# Kotlin language specification

Version 1.9-rfc+0.1

Marat Akhin Mikhail Belyaev

# Chapter 18

# Asynchronous programming with coroutines

## 18.1 Suspending functions

Most functions in Kotlin may be marked *suspending* using the special `suspend` modifier. There are almost no additional restrictions: regular functions, extension functions, top-level functions, local functions, lambda literals, all these may be suspending functions.

> Note: the following functions and function values cannot be marked as suspending.
>
> - anonymous function declarations;
> - constructors;
> - property getter/setters;
> - delegation-related operator functions.
>
> Note: platform-specific implementations may extend the restrictions on which kinds of functions may be suspending.

Suspending functions have a suspending function type, also marked using the `suspend` modifier.

A suspending function is different from non-suspending functions by potentially having zero or more *suspension points* — statements in its body which may pause the function execution to be resumed at a later moment in time. The main source of suspension points are calls to other suspending functions which represent possible suspension points.

> Note: suspension points are important because at these points another function may start in the same flow of execution, leading to potential

changes in the shared state.

Non-suspending functions may not call suspending functions directly, as they do not support suspension points. Suspending functions may call non-suspending functions without any limitations; such calls do not create suspension points. This restriction is also known as "function colouring".

> Important: an exception to this rule are non-suspending inlined lambda parameters: if the higher-order function invoking such a lambda is called from a suspending function, this lambda is allowed to also have suspension points and call other suspending functions.
>
> Note: suspending functions interleaving each other in this manner are not dissimilar to how functions from different threads interact on platforms with multi-threading support. There are, however, several key differences. First, suspending functions may pause only at suspension points, i.e., they cannot be paused at an arbitrary execution point. Second, this interleaving may happen on a single platform thread.
>
> In a multi-threaded environment suspending functions may also be interleaved by the platform-dependent concurrent execution, independent of the interleaving of coroutines.

The implementation of suspending functions is platform-dependent. Please refer to the platform documentation for details.

## 18.2 Coroutines

A *coroutine* is a concept similar to a thread in traditional concurrent programming, but based on *cooperative multitasking*, e.g., the switching between different execution contexts is done by the coroutines themselves rather than the operating system or a virtual machine.

In Kotlin, coroutines are used to implement suspending functions and can switch contexts only at suspension points.

A call to a suspending function creates and starts a coroutine. As one can call a suspending function only from another suspending function, we need a way to bootstrap this process from a non-suspending context.

> Note: this is required as most platforms are unaware of coroutines or suspending functions, and do not provide a suspending entry point. However, a Kotlin compiler may elect to provide a suspending entry point on a specific platform.

One of the ways of starting suspending function from a non-suspending context is via a *coroutine builder*: a non-suspending function which takes a suspending

function type argument (e.g., a suspending lambda literal) and handles the coroutine lifecycle.

The implementation of coroutines is platform-dependent. Please refer to the platform documentation for details.

## 18.3 Implementation details

Despite being platform-dependent, there are several aspects of coroutine implementation in Kotlin, which are common across all platforms and belong to the Kotlin/Core. We describe these details below.

### 18.3.1 `kotlin.coroutines.Continuation<T>`

Interface `kotlin.coroutines.Continuation<T>` is the main supertype of all coroutines and represents the basis upon which the coroutine machinery is implemented.

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

Every suspending function is associated with a generated `Continuation` subtype, which handles the suspension implementation; the function itself is adapted to accept an additional continuation parameter to support the Continuation Passing Style. The return type of the suspending function becomes the type parameter `T` of the continuation.

`CoroutineContext` represents the context of the continuation and is an indexed set from `CoroutineContext.Key` to `CoroutineContext.Element` (e.g., a special kind of map). It is used to store coroutine-local information, and takes important part in Continuation interception.

`resumeWith` function is used to propagate the results in between suspension points: it is called with the result (or exception) of the last suspension point and resumes the coroutine execution.

To avoid the need to explicitly create the `Result<T>` when calling `resumeWith`, the coroutine implementation provides the following extension functions.

```
fun <T> Continuation<T>.resume(value: T)
fun <T> Continuation<T>.resumeWithException(exception: Throwable)
```

### 18.3.2 Continuation Passing Style

Each suspendable function goes through a transformation from normally invoked function to continuation passing style (CPS). For a suspendable function with parameters $p_1, p_2, \ldots, p_N$ and result type $T$ a new function is generated, with an

additional parameter $p_{N+1}$ of type `kotlin.coroutines.Continuation<T>` and return type changed to `kotlin.Any?`. The calling convention for such function is different from regular functions as a suspendable function may either *suspend* or *return*.

- If the function returns a result, it is returned directly from the function as normal;
- If the function suspends, it returns a special marker value `COROUTINE_SUSPENDED` to signal its suspended state.

The calling convention is maintained by the compiler during the CPS transformation, which prevents the user from manually returning `COROUTINE_SUSPENDED`. If the user wants to suspend a coroutine, they need to perform the following steps.

- Access the coroutine's continuation object by calling `suspendCoroutineUninterceptedOrReturn` intrinsic or any of its wrappers;
- Store the continuation object to resume it later;
- Pass the `COROUTINE_SUSPENDED` marker to the intrinsic, which is then returned from the function.

As Kotlin does not currently support denotable union types, the return type is changed to `kotlin.Any?`, so it can hold both the original return type $T$ and `COROUTINE_SUSPENDED`.

### 18.3.3 Coroutine state machine

Kotlin implements suspendable functions as state machines, since such implementation does not require specific runtime support. This dictates the explicit `suspend` marking (function colouring) of Kotlin coroutines: the compiler has to know which function can potentially suspend, to turn it into a state machine.

Each suspendable lambda is compiled to a continuation class, with fields representing its local variables, and an integer field for current state in the state machine. Suspension point is where such lambda can suspend: either a suspending function call or `suspendCoroutineUninterceptedOrReturn` intrinsic call. For a lambda with $N$ suspension points and $M$ return statements, which are not suspension points themselves, $N + M$ states are generated (one for each suspension point plus one for each non-suspending return statement).

Example:

```
// Lambda body with multiple suspension points
val a = a()
val y = foo(a).await() // suspension point #1
b()
val z = bar(a, y).await() // suspension point #2
c(z)
```

```
// State machine code for the lambda after CPS transformation
//     (written in pseudo-Kotlin with gotos)
class <anonymous> private constructor(
    completion: Continuation<Any?>
): SuspendLambda<...>(completion) {
    // The current state of the state machine
    var label = 0

    // local variables of the coroutine
    var a: A? = null
    var y: Y? = null

    fun invokeSuspend(result: Any?): Any? {
        // state jump table
        if (label == 0) goto L0
        if (label == 1) goto L1
        if (label == 2) goto L2
        else throw IllegalStateException()

      L0:
        // result is expected to be `null` at this invocation

        a = a()
        label = 1
        // 'this' is passed as a continuation
        result = foo(a).await(this)
        // return if await had suspended execution
        if (result == COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED
      L1:
        // error handling
        result.throwOnFailure()
        // external code has resumed this coroutine
        // passing the result of .await()
        y = (Y) result
        b()
        label = 2
        // 'this' is passed as a continuation
        result = bar(a, y).await(this)
        // return if await had suspended execution
        if (result == COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED
      L2:
        // error handling
        result.throwOnFailure()
        // external code has resumed this coroutine
```

```
            // passing the result of .await()
            Z z = (Z) result
            c(z)
            label = -1 // No more steps are allowed
            return Unit
        }

        fun create(completion: Continuation<Any?>): Continuation<Any?> {
          <anonymous>(completion)
        }

        fun invoke(completion: Continuation<Any?>): Any? {
            create(completion).invokeSuspend(Unit)
        }
    }
```

### 18.3.4 Continuation interception

Asynchronous computations in many cases need to control how they are executed, with varying degrees of precision. For example, in typical user interface (UI) applications, updates to the interface should be executed on a special UI thread; in server-side applications, long-running computations are often offloaded to a separate thread pool, etc.

Continuation interceptors allow us to intercept the coroutine execution between suspension points and perform some operations on it, usually wrapping the coroutine continuation in another continuation. This is done using the `kotlin.coroutines.ContinuationInterceptor` interface.

```
interface ContinuationInterceptor : CoroutineContext.Element {
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>
    fun <T> interceptContinuation(continuation: Continuation<T>): Continuation<T>
    fun releaseInterceptedContinuation(continuation: Continuation<*>)
}
```

As seen from the declaration, `ContinuationInterceptor` is a `CoroutineContext.Element`, and to perform the continuation interception, an instance of `ContinuationInterceptor` should be available in the coroutine context, where it is used similarly to the following line of code.

```
val intercepted = continuation.context[ContinuationInterceptor]?.interceptContinuation
```

When the cached `intercepted` continuation is no longer needed, it is released using `ContinuationInterceptor.releaseInterceptedContinuation(...)`.

> Note: this machinery is performed "behind-the-scenes" by the coroutine framework implementation.

### 18.3.5 Coroutine intrinsics

Accessing the low-level continuations is performed using a limited number of built-in intrinsic functions, which form the complete coroutine API. The rest of asynchronous programming support is provided as a Kotlin library `kotlinx.coroutines`.

The complete built-in API for working with coroutines is shown below (all of these are declared in package `kotlin.coroutines.intrinsics` of the standard library).

```
fun <T> (suspend () -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit>

suspend fun <T>
    suspendCoroutineUninterceptedOrReturn(
        block: (Continuation<T>) -> Any?): T

fun <T> (suspend () -> T).
    startCoroutineUninterceptedOrReturn(
        completion: Continuation<T>): Any?

fun <T> Continuation<T>.intercepted(): Continuation<T>

// Additional functions for types with explicit receiver

fun <R, T> (suspend R.() -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit>

fun <T> (suspend R.() -> T).
    startCoroutineUninterceptedOrReturn(
        completion: Continuation<T>): Any?
```

Function `createCoroutineUnintercepted` is used to create a coroutine corresponding to its extension receiver suspending function, which invokes the passed `completion` continuation upon completion. This function does not start the coroutine, however; to do that, one have to call `Continuation<T>.resumeWith` function on the created continuation object. Suspending function `suspendCoroutineUninterceptedOrReturn` provides access to the current continuation (similarly to how `call/cc` works in Scheme). If its lambda returns the `COROUTINE_SUSPENDED` marker, it also suspends the coroutine. Together with `Continuation<T>.resumeWith` function, which resumes or starts a coroutine, these functions form a complete coroutine API built into the Kotlin compiler.