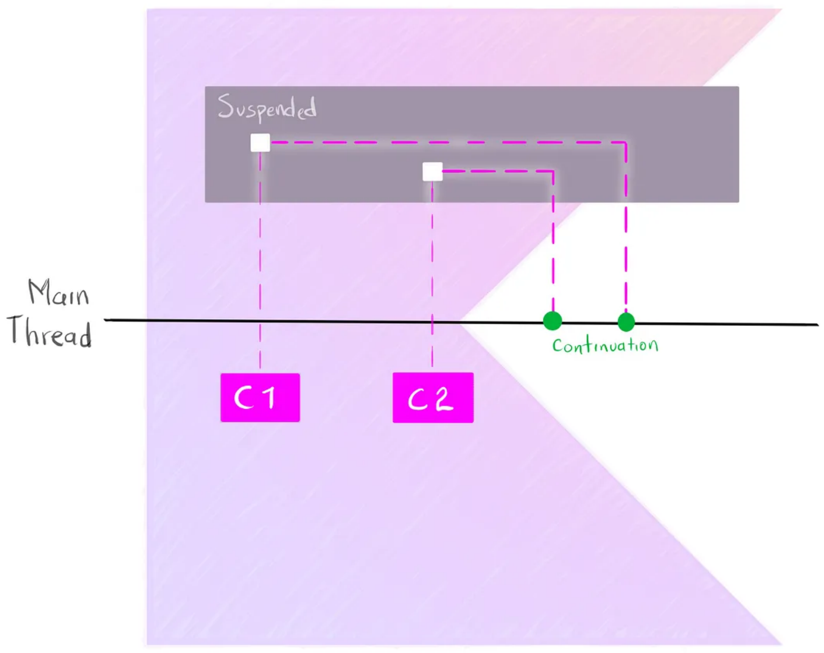
# Kotlin Coroutines Concepts

Jorge Luis Castro Medina · Follow
6 min read · Jan 9, 2024

279 2



### What is a Coroutine?

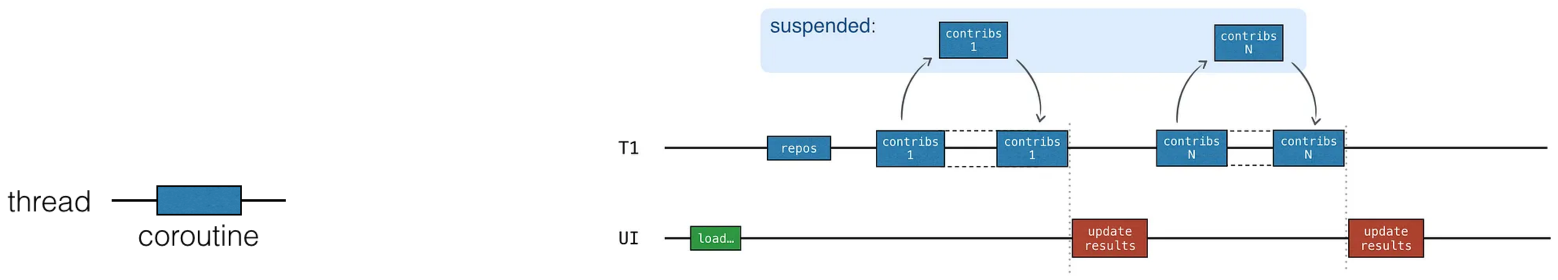
Coroutines, a concept not exclusive to Kotlin, have been officially available in the language since version 1.3. Unlike traditional threads or multiprocessing, coroutines represent a subroutine that can transfer its execution flow to another part of the program and resume later. This ability to suspend execution allows efficient multitasking without blocking the current thread. Essentially, coroutines serve as cooperative functions for task execution.

The Kotlin Coroutines leverages the principle of structured concurrency which mean each coroutine should be launched inside a specific context with a determinate life-time.

Coroutine is part of kotlinx.coroutines, which is a library developed by JetBrains that gives us all the ease and power of coroutines in Kotlin. Cool, right? I think so, You can find the kotlinx.coroutines repository on Github.

## Suspending Function

Functions that can suspend the execution of a coroutne, they are used for tasks that can take time to complete, such as reading or writing file data, connection to a database, or making a HTTP request.



## Coroutine Builders

These are functions for initializing or creating new coroutines. Let’s take a look:

- **launch**: Launches a new coroutine concurrently with the rest of the code. It returns a `Job`, which is a cancellable object with a life cycle that culminates upon completion.

```
fun main() = runBlocking<Unit> { // this: CoroutineScope
    launch {
        delay(1000L)
        println("Hello World!")
    }
}
```

- **withContext**: Allows you to change the execution context of a block of code in Kotlin, thereby controlling the thread or thread pool in which it will be executed. The result of the executed code is returned as the function’s result, or `Unit` if there is no specific result.

```
fun main() = runBlocking {
    val data = withContext(Dispatchers.IO) {
        fetchData()
    }
    println("Response = $data")
}

suspend fun fetchData(): String {
    return "Hello world!"
}
```

withContext example

- **async**: Initiates a separate coroutine, which is a lightweight thread that works concurrently with all the other coroutines and returns its future result as an implementation of the `Deferred` interface. The running coroutine is cancelled when the resulting deferred is cancelled.

```kotlin
fun main() = runBlocking<Unit> {

    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")

}
```

async example

- **coroutineScope**: This is a suspending function that allows us to declare our own scope. It creates a coroutine scope and does not complete until all launched children complete.

```kotlin
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

Open in Playground →    Target: JVM    Running on v.1.9.22

coroutineScope example

- **runBlocking**: This is a regular function similar to `coroutineScope`. The main difference is that the current thread gets blocked for the duration of the call until all the coroutines inside complete their execution, while `coroutineScope` just suspends.

```
fun main() = runBlocking { // this: CoroutineScope
    doWorld()
}

suspend fun doWorld() {
    delay(1000L)
    println("Hello World!")
}
```

Open in Playground → Target: JVM Running on v.1.9.22

runBlocking example

- produce: Creates a coroutine that produces a stream of values. The values are generated by the coroutine and sent to a channel. The channel can be used to receive the values produced by the coroutine.

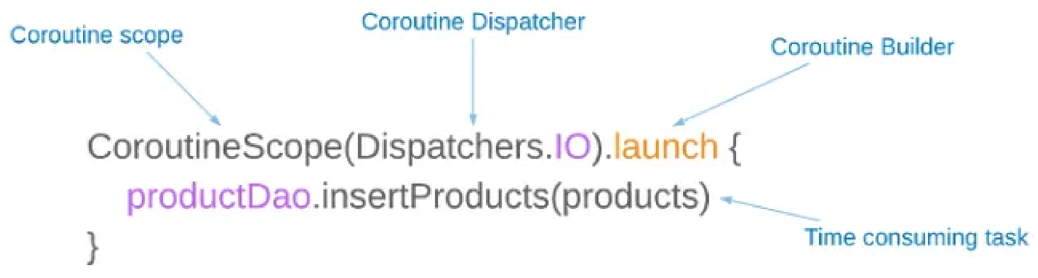
```
fun main() = runBlocking {
    // Create a channel to send numbers
    val channel = produce {
        for (i in 1..10) {
            send(i)
        }
    }

    // Consume the numbers from the channel
    for (i in channel) {
        println(i)
    }
}
```

Open in Playground → Target: JVM Running on v.1.9.22

produce example

## CoroutineScope



```
CoroutineScope(Dispatchers.IO).launch {
    productDao.insertProducts(products)
}
```

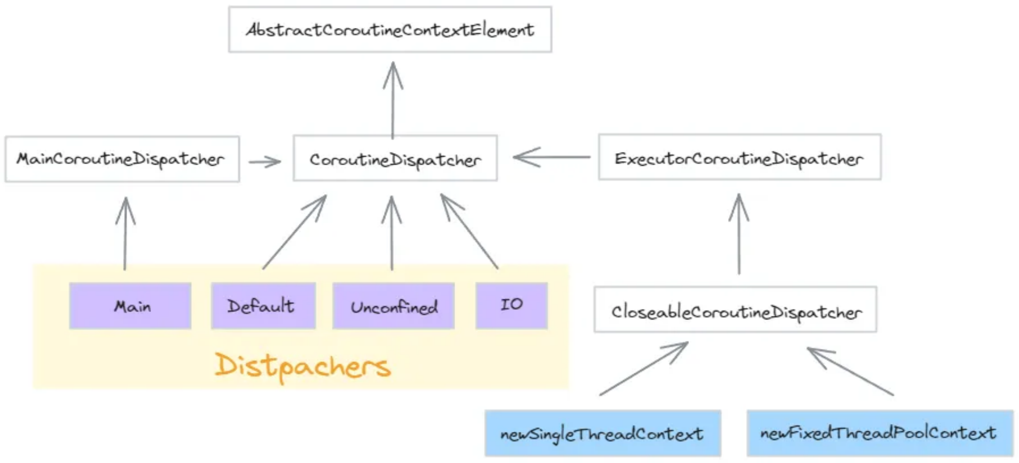
It is an object that defines a context or lifecycle for new coroutines created and launched within it. When a `CoroutineScope` is created, its lifecycle starts and only ends when it’s canceled or its associated `Job` or `SupervisorJob` finishes.

## CoroutineContext

It is a set of elements that defines where and how a routine is executed, with its main elements being the Job and its Dispatcher.

## Distpatchers and threads

> *The coroutine context includes a coroutine dispatcher (see CoroutineDispatcher) that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.*
>
> *All coroutine builders like launch and async accept an optional CoroutineContext parameter that can be used to explicitly specify the dispatcher for the new coroutine and other context elements.*
>
> *— Coroutine context and Dispatcher —*


AbstractCoroutineContextElement
MainCoroutineDispatcher
CoroutineDispatcher
ExecutorCoroutineDispatcher
Main
Default
Unconfined
IO
Distpachers
CloseableCoroutineDispatcher
newSingleThreadContext
newFixedThreadPoolContext

- **Default**: The default dispatcher is used when no other dispatcher is explicitly specified in the scope. It is represented by Dispatchers.Default and uses a shared background pool of threads.
- **Unconfined**: Allows a coroutine to run on any thread, even on different threads for each resume. It is appropriate for coroutines that do not consume CPU or update shared data confined to a specific thread.
- **IO**: Is a coroutine dispatcher that is designed for blocking I/O operations. This dispatcher uses a shared pool of threads that is created on demand. This one is suitable for I/O operations that can block the execution thread, such as reading or writing files, making database queries, or making network requests.
- **Main**: Manages coroutines specifically on the main thread, ensuring smooth UI interactions and seamless user experiences. It typically operates with a single thread to maintain UI consistency.
- **newSingleThreadContext**: Crafts a coroutine execution environment using a dedicated thread with built-in yield support. It’s a delicate API that allocates native resources (the thread itself), requiring careful management.

- newFixedThreadPoolContext: Establishes a coroutine execution environment with a fixed-size thread pool, enabling parallel execution of coroutines while managing thread resources carefully.

## Job

In Kotlin Coroutines is an inteface that represents an asynchronous task, this one is a cancellable piece of work. There are two basic ways to create instances of the Job: using the `launch` coroutine builder and `CompletableJob` with the `Job()` factory function.

*With launch*

```
val job = launch { // launch a new coroutine and keep a reference to its Job
    delay(1000L)
    println("Hello World!")
}
job.join() // wait until child coroutine completes
println("Done")
```

*With CompletableJob*

```
val completableJob = Job()

// ...

completableJob.complete()
```

## SupervisorJob

It is an implementation of Job that acts as a supervisor for child coroutines. If a coroutine launched within a SupervisorJob scope fails, only that child coroutine will be canceled, and it will not affect other child coroutines or the parent scope.

```
fun main() = runBlocking {
    val supervisorJob = SupervisorJob()

    val coroutine1 = launch(supervisorJob) {
        println("Coroutine 1")
        throw RuntimeException("Error in Coroutine 1")
    }

    val coroutine2 = launch(supervisorJob) {
        println("Coroutine 2")
        delay(500)
        println("Coroutine 2 completed")
    }

    coroutine1.join()
    coroutine2.join()

    println("Parent coroutine: ${supervisorJob.isActive}") // Output: Parent coroutine: tru
}
```

SupervisorJob example

## suspendCoroutine

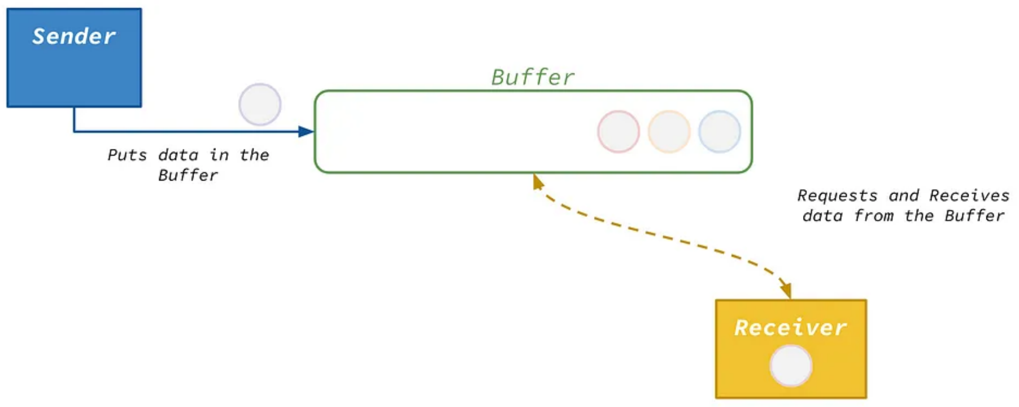
It is a function that suspends the execution of a coroutine. It is used to

```
fun main() = runBlocking {
    println("Response = ${fetchData()}")
}

suspend fun fetchData(): String = suspendCoroutine { continuation ->
    val block: (String) -> Unit = {
        continuation.resume(it)
    }
    block("String from a callback")
}
```

suspendCoroutine example

## Channels



Channels facilitate asynchronous communication between coroutines, offering a unidirectional conduit for sending and receiving values. They provide explicit control over data transfer and can be closed when no longer needed for efficient resource management.

Accompanying video to "Intro to Coroutines and Channels" hands-on lab: c...

## Flow

Flows are cold streams similar to sequences — the code inside a flow builder does not run until the flow is collected.

```kotlin
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
```

Flow example

## StateFlow

StateFlow in Kotlin Coroutines represents and observes a mutable state, automatically notifying observers of any changes in that state. It is ideal for efficiently managing and observing specific changes in the application state.

```kotlin
fun main() = runBlocking<Unit> {
    val mutableCounter = MutableStateFlow(0)
    val counter: StateFlow<Int> = mutableCounter
```

```
        launch {
            counter.collect {
                println("Current value: $it")
            }
        }

        repeat(3) { i ->
            delay(1000)
            mutableCounter.value = i
        }
    }
```

**Output:**

Current value: 0
Current value: 1
Current value: 2

### SharedFlow

Allows the emission of events to multiple consumers, sharing the emitted elements without maintaining an internal mutable state. It is useful when various components need to be aware of general events, streamlining efficient event communication among different parts of the application.

```
    fun main() = runBlocking<Unit> {
        val sharedFlow = MutableSharedFlow<Int>(10)

        launch {
            sharedFlow.collect {
                println("Current value: $it")
            }
        }

        sharedFlow.emit(1)
        sharedFlow.emit(2)
        sharedFlow.emit(3)
    }
```

**Output:**

Current value: 1
Current value: 2
Current value: 3

## Sources

- Kotlin Coroutines
- Difference between Flows and Channels in Kotlin