Podemos criar uma coroutine de várias maneiras, e nessa aula vamos aprender as formas de criar elas.

# Thread Principal UI

Não é recomendado fazer isso, porque se o volume de dados for muito grande, podemos travar a thread Principal

Aqui nesse exemplo estamos fazendo a thread e atualizando o botão com o texto para indicar quando terminou a thread.

binding.btnIniciar.setOnClickListener **{** */\*\* Aqui estamos criando uma coroutine que vai atualizar a thread principal a UI  
 \* Vamos mandar repetir 15x  
 \* INDICE = quantidade de vezes que esta repetindo  
 \* Thread.currentThread.name = nome da Thread  
 \* Delay = tempo que vai demorar para repetir todo o codigo de novo  
 \*/  
 CoroutineScope*(Dispatchers.Main).*launch* **{** *repeat*(15) **{** indice **->**

binding.btnIniciar.setText("Terminou")Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
 }  
  
}**

## Outra maneira de fazer com menos código

**//CoroutineScope(Dispatchers.Main).launch { // MODO ANTIGO**  
 *MainScope*().*launch* **{ // Modo novo** binding.btnIniciar.setText("Terminou")  
 *repeat*(15) **{** indice **->** Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
 }  
}**

Diferença de um para o outro, usando o Dispatchers você define em qual coroutine você quer fazer a atualização

Usando o MainScope ele já atualiza na thread principal.

Isso seria um atalho

# GlobalScope

Ele funciona igual a IO é apenas uma maneira de simplificar o código.

GlobalScope.*launch* **{** //binding.btnIniciar.setText("Terminou")  
 *repeat*(15) **{** indice **->** Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
}**

Utilizamos o global quando queremos que alguma coisa seja executada por tempo indeterminado, independente da navegação

Atenção, sabemos que quando vamos para outra tela, a coroutine continua executando, mas para cancelar isso podemos fazer da seguinte forma, podemos colocar a coroutine dentro de um JOB e usar o método job.cancel()

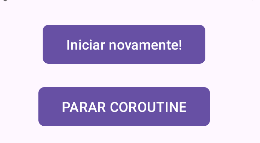
Ficando dessa forma

# Cancelando uma coroutine

class MainActivity : AppCompatActivity() {  
  
 private val binding by *lazy* **{** ActivityMainBinding.inflate(*layoutInflater*) **}** private var job: Job? = null

binding.btnIniciar.setOnClickListener **{** */\*\* Aqui estamos criando uma coroutine que vai atualizar a thread principal a UI  
 \* Vamos mandar repetir 15x  
 \* INDICE = quantidade de vezes que esta repetindo  
 \* Thread.currentThread.name = nome da Thread  
 \* Delay = tempo que vai demorar para repetir todo o codigo de novo  
 \*/* //CoroutineScope(Dispatchers.Main).launch { // MODO ANTIGO  
 //MainScope().launch {  
  
 job = GlobalScope.*launch* **{** //binding.btnIniciar.setText("Terminou")  
 *repeat*(15) **{** indice **->** Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
 }  
}**binding.btnParar.setOnClickListener **{** job?.cancel()  
 binding.btnIniciar.*text* = "Iniciar novamente!"  
**}**





# Cancelando coroutine de forma automática

## Lifecycle

Existe uma maneira de fazer isso de forma automática, sem precisar usar botões ou ciclos de vida.

Podemos fazer isso usando um outro Scopo que é uma biblioteca do JETPACK

Ela é o Lifecycle com ela podemos controlar de acordo com o clico de vida de uma activity.

Pra isso vamos precisar adicionar a dependência

implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle\_version"

Também tem o site oficial com todas as dependências

https://developer.android.com/jetpack/androidx/releases/lifecycle?hl=pt-br

Atenção aqui precisamos configurar a versão do da dependência

dependencies **{** *implementation*(*libs*.*androidx*.*core*.*ktx*)  
 *implementation*(*libs*.*androidx*.*appcompat*)  
 *implementation*(*libs*.*androidx*.*constraintlayout*)  
 *testImplementation*(*libs*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*espresso*.*core*)  
 implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle\_version"  
**}**

**Ficando dessa forma**

*dependencies* **{** *implementation*(*libs*.*androidx*.*core*.*ktx*)  
 *implementation*(*libs*.*androidx*.*appcompat*)  
 *implementation*(*libs*.*material*)  
 *implementation*(*libs*.*androidx*.*activity*)  
 *implementation*(*libs*.*androidx*.*constraintlayout*)  
 *testImplementation*(*libs*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*junit*)  
 *androidTestImplementation*(*libs*.*androidx*.*espresso*.*core*)  
 *implementation* ("androidx.lifecycle:lifecycle-runtime-ktx:2.8.7")

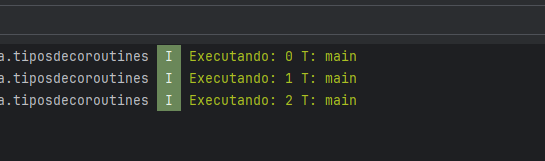
*lifecycleScope*.*launch* **{** //binding.btnIniciar.setText("Terminou")  
 *repeat*(15) **{** indice **->** withContext(Dispatchers.Main) **{** binding.btnIniciar.*text* = "Terminou!"  
 **}** Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
}**

Agora usando o lifecycle do JatPack agora ele vai fazer o controle automático.

Agora quando clicamos em abrir nova tela

Ele vai abrir a nova tela e quando a gente fechar essa tela ele vai parar automaticamente o coroutine

Atenção: no nosso código colocamos abrir e logo em seguida fechar, não colocamos botão de fechar, apenas para teste.



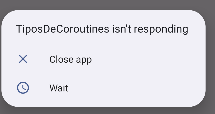
# runBlock

Dessa maneira vamos fazer nossa coroutine, mas bloqueando a thread principal

Usando ele vamos ter o mesmo erro do aplicativo do GNV, porque ele vai deixar a tela principal bloqueada

Não é recomendado usar.

*runBlocking* **{** //binding.btnIniciar.setText("Terminou")  
 *repeat*(15) **{** indice **->** withContext(Dispatchers.Main) **{** binding.btnIniciar.*text* = "Terminou!"  
 **}** Log.i("info\_coroutine", "Executando: $indice T: ${Thread.currentThread().*name*}")  
 delay(1000L) //1s  
 **}  
 }  
}**



Se tentar clicar em algo ele vai da esse erro.