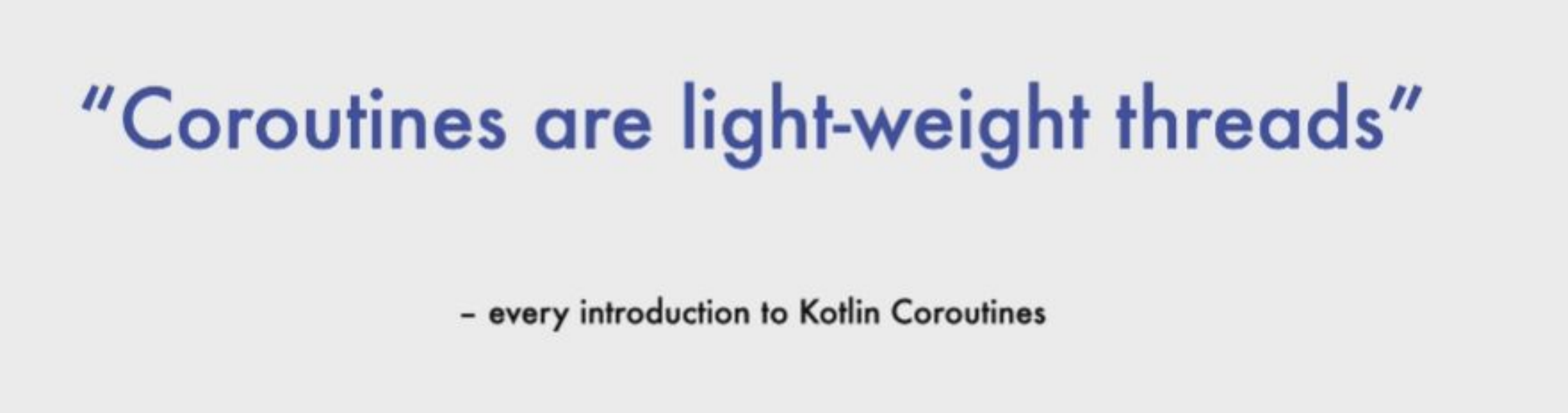
Coroutines

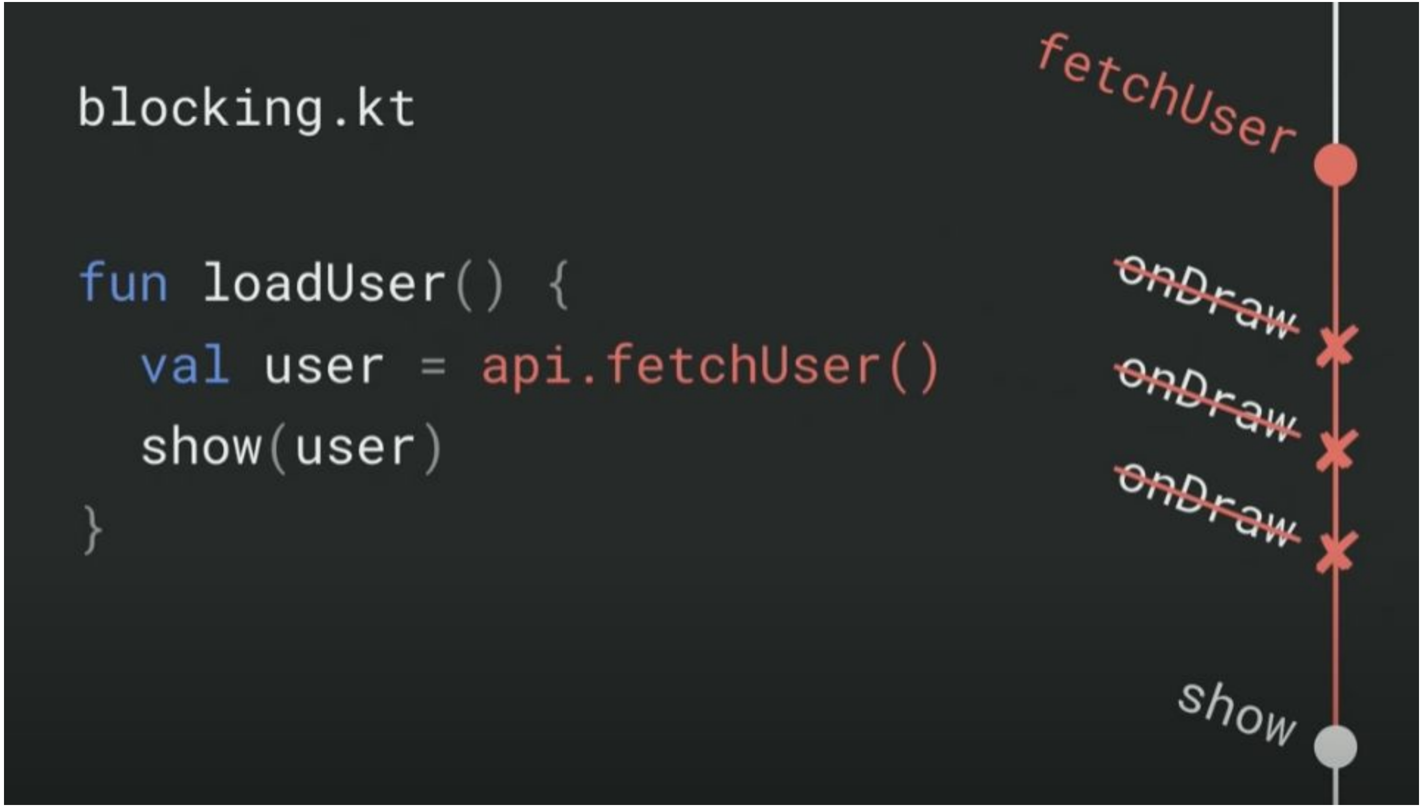


> “Coroutines are light-weight threads”
>
> – every introduction to Kotlin Coroutines

Coroutines

coroutines => co routine => cooperative routine

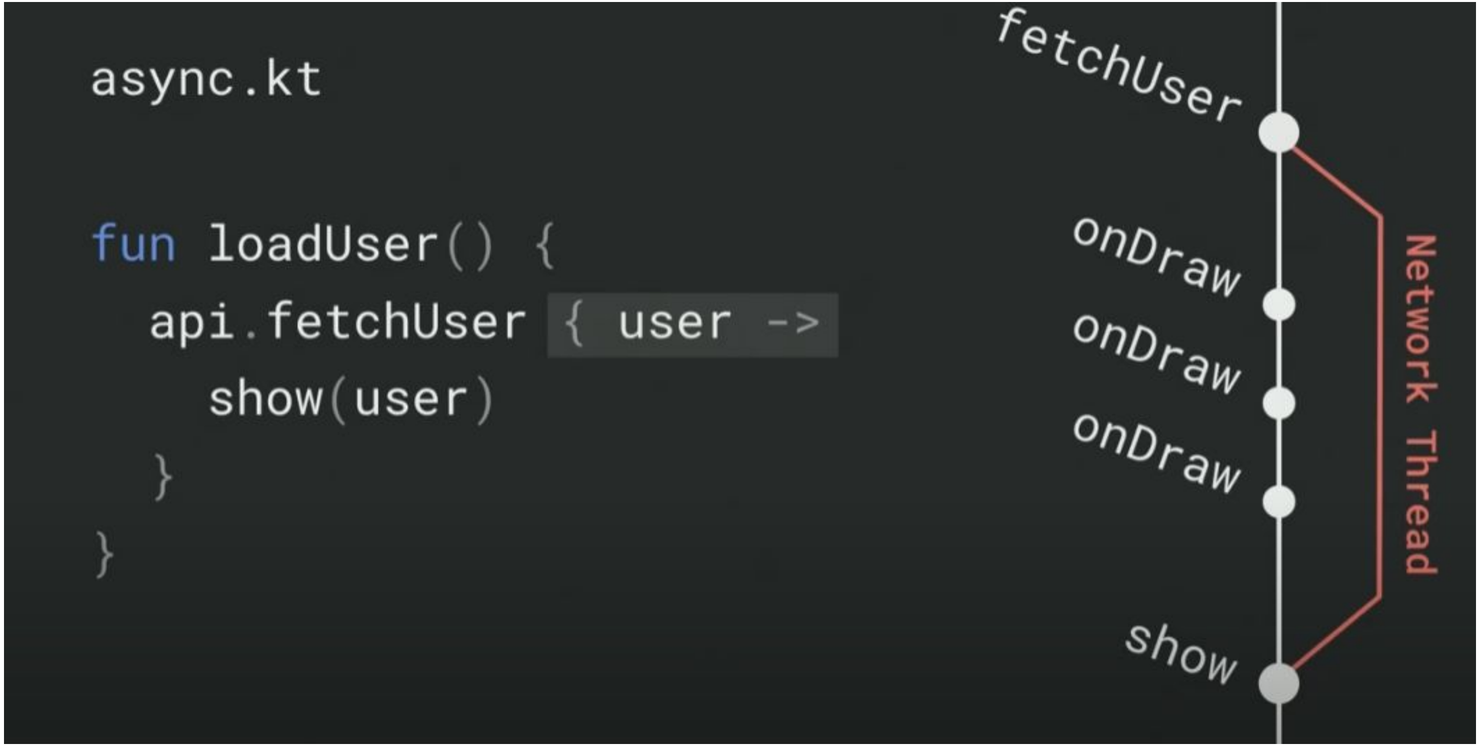
Why do we need this?



blocking.kt

```kotlin
fun loadUser() {
   val user = api.fetchUser()
   show(user)
}
```

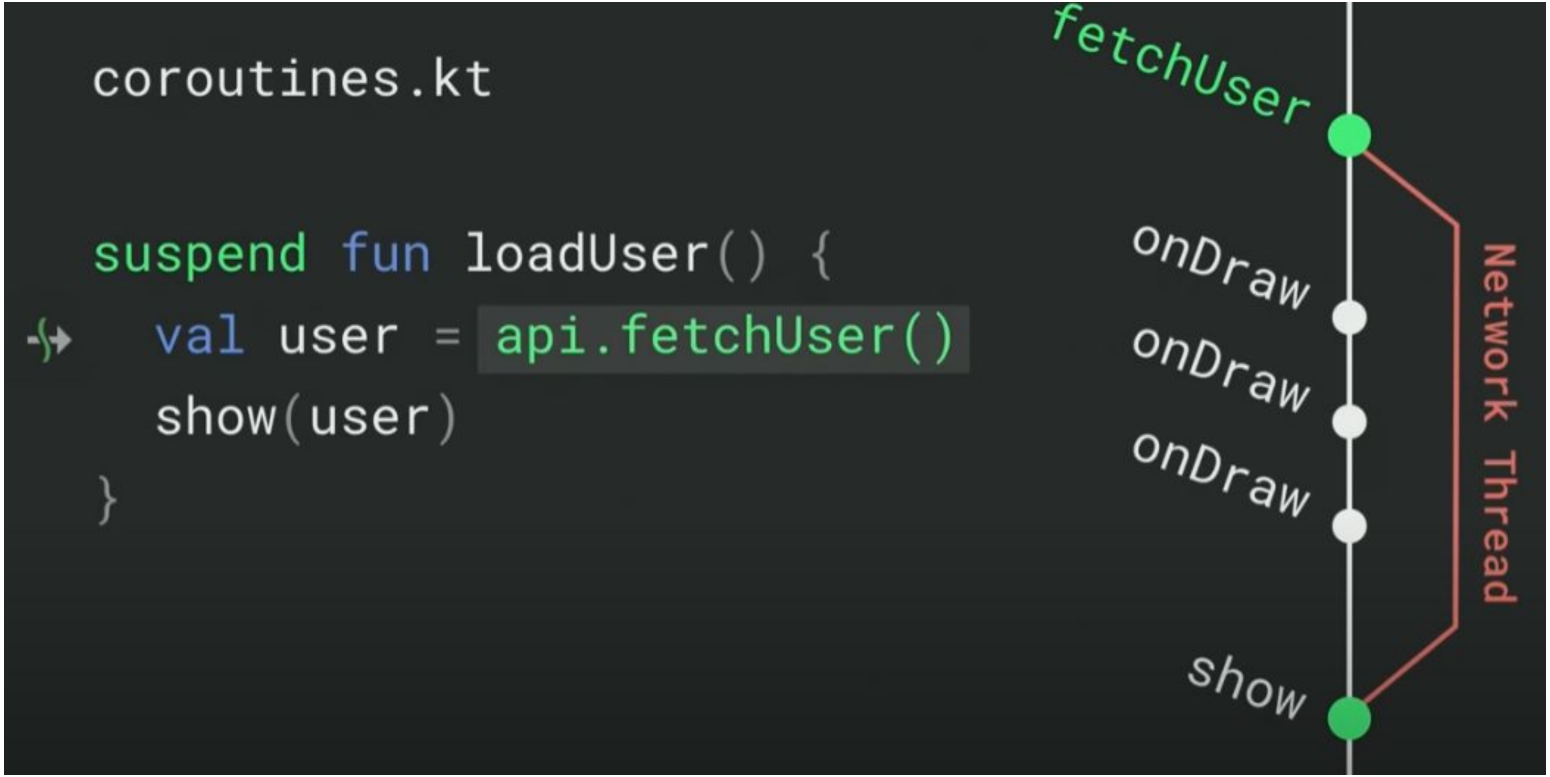
Why do we need this?



async.kt

```
fun loadUser() {
  api.fetchUser { user ->
    show(user)
  }
}
```

Why do we need this?



coroutines.kt

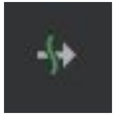
```kotlin
suspend fun loadUser() {
    val user = api.fetchUser()
    show(user)
}
```

Why do we need this?

Callbacks vs RX vs Coroutines example

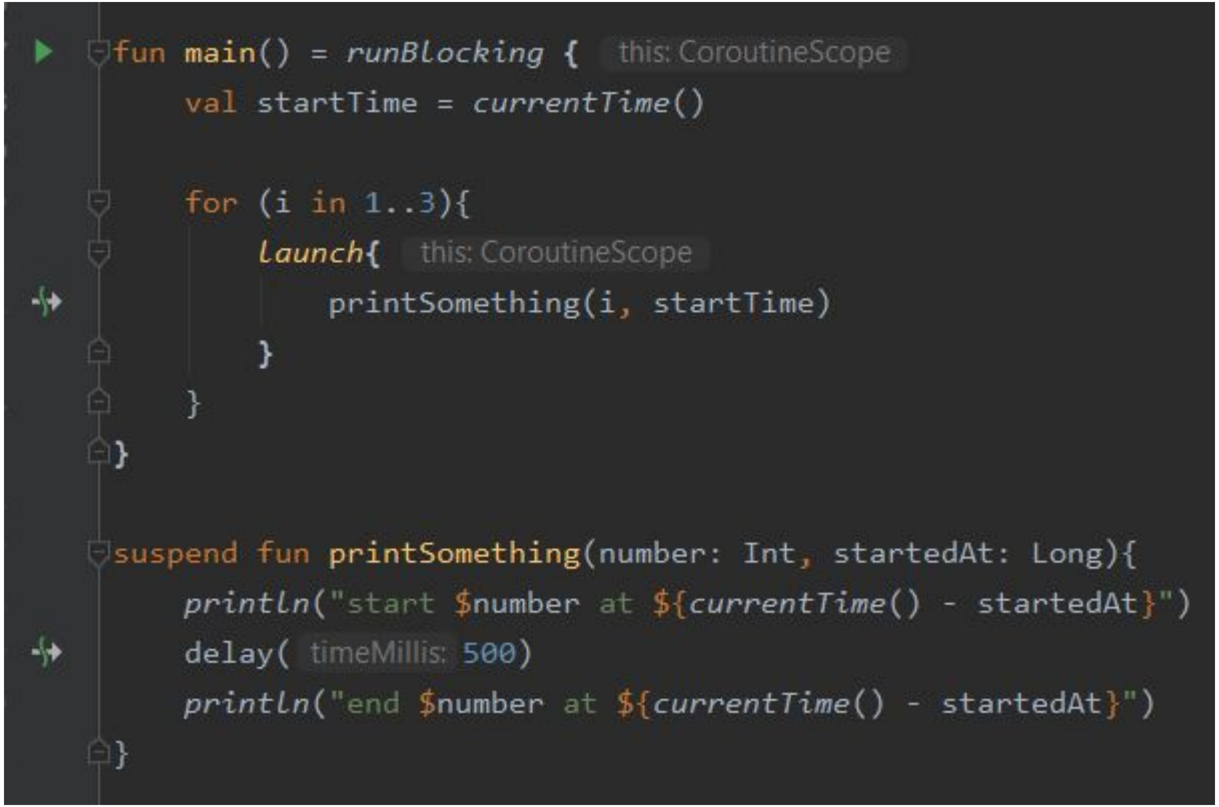
Suspend functions

- special funs that perform some long operation(s) and can be suspended
- can only be called from another suspend fun or a Coroutine
- if you calling suspend fun in another suspend fun, new coroutine is not launched
- Suspend fun can be paused at suspension points

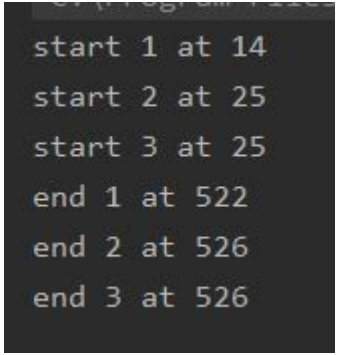


Suspension points

Suspension points

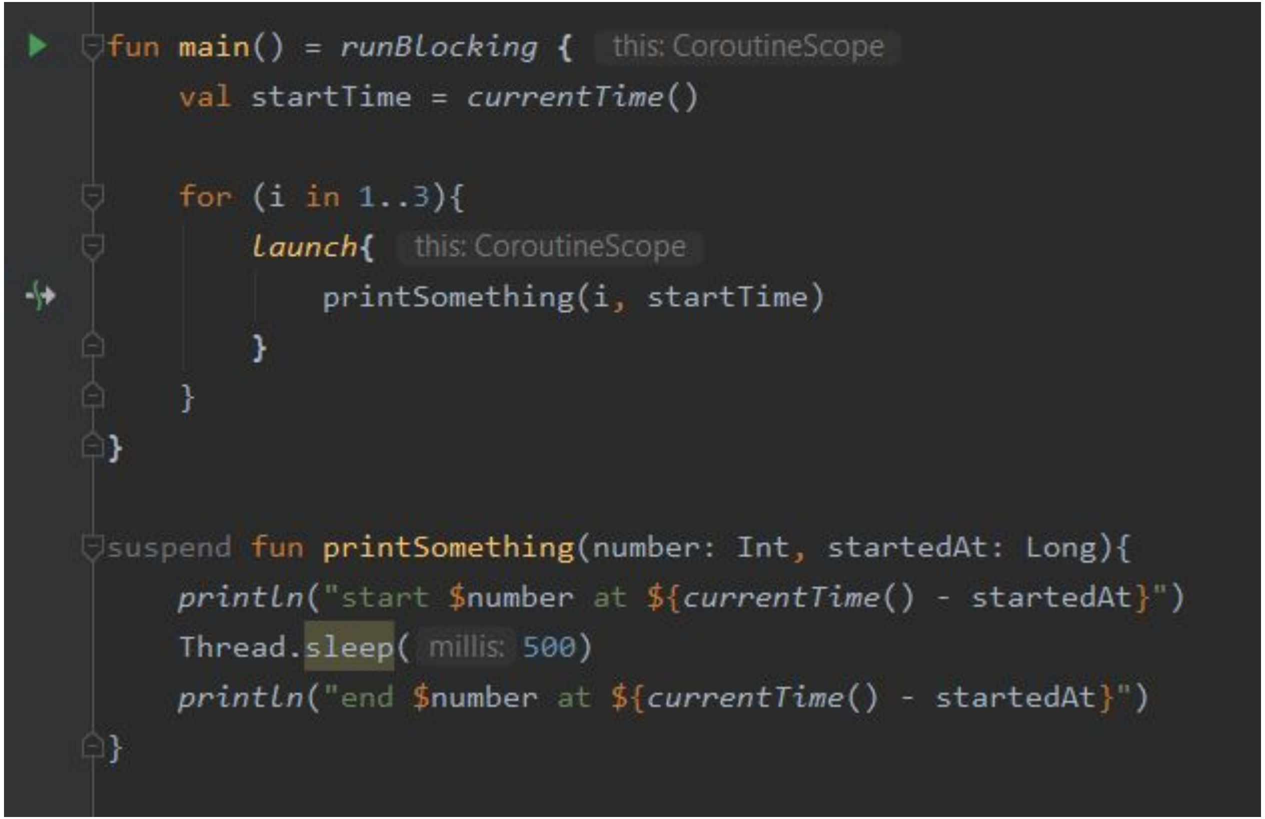


```kotlin
fun main() = runBlocking {  this: CoroutineScope
    val startTime = currentTime()

    for (i in 1..3){
        launch{  this: CoroutineScope
            printSomething(i, startTime)
        }
    }
}

suspend fun printSomething(number: Int, startedAt: Long){
    println("start $number at ${currentTime() - startedAt}")
    delay( timeMillis: 500)
    println("end $number at ${currentTime() - startedAt}")
}
```

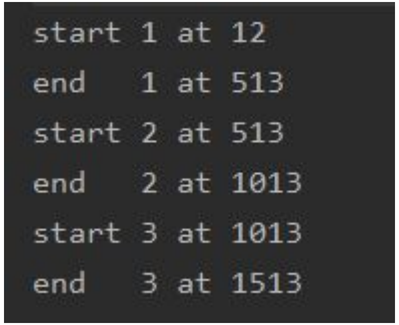


```
start 1 at 14
start 2 at 25
start 3 at 25
end 1 at 522
end 2 at 526
end 3 at 526
```

Suspension points

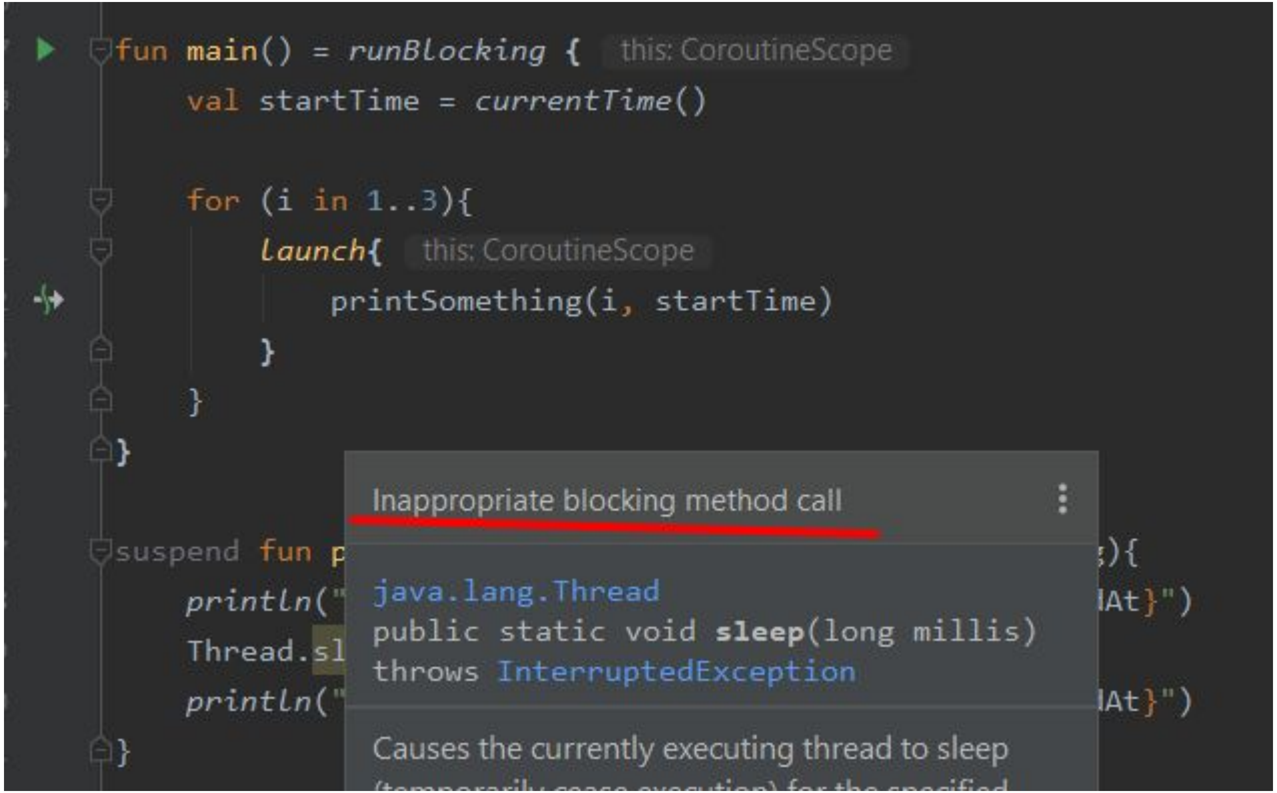


```
fun main() = runBlocking {  this: CoroutineScope
    val startTime = currentTime()

    for (i in 1..3){
        launch{  this: CoroutineScope
            printSomething(i, startTime)
        }
    }
}

suspend fun printSomething(number: Int, startedAt: Long){
    println("start $number at ${currentTime() - startedAt}")
    Thread.sleep( millis: 500)
    println("end $number at ${currentTime() - startedAt}")
}
```



```
start 1 at 12
end   1 at 513
start 2 at 513
end   2 at 1013
start 3 at 1013
end   3 at 1513
```

Suspension points

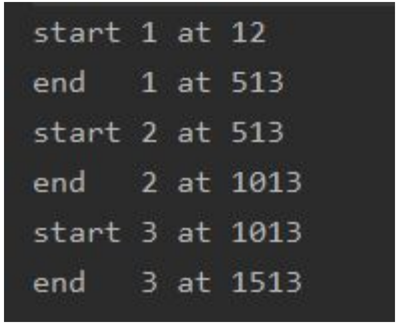


```
fun main() = runBlocking {  this: CoroutineScope
    val startTime = currentTime()

    for (i in 1..3){
        launch{  this: CoroutineScope
            printSomething(i, startTime)
        }
    }
}

suspend fun p                                          ){
    println("                                          At}")
    Thread.sl
    println("                                          At}")
}
```

Inappropriate blocking method call

```
java.lang.Thread
public static void sleep(long millis)
throws InterruptedException
```

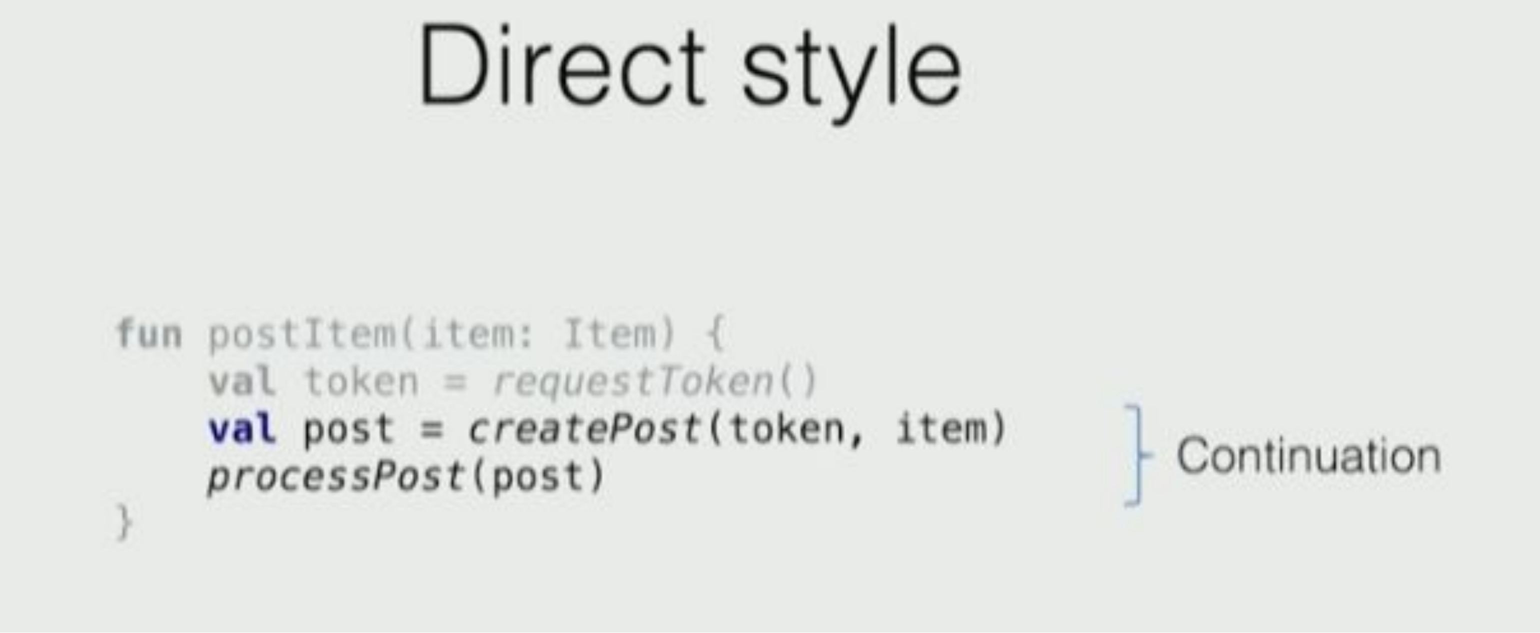
Causes the currently executing thread to sleep (temporarily cease execution) for the specified



```
start 1 at 12
end   1 at 513
start 2 at 513
end   2 at 1013
start 3 at 1013
end   3 at 1513
```

There is no magic in coroutines

Coroutines under the hood

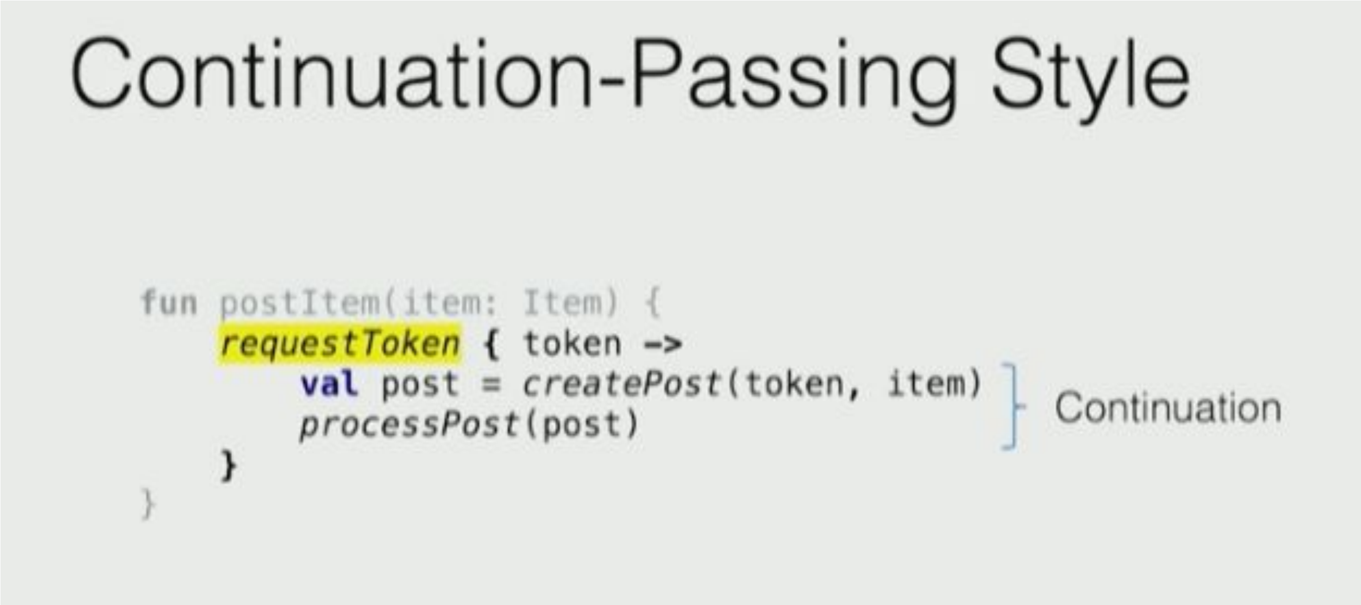


Direct style

```
fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

Continuation

Coroutines under the hood

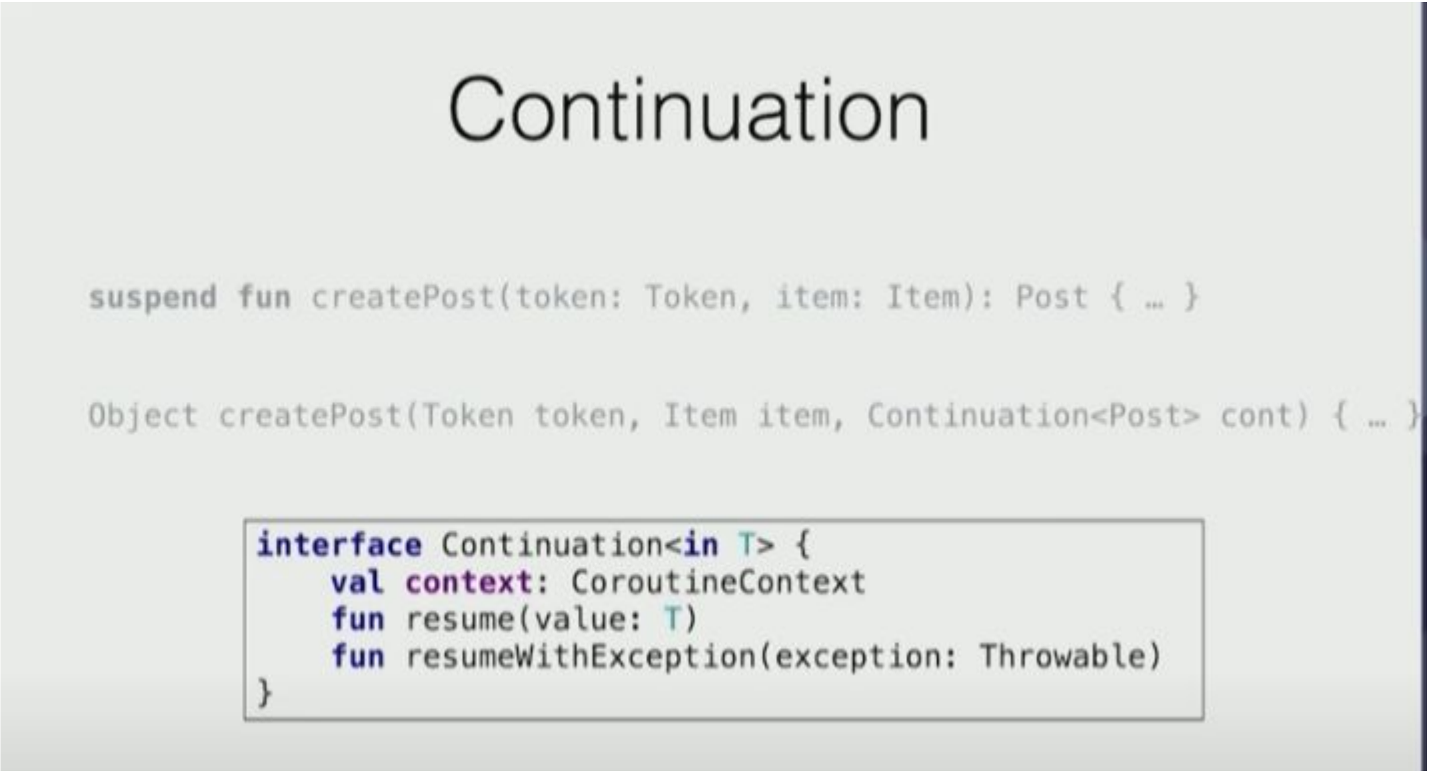


Continuation-Passing Style

```
fun postItem(item: Item) {
    requestToken { token ->
        val post = createPost(token, item)
        processPost(post)
    }
}
```

Continuation

Coroutines under the hood



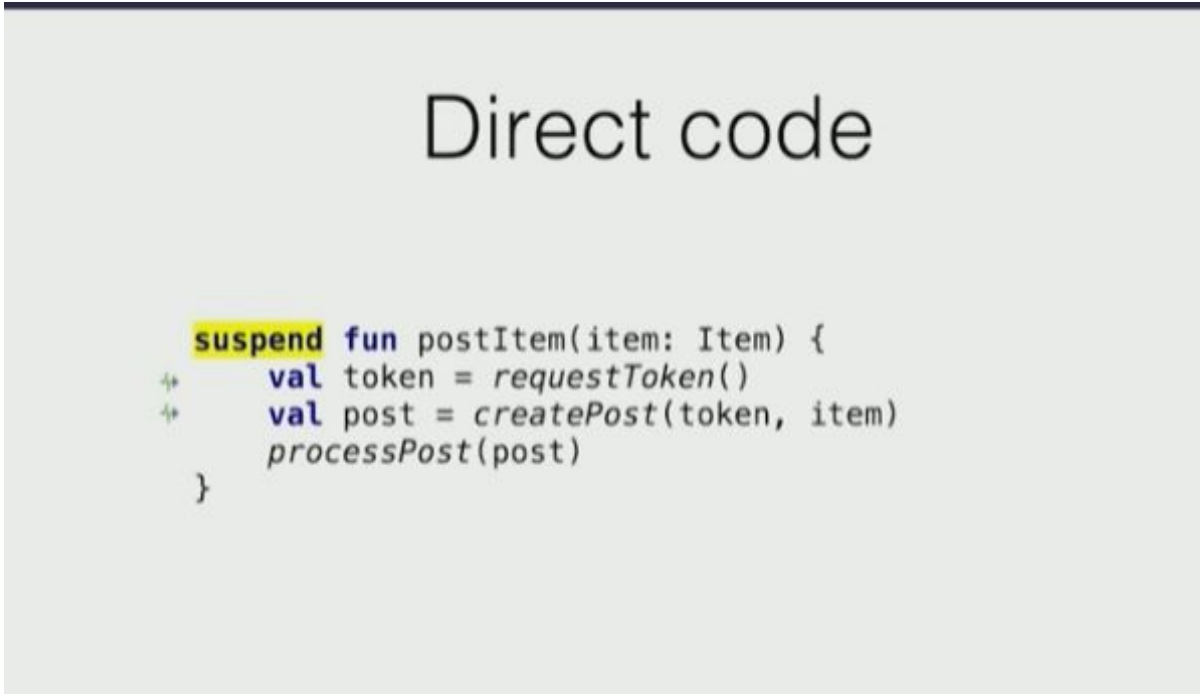
Continuation

```
suspend fun createPost(token: Token, item: Item): Post { … }
```

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

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

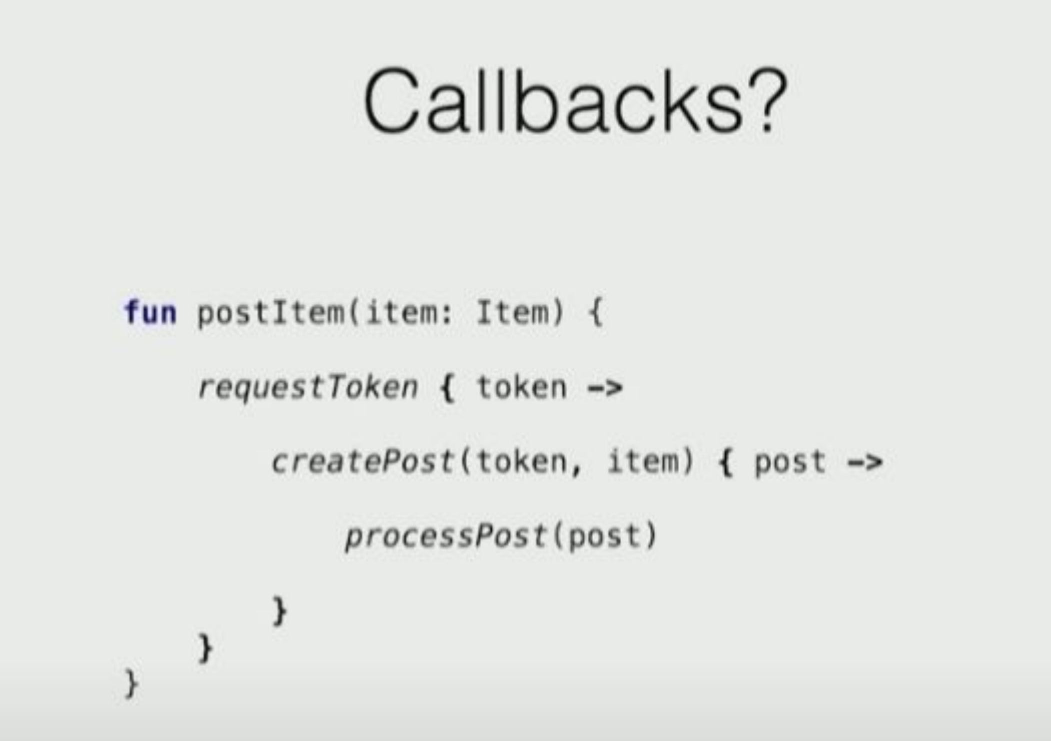
Coroutines under the hood



Direct code

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

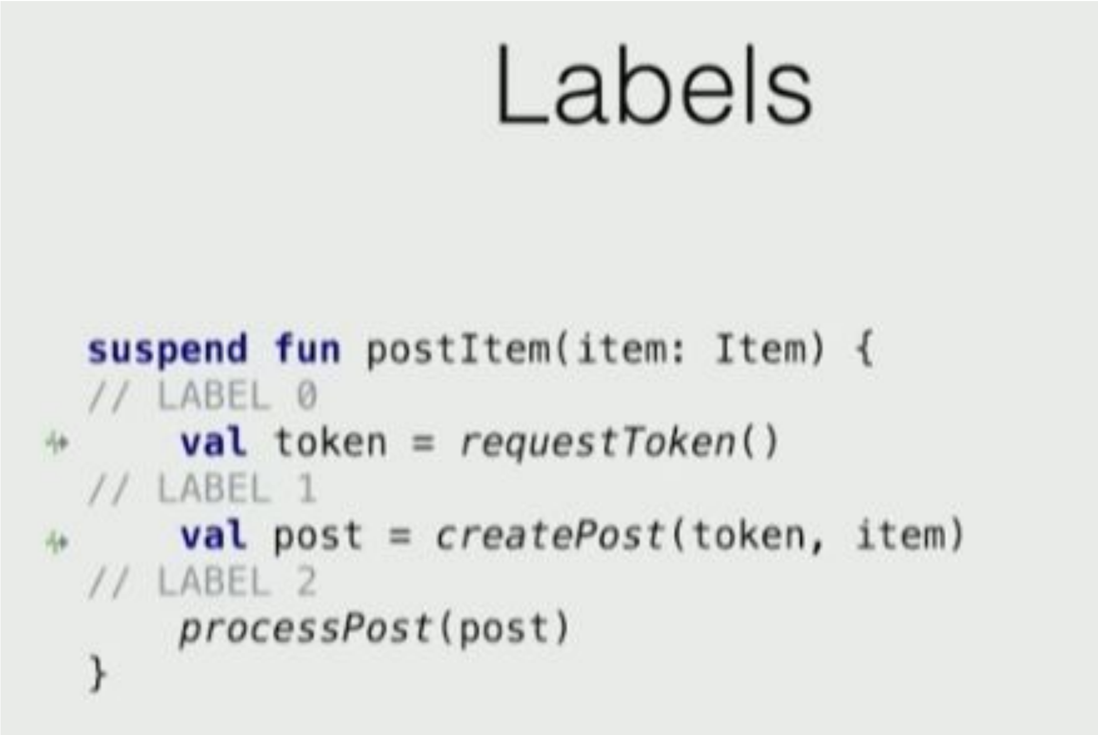
Coroutines under the hood



Callbacks?

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

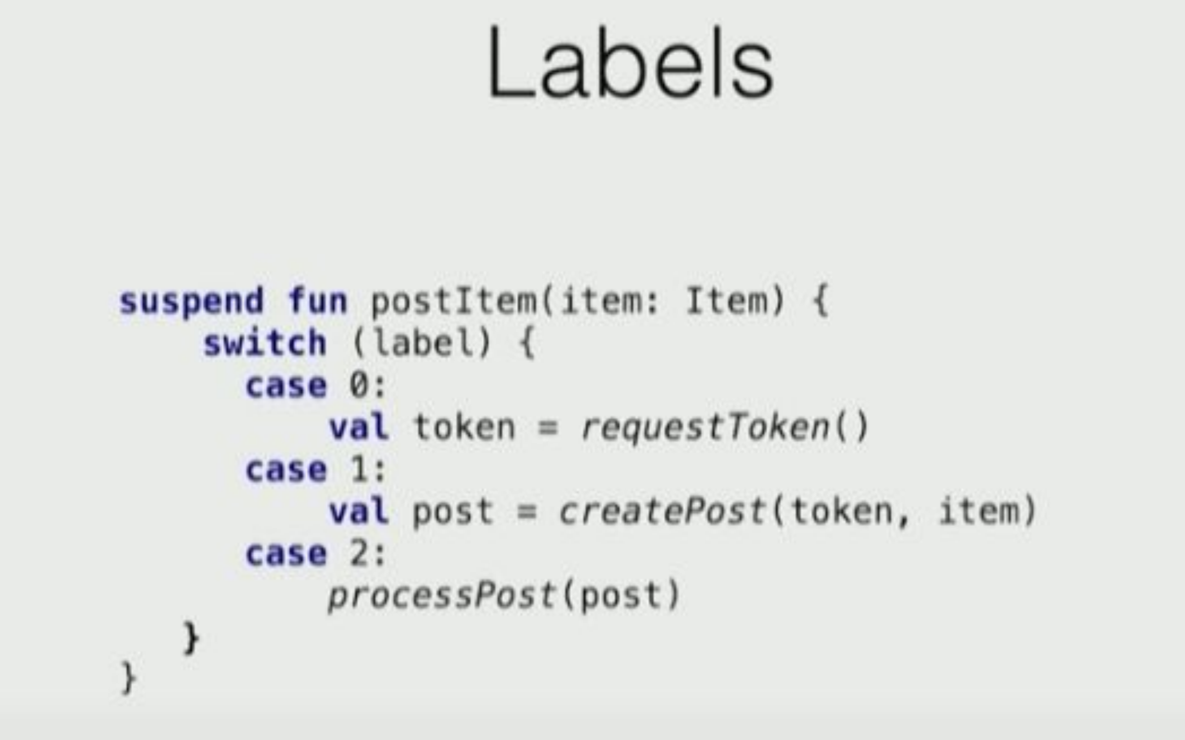
Coroutines under the hood



Labels

```
suspend fun postItem(item: Item) {
// LABEL 0
    val token = requestToken()
// LABEL 1
    val post = createPost(token, item)
// LABEL 2
    processPost(post)
}
```

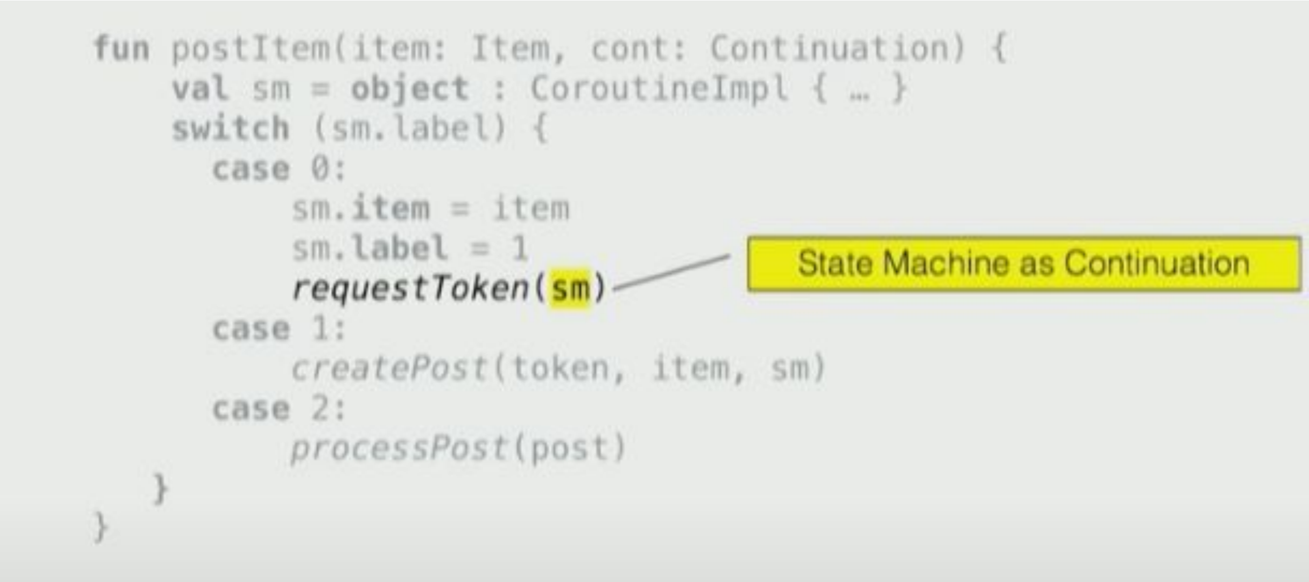
Coroutines under the hood



Labels

```
suspend fun postItem(item: Item) {
    switch (label) {
      case 0:
          val token = requestToken()
      case 1:
          val post = createPost(token, item)
      case 2:
          processPost(post)
   }
}
```

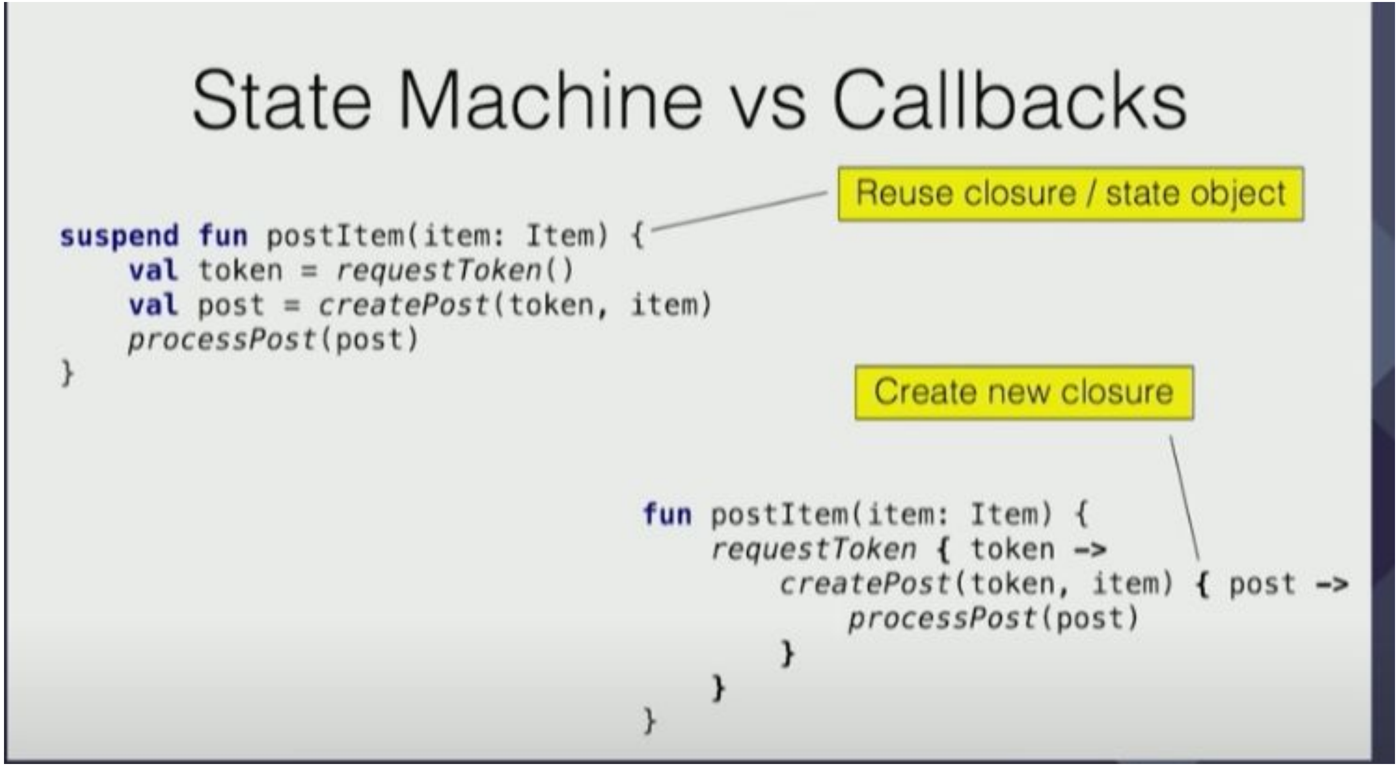
Coroutines under the hood



```
fun postItem(item: Item, cont: Continuation) {
    val sm = object : CoroutineImpl { … }
    switch (sm.label) {
      case 0:
          sm.item = item
          sm.label = 1
          requestToken(sm)
      case 1:
          createPost(token, item, sm)
      case 2:
          processPost(post)
    }
}
```

State Machine as Continuation

Coroutines under the hood



State Machine vs Callbacks

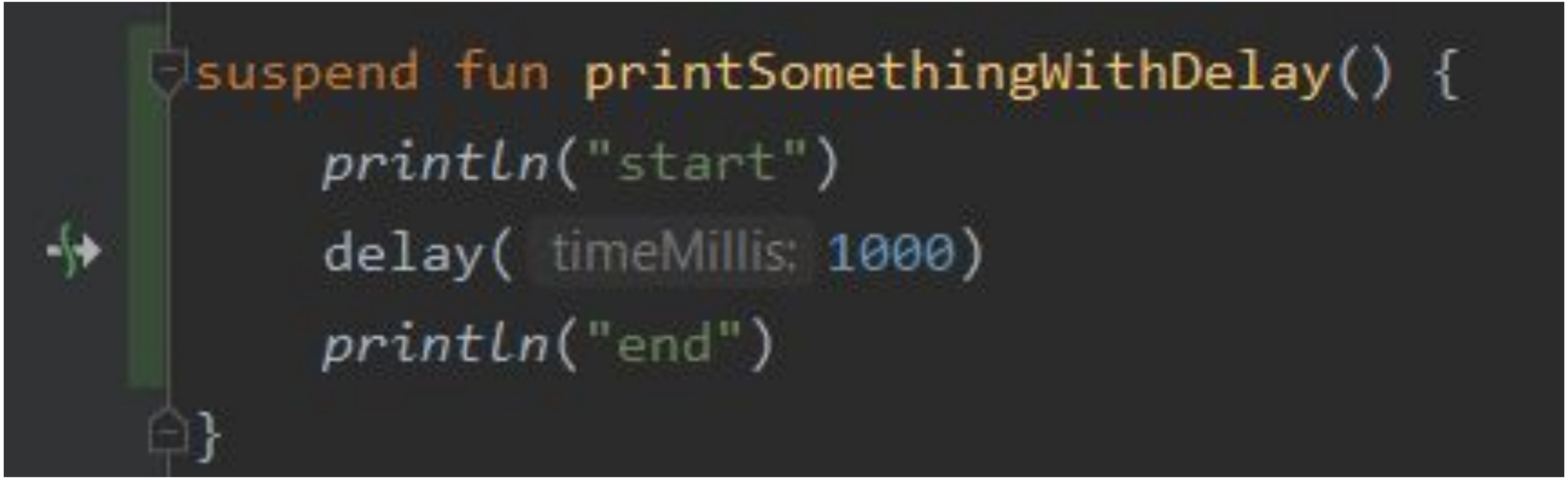
Reuse closure / state object

```
suspend fun postItem(item: Item) {
    val token = requestToken()
    val post = createPost(token, item)
    processPost(post)
}
```

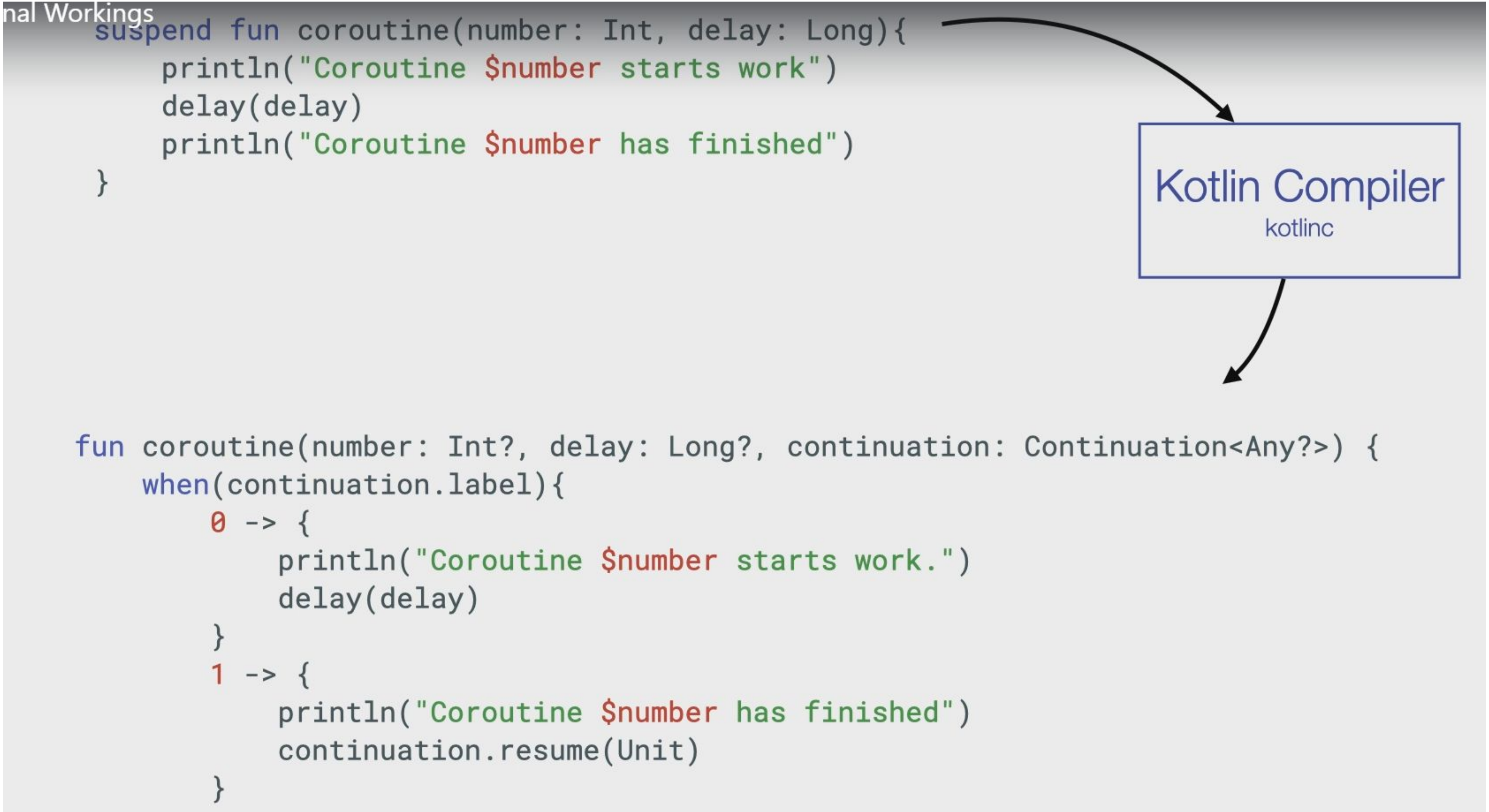
Create new closure

```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

Coroutines under the hood



```
suspend fun printSomethingWithDelay() {
    println("start")
    delay( timeMillis: 1000)
    println("end")
}
```



nal Workings

```
suspend fun coroutine(number: Int, delay: Long){
    println("Coroutine $number starts work")
    delay(delay)
    println("Coroutine $number has finished")
}
```

Kotlin Compiler
kotlinc

```
fun coroutine(number: Int?, delay: Long?, continuation: Continuation<Any?>) {
    when(continuation.label){
        0 -> {
            println("Coroutine $number starts work.")
            delay(delay)
        }
        1 -> {
            println("Coroutine $number has finished")
            continuation.resume(Unit)
        }
```

Coroutine builders

- launch{}
- async{}
- runBlocking{}

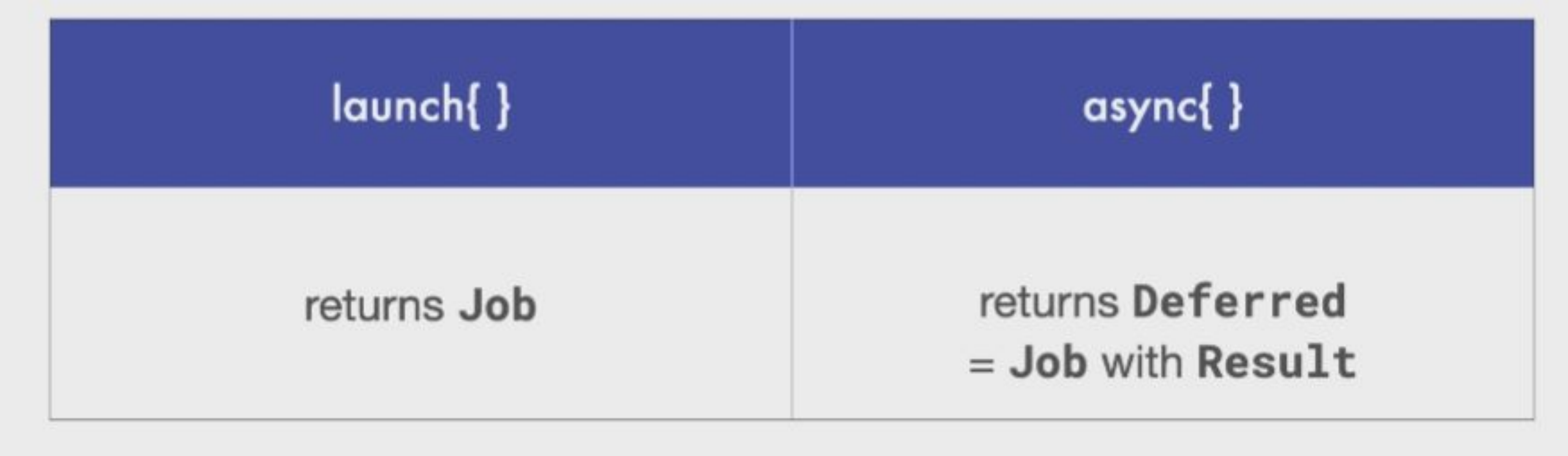
Coroutine builders - **launch{}**

- Launches a new coroutine without blocking the current thread (“fire and forget”)
- Returns a reference to the coroutine as a Job

Coroutine builders - **async{}**

Creates a coroutine and returns its future result as an implementation of Deferred

Coroutine builders - **launch{}** vs **async{}**



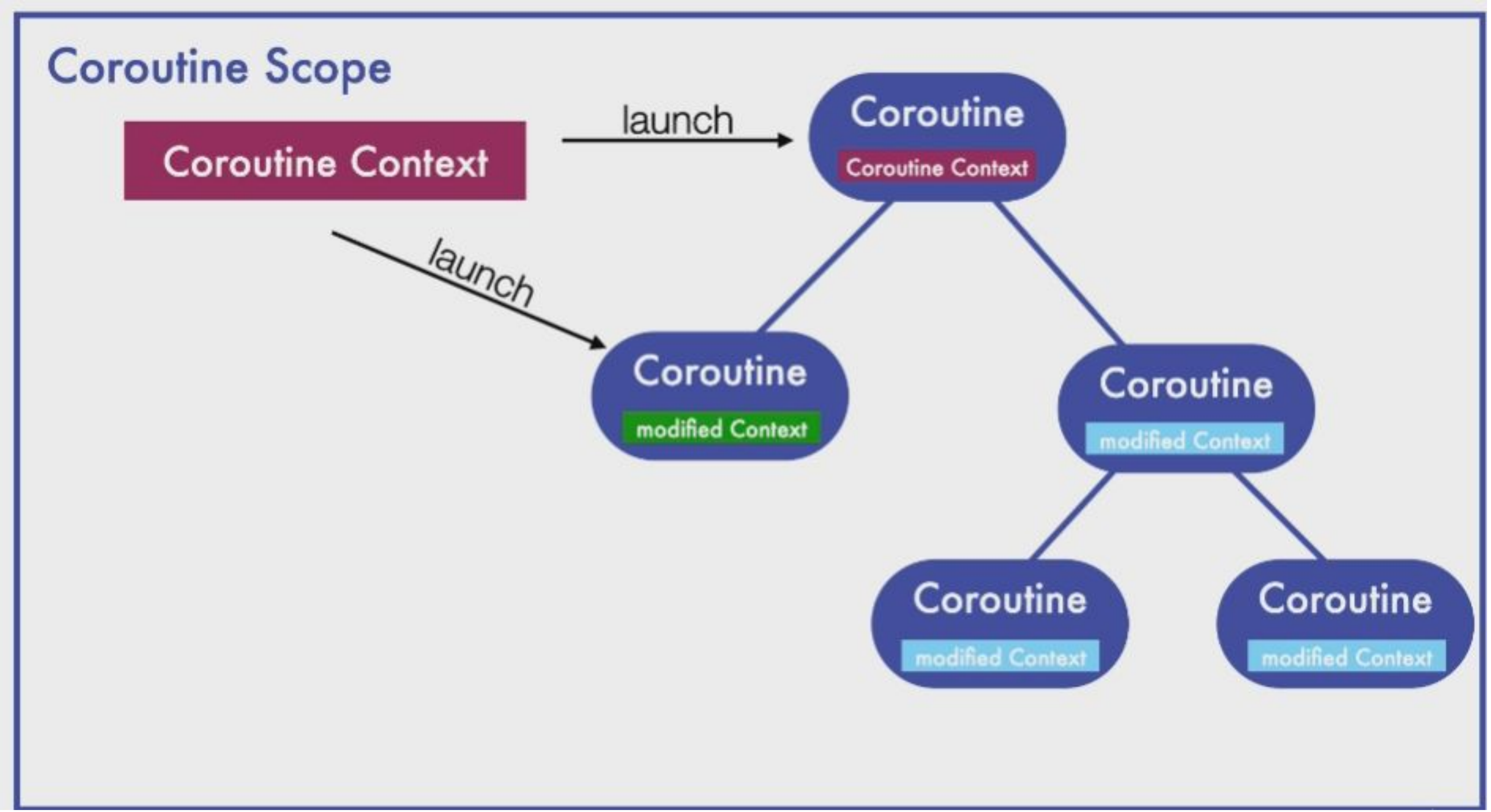
launch{ }	async{ }
returns **Job**	returns **Deferred** = **Job** with **Result**

Coroutine builders - **runBlocking{}**

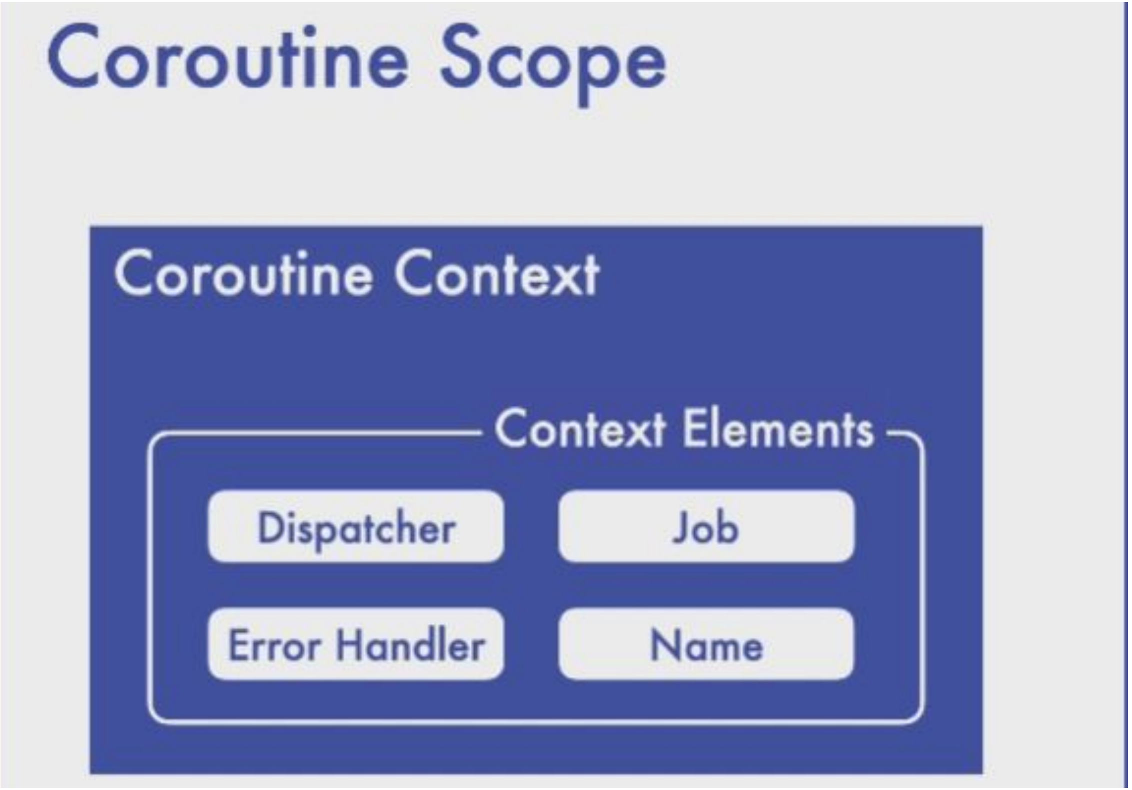
not extension fun on CoroutineScope, but regular fun

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.

Coroutine **scope** vs Coroutine **context**



Coroutine **scope** vs Coroutine **context**



CoroutineScope(...)

GlobalScope

A global CoroutineScope not bound to any job => no hierarchy

is used to launch top-level coroutines which are operating on the whole application lifetime and are not cancelled prematurely

Using ***async*** or ***launch*** on the instance of GlobalScope is highly discouraged.

ViewModelScope

Any coroutine launched in this scope is automatically canceled if the ViewModel is cleared.

uses Dispatchers.Main.immediate

uses SupervisorJob

uses CloseableCoroutineScope

LifecycleScope

Any coroutine launched in this scope is canceled when the Lifecycle is destroyed

Will be canceled in case of device rotation

(launch)whenCreated{} / whenStarted{} / whenResumed{}

Dispatchers

Dispatchers. **MAIN**

- only available in applications with UI
- special thread that can perform UI operations
- Defined as the dispatcher for the viewModelScope
- android main dispatcher uses Handler for main looper internally

Main-safety

Room, retrofit etc are not block thread, so could be run from main thread.

Retrofit manage this and notify coroutine after response received

When using the Room library to perform a database operation, Room uses a Dispatchers.IO to perform the database operations in a background thread. You don't have to explicitly specify any Dispatchers. Room does this for you.

but not-suspendable operations (like read from file etc) should be run from background

Dispatchers. IO

- to perform IO-related blocking operations
- Limit of 64 threads or the number of cores (whichever is larger)
- Uses shared thread pool internally
- This dispatcher shares threads with a Dispatchers.Default

Dispatchers. **DEFAULT**

- Default dispatcher if no other is defined
- optimized for cpu-intensive work
- Uses shared thread pool internally
- max number of threads = CPUcores number

Dispatchers. **UNCONFINED**

- not confined to any thread
- internally running on thread coroutine was started on
- might switches threads on context switches in suspend functions
- should not normally be used

Custom dispatcher

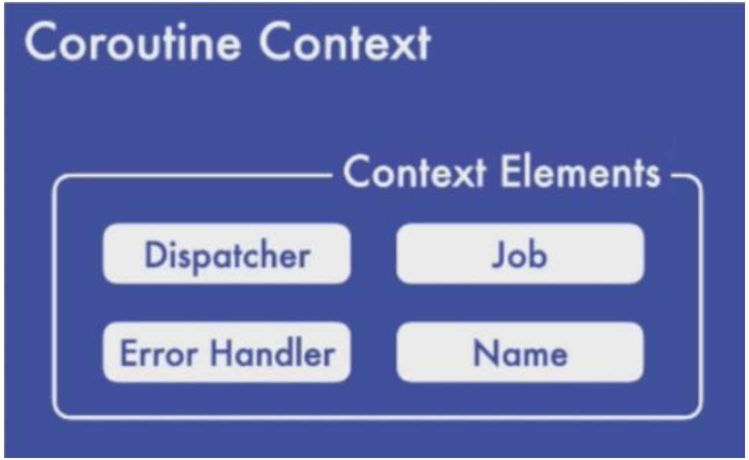
- newFixedThreadPool
- newSingleThreadExecutor

```
Executors.newFixedThreadPool(3).asCoroutineDispatcher()
```

Dispatchers

withContext{}

- another Dispatcher
- another Job
- another ErrorHandler
- another Name



```
withContext(Dispatchers.Default + CoroutineName("some cool name for coroutine")){}
```

Coroutine Start

[DEFAULT] -- immediately schedules coroutine for execution according to its context;

[LAZY] -- starts coroutine lazily, only when it is needed;

[ATOMIC] -- atomically (in a non-cancellable way) schedules coroutine for execution according to its context;

[UNDISPATCHED] -- immediately executes coroutine until its first suspension point in the current thread.

CoroutineStart.LAZY

If coroutine [Job] is cancelled before it even had a chance to start executing, then it will not start its execution at all, but will complete with an exception.

lazy_start.kt

Coroutine scope & Structured Concurrency

Coroutine scope & Structured Concurrency

Every coroutine needs to be started in a logical scope with a limited life-time

Coroutine scope & Structured Concurrency

Coroutines (actually, Jobs) started in the same scope form a hierarchy

Coroutine scope & Structured Concurrency

Coroutines (actually, Jobs) started in the same scope form a hierarchy

“We don’t recommend passing jobs in the context parameter to coroutine builders in modern code”

job_hierarchy_0.kt
job_hierarchy_1.kt

Coroutine scope & Structured Concurrency

A parent job won't complete, until all of its children have completed

Coroutine scope & Structured Concurrency

Cancelling a parent will cancel all children.
Cancelling a child won't cancel the parent or siblings

Coroutine scope & Structured Concurrency

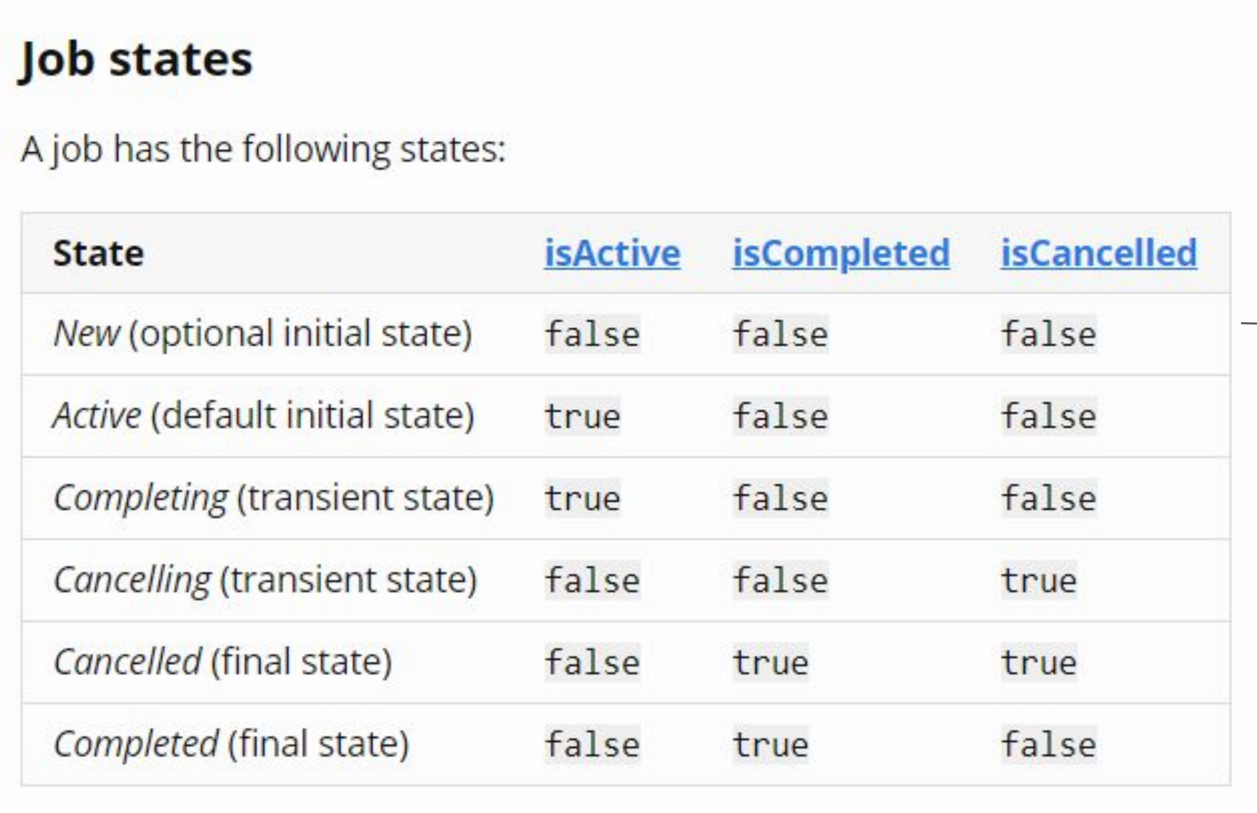
If a child coroutine fails, the exception is propagated upwards and depending on the job type, either all siblings are cancelled or not

Job vs SupervisorJob

cancellable thing with a life-cycle that culminates in its completion

`Job` interface and all its derived interfaces are not stable for inheritance in 3rd party libraries, as new methods might be added to this interface in the future, but is stable for use.

job_vs_supervisor_job.kt



Job states

A job has the following states:

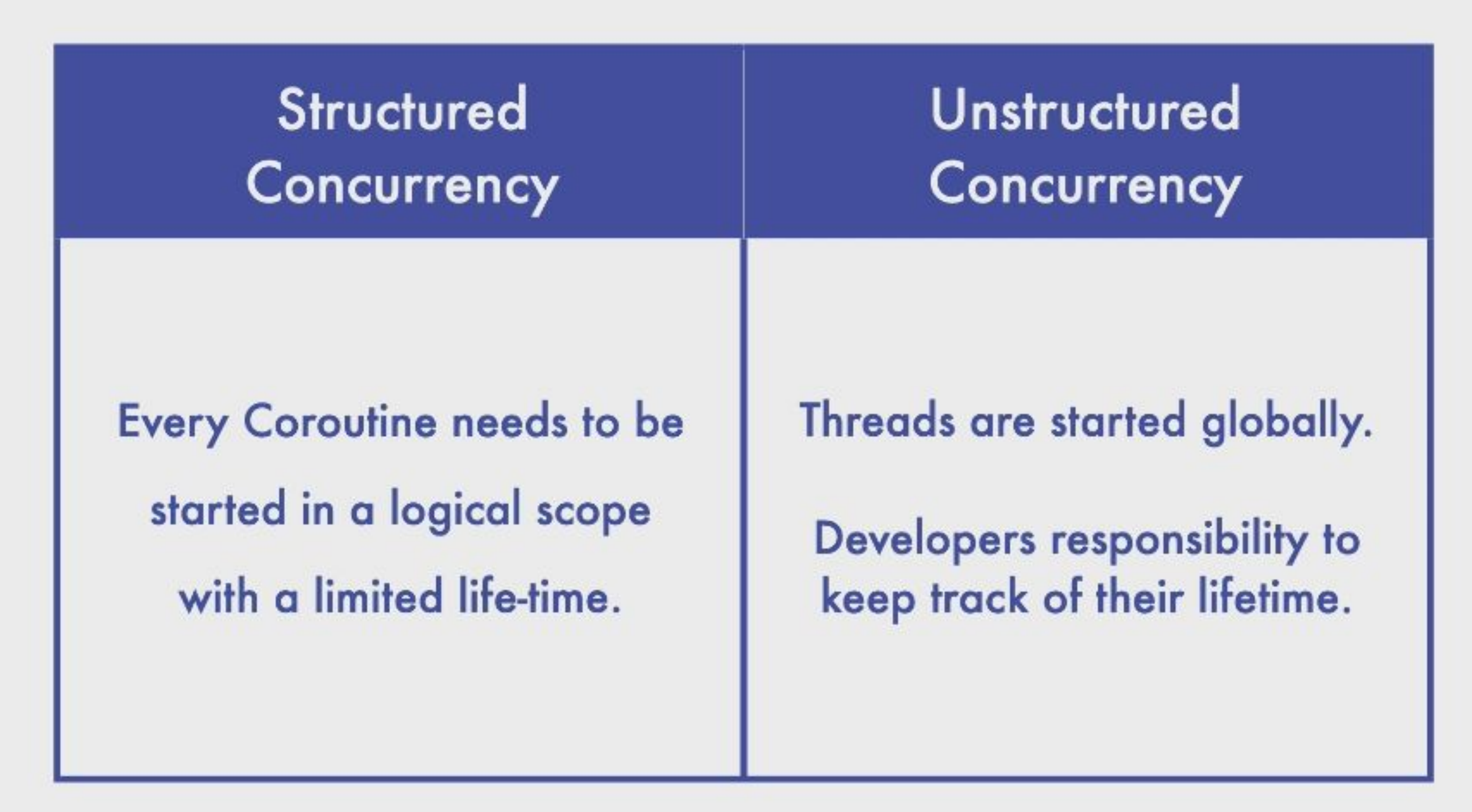
State	isActive	isCompleted	isCancelled
New (optional initial state)	`false`	`false`	`false`
Active (default initial state)	`true`	`false`	`false`
Completing (transient state)	`true`	`false`	`false`
Cancelling (transient state)	`false`	`false`	`true`
Cancelled (final state)	`false`	`true`	`true`
Completed (final state)	`false`	`true`	`false`

→ CoroutineStart.*LAZY*

coroutineScope{} / supervisorScope{}

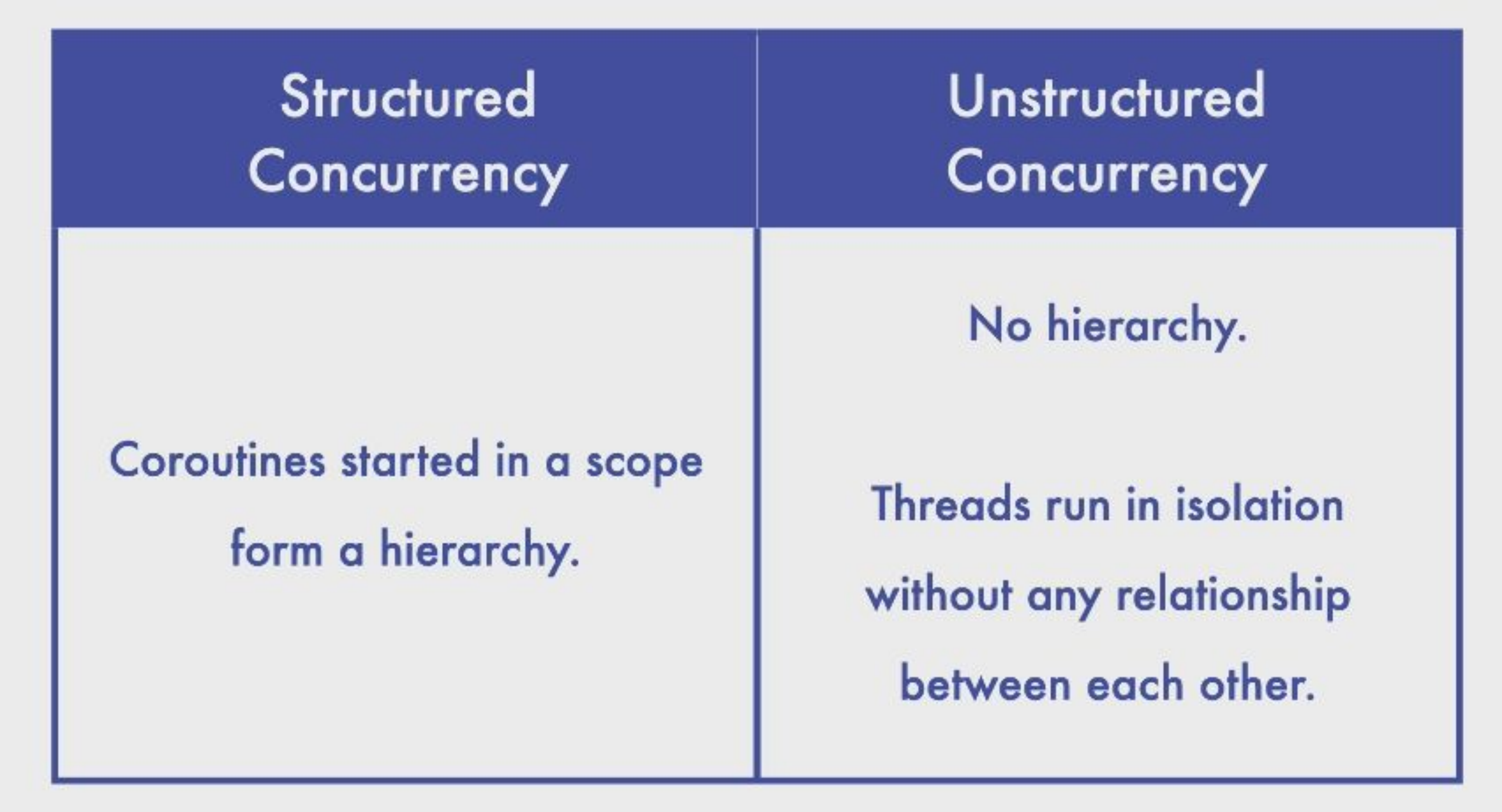
scoping_functions.kt

Coroutine scope & Structured Concurrency



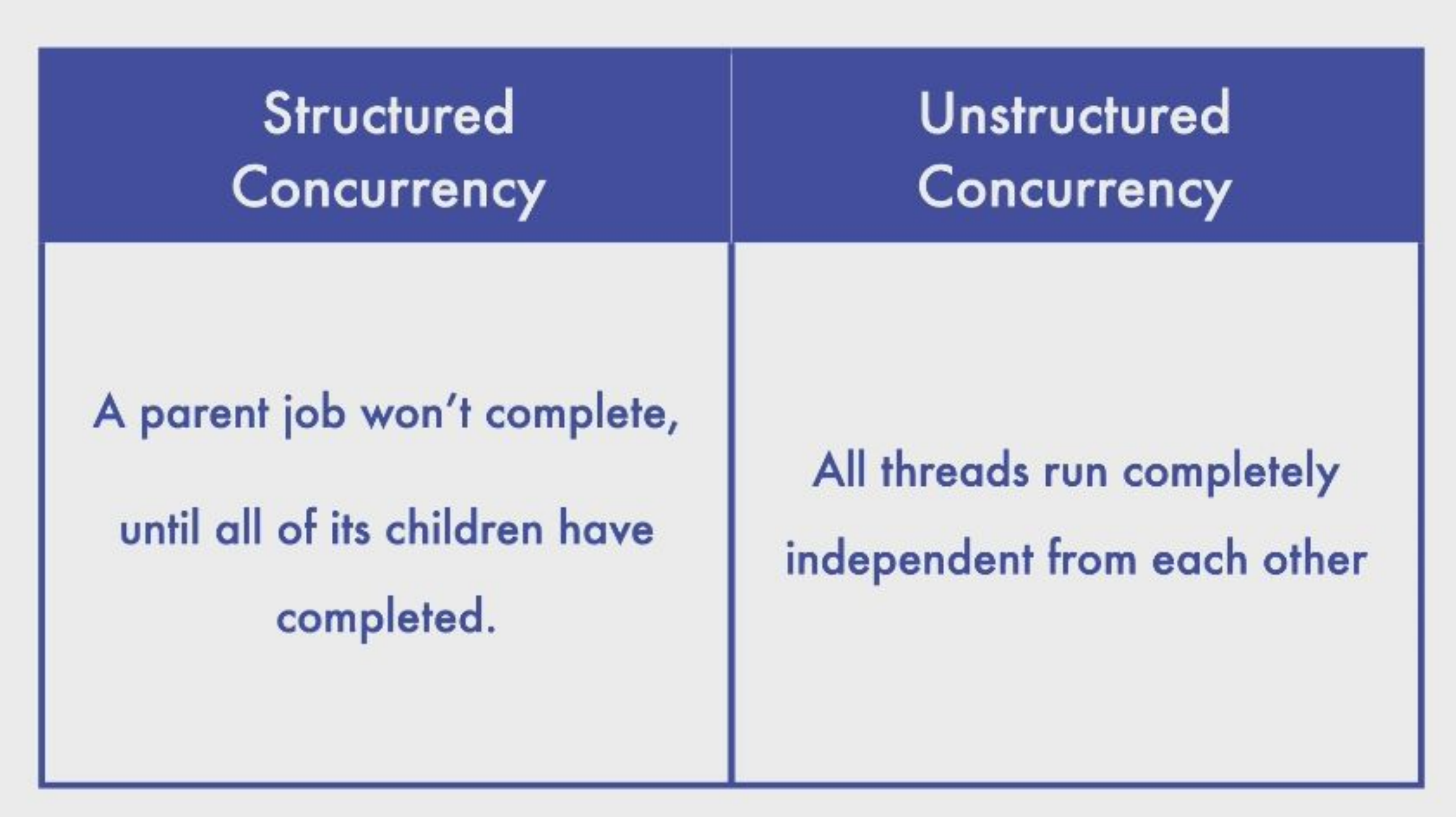
Structured Concurrency	Unstructured Concurrency
Every Coroutine needs to be started in a logical scope with a limited life-time.	Threads are started globally. Developers responsibility to keep track of their lifetime.

Coroutine scope & Structured Concurrency



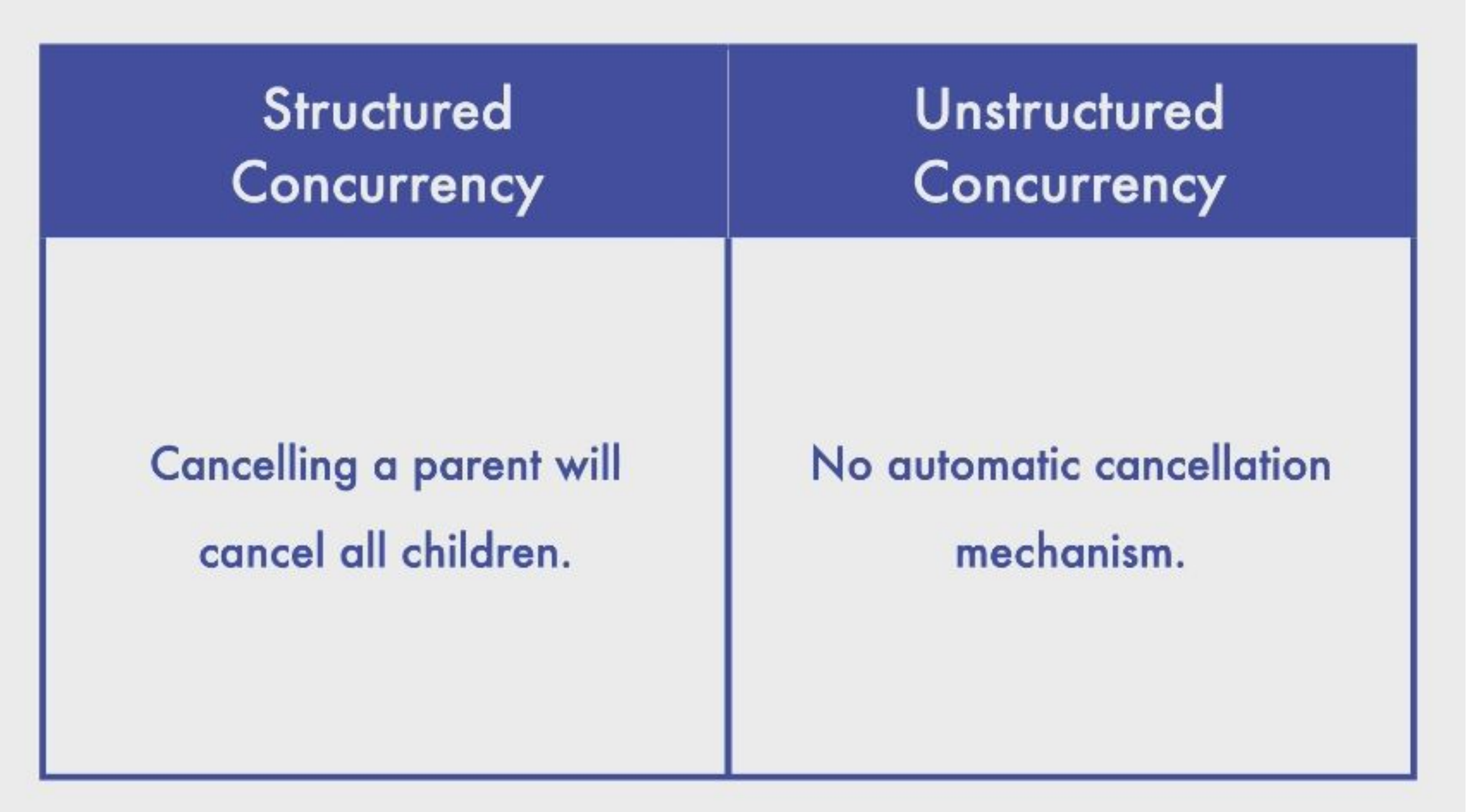
Structured Concurrency	Unstructured Concurrency
Coroutines started in a scope form a hierarchy.	No hierarchy. Threads run in isolation without any relationship between each other.

Coroutine scope & Structured Concurrency



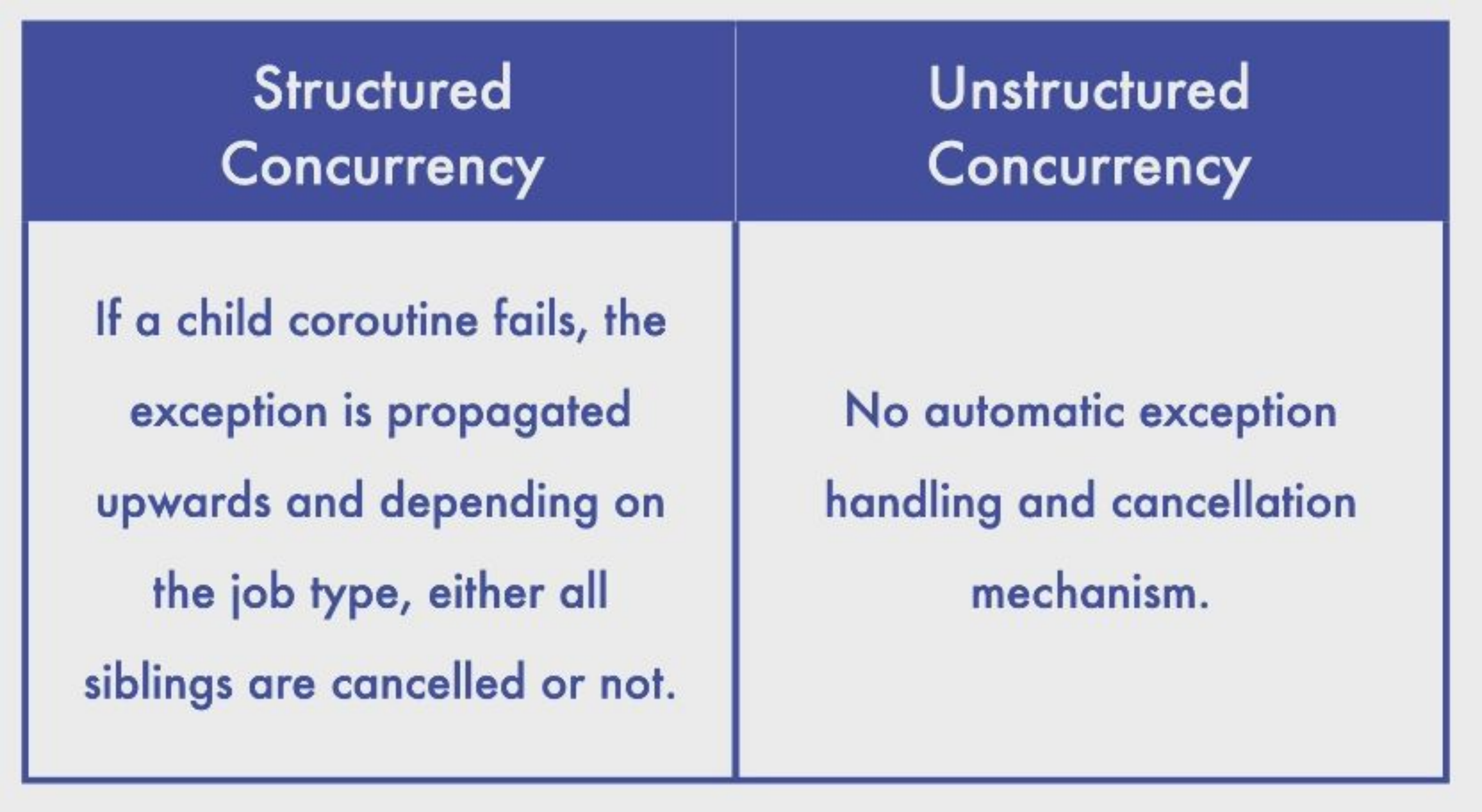
Structured Concurrency	Unstructured Concurrency
A parent job won't complete, until all of its children have completed.	All threads run completely independent from each other

Coroutine scope & Structured Concurrency



Structured Concurrency	Unstructured Concurrency
Cancelling a parent will cancel all children.	No automatic cancellation mechanism.

Coroutine scope & Structured Concurrency



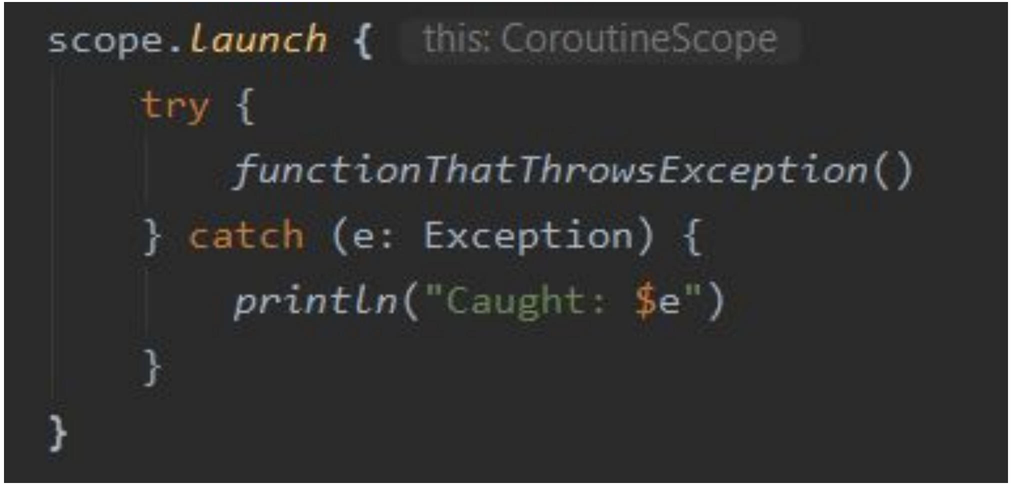
Structured Concurrency	Unstructured Concurrency
If a child coroutine fails, the exception is propagated upwards and depending on the job type, either all siblings are cancelled or not.	No automatic exception handling and cancellation mechanism.

Cancellation

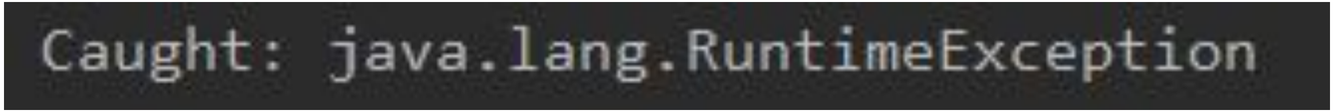
cancellation.kt
cancellation_cooperative.kt

Error handling

try-catch



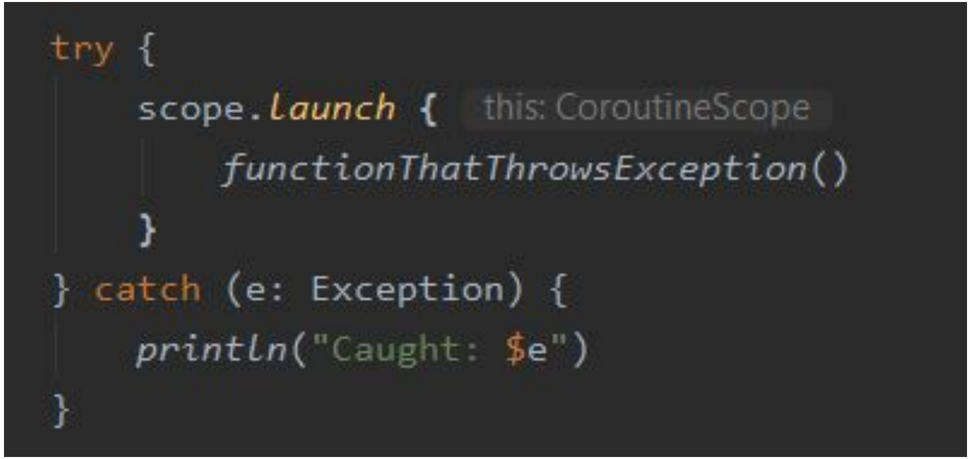
```
scope.launch {  this: CoroutineScope
    try {
        functionThatThrowsException()
    } catch (e: Exception) {
        println("Caught: $e")
    }
}
```



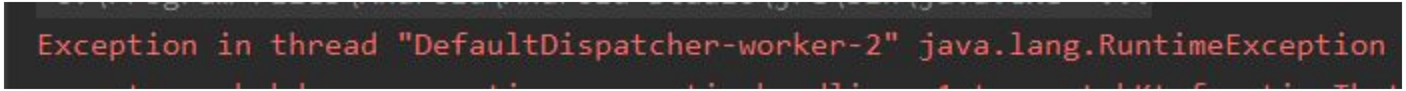
```
Caught: java.lang.RuntimeException
```

Error handling

try-catch



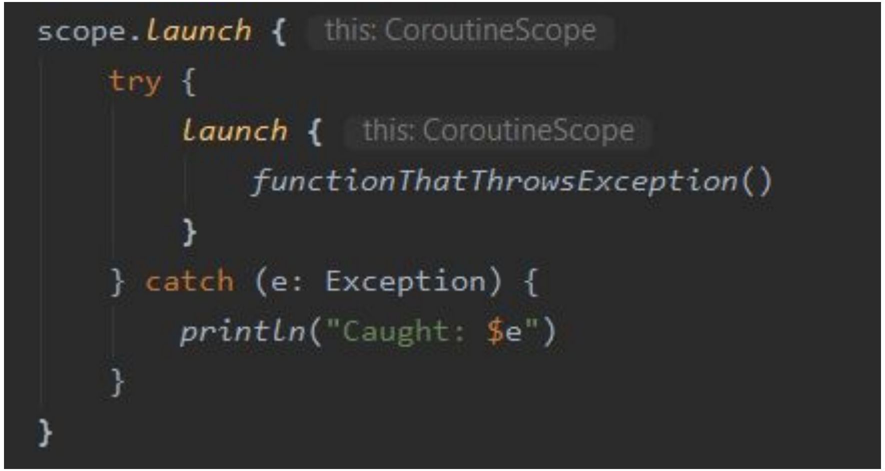
```
try {
    scope.launch {  this: CoroutineScope
        functionThatThrowsException()
    }
} catch (e: Exception) {
    println("Caught: $e")
}
```



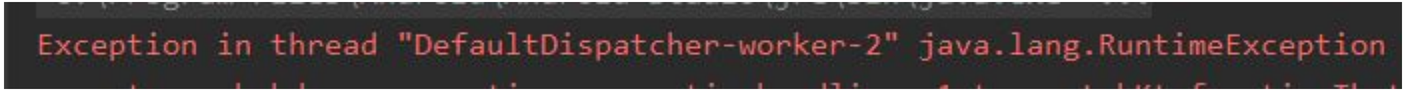
```
Exception in thread "DefaultDispatcher-worker-2" java.lang.RuntimeException
```

Error handling

try-catch

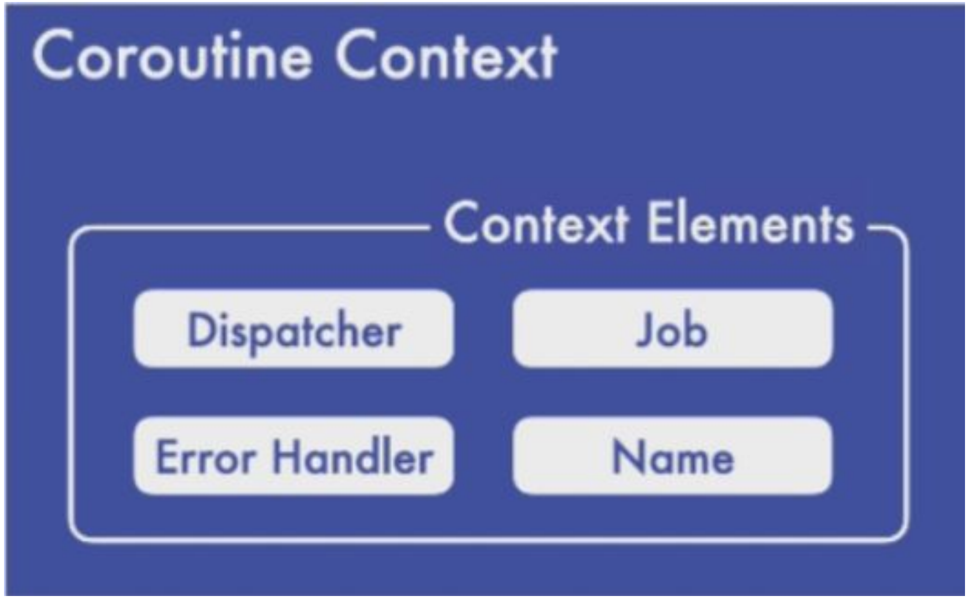


```
scope.launch {  this: CoroutineScope
    try {
        launch {  this: CoroutineScope
            functionThatThrowsException()
        }
    } catch (e: Exception) {
        println("Caught: $e")
    }
}
```



```
Exception in thread "DefaultDispatcher-worker-2" java.lang.RuntimeException
```

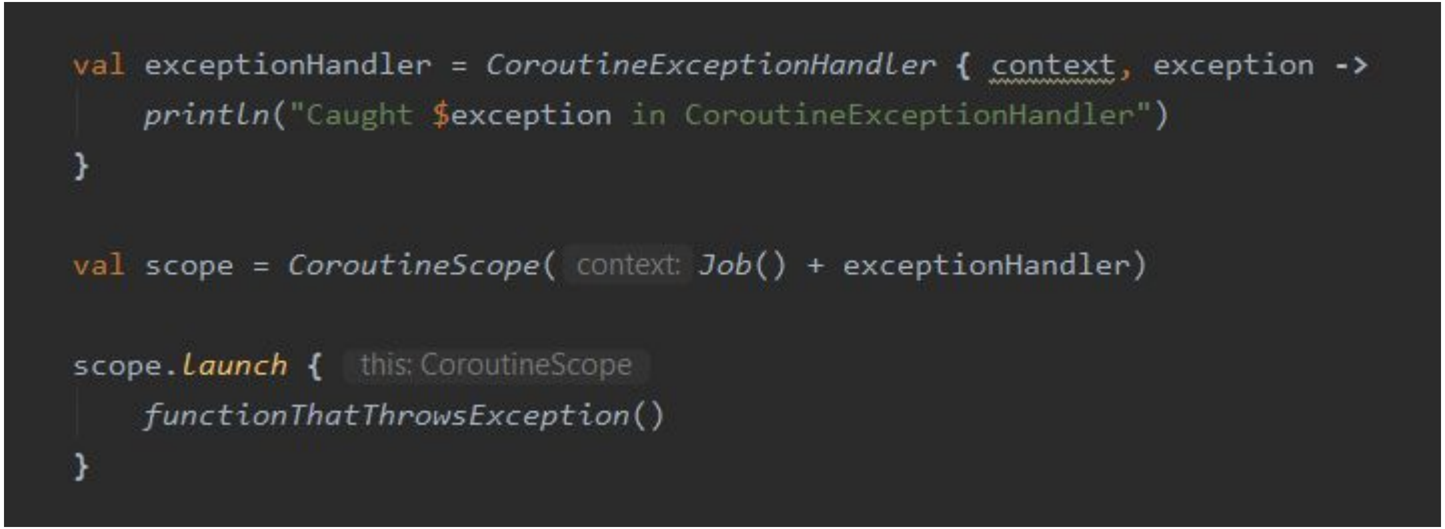
Error handling



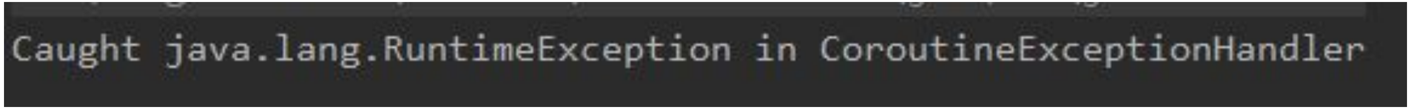
CoroutineExceptionHandler

- An optional element in the coroutine context to handle **uncaught** exceptions.
- All *children* coroutines delegate handling of their exceptions to their parent coroutine, which also delegates to the parent, and so on until the root, so the `CoroutineExceptionHandler` installed in their context is never used.
- Coroutines running with [SupervisorJob] do not propagate exceptions to their parent and are treated like root coroutines.
- A coroutine that was created using [async] always catches all its exceptions and represents them in the resulting [Deferred] object, so it cannot result in uncaught exceptions.
- [CancellationException] will **not** get handled

CoroutineExceptionHandler - 1

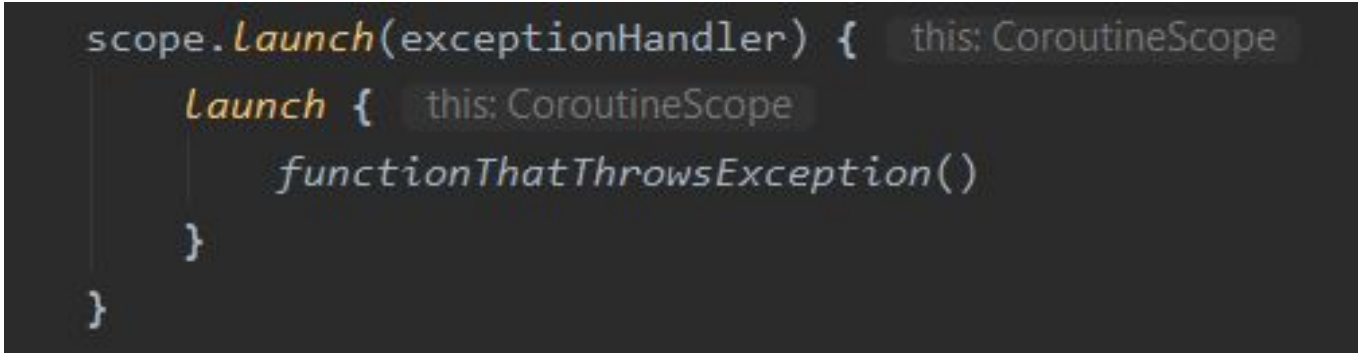


```
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Caught $exception in CoroutineExceptionHandler")
}

val scope = CoroutineScope( context: Job() + exceptionHandler)

scope.launch {  this: CoroutineScope
    functionThatThrowsException()
}
```

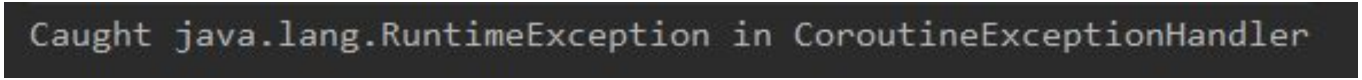


```
Caught java.lang.RuntimeException in CoroutineExceptionHandler
```

CoroutineExceptionHandler - 2.1



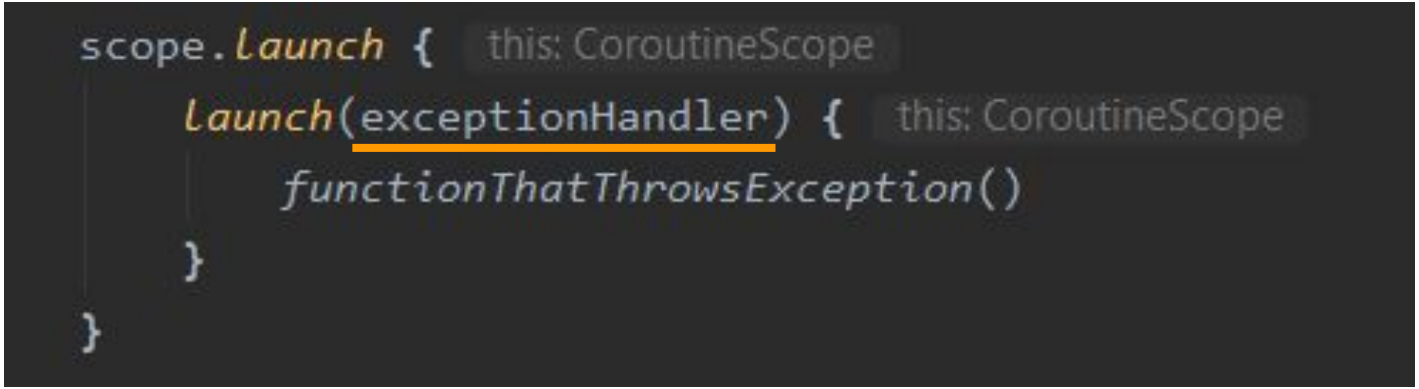
```
scope.launch(exceptionHandler) {  this: CoroutineScope
    launch {  this: CoroutineScope
        functionThatThrowsException()
    }
}
```



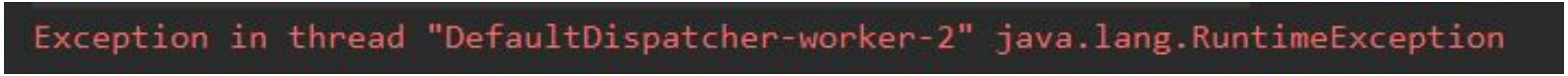
```
Caught java.lang.RuntimeException in CoroutineExceptionHandler
```

Error handling

CoroutineExceptionHandler - 2.2

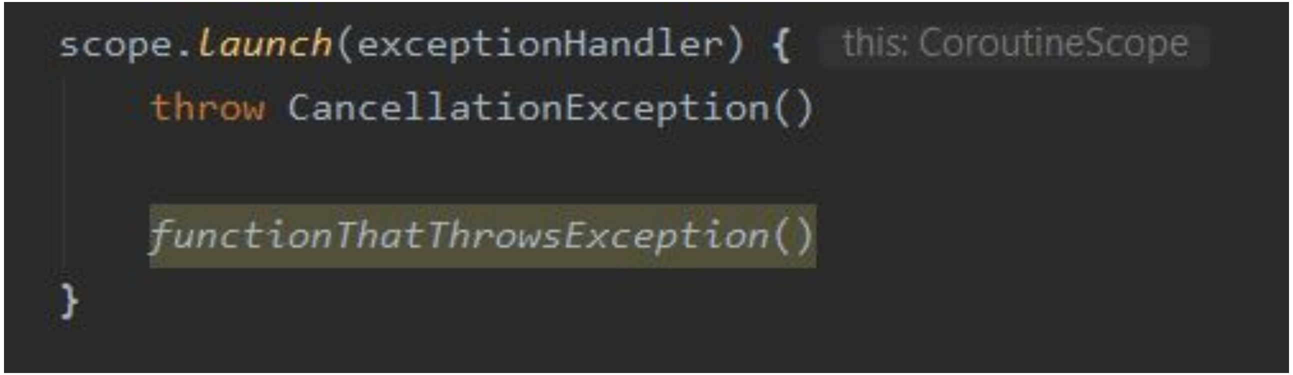


```
scope.launch {  this: CoroutineScope
    launch(exceptionHandler) {  this: CoroutineScope
        functionThatThrowsException()
    }
}
```



```
Exception in thread "DefaultDispatcher-worker-2" java.lang.RuntimeException
```

CoroutineExceptionHandler - 3



```
scope.launch(exceptionHandler) {  this: CoroutineScope
    throw CancellationException()

    functionThatThrowsException()
}
```

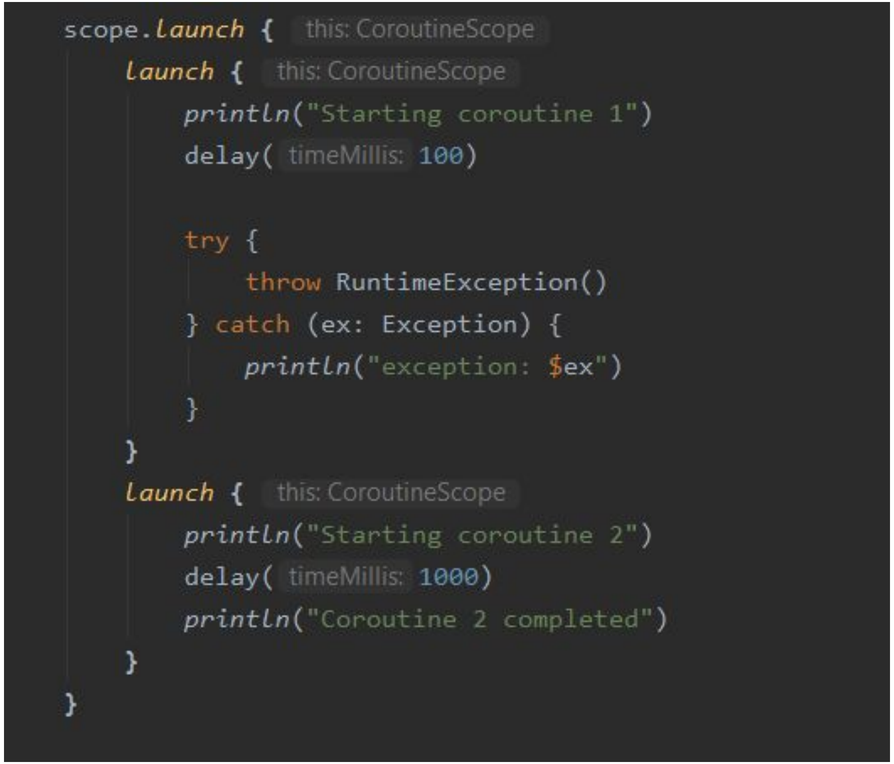
nothing happens

Error handling

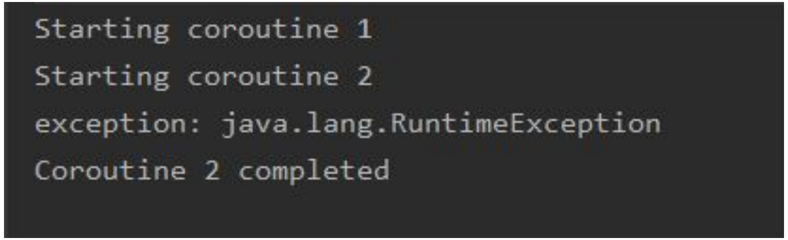
CoroutineExceptionHandler	try-catch{}
last-resort mechanism for global "catch all" behavior. You cannot recover from the exception	You **can** recover from the exception
the handler is used to log the exception, show some kind of error message, terminate, and/or restart the application.	retry operation, handle exceptional behavior
aligned with structured concurrency	can ruin structured concurrency

3_try_catch_vs_exception_handler.kt

CoroutineExceptionHandler vs try-catch



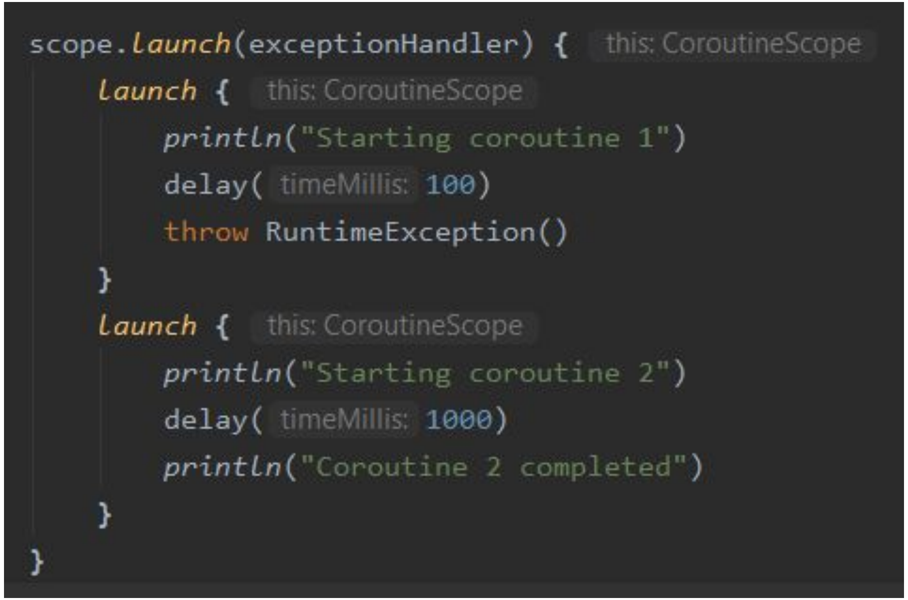
```
scope.launch {  this: CoroutineScope
    launch {  this: CoroutineScope
        println("Starting coroutine 1")
        delay( timeMillis: 100)

        try {
            throw RuntimeException()
        } catch (ex: Exception) {
            println("exception: $ex")
        }
    }
    launch {  this: CoroutineScope
        println("Starting coroutine 2")
        delay( timeMillis: 1000)
        println("Coroutine 2 completed")
    }
}
```



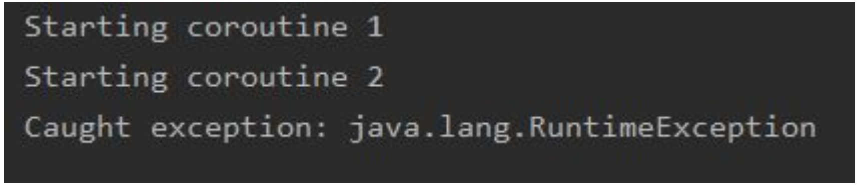
```
Starting coroutine 1
Starting coroutine 2
exception: java.lang.RuntimeException
Coroutine 2 completed
```

Error handling

CoroutineExceptionHandler vs try-catch



```
scope.launch(exceptionHandler) {  this: CoroutineScope
    launch {  this: CoroutineScope
        println("Starting coroutine 1")
        delay( timeMillis: 100)
        throw RuntimeException()
    }
    launch {  this: CoroutineScope
        println("Starting coroutine 2")
        delay( timeMillis: 1000)
        println("Coroutine 2 completed")
    }
}
```



```
Starting coroutine 1
Starting coroutine 2
Caught exception: java.lang.RuntimeException
```

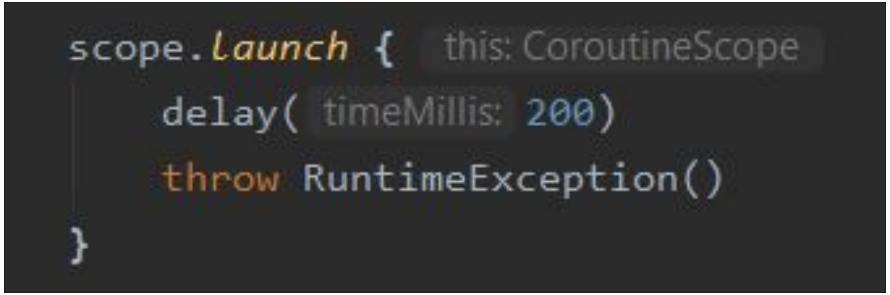
Error handling

async{} vs launch{}

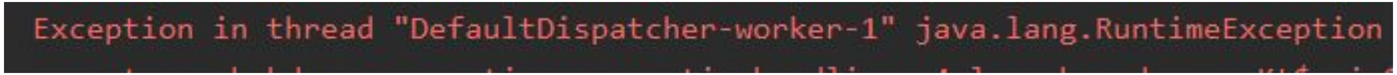
with async{}, the exception is encapsulated in the Deferred object

4_launch_and_async.kt

launch vs async - 1.1

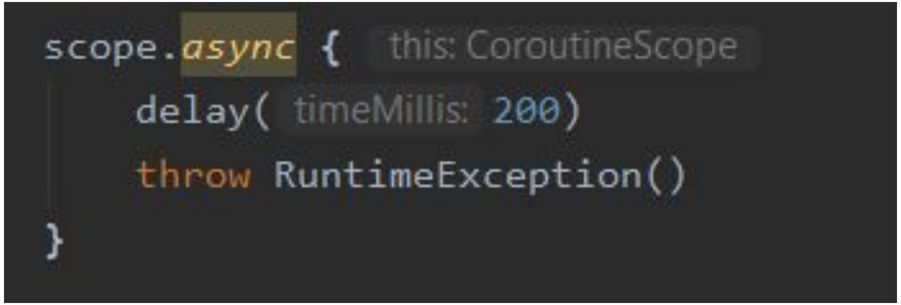


```
scope.launch {  this: CoroutineScope
    delay( timeMillis: 200)
    throw RuntimeException()
}
```



```
Exception in thread "DefaultDispatcher-worker-1" java.lang.RuntimeException
```

launch vs async - 1.2

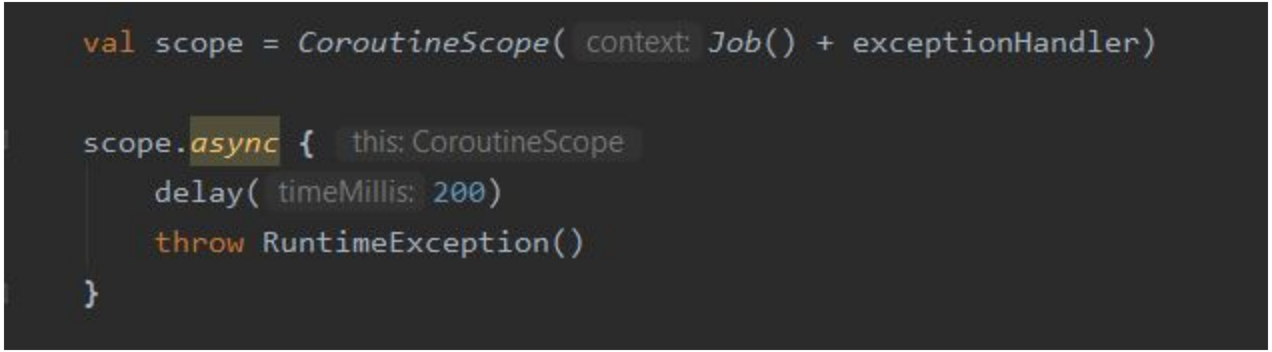


```
scope.async {  this: CoroutineScope
    delay( timeMillis: 200)
    throw RuntimeException()
}
```

nothing happens

Error handling

launch vs async - 1.3

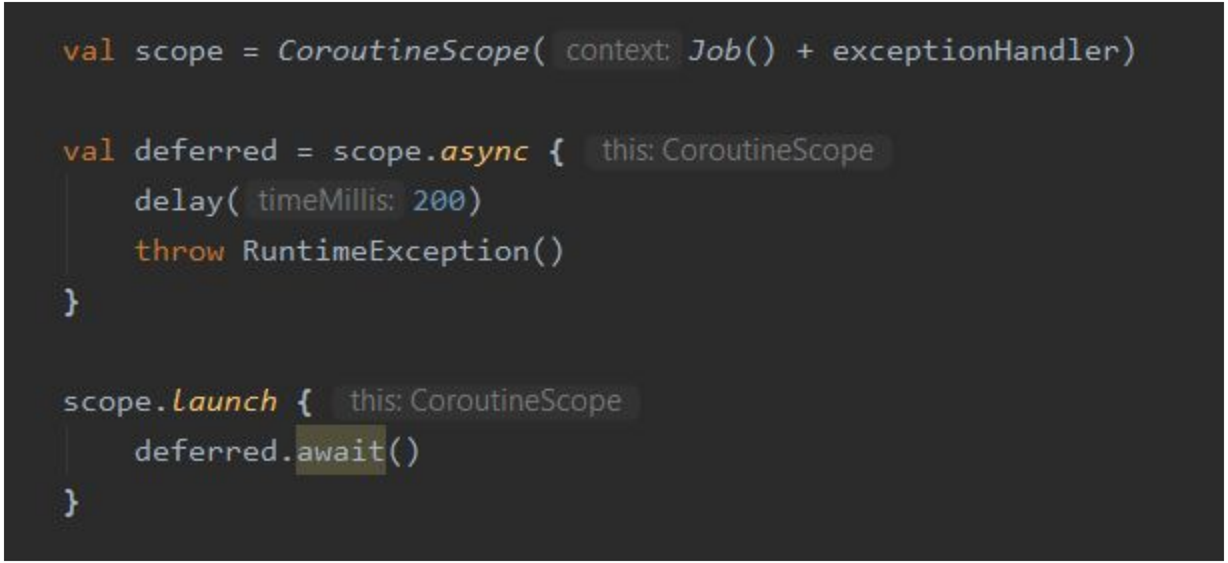


```
val scope = CoroutineScope( context: Job() + exceptionHandler)

scope.async {  this: CoroutineScope
    delay( timeMillis: 200)
    throw RuntimeException()
}
```

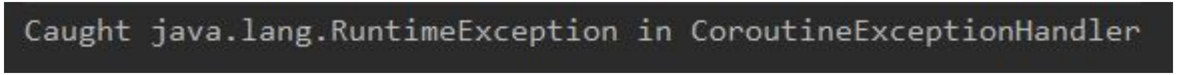
nothing happens

Error handling

launch vs async - 1.4

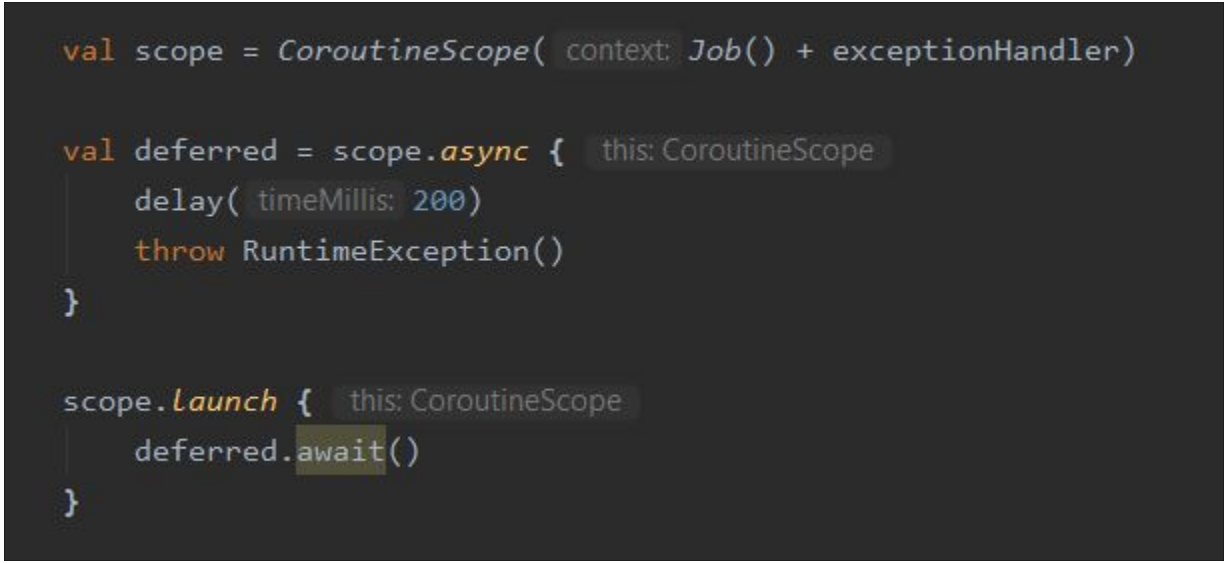


```
val scope = CoroutineScope( context: Job() + exceptionHandler)

val deferred = scope.async {  this: CoroutineScope
    delay( timeMillis: 200)
    throw RuntimeException()
}

scope.launch {  this: CoroutineScope
    deferred.await()
}
```

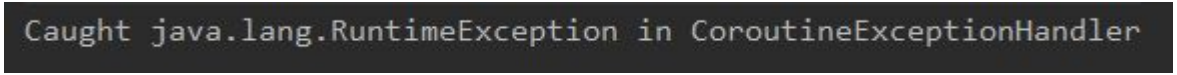


```
Caught java.lang.RuntimeException in CoroutineExceptionHandler
```

launch vs async - 1.4



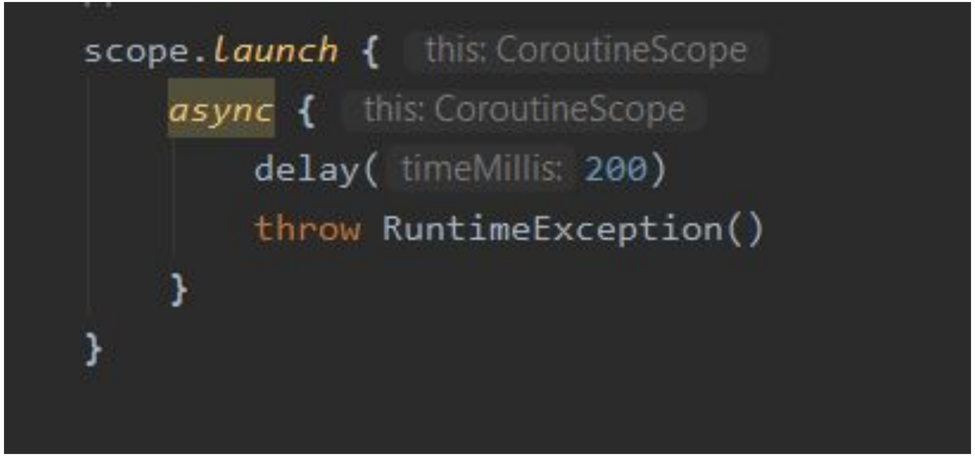
```
val scope = CoroutineScope( context: Job() + exceptionHandler)

val deferred = scope.async {  this: CoroutineScope
    delay( timeMillis: 200)
    throw RuntimeException()
}

scope.launch {  this: CoroutineScope
    deferred.await()
}
```



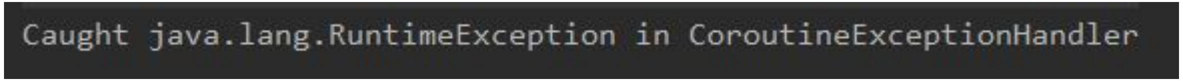
```
Caught java.lang.RuntimeException in CoroutineExceptionHandler
```

Error handling

launch vs async - 2.1



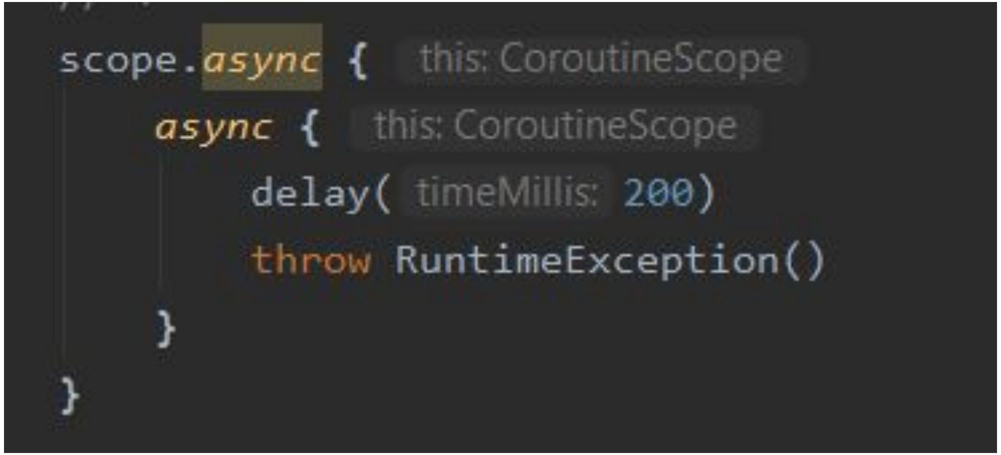
```
scope.launch {  this: CoroutineScope
    async {  this: CoroutineScope
        delay( timeMillis: 200)
        throw RuntimeException()
    }
}
```



```
Caught java.lang.RuntimeException in CoroutineExceptionHandler
```

Error handling

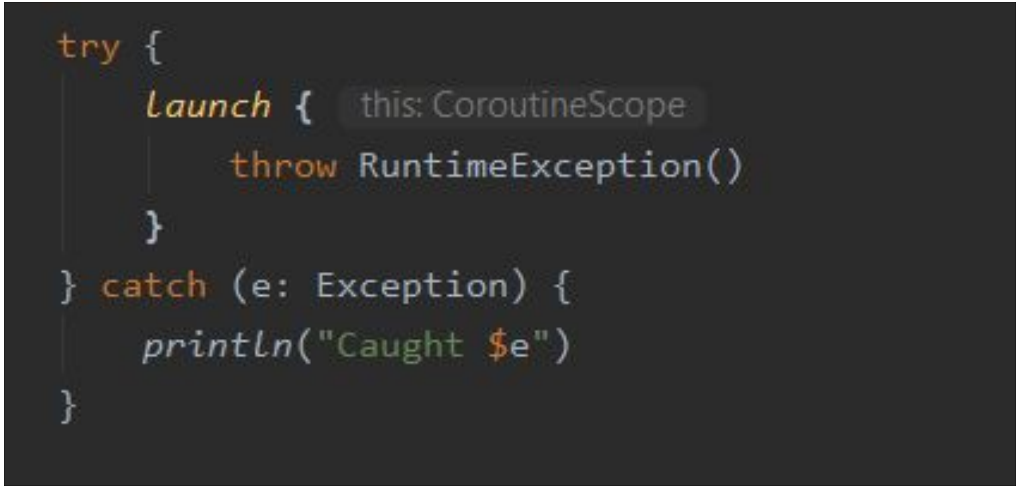
launch vs async - 2.2



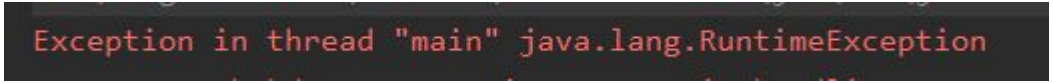
```
scope.async {  this: CoroutineScope
    async {  this: CoroutineScope
        delay( timeMillis: 200)
        throw RuntimeException()
    }
}
```

nothing happens

coroutineScope exception handling - 1.1

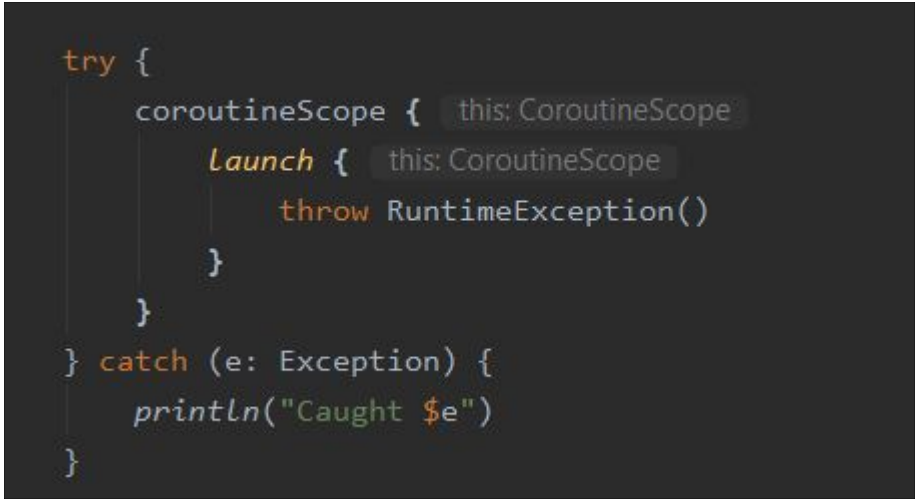


```
try {
    launch {  this: CoroutineScope
        throw RuntimeException()
    }
} catch (e: Exception) {
    println("Caught $e")
}
```

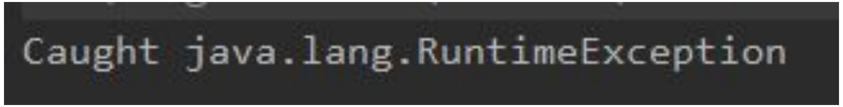


```
Exception in thread "main" java.lang.RuntimeException
```

coroutineScope exception handling - 1.2



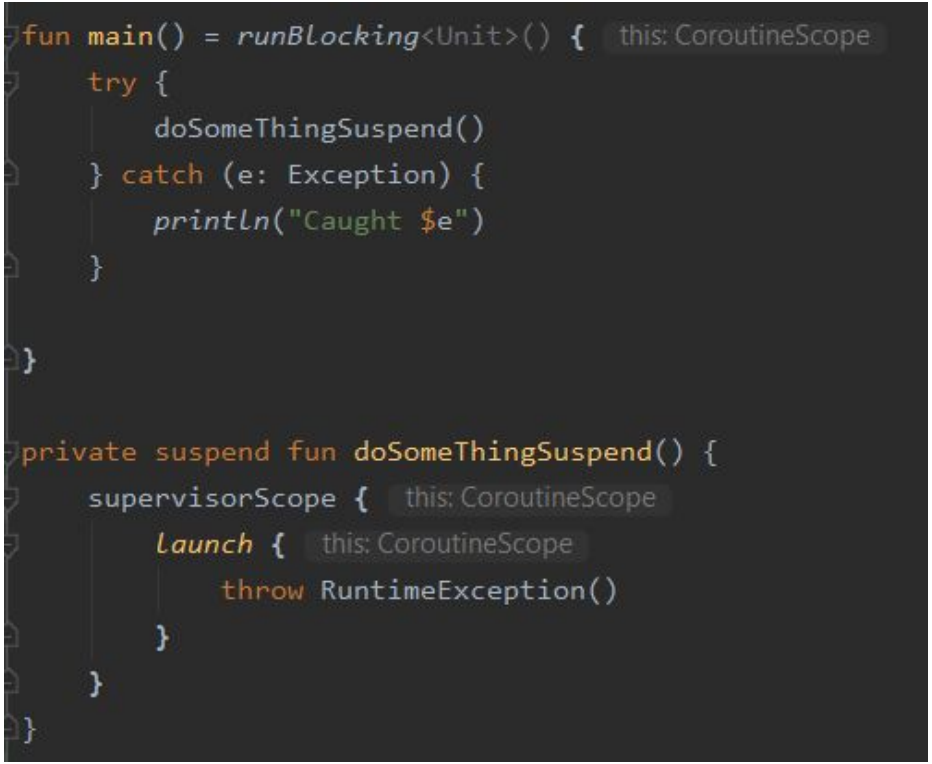
```
try {
    coroutineScope {  this: CoroutineScope
        launch {  this: CoroutineScope
            throw RuntimeException()
        }
    }
} catch (e: Exception) {
    println("Caught $e")
}
```



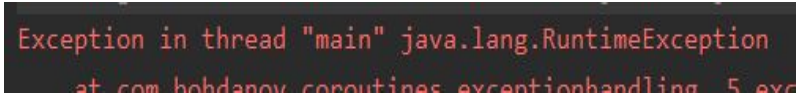
```
Caught java.lang.RuntimeException
```

Error handling

supervisorScope{} exception handling - 1.1

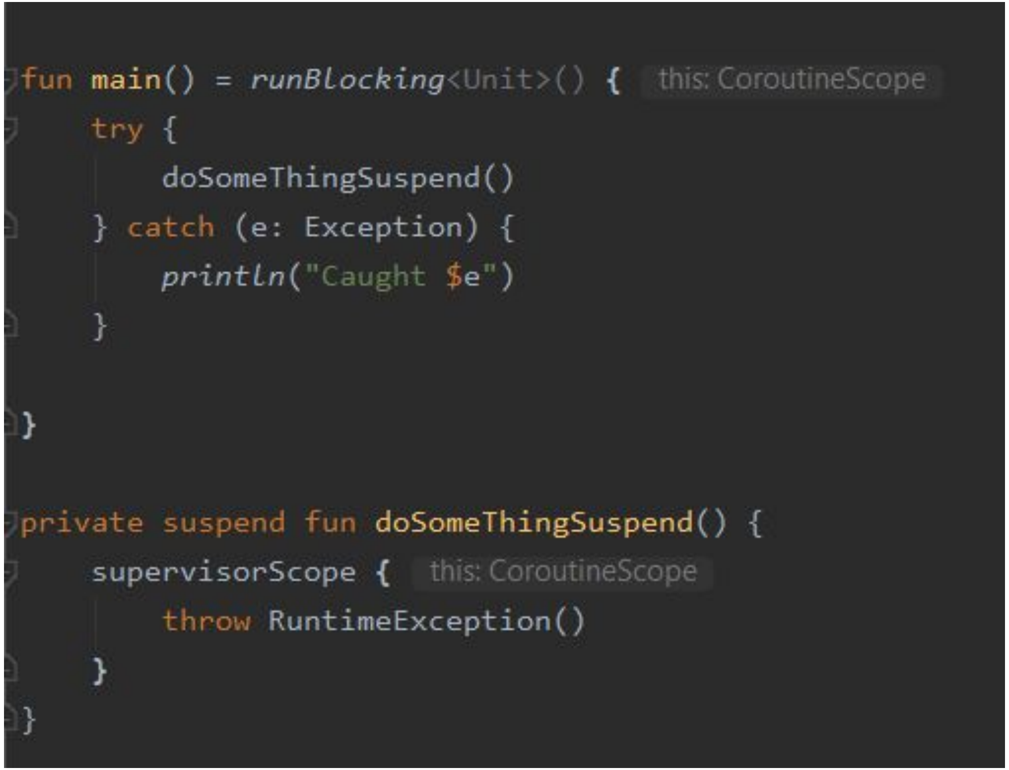


```
fun main() = runBlocking<Unit>() {  this: CoroutineScope
    try {
        doSomeThingSuspend()
    } catch (e: Exception) {
        println("Caught $e")
    }

}

private suspend fun doSomeThingSuspend() {
    supervisorScope {  this: CoroutineScope
        launch {  this: CoroutineScope
            throw RuntimeException()
        }
    }
}
```

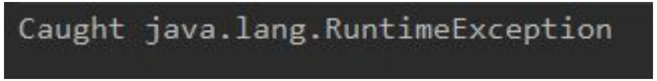


```
Exception in thread "main" java.lang.RuntimeException
    at com.bohdanov.coroutines.exceptionhandling._5.exc
```

supervisorScope{} exception handling - 1.2

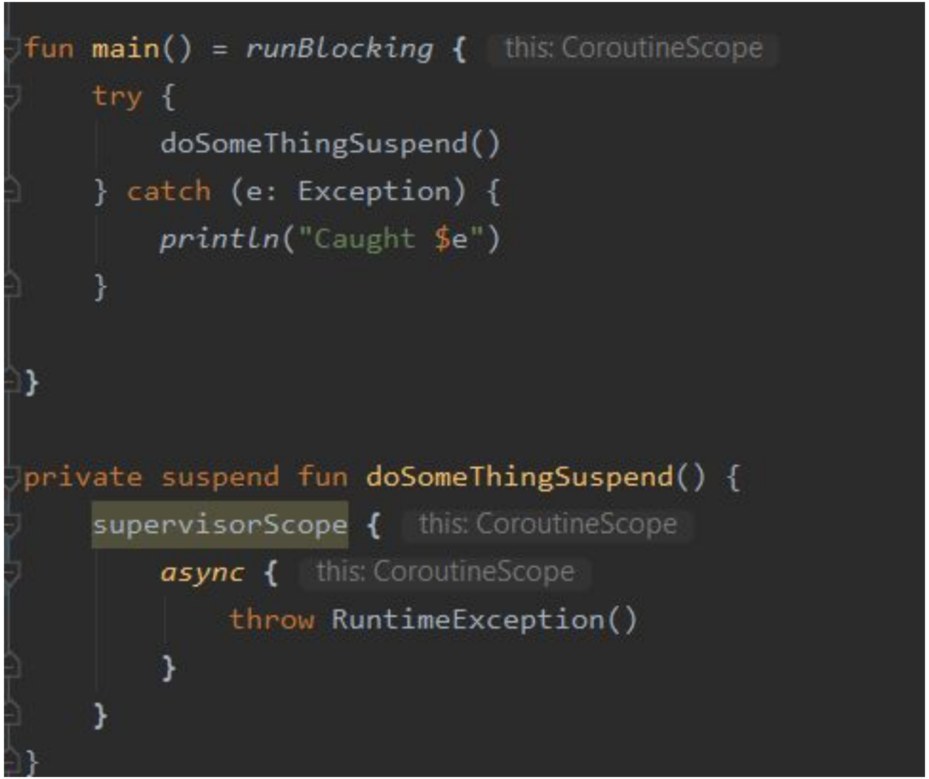


```kotlin
fun main() = runBlocking<Unit>() {
    try {
        doSomeThingSuspend()
    } catch (e: Exception) {
        println("Caught $e")
    }

}

private suspend fun doSomeThingSuspend() {
    supervisorScope {
        throw RuntimeException()
    }
}
```



```
Caught java.lang.RuntimeException
```

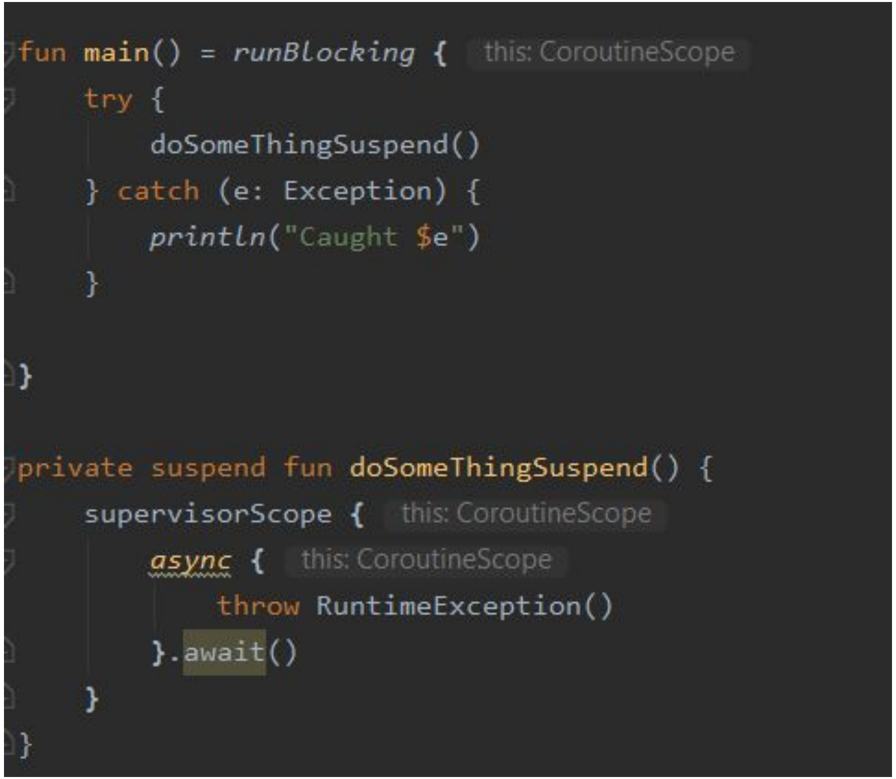
supervisorScope{} exception handling - 1.3



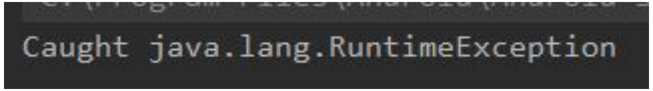
```
fun main() = runBlocking {  this: CoroutineScope
    try {
        doSomeThingSuspend()
    } catch (e: Exception) {
        println("Caught $e")
    }

}

private suspend fun doSomeThingSuspend() {
    supervisorScope {  this: CoroutineScope
        async {  this: CoroutineScope
            throw RuntimeException()
        }
    }
}
```

nothing happens

supervisorScope{} exception handling - 1.4



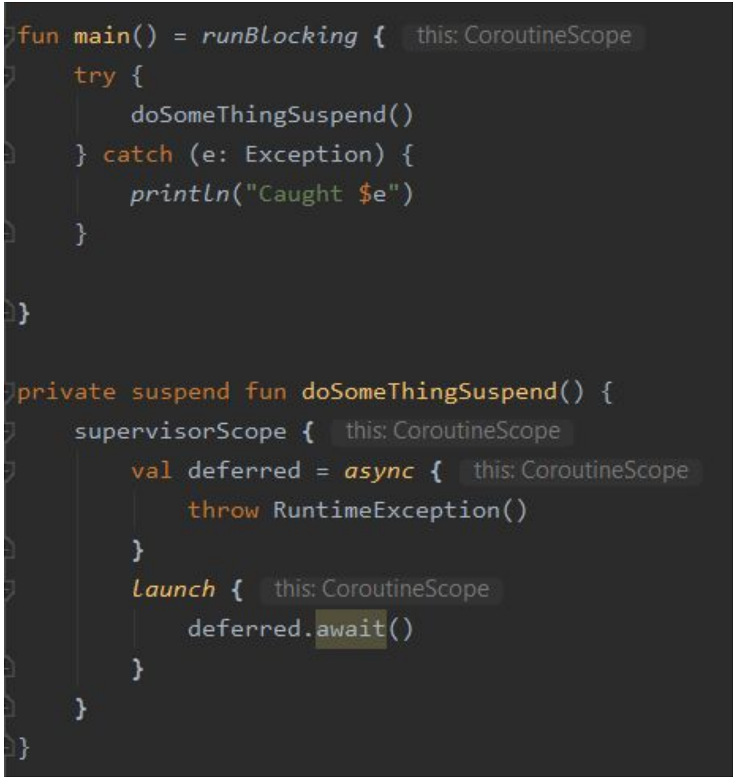
```
fun main() = runBlocking {
    try {
        doSomeThingSuspend()
    } catch (e: Exception) {
        println("Caught $e")
    }

}

private suspend fun doSomeThingSuspend() {
    supervisorScope {
        async {
            throw RuntimeException()
        }.await()
    }
}
```



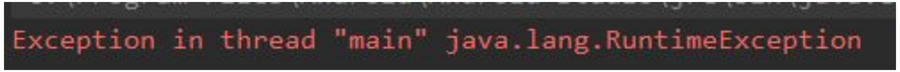
```
Caught java.lang.RuntimeException
```

Error handling

supervisorScope{} exception handling - 1.6



```kotlin
fun main() = runBlocking {
    try {
        doSomeThingSuspend()
    } catch (e: Exception) {
        println("Caught $e")
    }

}

private suspend fun doSomeThingSuspend() {
    supervisorScope {
        val deferred = async {
            throw RuntimeException()
        }
        launch {
            deferred.await()
        }
    }
}
```



```
Exception in thread "main" java.lang.RuntimeException
```

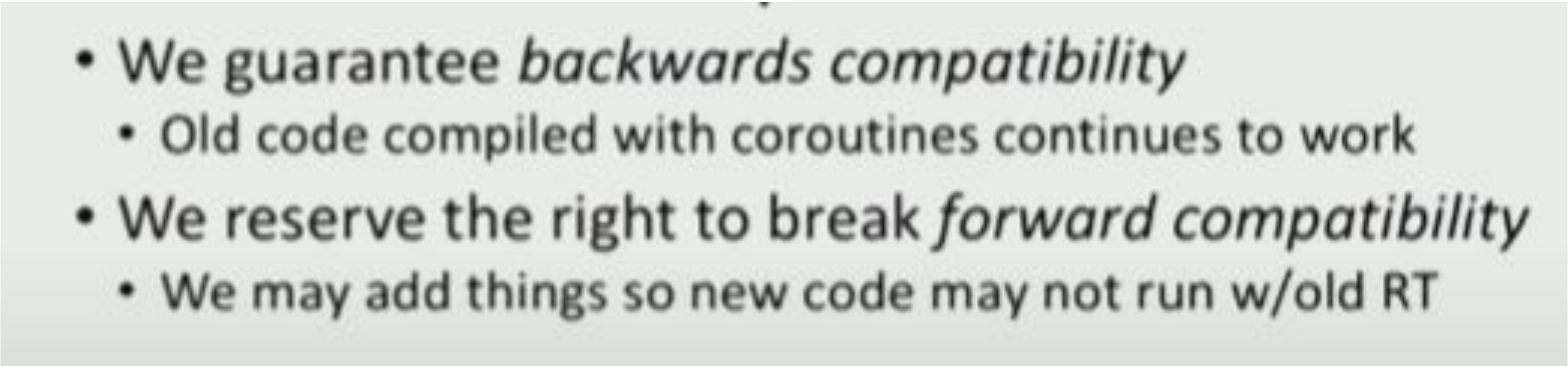
Some tips

tips

What does “Experimental” mean?

it is **not** alpha or beta

Can I use it in production? Yes! you should



- We guarantee *backwards compatibility*
  - Old code compiled with coroutines continues to work
- We reserve the right to break *forward compatibility*
  - We may add things so new code may not run w/old RT

tips

Concurrency have to be explicit

tips

Can we write our own launch{}? Yes.

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

tips

withTimeout(){}

with_timeout.kt

tips

repeat + withTimeout

repeat.kt

tips

How to remove callbacks

remove_callbacks_init.kt

remove_callbacks_with_coroutines.kt

links

Mastering Kotlin Coroutines
https://www.udemy.com/share/103K0YAEcbdFxURnkF/

Корутины в Kotlin — Роман Елизаров
https://www.youtube.com/watch?v=b4mBmi1QNF0

Kotlin Coroutines Exception Handling Cheat Sheet
https://www.lukaslechner.com/coroutines-exception-handling-cheat-sheet/

Deep Dive into Coroutines on JVM by Roman Elizarov:
https://www.youtube.com/watch?v=YrrUCSi72E8&t

links

Codelab: Use Kotlin Coroutines in your Android App
https://codelabs.developers.google.com/codelabs/kotlin-coroutines/#4

Suspending over views
https://medium.com/androiddevelopers/suspending-over-views-19de9ebd7020

kotlinx.coroutines by example
https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md

Thanks :)