**What is coroutine**

1. A coroutine is a concurrency design pattern that you can use to simplify & executes asynchronously task.

2. On Android, coroutines help to manage long-running tasks that might otherwise block the main thread

3. Lightweight: You can run many coroutines on a single thread due to support for [suspension](https://kotlinlang.org/docs/reference/coroutines/basics.html), which doesn't block the thread where the coroutine is running.

4. Fewer memory leaks: Use [structured concurrency](https://kotlinlang.org/docs/reference/coroutines/basics.html%23structured-concurrency) to run operations within a scope(GlobalScope, LifecycleScope, ViewModelScope).

5. Built-in cancellation support.

6. Coroutine is light weight thread that run on top on thread. There is two type of coroutine thread builder 1. lunch 2. async

Note: **Concurrency:** Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.

**Asynchonously :** Async is multi-thread, which means operations or programs can run in parallel. Sync is single-thread, so only one operation or program will run at a time

7. lightweight, it means that creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools and smart scheduling for the purpose of which task to execute next and which tasks later

**Coroutine Scope**

1. **globalScope**: When we use GlobalScope.launch we create a top-level coroutine and depend on app label scope. Global scope is used to launch top-level coroutines which are operating on the whole application lifetime and are not cancelled prematurely.

GlobalScope.launch {

//do some work

delay(1000L)

db.update(someProcessedData)

}

2. **lifecycleScope**: A LifecycleScope is defined for each Lifecycle object. Any coroutine launched in this scope is canceled when the Lifecycle is destroyed. You can access the CoroutineScope of the Lifecycle either via lifecycle. lifecycleScope is a scope that is managed by a SupervisorJob whose lifecycle is tied to the Fragment's lifecycle. So just by using lifecycleScope your coroutines will be cancelled when the underlying Lifecycle instance (the Fragment's LifecycleRegistry in this case) is destroyed.

launch(Dispatchers.Main) {

//do some work

delay(1000L)

updateResults()

}

3. **viewModelScope:**  viewModelScope is a CoroutineScope which is tied to your ViewModel . it means that when ViewModel has cleared coroutines inside that scope are canceled too. Dispatchers.IO means **that suspend fun repository**.

viewModelScope.launch(Dispatchers.IO) {

            val jsonBody = "{ username: \"$username\", token: \"$token\"}"

delay(1000L)

            loginRepository.makeLoginRequest(jsonBody)

        }

**What is runBlocking**: Runs a new coroutine and blocks the current thread interruptibly until its completion. This function should not be used from a coroutine. It is designed to bridge regular blocking code to libraries that are written in suspending style, to be used in main functions and in tests. [runBlocking](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/run-blocking.html) is also a coroutine builder that bridges the non-coroutine world of a regular fun main() and the code with coroutines inside of runBlocking { ... } curly braces. The name of runBlocking means that the thread that runs it (in this case — the main thread) gets *blocked* for the duration of the call, until all the coroutines inside runBlocking { ... } complete their execution.

fun main() = runBlocking { // this: CoroutineScope

launch { doWorld() }

println("Hello")

}

// this is your first suspending function

suspend fun doWorld() {

delay(1000L)

println("World!")

}

**What is launch**: [launch](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) is a coroutine builder. It’s  create & starts/run a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch. It launches a new coroutine with current thread and continues work independently.  [launch](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/launch.html) is used to **fire and forget coroutine.**  [join](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/-job/join.html) is used to wait for completion of the launched coroutine and it does not propagate its exception.

val job = launch { // launch new coroutine and keep a reference to its Job

delay(1000L)

println("World!")

}

println("Hello,")

job.join() // wait until child coroutine completes

**What is async:** [async](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) starts a new coroutine and allows you to return a result with a suspend function called await. [async](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/async.html) is used to **start a coroutine that computes some result**. The result is represented by an instance of [Deferred](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/-deferred/index.html) and you **must** use [await](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.experimental/-deferred/await.html) on it

CoroutineScope(Dispatchers.Main).launch {

val retVal1 = async(Dispatchers.IO) { downloadTask1() }

val retVal2 = async(Dispatchers.IO) { downloadTask2() }

Toast.makeText(applicationContext, "All tasks downloaded! ${retVal1.await()}, ${retVal2.await()}", Toast.LENGTH\_LONG).show();

}

**What is Delay:** [delay](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/delay.html) is a special *suspending function*. It *suspends* the coroutine for a specific time. Suspending a coroutine does not *block* the underlying thread, but allows other coroutines to run and use the underlying thread for their code.

**what is Dispatcher & type of Dispatcher**

1. **Dispatchers.Main** - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work

CoroutineScope(Dispatchers.Main).launch {

// update UI

delay(1000L)

println("World!")

}

2. **Dispatchers.IO** - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the [Room component](https://developer.android.com/topic/libraries/architecture/room), reading from or writing to files, and running any network operations.

CoroutineScope(Dispatchers.IO).launch {

// API & Db request or heavy request

delay(1000L)

println("World!")

}

3. **Dispatchers.Default** - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

CoroutineScope(Dispatchers.Default).launch {

// Do some light work like searching & sorting

delay(1000L)

println("World!")

}

**What is suspend:** suspend pauses the execution of the current coroutine, saving all local variables.

**what is Resume:** resume continues execution of a suspended coroutine from the place where it was suspended

**what is JOB/Join :** A launch coroutine builder returns a Job object that is a handle to the launched coroutine and can be used to explicitly wait for its completion. For example, you can wait for completion of the child coroutine and then print "Done" string:

val job = launch { // launch a new coroutine and keep a reference to its Job

delay(1000L)

println("World!")

}

println("Hello")

job.join() // wait until child coroutine completes

println("Done")

// World

// Hello

// Done

**what is Cancel** coroutine :

scope.cancel()

**Job/Deferred:** Conceptually, [async](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) is just like [launch](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html). It starts a separate coroutine which is a light-weight thread that works concurrently with all the other coroutines. The difference is that launch returns a [Job](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-job/index.html) and does not carry any resulting value, while async returns a [Deferred](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-deferred/index.html) — a light-weight non-blocking future that represents a promise to provide a result later. You can use .await() on a deferred value to get its eventual result, but Deferred is also a Job, so you can cancel it if needed.

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")

}

suspend fun doSomethingUsefulOne(): Int {

delay(1000L) // pretend we are doing something useful here

return 13

}

suspend fun doSomethingUsefulTwo(): Int {

delay(1000L) // pretend we are doing something useful here, too

return 29

}

**fun** main() = runBlocking {

**val** job = GlobalScope.launch { // root coroutine with launch

println(**"Throwing exception from launch"**)

**throw** IndexOutOfBoundsException() // Will be printed to the console by Thread.defaultUncaughtExceptionHandler

}

job.join()

println(**"Joined failed job"**)

**val** deferred = GlobalScope.async { // root coroutine with async

println(**"Throwing exception from async"**)

**throw** ArithmeticException() // Nothing is printed, relying on user to call await

}

**try** {

deferred.await()

println(**"Unreached"**)

} **catch** (e: ArithmeticException) {

println(**"Caught ArithmeticException"**)

}

}

Launch vs Async in Kotlin Coroutines

|  |  |
| --- | --- |
| **Launch** | **Async** |
| The launch is fire and forget. | Async is performing a task and return a result. |
| launch{} does not return anything. | async{ }, which has an await() function that returns the result of the coroutine. |
| launch{} cannot be used when you need the parallel execution of network calls. | Use async only when you need the parallel execution network calls. |
| launch{} will not block your main thread. | Async will block the main thread at the entry point of the await() function. |
| Execution of other parts  of the code will not wait for the launch result since launch is not a suspend call | Execution of the other parts of the code will have to wait for the result of the await() function. |
| The launch can’t work like async in any case or condition. | If you use async and do not wait for the result, it will work the same as the launch. |
| Launch can be used at places if you don’t need the result from the method called. | Use async when you need the results from the multiple tasks that run in parallel. |

|  |
| --- |
|  |