Na aula 217 anterior aprendemos como fazer coroutine de forma simultânea

Agora vamos usar o Async e await

Aqui a gente usava o launch para criar outra coroutine

Mas gora vamos usar o async = assíncrona

val job2 = *launch* **{** // 2 coroutine  
 resultado2 = tarefa2()  
**}**

Depois

*/\*\* Utlizando agora o async  
 \* Async = assincrona vai funcionar da mesma forma que fizemos la em cima, criando outra coroutine  
 \* Mas vai ser mais simples de fazer  
 \*/*job = *CoroutineScope*(Dispatchers.IO).*launch* **{** // Esse metodo serve para medir quanto tempo vai levar para ficar pronto.  
 val tempo = *measureTimeMillis* **{** val resultado1 = *async* **{**tarefa1() **}** val resultado2 = *async* **{**tarefa2()**}** Log.i("info\_coroutine", "Resultado1: ${resultado1.await()}")  
 Log.i("info\_coroutine", "Resultado1: ${resultado2.await()}")  
 **}** Log.i(  
 "info\_coroutine",  
 "Tempo: $tempo"  
 ) // Vai calcular quanto tempo vai levar para executar esse trecho de codigo.  
  
**}**

No async temos uma diferença a tarefa vai ser executada de forma assíncrona

E depois o resultado da execução vai ser jogada dentro da variável job.

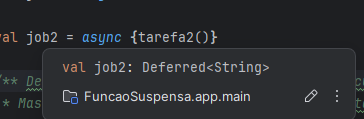
Antes diferença

val job2 = *launch* **{** // 2 coroutine  
 resultado2 = tarefa2()  
 **}**

Depois

val job1 = *async* **{**tarefa1() **}**val job2 = *launch* **{**tarefa2()**}**

Agora se você ver o tipo da variável ela não é mais do tipo job e sim do tipo retorno String



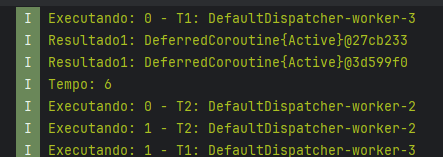
# Await

Lembra que temos o problema de: função suspensa? Nesse código ele vai sair executando tudo sem esperar essa função suspensa, e ai teríamos um problema porque o código não estaria pronto para ser exibido.

E com isso vamos usar essa nova palavra chave await = aguardar.

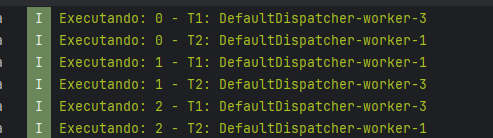
Ele vai aguardar o código esta pronto para depois exibir.

Repara antes de usar o await oque acontece



Isso é porque o resultado ainda não esta pronto, a função suspensa ainda não terminou de processar tudo la, mas a coroutina já executou antes.

Agora com o await



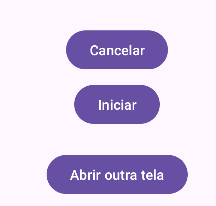
Agora sim, ele vai aguardar e só quando o tarefa atribuir o valor no resultado1 que vamos exibir o código.

Agora vamos modificar o botão de acordo com o async para teste

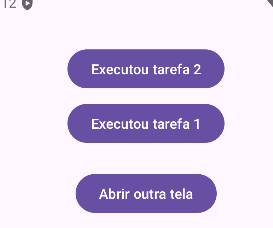
Quando ele terminar de executar o código o novo valor vai para dentro da variável e altera o nome do botão

job = *CoroutineScope*(Dispatchers.IO).*launch* **{** // Esse metodo serve para medir quanto tempo vai levar para ficar pronto.  
 val tempo = *measureTimeMillis* **{** val resultado1 = *async* **{**tarefa1() **}** val resultado2 = *async* **{**tarefa2()**}** withContext(Dispatchers.Main) **{** binding.btnIniciar.*text* = "${resultado1.await()}"  
 binding.btnCancelar.*text* = "${resultado2.await()}"  
 **}** Log.i("info\_coroutine", "Resultado1: ${resultado1.await()}")  
 Log.i("info\_coroutine", "Resultado1: ${resultado2.await()}")  
 **}** Log.i(  
 "info\_coroutine",  
 "Tempo: $tempo"  
 ) // Vai calcular quanto tempo vai levar para executar esse trecho de codigo.  
  
**}**

Antes



Depois



O tempo para alterar os botões também foi diferente, pq tarefa1 é pra ser executado em 3 segundos, já terafa2 em 5 segundos.

Mas ele só alterou o botão quando ele terminou de executar, por causa do await

Await em resumo é: espere terminar o processo e depois executa.