goto; copenhagen

GOTO Copenhagen 2018
Conference Nov. 19 - 21

Fresh Async with Kotlin

Roman Elizarov



Follow us @gotocph

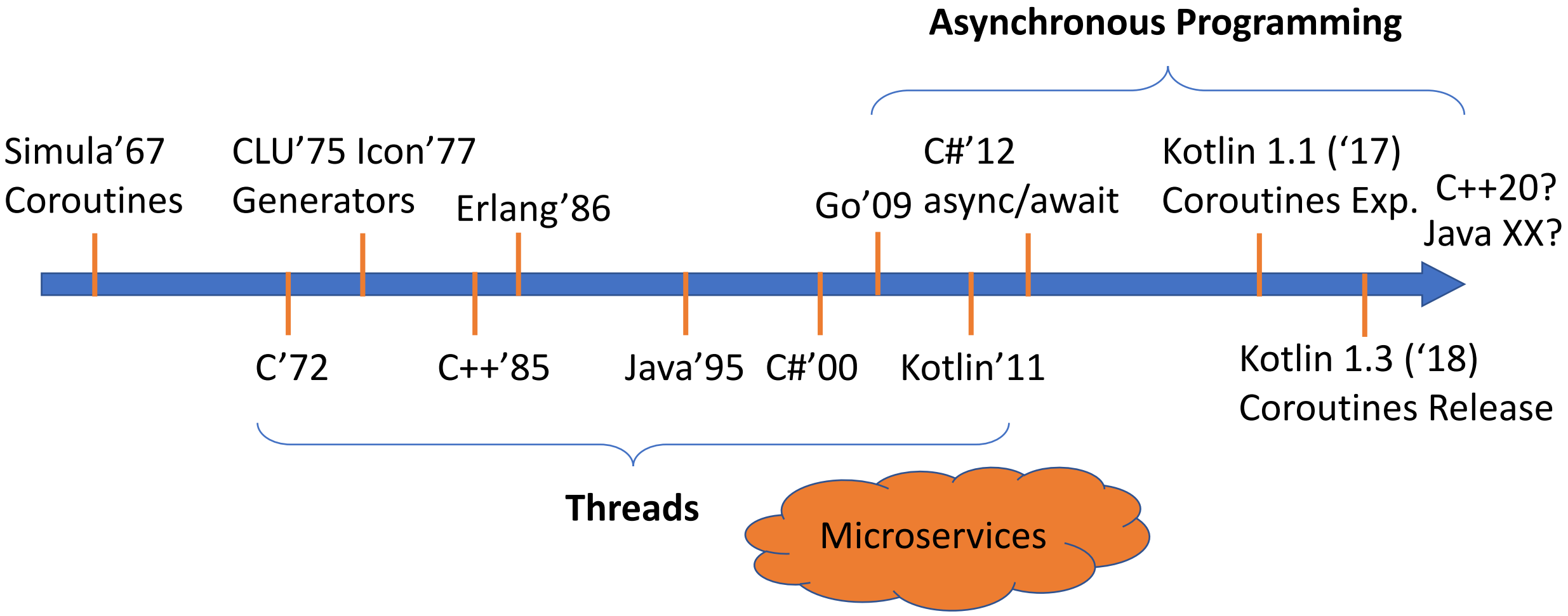
Speaker: Roman Elizarov



- Professional developer since 2000
- Previously developed high-perf trading software
  @ Devexperts
- Teach concurrent & distributed programming
  @ St. Petersburg ITMO University
- Chief judge
  @ Northern Eurasia Contest / ACM ICPC
- Now team lead in Kotlin Libraries
  @ JetBrains



History


Asynchronous Programming
Simula'67
Coroutines
CLU'75 Icon'77
Generators
Erlang'86
C#'12
Go'09 async/await
Kotlin 1.1 ('17)
Coroutines Exp.
C++20?
Java XX?
C'72
C++'85
Java'95
C#'00
Kotlin'11
Kotlin 1.3 ('18)
Coroutines Release
Threads
Microservices

Asynchronous Programming with **Callbacks**



Callbacks: before

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

Callbacks: after

callback

1
```
fun requestTokenAsync(cb: (Token) -> Unit) {
    // makes request for a token, invokes callback when done
    // returns immediately
}
```

Callbacks: before

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPost(token: Token, item: Item): Post {
    // sends item to the server & waits
    return post // returns resulting post
}
```

2

Callbacks: after

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) {
    // sends item to the server, invokes callback when done
    // returns immediately
}
```

2

callback

Callbacks: before

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) { … }
fun processPost(post: Post) { … }
```

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

Callbacks: after

```
fun requestTokenAsync(cb: (Token) -> Unit) { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) { … }
fun processPost(post: Post) { … }
```

This is simplified. Handling exceptions makes it a real mess

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

aka “callback hell”

Futures/Promises
make it nicer



Futures: before

```
1 fun requestTokenAsync(cb: (Token) -> Unit) {
      // makes request for a token, invokes callback when done
      // returns immediately
  }
```

Futures: after

future

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

Futures: before

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item,
                    cb: (Post) -> Unit) {
    // sends item to the server, invokes callback when done
    // returns immediately
}
```

Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> {
    // sends item to the server
    // returns promise for a future result immediately
}
```

2

future

Futures: before

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

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

Futures: after

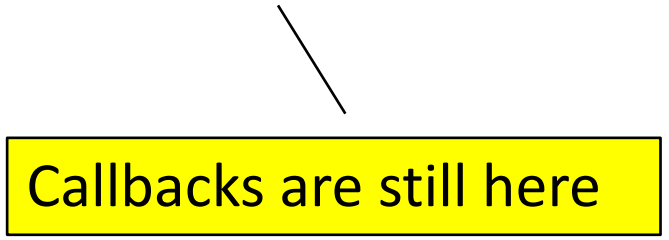
```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```



Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Composable & propagates exceptions

No nesting indentation

Callbacks are still here

Futures: after

```
fun requestTokenAsync(): Promise<Token> { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> …
fun processPost(post: Post) { … }
```

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

But all those operators…

Kotlin coroutines to the rescue



Coroutines: before

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

Coroutines: after

natural signature

1
```
suspend fun requestToken(): Token {
    // makes request for a token & suspends
    return token // returns result when received
}
```

Coroutines: before

```
suspend fun requestToken(): Token { … }
fun createPostAsync(token: Token, item: Item): Promise<Post> {
    // sends item to the server
    // returns promise for a future result immediately
}
```

Coroutines: after

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

2

natural signature

Coroutines: before

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

```
fun postItem(item: Item) {
    requestTokenAsync()
        .thenCompose { token -> createPostAsync(token, item) }
        .thenAccept { post -> processPost(post) }
}
```

Coroutines: after

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

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

Coroutines: after

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

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

Like *regular* code

Coroutines: after

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

suspension points

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

Bonus features

- *Regular* loops



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

Bonus features

- *Regular* exception handing

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

Bonus features

- *Regular* higher-order functions



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

Bonus features

- *Custom* higher-order functions

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

Everything like in regular code



How does it work?

A quick peek behind the scenes

Kotlin suspending functions

Kotlin

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



Java/JVM

callback

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

Kotlin suspending functions

Kotlin

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



Java/JVM

callback

```
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

Kotlin suspending functions

Kotlin

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



Java/JVM

callback

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

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

Kotlin suspending functions

Kotlin

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



Java/JVM

callback

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

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

Integration

Zoo of futures on JVM



Retrofit async

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}
```

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}

suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

natural signature

```
interface Service {
    fun createPost(token: Token, item: Item): Call<Post>
}

suspend fun createPost(token: Token, item: Item): Post =
    serviceInstance.createPost(token, item).await()
```

Suspending extension function from integration library

```
suspend fun <T> Call<T>.await(): T {
    …
}
```

Callbacks everywhere

```
suspend fun <T> Call<T>.await(): T {
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            // todo
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            // todo
        }
    })
}
```

```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

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

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

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

Regular function

Inspired by **call/cc** from **Scheme**



```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Install callback

```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Install callback

```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Analyze response

```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```

Analyze response

```
suspend fun <T> 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(ErrorResponse(response))
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            cont.resumeWithException(t)
        }
    })
}
```



Coroutine builders



Coroutines revisited

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

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

Coroutines revisited

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

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

Coroutines revisited

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

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

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

Coroutines revisited

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

Can *suspend* execution

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

Coroutines revisited

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

A regular function *cannot*

Can *suspend* execution

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

Coroutines revisited

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

A regular function *cannot*

Can *suspend* execution

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



One cannot simply invoke a suspending function

Launch

coroutine builder

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



Fire and forget!

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



Fire and forget!

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

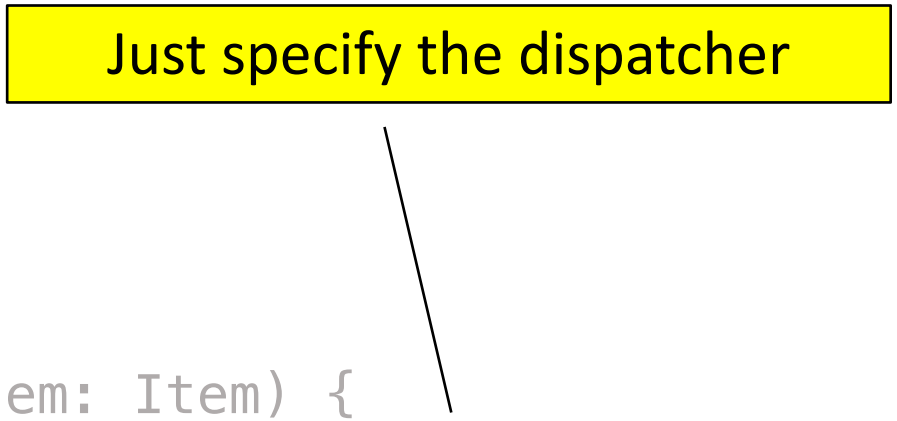
We launch for its **side effects**

Returns immediately, coroutine works in **background thread pool**

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

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

Dispatcher




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

Dispatcher

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

And it gets executed on the Main thread

Where’s the magic of launch?

A regular function

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

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

suspending lambda

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

async / await

The classic approach

Kotlin-way

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

Kotlin

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

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

C# approach to the same problem (also Python, TS, Dart, coming to JS)

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

mark with async

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

use await to suspend

Classic-way

```
async Task<Token> requestToken() { … }
async Task<Post> createPost(Token token, Item item) { … }
void processPost(Post post) { … }
```

returns a future

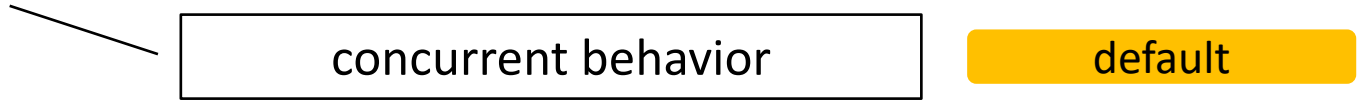
C#

```
async Task postItem(Item item) {
    var token = await requestToken();
    var post = await createPost(token, item);
    processPost(post);
}
```

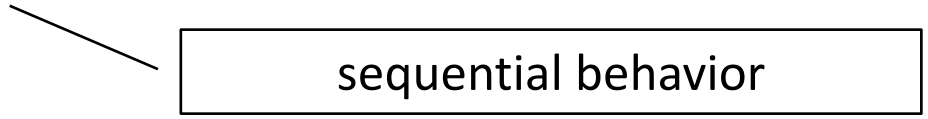
Why no **await** keyword in Kotlin?

The problem with async

C# `requestToken()` **VALID** –> produces `Task<Token>`



C# `await requestToken()` **VALID** –> produces `Token`



Kotlin **suspending functions** are designed to imitate sequential behavior *by default*



Concurrency is hard
Concurrency has to be explicit

Kotlin approach to async

Concurrency where you need it

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

```
var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);
```

Start multiple operations **concurrently**

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

```
var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);

var image1 = await promise1;
var image2 = await promise2;
```

and then wait for them

Use-case for async

C#

```
async Task<Image> loadImageAsync(String name) { … }
```

```
var promise1 = loadImageAsync(name1);
var promise2 = loadImageAsync(name2);

var image1 = await promise1;
var image2 = await promise2;

var result = combineImages(image1, image2);
```

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

Kotlin async function

A regular function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

Kotlin async function

Kotlin’s future type

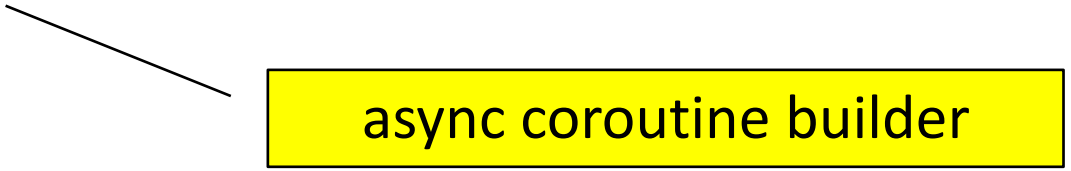
Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

Kotlin async function

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```



Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

```
val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)
```

Start multiple operations **concurrently**

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }


val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)


val image1 = deferred1.await()
val image2 = deferred2.await()
```

and then wait for them

await *function*

Suspends until deferred is complete

Crash?

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

```
val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)

val image1 = deferred1.await()
val image2 = deferred2.await()

val result = combineImages(image1, image2)
```

But what if this crashes?

Crash?

Kotlin

```
fun loadImageAsync(name: String): Deferred<Image> =
    GlobalScope.async { … }
```

```
val deferred1 = loadImageAsync(name1)
val deferred2 = loadImageAsync(name2)

val image1 = deferred1.await()
val image2 = deferred2.await()

val result = combineImages(image1, image2)
```

This one *leaks*

But what if this crashes?

Idiomatic: Using async function when needed

Is defined as suspending function, not async

```
suspend fun loadImage(name: String): Image { … }
```

Idiomatic: Using async function when needed

```
suspend fun loadImage(name: String): Image { … }
```

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

Structured concurrency

Idiomatic: Using async function when needed

```
suspend fun loadImage(name: String): Image { … }


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

Idiomatic: Using async function when needed

```
suspend fun loadImage(name: String): Image { … }


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

Idiomatic: Using async function when needed

```
suspend fun loadImage(name: String): Image { … }


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

Kotlin approach to async

Kotlin

```
requestToken()
```

VALID –> produces `Token`

sequential behavior

default

Kotlin

```
async { requestToken() }
```

VALID –> produces `Deferred<Token>`

concurrent behavior

Structured concurrency

Structured concurrency

```
suspend fun loadImage(name: String): Image { … }
```

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

Crash

```
suspend fun loadImage(name: String): Image { … }
```

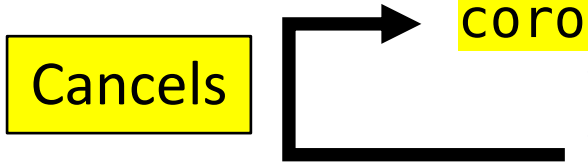
```
suspend fun loadAndCombine(name1: String, name2: String): Image =
    coroutineScope {  this: CoroutineScope
        val deferred1 = async { loadImage(name1) }
        val deferred2 = async { loadImage(name2) }
        combineImages(deferred1.await(), deferred2.await())
    }
```

Crashes?

Cancels scope

```
suspend fun loadImage(name: String): Image { … }
```

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



Cancels children

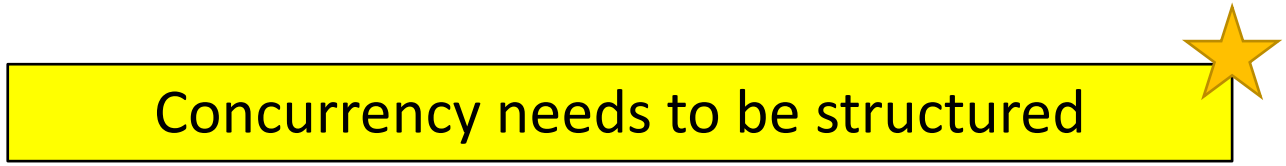
```
suspend fun loadImage(name: String): Image { … }
```

Cancels



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

Kotlin **suspending functions**
may use concurrency internally,
but don’t leak it




Parallelism is optional

Beyond asynchronous code

Kotlin’s approach to generate/yield – synchronous coroutines

Fibonacci sequence

```
val fibonacci: Sequence<Int> = …
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

println(fibonacci.take(10).toList())
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

println(fibonacci.take(10).toList())

>> [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Fibonacci sequence

A coroutine builder with restricted suspension

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

A suspending function in the scope of **buildSequence**

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

println(fibonacci.take(10).toList())
```



Fibonacci sequence

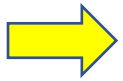
```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
```

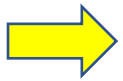
Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```



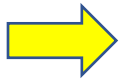
Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```

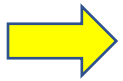
Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```

Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next())
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

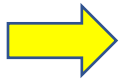
val iter = fibonacci.iterator()
println(iter.next()) // 1
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

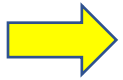
Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

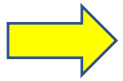
Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

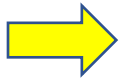
Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

Fibonacci sequence



```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next())
```

Fibonacci sequence

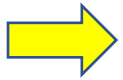
```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
```

Fibonacci sequence

```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

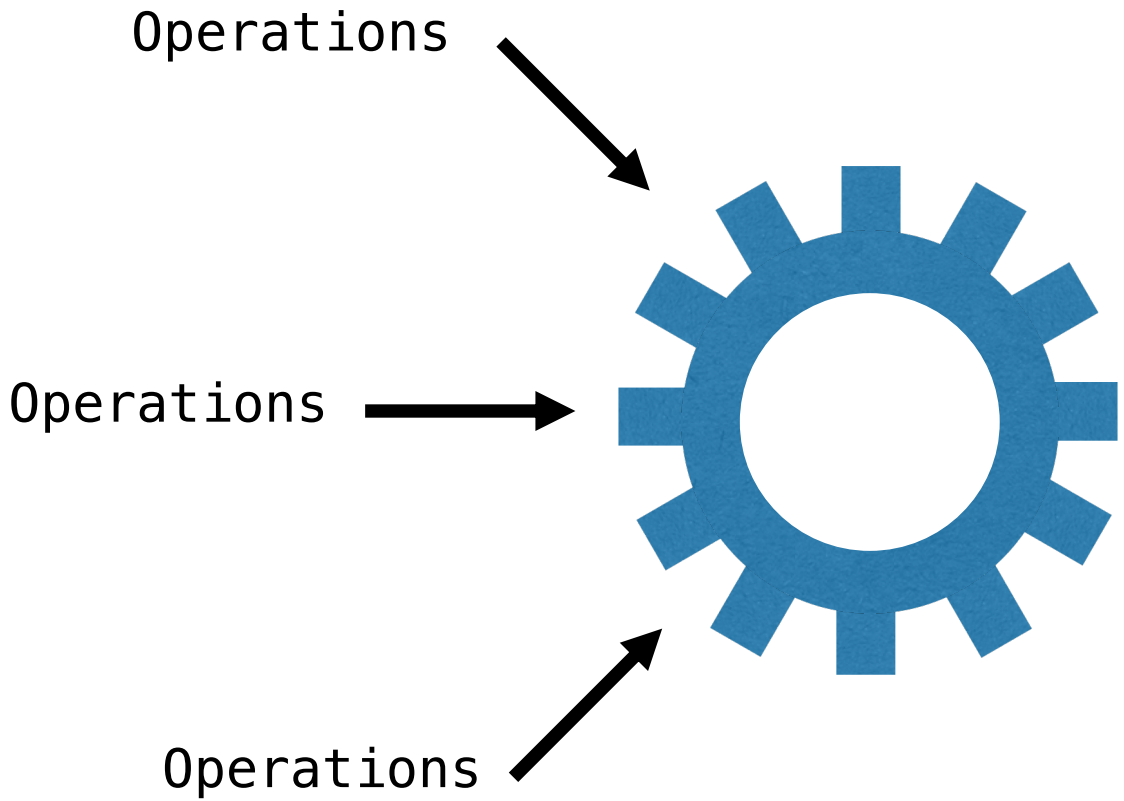
val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
```



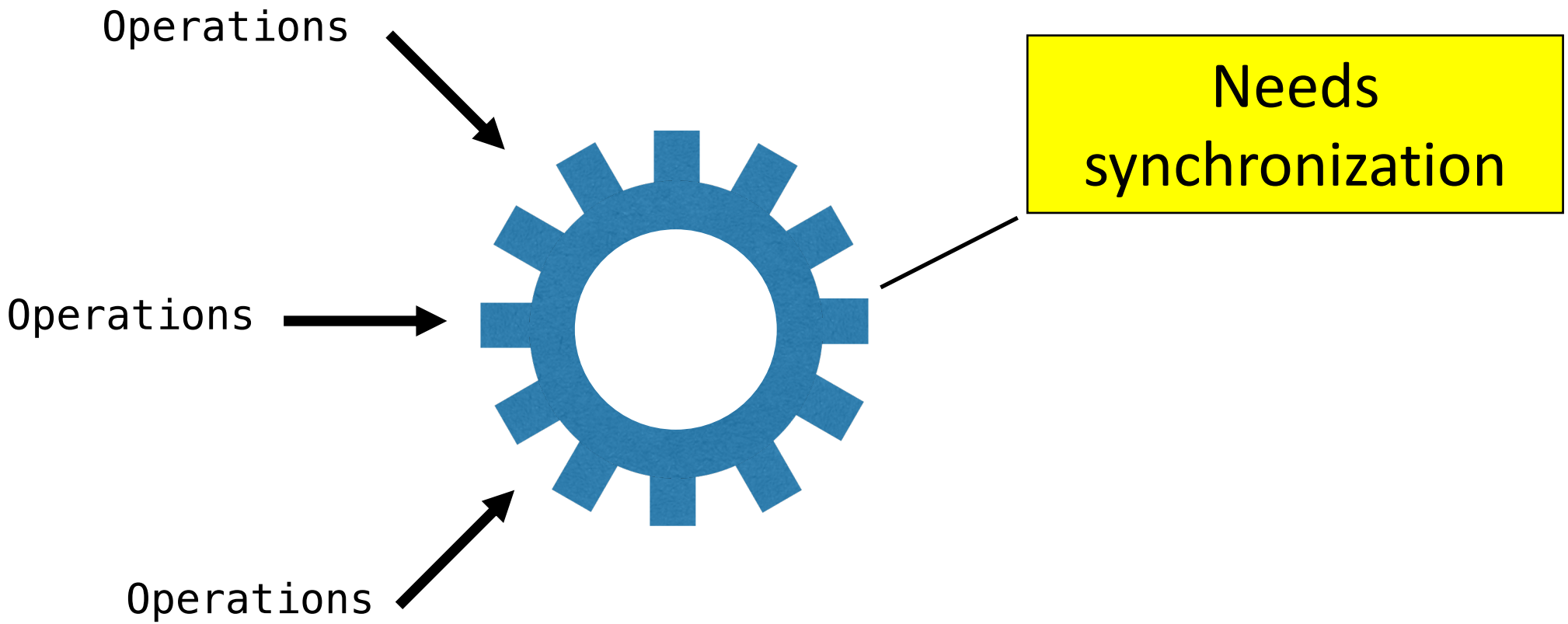
etc ad infinum

Sharing state?

Shared Mutable State



Shared Mutable State



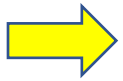
Shared Mutable State



Communicating
Sequential
Processes

Concurrent Fibonacci sequence

```
val fibonacci = GlobalScope.produce {
    var cur = 1
    var next = 1
    while (true) {
        send(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```



Asynchronous
(can be in different thread)

Concurrent Fibonacci sequence

```
val fibonacci = GlobalScope.produce {
    var cur = 1
    var next = 1
    while (true) {
        send(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive())
}
```

Asynchronous
(can be in different thread)

Another coroutine

Concurrent Fibonacci sequence

```
val fibonacci = GlobalScope.produce {
    var cur = 1
    var next = 1
    while (true) {
        send(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive())
}
```

Asynchronous
(can be in different thread)

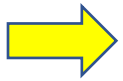
Another coroutine

Receives from **channel**

Concurrent Fibonacci sequence

```
val fibonacci = GlobalScope.produce {
    var cur = 1
    var next = 1
    while (true) {
        send(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

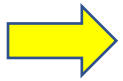
fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive()) // 1
}
```



Concurrent Fibonacci sequence

```
val fibonacci = GlobalScope.produce {
    var cur = 1
    var next = 1
    while (true) {
        send(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}

fun main(args: Array<String>) = runBlocking {
    println(fibonacci.receive()) // 1
    println(fibonacci.receive()) // 1
}
```



etc ad infinum

Library vs Language

Keeping the core language small

Classic async

async/await
generate/yield

} Keywords

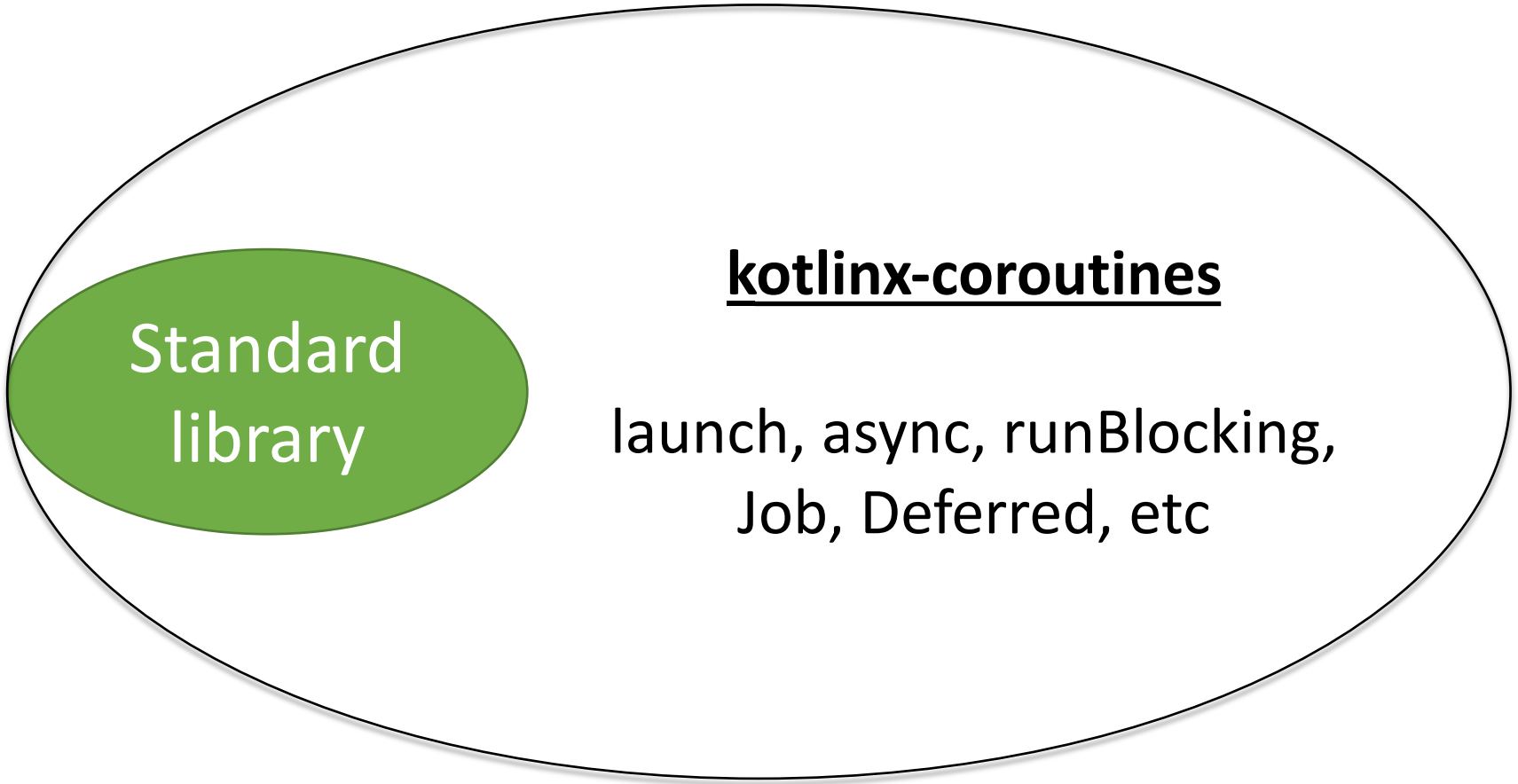
Kotlin coroutines

suspend } Modifier

Kotlin coroutines



Kotlin coroutines



http://github.com/kotlin/kotlinx.coroutines

https://kotlinlang.org

KotlinConf 2018 Watch Opening Keynote

Kotlin

LEARN COMMUNITY TRY ONLINE

Don't block
Keep moving

COROUTINES



Learn More ⟶

Kotlin 1.3 Released

Thank you

Any questions?



Roman Elizarov

elizarov @ relizarov

goto; copenhagen

GOTO Copenhagen 2018

Conference Nov. 19 - 21





Click ‘Rate Session’ to rate session and ask questions.



Follow us @gotocph