

# Coroutine against Goroutine: comparison between *Kotlin* and *GoLang* concurrency

Activity Project in *Operating Systems M*

Luca Marchegiani

August 20, 2022

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview of the concurrency in Kotlin and Go</b>	<b>2</b>
2.1	Kotlin concurrency overview . . . . .	2
2.1.1	Realization of coroutines in Kotlin . . . . .	3
2.1.2	Synchronization between coroutines in Kotlin . . . . .	6
2.1.3	Suspending function . . . . .	7
2.2	Go concurrency overview . . . . .	7

## Abstract

In this paper we are going to make a comparison between `Kotlin` and `Go` concurrency that is the main focus of the activity project in *Operating Systems M* course of the master degree in *Computer Science Engineering* at the *Alma Mater Studiorum* University of Bologna.

After a description of the concurrency management in these two languages, we will try to go into an experimental comparison using a previously made projects of the author in the courses of *Software System Engineering* and *Mobile Systems M*.

## 1 Introduction

`Kotlin` is a modern multiplatform programming language developed by JetBrains that works on JVM such as `Scala`. `Kotlin` is completely interoperable with `Java`<sup>1</sup> and it is *object-oriented* with strong elements of functional programming that make it more powerful than his father `Java`. As specified in the main page of the official website, `Kotlin` has also the advantages to be *concise*, *safe in nullability*, *expressive*, *interoperable* and *multiplatform*.

---

<sup>1</sup>All classes written in `Kotlin` are callable from `Java` code and vice-versa.

In addition to this, from 2019, Google announced that **Kotlin** is the preferred language for developing **Android** application. **Kotlin** supports also multiplatform allowing the developer to write **Kotlin** code that can be compiled for native platform (including **Android** and **iOS**), **JVM** and **JavaScript**.

**Go** is an open source programming language developed and supported by **Google**. It's an *imperative* and *object-oriented* language strongly designed for concurrency thanks to its very easy way to launch process and its efficiency. The idea of this language is to maintain the run-time efficiency of **C** but with more readability and usability. Differently from **C**, **Go** has *memory safety*, *garbage collection* and *structural typing* as said by Wikipedia.

In the last years, **Go** also supports mobile platform (**Android** and **iOS**) as described in the official wiki, by writing *all-Go native mobile applications* or *SDK applications* with bindings for **Java** or **Objective-C**. There is also a toolkit called **Fyne** that is free and open source that makes easy to build graphical application also for mobile using **Go** .

From the concurrency point of view, **both of this language supports coroutines** as concurrent units of execution. *Coroutines* are lightweight processes that can run over multiple OS threads, allowing to save on thread management costs.

Coroutines and threads are very similar, but the main difference is that the firsts are *non-preemptive* (or *cooperatives*) differently from the seconds that are typically *preemptive* and scheduled by the OS. Indeed, the execution of a coroutine can be suspended and resumed by the developer, calling some operations, and not by the OS.

We will go in the details for both of this language.

## 2 Overview of the concurrency in Kotlin and Go

As specified in the introduction, **Kotlin** and **Go** exposes concurrency thanks to **coroutines** and other tools that let the developer manage their synchronization.

As we already said, coroutines are lightweight processes for cooperation that executes on OS threads and that can suspend at a certain point and later resumed at the same point but with the possibility to execute on a different thread. The main advantage by using coroutines instead threads is that switching between then does not require any *system call* ensuring lower management costs. This introduces great advantages especially for *asynchronous* computation.

### 2.1 Kotlin concurrency overview

We said that **Kotlin** is based on the **JVM** (but can also compile **JavaScript** or native using **LLVM**) and is interoperable with **Java**. The main implementation of **Kotlin** is done in its compiler: for **Kotlin** on **JVM**, all classes are compiled as normal **Java** classes. This means that **Kotlin** **can access to all threading** packages exposed by **Java** (and this is also valid for **Android**). So, in **Kotlin** **it is possible to use the standard threads** that are provided by **Java**.

Even if there is the possibility to use the standard **Java** threads, as anticipated, **Kotlin** introduces the new **kotlinx.coroutines** library for realizing concurrency by adopting *coroutines*. Coroutines are *instances of suspendable computation* that let the developer to

easily write **asynchronous and non-blocking code** that can run concurrently, without using *callback* or *promises*. The main mechanism in which Kotlin coroutines are based is the **suspending function**: special Kotlin method that can suspend the execution of the current coroutine without blocking the current thread.

### 2.1.1 Realization of coroutines in Kotlin

To go into the details of the coroutine in Kotlin , we have to introduce some basic concepts<sup>2</sup>:

- **Job**:

The object that represents a *background job* of a coroutine. When a coroutine is launched, the `launch` method immediately returns the reference to the `Job` associated to the coroutine. **A job represents the lifecycle of a coroutine** and can be used to *cancel* its execution. Then, it can have six possible states, each coded by a combination of the three properties of the `Job` class: `isActive`, `isCompleted` and `isCancelled`. This table summarizes the possible states of a `Job` and the value of the three properties for each state:

State	Type	isActive	isCompleted	isCancelled
<i>New</i>	initial	false	false	false
<i>Active</i>	initial	true	false	false
<i>Completing</i>	transient	true	false	false
<i>Cancelling</i>	transient	false	false	true
<i>Cancelled</i>	final	false	true	true
<i>Completed</i>	final	false	true	false

Table 1: States of a Job

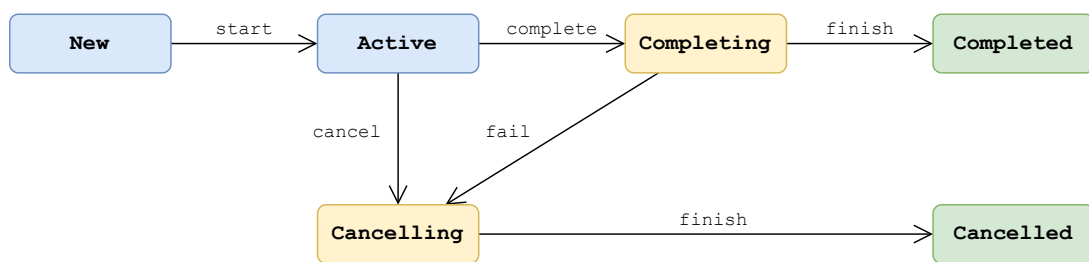


Figure 1: Lifecycle of Kotlin coroutine in Job

The figure 1 represents the entire lifecycle of a `Job`, so it also represents the lifecycle of a Kotlin coroutine.

- **CoroutineDispatcher**:

As we already said, in their lifecycle coroutine can run in different threads. For example, suppose to have a coroutine  $C_1$  that is started on the thread  $T_1$  that executes its code:

1.  $C_1$  starts its execution on thread  $T_1$ ;

---

<sup>2</sup>See [medium.com](https://medium.com) for additional details.

2. during its execution,  $C_1$  encounter an instruction  $I_1$  that suspends it waiting for something;
3.  $C_1$  is suspended by  $I_1$  and another coroutine  $C_2$  starts to execute on  $T_1$ ;
4.  $C_2$  is executing on  $T_1$  while  $I_1$  returns resuming  $C_1$  from its suspension, but now  $T_1$  is not available because it is executing the code of  $C_2$ ;
5.  $C_1$  may execute on another available thread  $T_2$  while  $C_2$  continue to run in parallel on  $T_1$  (if the configuration allows it).

**CoroutineDispatcher is the object that *dispatch* the coroutine between the different available threads.** The `CoroutineDispatcher` is important because it determines in which thread a coroutine can run: for example, in Android using `Dispatchers.Main` means that the coroutine will be executed confined to the Main thread<sup>3</sup>.

By default, when a coroutine is created, it is used the `Dispatchers.Default` that uses *worker* threads, a shared pool of threads on JVM in which coroutines can execute in parallel.

- **CoroutineContext:**

Each coroutine in Kotlin has a *context* that is *immutable*. A context is simply a set of *elements* that realizes the concept of *context* in which the coroutine executes. The main elements in a context are:

- the `Job` that represents the coroutine;
- the `CoroutineDispatcher` that dispatches the execution of coroutine over the threads;
- the `CoroutineName` that is the name associated to the coroutine (useful for debugging);
- the `CoroutineExceptionHandler` that is an handler for all the exception thrown during the execution of the coroutine;
- the `ContinuationInterceptor` that allows to define *how* the coroutine should continue after a resume (a sort of *callback* that is invoked on coroutine resume).

Notice that `CoroutineContext` is **immutable**, but it is possible to add elements using the **plus operator** that produces a new context instance. In addition, **all of these elements extends `CoroutineContext`** so, using plus operator let to easily create a context that is a *join* of the other. For example:

```
val newContext = CoroutineName("MyCoroutine") + Dispatchers.Main
```

creates a new context named *MyCoroutine* in which coroutine will be executed using `Dispatchers.Main`.

**A context can be passed to the coroutine builder before launching it or, if the context has to be changed while coroutine is running, it is possible to use `withContext` suspend function.** Kotlin has also a default context for

---

<sup>3</sup>In this case, the coroutine can update the UI. There are also dispatchers for `JavaFX` or `Swing` for Kotlin JVM to force coroutines to be executed on the thread that can update the user interface.

builders that is `EmptyCoroutineContext` which can be also used with plus operator to create new contexts.

- **CoroutineScope**:

Each coroutine in Kotlin must have a *scope* which delimits the lifetime of the coroutine. The `CoroutineScope` consists in only one property: `coroutineContext`, an instance of `CoroutineContext`. In addition to this, the `CoroutineScope` has also some *extension functions* such as `launch` that is a builder for coroutines.

Then, when `launch` is invoked using a `CoroutineScope`, the function launches a new coroutine and its context is *inherited* from those of the scope. In this way, all the elements of the parents and its cancellation are propagated to the child; then, if a scope is cancelled, all the coroutine launched starting from it will be cancelled.

In Kotlin the concept of *coroutine* can be summarized by the formula:

$$\text{Coroutine} = \text{CoroutineContext} + \text{Job}$$

In order to launch a coroutine, the developer has to:

1. create an instance of `CoroutineScope`, for example using the `runBlocking` scope builder;
2. call a coroutine builder starting from the created scope, such as `launch`, that returns the `Job` associated to the coroutine.

Here is the example of the creation of a simple coroutine taken from the official documentation on [kotlinlang.org](http://kotlinlang.org):

```
1 fun main() = runBlocking { // this: CoroutineScope
2   launch { // launch a new coroutine and continue
3     delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
4     println("World!") // print after delay
5   }
6   println("Hello") // main coroutine continues while a previous one is delayed
7 }
```

that produces this result on the console:

```
Hello
World!
```

To fully understand this snippet, the reader should know *higher-order functions* and *receivers* which are concepts that came from *functional programming* available in Kotlin .

Notice that `runBlocking` as also an optional `CoroutineContext` argument that can be used to pass elements that will be added to the context of the scope. All of these elements are inherited by child except for the `Job` that is created by the coroutine builder.

For example:

```
1 runBlocking(CoroutineName("MyCoroutine")) {
2   val parentScope = this
3   println("parent : $coroutineContext")
4   val job1 = launch {
5     println("launch1 : $coroutineContext," +
6       " childScope == parentScope : ${this == parentScope}")
7 }
```

```

7  }
8  val job2 = launch {
9      println("launch2 : $coroutineContext, " +
10         "childScope == parentScope : ${this == parentScope}")
11  }
12  joinAll(job1, job2)
13 }

```

produces an output similar to:

```

parent  : [CoroutineName(MyCoroutine),
BlockingCoroutine{Active}@68f7aae2, BlockingEventLoop@4f47d241]
launch1 : [CoroutineName(MyCoroutine),
StandaloneCoroutine{Active}@d70c109, BlockingEventLoop@4f47d241],
childScope == parentScope : false
launch2 : [CoroutineName(MyCoroutine),
StandaloneCoroutine{Active}@1bc6a36e, BlockingEventLoop@4f47d241],
childScope == parentScope : false

```

As you can see, both child scopes are different from parent even if they are in relationship: cancelling parent scope cancels those of the children, but the reverse is not true. About the context, it's clear that child contexts are completely inherited from the parent except for the `Job` instances<sup>4</sup> that are different.

### 2.1.2 Synchronization between coroutines in Kotlin

We highlight that **coroutines in Kotlin can use shared memory or *messages*** to synchronize themselves. In particular:

- The package `kotlinx.coroutines.sync` exposes classical tools for synchronization in a shared memory environment (*mutex* and *semaphore*).

Notice that this type of synchronization is very basic if compared with the standard Java tools for concurrency such as `Lock` and `Condition`; at this moment, Kotlin does not define any mechanism similar to Java condition, but however it's simple to implement it (for example, we have an implementation made by the author called `CoroutineCondition` that uses the `Continuation` object of a coroutine).

[See `MutexPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.channels` exposes classical tools for synchronization by exchanging messages in no shared memory environment (*channels*). The main entity of this package is `Channel`, that is very similar to `BlockingQueue` but with suspending operations instead of the Java blocking methods.

Two coroutines can use a channel in order to transfer a single value that came from the *producer* (the coroutine that invoke the `send` operation) and goes to the *consumer* (the coroutine that invoke the `receive` operation); in Kotlin channels are **bidirectional** and **symmetric** (one-to-one).

**The semantic of *send/receive* operations depends on the nature of the channel** that is determinated by its capacity, but the communication can be **synchronous** or **asynchronous** with also some little variations of these (for example,

---

<sup>4</sup>`BlockingCoroutine` and `StandaloneCoroutine` are `Job` extensions.

**rendez-vous**). Notice that a channel can safely be shared between coroutines, but the developer has to pay attention because the **receive** operation can easily lead to competition problems if invoked parallel from two or more coroutines.

[See `ChannelPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.flow` exposes the tools for using *flows* that are defined by the documentation as *asynchronous cold stream of elements* that can safely be used to synchronize multiple coroutine at the same time.

Flow can be more formally defined as **mono-directional, one-to-many and asynchronous** channels with the possibility to be **buffered** for replay strategies. In the latest versions of Kotlin , flows replaced `BroadcastChannel`.

### 2.1.3 Suspending function

*Suspending function* are normal Kotlin methods but with the feature that they can *suspend* the execution of a coroutine. The main example of suspending function it's `delay` that suspend the execution of coroutine which calls it for a specified time.

Look at this code:

```
1 suspend fun sleep(timeMillis : Long) {
2     println("I'm going to sleep for $timeMillis milliseconds...")
3     delay(timeMillis)
4     println("Good morning, I wake up!")
5 }
6
7 fun main() {
8     runBlocking(CoroutineName("MyCoroutine")) {
9         val job = launch {
10             println("Hello... I'm a coroutine")
11             sleep(5000)
12             println("I'm out of my bed. Goodbye")
13         }
14         job.join()
15     }
16 }
```

it produces:

```
Hello... I'm a coroutine
I'm going to sleep for 5000 milliseconds...
Good morning, I wake up!
I'm out of my bed. Goodbye
```

## 2.2 Go concurrency overview

As said in the official page, Go has a *built-in concurrency and a robust standard library* which is one of the central features of the language.