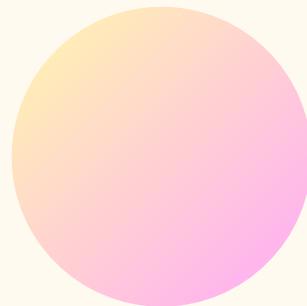


ANDROID

SESSION THREE



Agenda

- UI freezing
 - Concurrency
 - Asynchronous code
 - Coroutines Getting-Ready
 - Key Components of Kotlin
- ## Coroutines



UI FREEZING



UI freezing issue

Why Does the UI Freeze in Android? 🤔

In Android, the UI freezes when **long-running tasks** are executed on the Main Thread. This happens because Android processes all UI updates and user interactions on a single thread—called the **Main (UI) Thread**.

Let's walk through a demo to show UI freezing



UI freezing issue

Why Does the UI Freeze in Android? 🤔

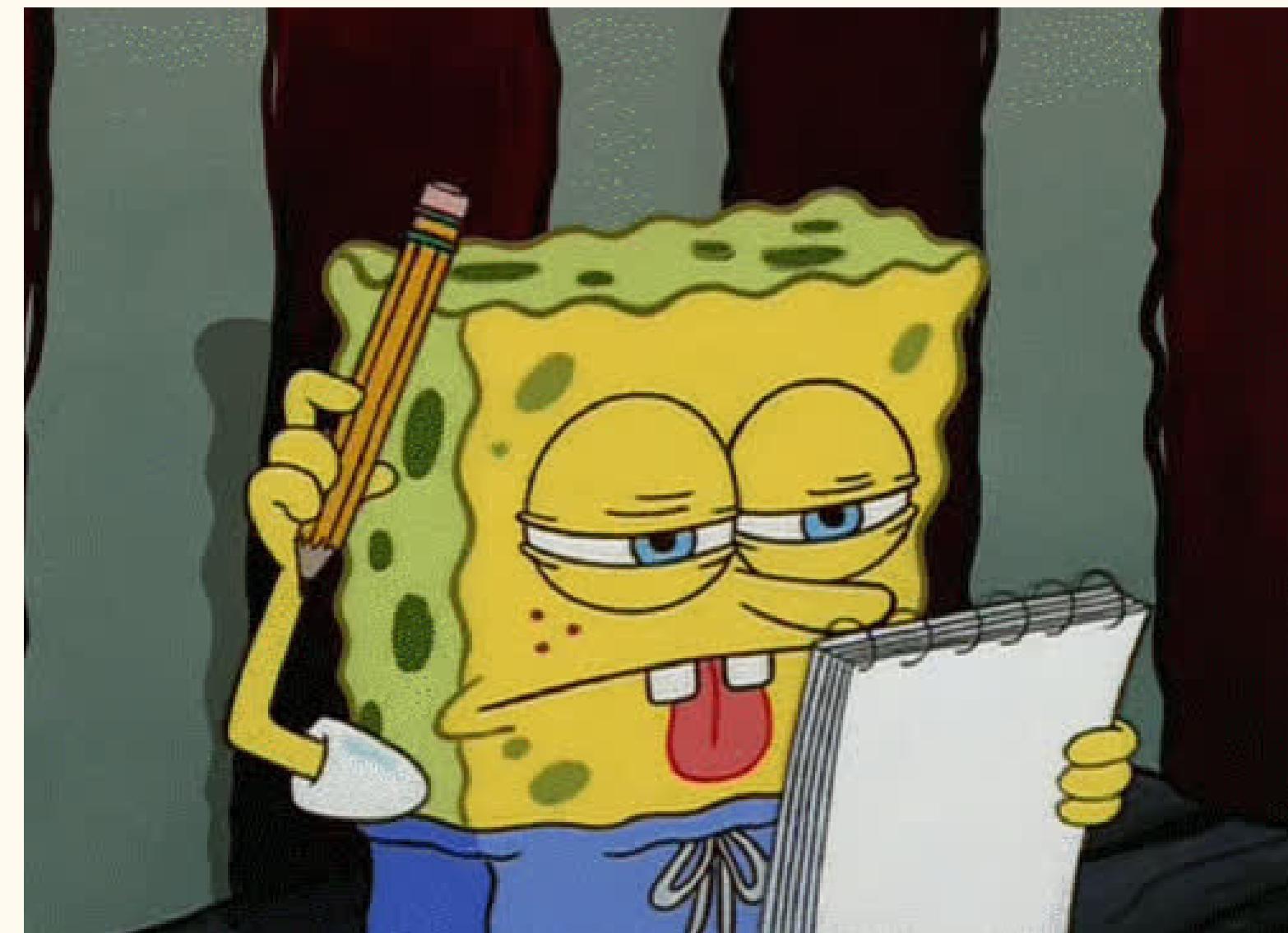
System UI isn't responding

- ✗ Close app
- ⌚ Wait

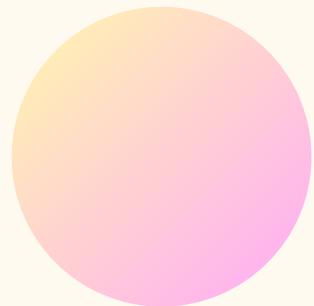


How to solve this issue ??!

let's brainstorm
concurrency



CONCURRENCY



Concurrency Definition:

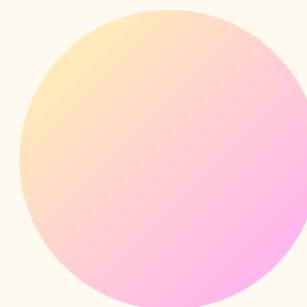
Concurrency is the ability of a system to handle multiple tasks at the same time.

Pros of Concurrency:

- Better Resource Utilization
- Improved Performance
- Increased Responsiveness
- Handling Long-Running Operations



ASYNCHRONOUS CODE



Sequential Code vs Asynchronous Code

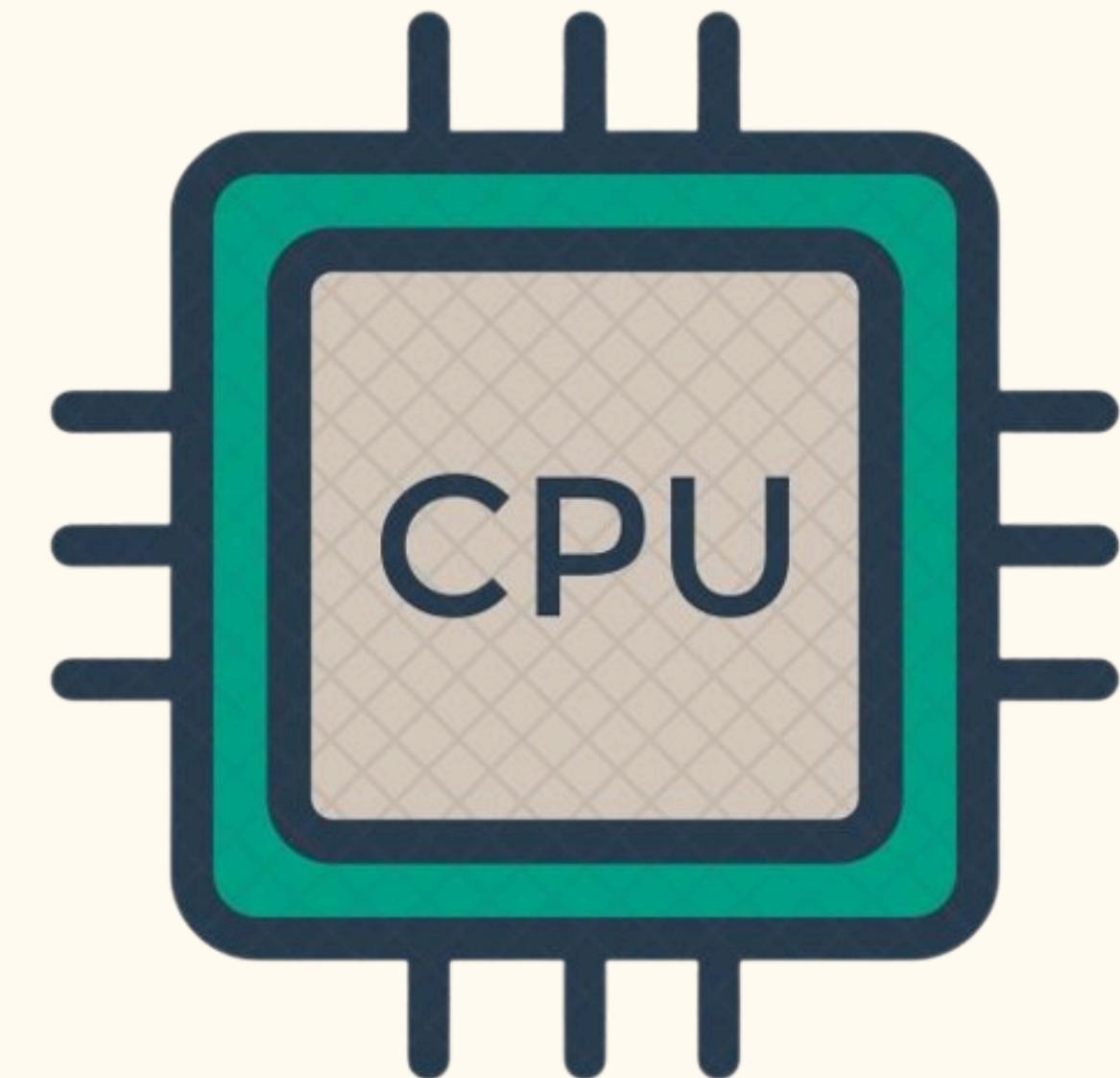
Aspect	Sequential Code	Asynchronous Code
CPU Usage	Single CPU	Multiple CPUs
Execution Time	Longer execution as each task blocks the next.	Shorter execution time by allowing tasks to run concurrently.
Responsiveness	Can cause UI freezes as tasks block the main thread.	UI remains responsive as tasks run concurrently without blocking.

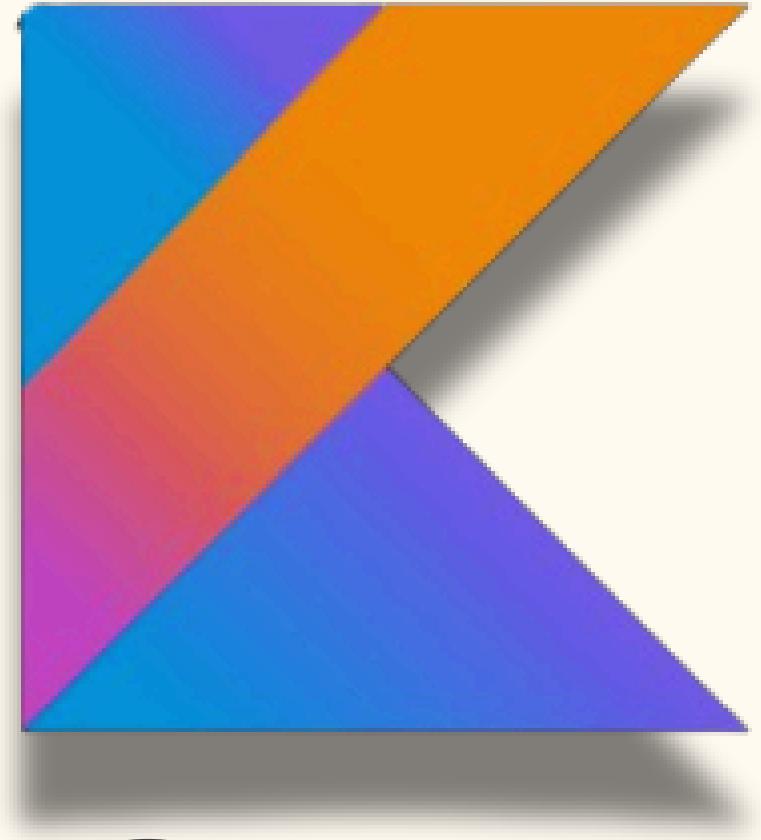


Sequential Code vs Asynchronous Code

you always write Sequential code.

How to write Asynchronous code
in android ? **Kotlin Coroutines**





COROUTINES GETTING-READY



Getting-Ready



Coroutine not a part from Kotlin standard libraries, **so we need to include it !!**

The screenshot shows the 'New Project' dialog in IntelliJ IDEA. On the left, there's a sidebar with icons for 'New Project', 'Java', 'Kotlin' (which is highlighted with a blue selection bar), 'Groovy', and 'Empty Project'. Below that is a 'Generators' section with 'Maven Archetype', 'JavaFX', and 'Spring'. The main right panel has fields for 'Name' (set to 'untitled3'), 'Location' (set to '~\IdeaProjects'), and 'Build system' (with 'Gradle' selected). A yellow banner with the word 'Choose' is overlaid on the 'Gradle' button, and a red arrow points from the text 'we need to include it !!' to this banner. At the bottom, there's a lock icon and 'Gradle DSL' options for 'Kotlin' and 'Groovy'.



Getting-Ready



Coroutine not a part from Kotlin standard libraries, **so we need to include it !!**

Your project → build.gradle.kts → dependencies

Add:

```
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")
```



KEY COMPONENTS OF KOTLIN COROUTINES



Key Components of Kotlin Coroutines

Coroutine Scope:

Defines the lifecycle of the coroutines and controls when they are launched and cancelled.

Scopes:

- GlobalScope: A global scope that is not tied to any lifecycle.
- lifecycleScope: Tied to the lifecycle of an Activity or Fragment
- viewModelScope: Tied to the lifecycle of a ViewModel



Coroutines Builder

Launch(Fire & Forget 🚀): Create coroutines that we do not expect get a value from it (We do not wait from an output from it)

```
fun main() {  
    GlobalScope.launch(){ this: CoroutineScope  
        |> hello()  
    }  
}  
  
fun hello(){  
    println("Hello world")  
}
```



Coroutines Builder

Async, await(Runs & Returns a Result ⚪): Create coroutines which we need or expect a value from it (Waiting for an output)

```
fun main() {  
    runBlocking{ this: CoroutineScope  
        var sum=async{ this: CoroutineScope  
            calcSum(5,3)  
        }  
        println(sum.await())  
    }  
}  
fun calcSum(x:Int,y: Int)=x+y
```



Coroutines Builder

RunBlocking(Blocks & Waits ⏪): Pauses the current thread until everything inside finishes ,not recommended for long-running tasks, but useful in main().

```
fun main() {  
    runBlocking{ this: CoroutineScope  
        //Block main thread until the calcSum() finishes  
        calcSum(5,5)  
    }  
}  
fun calcSum(x:Int,y: Int){println(x+y)}
```



This code return no output !!

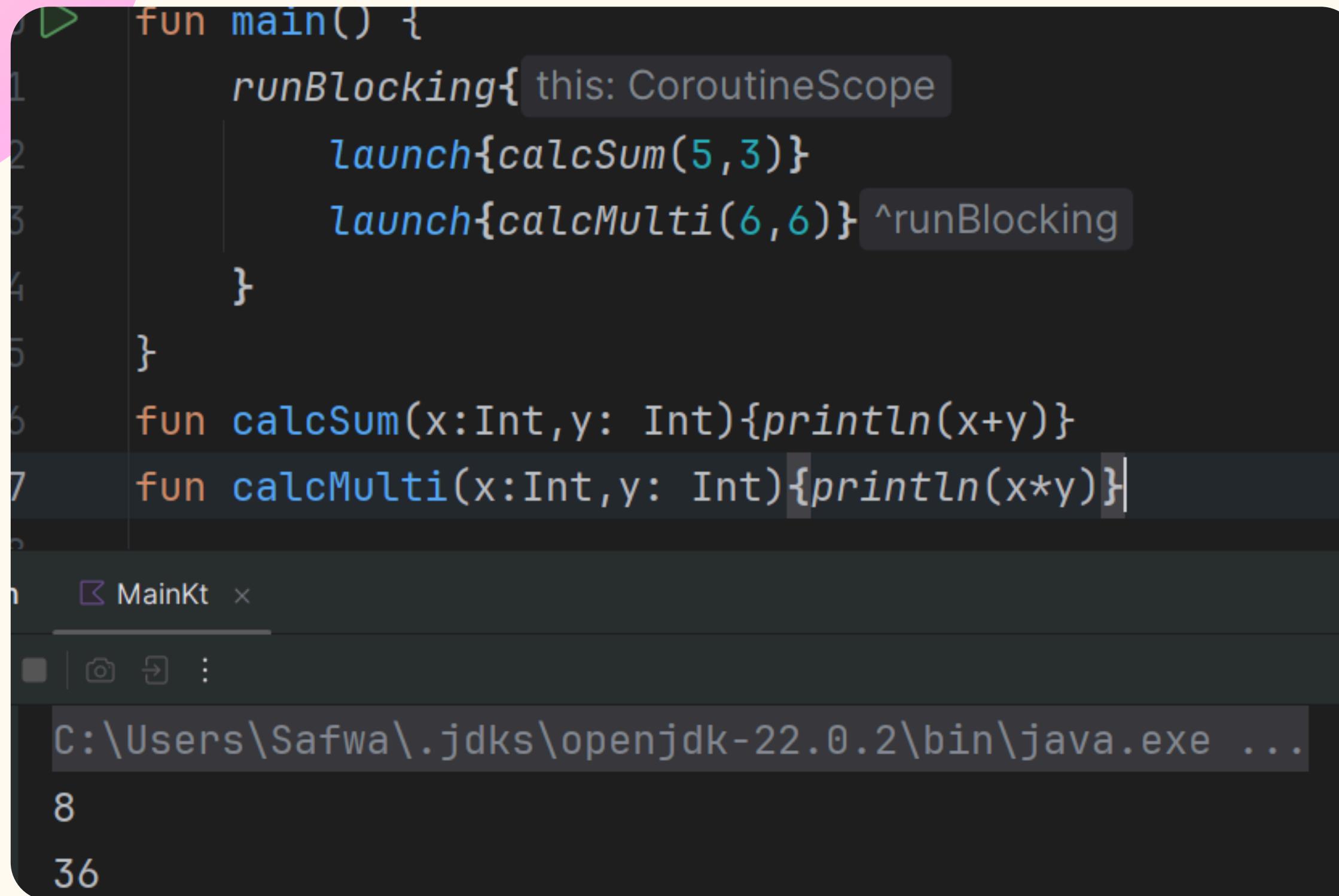
```
0 ► fun main() {
1     GlobalScope.launch(){ this: CoroutineScope
2         launch{calcSum(5,3)}
3         launch{calcMulti(6,6)}
4     }
5 }
6 fun calcSum(x:Int,y: Int){println(x+y)}
7 fun calcMulti(x:Int,y: Int){println(x*y)}
```

Why !! How to solve?



RunBlocking !!

Force the main to not be terminated until its body ended



The screenshot shows an IDE interface with a dark theme. A file named 'MainKt' is open, containing the following Kotlin code:

```
fun main() {
    runBlocking{ this: CoroutineScope
        launch{calcSum(5,3)}
        launch{calcMulti(6,6)} ^runBlocking
    }
}
fun calcSum(x:Int,y: Int){println(x+y)}
fun calcMulti(x:Int,y: Int){println(x*y)}
```

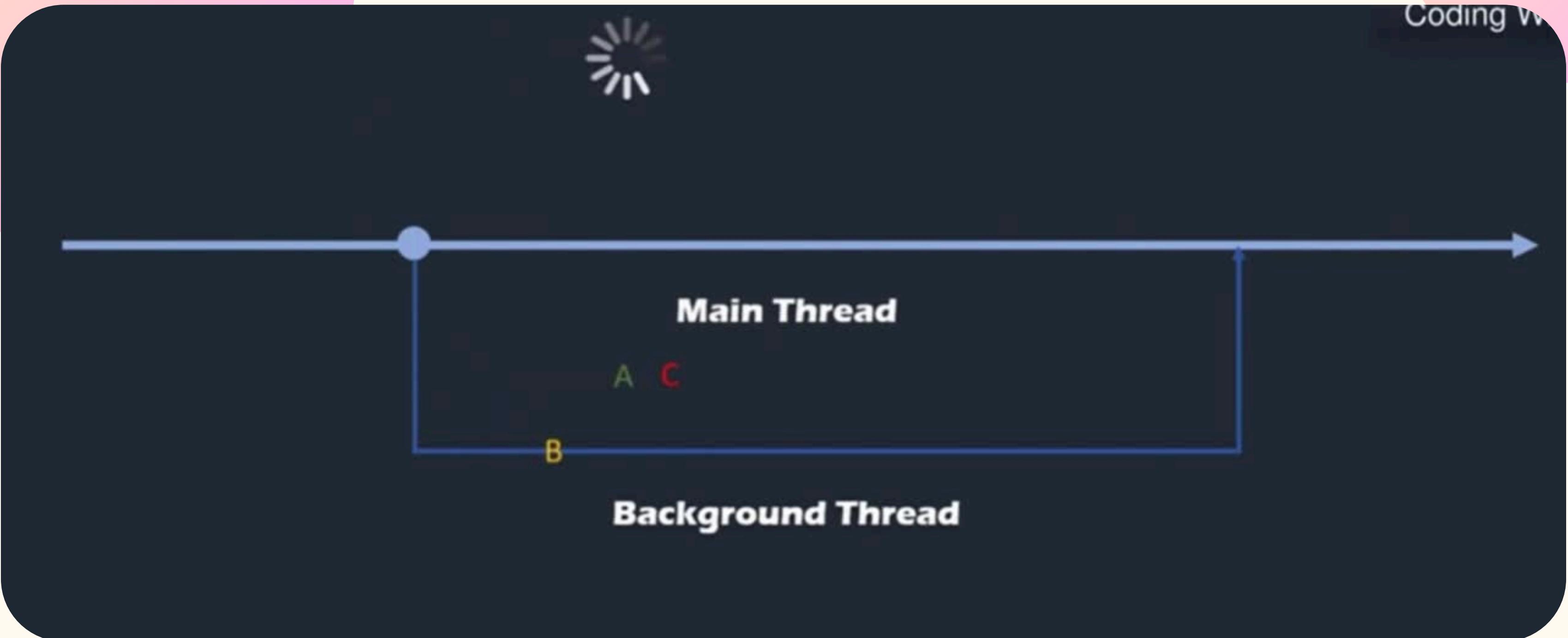
The code uses the `runBlocking` function to delay the termination of the `main` function until both parallel `launch` blocks have completed their execution.

The status bar at the bottom of the IDE shows the path: C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...

Page numbers 8 and 36 are visible at the bottom left of the slide.



Coding w

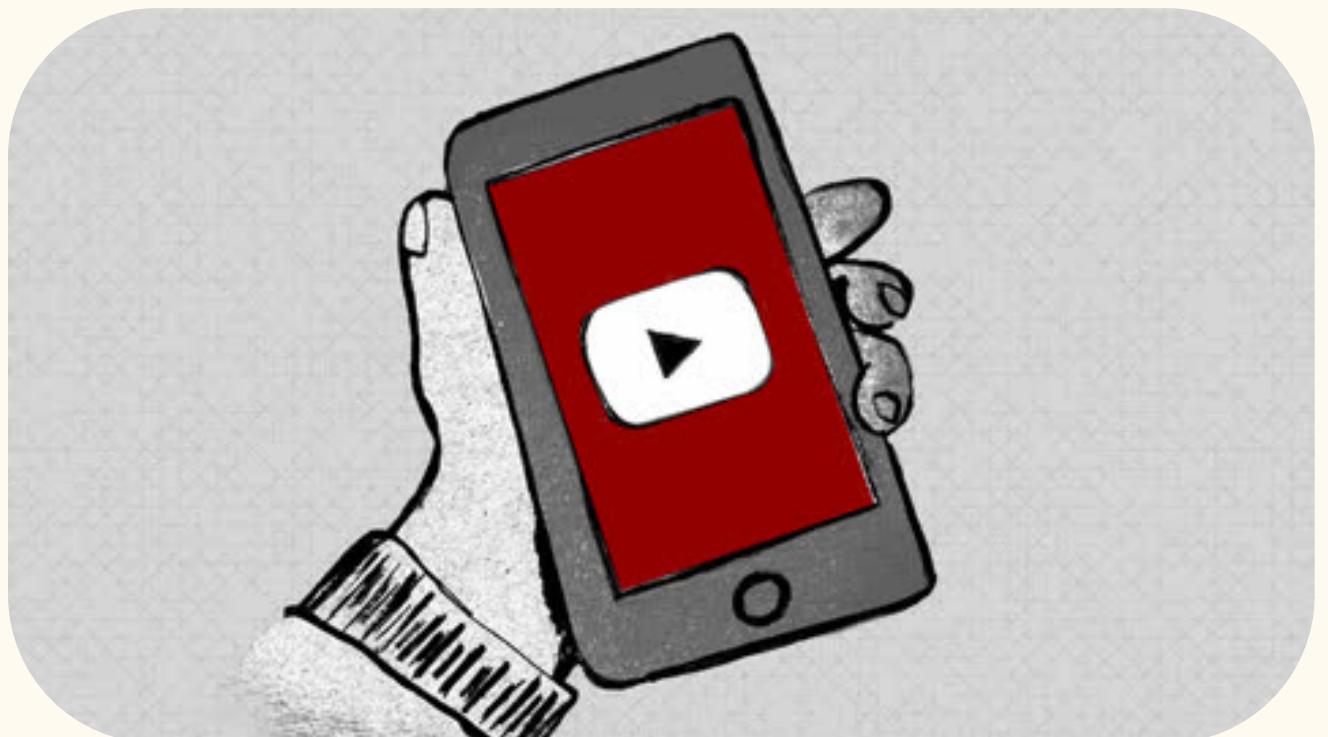


Suspended functions

Function the system has the ability to pause their execution and resume it later on , ***and it can not run out coroutine scope or another suspended function***

Example:

Imagine watching a YouTube video, you pause it, and when you resume, it **continues from the exact same point instead of restarting !**



Suspended functions

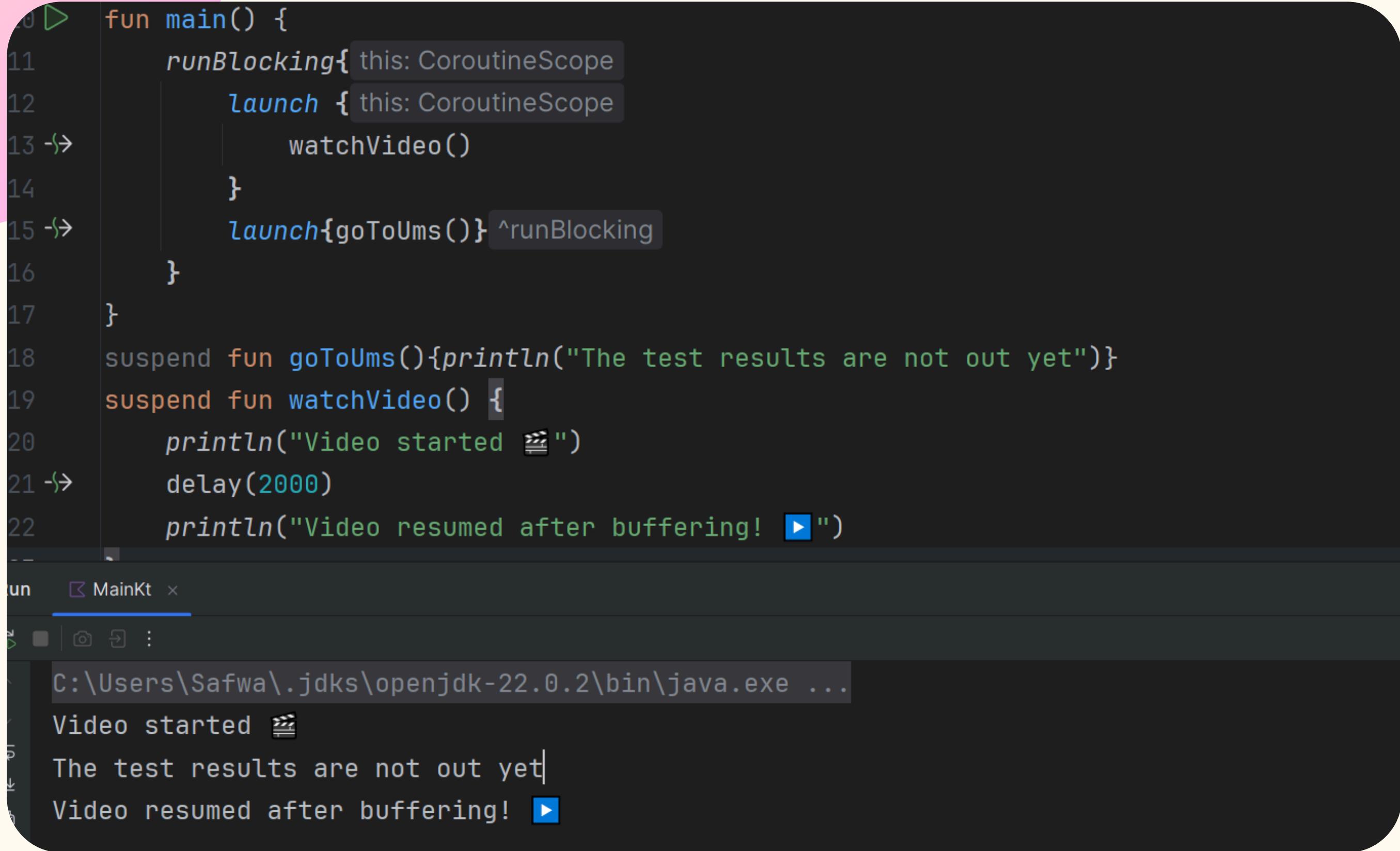
Example:

Imagine watching a YouTube video, you pause it, and when you resume, it **continues from the exact same point instead of restarting !**

```
suspend fun watchVideo() {  
    println("Video started 🎥")  
    delay(2000)  
    println("Video resumed after buffering! ⏴")  
}
```



Suspended functions



The image shows a screenshot of a Kotlin code editor. The code is as follows:

```
10 fun main() {
11     runBlocking{ this: CoroutineScope
12         launch { this: CoroutineScope
13             -> watchVideo()
14         }
15     -> launch{goToUms()} ^runBlocking
16     }
17 }
18 suspend fun goToUms(){println("The test results are not out yet")}
19 suspend fun watchVideo() {
20     println("Video started 🎥")
21     -> delay(2000)
22     println("Video resumed after buffering! ➡")
```

The code uses `runBlocking`, `launch`, and `suspend` functions. The `watchVideo()` function is annotated with `@Suspend`. The `main` function runs `goToUms()` in a separate coroutine.

The terminal output at the bottom shows the execution of the code:

```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Video started 🎥
The test results are not out yet
Video resumed after buffering! ➡
```



Dispatchers

Defines **thread pools** to launch your Kotlin Coroutines in.

They are used to switch between the main and background threads

Types:

- **Dispatchers.Default**→ "Heavy Calculator": Optimized for CPU-intensive tasks, such as complex calculations or processing.
- **Dispatchers.IO**→ "File Worker": Dedicated to I/O-bound operations, including file reading and writing, as well as network requests.



Dispatchers Types:

- **Dispatchers.Main** → "UI Handler": Used for operations that interact with the user interface, ensuring updates are performed on the main thread.
- **Dispatchers.Unconfined** → "Wherever Available": Not tied to any specific thread; the coroutine can be executed on whichever thread is available at the moment.



Dispatchers Types:

newSingleThreadContext → "One-Man Army": Creates a custom thread, providing a dedicated environment for tasks that need to run on a single thread.



```
0 ▶ fun main() {  
1     GlobalScope.launch(Dispatchers.)  
2 }  
3  
4  
5  
6
```

The screenshot shows an Android Studio code editor with a completion dropdown menu open over the line `GlobalScope.launch(Dispatchers.)`. The dropdown lists several dispatcher types: IO, Main, Default, Unconfined, and others like ToString and Equals. The 'IO' option is currently selected. The code editor interface includes tabs for 'MainKt' and 'MainKt.kt', and standard navigation icons at the bottom.



Press Ctrl+ to choose the selected (or first) suggestion and insert a dot afterwards Next Tip :

Dispatchers Types:

Example:

```
9
10 > fun main() {
11     runBlocking { this: CoroutineScope
12         launch(Dispatchers.Unconfined) { this: CoroutineScope
13             println("Running on: ${Thread.currentThread().name}")
14             delay(1000)
15             println("Still running on: ${Thread.currentThread().name}")
16
17         }
18     }
19 }
```

Run MainKt

```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Running on: main
Still running on: kotlinx.coroutines.DefaultExecutor
Process finished with exit code 0
```



Dispatchers Types:



Example:

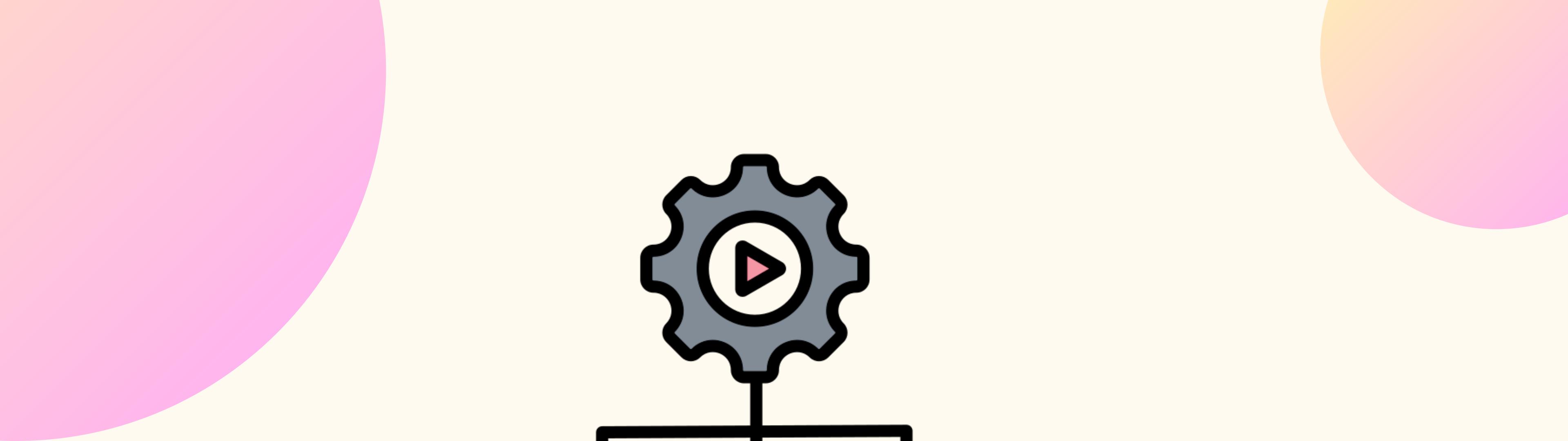
```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         val myNewThread= newSingleThreadContext("MyNewThread")
14         launch(myNewThread) { this: CoroutineScope
15             println("Running on: ${Thread.currentThread().name}")
16         } ^runBlocking
17     }
18 }
```

Run MainKt

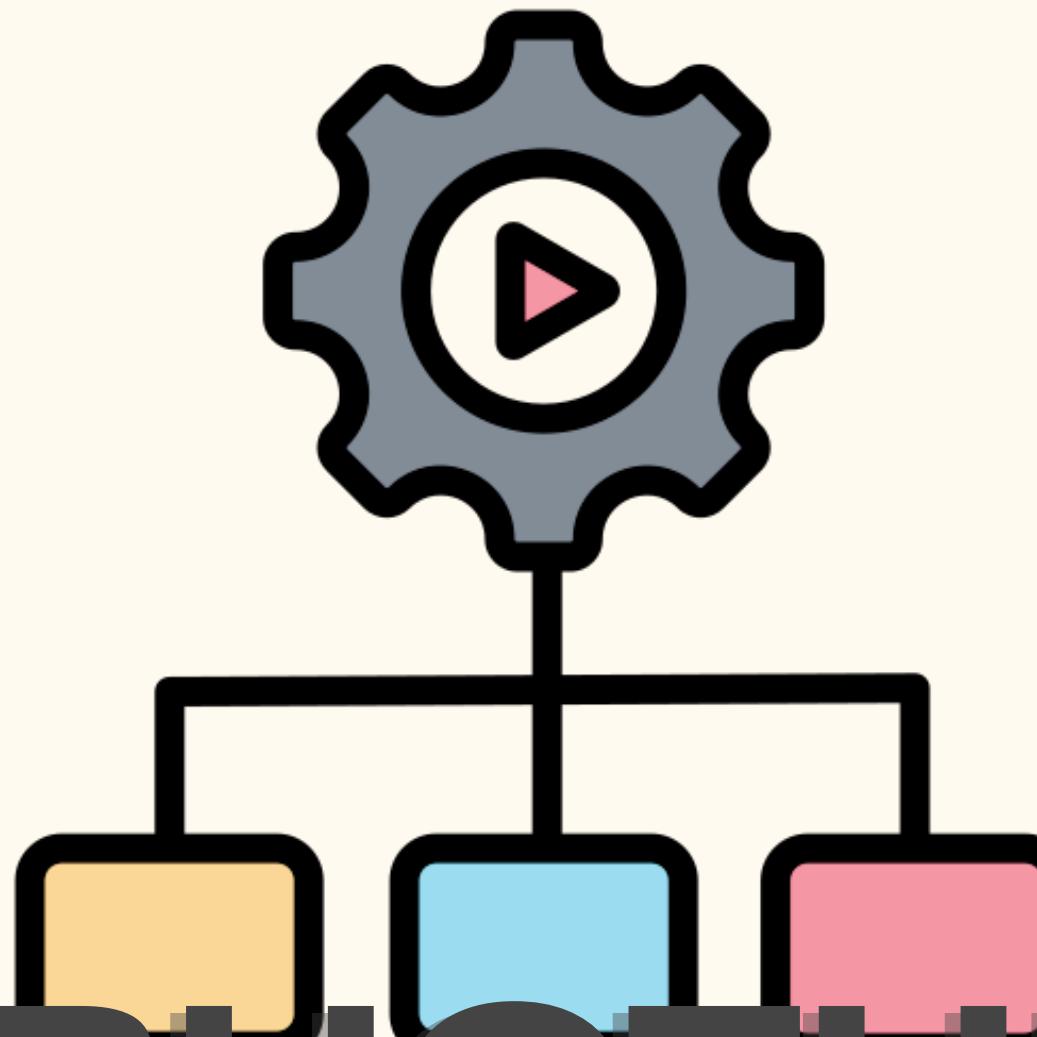


```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Running on: MyNewThread
```



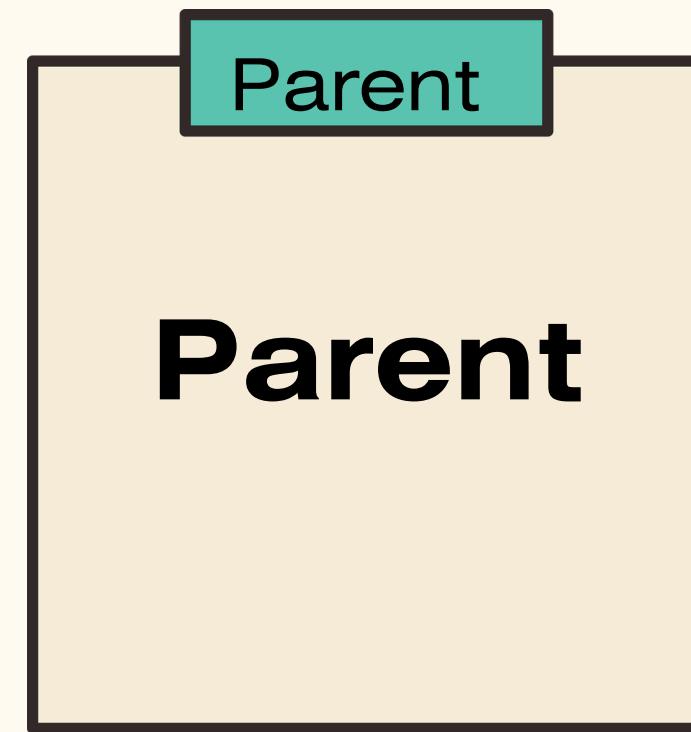


STRUCTURED CONCURRENCY



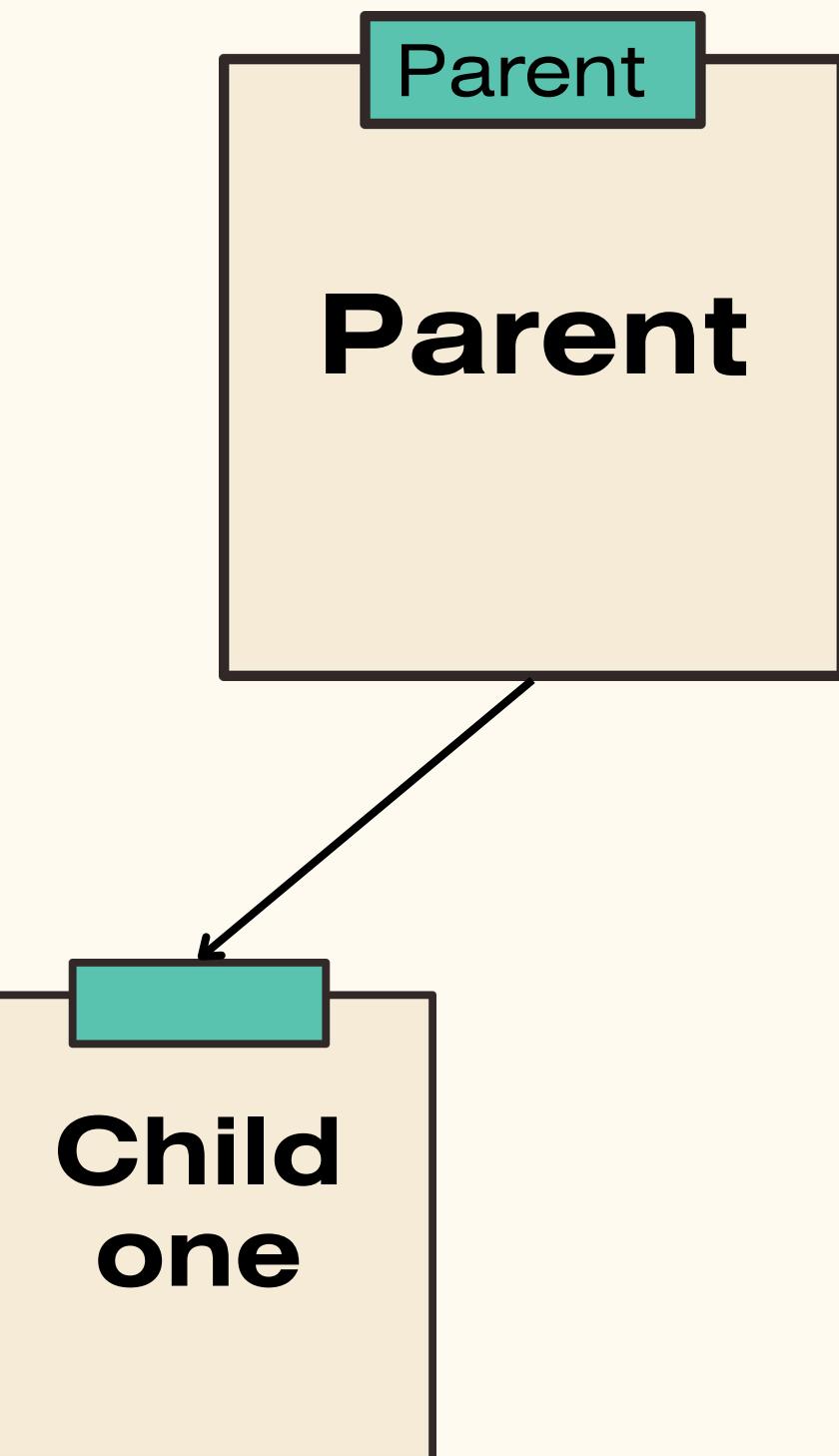
Structured concurrency

```
> fun main() {  
    runBlocking { this: CoroutineScope  
  
    }  
}
```



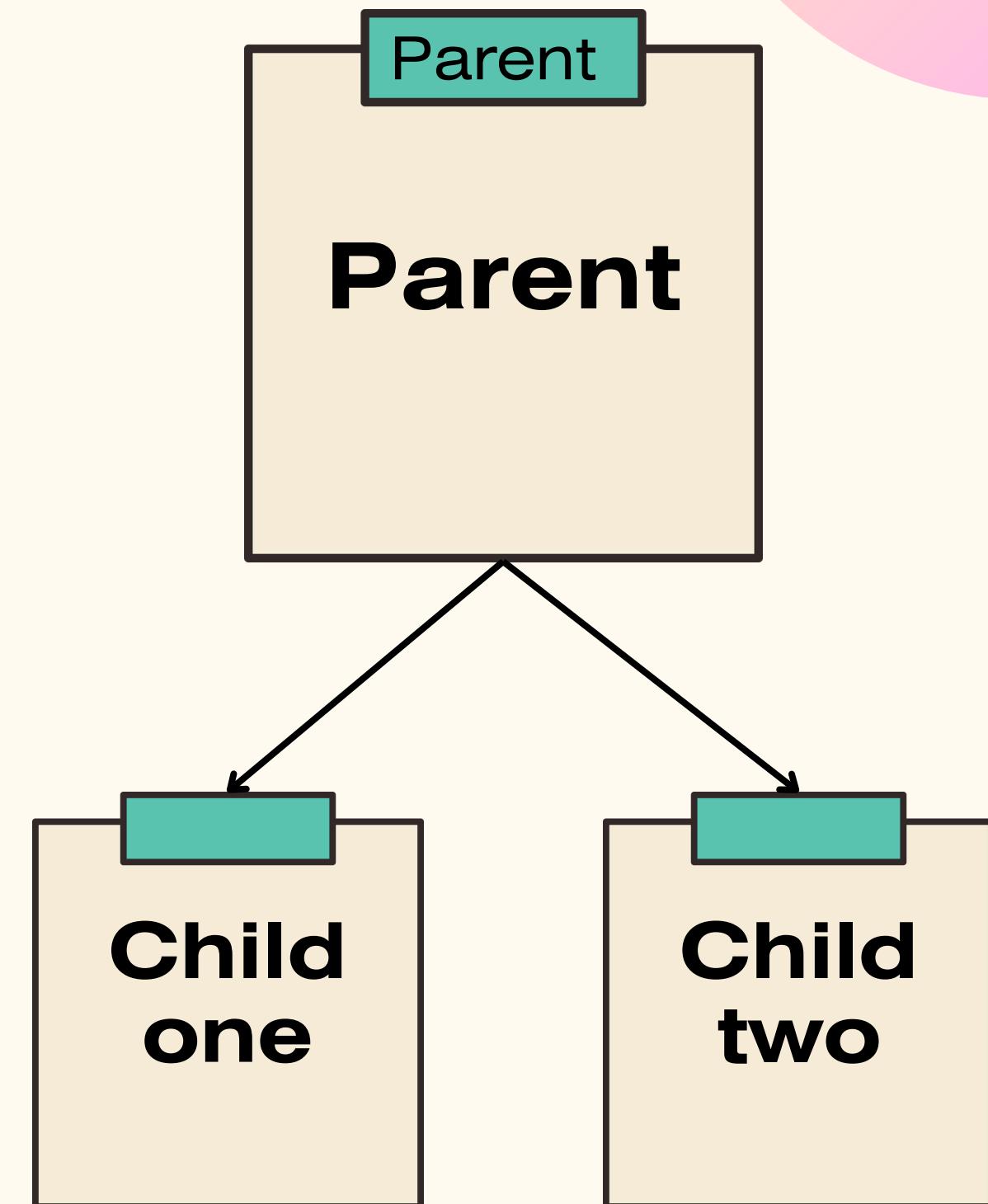
Structured concurrency

```
fun main() {  
    runBlocking { this: CoroutineScope  
        launch{ this: CoroutineScope  
            println("Child one")  
        }  
    }  
}
```

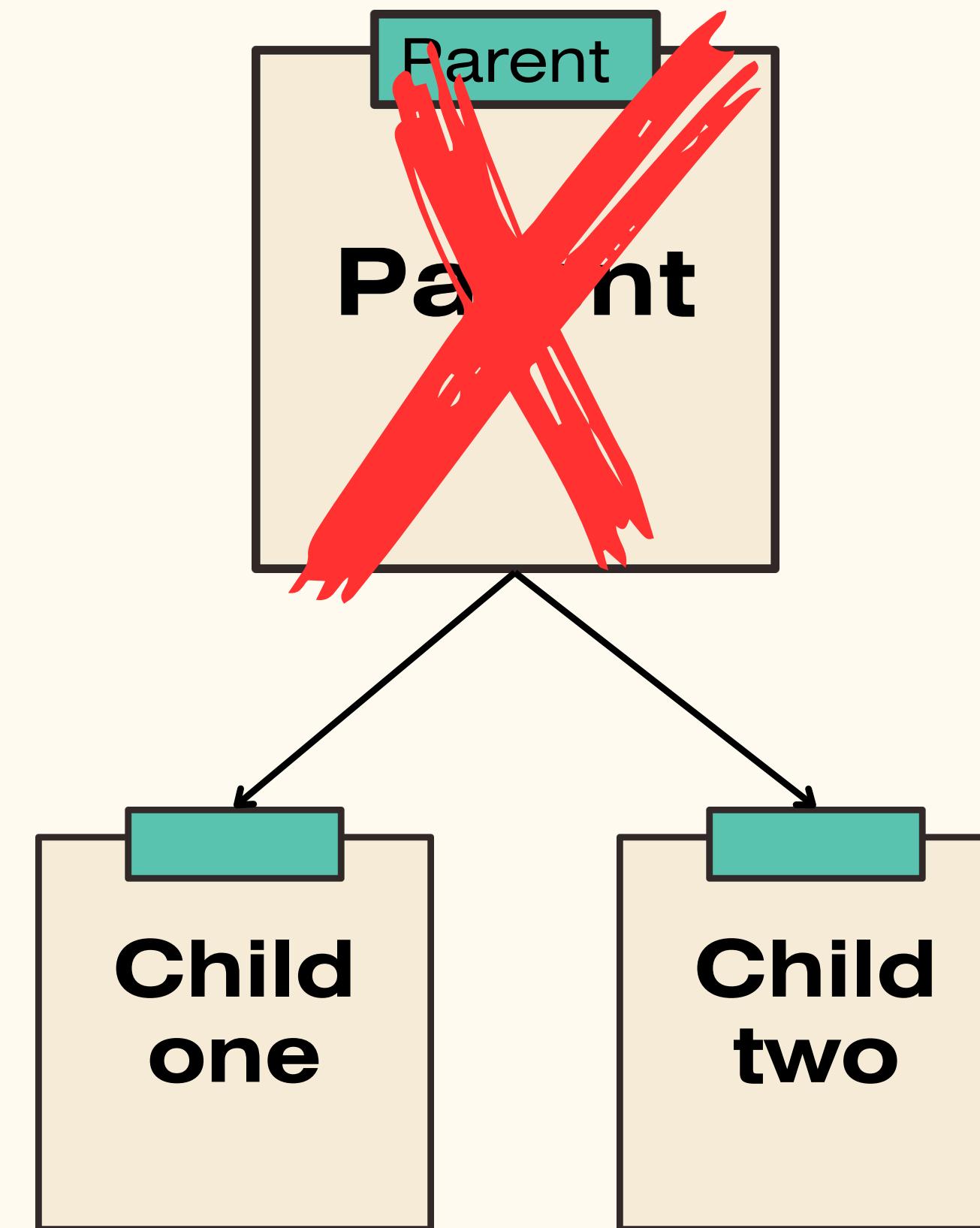


Structured concurrency

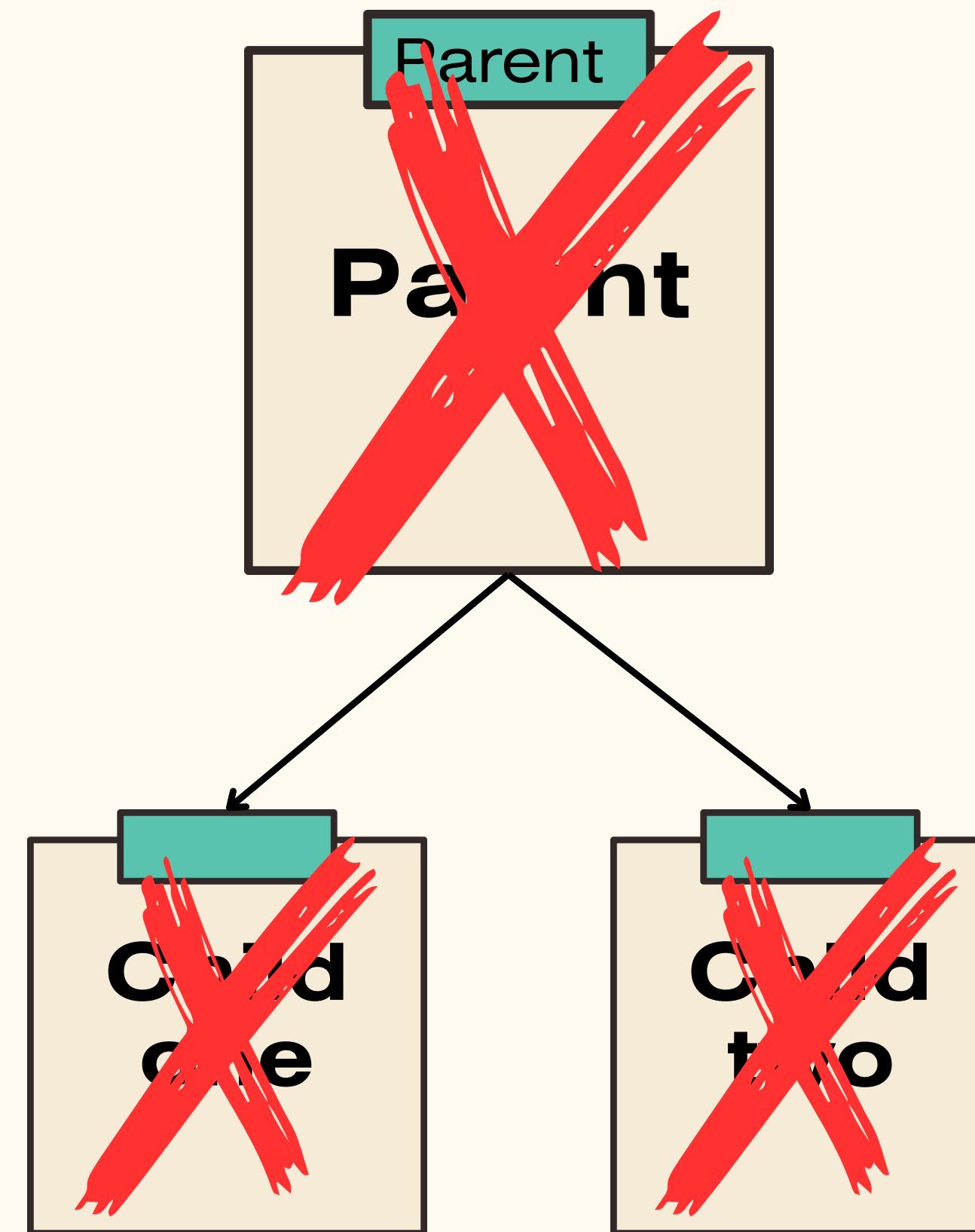
```
1 fun main() {  
2     runBlocking { this: CoroutineScope  
3         launch{ this: CoroutineScope  
4             println("Child one")  
5         }  
6         launch{ this: CoroutineScope  
7             println("Child two")  
8         } ^runBlocking  
9     }  
10 }
```



Structured concurrency



Structured concurrency



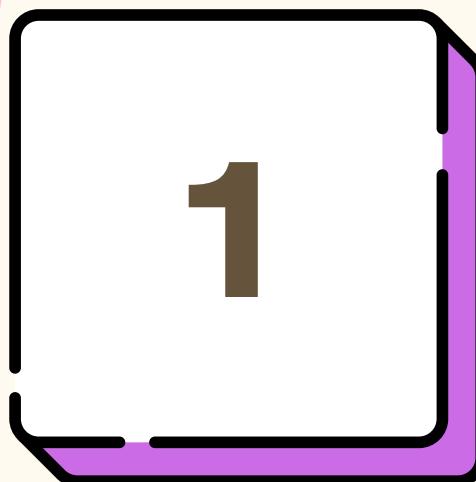
Structured concurrency

Structured concurrency in Kotlin Coroutines ensures that coroutines are managed hierarchically within a defined scope. A parent scope controls the lifecycle of its child coroutines, ensuring they are automatically completed or canceled when the parent finishes or is canceled.

```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         launch{ this: CoroutineScope
14             println("In the first child")
15             delay(1000)
16             println("Ending first")
17         }
18         launch{ this: CoroutineScope
19             println("In the second child")
20             delay(500)
21             println("Ending two")
22     } ^runBlocking
23 }
```



Structured concurrency



Can we
make job 3
don't run
until 2 and 1
finish ?!



Jobs

- **Reference to coroutine scope or a coroutine**, and a part of the context of the Coroutine.
- **To better manage a Coroutine**, a job is provided when we launch (or async, etc.).
- Jobs can be hierarchical (parent-child relationship).
- If a launch is triggered in another coroutine (under the same scope context), the job of the launch will be made the child job of the coroutine.



Jobs

job.join():Suspends the current coroutine until the given job finishes.

```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         println("Main program starts")
14         val job = launch { this: CoroutineScope
15             println("Coroutine starts")
16             delay(1000)
17             println("Coroutine ends")
18         }
19         delay(500)
20         job.join()
21         println("Main program ends")
22     }
}
Run MainKt x
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Main program starts
Coroutine starts
Coroutine ends
Main program ends
```



Jobs

job.cancel(): Cancels
the coroutine

```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         println("Main program starts")
14         val job = launch { this: CoroutineScope
15             repeat(1000){ it: Int
16                 println(it)
17             }
18         }
19     ->
20         delay(10)
21         println("Job is canceling")
22         job.cancel()
23         println("Main program ends")
24     }
```

Run MainKt ×



```
996
997
998
999
Job is canceling
Main program ends
```



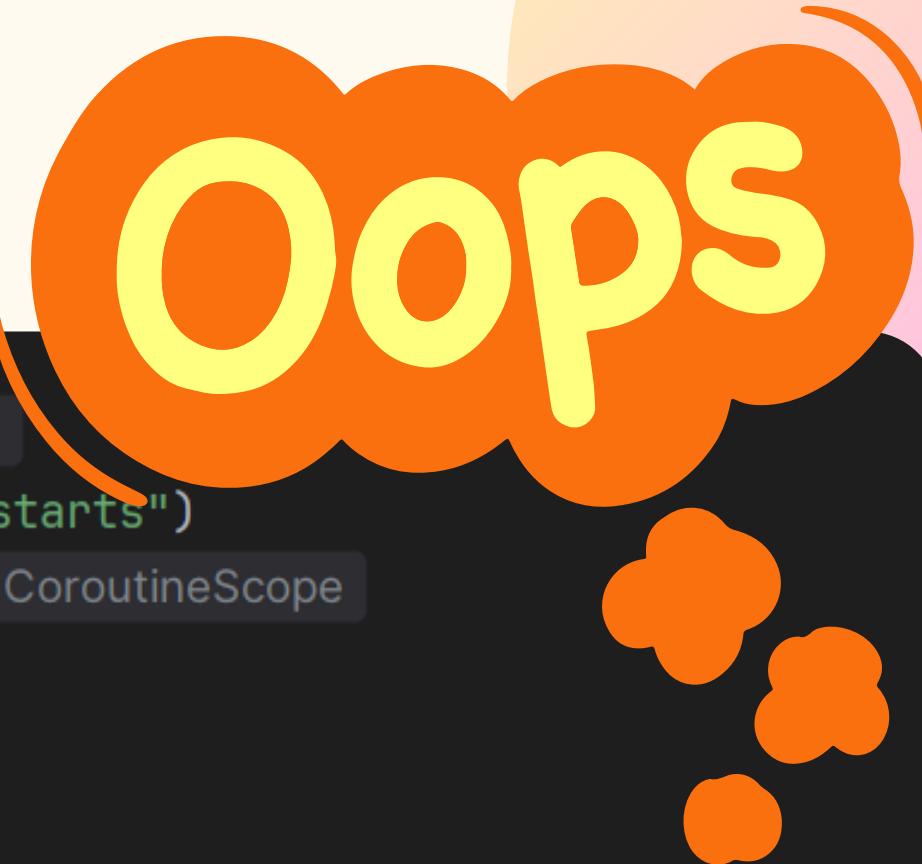
Jobs

**job.cancel(): Cancels
the coroutine**

```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         println("Main program starts")
14         val job = launch { this: CoroutineScope
15             repeat(1000){ it: Int
16                 println(it)
17             }
18         }
19     ->
20         delay(10)
21         println("Job is canceling")
22         job.cancel()
23         println("Main program ends")
24 }
```

Run MainKt

996
997
998
999
Job is canceling
Main program ends



Jobs

Co-operative Code:

- To cancel a coroutine, it must include cooperative mechanisms that check for cancellation requests.
- Without such mechanisms, the coroutine will continue to run even after cancellation is requested.



Jobs



Two Ways to Write Cooperative Code:

- **Using Built-in Suspend Functions:**

- Examples: `yield()`, `delay()`
 - These functions automatically check for cancellation requests and throw a `CancellationException` if the coroutine is canceled.

- **Explicitly Checking for Cancellation:**

- Example: `isActive`
 - This boolean property can be checked within the coroutine to determine whether it should stop.



Jobs

Co-operative Code:

```
11 > fun main() {
12     runBlocking { this: CoroutineScope
13         println("Main program starts")
14         val job = launch { this: CoroutineScope
15             repeat(1000){ it: Int
16                 println(it)
17                 delay(100)
18             }
19         }
20     }->
21         delay(1000)
22         println("Job is canceling")
23         job.cancel()
24         println("Main program ends")
```

Run MainKt

- Run
- Stop
- Start
- Cancel
- ⋮

↑	7
↓	8
⤵	9
⤷	Job is canceling
⤸	Main program ends





Time out

Sometimes, a coroutine needs to be canceled if it takes too long to complete. Kotlin provides two functions for this:

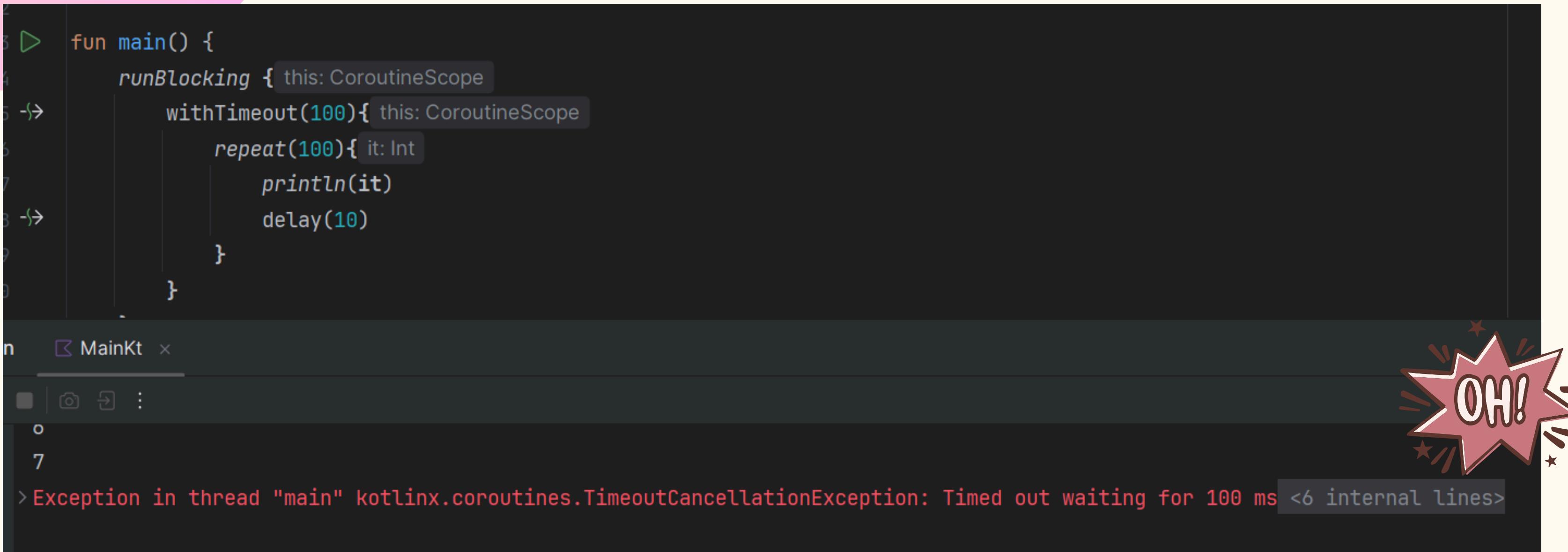
- **withTimeout(time)** : Runs a coroutine for a set time. If it exceeds that time, it throws a `TimeoutCancellationException` and stops execution.
- **withTimeoutOrNull(time)** :-Runs a coroutine for a set time. If it exceeds that time, it does not throw an exception, but instead returns null and allows the program to continue.

These functions help prevent long-running tasks from blocking your program.



Time out

- **withTimeout(time)** :

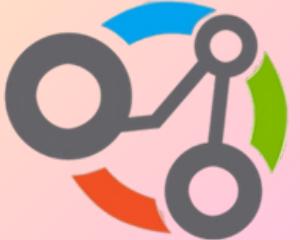


```
3 D fun main() {
4     runBlocking { this: CoroutineScope
5     ->     withTimeout(100){ this: CoroutineScope
6         |     repeat(100){ it: Int
7             |         println(it)
8             |         delay(10)
9         }
10    }
11 }
```

MainKt

Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out waiting for 100 ms <6 internal lines>

OH!



Time out

- **withTimeoutOrNull(time) :**

```
24 D fun main() {  
25     runBlocking { this: CoroutineScope  
26         -> withTimeoutOrNull(100){ this: CoroutineScope  
27             repeat(100){ it: Int  
28                 println(it)  
29                 delay(10)  
30             }  
31         }  
32     }  
33 }
```

Run MainKt x



4
5
6

Process finished with exit code 0

Thank You

