Kotlin coroutines

김정선 (金 正 善, Jungsun Kim)
한양대학교 소프트웨어학부
소프트웨어융합대학
(College of Computing)

Office: 4공학관 316호
Email: kimjs@hanyang.ac.kr

What is a Coroutine?

What is a Coroutine?

What is a Coroutine?

(Generators = semi-coroutines)

Questions

What's the difference between a CoroutineScope and a CoroutineContext?

What's the difference between a coroutineContext and a CoroutineContext?

When to use suspend function?

Do I need to switch the dispatcher?

What is Structured Concurrency?

How to handle cancellation?

How to handle exceptions?

How to test ...?

Coroutines (Co + Routines)

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing

Coroutines (Co + Routines)

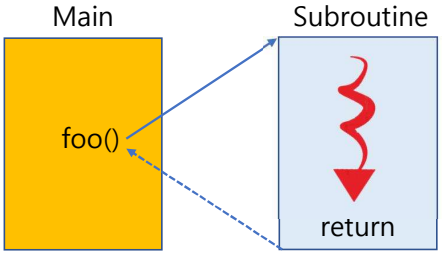
- Melvin Conway coined the term in 1958.
- Donald knuth - “The Art of Computer Programming”

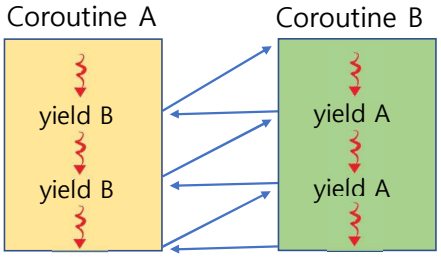
A main routine and subroutines

vs.

Coroutines, which call on each other

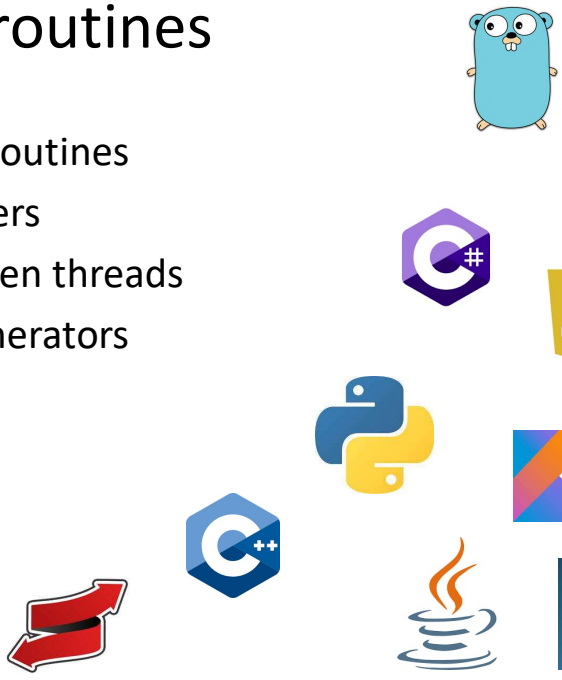
Cooperative multitasking
(aka, Non-preemptive multitasking)





Coroutines

- goroutines
- fibers
- green threads
- generators



Kotlin official coroutines documentation

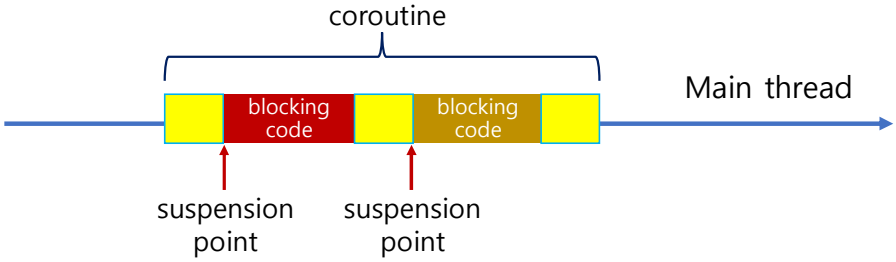
https://kotlinlang.org/docs/reference/coroutines.html#blocking-vs-suspending

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

Suspended = stop and continue? That sounds like blocking to me!

What is a Coroutine? (Warning: My Definition)

A coroutine is *a sequence of computations*, (sub-tasks)
each of which may be *suspended* (or *paused*) and *resumed* at some point,
without blocking the thread that executes it.



How can a thread be blocked?

Blocking threads, suspending coroutines

Roman Elizarov Nov 4, 2018 · 7 min read

- Using *blocking IO* (*IO-bound* task)

```
fun BufferedReader.readMessage(): Message? =
  readLine()?.parseMessage()
```

- Run a *CPU-intensive* computation (*CPU-bound* task)

```
fun findBigPrime(): BigInteger =
  BigInteger.probablePrime(4096, Random())
```


Threads are expensive, so blocking a thread is something that should be avoided

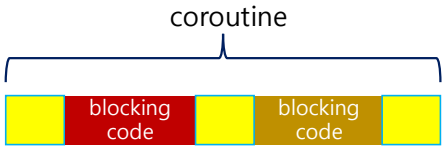
- Thread calling those functions cannot do anything else
  - it cannot execute other requests,
  - it cannot process UI events.

- You should avoid blocking
  - limited request-processing threads in backend application, or
  - main UI thread.

Use non-blocking I/O library

Have no choice but to block *some* thread, but always have a choice of *what* thread to block.

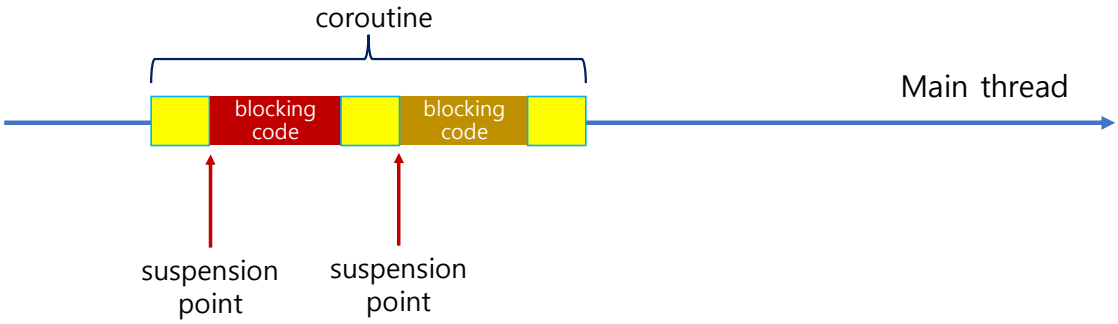
Suspending coroutines



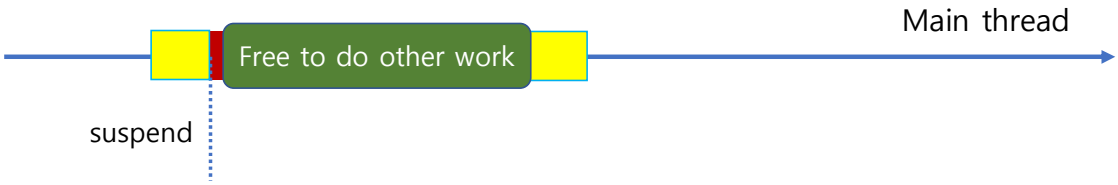
- Coroutines provide an *alternative to thread blocking* by supporting *suspension*.
- So, what is the difference between *blocking* a thread and *suspending* a coroutine?

```
val data = awaitData() // does it block or suspend?
processData(data)
```

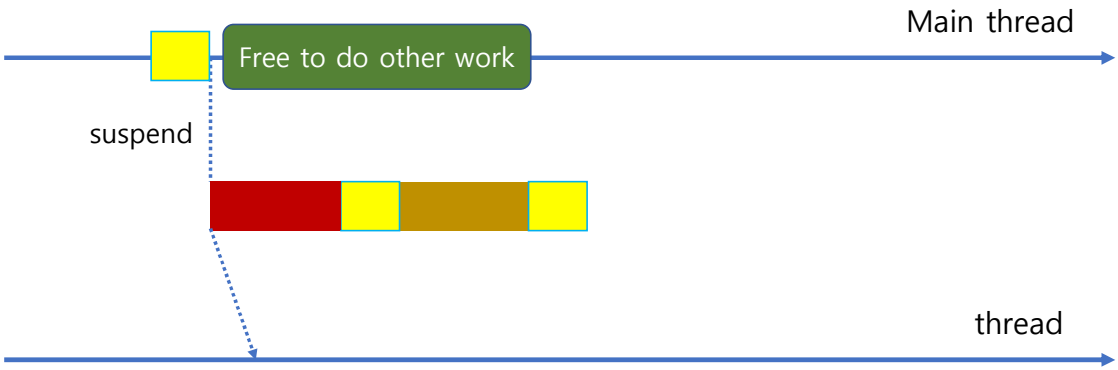

Coroutine is a non-blocking suspend computation



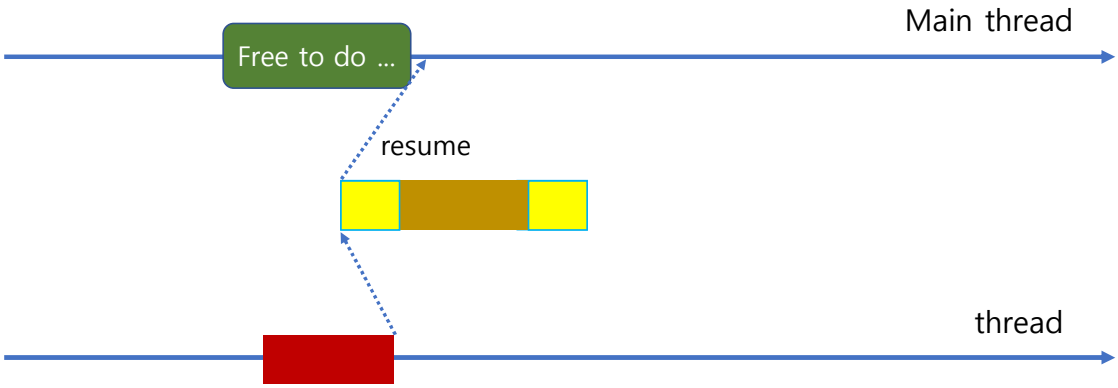
Coroutine is a non-blocking suspend computation



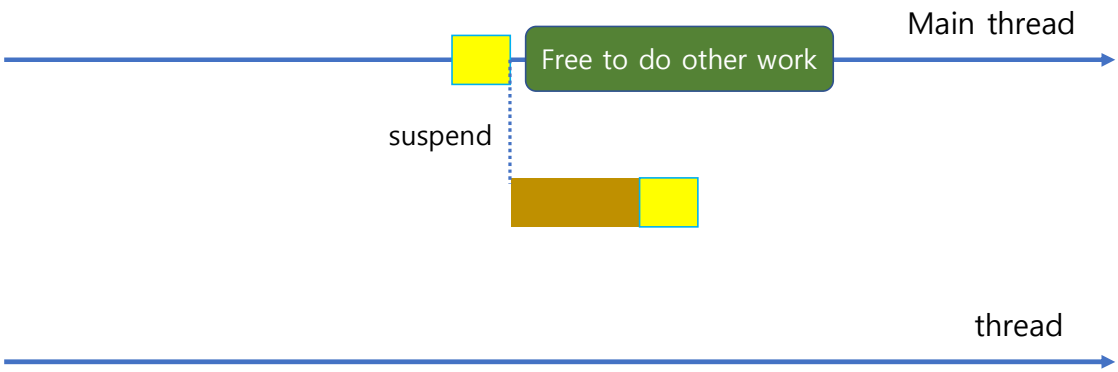
Coroutine is a non-blocking suspend computation



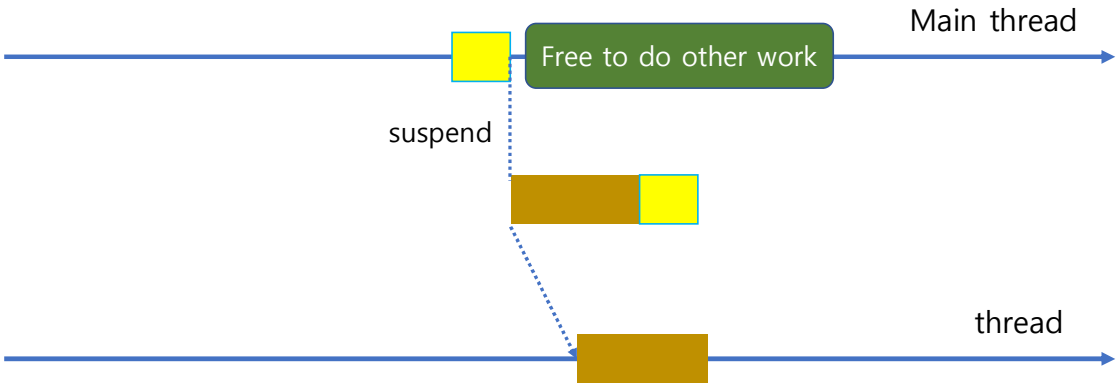
Coroutine is a non-blocking suspend computation



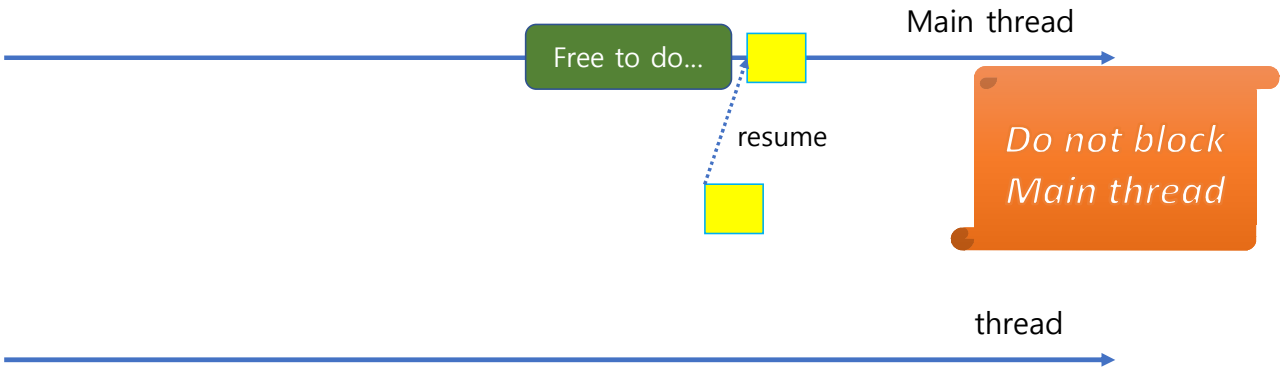
Coroutine is a non-blocking suspend computation



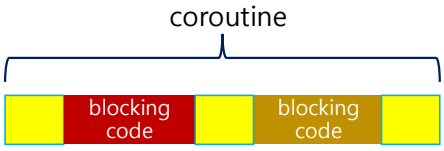
Coroutine is a non-blocking suspend computation



Coroutine is a non-blocking suspend computation



Suspending Functions



- A *suspending function* is a function defined with `suspend` modifier.

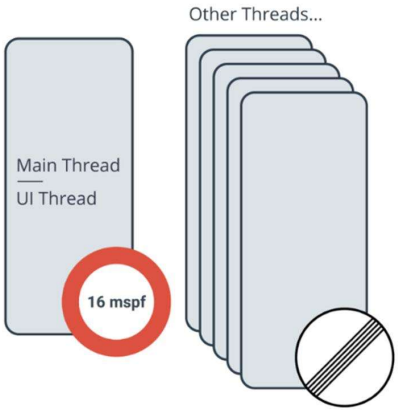
```
suspend fun createPost(token: Token, item: Item): Post {…}
```

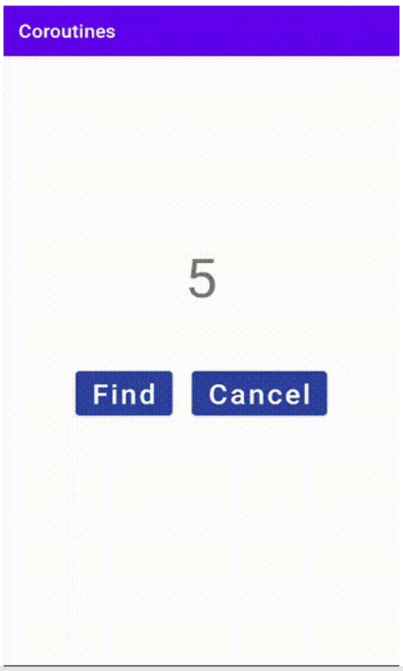
- Enable us to explicitly recognize the blocking code.
- Tells the compiler that this function may take long time to execute, so needs to be executed inside a coroutine.
- One mistake that is often made is that adding a `suspend` modifier to a function makes it either asynchronous or non-blocking.

```
suspend fun findBigPrime(): BigInteger =
    BigInteger.probablePrime(4096, Random())
```

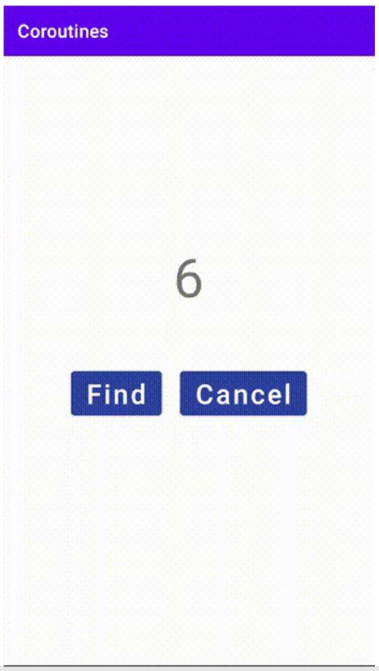

Why Coroutines in Android?

- On Android, the *main thread* (aka *UI thread*) is a single *default thread* that handles:
  - all updates to the UI.
  - calls all click handlers and other UI and lifecycle callbacks
- Without explicit thread switching, everything app does is on the main thread.
- Blocking in this context means the UI thread is not doing anything at all while it waits for something like a database to finish updating.
- We need *a way to handle long-running tasks without blocking the main thread*.






```
suspend fun findBigPrime() =
  BigInteger.probablePrime(4096, Random())
```




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

Asynchronous Programming

- Callbacks
- Future/Promise/Rx
- Coroutines

From Synchronous to Asynchronous

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

```
fun requestToken(): Token {
    // makes request for a token & waits
    return token // returns result when received
}

fun createPost(token: Token, item: Item): Post {
    // sends item to the server & waits
    return post // returns resulting post
}

fun showPost(post: Post) {
    // does some local processing of result
}
```

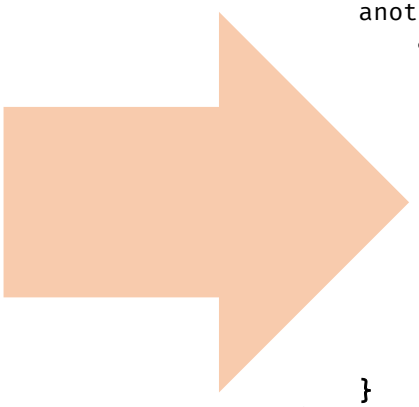

Callbacks

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

- hard to read and harder to reason about
- Handling exceptions makes it a real mess

```
fun requestToken(cb: (Token) -> Unit) { // returns immediately
    DefaultScehduler.execute {
        // Blocking network request code here …
        cb(token)
    }
}

fun createPost(token: Token, item: Item, cb: (Post) -> Unit) { // returns immediately
    DefaultScehduler.execute {
        // Blocking network request code here …
        cb(post)
    }
}

fun showPost(post: Post) { … }
```

```
private fun loadData() {
    networkRequest { data ->
        anotherRequest(data) { data1 ->
            anotherRequest(data1) { data2 ->
                anotherRequest(data2) { data3 ->
                    anotherRequest(data3) { data4 ->
                        anotherRequest(data4) { data5 ->
                            anotherRequest(data5) { data6 ->
                                anotherRequest(data6) { data7 ->
                                    anotherRequest(data7) { data8 ->
                                        anotherRequest(data8) { data9 ->
                                            anotherRequest(data9) {
                                                // How many more do you want?
                                                println(it)
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```



Callback Hell

Promise/Future

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


No nesting indentation


Composable & propagates exceptions


Library-specific operators

```
fun requestToken(): CompletableFuture<Token> {
    // makes request for a token
    // returns promise for a future result immediately
}

fun createPost(token: Token, item: Item): CompletableFuture<Post> {
    // sends item to the server
    // returns promise for a future result immediately
}

fun showPost(post: Post) { … }
```

RxJava

```
fun requestToken(): Single<Token>
fun createPost(token: Token, item: Item): Single<Post>
fun showPost(post: Post)

fun postItem(item: Item) {
    requestToken()
        .flatMap { token ->
            createPost(token, item)
        }
        .subscribeOn(Schedulers.io())
        .observerOn(AndroidSchedulers.mainThread())
        .subscribe { post ->
            showPost(post)
        }
}
```

Looks complicated …

Coroutines

The suspending world is nicely sequential!

suspension points

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

VS.

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

```
suspend fun requestToken(): Token {
    // makes request for a token & suspends
    return token // returns result when received
}
```

A function with a `suspend` modifier

- Take long time to execute
- Suspend and coroutine

suspending function

```
suspend fun createPost(token: Token, item: Item): Post {
    // sends item to the server & suspends
    return post // returns result when received
}
fun showPost(post: Post) { … }
```

Bonus Features

- Regular loops

```
for ((token, item) in list) {
    createPost(token, item)
}
```

- Regular exception handling

```
try {
    createPost(token, item)
} catch (e: BadTokenException) {
    …
}
```

- Regular higher-order functions
  - `forEach`, `let`, `apply`, `repeat`, `filter`, `map`, `use`, etc

```
file.readLines().forEach { line ->
    createPost(token, line.toItem())
}
```

Everything like blocking code!

Higher-Order Functions

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

val post = retryIO {
    createPost(token, item)
}

suspend fun <T> retryIO(block: suspend () -> T): T {
    var backOffTime = 1000L // start with 1 sec
    while (true) {
        try {
            return block()
        } catch (e: IOException) {
            e.printStackTrace() // log the error
        }
        delay(backOffTime)
        backOffTime = minOf(backOffTime * 2, 60_000L)
    }
}
```

suspending lambda

Calling Suspending Functions

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

Regular function *cannot* suspend execution

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

Can *suspend* execution

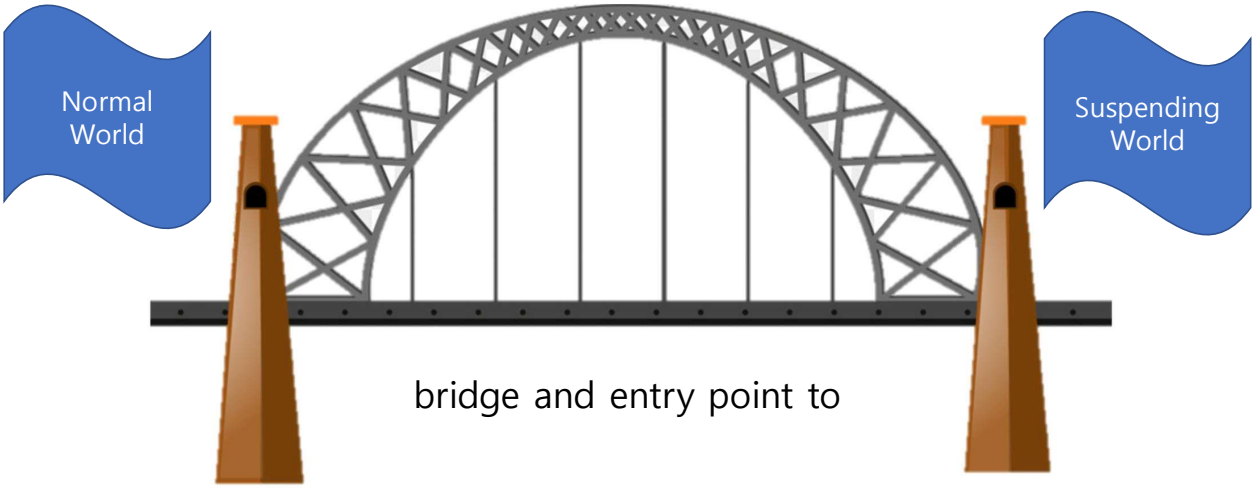
Error: Suspend function should be called only from a **coroutine** or **another suspend function**

Calling Suspending Functions

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

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

Coroutine Builders are bridges between …

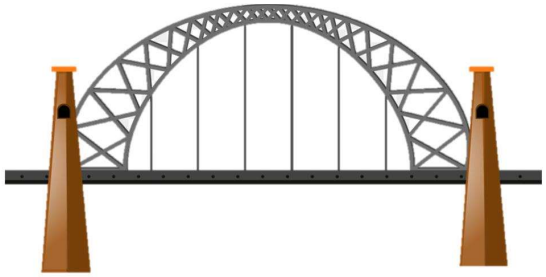


Bridging the normal world and the suspending world

- *Coroutine builders* are simple functions that create a new coroutine to run a given suspending function.

Frequently used builders

- `launch`
  - to fire and forget
- `async`
  - to get a result asynchronously
- `runBlocking`
  - block the current thread



runBlocking

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

- Block the current thread until the suspending lambda finishes executing.
- Often used from the `main()` function to give a sort of *top-level coroutine* from which to work, and keep the JVM alive while doing so.
- `runBlocking` is very *useful in tests*, you can wrap your tests in `runBlocking`.
  - This will make sure your *test code execute sequentially on the same thread* and will *not terminate until all coroutines are completed*.

```
fun main() {
    println("Hello,")
    // Create a coroutine, and block the main thread until it completes
    runBlocking {
        delay(2000L) // suspends the current coroutine for 2 seconds
    }
    println("World!") // will be executed after 2 seconds
}
```


launch

Coroutines should be created inside a CoroutineScope!

Returns immediately, coroutine works in *background thread pool* (`Dispatchers.Default` by default)

extension function on CoroutineScope

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

Fire and forget!

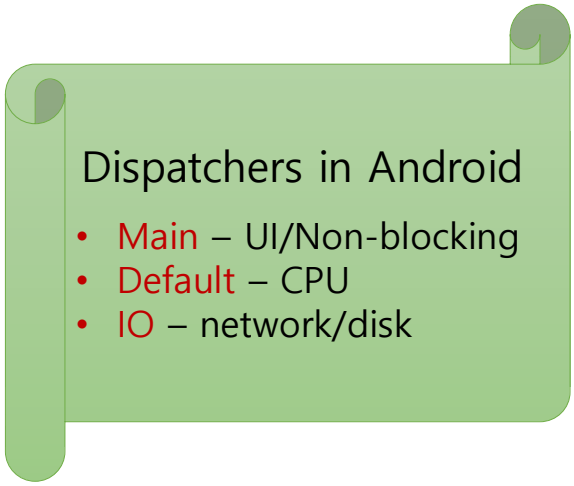
Sneak Preview of CoroutineScope

- It's just an object. So, create it, if needed.

```
val scope = CoroutineScope(Job())
scope.launch {
    println("Hello, I am coroutine")
}
```

- Use scope builder
  - `coroutineScope` or `supervisorScope` (← suspend functions)
- Use ready-made scopes provided by library or frameworks
  - `lifecycleScope` and `viewModelScope` in Android
  - `GlobalScope` in Kotlin (*not recommended, though*)

Sneak Preview of Dispatchers



Launch (Cont'd)

extension function on CoroutineScope

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

suspending lambda

```
job.cancel() // cancel the job
job.join()   // wait for job completion
```


launch: Don't do this

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

Warning: do not use `GlobalScope` if possible.

https://elizarov.medium.com/the-reason-to-avoid-globalscope-835337445abc

async/await

```
suspend fun loadImage(name: String): Image = { … }
fun combineImages(img1: Image, img2: Image): Image = { … }
```

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

Kotlin's future type

await function

suspends until deferred job is complete

Async (Cont'd)

extension function *on CoroutineScope*

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

suspending lambda

```
deferred.cancel()   // cancel the job
val result = deferred.await()    // wait for job completion
```

async/await: Don’t do this

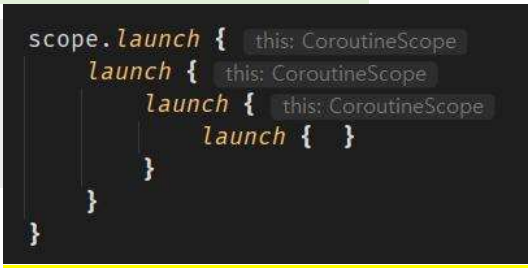
```
suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = GlobalScope.async { loadImage(name1) }
    val deferred2 = GlobalScope.async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

Warning: do not use `GlobalScope` if possible.

Magic of launch & async

```
fun CoroutineScope.launch(
    …
    block: suspend CoroutineScope.() -> Unit
): Job { … }
```

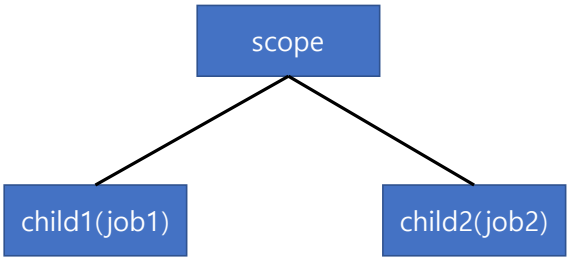
```
fun <T> CoroutineScope.async(
    …
    block: suspend CoroutineScope.() -> T
): Deferred<T>
```



Coroutines form a hierarchy (parent-child relationship)

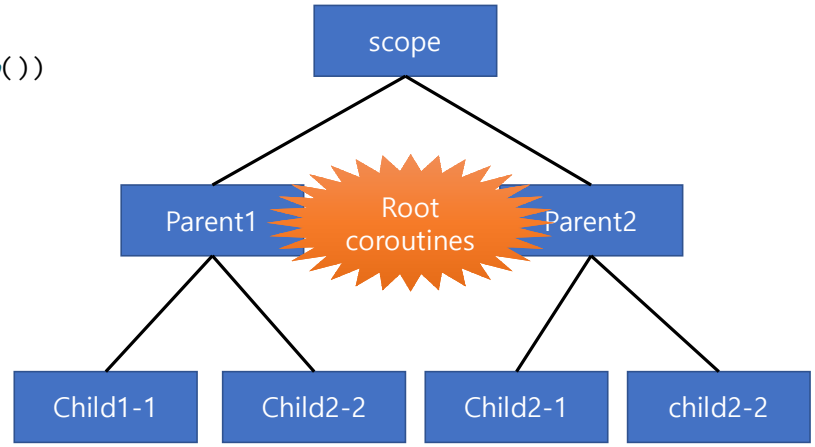
Coroutines form a hierarchy

```
val scope = CoroutineScope(Job())
val job1 = scope.launch {


}
val job2 = scope.launch {


}
joinAll(job1, job2)
```



Coroutines form a hierarchy

```
val scope = CoroutineScope(Job())
val job1 = scope.launch {
    launch { }
    launch { }
}
val job2 = scope.launch {
    launch { }
    launch { }
}
joinAll(job1, job2)
```



Coroutine behavior until Kotlin 1.2.0
(Concept of Structured Concurrency does not exist)

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

- What if the coroutine that calls the `loadAndCombine` cancelled?
  - Then loading of both images still proceeds unfazed.

Coroutine behavior until Kotlin 1.2.0 (Cont'd)
(Concept of Structured Concurrency does not exist)

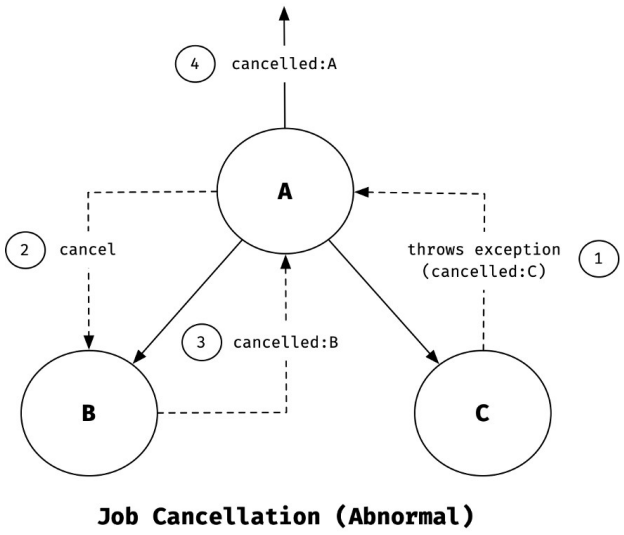
```
suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = async(coroutineContext) { loadImage(name1) }
    val deferred2 = async(coroutineContext) { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

- The solution was to write `async(coroutineContext){…}` so that loading of both images is performed in children coroutines that are cancelled when their parent coroutine is cancelled.
- But, what if the first `loadImage` fails?
  - Then `deferred1.await()` throws the corresponding exception, but the second `async` coroutine, that is loading the second image, still continues to work in background.

As of Kotlin 1.3.0

Structured Concurrency

- *Prevent resource leak and avoid unnecessary computation.*
- Coroutines can form a **hierarchy**, which allows a parent coroutine to automatically **manage the life cycle** of its child coroutines.
- The parent can for instance wait for its children to complete, or cancel all its children if an exception occurs in one of them.



Job Cancellation (Abnormal)

Essence of Structured Concurrency

1. Every coroutine needs to be started in a logical scope with a limited life-time.
2. Coroutines started in the same scope form a hierarchy.
3. A parent job won't complete until all its children have completed.
4. Cancelling a parent or failure (with its own exceptions) will cancels all its children. Cancelling a child won't cancel the parent and its siblings.
5. If a child coroutine fails, the exception is propagated upwards and depending on the job type (`Job` or `SupervisorSob`), either the parent and all of its siblings cancelled, or they are not affected.

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())
  }
}
```