Vamos fazer o teste fazendo duas funções trabalharem de forma simultâneas

Primeiro criamos esses 2 metodos para teste

private suspend fun tarefa1() : String {  
 *repeat*(15) **{** indice **->** Log.i("info\_coroutine", "Executando: $indice - T1: ${Thread.currentThread().*name*}")  
 delay(2000) //2s  
 **}** return "Executou tarefa 1"  
}  
  
private suspend fun tarefa2() : String {  
 *repeat*(15) **{** indice **->** Log.i("info\_coroutine", "Executando: $indice - T2: ${Thread.currentThread().*name*}")  
 delay(2000) //2s  
 **}** return "Executou tarefa 2"  
}

Aqui agora quando a gente clicar no botão, ele vai executar a tarefa1, depois a tarefa2

*/\*\* Criando nossa coroutine e executando tarefas simultaneas  
 \* Lembrar que criamos uma funçao suspensa então, ele primeiro termina uma execução para depois atribuir o valor (TAREFA1 / TAREFA)  
 \*/* job = *CoroutineScope*(Dispatchers.IO).*launch* **{** val resultado1 = tarefa1()  
 val resultado2 = tarefa2()  
  
 Log.i("info\_coroutine", "Resultado1: $resultado1")  
 Log.i("info\_coroutine", "Resultado1: $resultado2")  
 **}  
}**

Mas não é isso que queremos, queremos que ele execute de forma simultânea

Resumindo temos uma coroutine e ela vai executar os códigos de forma linear, de cima para baixo.

Então ela vai primeiro executar a primeira função suspensa, vai aguardar ela ser executada e depois vai para a próxima.

Agora se você quiser executar as duas tarefas ao mesmo tempo, você pode criar outra coroutina, uma dentro da outra.

Fazendo dessa forma.

Agora sim criamos 2 coroutine e estamos fazendo de forma paralelo

Mas imagina que as coroutine seja uma pessoa, então cada uma dessas coroutine vai ser executado por uma pessoa diferente.

job = *CoroutineScope*(Dispatchers.IO).*launch* **{ // Alleph** var resultado1: String? = null  
 var resultado2: String? = null  
  
 *launch* **{**// Nova coroutine // Gabriel  
 resultado1 = tarefa1()  
 **}** *launch* **{** // 2 coroutine // Guilherme  
 resultado2 = tarefa2()  
 **}** Log.i("info\_coroutine", "Resultado1: $resultado1")  
 Log.i("info\_coroutine", "Resultado1: $resultado2")  
 **}  
}**

Ai em cima teremos um problema, vai retornar um NULL em algum momento, porque Gabriel e Guilherme ainda não vai ter terminado o trabalho, então pra isso, vamos mandar na linha de execução o código esperar ficar pronto para processar.

job = *CoroutineScope*(Dispatchers.IO).*launch* **{** // Esse metodo serve para medir quanto tempo vai levar para ficar pronto.  
 val tempo = *measureTimeMillis* **{** var resultado1: String? = null  
 var resultado2: String? = null  
  
 val job1 = *launch* **{**// Nova coroutine  
 resultado1 = tarefa1()  
 **}** val job2 = *launch* **{** // 2 coroutine  
 resultado2 = tarefa2()  
 **}** */\*\* Dessa forma vamos incluir na linha de execução as tarefas la em cima.  
 \* Mas imagina que essas tarefas ainda nao estao prontas?  
 \* Então ele vai aguardar essas tarefas estarem prontas para poder executar.  
 \*   
 \*/* job1.join()   
 job2.join() // JOIN = Juntar  
  
  
 Log.i("info\_coroutine", "Resultado1: $resultado1")  
 Log.i("info\_coroutine", "Resultado1: $resultado2")  
 **}** Log.i(  
 "info\_coroutine",  
 "Tempo: $tempo"  
 ) // Vai calcular quanto tempo vai levar para executar esse trecho de codigo.  
  
**}**

Resumindo: Ele só vai permitir que avance para a próxima linha quando o job1 e job2 estiver pronto.

Resumindo2: ele vai juntar o job1 e job2 no fluxo de execução.

Então nessa aula aprendemos a criar coroutine de forma separada, para fazer trabalhos diferentes, e no final juntamos ela na mesma coroutine para fazer trabalhos simultâneos.