Part II

Agenda

- Deep dive into coroutines
- Coroutine Cancellation
- Coroutine Exception Handling
- Structured Concurrency

Main Building Blocks of Coroutines

- CoroutineScope
- CoroutineContext
  - Job
  - Coroutine Dispatchers
  - CoroutineName
  - CoroutineExceptionHandler
- CoroutineBuilder

What's the difference?

Coroutines …

- A **coroutine** is an instance of a *suspendable* computation.
  - Computations can be *suspended* *without blocking the thread* at suspending points, and can later be resumed.
- Empowered `Runnable`
  - A coroutine can be introduced as a sequence of well managed sub-tasks, each of which can be suspended and resumed.

Coroutines …

- To some extent, coroutines can be thought of as *light-weight threads*;
  - Executed within threads
  - Can switch contexts

How many threads we can have?

100 ☺

1000 😅

10 000 😩

100 000 😵

Threads vs. Coroutines

```
fun main() {
  repeat(200_000) {
    thread {
      println("Hello thread $it")
    }
  }
}
```

Supposed to crash, but …

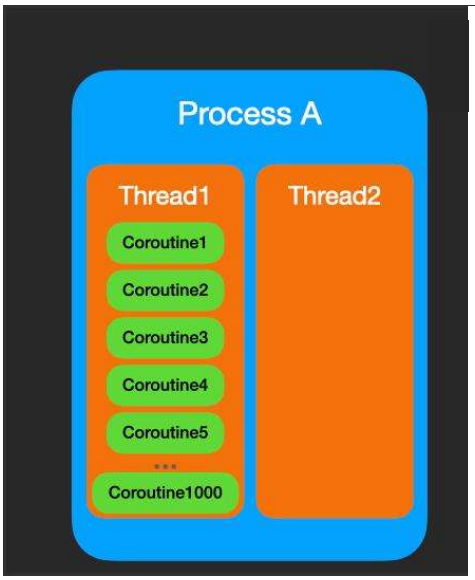
136615 ms

```
fun main() = runBlocking{
  repeat(200_000) {
    launch {
      println("Hello coroutine $it")
    }
  }
}
```

It is not just Ok, but it also very convenient to create coroutines as you need them, since they are so cheap.

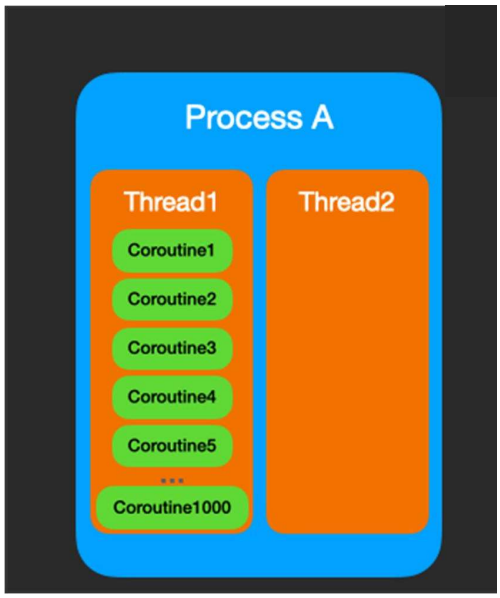
437 ms

Coroutines …



You can execute many coroutines in a single thread.

Coroutines …



A coroutine can switch between threads.

A coroutine can suspend from one thread and resume from another thread.

Coroutines in details

- Based on the abstraction of *Continuation Passing Style (CPS)*
- Actually, it is a sequence of *callbacks* behind the scenes.
- In kotlin, coroutine suspension/resume is implemented as a *state machine*.

Continuation Passing Style (CPS)

```
fun add(a: Int, b: Int): Int = a + b
fun mult(a: Int, b: Int): Int = a * b
```

```
fun <R> addCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(add(a, b))
}
fun <R> multCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(mult(a, b))
}
```

```
// (3 + 4) * (5 + 6)
fun doWork(): Int {
    // label1
    val step1 = add(3, 4)
    // label2
    val step2 = add(5, 6)
    // label3
    val step3 = mult(step1, step2)
    return step3
}
```

```
fun doWorkCPS(): Int =
    addCPS(3, 4) { step1 ->
        addCPS(5, 6) { step2 ->
            multCPS(step1, step2) { step3 ->
                step3
            }
        }
    }
```

Continuation Passing Style (CPS)

```
fun add(a: Int, b: Int): Int = a + b
fun mult(a: Int, b: Int): Int = a * b

fun <R> addCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(add(a, b))
}
fun <R> multCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(mult(a, b))
}
```

```
// (3 + 4) * (5 + 6)
fun doWork(): Int {
    // label1
    val step1 = add(3, 4)
    // label2
    val step2 = add(5, 6)
    // label3
    val step3 = mult(step1, step2)
    return step3
}
```

```
fun doWorkCPS(): Int =
    addCPS(3, 4) { step1 ->
        addCPS(5, 6) { step2 ->
            multCPS(step1, step2) { step3 ->
                step3
            }
        }
    }
```


Continuation Passing Style (CPS)

```
fun add(a: Int, b: Int): Int = a + b
fun mult(a: Int, b: Int): Int = a * b

fun <R> addCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(add(a, b))
}
fun <R> multCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(mult(a, b))
}
```

```
// (3 + 4) * (5 + 6)
fun doWork(): Int {
    // label1
    val step1 = add(3, 4)
    // label2
    val step2 = add(5, 6)
    // label3
    val step3 = mult(step1, step2)
    return step3
}
```

```
fun doWorkCPS(): Int =
    addCPS(3, 4) { step1 ->
        addCPS(5, 6) { step2 ->
            multCPS(step1, step2) { step3 ->
                step3
            }
        }
    }
```

Continuation Passing Style (CPS)

```
fun add(a: Int, b: Int): Int = a + b
fun mult(a: Int, b: Int): Int = a * b

fun <R> addCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(add(a, b))
}
fun <R> multCPS(a: Int, b: Int, cont: (Int) -> R): R {
    return cont(mult(a, b))
}
```

```
// (3 + 4) * (5 + 6)
fun doWork(): Int {
    // label1
    val step1 = add(3, 4)
    // label2
    val step2 = add(5, 6)
    // label3
    val step3 = mult(step1, step2)
    return step3
}
```

```
fun doWorkCPS(): Int =
    addCPS(3, 4) { step1 ->
        addCPS(5, 6) { step2 ->
            multCPS(step1, step2) { step3 ->
                step3
            }
        }
    }
```


Continuation

The Kotlin compiler converts the suspend function to an optimized version of **callbacks** using a **Finite State Machine**.

kotlin

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

Java/JVM

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

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

Continuation is a generic callback interface

The way suspend functions communicate with each other

```
inline fun <T> Continuation<T>.resume(value: T): Unit =
    resumeWith(Result.success(value))

inline fun <T> Continuation<T>.resumeWithException(exception: Throwable): Unit =
    resumeWith(Result.failure(exception))
```

Convert to CPS Style

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

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


CPS Transform

- The compiler replaces the `suspend` modifier with the extra parameter `completion` (of type `Continuation`) that will be used to communicate the result of the suspend function to the coroutine that called it.

suspension points

```
fun postItem(item: Item, completion: Continuation<Unit>): Any {
    val token = requestToken()
    val post = createPost(token, item)
    completion.resume(showPost(post))
}
```

`Unit | COROUTINE_SUSPENDED`

CPS Transform

- Every *suspension point* will be represented as a *state* in the finite state machine. These states are represented with *labels* by the compiler.

```
fun postItem(item: Item, completion: Continuation<Any?>) {
    // Label 0 -> first execution
    val token = requestToken()
    // Label 1 -> resumes from requestToken
    val post = createPost(token, item)
    // Label 2 -> resumes from createPost
    completion.resume(showPost(post))
}
```


CPS Transform

```
fun postItem(item: Item, completion: Continuation<Any?>) {
    when(label) {
        0 -> { // Label 0 -> first execution
            requestToken()
        }
        1 -> { // Label 1 -> resumes from requestToken
            createPost(token, item)
        }
        2 -> { // Label 2 -> resumes from createPost
            completion.resume(showPost(post))
        }
        else -> throw IllegalStateException(...)
    }
}
```

The compiler will use the same `Continuation` object in the function to share information between states.

CPS Transform

The compiler will create a private class that
1) holds the required data and
2) calls the `postItem` recursively to resume execution.

```
fun postItem(item: Item?, completion: Continuation<Any?>) {
    class PostItemStateMachine(
        completion: Continuation<Any?> // callback to the fun that called postItem
    ): CoroutineImpl(completion) {
        // Local variables of the suspend function
        var token: Token? = null
        var post: Post? = null
        // Common objects for all CoroutineImpls
        var result: Any? = null
        var label: Int = 0
        // this function calls the `postItem` again to trigger the
        // state machine (label will be already in the next state)
        override fun invokeSuspend(result: Any?) {
            this.result = result // result of the previous state's computation
            postItem(null, this)
        }
    }
    ...
}
```


Check if
1) it's the first time called or
2) resumed from a previous state.

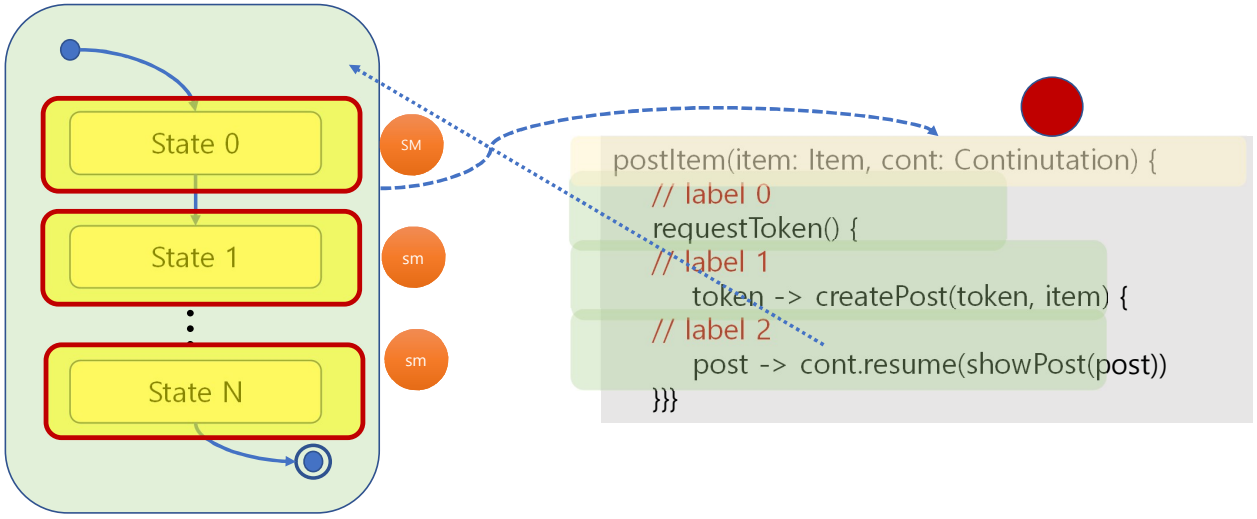
```
fun postItem(item: Item?, completion: Continuation<Any?>) {
    ...
    val continuation = completion as? PostItemStateMachine
                                   ?: PostIemStateMachine(completion)

    when(continuation.label) {
        0 -> {
            throwOnFailure(continuation.result) // Checks for failures
            // next time this coroutine is called, it should go to state 1
            continuation.label = 1
            // The continuation object is passed to requestToken to resume
            // this state machine's execution when it finishes
            requestToken(continuation)
        }
        1 -> {
            throwOnFailure(continuation.result)
            // Gets the result of the previous state
            continuation.token = continuation.result as Token
            continuation.label = 2
            createPost(continuation.token, item, continuation)
        }
            ... // leaving out the last state on purpose
    }
}
```

```
fun postItem(item: Item?, completion: Continuation<Any?>) {
    ...
    when(continuation.label) {
        ...
        2 -> {
            // Checks for failures
            throwOnFailure(continuation.result)
            // Gets the result of the previous state
            continuation.post = continuation.result as Post
            // Resumes the execution of the function that called this one
            continuation.cont.resume(showPost(continuation.post))
        }
        else -> throw IllegalStateException(...)
    }
}
```

The last state is different since it has to resume the execution of the function that called this one, it calls resume on the `cont` variable stored (at construction time) in `PostItemStateMachine`:

How Suspending Functions Evaluated?

State Machine (= Continuation)



CoroutineContext *vs.* CoroutineScope

https://elizarov.medium.com/coroutine-context-and-scope-c8b255d59055

- *Different uses of physically **near-identical things** are usually accompanied by giving those things **different names** to **emphasize the intended purpose**.*
- *Depending on the use, seamen have a dozen or more words for a rope though it might materially be the same thing. (Wikipedia on Hindley-Milner type system)*

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

CoroutineContext

- Every coroutine has a *coroutine context* which is *immutable*.
  - `CoroutineContext` can be inherited from parent to child.
- Accessible via `coroutineContext` property:

```
fun main() = runBlocking {
    println(Thread.currentThread().name)
    println("$coroutineContext")
    println("${coroutineContext[Job]}")
    println("${coroutineContext[ContinuationInterceptor]}")
}
```

```
main
[BlockingCoroutine{Active}@335eadca, BlockingEventLoop@210366b4]
BlockingCoroutine{Active}@335eadca
BlockingEventLoop@210366b4
```

Elements of CoroutineContext

- The `CoroutineContext` is an indexed set of elements (set + map) that define the behavior of a coroutine:
  - `Job`: controls the **lifecycle** of the coroutine.
  - `CoroutineDispatcher`: **dispatches** work to the appropriate thread.
  - `CoroutineName`: **name** of the coroutine, useful for debugging.
  - `CoroutineExceptionHandler`: handles **uncaught exceptions**.
- Each `Element` is a singleton context by itself.
- Each `Element` can be combined with plus('+') operator.

```
launch(Dispachers.IO + CoroutineName("test")) {
 …
}
```


Job

- A coroutine itself is represented by a `Job` .
- Responsible for *coroutine's lifecycle*, *cancellation*, and *parent-child relations*. A current job can be retrieved from a current coroutine's context:

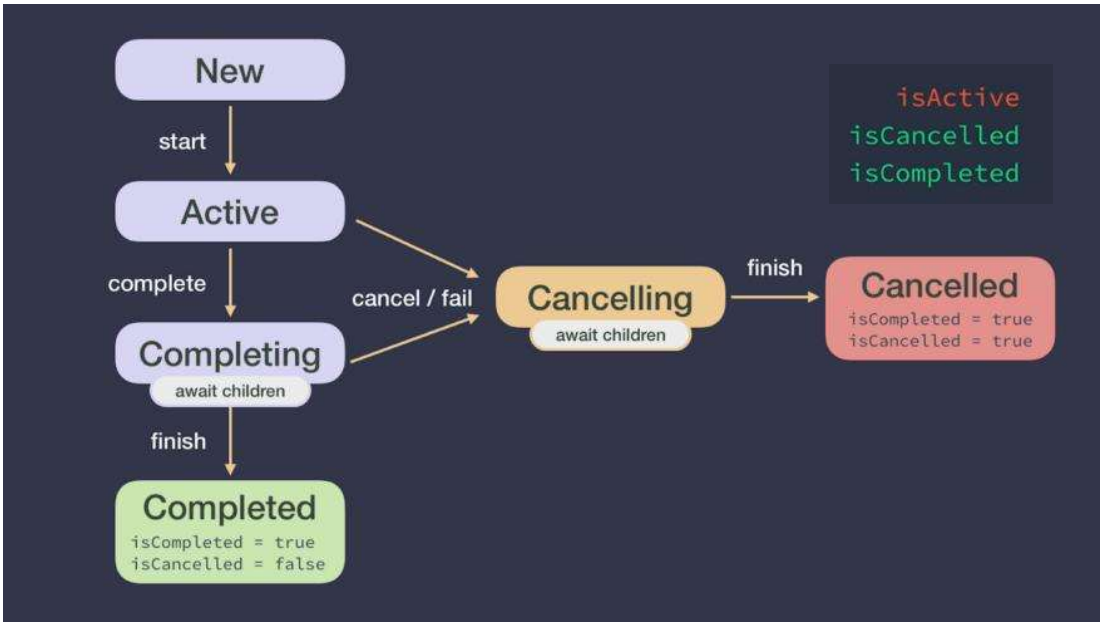
```
Println("My job is: ${coroutineContext[Job]}")
```

- Coroutine builders (`launch` or `async`) returns a `Job` instance that uniquely identifies the coroutine.
- You can also pass a `Job` to a `CoroutineScope` to keep a handle on its lifecycle. Otherwise, `default` Job created.

```
val scope = CoroutineScope(Job())
```

- `SupervisorJob` is a special kind of `Job` (*later!*)

Job lifecycle



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`

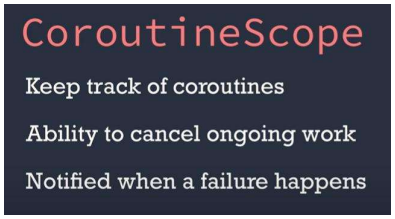
CoroutineScope

- Every coroutine must be created inside the **coroutine scope** to control lifecycle.

```
interface CoroutineScope {
    // The context of this scope.
    public val coroutineContext: CoroutineContext
}
```

- A `CoroutineScope` keeps track of any coroutine created using `launch` or `async`.
- Coroutines can be canceled by calling `scope.cancel()` at any time.
- Predefined scopes in Android: `viewModelScope` and `lifecycleScope`

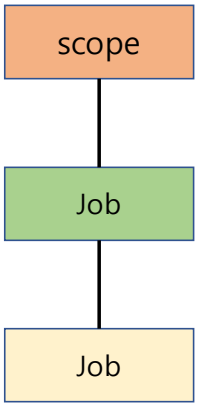
```
                                  CoroutineContext
                                  ┌──────────────┐
val scope = CoroutineScope(Job() + Dispatchers.Main)
val job = scope.launch {
    // new coroutine
}
```



Task Hierarchy

- Since a `CoroutineScope` can create coroutines and you can create more coroutines inside a coroutine, an implicit task hierarchy is created.

```
val scope = CoroutineScope(Job() + Dispatchers.Main)
val job = scope.launch {
    // New coroutine with CoroutineScope as a parent
    val result = async {
        // New coroutine that has the coroutine
        // started by launch as a parent
    }.await()
}
```



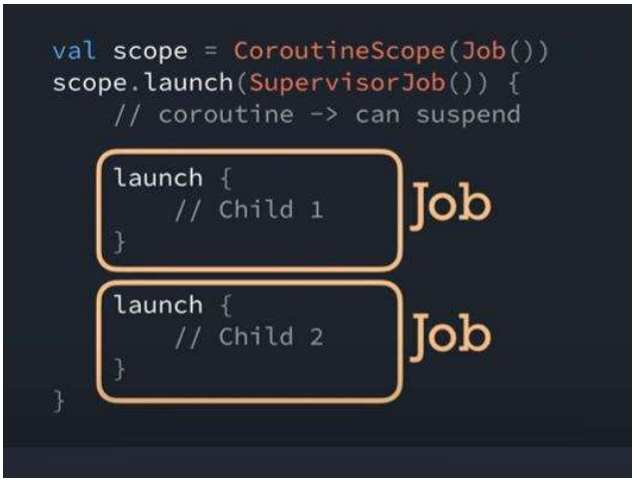
What's the CoroutineContext of a new coroutine?

- Whenever a new coroutine is created using `launch` or `async`, a ***new instance*** of `Job` will be created, allowing us to control its lifecycle.
- The rest of the elements will be ***inherited from*** the `CoroutineContext` of its parent.

Watch out quiz! Who's my parent? 🎯

Given the following code snippet, can you identify what kind of `Job` "child 1" has as a parent?

`Job` or `SupervisorJob`?



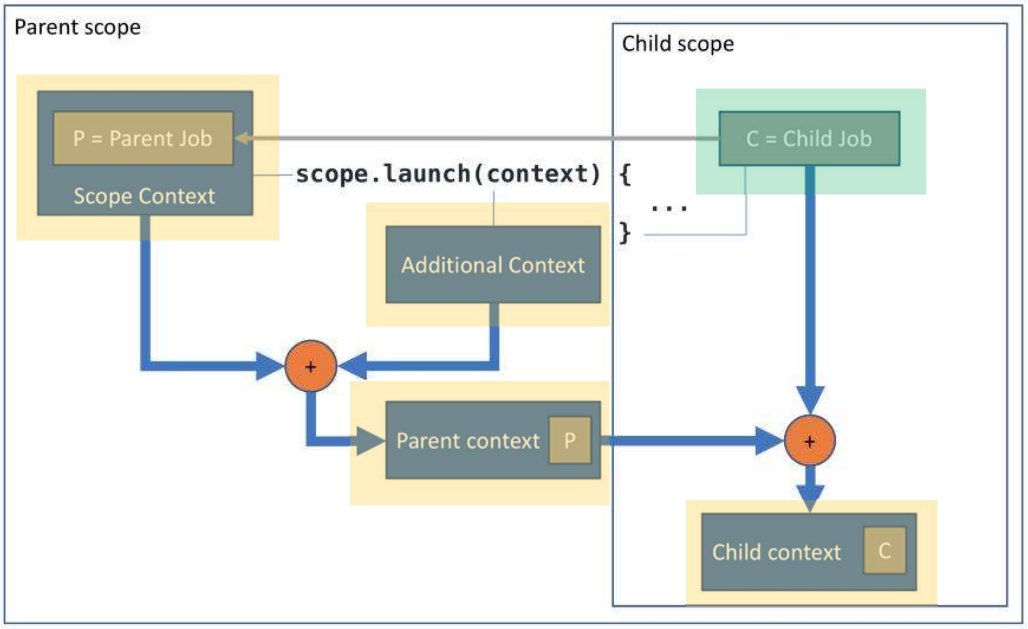
Watch out quiz! Who's my parent? 🎯

Given the following snippet of code, can you identify what kind of `Job` "child 1" has as a parent?

`Job` or `SupervisorJob`?

```
val scope = CoroutineScope(Job())
scope.launch(SupervisorJob()) {
    // coroutine -> can suspend
    launch {
        // Child 1
    }
    launch {
        // Child 2
    }
}
```


Parent Scope vs Child Scope



Parent CoroutineContext explained

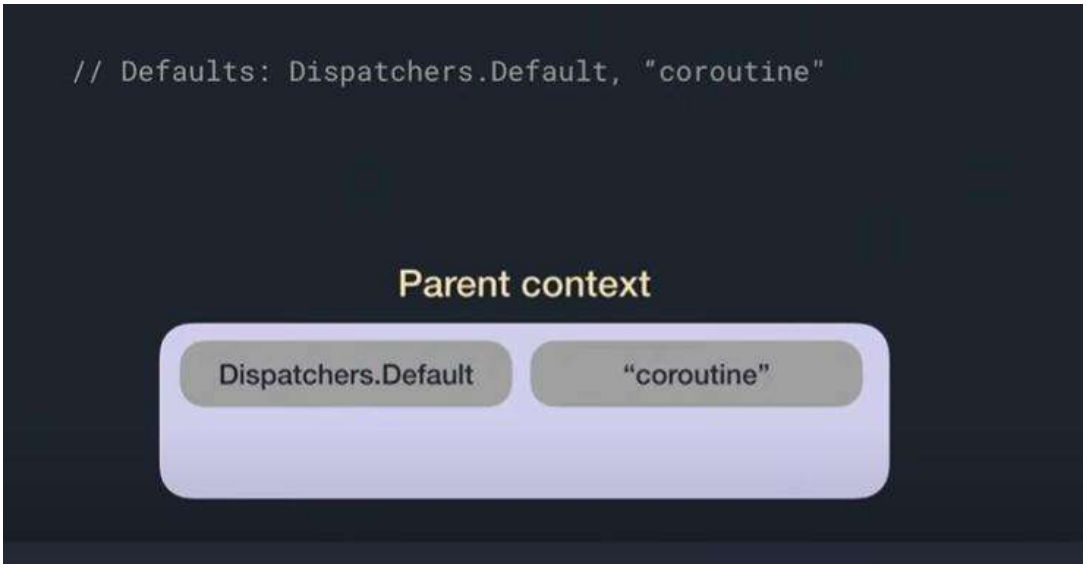
- Child's parent `CoroutineContext` can be different from that of the parent:

> ***Parent context*** = *Defaults* + *inherited* `CoroutineContext` + *arguments*

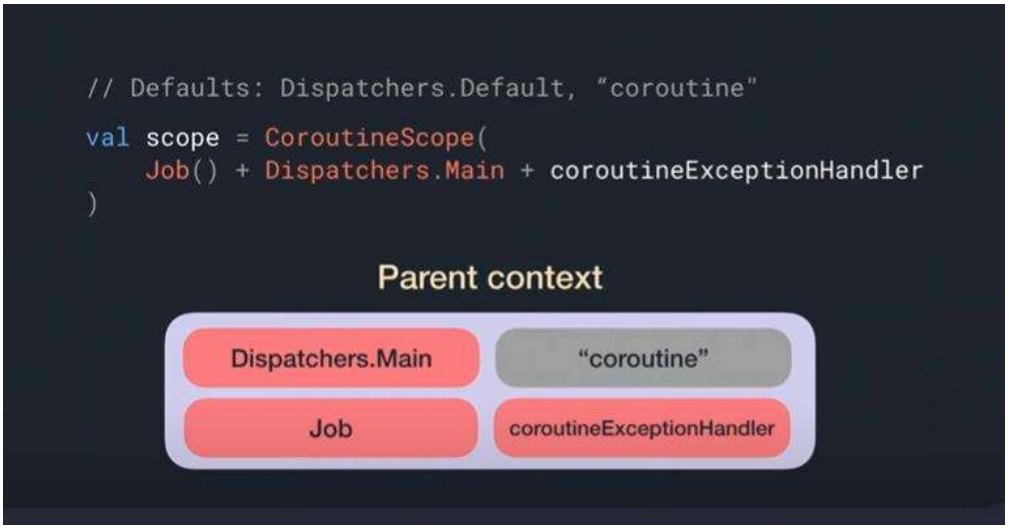
Where:

- Some elements have **default** values: `Dispatchers.Default` is the default of `CoroutineDispatcher` and `"coroutine"` the default of `CoroutineName`.
- The **inherited** **`CoroutineContext`** is the `CoroutineContext` of the `CoroutineScope` or coroutine that created it.
- **Arguments** passed in the coroutine builder will take precedence over those elements in the inherited context.

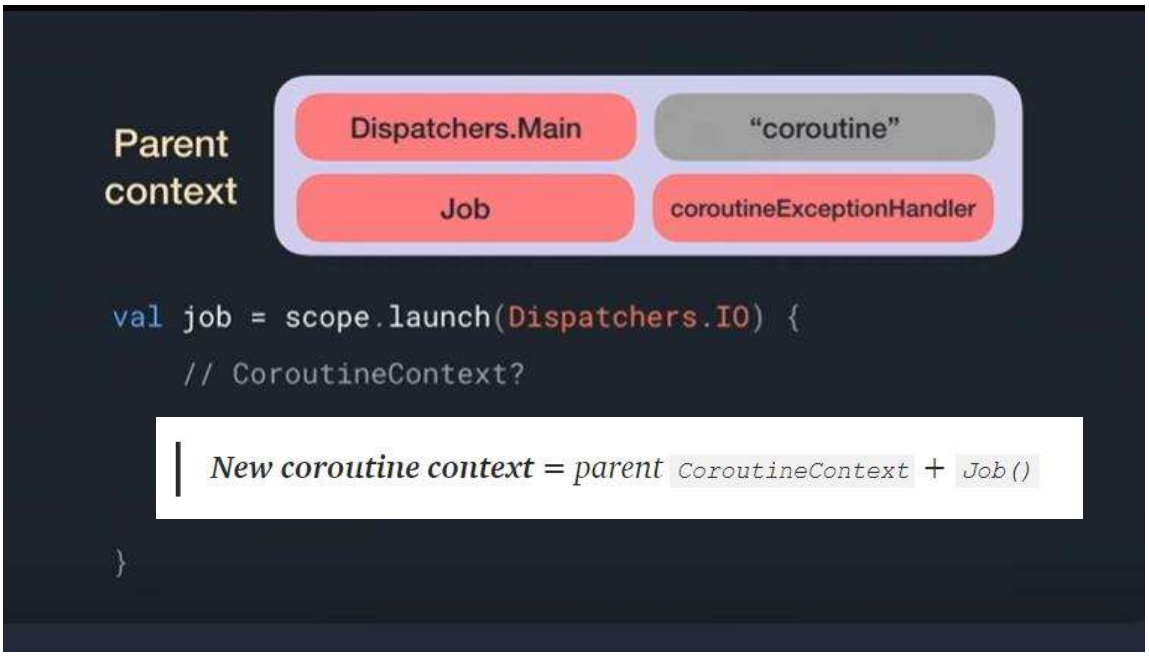
CoroutineContext of the Parent

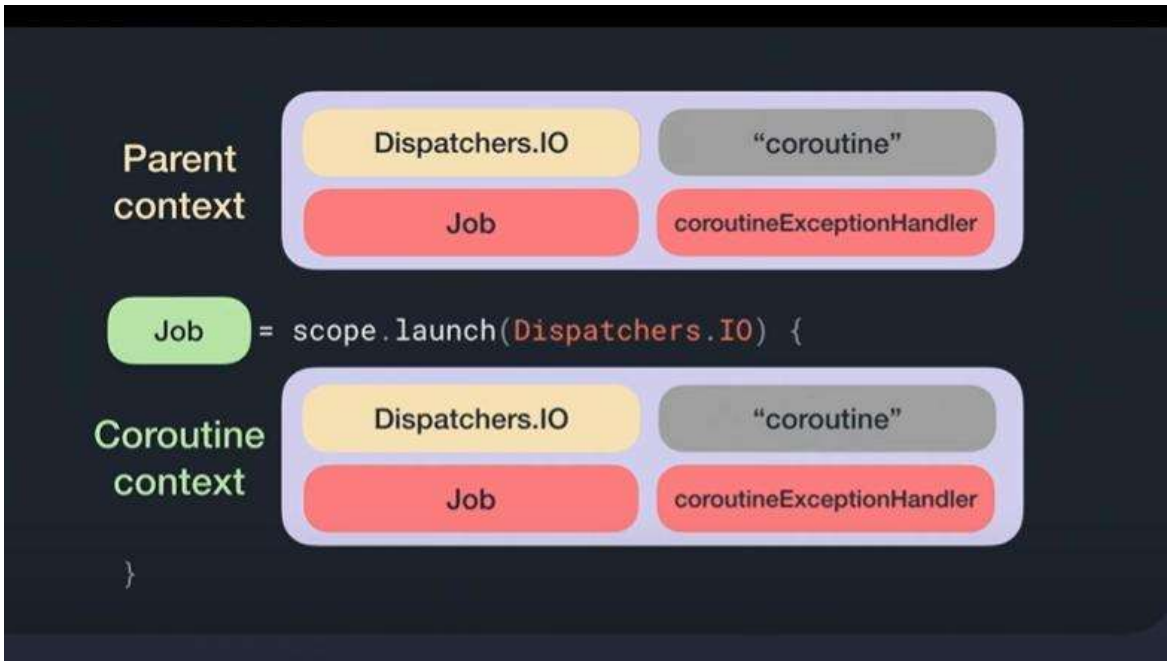


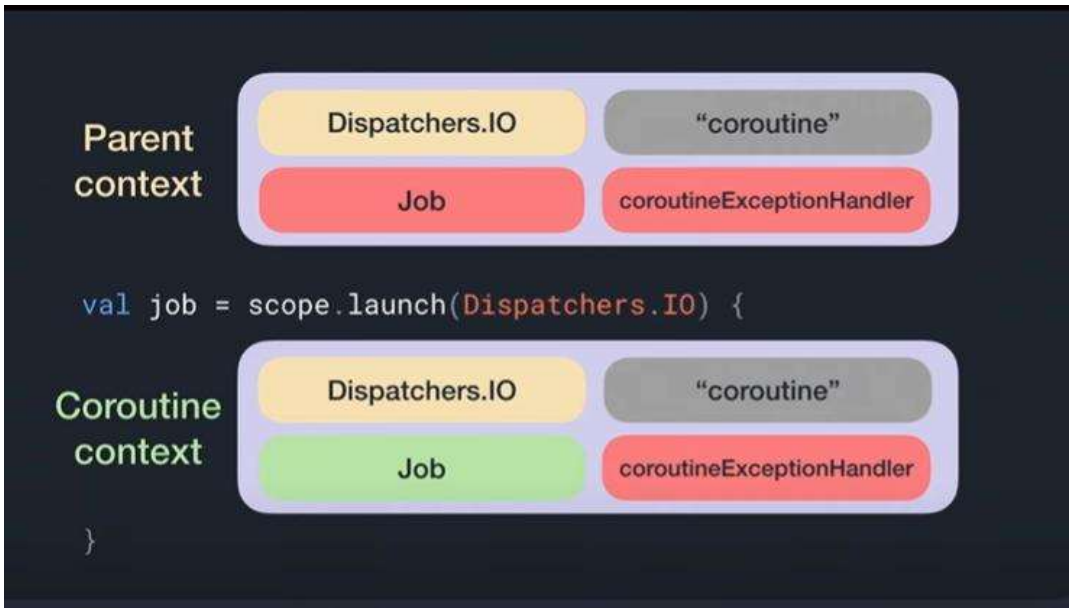
CoroutineContext of the Parent



Every coroutine started by this CoroutineScope will have at least those elements in the CoroutineContext. CoroutineName is gray because it comes from the default values.







The Job in the CoroutineContext and in the parent context will never be the same instance as a new coroutine always get a new instance of a Job

Dispatchers

- `Dispatchers.Default`
  - CPU-intensive computation
- `Dispatchers.Main`
  - UI events
  - Need to include dependencies like Android, Swing, JavaFX, etc.
- `Dispatchers.IO`
  - Network IO, Disk IO, etc.
- `Dispatchers.Unconfined`
  - Not recommended

Exception in thread "DefaultDispatcher-worker-3" java.lang.IllegalStateException: **Module with the Main dispatcher is missing**. Add dependency providing the Main dispatcher, e.g. 'kotlinx-coroutines-android'

Coroutine Scope Builders

- CoroutineScope (*already covered*)
- MainScope
- GlobalScope
- viewModelScope/lifecycleScope in Android (*will talk later!*)
- coroutineScope/supervisorScope (*will talk in coroutine scope functions*)

MainScope

- Creates the main `CoroutineScope` for UI components.

```
public fun MainScope(): CoroutineScope =
    ContextScope(SupervisorJob() + Dispatchers.Main)
```

```
class MyActivity : AppCompatActivity() {
    private val scope = MainScope() + CoroutineName("MyActivity")

    override fun onDestroy() {
        super.onDestroy()
        scope.cancel()
    }
}
```

`kotlinx-coroutines-android` — for Android Main thread dispatcher
`kotlinx-coroutines-javafx` — for JavaFx Application thread dispatcher
`kotlinx-coroutines-swing` — for Swing EDT dispatcher

GlobalScope

```
public object GlobalScope : CoroutineScope {
    // Returns [EmptyCoroutineContext].
    override val coroutineContext: CoroutineContext
        get() = EmptyCoroutineContext
}
```

- Not associated with any `Job` and launches **top-level** coroutines.
- Makes the coroutine lifecycle bound to the lifecycle of the **application**.
- An option when you don't care about coroutine results, posting to the UI thread or about the job completion.
- Use when you don't want to bind the jobs to the lifecycle of a certain object instance, like `Activity`, `Fragment`, `ViewModel` etc. in Android.

But, lose all the benefits you get from structural concurrency

The Reason to avoid GlobalScope
https://medium.com/@elizarov/the-reason-to-avoid-globalscope-835337445abc

Coroutine Builders (Revisited)

- launch
- async

} Coroutine scope extension functions

- runBlocking – regular functions
- runBlockingTest – regular functions (*later!*)
- withContext – suspending function

Coroutine Scope Functions

Functions that create a scope and behave similar to `coroutineScope`:

- **coroutineScope**
- **supervisorScope** is like `coroutineScope` using `SupervisorJob` instead of `Job`.
- **withContext** is `coroutineScope` that can modify coroutine context.
- **withTimeout**/**withTimeoutOrNull** is `coroutineScope` with a timeout.

Motivation for coroutineScope

- Imagine that in a suspending function you need to concurrently get data from two (or more) endpoints.

Approach 1: Sequential

- The first approach is calling suspending functions from a suspending function. The problem with this solution is that it is not concurrent.

```
// Data loaded sequentially, not simultaneously
suspend fun getUserProfile(): UserProfileData {
    val user = getUserData()
    val notifications = getNotifications()

    return UserProfileData(
        user = user,
        notifications = notifications,
    )
}
```

Approach 2: Concurrent, but not good idea

- To make two suspending calls concurrently, the easiest way is by wrapping them with `async`. But, using `GlobalScope` is not a good idea.

```
// DON'T DO THAT
suspend fun getUserProfile1(): UserProfileData {
    val user = GlobalScope.async { getUserData() }
    val notifications = GlobalScope.async {
        getNotifications()
    }

    return UserProfileData(
        user = user.await(),
        notifications = notifications.await(),
    )
}
```


Why not a good idea to use GlobalScope?

If we call `async` on a `GlobalScope`, we will have no relationship to the parent coroutine. It means:

- it cannot be canceled even if the parent canceled.
- it is not inheriting scope from any parent (it will always run on the default dispatcher, and will not respect any context from the parent).

The most important consequences are:

- potential memory leaks and unnecessary calculations,
- the tools for unit testing coroutines will not work here, and so testing this function is very hard.

Other Solutions?

- **Passing scope as an argument**, (or make as an extension function on `CoroutineScope`). Now, cancellation and proper unit testing are now possible.

```
suspend fun getUserProfile(scope: CoroutineScope): UserProfileData {
    val user = scope.async { getUserData() }
    val notifications = scope.async { getNotifications() }

    return UserProfileData(
        user = user.await(),
        notifications = notifications.await(),
    )
}

// DON'T DO THAT
suspend fun CoroutineScope.getUserProfile(): UserProfileData {
  …
}
```

Potentially ambiguous cotoutineContext!

But, can be tricky and potentially dangerous

- Good!
  - If there would be an exception in one `async`, the whole scope would be shut down (unless using `SupervisorJob`).
- Problematic
  - Requires passing the scope from function to function.
  - Any function that has access to the scope could easily abuse this access and for instance, cancel this scope with the `cancel` method.
  - **Parent coroutine that called** getUserProfile **cancels for no good reason.**

What will happen?

Exception in thread "main" java.lang.Error: Service exception

```
data class Details(val name: String, val followers: Int)
data class Tweet(val text: String)

suspend fun getUserName(): String { delay(500); return "geremy" }
suspend fun getFollowersNumber(): Int = throw Error("Service exception")

suspend fun getTweets(): List<Tweet> { delay(500); listOf(Tweet("Hello, world")) }

suspend fun getUserDetails(scope: CoroutineScope): Details {
    val userName = scope.async { getUserName() }
    val followersNumber = scope.async { getFollowersNumber() }
    return Details(userName.await(), followersNumber.await())
}

fun main() = runBlocking {
    val details = try {
        getUserDetails()
    } catch (e: Error) {
        null
    }
    println("User: $details")
    val tweets = async { getTweets() }
    println("Tweets: ${tweets.await()}")
}
```

We would like to see at least tweets, even if we have a problem calculating user details.

coroutineScope

```
suspend fun <R> coroutineScope(
    block: suspend CoroutineScope.() -> R
): R
```

- `coroutineScope` is a suspending function that starts a (sub) scope.
- Unlike `async` or `launch`, it does **not** really create new coroutines. The code block is *called in-place*.
- It inherits its `coroutineContext` from the outer scope, but overrides the context's `Job`.
- The produced scope respects parental responsibilities:
  - inherits a context from its parent,
  - awaits for all children before it can finish itself,
  - cancels all its children, when the parent is canceled.
- **An exception in `coroutineScope` or any of its children cancels other children and rethrows it.**
- Designed for *parallel decomposition* of work.

```
fun main() = runBlocking {
    val a = coroutineScope {
        delay(1000)
        10
    }
    println("a is calculated")
    val b = coroutineScope {
        delay(1000)
        20
    }
    println(a) // 10
    println(b) // 20
}
// (1 sec)
// a is calculated
// (1 sec)
// 10
// 20
```

Final Solution

- This all makes `coroutineScope` a perfect candidate for most cases when we just need to start a few concurrent calls in a suspending function.

```
suspend fun getUserProfile(): UserProfileData =
    coroutineScope {
        val user = async { getUserData() }
        val notifications = async { getNotifications() }

        UserProfileData(
            user = user.await(),
            notifications = notifications.await(),
        )
    }
```

- `coroutineScope` is nowadays often used to wrap suspending main body. Think of it as the modern replacement for the `runBlocking` function.

```
suspend fun main(): Unit = coroutineScope { ... }
```



```
data class Details(val name: String, val followers: Int)
data class Tweet(val text: String)
class ApiException(val code: Int, message: String) : Throwable(message)

suspend fun getUserName(): String { delay(500); return "paula abdul" }
suspend fun getFollowersNumber(): Int = throw ApiException(500, "Service unavailable")

suspend fun getTweets(): List<Tweet> { delay(500); return listOf(Tweet("Hello, world")) }

suspend fun getUserDetails(): Details = coroutineScope {
    val userName = async { getUserName() }
    val followersNumber = async { getFollowersNumber() }
    Details(userName.await(), followersNumber.await())
}

fun main() = runBlocking {
    val details = try {
        getUserDetails()
    } catch (e: ApiException) {
        null
    }
    println("User: $details")
    val tweets = async { getTweets() }
    println("Tweets: ${tweets.await()}")
}
// User: null
// Tweets: [Tweet(text=Hello, world)]
```

Proper Example of Parallel Decomposition

OK

```
suspend fun loadAndCombine(
  name1: String, name2: String, scope: CoroutineScope): Image {
  val deferred1 = scope.async { loadImage(name1) }
  val deferred2 = scope.async { loadImage(name2) }
  return combineImages(deferred1.await(), deferred2.await())
  }
}
```

Better

```
suspend fun loadAndCombine(name1: String, name2: String): Image {
coroutineScope {
  val deferred1 = async { loadImage(name1) }
  val deferred2 = async { loadImage(name2) }
  return combineImages(deferred1.await(), deferred2.await())
  }
}
```


supervisorScope

- The function `supervisorScope` also behaves a lot like `coroutineScope`.
- The difference is that it overrides context's `Job` with `SupervisorJob`, so it is not canceled when a child raises an exception.
- `supervisorScope` is mainly used in functions that start multiple independent tasks.

```
suspend fun notifyAnalytics(actions: List<UserAction>) =
    supervisorScope {
        actions.forEach { action ->
            launch {
                doSomething(action)
            }
        }
    }
```

Pro Tip: if you want parallel decomposition, then use a `coroutineScope` or, possibly a `supervisorScope` block

```
coroutineScope {
    launch {
        // ... task to run in the background
    }
    // ... more work while the launched task runs in parallel
}
// All work done by the time we reach this line
```

- `coroutineScope/supervisorScope` is a suspendable function and it won't complete until all the coroutines it launched complete.

withContext

 Very important for Main-Safety

```
public suspend fun <T> withContext(
    context: CoroutineContext, // caller's coroutineContext + context
    block: suspend CoroutineScope.() -> T
): T { … }
```

- Calls the suspending lambda with a given coroutine context, suspends until it completes, and returns the result.
- The resulting context for the block is derived by merging:
  - Caller's `coroutineContext` + the specified context.
- Often used to set a different coroutine dispatcher.
- if a new dispatcher specified, *shift* execution of the block *into the different thread*, and *back to the original dispatcher* when it completes.
- Note that the *result of `withContext` invocation is dispatched into the original context*.

Suspending Convention:
suspending functions do not block the caller thread

- To implement this convention, use `withContext` function.

```
suspend fun findBigPrime(): BigInteger =
    withContext(Dispatchers.Default) {
        BigInteger.probablePrime(4096, Random())
    }
```

```
suspend fun BufferedReader.readMessage(): Message? =
    withContext(Dispatchers.IO) {
        readLine()?.parseMessage()
    }
```

- Now you can call these suspending functions from the coroutine launched in the main thread of your UI application without blocking its main thread! Main Safety

Roman Elizarov

Project Lead for the Kotlin Programming Language @JetBrains

"Once you've isolated and encapsulated blocking code used by your application into suspending functions, you can call them at will from anywhere without having to double-check whether they are blocking or not."

*"You should always use **withContext()** inside a suspend function when you need **main-safety**, such as when reading from or writing to disk, performing network operations, or running CPU-intensive operations."*

withTimeout and withTimeoutOrNull

- `withTimeout` sets time limit for its body execution. If it takes too long, it cancels this body, and throws `TimeoutCancellationException`.
- `withTimeoutOrNull` just cancels its body and returns null instead of throwing an exception in case of exceeded timeout.
- Both are useful for testing.

```
suspend fun main(): Unit = coroutineScope {
  launch {
    launch { // cancelled by its parent
      delay(2000)
        println("Will not be printed")
    }
    withTimeout(1000) { // cancel launch
      delay(1500)
    }
  }
  launch {
    delay(2000)
    println("Done")
  }
}
// (2 sec)
// Done
```

```
class User()

suspend fun fetchUser(): User {
    while (true) { yield() } // Run forever
}

suspend fun getUserOrNull(): User? =
    withTimeoutOrNull(1000) {
        fetchUser()
    }

suspend fun main(): Unit = coroutineScope {
    val user = getUserOrNull()
    println("User: $user")
}
// (1 sec)
// User: null
```


Coroutine Scope Functions *vs.* Coroutine Builders

Coroutine Builders (except runBlocking)

- `launch`, `async`, `produce`
- Extension functions on `CoroutineScope`
- Take coroutine context from `CoroutineScope` receiver
- Starts another coroutine
- Exception propagates to parent Job

Coroutine Scope Functions

- `coroutineScope`, `supervisorScope`, `withContext`, `withTimeout`
- suspending functions
- Take coroutine context from suspending function continuation
- Runs on parent coroutine in-place
- Exception rethrows except `supervisorScope`

When to or NOT to mark a function as suspend

Mark it whenever it calls other suspending function

```
suspend fun loadData() {
    val data = networkRequest()
    show(data)
}
```

Do not mark it when it doesn't call suspending functions

```
fun onButtonClicked() {
    scope.launch() {
        loadData()
    }
}
```

Don't mark a function suspend unless you're forced to.

Suspension Points: Exactly When?

- A *suspension point* is a point where the execution of the coroutine may be suspended.
- Syntactically, a suspension point is an invocation of suspending function, ***but*** the actual suspension happens when the suspending function invokes the following standard library primitive.

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

suspendCoroutine or supendCancellableCoroutine

```
suspend fun <T : Any> Call<T>.await(): T = suspendCoroutine { cont ->
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            if (response.isSuccessful)
                cont.resume(response.body()!!)
            else
                cont.resumeWithException(HttpException(response))
        }
        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Inspired by **call/cc** from Scheme

Coroutines in Android

2018 Solution Alternatives

- **LiveData**
  - Observable data holder
  - Love it, but want a complete solution
- **RxJava**
  - Observable + Schedulers + Observer
  - Powerful, highly adopted but often misused and perceived as an overkill solution to concurrency
- **Coroutines**
  - Suspendable computations
  - Seems like the best solution, but here is a need for maturation of the extensions and developers feel that a steep learning curve awaits …

Posting to the UI thread (Non-**Main-Safety**)

- Using `Dispatchers.Main` as the context, you can post to the main thread in **Android**, **Swing** and **JavaFx** applications.

```
suspend fun loadData() {
    val data = apiService.networkRequest()
    withContext(Dispatchers.Main) {
        show(data)
    }
}
```

Exception in thread "DefaultDispatcher-worker-3" java.lang.IllegalStateException: **Module with the Main dispatcher is missing**. Add dependency providing the Main dispatcher, e.g. 'kotlinx-coroutines-android'

Posting to the UI thread (**Main-Safety**)

```
// Main safety
suspend fun loadData() {
    val data = withContext(Dispatchers.IO) {
        apiService.networkRequest()
    }
    show(data)
}
```


Autocancellation with lifecycleScope and viewModelScope

The two extension `CoroutineScope` properties *automatically* cancel the coroutine on the lifecycle's destroy event using built-in cancelation mechanism:

- `viewModelScope`
- `lifecycleScope`

viewModelScope

- An extension property of the `ViewModel` class.
- This scope is bound to `Dispatchers.Main.immediate` and will automatically be cancelled when the `ViewModel` is cleared.

```
public val ViewModel.viewModelScope: CoroutineScope
    get() { …
        CloseableCoroutineScope(
            SupervisorJob() + Dispatchers.Main.immediate))
    }
```

Executes coroutines immediately when they are launched from the main thread.

```
dependencies {
  implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$version
}
```


viewModelScope

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {
            // Coroutine that will be canceled when the ViewModel is cleared.
        }
    }
}

override fun onCleared() {
    super.onCleared()
    viewModelScope.cancel() // you don't need this!
}
```

lifecycleScope

- An extension property of the `LifecycleOwner` instance and it's bounded to the Lifecycle of the `Activity` or `Fragment`.
- Bound to `Dispatchers.Main.immediate` and will automatically be cancelled when the `LifeCycle` is destroyed.
- You can access the `CoroutineScope` of the Lifecycle either via `lifecycle.coroutineScope` or `lifecycleOwner.lifecycleScope` properties.

```
public val LifecycleOwner.lifecycleScope: LifecycleCoroutineScope
    get() = lifecycle.coroutineScope
```

```
dependencies {
  implementation "androidx.lifecycle:lifecycle-runtime-ktx:$version
}
```


lifecycleScope

```
class MyFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            val params = TextViewCompat.getTextMetricsParams(textView)
            val precomputedText = withContext(Dispatchers.Default) {
                PrecomputedTextCompat.create(longTextContent, params)
            }
            TextViewCompat.setPrecomputedText(textView, precomputedText)
        }
    }
}
```

Associating coroutine launch time with Lifecycle state

- `LifecycleCoroutineScope.launchWhenCreated()`
  – It launches when the Lifecycle is at least in the Lifecycle.State.CREATED state.
- `LifecycleCoroutineScope.launchWhenStarted()`
  – It launches when the Lifecycle is at least in the Lifecycle.State.STARTED state.
- `LifecycleCoroutineScope.launchWhenResumed()`
  – It launches when the Lifecycle is at least in the Lifecycle.State.RESUMED state.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        lifecycleScope.launchWhenResumed {
            doSomeLongRunningJob()
        }
    }
}
```


LiveData Exercise

```
data class Article(val id: String,
                   val author: String,
                   val title: String)
```

```
interface ApiService {
    /**
     * Get all articles
     */
    suspend fun getArticles(): Resource<List<Article>>

    /**
     * Get the most recommended (i.e., top-ranked) article
     */
    suspend fun getTopArticle(): Resource<Article>

    /**
     * Get all the articles written by the author of the current top-ranked article
     */
    fun getArticlesByAuthorId(id: String): LiveData<Resource<List<Article>>>
}
```

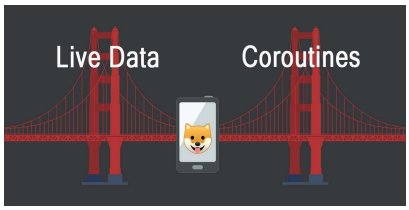
LiveData in ViewModel

```
class ArticleViewModel(private val apiService: ApiService) : ViewModel() {

    private val _articles = MutableLiveData<Resource<List<Article>>>()
    val articles: LiveData<Resource<List<Article>>> = _articles

    init {
        scope.launch {
            withContext(Dispatchers.IO) {
                _articles.postValue(apiService.getArticles())
            }
        }
    }

    val topArticle: LiveData<Resource<Article>> = ...

    val articlesByTopAuthor: LiveData<Resource<List<Article>>> = ...
    ...
}
```

canonical style

Try this …

liveData Builder



- Runs a coroutine when observed
- Automatically cancelled if no active observers within *timeout*
- No scope needed!!
- Emit values (`emit()`) or streams (`emitSource()`)

```
public fun <T> liveData(
    context: CoroutineContext = EmptyCoroutineContext,
    timeout: Duration,
    block: suspend LiveDataScope<T>.() -> Unit
): LiveData<T> = CoroutineLiveData(context, …, block)
```

Default: Dispatchers.Main.immediate

```
dependencies {
  implementation "androidx.lifecycle:lifecycle-livedata-ktx:$version
}
```

Creating *LiveData* emitting values (1)

```
val recipe = MutableLiveData<Resource<Recipe>>().apply {
    viewModelScope.launch {
        this@apply.value = Resource.Loading
        this@apply.value = longRunningTask()
    }
}
```

```
val recipe: LiveData<Resource<Recipe>> = liveData {
    emit(Resource.Loading)
    emit(longRunningTask())
}
```


Creating *LiveData* emitting values (2)

```
// a LiveData that fetches a `User` object based on a `userId`
// and refreshes it every 30 seconds as long as it is observed
val userId: LiveData<String> = MutableLiveData<>()
val user: LiveData<User> = userId.switchMap { id ->
    liveData {
        while(true) {
            // note that `while(true)` is fine because the `delay(30_000)`
            // below will cooperate in cancellation if LiveData is not
            // actively observed anymore
            val user = api.fetch(id) // errors are ignored for brevity
            emit(user)
            delay(30_000)
        }
    }
}
```

Creating *LiveData* that changes emitting source

```
// a LiveData that immediately receives a LiveData<User> from the
// database and yields it as a source but also tries to back-fill
// the database from the server
val user = liveData {
    val fromDb: LiveData<User> = roomDatabase.loadUser(id)
    emitSource(fromDb)
    val updated = api.fetch(id) // errors are ignored for brevity

    // Since we are using Room here, updating the database will
    // update the `fromDb` LiveData that was obtained above.
    roomDatabase.insert(updated)
}
```


Coroutines are good for …

There are two types of tasks that coroutines are a great solution for:

1. One shot requests are requests that are run each time they are called: they always complete after the result is ready.
   – Use **suspending functions**

2. Streaming requests are requests that continue to observe changes and report them to caller: they don't complete when the first result is ready.
   – Use **Kotlin Flow** *(This topic will be covered in Flow Course)*

The one shot request pattern

https://medium.com/androiddevelopers/coroutines-on-android-part-iii-real-work-2ba8a2ec2f45

Sean McQuillan

Add coroutines to the `ViewModel`, `Repository`, and `Room`, and each layer has a different responsibility.

- `ViewModel` **starts** a coroutine on the *main* thread — it completes when it has a result.
- `Repository` exposes regular suspend functions and ensures they are *main-safe*.
- The database and network expose regular suspend functions and ensures they are *main-safe*.

Note: `Room` uses its own dispatcher to run queries on a background thread. Your code should **not** use `withContext(Dispatchers.IO)` to call suspending room queries. It will complicate the code and make queries run slower. Likewise, `Retrofit` is also *main-safe* and run on a custom dispatcher.

The one shot request pattern (Cont'd)

- `suspendCoroutine`/`suspendCancellableCoroutine`
  - A typical use of this function is to suspend a coroutine while waiting for a result from a single-shot callback API and to return the result to the caller.
- `runCatching`

- When starting a new coroutine in response to a UI event, consider what happens if the user starts another before this one completes.