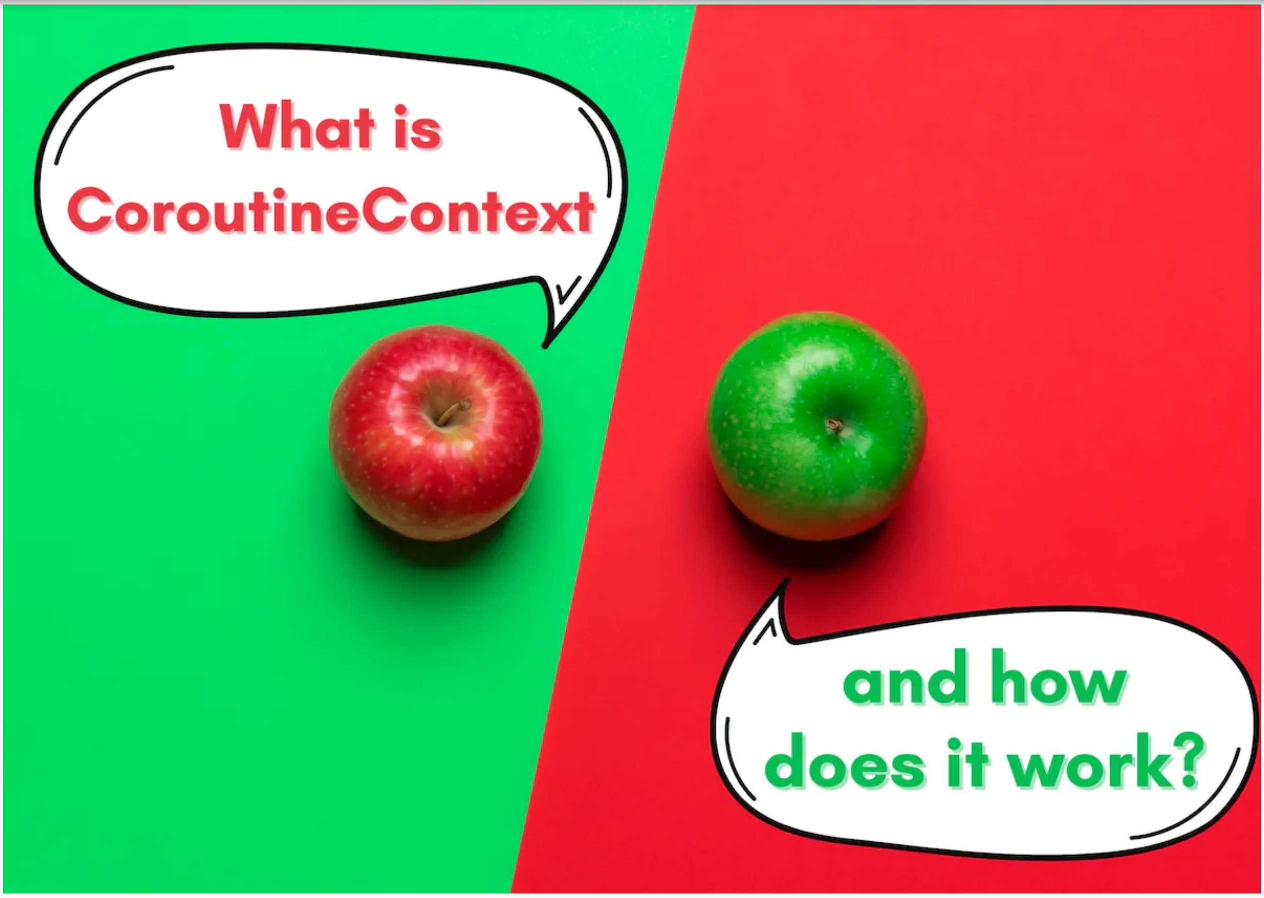
Kt. Academy



What is CoroutineContext and how does it work?

> This is a chapter from the book Kotlin Coroutines. You can find it on LeanPub or Amazon.

If you take a look at the coroutine builders' definitions, you will see that their first parameter is of type `CoroutineContext`.

```
public fun CoroutineScope.launch(
   context: CoroutineContext = EmptyCoroutineContext,
   start: CoroutineStart = CoroutineStart.DEFAULT,
   block: suspend CoroutineScope.() -> Unit
): Job {
   ...
}
```

The receiver and the last argument's receiver are of type `CoroutineScope`[1]. This `CoroutineScope` seems to be an important concept, so let's check out its definition:

```
public interface CoroutineScope {
    public val coroutineContext: CoroutineContext
```

PREVIOUS

NEXT

```
}
```

It seems to be just a wrapper around `CoroutineContext`. So, you might want to recall how `Continuation` is defined.

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resumeWith(result: Result<T>)
}
```

`Continuation` contains `CoroutineContext` as well. This type is used by the most important Kotlin coroutine elements. This must be a really important concept, so what is it?

`CoroutineContext` interface

`CoroutineContext` is an interface that represents an element or a collection of elements. It is conceptually similar to a map or a set collection: it is an indexed set of `Element` instances like `Job`, `CoroutineName`, `CouroutineDispatcher`, etc. The unusual thing is that each `Element` is also a `CoroutineContext`. So, every element in a collection is a collection in itself.

This concept is quite intuitive. Imagine a mug. It is a single element, but it is also a collection that contains a single element. When you add another mug, you have a collection with two elements.

In order to allow convenient context specification and modification, each `CoroutineContext` element is a `CoroutineContext` itself, as in the example below (adding contexts and setting a coroutine builder context will be explained later). Just specifying or adding contexts is much easier than creating an explicit set.

```
launch(CoroutineName("Name1")) { ... }
launch(CoroutineName("Name2") + Job()) { ... }
```

Every element in this set has a unique `Key` that is used to identify it. These keys are compared by reference.

For example `CoroutineName` or `Job` implement `CoroutineContext.Element`, which implements the `CoroutineContext` interface.

```
fun main() {
    val name: CoroutineName = CoroutineName("A name")
    val element: CoroutineContext.Element = name
    val context: CoroutineContext = element

    val job: Job = Job()
    val jobElement: CoroutineContext.Element = job
    val jobContext: CoroutineContext = jobElement
}
```

Open in Playground → Target: JVM Running on v.1.9.20

It’s the same with `SupervisorJob`, `CoroutineExceptionHandler` and dispatchers from the `Dispatchers` object. These are the most important coroutine contexts. They will be explained in the next chapters.

Finding elements in CoroutineContext

Since `CoroutineContext` is like a collection, we can find an element with a concrete key using `get`. Another option is to use square brackets, because in Kotlin the `get` method is an operator and can be invoked using square brackets instead of an explicit function call. Just like in `Map`: when an element is in the context, it will be returned. If it is not, `null` will be returned instead.

```
fun main() {
    val ctx: CoroutineContext = CoroutineName("A name")

    val coroutineName: CoroutineName? = ctx[CoroutineName]
    // or ctx.get(CoroutineName)
    println(coroutineName?.name) // A name
    val job: Job? = ctx[Job] // or ctx.get(Job)
    println(job) // null
}
```


> `CoroutineContext` is part of the built-in support for Kotlin coroutines, so it is imported from `kotlin.coroutines`, while contexts like `Job` or `CoroutineName` are part of the kotlinx.coroutines library, so they need to be imported from `kotlinx.coroutines`.

To find a `CoroutineName`, we use just `CoroutineName`. This is not a type or a class: it is a companion object. It is a feature of Kotlin that a name of a class used by itself acts as a reference to its companion object, so `ctx[CoroutineName]` is just a shortcut to `ctx[CoroutineName.Key]`.


```
data class CoroutineName(
    val name: String
) : AbstractCoroutineContextElement(CoroutineName) {

    override fun toString(): String = "CoroutineName($name)"

    companion object Key : CoroutineContext.Key<CoroutineName>
}
```

It is common practice in the kotlinx.coroutines library to use companion objects as keys to elements with the same name. This makes it easier to remember[2]. A key might point to a class (like `CoroutineName`) or to an interface (like `Job`) that is implemented by many classes with the same key (like `Job` and `SupervisorJob`).

```
interface Job : CoroutineContext.Element {
    companion object Key : CoroutineContext.Key<Job>

    // ...
}
```

Adding contexts

What makes `CoroutineContext` truly useful is the ability to merge two of them together.

When two elements with different keys are added, the resulting context responds to both keys.

```
fun main() {
    val ctx1: CoroutineContext = CoroutineName("Name1")
    println(ctx1[CoroutineName]?.name) // Name1
    println(ctx1[Job]?.isActive) // null

    val ctx2: CoroutineContext = Job()
    println(ctx2[CoroutineName]?.name) // null
    println(ctx2[Job]?.isActive) // true, because "Active"
    // is the default state of a job created this way

    val ctx3 = ctx1 + ctx2
    println(ctx3[CoroutineName]?.name) // Name1
    println(ctx3[Job]?.isActive) // true
}
```

Open in Playground → Target: JVM Running on v.1.9.20

When another element with the same key is added, just like in a map, the new element replaces the previous one.

```
fun main() {
    val ctx1: CoroutineContext = CoroutineName("Name1")
    println(ctx1[CoroutineName]?.name) // Name1

    val ctx2: CoroutineContext = CoroutineName("Name2")
    println(ctx2[CoroutineName]?.name) // Name2

    val ctx3 = ctx1 + ctx2
    println(ctx3[CoroutineName]?.name) // Name2
}
```

Open in Playground → Target: JVM Running on v.1.9.20

Empty coroutine context

Since `CoroutineContext` is like a collection, we also have an empty context. Such a context by itself returns no elements; if we add it to another context, it behaves exactly like this other context.

```
fun main() {
    val empty: CoroutineContext = EmptyCoroutineContext
    println(empty[CoroutineName]) // null
    println(empty[Job]) // null

    val ctxName = empty + CoroutineName("Name1") + empty
    println(ctxName[CoroutineName]) // CoroutineName(Name1)
```

```
}
```

Open in Playground → Target: JVM Running on v.1.9.20

Subtracting elements

Elements can also be removed from a context by their key using the `minusKey` function.

> The `minus` operator is not overloaded for `CoroutineContext`. I believe this is because its meaning would not be clear enough, as explained in Effective Kotlin Item 12: An operator's meaning should be consistent with its function name.

```kotlin
fun main() {
    val ctx = CoroutineName("Name1") + Job()
    println(ctx[CoroutineName]?.name) // Name1
    println(ctx[Job]?.isActive) // true

    val ctx2 = ctx.minusKey(CoroutineName)
    println(ctx2[CoroutineName]?.name) // null
    println(ctx2[Job]?.isActive) // true

    val ctx3 = (ctx + CoroutineName("Name2"))
        .minusKey(CoroutineName)
    println(ctx3[CoroutineName]?.name) // null
    println(ctx3[Job]?.isActive) // true
}
```

Open in Playground → Target: JVM Running on v.1.9.20

Folding context

If we need to do something for each element in a context, we can use the `fold` method, which is similar to `fold` for other collections. It takes:

- an initial accumulator value;
- an operation to produce the next state of the accumulator, based on the current state, and the element it is currently invoked in.

```kotlin
fun main() {
    val ctx = CoroutineName("Name1") + Job()

    ctx.fold("") { acc, element -> "$acc$element " }
        .also(::println)
    // CoroutineName(Name1) JobImpl{Active}@dbab622e

    val empty = emptyList<CoroutineContext>()
    ctx.fold(empty) { acc, element -> acc + element }
        .joinToString()
        .also(::println)
    // CoroutineName(Name1), JobImpl{Active}@dbab622e
}
```

Open in Playground → Target: JVM Running on v.1.9.20

Coroutine context and builders

So `CoroutineContext` is just a way to hold and pass data. By default, the parent passes its context to the child, which is one of the parent-child relationship effects. We say that the child inherits context from its parent.

```
fun CoroutineScope.log(msg: String) {
    val name = coroutineContext[CoroutineName]?.name
    println("[$name] $msg")
}

fun main() = runBlocking(CoroutineName("main")) {
    log("Started") // [main] Started
    val v1 = async {
        delay(500)
        log("Running async") // [main] Running async
        42
    }
    launch {
        delay(1000)
        log("Running launch") // [main] Running launch
    }
    log("The answer is ${v1.await()}")
    // [main] The answer is 42
}
```

Each child might have a specific context defined in the argument. This context overrides the one from the parent.

```
fun main() = runBlocking(CoroutineName("main")) {
    log("Started") // [main] Started
    val v1 = async(CoroutineName("c1")) {
        delay(500)
        log("Running async") // [c1] Running async
        42
    }
    launch(CoroutineName("c2")) {
        delay(1000)
        log("Running launch") // [c2] Running launch
    }
    log("The answer is ${v1.await()}")
    // [main] The answer is 42
}
```

A simplified formula to calculate a coroutine context is:

```
defaultContext + parentContext + childContext
```

Since new elements always replace old ones with the same key, the child context always overrides elements with the same key from the parent context. The defaults are used only for keys that are not specified anywhere else. Currently, the defaults only set `Dispatchers.Default` when no `ContinuationInterceptor` is set, and they only set `CoroutineId` when the application is in debug mode.

There is a special context called `Job`, which is mutable and is used to communicate between a coroutine's child and its parent. The next chapters will be dedicated to the effects of this communication.

Accessing context in a suspending function

`CoroutineScope` has a `coroutineContext` property that can be used to access the context. But what if we are in a regular suspending function? As you might remember from the Coroutines under the hood chapter, context is referenced by continuations, which are passed to each suspending function. So, it is possible to access a parent's context in a suspending function. To do this, we use the `coroutineContext` property, which is available in every suspending scope.

```kotlin
import kotlinx.coroutines.*
import kotlin.coroutines.coroutineContext

suspend fun printName() {
    println(coroutineContext[CoroutineName]?.name)
}

suspend fun main() = withContext(CoroutineName("Outer")) {
    printName() // Outer
    launch(CoroutineName("Inner")) {
        printName() // Inner
    }
    delay(10)
    printName() // Outer
}
```

Creating our own context

It is not a common need, but we can create our own coroutine context pretty easily. To do this, the easiest way is to create a class that implements the `CoroutineContext.Element` interface. Such a class needs a property `key` of type `CoroutineContext.Key<*>`. This key will be used as the key that identifies this context. The common practice is to use this class's companion object as a key. This is how a very simple coroutine context can be implemented:


```kotlin
class MyCustomContext : CoroutineContext.Element {

    override val key: CoroutineContext.Key<*> = Key

    companion object Key :
        CoroutineContext.Key<MyCustomContext>
}
```

Such a context will behave a lot like `CoroutineName`: it will propagate from parent to child, but any children will be able to override it with a different context with the same key. To see this in practice, below you can see an example context that is designed to print consecutive numbers.

```
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import kotlin.coroutines.CoroutineContext
import kotlin.coroutines.coroutineContext

class CounterContext(
    private val name: String
) : CoroutineContext.Element {
    override val key: CoroutineContext.Key<*> = Key
    private var nextNumber = 0

    fun printNext() {
        println("$name: $nextNumber")
        nextNumber++
    }

    companion object Key :CoroutineContext.Key<CounterContext>
}

suspend fun printNext() {
    coroutineContext[CounterContext]?.printNext()
}

suspend fun main(): Unit =
    withContext(CounterContext("Outer")) {
        printNext() // Outer: 0
        launch {
            printNext() // Outer: 1
            launch {
                printNext() // Outer: 2
            }
            launch(CounterContext("Inner")) {
                printNext() // Inner: 0
                printNext() // Inner: 1
                launch {
                    printNext() // Inner: 2
                }
            }
        }
        printNext() // Outer: 3
    }
```

I have seen custom contexts in use as a kind of dependency injection - to easily inject different values in production than in tests. However, I don't think this will become standard practice.

```
import kotlinx.coroutines.withContext
import java.util.*
import kotlin.coroutines.CoroutineContext
import kotlin.coroutines.coroutineContext
import kotlin.test.assertEquals

data class User(val id: String, val name: String)

abstract class UuidProviderContext :
    CoroutineContext.Element {

    abstract fun nextUuid(): String

    override val key: CoroutineContext.Key<*> = Key

    companion object Key :
        CoroutineContext.Key<UuidProviderContext>
}

class RealUuidProviderContext : UuidProviderContext() {
    override fun nextUuid(): String =
        UUID.randomUUID().toString()
}

class FakeUuidProviderContext(
    private val fakeUuid: String
) : UuidProviderContext() {
    override fun nextUuid(): String = fakeUuid
```

```
}

suspend fun nextUuid(): String =
    checkNotNull(coroutineContext[UuidProviderContext]) {
        "UuidProviderContext not present"
    }
        .nextUuid()

// function under test
suspend fun makeUser(name: String) = User(
    id = nextUuid(),
    name = name
)

suspend fun main(): Unit {
    // production case
    withContext(RealUuidProviderContext()) {
        println(makeUser("Michał"))
        // e.g. User(id=d260482a-..., name=Michał)
    }

    // test case
    withContext(FakeUuidProviderContext("FAKE_UUID")) {
        val user = makeUser("Michał")
        println(user) // User(id=FAKE_UUID, name=Michał)
        assertEquals(User("FAKE_UUID", "Michał"), user)
    }
}
```

Summary

`CoroutineContext` is conceptually similar to a map or a set collection. It is an indexed set of `Element` instances, where each `Element` is also a `CoroutineContext`. Every element in it has a unique `Key` that is used to identify it. This way, `CoroutineContext` is just a universal way to group and pass objects to coroutines. These objects are kept by the coroutines and can determine how these coroutines should be running (what their state is, in which thread, etc). In the next chapters, we will discuss the most essential coroutine contexts in the Kotlin coroutines library.

1: Let's clear up the nomenclature. `launch` is an extension function on `CoroutineScope`, so `CoroutineScope` is its receiver type. The extension function's receiver is the object we reference with `this`.

2: The companion object below is named `Key`. We can name companion objects, but this changes little in terms of how they are used. The default companion object name is `Companion`, so this name is used when we need to reference this object using reflection or when we define an extension function on it. Here we use `Key` instead.

The author:

Marcin Moskała

Marcin Moskala is an experienced developer and Kotlin trainer. He is the founder of the Kt. Academy, an official JetBrains partner for Kotlin

training, author of the books Effective Kotlin, Kotlin Coroutines, Functional Kotlin and Android Development with Kotlin. He is also the main author of the biggest medium publication about Kotlin and a speaker invited to many programming conferences.

Reviewers:

Dean Djermanović

Dean Djermanović is an experienced Android developer from Croatia. He's currently employed at Five, which was officially a Google Developers Certified Agency in 2018. He worked on a variety of apps, from small apps for local startups to big popular apps with 10+ million users. He also worked on a couple of Android-related books and tutorials, either as an author, tech editor, or final pass editor. Occasionally, he participates at local tech meetups as a speaker.

Dean is very passionate about Android, technology in general and he's always trying to learn something new and improve himself as a developer.

He spends his free time working out at the gym, reading books, watching movies, or cycling.

Nicola Corti

Nicola Corti is a Google Developer Expert for Kotlin. He has been working with the language since before version 1.0 and he is the maintainer of several open-source libraries and tools.

He's currently working as Android Infrastructure Engineer at Spotify in Stockholm, Sweden.

Furthermore, he is an active member of the developer community. His involvement goes from speaking at international conferences about Mobile development to leading communities across Europe (GDG Pisa, KUG Hamburg, GDG Sthlm Android).

In his free time, he also loves baking, photography, and running.

Richard Schielek

Garima Jain

Garima Jain, Google Developer Expert - Android, also known as @ragdroid works as a Principal Android Engineer at GoDaddy Studio. She is an international speaker and an active technical blogger. She enjoys interacting with other people from the community and sharing her thoughts with them. In her leisure time, she loves watching television shows, playing TT, and basketball. Due to her love for fiction and coding, she loves to mix technology with fiction to present her ideas and experiments with others.

Jana Jarolimova

Add a comment

Write here...

Submit

Comments

Paul Klein 2023-11-22T08:57:44.975Z

Your prints in the custom CoroutineContext are not right if you consider the launch order of the coroutines. Here is an extended UnitTest Version of your code which shows the problem of shared mutable state:

class CustomCoroutineContext {

```
class CounterContext(
    private val name: String
) : CoroutineContext.Element {

    override val key: CoroutineContext.Key<*> = Key

    private var nextNumber = 0

    fun getNumberName(): String {
        return "$name: ${nextNumber++}"
    }

    companion object Key :CoroutineContext.Key<CounterContex
}

suspend fun getNumberName(): String? {
    return coroutineContext[CounterContext]?.getNumberName()
}

@Test
fun test() = runBlocking {
    withContext(CounterContext("Outer")) {
        assertEquals(getNumberName(), "Outer: 0")
        launch {
            assertEquals(getNumberName(), "Outer: 2")
            launch {
                assertEquals(getNumberName(), "Outer: 3")
            }
            launch(CounterContext("Inner")) {
                assertEquals(getNumberName(), "Inner: 0")
                assertEquals(getNumberName(), "Inner: 1")
```

```
                        launch {
                            assertEquals(getNumberName(), "Inner: 3"
                        }
                        assertEquals(getNumberName(), "Inner: 2")
                    }
                }
                assertEquals(getNumberName(), "Outer: 1")
            }
        }
        }
```