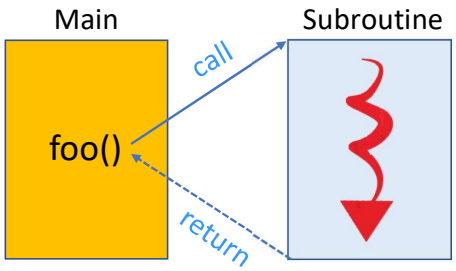
# Kotlin coroutines

**Melvin Conway coined the term in 1958**. Coroutines are lightweight, independent instances of code that can be launched to do certain tasks, get suspended, usually to wait for asynchronous events, and be resumed to continue their jobs. Coroutines make it easier to build highly-concurrent software that performs many tasks at the same time or keeps track of many independent event streams.




# Normal Functions

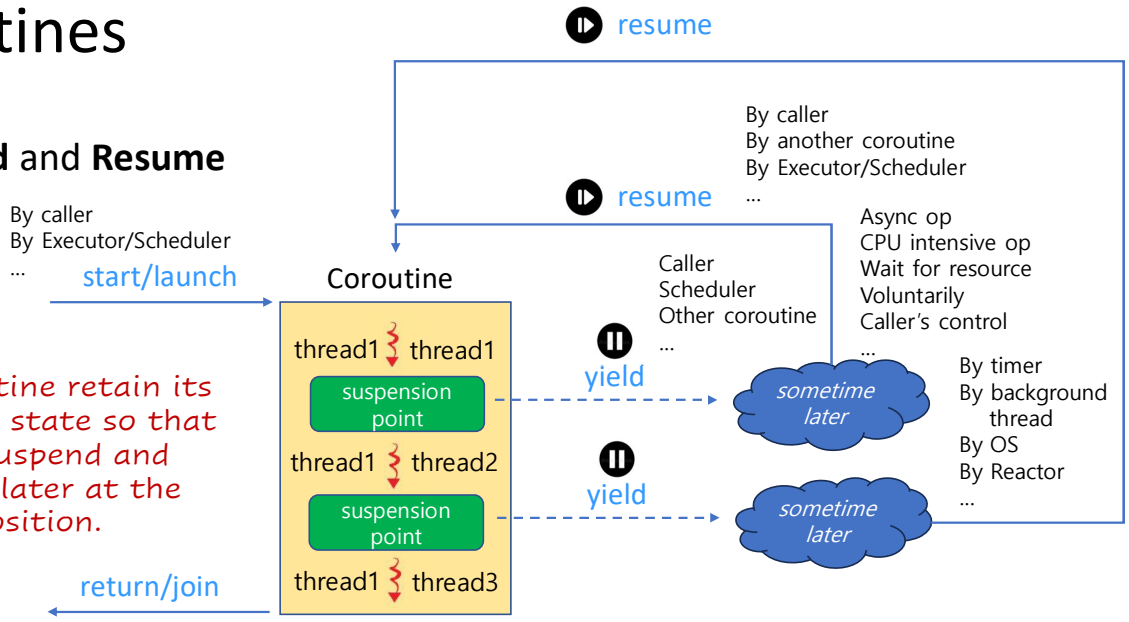
- **Call** and **Return**




- This calling process is usually a ***one-time action***.
- If the function is called again, it is treated as another independent action.

# Coroutines

- **Suspend** and **Resume**




A coroutine retain its internal state so that it can suspend and resume later at the same position.

- Coroutines can be seen as a generalization of regular functions.



# What is a Coroutine?

https://godbolt.org/z/sW1socfxv

1 7 2 8 6 5 3 0 2 4 7

# What is a Coroutine?

(Generators = semi-coroutines)

https://godbolt.org/z/8GWE6e1bx

https://pl.kotl.in/lrA-HqVd3



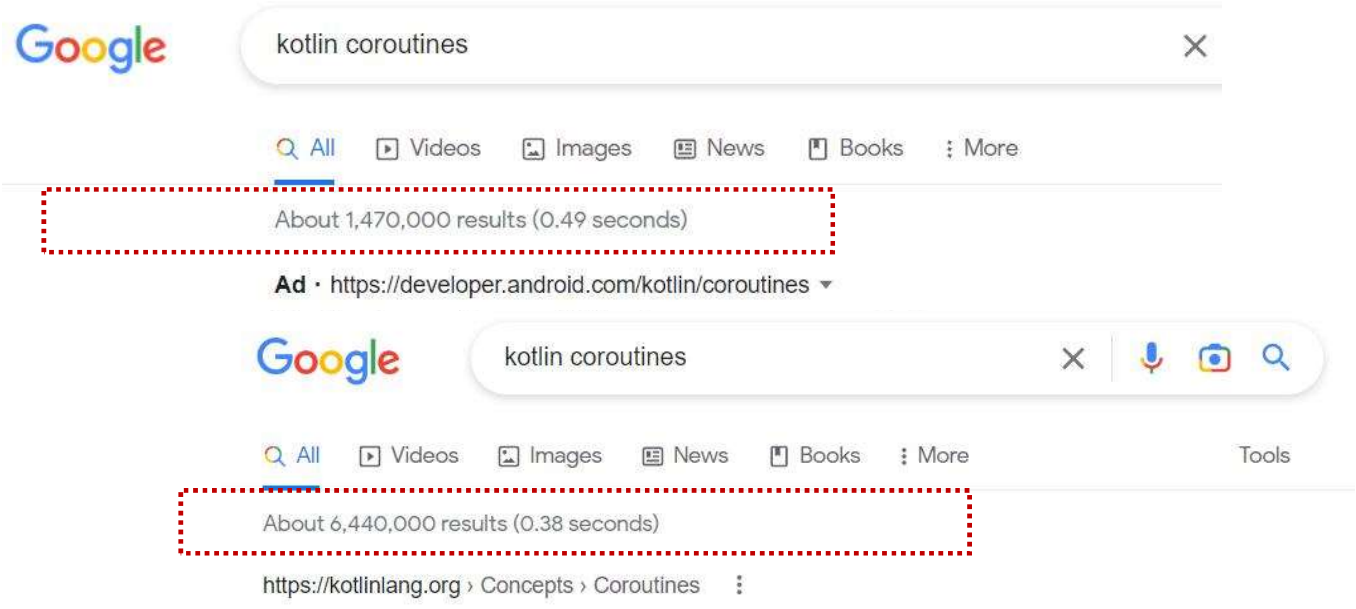
# Tail Recursions

```
fun fact(n: Int): Int =
    if (n <= 1) {
        1
    } else {
        n * fact(n - 1)
    }
```

```
tailrec
fun fact(n: Int, acc: Int): Int =
    if (n <= 1) {
        acc
    } else {
       fact(n - 1, n * acc)
    }
```

```
fun fact(n: Int, acc: Int): () -> Int =
    if (n <= 1) {
        { acc }
    } else {
        fact(n - 1, n * acc)
    }
```





# Kotlin
## Coroutine-specific
# Questions

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

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

What is **suspend function** and when to use?

What is the **dispatcher** and do I need to switch dispatchers?

What is **Structured Concurrency** and why we need it?

How to handle **cancellation**?

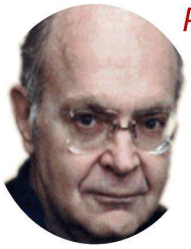
How to handle **exceptions**?

How to **test** ...?



# Coroutines (Co + Routines)

- Melvin Conway coined the term in 1958.
- Donald Knuth - "The Art of Computer Programming"



Donald Knuth

*Premature optimization is the root of all evil.*

**Conway's law**
"Organizations, who design systems, are constrained to produce designs which are copies of the communication structures of these organizations."

# Coroutines (Co + Routines)

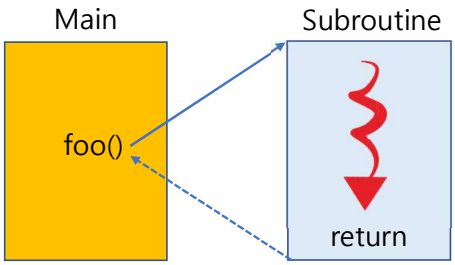
- Melvin Conway coined the term in 1958.
- Donald knuth - “The Art of Computer Programming”

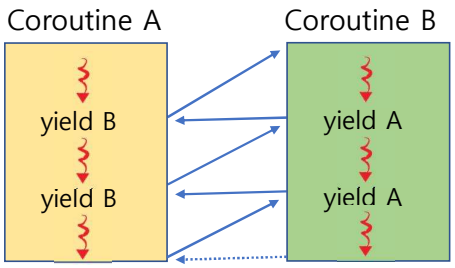
A main routine and subroutines

vs.

Coroutines, which call on each other

Cooperative multitasking
(aka, Non-preemptive multitasking)






# Coroutines

- goroutines
- fibers
- green threads
- virtual threads
- generators


# History of Coroutines

- Term coined in 1958 by Melvin Conway
- 1960s – Simula
- 1970s – Modula
- 2000s – C# Generators, D, Python, Kotlin, Scala
- 2010s – C#, Javascript, Rust, Go
- 2020s – C++, Java



# Stackless vs. Stackful Coroutines

**Stackless** (async/await-like)

- C++
- C#
- JavaScript
- Python
- Rust
- Swift
- Kotlin

```
function g() { return 0; };

async function f() {
  let val = g();
  await sleep_async(10);
  return val;
}
```

**Stackful** (Fibers or Green Threads)

- Erlang
- Go
- Lua
- Scheme
- Java

```
var fiber = Fiber.new {
  (1..10).map { |i|
    // Wren can yield from inside this block.
    Fiber.yield(i)
  }
}
```

# 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!

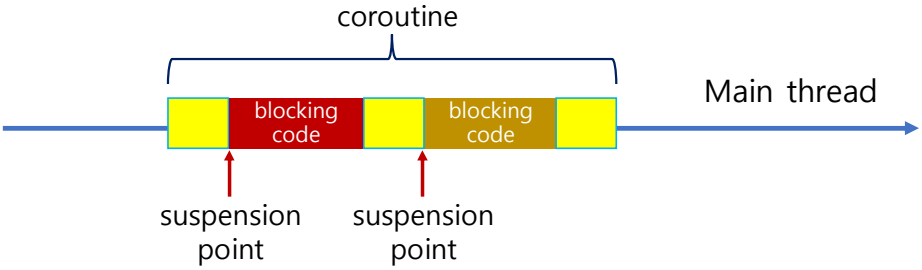


# What is a Coroutine? (Warning: My Definition)

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

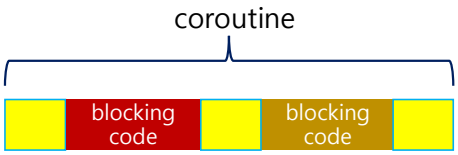
- Thread calling those blocking functions cannot do anything else
  - it cannot execute other requests,
  - it cannot process UI events.
- You should avoid blocking
  - limited request-processing threads in backend application, or
  - main UI thr

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

A coroutine is *a sequence of computations*,
each of which may *suspend* now and *resume* at some point,
*without blocking the thread* that executes it.

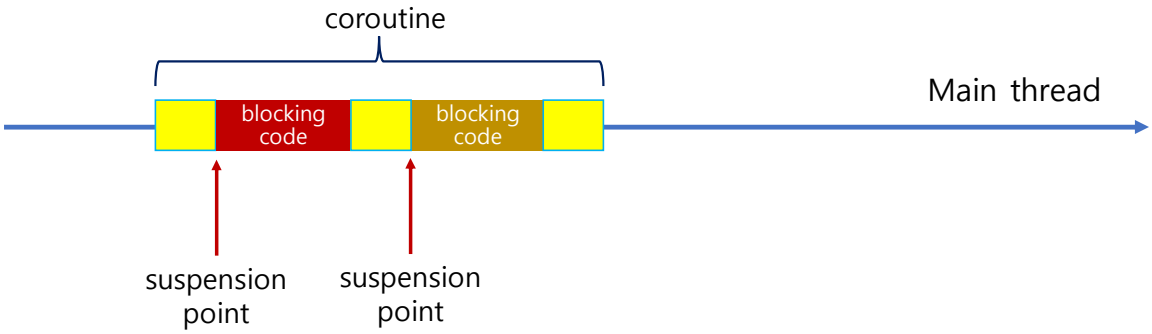



- Coroutines provide an *alternative to thread blocking* by supporting ***suspension***.
- So, what is the difference between ***blocking*** a thread and ***suspending*** a coroutine?

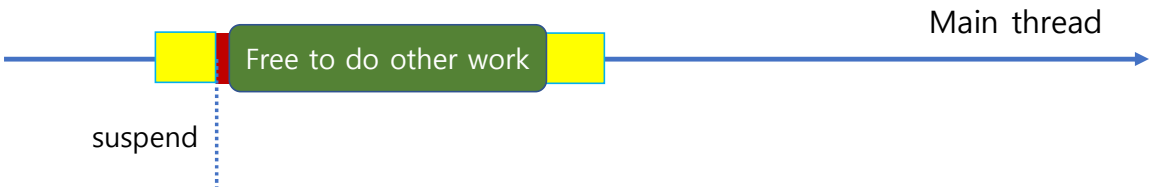
```
val data = awaitData() // does it block or suspend?
processData(data)
```



# Coroutine is a non-blocking suspend computation

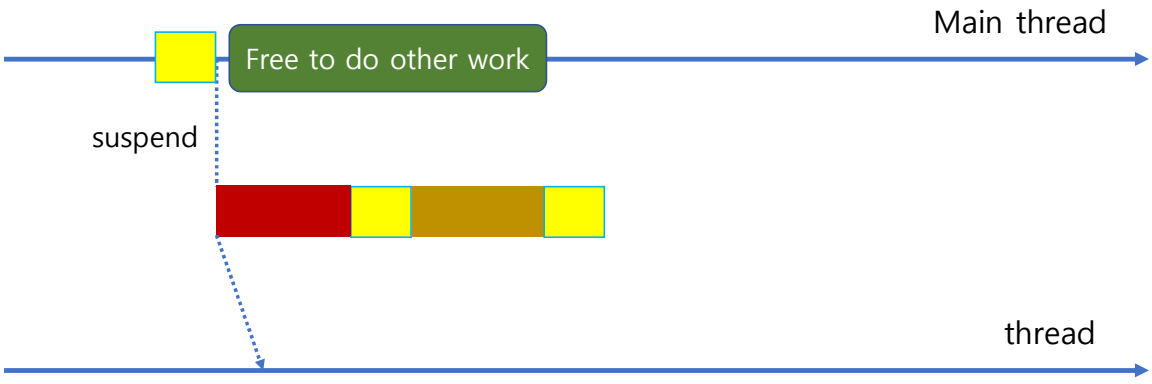


# Coroutine is a non-blocking suspend computation

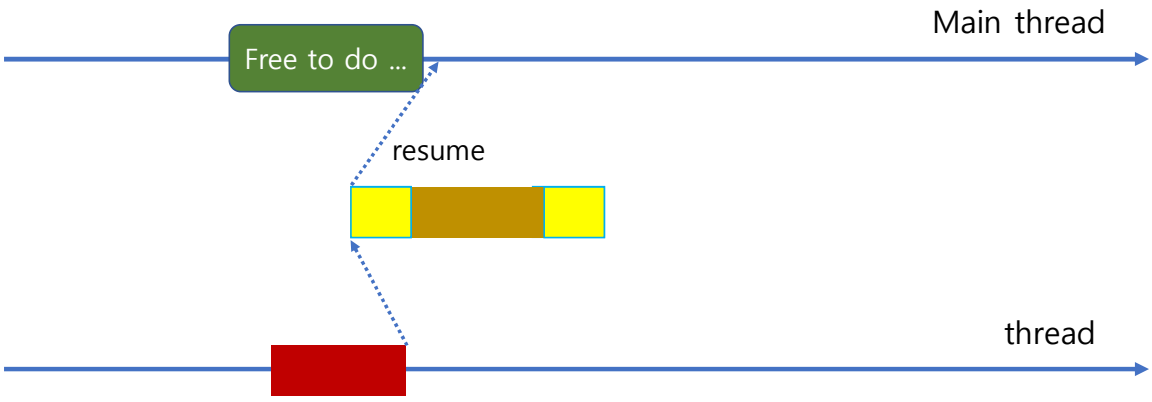





# Coroutine is a non-blocking suspend computation

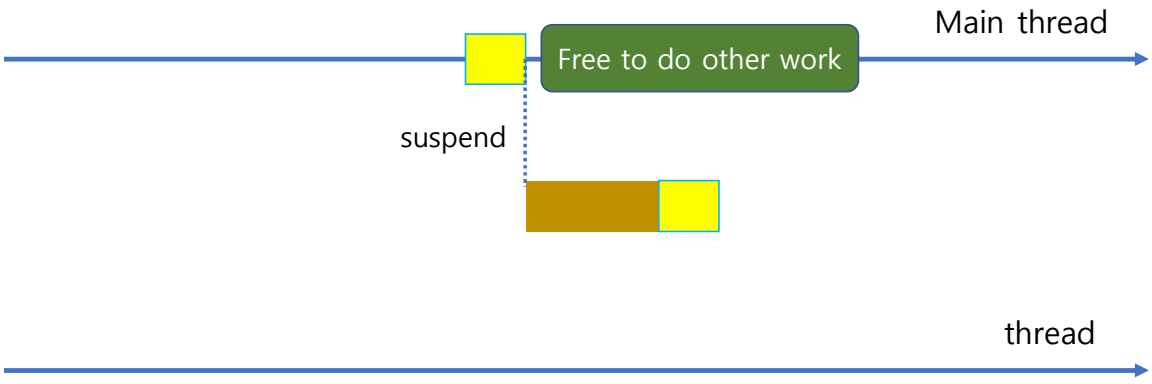


# Coroutine is a non-blocking suspend computation

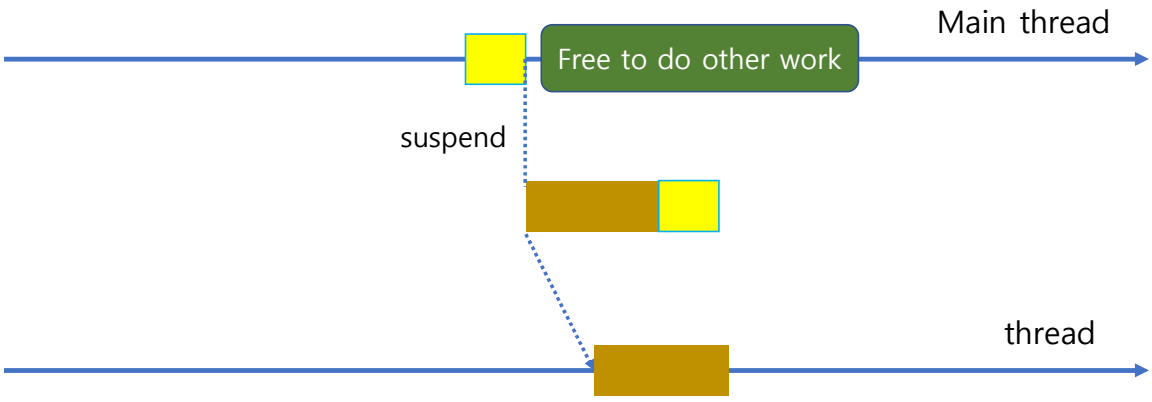





# Coroutine is a non-blocking suspend computation

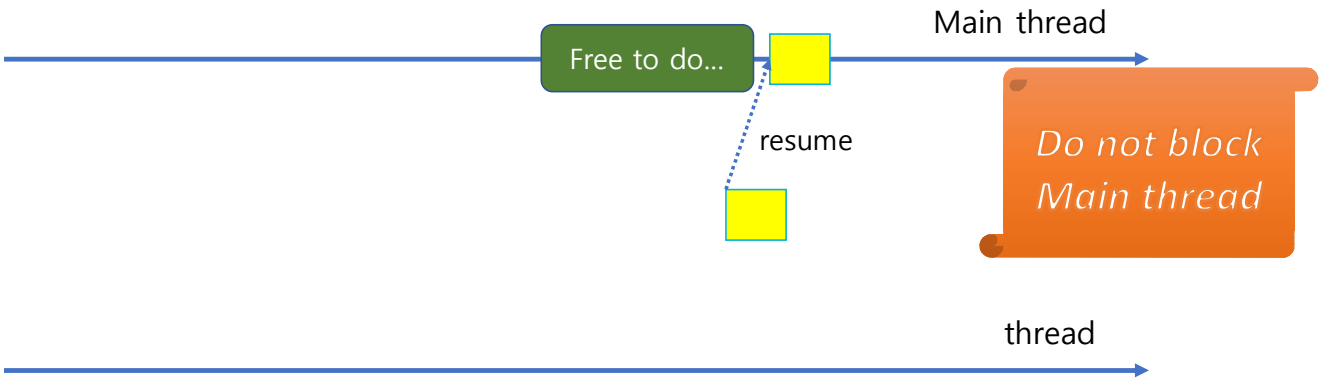


# Coroutine is a non-blocking suspend computation

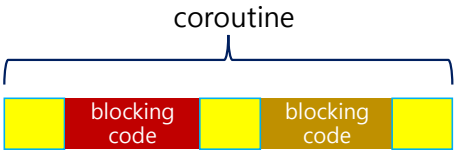





# Coroutine is a non-blocking suspend computation



# 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.

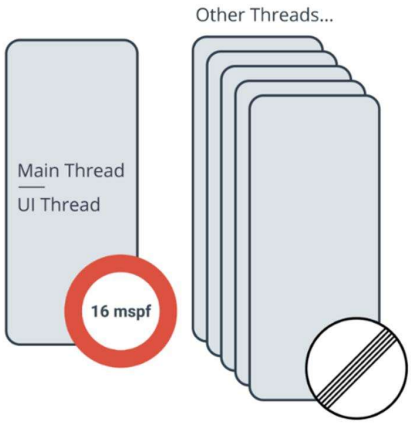
⚠ *One mistake that is often made is that adding a `suspend` modifier to a function makes it either asynchronous or non-blocking.*

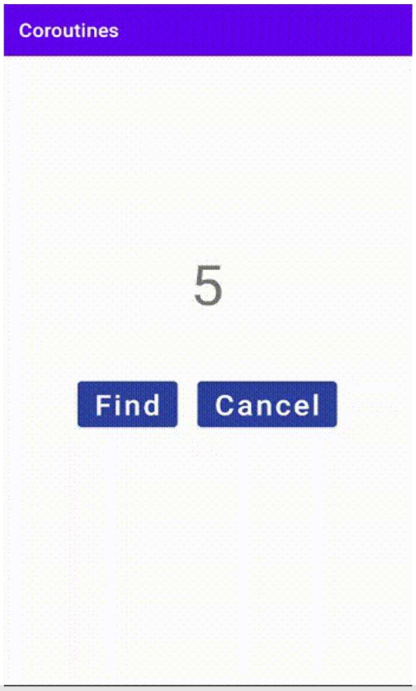
```
suspend fun findBigPrime(): BigInteger =
       BigInteger.probablePrime(4096, Random())
```



# Why Coroutines in Android?

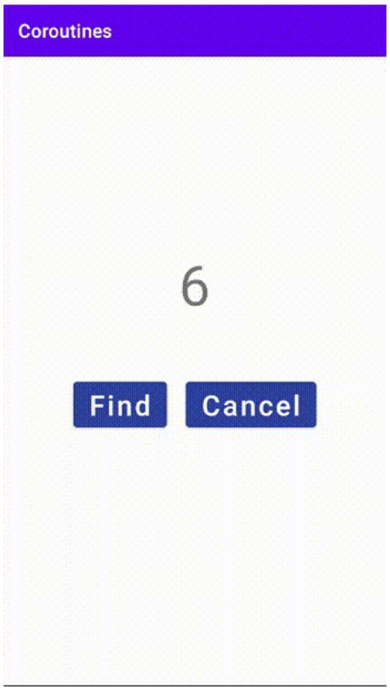
- On Android, the *main thread* (aka *UI thread*) is a single *default thread* that handles:
  - all updates to the UI.
  - calls all click handlers and other UI and lifecycle callbacks
- Without explicit thread switching, everything app does is on the main thread.
- Blocking in this context means the 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

✅ Responsiveness
✅ Performance




# Challenges of Asynchrony

- Race Conditions
- Back Pressure
- Leaked Resources
- Threading
  - Expensive
  - Starvation
  - Deadlocks

and more …

# 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

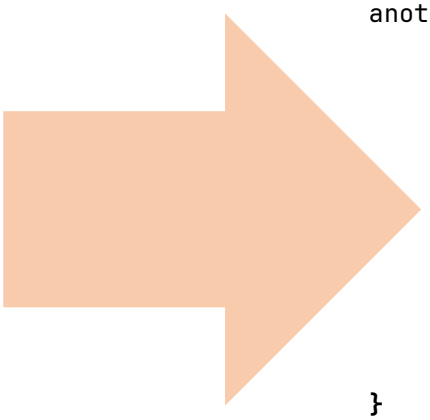
```
fun postItem(item: Item) {
    requestToken { token ->
        createPost(token, item) { post ->
            showPost(post)
        }
    }
}
```

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())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe { post ->
            showPost(post)
        }
}
```


No nesting indentation


Composable & propagates exceptions


Library-specific operators

Looks complicated …
Steep learning curve!



# Synchronous *vs.* …

```
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
}
```

# 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 continue

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)
}
```

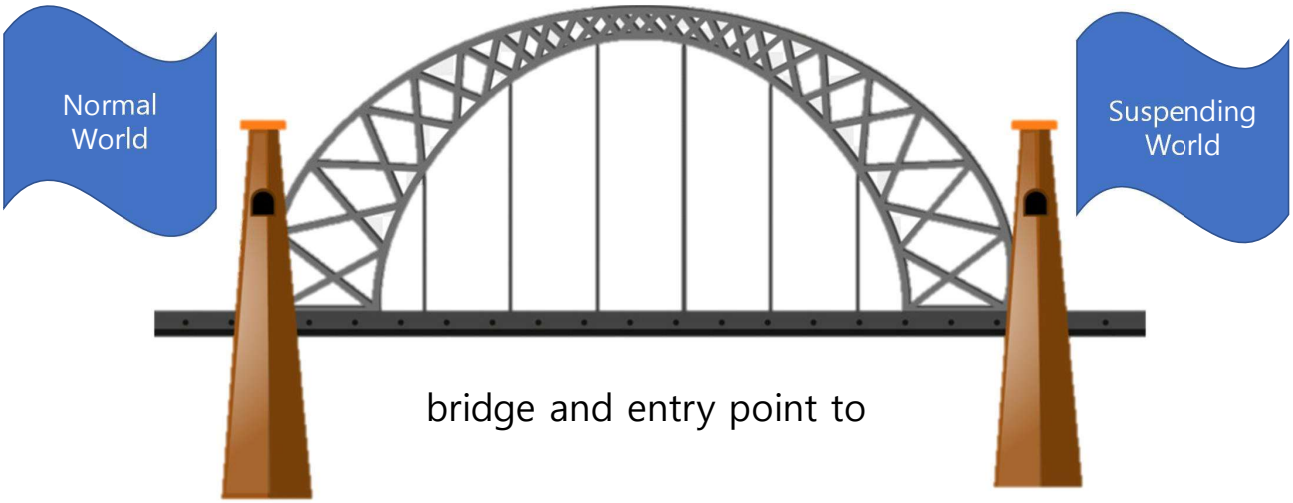


# What if to call Suspending Functions from Normal Functions?

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

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

# Coroutine Builders are bridges between …



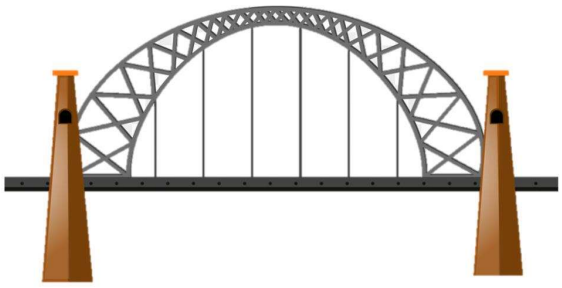



# Bridging the normal world and the suspending world

- *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



# 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!**



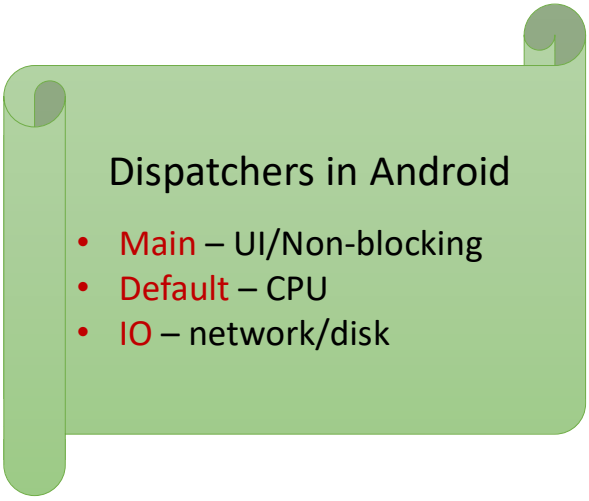
# Sneak Preview of CoroutineScope

- 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` *(← suspending 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

DEFAULT
LAZY
ATOMIC
UNDISPATCHED

```
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

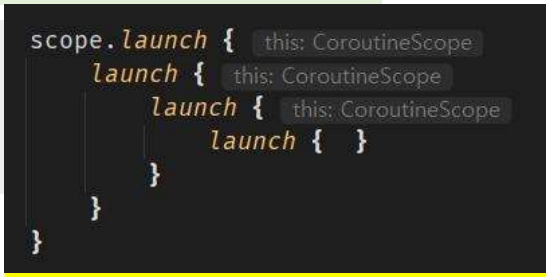
```
suspend fun loadAndCombine(name1: String, name2: String): Image {
    val deferred1 = GlobalScope.async { loadImage(name1) }
    val deferred2 = GlobalScope.async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

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)



# runBlocking

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

- Block the current thread until the suspending lambda finishes executing.

```
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
}
```

# runBlocking (Cont'd)

- 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.

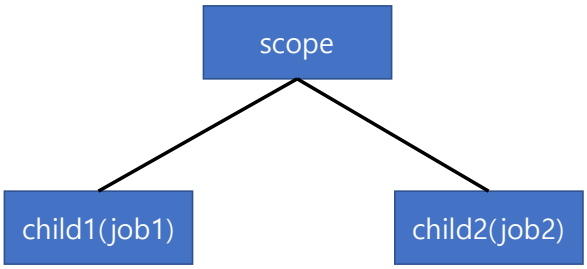
```
// Create a coroutine, and block the main thread until it completes
fun main() = runBlocking {
    println("Hello,")
    delay(2000L) // suspends the current coroutine for 2 seconds
    println("World!") // will be executed after 2 seconds
}
```

- `runBlocking` is very *<u>useful in tests</u>*, you can wrap your tests in `runBlocking`.
  - This will make sure your *<u>test code execute sequentially on the same thread</u>* and will *<u>not terminate until all coroutines are completed</u>*.



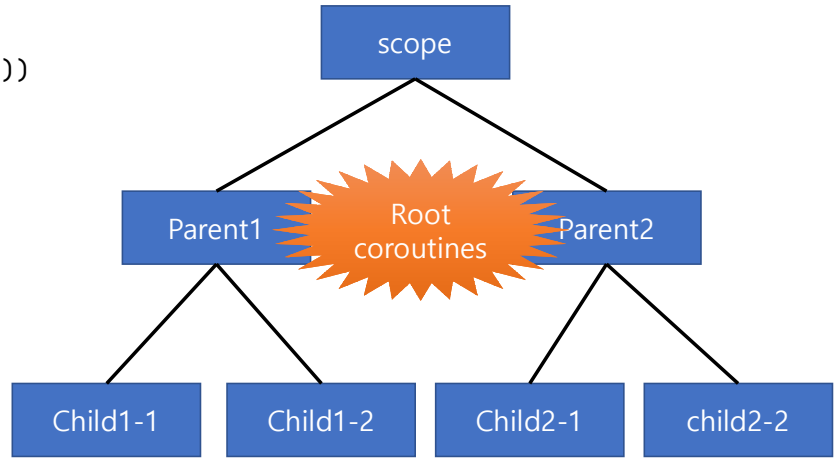
# Coroutines form a hierarchy

```
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)
```




Root coroutines *vs.* Top-level coroutines



# 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.

*Solution?*

# 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 { loadImage(name1) }
    val deferred2 = async { loadImage(name2) }
    return combineImages(deferred1.await(), deferred2.await())
}
```

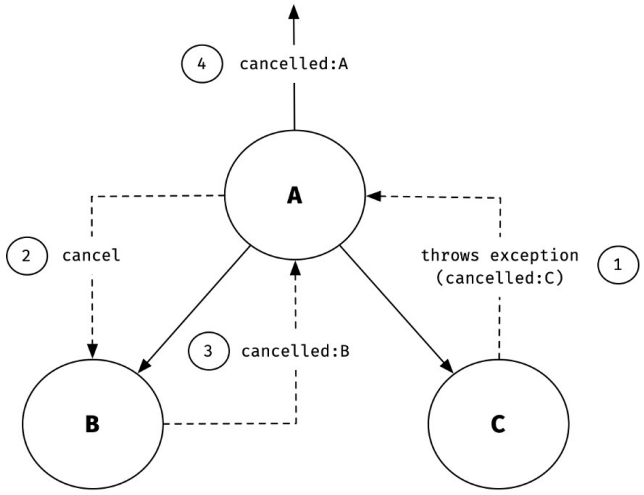
- The solution was to write `async(coroutineContext){…}` so that loading of both images is performed in children coroutines that are cancelled when their parent coroutine is cancelled.
- But, what happened when the first `loadImage` fails?
  - Then `deferred1.await()` throws the corresponding exception, 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 *must 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 (in *completed* or *cancelled* state).
4. Cancelling a parent or failure (with its own exceptions) will cancels all its children.
5. *Cancelling a child won't cancel* the parent and its siblings.
6. *Failure of a child cancels the parent and all of its siblings*, unless its parent has `SupervisorJob` (which is a special type of `Job`).



# Proper Example of Parallel Decomposition

(more on later …)

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 { // or supervisorScope
  val deferred1 = async { loadImage(name1) }
  val deferred2 = async { loadImage(name2) }
  return combineImages(deferred1.await(), deferred2.await())
  }
}
```