Kotlin Coroutines Explained with Examples

What is Coroutine?

Coroutines are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for **suspending** and **resuming** execution at certain locations

– *Coroutine - Wikipedia*

Basically, coroutines are computations that can be suspended without blocking a thread.

– Kotlin Reference / Functions and Lambdas / Coroutines

Overview

- Start a coroutine
- Create a coroutine
  - Coroutine Internals
  - How to **Suspend**
  - How to **Resume**
- Concurrency Model
- More about Threading

Start a coroutine

Coroutine Builder

- Use built-in **coroutine builder**
  - **`runBlocking{}`**
  - **`launch{}`**
  - **`async{}`**
  - **`buildSequence{}`**
  - **`produce{}`**
- All these builders accept a *suspending lambda*

Example: runBlocking

```
fun <T> runBlocking(
 context: CoroutineContext = EmptyCoroutineContext,
 block: suspend CoroutineScope.() -> T
): T

runBlocking { // executed in calling thread
 println("current thread: ${Thread.currentThread()}'
 delay(10, TimeUnit.SECONDS)
}
```

Don’t do this in UI thread ’cause it is **blocking**.

Example: launch

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

for (i in 1..100L) {
 launch(CommonPool) { // executed in common thread p
  println("current thread: ${Thread.currentThread()}
  delay(i, TimeUnit.SECONDS)
  println("after delay($i)")
 }
}
```

Pitfall: launch

```
val c = AtomicInteger()
for (i in 1..1_000_000) {
 launch(CommonPool) {
  c.addAndGet(1)
 }
}
println("c = ${c.get()}") // WRONG!
```

Print random value because of getting premature result before the computation is done.

Solution: Just Wait

```
val c = AtomicInteger()
val jobs = (1..1_000_000).map {
    launch(CommonPool) {
     c.addAndGet(1)
    }
}
runBlocking {
 jobs.foreach { it.join() }
}
println("c = ${c.get()}") // CORRECT
```

Example: **async{}**

```
fun <T> async(
 context: CoroutineContext,
 start: CoroutineStart = CoroutineStart.DEFAULT,
 block: suspend CoroutineScope.() -> T
): Deferred<T>

val deferred = (0..1000).map { n ->
 async(CommonPool) {
  1
 }
}
runBlocking {
 val sum = deferred.sumBy { it.await() }
 println("sum = $sum") // CORRECT
}
```

Example: buildSequence{}

```
fun <T> buildSequence(
 builderAction: suspend SequenceBuilder<T>.() -> Uni
): Sequence<T>

val fibonacci = buildSequence {
 yield(1)
 var cur = 1
 var next = 1
 while (true) {
  yield(next)
  val tmp = cur + next
  cur = next
  next = tmp
 }
}
println(fibonacci.take(10).joinToString())
```

Example: produce

```
fun <E> produce(
 context: CoroutineContext,
 capacity: Int = 0,
 block: suspend ProducerScope<E>.() -> Unit
): ProducerJob<E>

fun produceSquares() = produce<Int>(CommonPool) {
    for (x in 1..5) send(x * x)
}
```

Create a Coroutine

Suspend/Resume

- Convert your asynchronous operations into sequential logic by **suspending** and **resuming**.
- Suspending is done by calling:
  - Suspending functions provided by specific coroutine builder, e.g. **`send`** of **`produce`**
  - Common suspending functions provided by standard library, e.g. **`delay`**, **`run`**, **`withTimeout`**
  - Your own suspending functions

Suspending Functions Provided Coroutine Builder

- launch, async, runBlocking: **`CoroutineScope`**
- buildSequence: **`SequenceBuilder`**
- produce: **`ProducerScope`**

Suspending Function

```
suspend fun doSomething(foo: Foo): Bar {
 // ...
 otherSuspendFunc()
 // ...
 anotherSuspendFunc()
}
```

- Can only be called from coroutines and other suspending functions.
- Transformed to a **state machine** where states correspond to suspending calls.

Suspending Function

Why marking a function explicitly as suspending function?

- Tell compiler that this function will suspend so that it should **NOT** be called by normal functions.
- It also indicates the position of *suspension points* so that the compiler can figure out how many **states** this function has.
- Compiler also adds a hidden **`Continuation<T>`** parameter to help this function resume from where it was suspended.

Implement a Suspending Function

1. Start the asynchronous operation
   - There is no magic. The job still has to be done by some thread.
2. Setup the resume
   - Use whatever mechanism to detect the job is done and then resume the coroutine.
3. And finally suspend

2 and 3 can done in a single call to low-level API

Resuming a Coroutine

Resuming is done by **scheduling** calling to **`Continuation<T>`** method in the future inside a suspending function.

```
interface Continuation<in T> {
 val context: CoroutineContext
 fun resume(value: T)
 fun resumeWithException(exception: Throwable)
}
```

How can I get a **`Continuation<T>`**?

Scheduling Resume

- By timer
- By callback (yes!)
- By another thread

Low-level API

Continuation can be got when **suspending** a coroutine:

```
inline suspend fun <T> suspendCoroutine(
 crossline block: (Continuation<T>) -> Unit
): T

inline suspend fun <T> suspendCancellableCoroutine(
 holdCancellability: Boolean = false,
 crossline block: (CancellableContinuation<T>) -> Un
): T
```

Obtains **the current continuation instance** inside suspend functions and **suspends currently running coroutine**.

A Simple Implementation of Delay

```
val executor =
 Executors.newSingleThreadScheduledExecutor {
  Thread(it, "scheduler").apply { isDaemon = true }
 }

suspend fun delay(
 time: Long,
 unit: TimeUnit = TimeUnit.MILLISECONDS
): Unit =
 suspendCoroutine { cont ->
  executor.schedule({ cont.resume(Unit) }, time, uni
 }
```

The Offical Implementation of Delay

```
// simplified to fit in the slide
suspend fun delay(t: Long) {
 if (t <= 0) return // don't delay
 return suspendCancellableCoroutine { cont ->
  // relys on a Delay implementation provided in the
  cont.context.delay.scheduleResumeAfterDelay(time,
 } // suspension happens after running this block
}
```

suspendCancellableCoroutine provides an implementation of **CancellableContinuation** to the block.

Example: **await()** for Retrofit

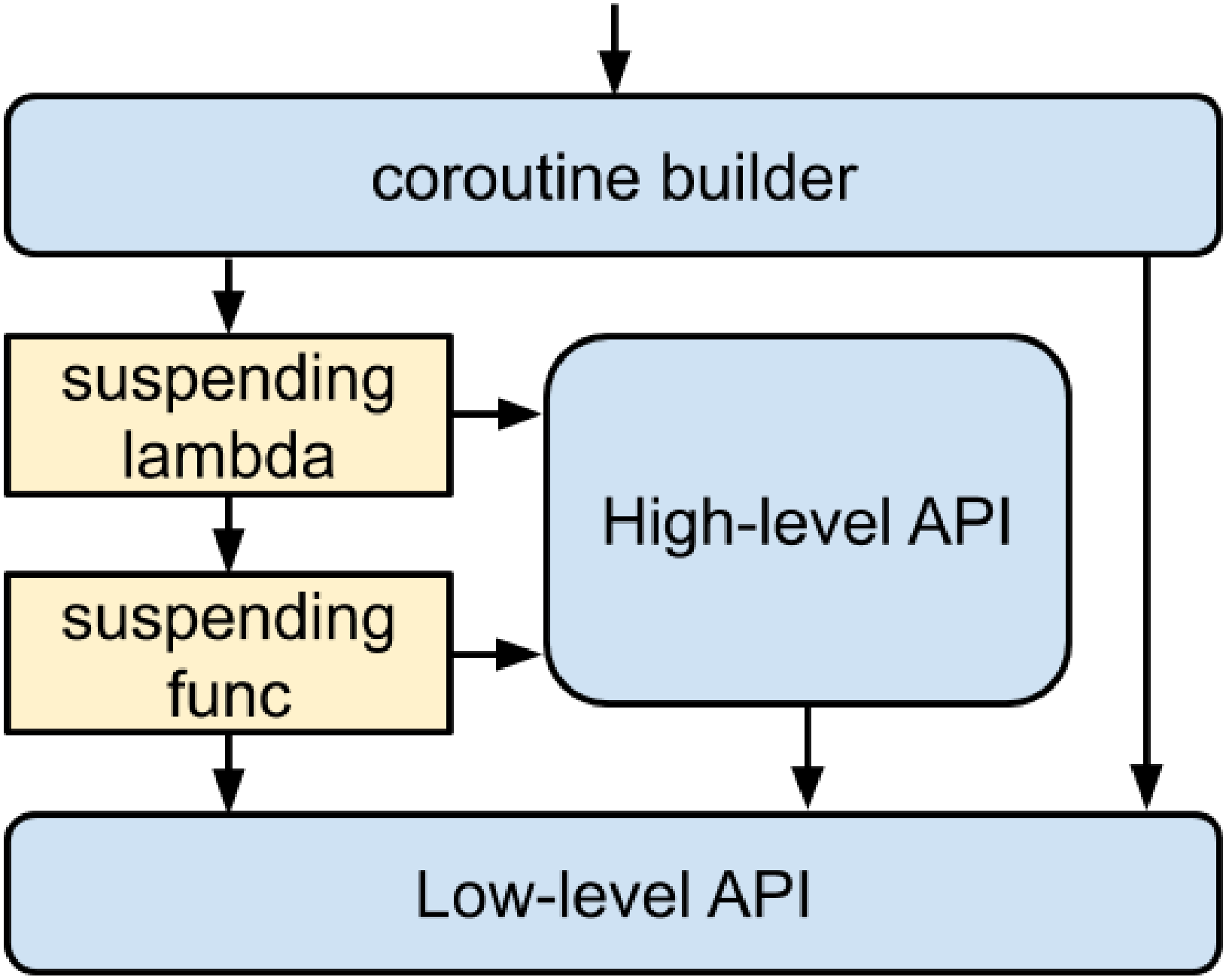
```
suspend fun<T> Call<T>.await(): T =
 suspendCancellableCoroutine { cont ->
  cont.invokeOnCompletion {
   if (continuation.isCancelled) cancel()
  }

  val callback = object: Callback<T> {
   override fun onFailure(c: Call<T>, t: Throwable)
    cont.tryToResume { throw t }
   override fun onResponse(c: Call<T>, r: Response<T
    cont.tryToResume {
     r.isSuccessful || throw IllegalStateException("
     r.body() ?: throw IllegalStateException("...")
    }
  }
  enqueue(callback)
 } // The actual suspension happens after running bl
```

Coroutines Ingredients

- **Language support**
  - Suspending functions
- **High-level APIs**
  - Can be used directly in the user code
- **Low-level APIs**
  - Core API in the Kotlin Standard Library
  - Can be used to create custom concurrency model

Cooking a Coroutine



Suspend/Resume

- Suspend
  - When: by the end of lambda provide to low-level API is called
  - How: by calling low-level API
  - What: depending on which low-evel API is used
- Resume
  - When: depending on how resume is **scheduled** in suspending function
  - How: depending on the **continuation implementation** in **`CoroutineContext`**

High-level Concurrency Model

- async/await
- channels and select
- generators/yield

Custom Concurrency Model

Two ways:

- Use existing coroutine builder and create custom suspending functions
- Create custom **coroutine builder**
  - **Coroutine Context**: Persistent context for the coroutine
  - **Coroutine Scope**: Receiver interface for generic coroutine builders

Low-level API for Creating a Coroutine Builder

```
fun <T> (suspend () -> T).startCoroutine(
  c: Continuation<T>
)

fun <R, T> (suspend R.() -> T).createCoroutine(
 r: R, c: Continuation<T>
): Continuation<Unit>
```

Let’s Talk about Thread

What thread?

- Depending on the context
  - a common pool of shared background threads if **`CommonPool`** is specified
  - **`Unconfined`**
  - Custom ones, e.g. **`Android`** for always running on Android main thread

What will thread do after a coroutine is suspended?

- Execute another coroutine
- Idle until any coroutine is resumed and assigned to the thread

Who will resume the coroutine?

- The thread calling **`Continuation.resume`**.
- Depending on how the task is done asynchronously.

Reference

- Kotlin Reference - Coroutines
- Introduction to Kotlin Coroutines on the JVM
- Coroutines for Kotlin (Revision 3.2)
- Bytecode behind coroutines in Kotlin

Thanks