Coroutines Part 2

Migrating from callbacks to kotlin coroutines

00:00 We are going to start off by showing the project structure and what we currently have

We used Koin as the dependency injection framework.

00:11 This is the app module for the database, data access objects and repository

00:16 This is the network module and we use retrofit

00:20 This is the presenter module

00:30 Our presenter

00:36 As you can see, we are using callbacks when communicating with the repository

00:48 This is the response model we receive from the api the we are consuming. The api we are using takes in a first name and second name in gives us a match percentage

01:02 This is our database that currently does work on the main thread

01:12 This is our repository layer that uses callbacks, On success and on Error

01:18 This our repository implementation. Our current retrofit implementation enqueues requests and we handle the responses with the onSuccess and onError Callbacks

1:38 Finally we have our activity that displays our results0

01:52 To start using coroutines, our interface needs to implement the coroutine scope

02:01 We override the coroutine context and use the main dispatcher as we are following the coroutine main first approach which means everything will run on the main thread unless otherwise specified

02:16 we then move our repository call into the coroutine launch method so that we can get rid of the callbacks

02:22 At this point we are just fixing compilation errors

02:33 We do the same for the getData method

02:48 Since we are getting rid of callbacks we will need some way to handle the results from the repository. We do this by creating our own result class

03:03 Our result class have a generic model and a throwable

03:26 We can then go to the repository and change the return type for our calculateLovePercentage method to use our new result class we created. Our result class will use the same model that our onSuccess callback used

03:55 We will need some way to handle network errors so we created a try catch block in which we will move our network call to

04:08 If we do encounter an error, we just propagate that error to the presenter

04:14 We can now rewrite our network request by using retrofits synchronous execute method instead of the asynchronous enqueue method

04:34 Typically you would check if the response is empty

04:46 We can use our old code to save the response to room and remove the onSucess callback. We then need to return the our response wrapped in our result construct that we created earlier

05:09 we can then remove all our enqueue code

05:19 at this point we can change the signature of the calculateLovePercentage and remove the callbacks

05:25 Since we using the Coroutine main first approach we need to set the context of the method to use the IO thread pool. We then also need to make our functions suspendable by using the suspend keyword

06:00 Suspendable functions allows us to use the async and await methods. We move our network call into a async block that’s deferred until we await our response

06:29 the calculateLovePercentage function will now be suspened until we get a response from the network call and then it will be resumed. This demonstrates the purpose of coroutines: writing asynchronous code in a asynchronous manner

06:46 In our presenter we need to handle our result because the callbacks were removed

06:57 We check if our result has a value or an error and then handle it each case accordingly

07:09 if there’s a value and no error, we update the view with the value we have received from the repository

07:31 if there’s an error, we let the view know that an error has occurred

08:07 We get the latest data stored in room so that view update the view accordingly

08:14 We can finally remove the line that allows queries on the main thread

08:23 let’s see if our app still works, as you can see, Ron and Hermione are an ok match

08:30 it all looks good right? However, we still have a problem, if the user presses back while the network call is ongoing, the network call won’t be cancelled and could cause leaks. To fix this, we use coroutine Jobs

08:44 We use a superviser job which is like a parent job that remains active while it’s children can be cancelled

09:00 We override the activities onstop method to let the presenter and inturn the job know that it needs to be cancelled

09:13 We can finally remove the custom result class we created by using the coroutine exception handler

09:23 The coroutineContext is the context in which the error occured

09:32 we previously forget to add the job we created to the context, so we will add both the job and our coroutine exception handler

09:42 Finally we can remove result class we created earlier depend on the coroutine exception handler. We change the return type to be just our model instead of a result wrapper

09:58 we can also get rid of our try catch code

10:10 We need to handle an empty response, we simply just returned an empty model. You should decide on your own implementation there

10:35 we can go back to the presenter wrap our repository call in a runCatching block which has it’s own kotlin result. The kotlin result exposes an onSuccess, OnFaiture, map and other methods that can be used

11:15 lets see if our changes work ☺ .Sarah and Lena seem like a catch too

11:30 Coroutines can be tested. Lets test our presenter that uses coroutines. We will create a new test and remove the groovy inherit which got added automatically

11:48 Our test will use the PowerMockRunner and we will also cater for logs. It is important to note that this test is a unit test and not instrumented test so it does not have access to the android framework

12:28 We created an instance of our presenter but it needs a repository that we will have to mock

12:45 we will also mock the view that the presenter communicates with

13:02 We give our presenter our mocked repository and can start writing the test setUp code which runs before every test

13:26 We mock the log class, not necessary in our example, but useful if you have logs. We then give our presenter the mocked view

13:37 Since it is local unit test, we will need to change the Dispatchers.Main to use a test dispatcher.

13:56 We will be testing that our getPreviousResults works fine. As always, tests should have an arrange act assert

14:27 We will also need to use a TestCoroutineScope that our test will use via the runBlockingTest method

14:45 Similarly to our presenter, Our Test coroutine scope will need a context, we give it the testDispatcher

15:05 we Can then finally start with the actual test, we use Mockito to return an empty list whenever our repository requests for previous data

15:20 we ask the presenter to getPreviousLoveCalculations from the repository

15:32 we then need to verify that our presenter indeed actually called the repository and let the view know that there was an update. We do this by verifying the getPreviousLoveCalculations and showPreviousResults methods are called

15:57 we can advance time to simulate delays, this is powerful as it doesn’t actually delay the test execution time like thread.sleep does

16:10 Finally we can run our test and see that all is ok