# ANDROID

## SESSION THREE

# Agenda

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


# UI FREEZING

# UI freezing issue

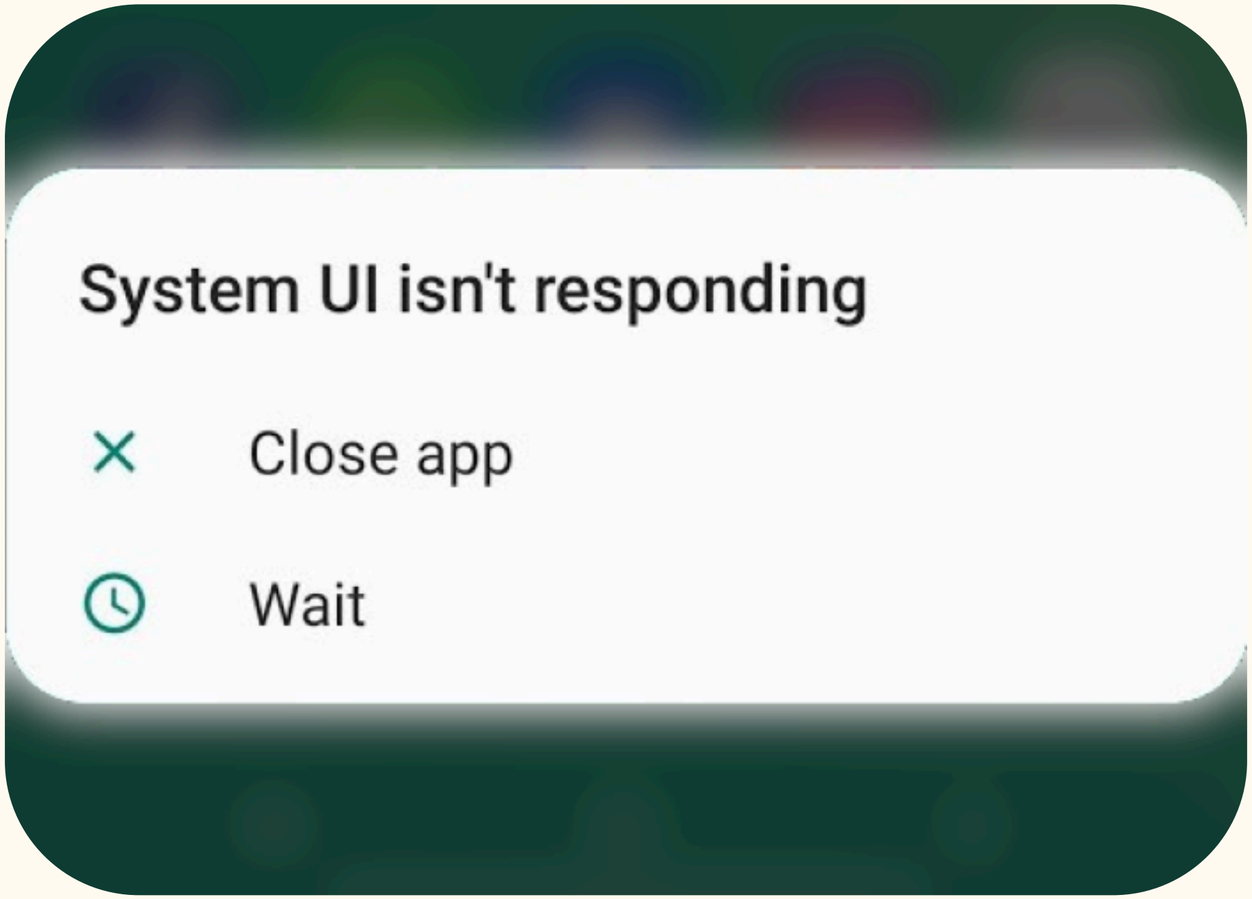
## Why Does the UI Freeze in Android? 🤔

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

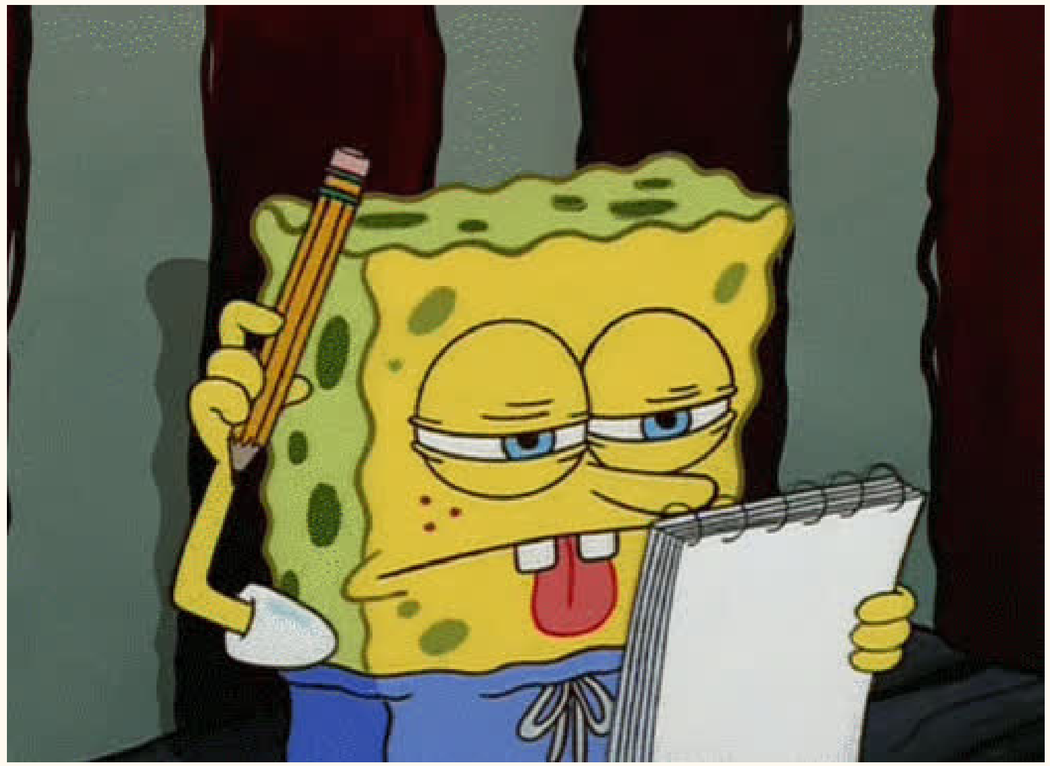
## Why Does the UI Freeze in Android? 🤔



# 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

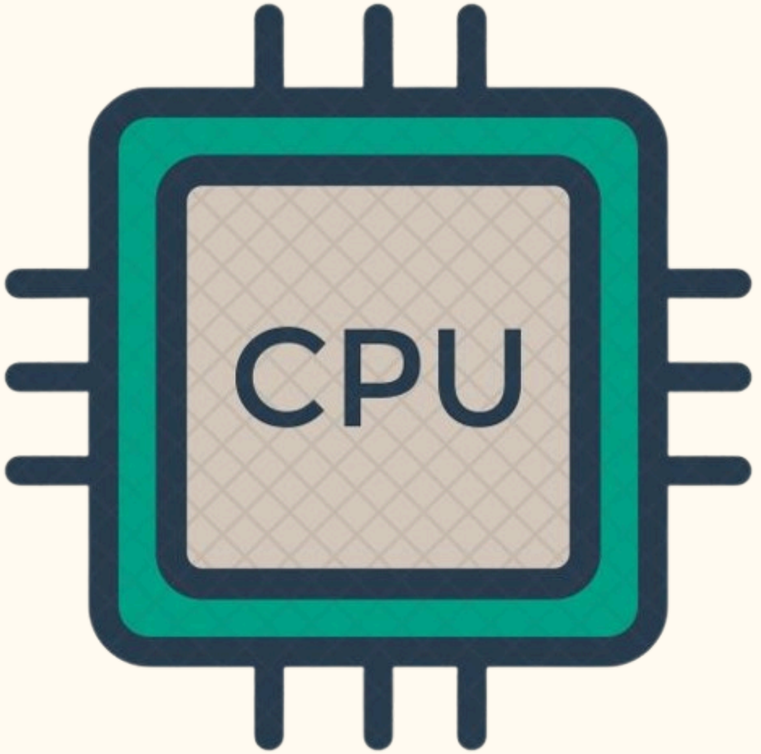
# Sequential Code vs Asynchronous Code

| 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.

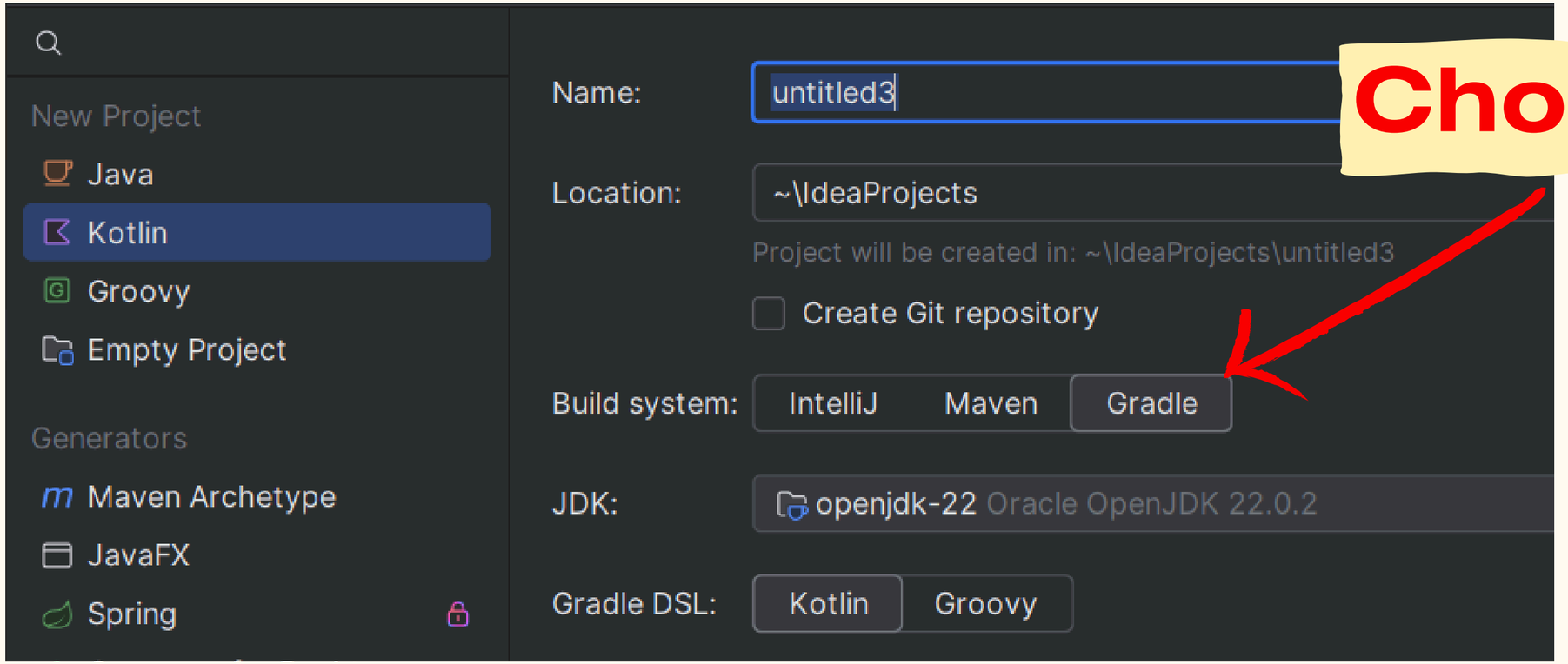
**How to write Asynchronous code in android  ?**  **Kotlin Coroutines**



# COROUTINES GETTING-READY

# Getting-Ready

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



# 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 !!

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

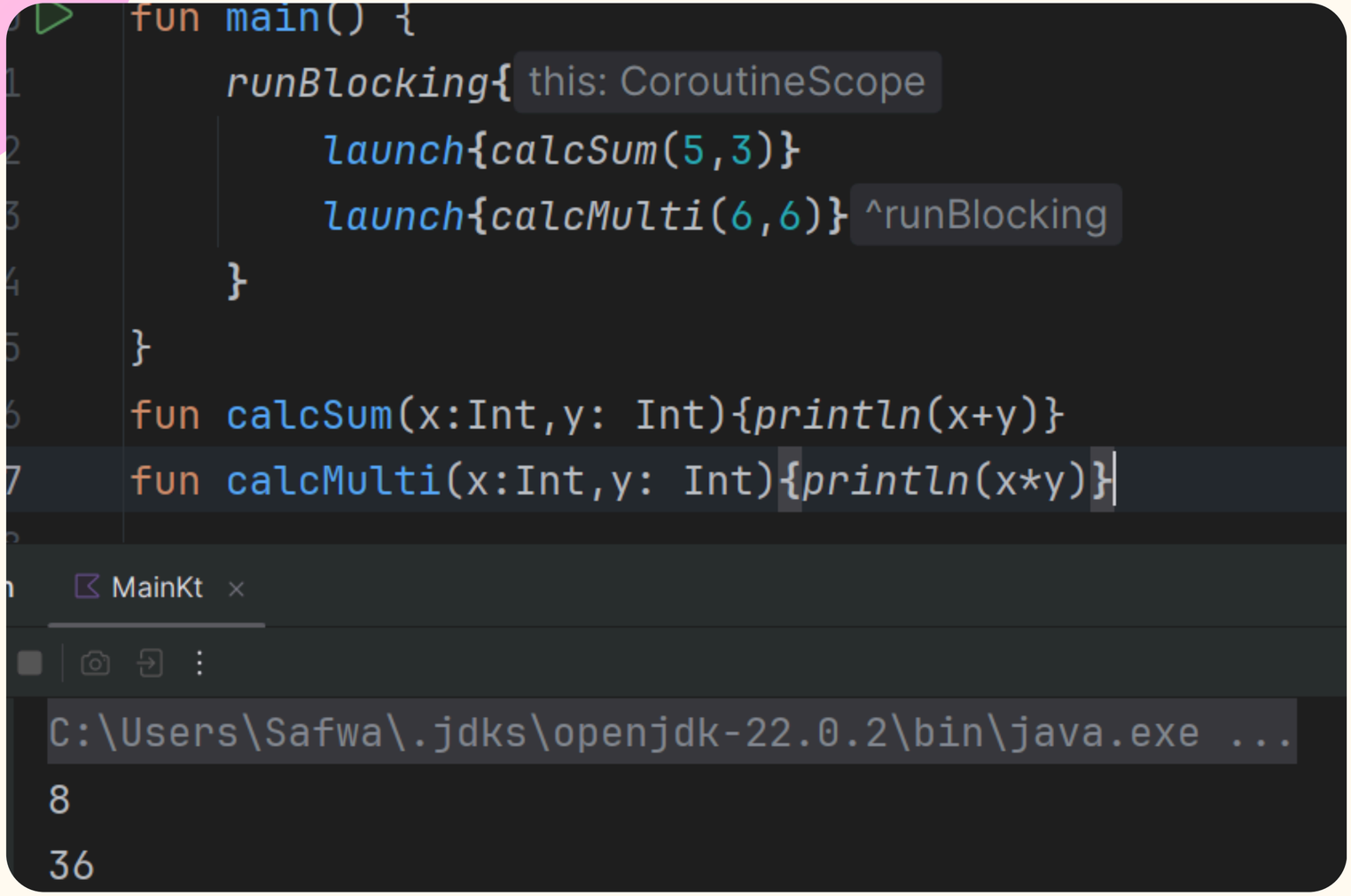
MainKt

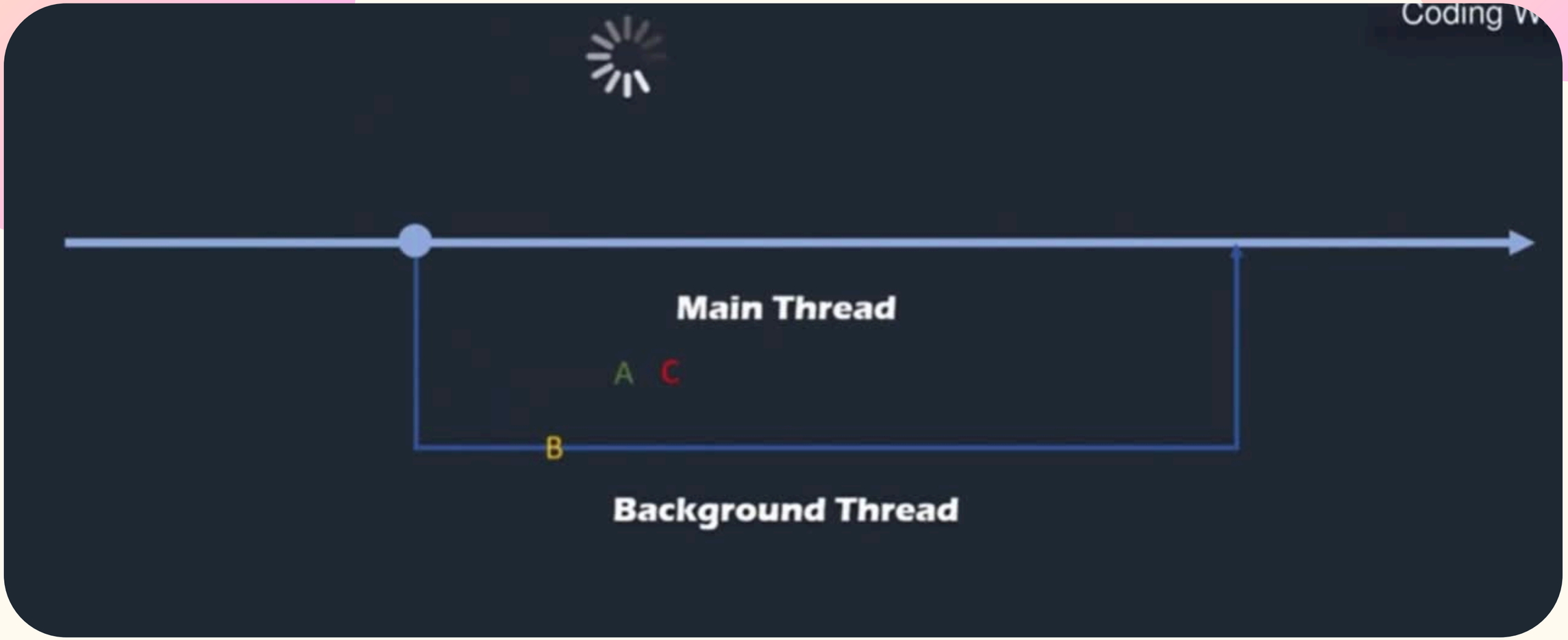
```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...

Process finished with exit code 0
```

Why !!
How to solve?

# RunBlocking !!

Force the main to not be terminated until its body ended




Coding W
Main Thread
A
C
B
Background Thread

# 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 !**

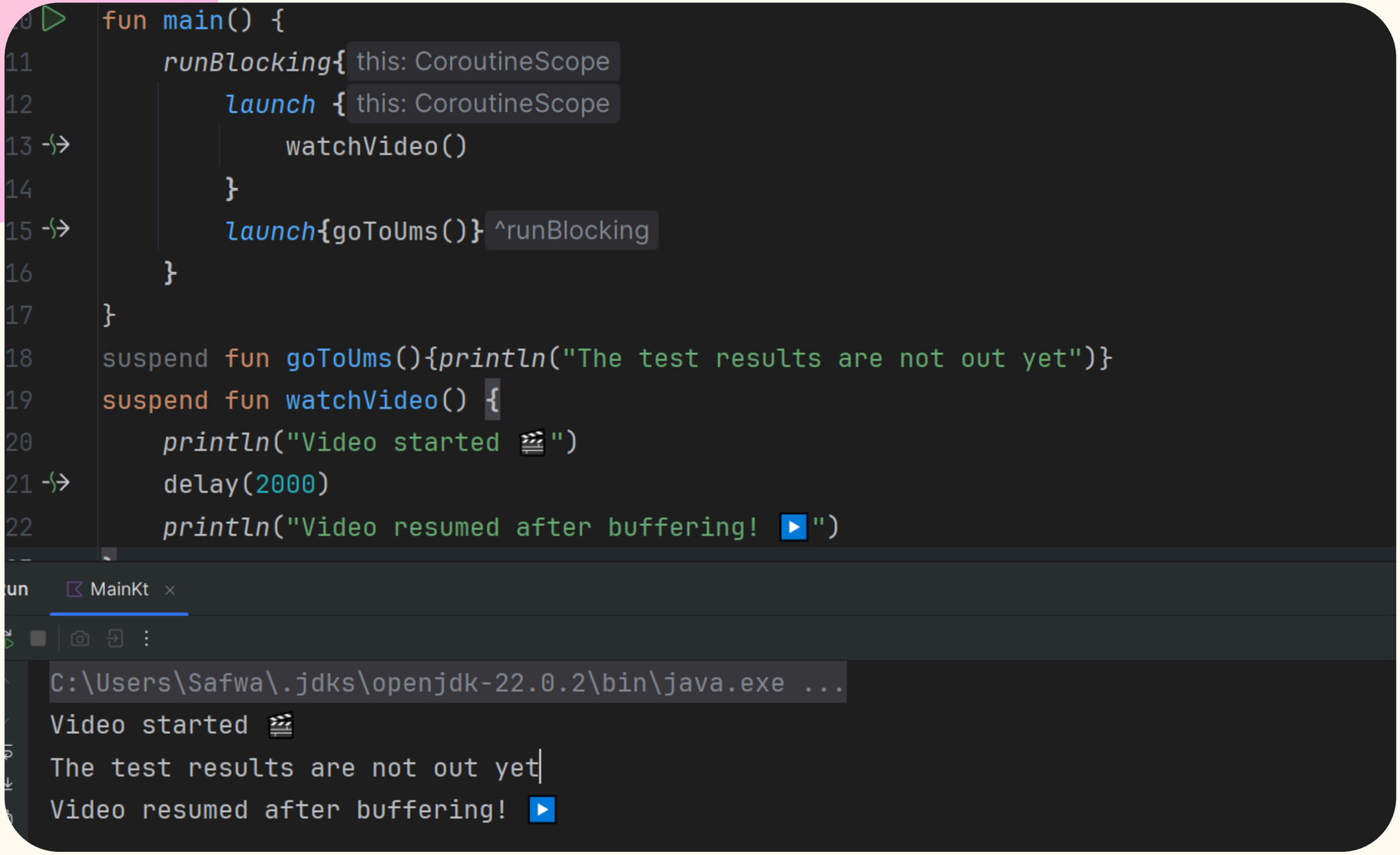
# Suspended functions

**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



# 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.

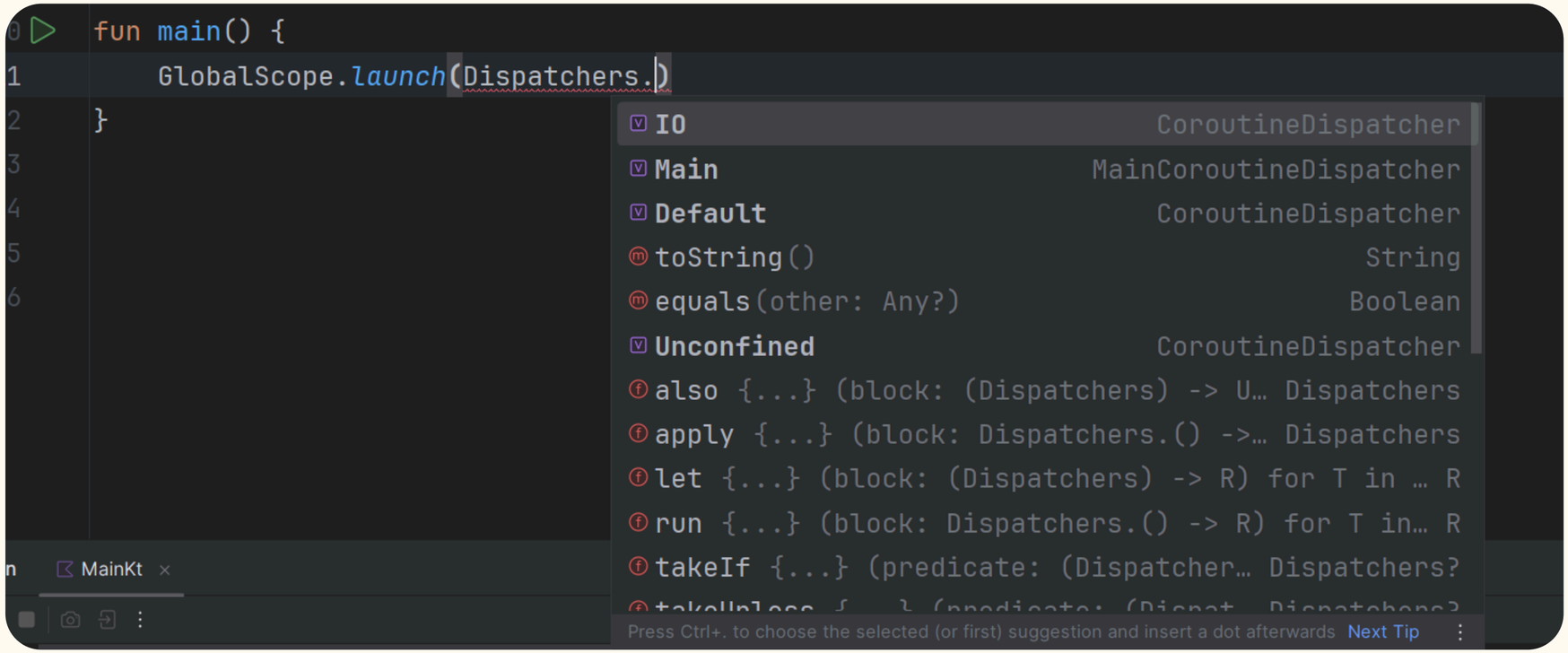
# Dispatchers Types:

- **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.



# Dispatchers Types:

## Example:

```kotlin
fun main() {
    runBlocking { this: CoroutineScope
        launch(Dispatchers.Unconfined) { this: CoroutineScope
            println("Running on: ${Thread.currentThread().name}")
            delay(1000)
            println("Still running on: ${Thread.currentThread().name}")

        }
    }
}
```

```
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
```

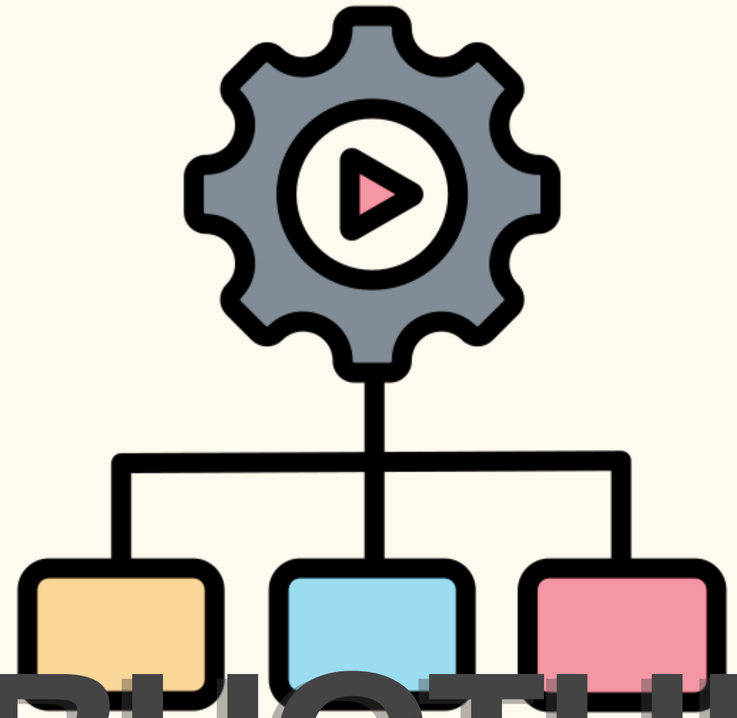
# Dispatchers Types:

## 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 ×

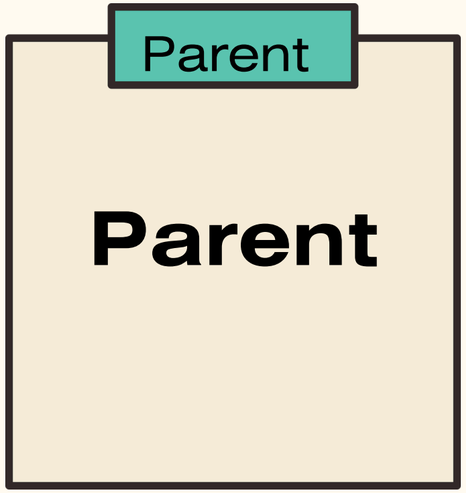
```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Running on: MyNewThread
```



# STRUCTURED CONCURRENCY

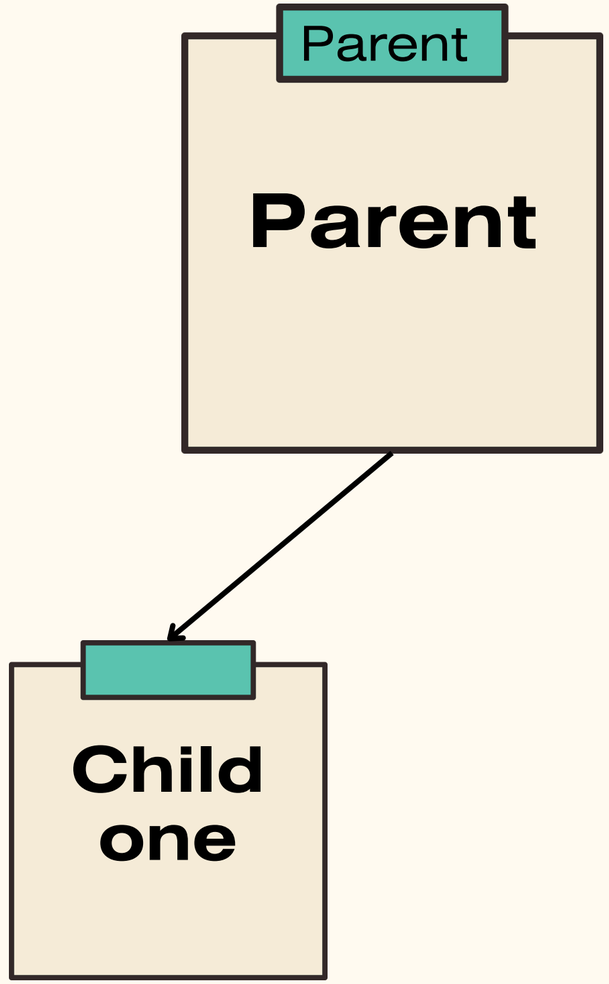
# Structured concurrency

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

    }
}
```



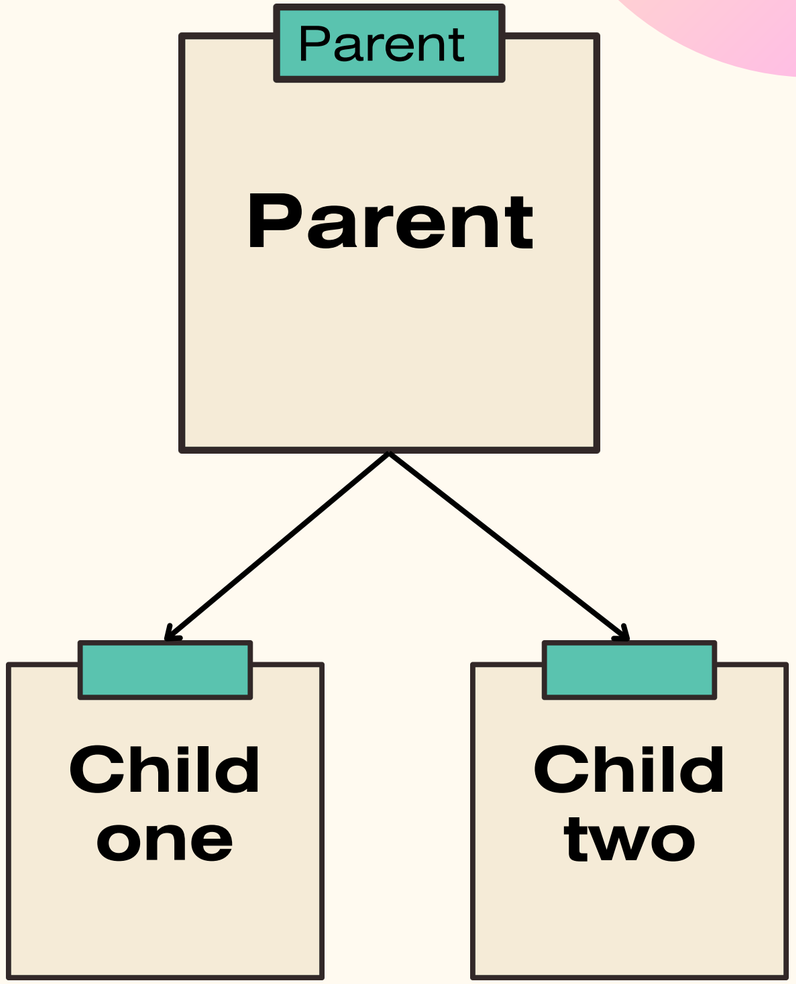

# Structured concurrency

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

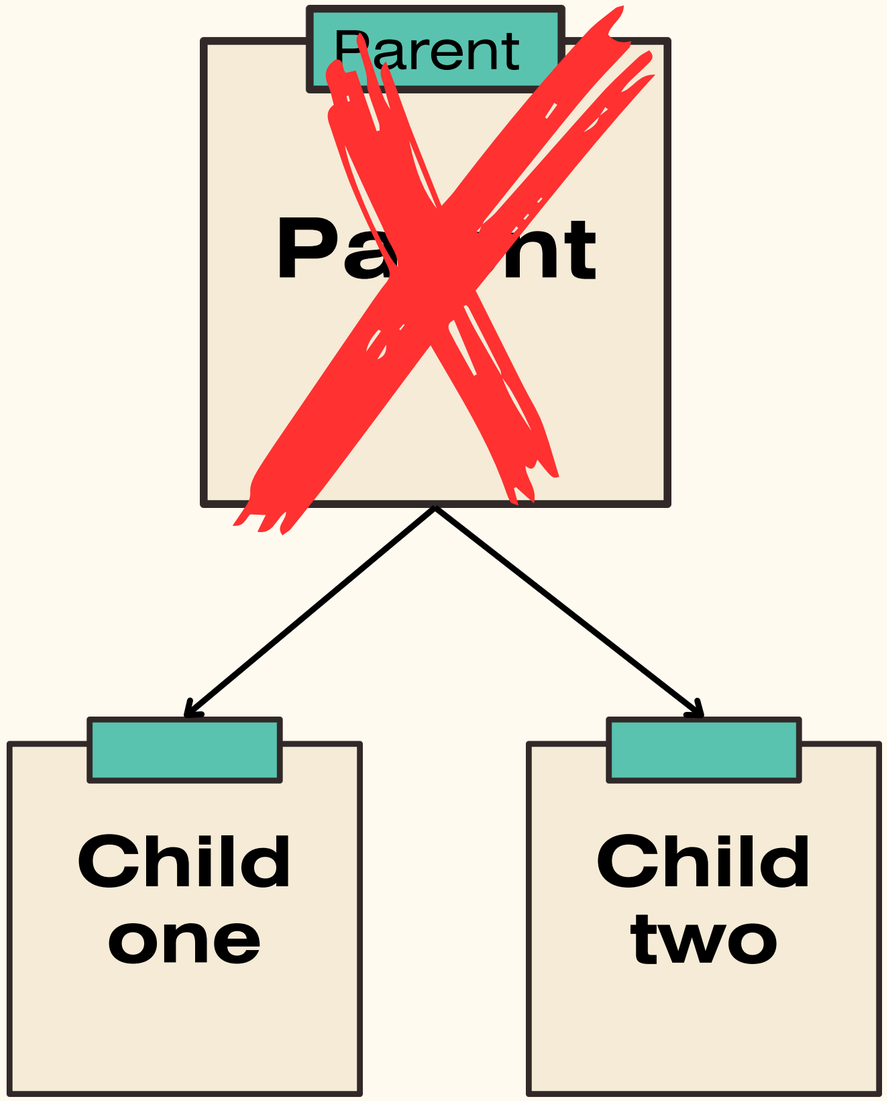



# Structured concurrency

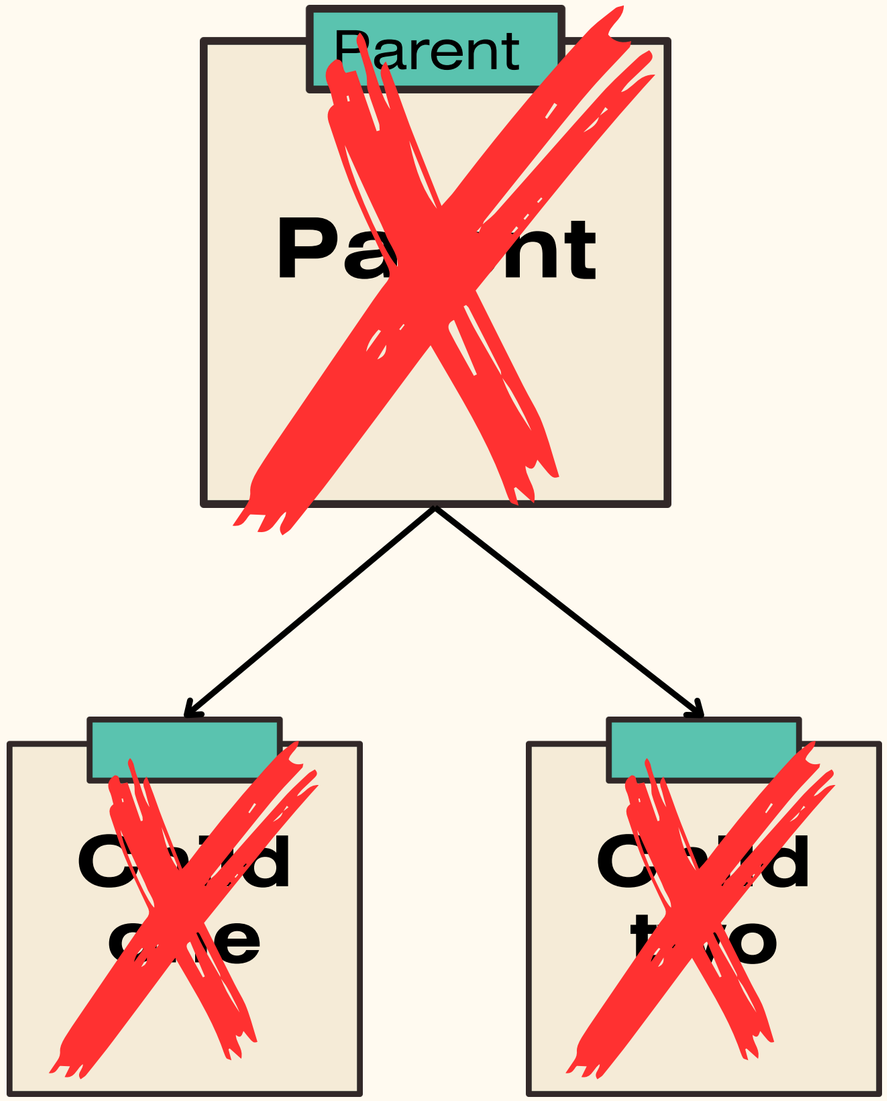
```
fun main() {
    runBlocking { this: CoroutineScope
        launch{ this: CoroutineScope
            println("Child one")
        }
        launch{ this: CoroutineScope
            println("Child two")
        } ^runBlocking
    }
}
```




# Structured concurrency

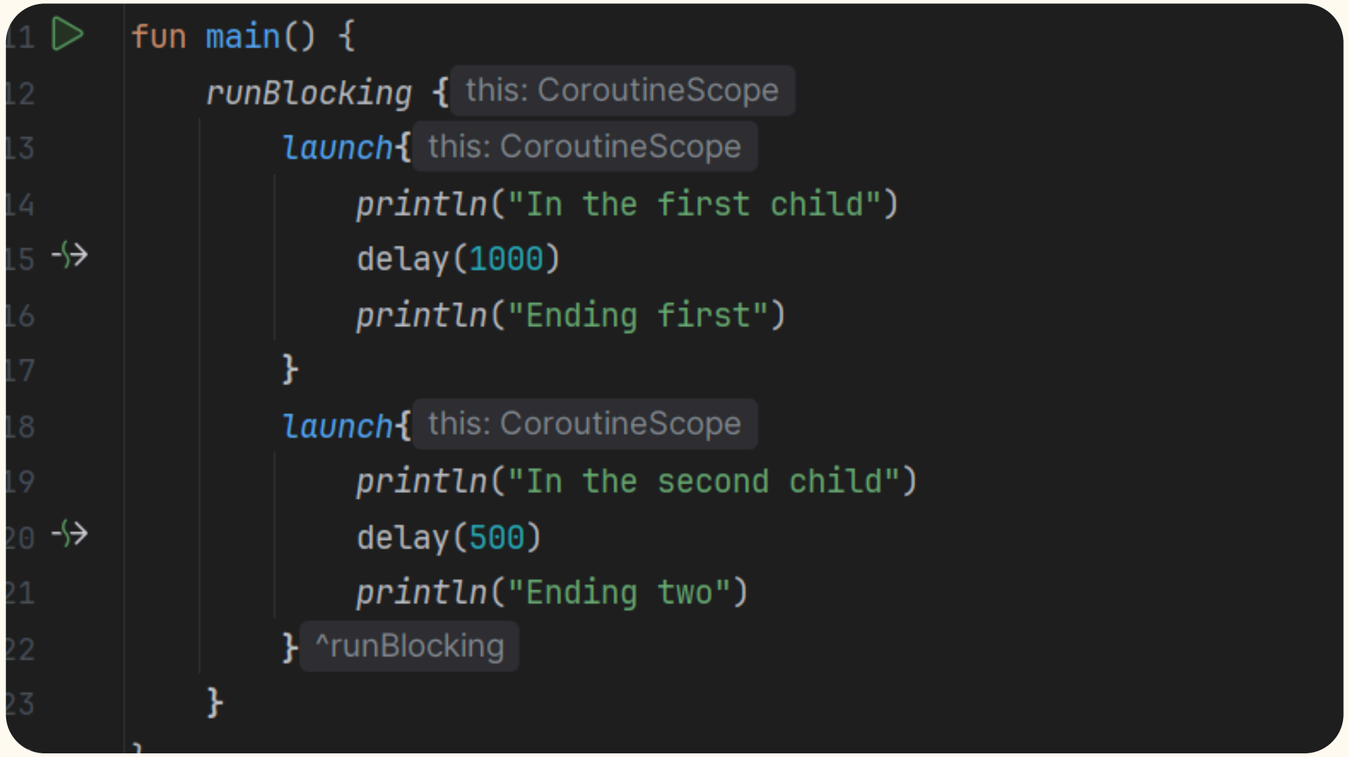


# Structured concurrency



# 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.




# 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.

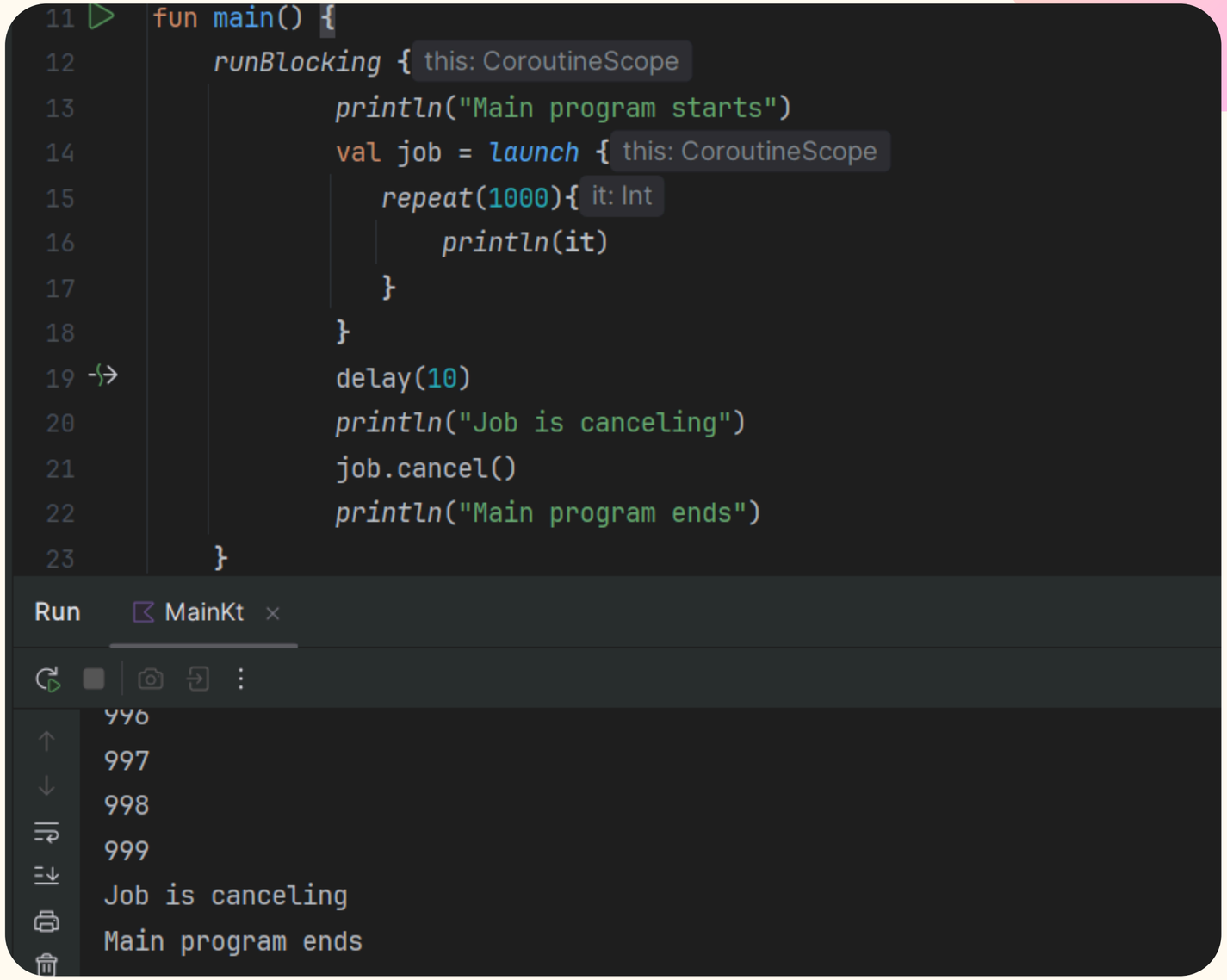
```
11  fun main() {
12      runBlocking { this: CoroutineScope
13          println("Main program starts")
14          val job = launch { this: CoroutineScope
15              println("Coroutine starts")
16              delay(1000)
17              println("Coroutine ends")
18          }
19          delay(500)
20          job.join()
21          println("Main program ends")
22      }
```

Run MainKt

```
C:\Users\Safwa\.jdks\openjdk-22.0.2\bin\java.exe ...
Main program starts
Coroutine starts
Coroutine ends
Main program ends
```

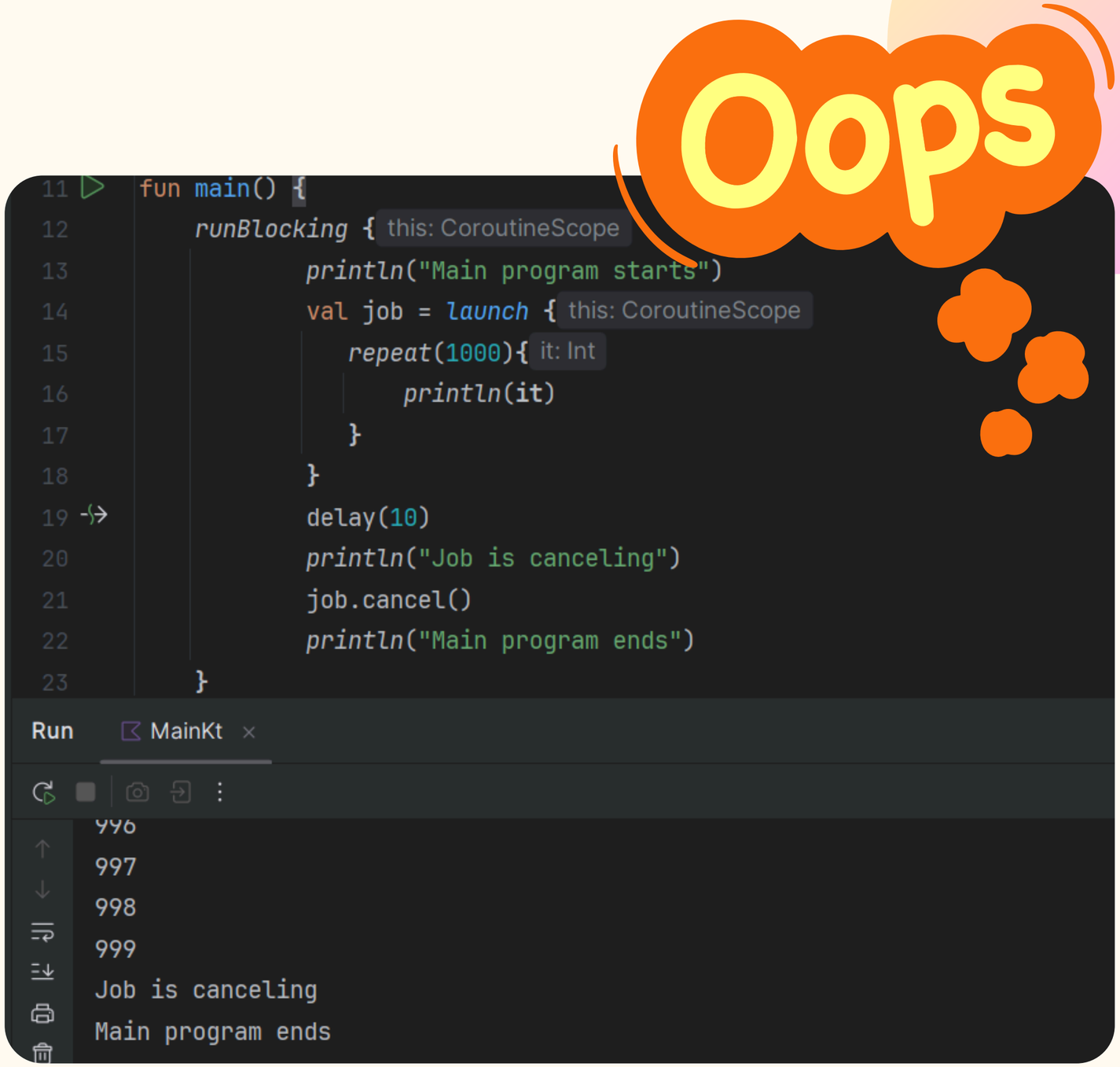
# Jobs

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



# Jobs

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



# 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          delay(1000)
21          println("Job is canceling")
22          job.cancel()
23          println("Main program ends")
```

Run MainKt ×

```
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) :**

```
fun main() {
    runBlocking { this: CoroutineScope
        withTimeout(100){ this: CoroutineScope
            repeat(100){ it: Int
                println(it)
                delay(10)
            }
        }
```

```
n   MainKt ×
0
7
Exception in thread "main" kotlinx.coroutines.TimeoutCancellationException: Timed out waiting for 100 ms <6 internal lines>
```

# Time out

- **withTimeoutOrNull(time) :**

```
fun main() {
    runBlocking { this: CoroutineScope
        withTimeoutOrNull(100){ this: CoroutineScope
            repeat(100){ it: Int
                println(it)
                delay(10)
            }
        }
```

Run MainKt

```
4
5
6

Process finished with exit code 0
```

# Thank You