send one time events. For example when we rotate our device, in the previous example the counter would maintain its value but if we had a firebase call we wouldn't want the code to make the call again when we rotate that's why we use SharedFlow.

```
private val _sharedFlow = MutableSharedFlow<Int>()
val sharedFlow = _sharedFlow.asSharedFlow()
```

Now we can use the SharedFlow in whatever part we want of our ViewModel class.

```
fun squareNumber(number: Int) {
    viewModelScope.launch {
        _sharedFlow.emit(number * number)
    }
}

init {
    viewModelScope.launch {
        sharedFlow.collect {
            delay(2000L)
            println("FIRST FLOW: Thre received number is $it")
        }
    }

    // in another corroutine
    viewModelScope.launch {
        sharedFlow.collect {
```

```
            delay(3000L)
            println("SECOND FLOW: Thre received number is $it")
        }
    }

    // call the function after defining the collectors (it is only a one
    squareNumber(3)
}
```

In this case the sharedFlow will suspend the coroutine for 3 seconds because it is the biggest amount of time that all the collectors collect its value.

To get that value in Compose:

```
LaunchedEffect(key1 = true) {
    viewMode.sharedFlow.collect { number ->

    }
}
```