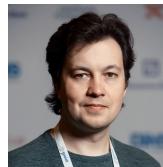


Kotlin Coroutines Workshop



Roman Elizarov

 elizarov @
relizarov

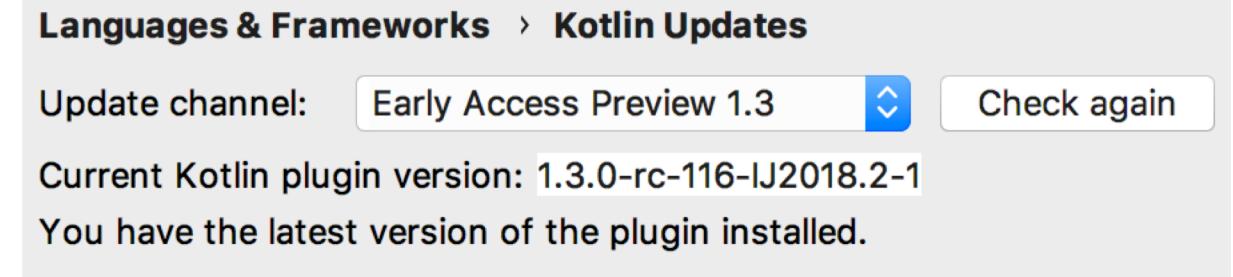


Prerequisites (download)

- IntelliJ IDEA: 2018.1, 2018.2, or 2018.3 (EAP)

- Kotlin Plugin: 1.3-rc-116 (EAP)

➤ Tools | Kotlin | Configure
Kotlin Plugin Updates



- GitHub Project:

```
$ git clone https://github.com/elizarov/CoroutinesWorkshop
```

Outline

1. Asynchronous programming, Introduction to Coroutines
2. Coroutines vs Threads, Context and Cancellation
3. Coroutines and Concurrency
4. CSP with Channels and Actors

Part 1:

Asynchronous programming

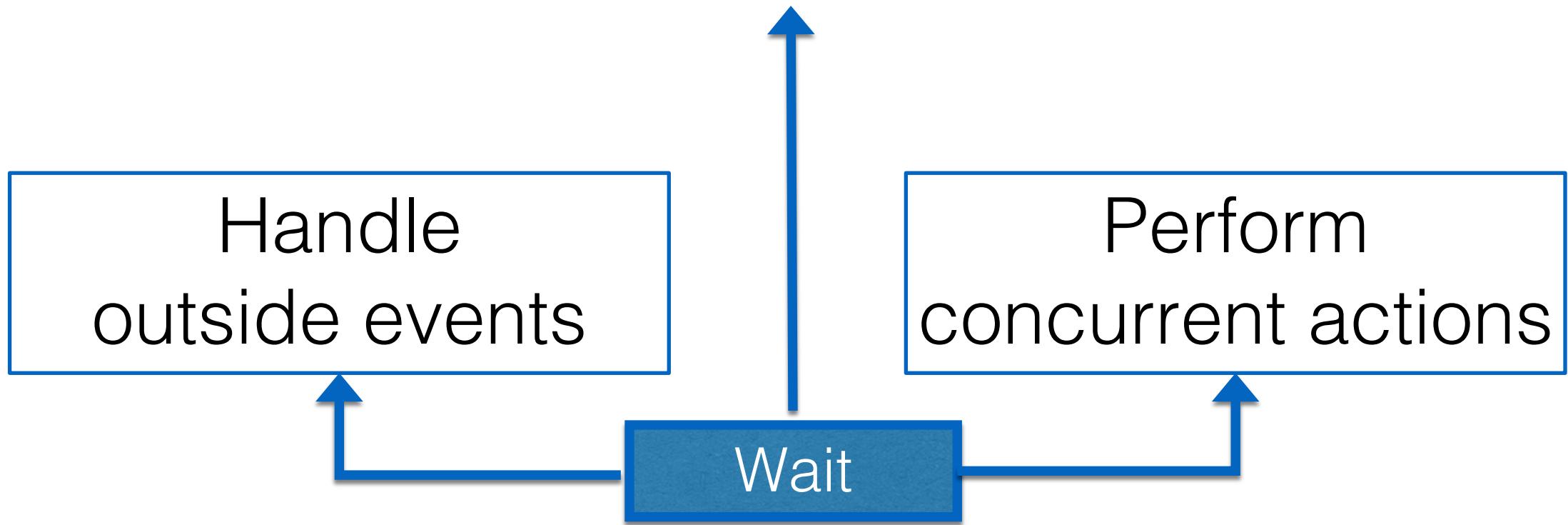
“Asynchrony, refers to the occurrence of events independent of the main program flow and ways to deal with such events. These may be "outside" events such as the arrival of signals, or actions instigated by a program that take place concurrently with program execution, without the program *blocking* to wait for results.”

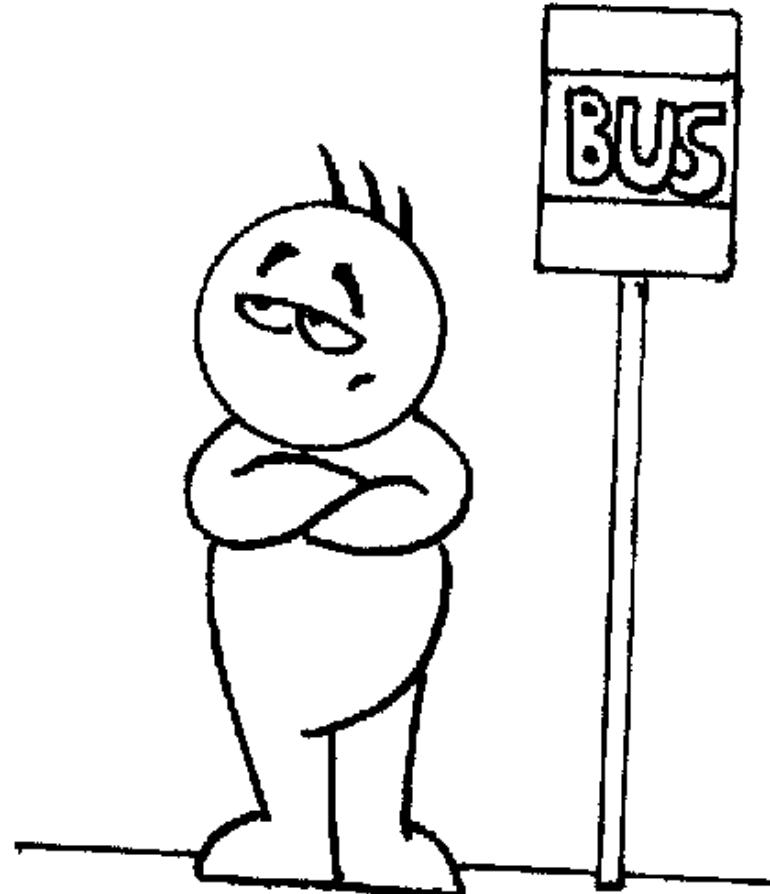
[https://en.wikipedia.org/wiki/Asynchrony_\(computer_programming\)](https://en.wikipedia.org/wiki/Asynchrony_(computer_programming))

Asynchronous programming

==

without blocking





Writing code that waits

Programming styles

1. Blocking threads
2. Callbacks
3. Futures
4. Kotlin coroutines

A toy problem

Kotlin

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

A toy problem

Kotlin

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

A toy problem

Kotlin

```
fun requestToken(): Token { ... }
fun createPost(token: Token, item: Item): Post { ... }
3 fun processPost(post: Post) {
    // does some local processing of result
}
```

A toy problem

Kotlin

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

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

Threads

Threads

```
fun requestToken(): Token {  
    // makes request for a token  
    // blocks the thread waiting for result  
    return token // returns result when received  
}  
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)  
}
```

A problem with threads?

How many threads can we have?

100 😊

How many threads can we have?

1000 

How many threads can we have?

10 000 😞

How many threads can we have?

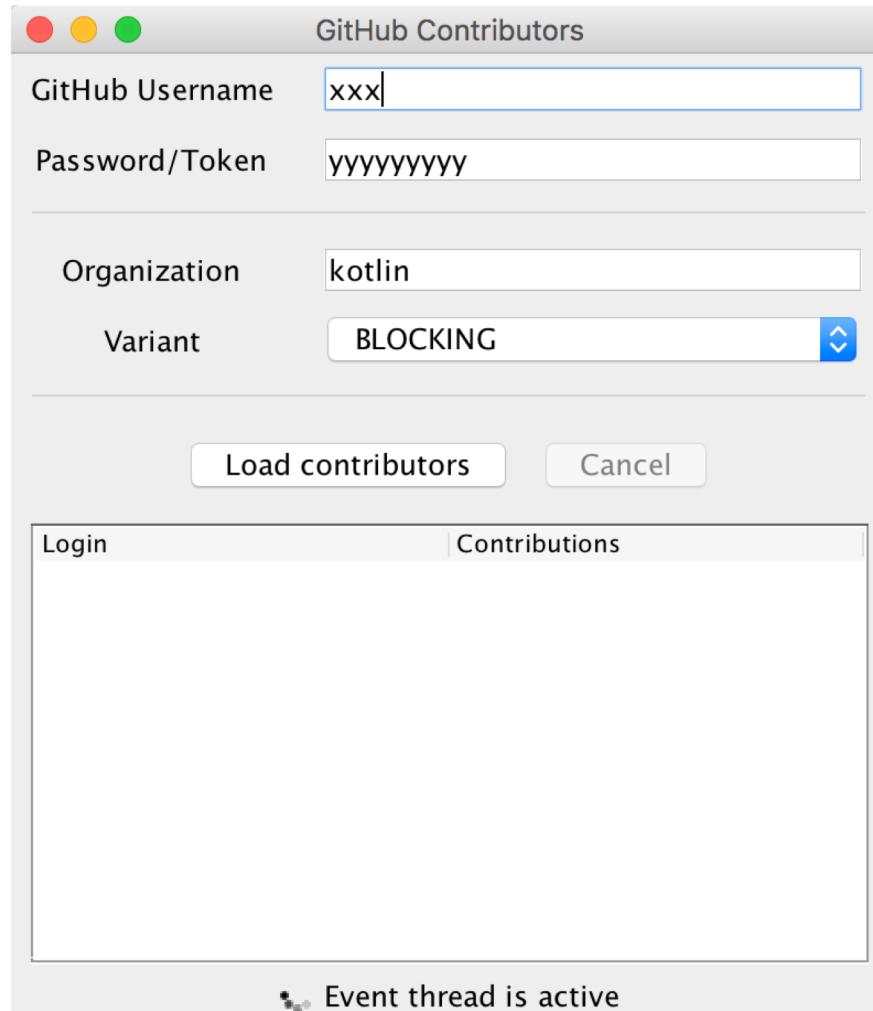
100 000 

Real need for async when ...

- ✓ You cannot afford threads
 - They are expensive to keep and to switch
 - High-load server-side, micro-tasks, etc
- ✓ You cannot do threads
 - Your code is single thread only
 - JS/web target, µC, etc
- ✓ You do not want threads
 - You have lot of mutable state
 - Mobile/Desktop UI apps, etc

Project intro

GitHub Contributors



src/project/ContributorsUI.kt

```
▶ fun main() {
    ... setDefaultFontSize(18f)
    ... ContributorsUI().apply { this: ContributorsUI
        ... pack()
        ... setLocationRelativeTo(null)
        isVisible = true
    }
}
```

<https://github.com/settings/tokens>

OAuth Apps

GitHub Apps

Personal access tokens

Generate new token

Variant

BLOCKING



```
fun loadContributorsBlocking(req: RequestData) : List<User>
```



Exercise: Implement aggregate

```
fun List<User>.aggregate(): List<User> = ...
```

- File **src/project/Util.kt**
- Result list must have aggregated (sum) contributions per login
- Sorted in descending order by no. of contributions

Hint: answer is in the **instructor** branch

Hands on!

Exercise: Implement aggregate

```
fun List<User>.aggregate(): List<User> =  
    groupingBy { it.login }  
        .reduce { login, a, b ->  
            User(login, a.contributions + b.contributions) }  
        .values  
        .sortedByDescending { it.contributions }
```

Answer

Variant

BACKGROUND



fun loadContributorsBackground(...)



Demo

Callbacks

Callbacks: before

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

Callbacks: after

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

callback

Continuation Passing Style

“In functional programming, continuation-passing style (CPS) is a style of programming in which control is passed explicitly in the form of a continuation. This is contrasted with *direct style*, which is the usual style of programming...

A function written in continuation-passing style takes an extra argument: an explicit “*continuation*”, i.e. a function of one argument. When the CPS function has computed its result value, it “returns” it by calling the continuation function with this value as the argument.”

Callbacks: before

```
fun requestToken(): Token { ... }

② fun createPost(token: Token, item: Item): Post {
    // sends item to the server & waits
    return post // returns resulting post
}
```

Direct Style

Callbacks: after

```
fun requestTokenAsync(cb: (Token) -> Unit) { ... }

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

Continuation Passing Style

Callbacks: before

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

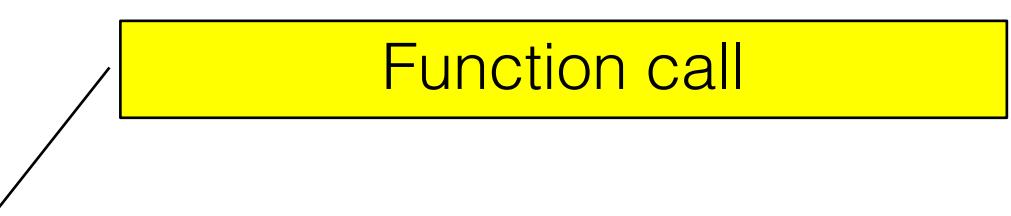
```
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```

Direct Style

Callbacks: before

```
fun requestToken(): Token { ... }  
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)  
}
```



A yellow callout box with a black border and white text is positioned above the line 'val token = *requestToken()*'. A thin black line extends from the top right corner of the box to the word 'requestToken()' in the code.

Function call

Callbacks: before

```
fun requestToken(): Token { ... }  
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)  
}
```

Syntactic continuation

Callbacks: after

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

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

Callbacks: after

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

Function call

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

Callbacks: after

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

Explicit continuation

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

```
1  function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SCRIPTS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });
23     });
24   );
25 };
26 }
```



Project: Callbacks

Variant

CALLBACKS



fun loadContributorsCallbacks(...)



Callbacks with retrofit

```
inline fun <T> Call<T>.responseCallback(crossinline callback: (T) -> Unit) {
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            checkResponse(response)
            callback(response.body()!!)
        }

        override fun onFailure(call: Call<T>, t: Throwable) {
            log.error("Call failed", t)
        }
    })
}
```

Exercise: Implement with callbacks

- File **src/project/Request3Callbacks.kt**
- Write code based on **loadContributorsBlocking**
- Use helper extension **Call.responseCallback** to install callbacks
- Has to keep the sequential logic of the code

Hint: use local function to implement loop

Hands on!

Futures

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

Futures

“In computer science, **future**, **promise**, **delay**, and **deferred** refer to constructs used for synchronizing program execution in some concurrent programming languages. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete.”

Futures: Encapsulate callback

```
callback  
requestTokenAsync({ token ->  
    doSomething()  
})
```

Futures: Encapsulate callback

```
future          callback  
requestTokenAsync().thenAccept({ token ->  
    doSomething()  
})
```

Futures: before

```
fun requestTokenAsync(cb: (Token) -> Unit) { ... }

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

Futures: after

```
fun requestTokenAsync(): CompletableFuture<Token> { ... }  
    future  
② fun createPostAsync(token: Token, item: Item): CF<Post> {  
    // sends item to the server  
    // returns promise for a future result immediately  
}
```

Futures: Compose

```
class CompletableFuture<T> {
    // Install “terminal” callback
    fun thenAccept(action: (T) -> Unit): CF<Unit>
}
```

Futures: Compose

```
class CompletableFuture<T> {
    // Install “terminal” callback
    fun thenAccept(action: (T) -> Unit): CF<Unit>

    // Transform and return another future
    fun <U> thenCompose(transform: (T) -> CF<U>): CF<U>
}
```

Futures: Compose

```
class CompletableFuture<T> {
    // Install “terminal” callback
    fun thenAccept(action: (T) -> Unit): CF<Unit>

    // Transform and return another future
    fun <U> thenCompose(transform: (T) -> CF<U>): CF<U>

    // ... and lots of other combinators ...
}
```

Futures: before

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

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

Futures: after

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

Composable &
propagates exceptions

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

No nesting indentation

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 combinator...

Coroutines

Coroutines

“**Coroutines** are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for *suspending* and resuming execution at certain locations.”

Coroutines: before

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

Coroutines: after

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

Coroutines: after

natural signature

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

Suspending function

“A suspending function — a function that is marked with **suspend** modifier. It may suspend execution of the code without blocking the current thread of execution by invoking other suspending functions.”

Coroutines: before

```
fun requestTokenAsync(): CompletableFuture<Token> { ... }

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

Coroutines: after

```
suspend fun requestToken(): Token { ... }
```

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

Coroutines: before

```
fun requestTokenAsync(): CF<Token> { ... }
fun createPostAsync(token: Token, item: Item): CF<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)  
}
```



Direct 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

- *Direct* loops

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

Bonus features

- *Direct* exception handing

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

Bonus features

- *Direct* higher-order functions

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

- `foreach`, `let`, `apply`, `repeat`, `filter`, `map`, `use`, *etc*

Everything like in blocking code



Behind the scenes

Kotlin suspending functions

Kotlin

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

CPS Transformation

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

Java/JVM

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

CPS Transformation

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

Java/JVM

callback

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

Continuation

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

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

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

Continuation

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

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

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

Continuation is a generic callback interface



Coroutines (like futures!) use callbacks at their low level, but allow asynchronous programming in direct style

Integration

Zoo of futures on JVM

Legacy strikes back

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

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

serviceInstance.createPost(token, item).**await()**

```
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
```

Regular function

Inspired by **call/cc** from Scheme



Call with Current Continuation

```
val post = suspendCoroutine<Post> { cont: Continuation<Post> ->  
    // ... Block of code that is invoked with current continuation  
}  
doSomething(post)
```

Syntactic continuation

Reified continuation

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)
        }
    })
}
```

That's all 

Project: Coroutines

Variant

COROUTINE

```
suspend fun loadContributors(req: RequestData) : List<User>
```



Exercise: Implement with coroutine

Variant

COROUTINE



- File **src/project/Request4Coroutine.kt**
- Write code based on **loadContributorsBlocking**
 - File **src/project/Request1Blocking.kt**
- Use helper extension **Call.await** to suspend
- Keep the logic of the code

Hands on!

A note on style

- Mark functions with **suspend** that are supposed to wait for asynchronous events
- Suspending functions should not block

Coroutine builders

```
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)
}
```

```
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)
}
```

```
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

```
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)  
}
```

```
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)  
}
```

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

A regular function *cannot*

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

Can *suspend* execution



One cannot simply invoke
a suspending function

Launch

coroutine builder

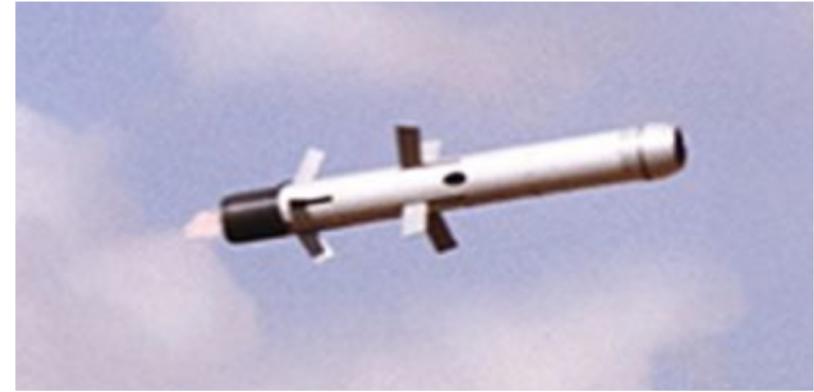
```
fun postItem(item: Item) {  
    GlobalScope.launch {  
        val token = requestToken()  
        val post = createPost(token, item)  
        processPost(post)  
    }  
}
```

```
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!

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

Just specify the 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 { ... }
```

Project: Progress

Variant

COROUTINE

listOrgRepos

listRepoContributors1

listRepoContributors2

...

listRepoContributorsN

aggregate

update UI

Variant

PROGRESS



listOrgRepos

listRepoContributors1

aggregate

update UI

listRepoContributors2

aggregate

update UI

...

Demo!

Part 2:

Coroutines vs

Threads

Coroutines

What are coroutines conceptually?

Demo: Threads

```
 fun main() {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo

Demo: Threads

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

```
fun main() {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo: Threads (before)

```
fun main() {
    val jobs = List(100_000) {
        thread {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo: Coroutines (after)

```
fun main() {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo

Demo: Coroutines (after)

```
fun main() {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Suspend function 'join' should be called only from a coroutine or another suspend function

Demo: Coroutines

```
fun main() {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    //jobs.forEach { it.join() }
}
```

Demo

Coroutines are like *daemon*
threads

Demo: Run blocking

This coroutine builder runs coroutine
in the context of invoker thread

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".)
        }
    }
    jobs.forEach { it.join() }
}
```

Demo: Run blocking

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

We can join a job
just like a thread,
but *suspending*

Demo: Run blocking

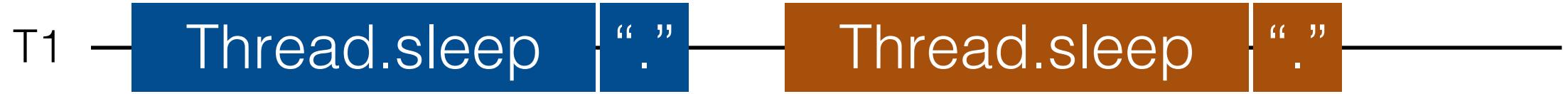
```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo

Demo: Run blocking

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            Thread.sleep(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Blocks underlying threads



Coroutines are *backed* by
some (few) threads

Demo: delay

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            delay(5000) ←
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```



Suspends for 5 seconds

Demo: delay

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            delay(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo

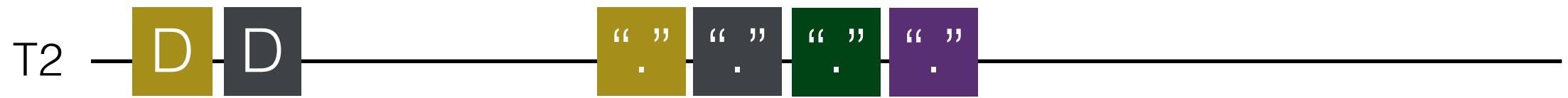
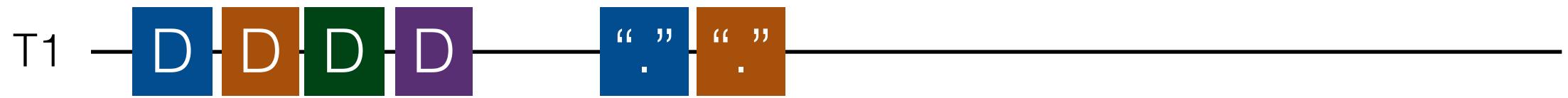
Demo: delay

```
fun main() = runBlocking<Unit> {
    val jobs = List(100_000) {
        GlobalScope.launch {
            delay(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Prints 100k dots after 5 second delay



Coroutines are like
very light-weight threads



Demo: delay

```
suspend fun main() {
    val jobs = List(100_000) {
        GlobalScope.launch {
            delay(5000)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```

Demo

Coroutine context

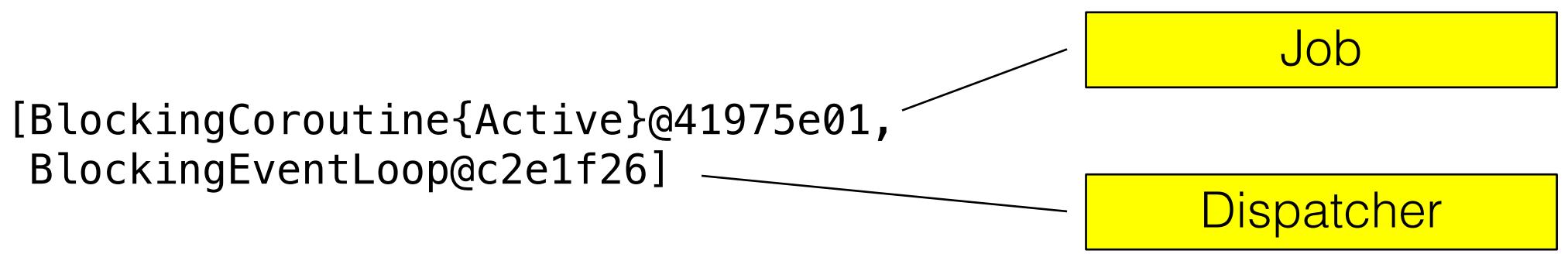
Demo: coroutine context

```
fun main() = runBlocking<Unit> {
    val ctx: CoroutineContext = coroutineContext
    println(ctx)
}
```

Demo: coroutine context

```
fun main() = runBlocking<Unit> {
    val ctx: CoroutineContext = coroutineContext
    println(ctx)
}
```

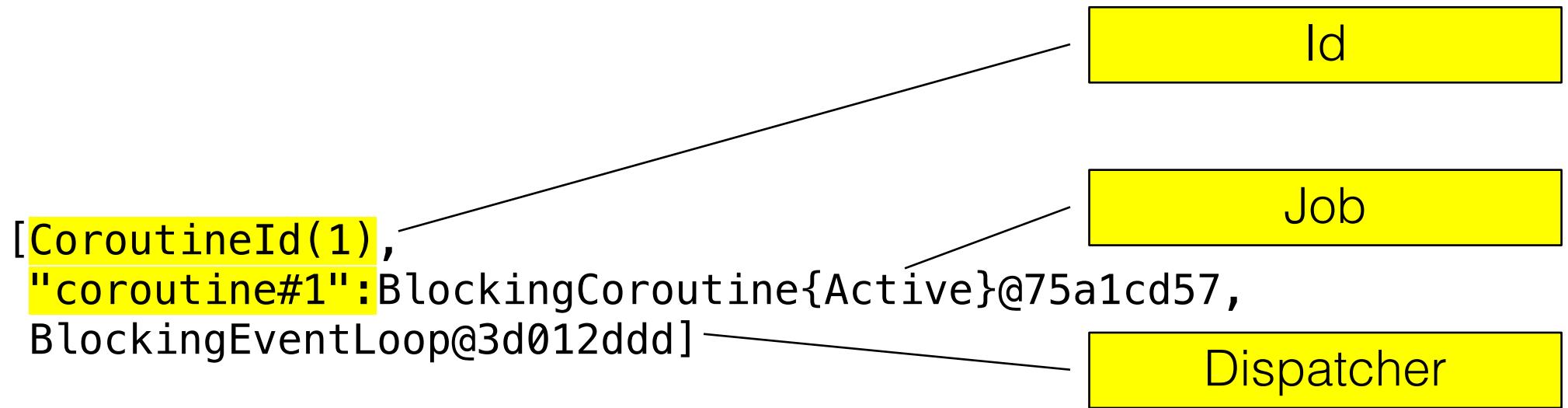
Demo



Coroutine context is an
immutable set of context
elements

| | |
|------------------------|---|
| Main class: | <input type="text" value="part2blocking.exerciseB.Context1ShowKt"/> ... |
| VM options: | <input type="text" value="-ea"/> ... |
| Program arguments: | <input type="text"/> |
| Working directory: | <input type="text"/> ... |
| Environment variables: | <input type="text"/> ... |

Demo



In *debug mode* a unique
coroutine identifier is
generated

Demo: coroutine context

```
fun main() = runBlocking<Unit> {
    val job = GlobalScope.launch {
        val ctx: CoroutineContext = coroutineContext
        println(ctx)
    }
    job.join()
}
```

Demo

```
[CoroutineId(2),  
 "coroutine#2":StandaloneCoroutine{Active}@45007883,  
 DefaultDispatcher]
```

```
graph LR; A["[CoroutineId(2),\n\"coroutine#2\":StandaloneCoroutine{Active}@45007883,\nDefaultDispatcher]"] --> B[Dispatcher]
```

Dispatcher

Demo: inherited context

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val job = launch {
        val ctx: CoroutineContext = coroutineContext
        println(ctx)
    }
    job.join()
}
```

Demo

```
[CoroutineId(2),  
 "coroutine#2":StandaloneCoroutine{Active}@45007883,  
 BlockingEventLoop@1a93a7ca]
```

Inherited
Dispatcher

Structured concurrency

Demo: Structured concurrency

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    val job = launch {
        val ctx: CoroutineContext = coroutineContext
        println(ctx)
    }
    // job.join()
}
```

Demo

Demo: Structured concurrency

```
fun main() = runBlocking<Unit> {    this: CoroutineScope
    println(coroutineContext)
    val job = launch {
        println(coroutineContext)
    }
}
```

```
[BlockingCoroutine{Active}@e2144e4, BlockingEventLoop@6477463f]
[StandaloneCoroutine{Active}@f2a0b8e, BlockingEventLoop@6477463f]
```

Demo: Structured concurrency

Parent

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    println(coroutineContext)
    val job = launch {
        println(coroutineContext)
    }
}
```

Demo: Structured concurrency

```
fun main() = runBlocking<Unit> { this: CoroutineScope  
    println(coroutineContext)  
    val job = launch {  
        println(coroutineContext)  
    }  
}
```

Parent

Child

CoroutineScope

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

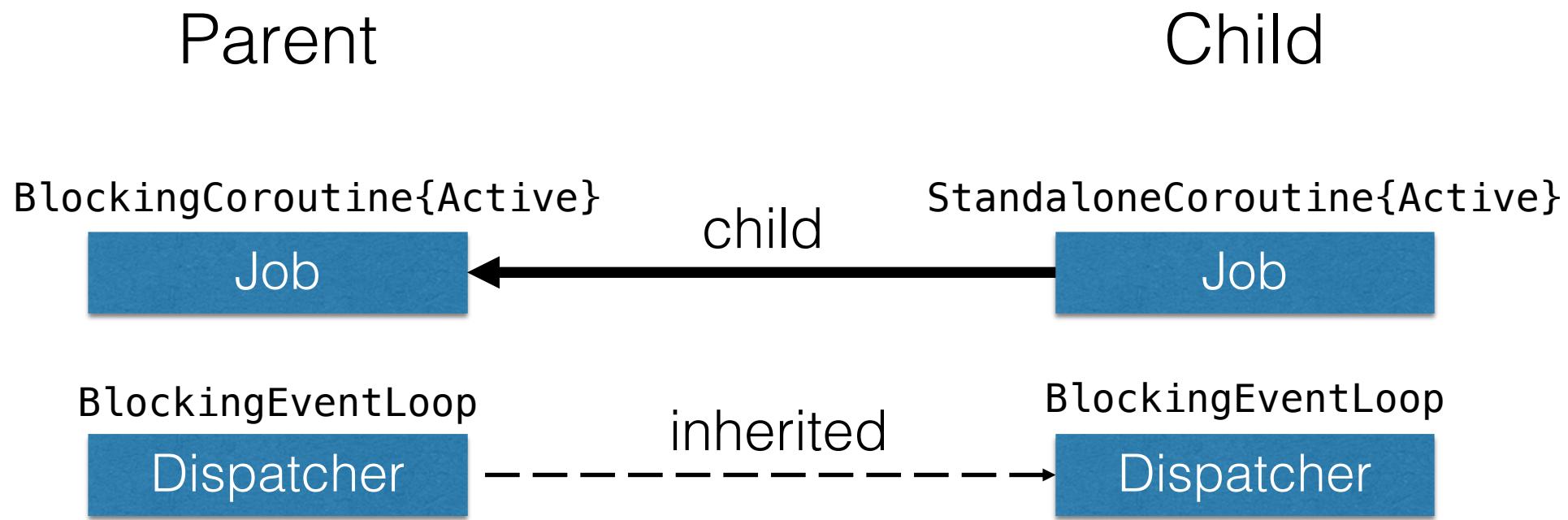
Parent

BlockingCoroutine{Active}

Job

BlockingEventLoop

Dispatcher



Dispatcher

Dispatchers

```
fun display(name: String) =  
    println("launch($name) in ${Thread.currentThread().name}")  
  
fun main() = runBlocking<Unit> {  
    launch(Dispatchers.Default) { display("DefaultDispatcher") }  
    launch(Dispatchers.Swing) { display("Swing") }  
    launch { display("inherited") }  
}
```

Demo

Dispatchers

```
fun display(name: String) =  
    println("launch($name) in ${Thread.currentThread().name}")  
  
fun main() = runBlocking<Unit> {  
    launch(Dispatchers.Default) { display("DefaultDispatcher") }  
    launch(Dispatchers.Swing) { display("Swing") }  
    launch { display("inherited") }  
}
```

```
launch(DefaultDispatcher) in DefaultDispatcher-worker-1  
launch(Swing) in AWT-EventQueue-0  
launch(inherited) in main
```

Dispatchers

```
fun display(name: String) =  
    println("launch($name) in ${Thread.currentThread().name}")  
  
fun main() = runBlocking<Unit> {  
    launch(Dispatchers.Default) { display("DefaultDispatcher") }  
    launch(Dispatchers.Swing) { display("Swing") }  
    launch { display("inherited") }  
}
```

```
launch(DefaultDispatcher) in DefaultDispatcher-worker-1  
launch(Swing) in AWT-EventQueue-0  
launch(coroutineContext) in main
```

Unconfined

```
fun main() = runBlocking<Unit> {
    launch { display("inherited") }
    launch(Dispatchers.Unconfined) { display("Unconfined") }
}
```

Demo

Unconfined

```
fun main() = runBlocking<Unit> {
    launch { display("inherited") }
    launch(Dispatchers.Unconfined) { display("Unconfined") }
}
```

```
launch(Unconfined) in main
launch(inherited) in main
```

Unconfined + delay

```
suspend fun displayDelay(name: String) {  
    display(name)  
    delay(1000)  
    display(name)  
}  
  
fun main() = runBlocking<Unit> {  
    launch { displayDelay("inherited") }  
    launch(Dispatchers.Unconfined) { displayDelay("Unconfined") }  
}
```

Demo

Unconfined + delay

```
suspend fun displayDelay(name: String) {  
    display(name)  
    delay(1000)  
    display(name)  
}  
  
fun main() = runBlocking {  
    val jobs = mutableListOf<Job>()  
    jobs += launch(coroutineContext) { displayDelay("coroutineContext") }  
    jobs += launch(Unconfined) { displayDelay("Unconfined") }  
    launch(Unconfined) in main  
    launch(inherited) in main  
    launch(Unconfined) in kotlinx.coroutines.DefaultExecutor  
    launch(inherited) in main
```

Unconfined dispatcher
does not constrain the
thread the the coroutine
resumes on

Explore unconfined in
Contributors project

Try it

Combining contexts

Demo: combine context

```
fun main() = runBlocking<Unit> {
    val ctx: CoroutineContext = coroutineContext
    println(ctx)
}
```

[BlockingCoroutine{Active}@e2144e4, BlockingEventLoop@6477463f]

Demo: combine context

```
fun main() = runBlocking<Unit> {
    val ctx: CoroutineContext = coroutineContext
    println(ctx)
    val ctx2 = ctx + Dispatchers.Default
    println(ctx2)
}
```

[BlockingCoroutine{Active}@e2144e4, BlockingEventLoop@6477463f]

Demo: combine context

```
fun main() = runBlocking<Unit> {
    val ctx: CoroutineContext = coroutineContext
    println(ctx)
    val ctx2 = ctx + Dispatchers.Default
    println(ctx2)
}
```

```
[BlockingCoroutine{Active}@e2144e4, BlockingEventLoop@6477463f]
[BlockingCoroutine{Active}@e2144e4, DefaultScheduler]
```

Demo: Getting elements

```
fun main() = runBlocking<Unit> {
    println(coroutineContext[Job])
    println(coroutineContext[ContinuationInterceptor])
}
```

Demo

Root CoroutineScope

```
class ContributorsUI : JFrame("GitHub Contributors"), CoroutineScope {  
}
```

```
class ContributorsUI : JFrame("GitHub Contributors"), CoroutineScope {  
  
    private val job = Job()  
  
    override val coroutineContext: CoroutineContext  
        get() = job + Dispatchers.Swing  
  
}
```

Our default UI context

Cancellation

Thread.stop

Deprecated. *This method is inherently unsafe.* Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the **interrupt** method should be used to interrupt the wait.

[https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#stop\(\)](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#stop())

Cancellation has to be
cooperative

Thread.interrupt

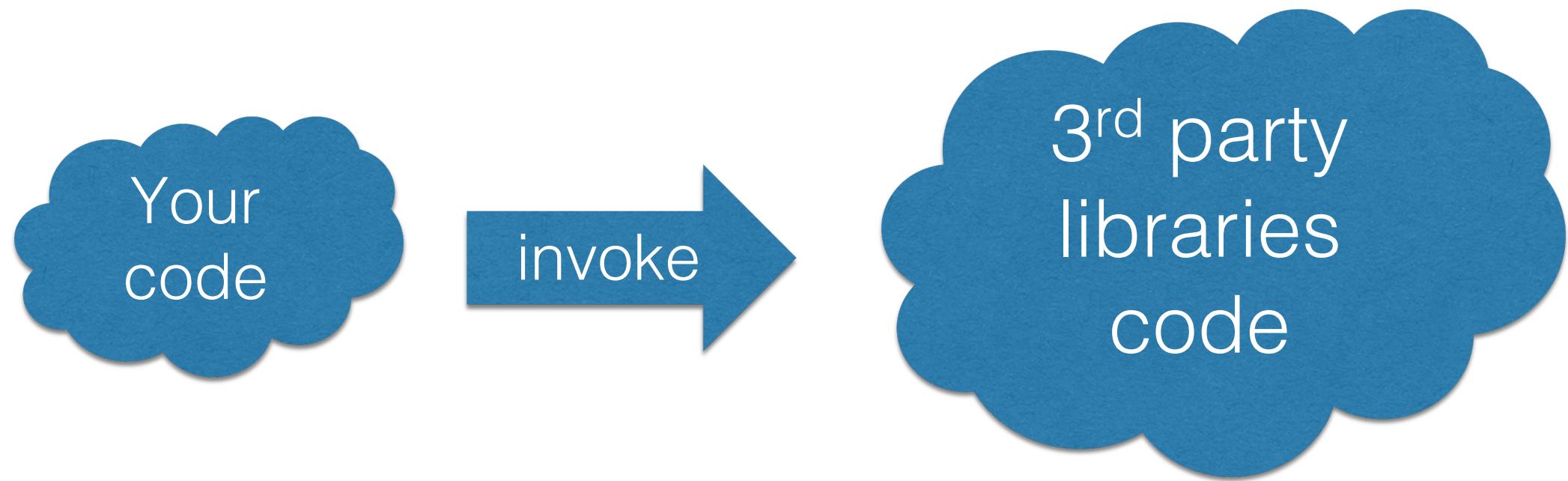
If this thread is **blocked** in an invocation of the [wait\(\)](#), [wait\(long\)](#), or [wait\(long, int\)](#) methods of the [Object](#) class, or of the [join\(\)](#), [join\(long\)](#), [join\(long, int\)](#), [sleep\(long\)](#), or [sleep\(long, int\)](#), methods of this class, then its *interrupt status will be cleared* and it will receive an [InterruptedException](#).

Real Life™



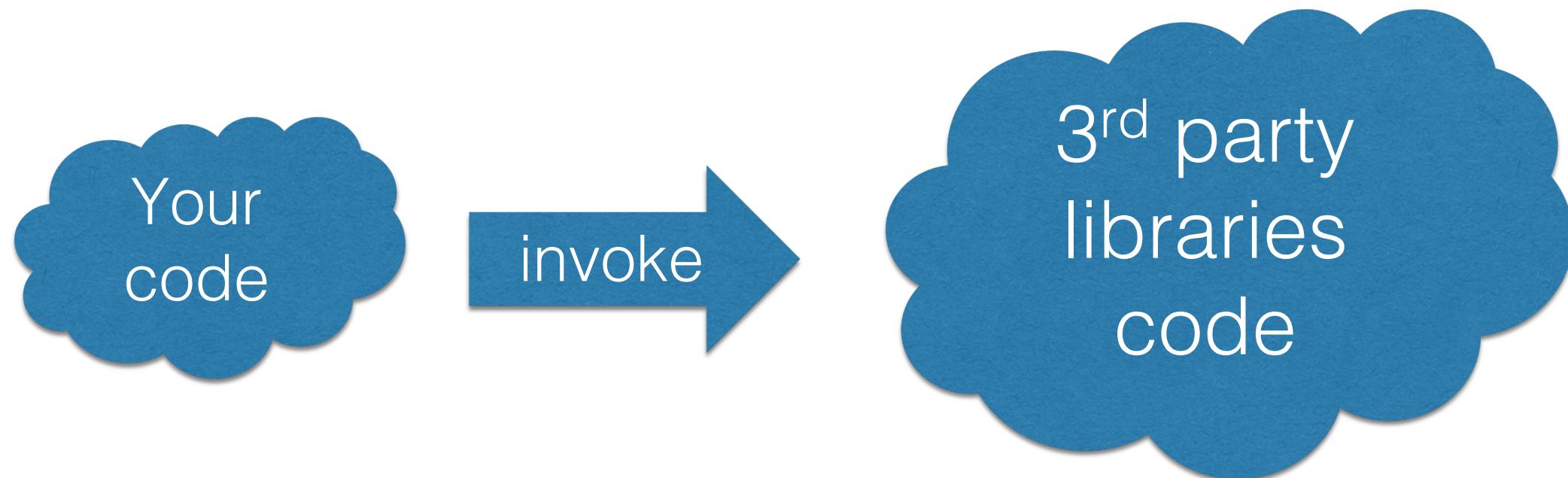
```
// start new thread for a background job
thread {
    while (!Thread.currentThread().isInterrupted) {
        // do something
    }
}
```

Real Life™



```
// start new thread for a background job
thread {
    while (!Thread.currentThread().isInterrupted) {
        // do something
    }
}
```

Real Life™



```
try {
    Object.wait()
} catch (e: InterruptedException) {
    // ignore
}
```

There is no way to reliably
cancel thread-based
background activities

Coroutines to the rescue

- Coroutines are very cheap
- We don't have to reuse them (can always create a new one)
- We can make cancellation flag non-resettable
- 3rd party code cannot break it anymore
- PROFIT!

Demo: Cancellation

```
fun main() = runBlocking<Unit> {
    val job = launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
    delay(1300) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancel() // cancels the job
    job.join() // waits for job's completion
    println("main: Now I can quit.")
}
```

Demo: Cancellation

```
fun main() = runBlocking<Unit> {
    val job = launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
    delay(1300) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancel() // cancels the job
    job.join() // waits for job's completion
    println("main: Now I can quit.")
}
```

Demo

Demo: Cancellation

```
fun main() = runBlocking<Unit> {
    val job = launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
    delay(1300) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
}
```

Demo

Demo: Cooperative

```
fun main() = runBlocking<Unit> {
    val job = launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            Thread.sleep(500)
        }
    }
    delay(1300) // delay a bit
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // cancels the job and waits for its completion
    println("main: Now I can quit.")
}
```

Demo

Only coroutine that is suspended in *cancellable suspending function* can be cancelled

Project: Cancellation

Variant

CANCELLABLE



Does the project work
with cancellation?

Try it

Cancellable suspension

```
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)
        }
    })
}
```

Cancellable suspension

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

Cancellable suspension

```
suspend fun <T> Call<T>.await(): T = suspendCancellableCoroutine { cont ->
    enqueue(object : Callback<T> {
        ...
    })
    cont.invokeOnCancellation {
        cancel()
    }
}
```

Call.cancel()

Exercise: Implement cancellation

CANCELLABLE



```
suspend fun <T> Call<T>.await(): T = suspendCancellableCoroutine { cont ->
    enqueue(object : Callback<T> {
        ...
    })
    cont.invokeOnCancellation {
        cancel()
    }
}
```

- File **src/project/Integration.kt**
- Add cancellation support to await

Hands on!

Timeouts

Demo: Timeout

```
fun main() = runBlocking<Unit> {
    withTimeout(1300) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
}
```

Demo

Demo: TimeoutOrNull

```
fun main() = runBlocking<Unit> {
    withTimeoutOrNull(1300) {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
}
```

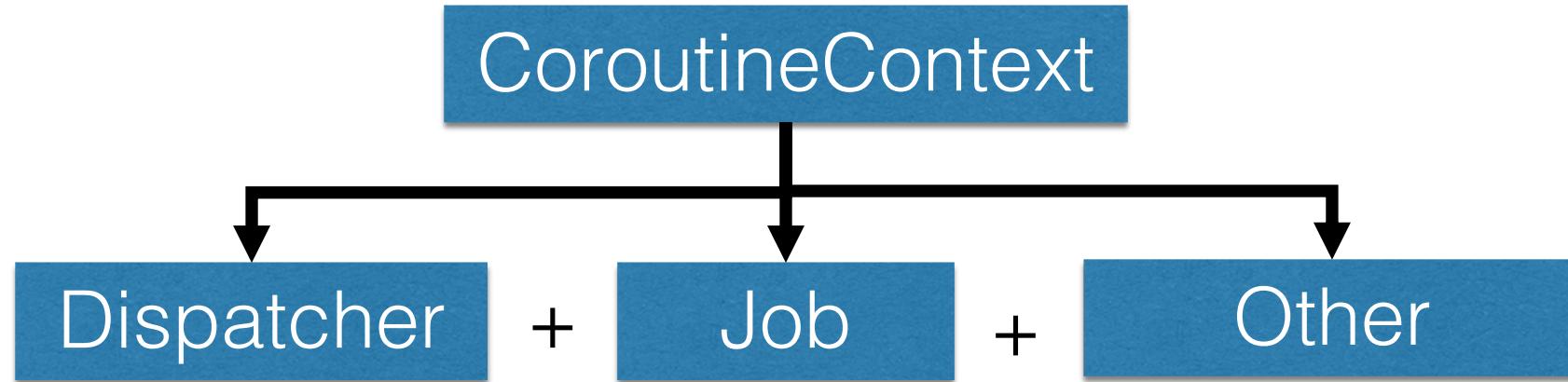
Demo

Demo: Timeout Job

```
fun main() = runBlocking<Unit> {
    println(coroutineContext[Job])
    withTimeoutOrNull(1300) {
        println(coroutineContext[Job])
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500)
        }
    }
}
```

Demo

Recap: Coroutine Context



- **Dispatcher** – determines thread(s) where coroutine executed
- **Job** – represents a *lifetime* of coroutine and keeps its state

**CPU consuming or
blocking code**

Demo: CPU Consuming

```
fun doSomethingSlow() = Thread.sleep(500)

fun main() = runBlocking<Unit> {
    withTimeoutOrNull(1300) {
        repeat(1000) { i ->
            println("I'm working $i ...")
            doSomethingSlow()
        }
    }
}
```

Demo

Demo: Cooperative



```
fun main() = runBlocking<Unit> {
    withTimeoutOrNull(1300) {
        repeat(1000) { i ->
            println("I'm working $i ...")
            doSomethingSlow()
            yield()
        }
    }
}
```

Demo

Demo: Active check

```
fun main() = runBlocking<Unit> {
    withTimeoutOrNull(1300) {
        for (i in 0 until 1000) {
            if (!isActive) break
            println("I'm working $i ...")
            doSomethingSlow()
        }
    }
}
```

Demo

Project: CPU consuming code

Variant

PROGRESS



listOrgRepos

listRepoContributors1

aggregate

SLOW

update UI

listRepoContributors2

aggregate

SLOW

update UI

...

Variant

PROGRESS



```
fun List<User>.aggregateSlow(): List<User> =  
    aggregate()  
    .also {  
        // Imitate CPU consumption / blocking  
        Thread.sleep(500)  
    }
```

- File **src/project/Util.kt**
- Add `aggregateSlow`
- Use `aggregateSlow` in `loadContributorsProgress`

Hands On!

withContext

Demo: withContext

```
suspend fun doSomething() = withContext(Dispatchers.Default) {  
    doSomethingSlow()  
}  
  
fun main() = runBlocking<Unit> {  
    withTimeoutOrNull(1300) {  
        for (i in 0 until 1000) {  
            println("I'm working $i ...")  
            doSomething()  
        }  
    }  
}
```

Demo: withContext

```
suspend fun doSomething() = withContext(Dispatchers.Default) {  
    doSomethingSlow()  
}  
  
fun main() = runBlocking<Unit> {  
    withTimeoutOrNull(1300) {  
        for (i in 0 until 1000) {  
            println("I'm working $i ...")  
            doSomething()  
        }  
    }  
}
```

Demo

Main thread is *suspended*, not blocked



Custom dispatcher

```
val computation =  
  newFixedThreadPoolContext(2, "Computation")
```

Demo: Custom dispatcher

```
val computation =  
    newFixedThreadPoolContext(2, "Computation")  
  
fun main() = runBlocking<Unit> {  
    val job = launch(computation) {  
        display("computation")  
    }  
}
```

Demo

Demo: withContext + custom

```
suspend fun doSomething() = withContext(computation) {  
    doSomethingSlow()  
}  
  
fun main() = runBlocking<Unit> {  
    withTimeoutOrNull(1300) {  
        for (i in 0 until 1000) {  
            println("I'm working $i ...")  
            doSomething()  
        }  
    }  
}
```

Demo

Project: withContext

```
val computation =  
    newFixedThreadPoolContext(2, "Computation")  
  
suspend fun List<User>.aggregateSlow(): List<User> = withContext(computation) {  
    aggregate()  
    .also {  
        // Imitate CPU consumption / blocking  
        Thread.sleep(500)  
    }  
}
```

```
val computation =  
    newFixedThreadPoolContext(2, "Computation")  
  
suspend fun List<User>.aggregateSlow(): List<User> = withContext(computation)  
    aggregate()  
    .also {  
        // Imitate CPU consumption / blocking  
        Thread.sleep(500)  
    }  
}
```

- File **src/project/Util.kt**
- Add computation & withContext

Hands on!

Coroutines and Concurrency

async / await

The classic approach to async

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

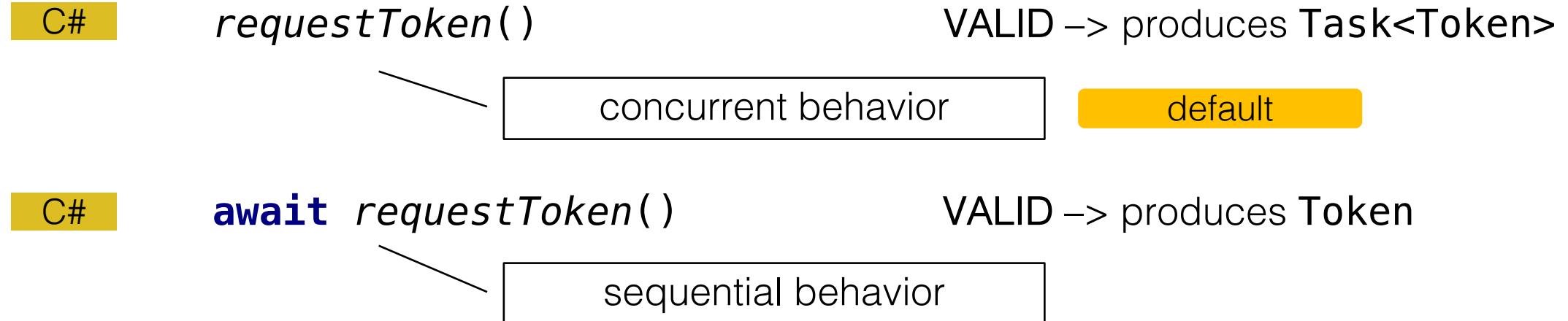
```
async Task<Token> requestToken() { ... }  
async Task<Post> createPost(Token token, Item item) { ... }  
void processPost(Post post) { ... }
```

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



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 { ... }
```

async coroutine builder

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()

await function

and then wait for them

Suspends until deferred is complete

Failure?

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

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

```
val image1 = deferred1.await() —
```

But what if this fails?

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

Failure?

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

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

This one *leaks*

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

But what if this fails?

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

Using async function when needed

Is defined as suspending function, not async

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

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

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())  
    }
```

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())  
    }
```

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

Project: Concurrent

Variant

COROUTINE



listOrgRepos

listRepoContributors1

listRepoContributors2

...

listRepoContributorsN

aggregate

Variant

CONCURRENT

listOrgRepos

listRepoContributors1

listRepoContributors2

...

aggregate

Project with concurrency

```
val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
    async {
        val users = ...
        // ...
        users
    }
}

deferrals.awaitAll() // List<List<User>>
```

Project with concurrency

```
val deferreds: List<Deferred<List<User>>> = repos.map { repo ->
    async {
        val users = ...
        // ...
        users
    }
}

deferreds.awaitAll() // List<List<User>>
```

- File **src/project/Request6Concurrent.kt**
- Use **src/project/Request4Coroutine.kt**

Hands on!

Concurrency and cancellation (in contributors project)

Try it

Children and structured concurrency

Demo: GlobalScope vs Child

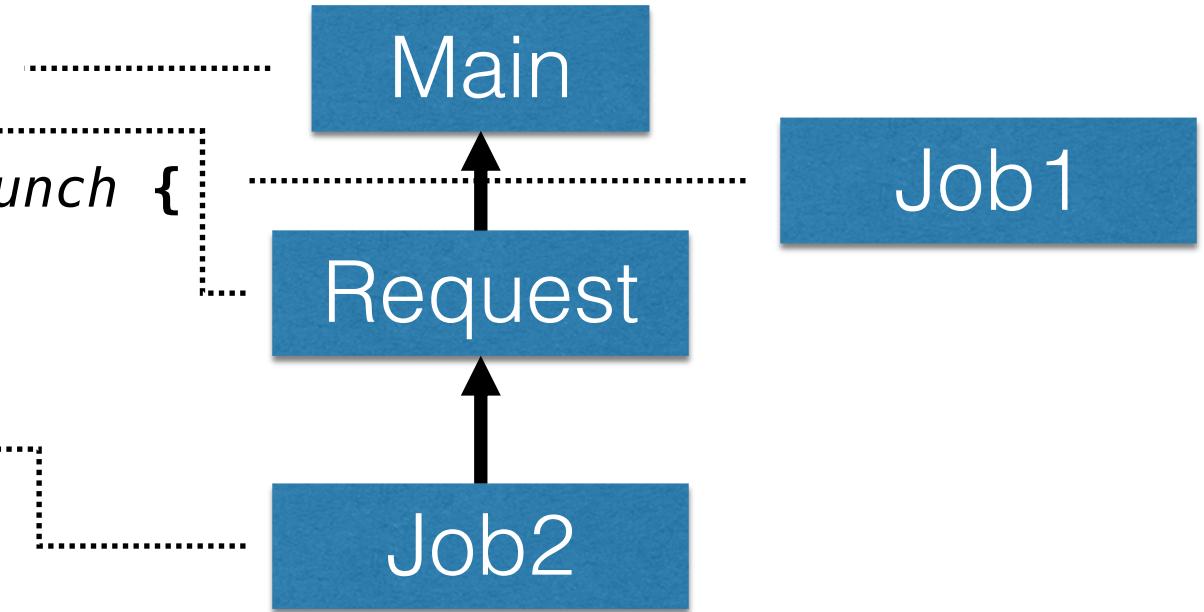
```
fun main() = runBlocking<Unit> {
    val request = launch {
        val job1 = GlobalScope.launch {
            delay(1000)
            println("job1: done")
        }
        val job2 = launch {
            delay(1000)
            println("job2: done")
        }
        job1.join()
        job2.join()
    }
    delay(500)
    request.cancelAndJoin()
    delay(2000)
}
```

Demo

```
fun main() = runBlocking<Unit> {
    val request = launch { this: CoroutineScope
        val job1 = GlobalScope.launch {
            delay(1000)
            println("job1: done")
        }
        val job2 = launch { ——————
            delay(1000)
            println("job2: done")
        }
        job1.join()
        job2.join()
    }
    delay(500)
    request.cancelAndJoin()
    delay(2000)
}
```



```
fun main() = runBlocking<Unit> {  
    val request = launch {  
        val job1 = GlobalScope.launch {  
            delay(1000)  
            println("job1: done")  
        }  
        val job2 = launch {  
            delay(1000)  
            println("job2: done")  
        }  
        job1.join()  
        job2.join()  
    }  
    delay(500)  
    request.cancelAndJoin()  
    delay(2000)  
}
```



Launching coroutine in the
scope of another coroutine
creates *child* coroutine

Async vs Launch

```
val job: Job
```

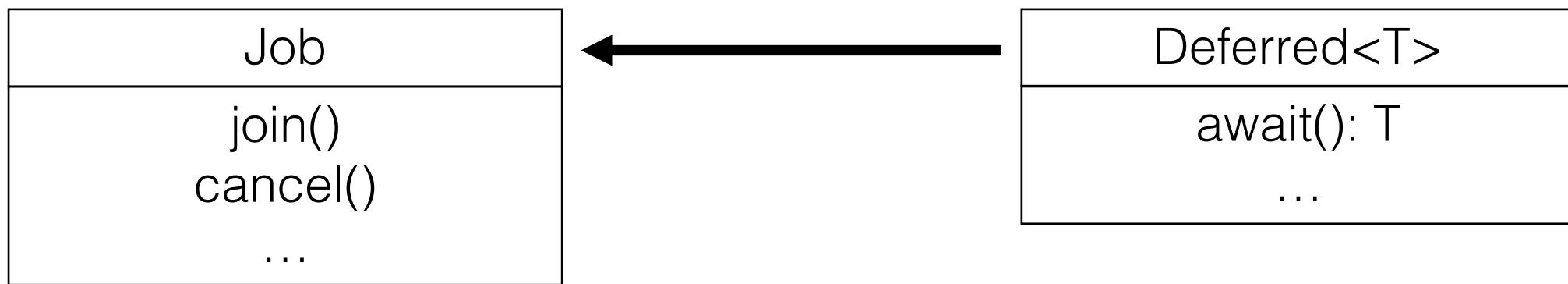
```
= launch { /* ... */ }
```

“Fire and forget”

```
val deferred: Deferred<T> = async { /* ... */ }
```

Compute value, process later

Deferred is a Job



Demo: Async vs Launch

```
fun main() = runBlocking<Unit> {
    val job = GlobalScope.launch {
        delay(1000)
        println("launch: done")
        error("Something went wrong inside launch")
    }
    val deferred = GlobalScope.async {
        delay(1000)
        println("async: done")
        error("Something went wrong inside async")
    }
    job.join()
    deferred.join()
}
```

Demo

async

Deferred (Future)
captures both
successful and failed
result of execution

launch

Delegates exception
handling to parent or
handles (logs) it itself

Demo: Join vs Await

```
fun main() = runBlocking<Unit> {
    val job = GlobalScope.launch {
        delay(1000)
        println("launch: done")
        error("Something went wrong inside launch")
    }
    val deferred = GlobalScope.async {
        delay(1000)
        println("async: done")
        error("Something went wrong inside async")
    }
    job.join()
    deferred.await()
}
```

Demo

Demo: Async vs Launch Children

```
fun main() = runBlocking<Unit> {
    val job = launch {
        delay(1000)
        println("launch: done")
        error("Something went wrong inside launch")
    }
    val deferred = async {
        delay(1000)
        println("async: done")
        error("Something went wrong inside async")
    }
}
```

Demo

Parent-Child recap



- Waits for children completion
- Cancels children on cancel / fail
- Cancels parent on failure
- Launch w/o parent handles exception

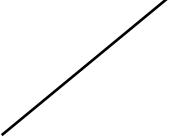
| | Launch | Async |
|---------------------------|------------|------------|
| Has result value/ex. | No | <u>Yes</u> |
| Handles ex. w/o parent | <u>Yes</u> | No |
| Can have parent | Yes | Yes |
| Cancels parent on failure | Yes | Yes |
| Can cancel() it | Yes | Yes |
| Can join() it | Yes | Yes |
| Can await() it | No | <u>Yes</u> |

Java interoperability

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

CompletableFuture<Image> loadAndCombineAsync(String name1,
 String name2)



Imagine implementing it in Java...

Java

```
CompletableFuture<Image> loadImageAsync(String name) { ... }

CompletableFuture<Image> loadAndCombineAsync(String name1,
                                              String name2)
{
    CompletableFuture<Image> future1 = loadImageAsync(name1);
    CompletableFuture<Image> future2 = loadImageAsync(name2);
    return future1.thenCompose(image1 ->
        future2.thenCompose(image2 ->
            CompletableFuture.supplyAsync(() ->
                combineImages(image1, image2))));
```

Java

```
CompletableFuture<Image> loadImageAsync(String name) { ... }

CompletableFuture<Image> loadAndCombineAsync(String name1,
                                              String name2)
{
    CompletableFuture<Image> future1 = loadImageAsync(name1);
    CompletableFuture<Image> future2 = loadImageAsync(name2);
    return future1.thenCombine(future2, this::combineImages);
}
```

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Kotlin **fun** loadAndCombineAsync(
 name1: String,
 name2: String
): CompletableFuture<Image> =
 GlobalScope.future {
 val future1 = loadImageAsync(name1)
 val future2 = loadImageAsync(name2)
 combineImages(future1.await(), future2.await())
 }

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Kotlin **fun** loadAndCombineAsync(
 name1: String,
 name2: String
): CompletableFuture<Image> =
 GlobalScope.future {
 val future1 = loadImageAsync(name1)
 val future2 = loadImageAsync(name2)
 combineImages(future1.await(), future2.await())
 }

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Kotlin

```
fun loadAndCombineAsync(
    name1: String,
    name2: String
): CompletableFuture<Image> =
    GlobalScope.future {
        val future1 = loadImageAsync(name1)
        val future2 = loadImageAsync(name2)
        combineImages(future1.await(), future2.await())
    }
```

future coroutine builder

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Kotlin **fun** loadAndCombineAsync(
 name1: String,
 name2: String
): CompletableFuture<Image> =
 GlobalScope.future {
 val future1 = loadImageAsync(name1)
 val future2 = loadImageAsync(name2)
 combineImages(future1.await(), future2.await())
 }

Java CompletableFuture<Image> loadImageAsync(String name) { ... }

Kotlin **fun** loadAndCombineAsync(
 name1: String,
 name2: String
): CompletableFuture<Image> =
 GlobalScope.future {
 val future1 = loadImageAsync(name1)
 val future2 = loadImageAsync(name2)
 combineImages(future1.**await()**, future2.**await()**)
 }

Extension for Java's CompletableFuture

Project: Java futures

Variant

FUTURE

```
loadContributorsConcurrentAsync(req).thenAccept { users ->  
    updateResults(users)  
}
```

What thread did it run in?

Try it

Variant

FUTURE

```
loadContributorsConcurrentAsync(req).thenAccept { users ->
    SwingUtilities.invokeLater {
        updateResults(users)
    }
}
```

Try it

Thread locals

Demo: Thread local

```
val userId = ThreadLocal<String>()
```

Demo: Thread local

```
val userId = ThreadLocal<String>()

fun main() = runBlocking<Unit> {
    userId.set("foo")
}
```

Demo: Thread local

```
val userId = ThreadLocal<String>()

fun main() = runBlocking<Unit> {
    userId.set("foo")
    launch(Dispatchers.Default) {
        println("userId = ${userId.get()}")
    }
}
```

Demo

Demo: Thread local

```
val userId = ThreadLocal<String>()

fun displayUserId() =
    println("[${Thread.currentThread().name}] userId = ${userId.get()}")

fun main() = runBlocking<Unit> {
    userId.set("foo")
    displayUserId()
    launch(Dispatchers.Default) {
        displayUserId()
    }
}
```

Demo: Thread local

```
val userId = ThreadLocal<String>()

fun displayUserId() =
    println("[${Thread.currentThread().name}] userId = ${userId.get()}")

fun main() = runBlocking<Unit> {
    userId.set("foo")
    displayUserId()
    launch(Dispatchers.Default) {
        displayUserId()
    }
}
```

```
[main] userId = foo
[DefaultDispatcher-worker-1] userId = null
```

Demo: Context Element

```
val userId = ThreadLocal<String>()

fun main() = runBlocking<Unit> {
    userId.set("foo")
    withContext(userId.asContextElement()) {
        launch(Dispatchers.Default) {
            println("userId = ${userId.get()}")
        }
    }
}
```

Demo

Demo: Calls

```
val userId = ThreadLocal<String>()

fun main() = runBlocking<Unit> {
    userId.set("foo")
    withContext(userId.asContextElement()) {
        doSomething()
    }
}

suspend fun doSomething() = withContext(Dispatchers.Default) {
    println("userId = ${userId.get()}")
}
```

Demo: Calls

```
val userId = ThreadLocal<String>()

fun main() = runBlocking<Unit> {
    userId.set("foo")
    withContext(userId.asContextElement()) {
        doSomething()
    }
}

suspend fun doSomething() = withContext(Dispatchers.Default) {
    println("userId = ${userId.get()}")
}
```

Demo

Beyond asynchronous code

Fibonacci sequence

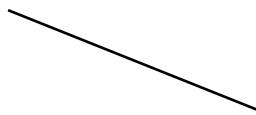
```
val fibonacci = sequence {
    var cur = 1
    var next = 1
    while (true) {
        yield(cur)
        val tmp = cur + next
        cur = next
        next = tmp
    }
}
```

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

fun main() {
    println(fibonacci.take(10).toList())
}
```

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

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



A coroutine builder with restricted suspension

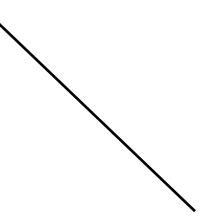
```
val fibonacci = sequence {  
    var cur = 1  
    var next = 1  
    while (true) {  
        yield(cur)  
        val tmp = cur + next  
        cur = next  
        next = tmp  
    }  
}
```

A suspending function

The same building blocks

```
fun <T> sequence(  
    block: suspend SequenceScope<T>.() -> Unit  
): Sequence<T> { ... }
```

```
fun <T> sequence(  
    block: suspend SequenceScope<T>.() -> Unit  
) : Sequence<T> { ... }
```



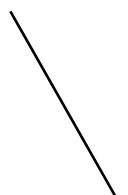
Result is a *synchronous* sequence

```
fun <T> sequence(  
    block: suspend SequenceScope<T>.() -> Unit  
): Sequence<T> { ... }
```

Suspending lambda with receiver

```
fun <T> sequence(  
    block: suspend SequenceScope<T>.() -> Unit  
): Sequence<T> { ... }
```

```
@RestrictsSuspension  
abstract class SequenceScope<in T> {  
    abstract suspend fun yield(value: T)  
}
```



Coroutine is restricted only to suspending functions defined here

Synchronous

```
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()
```

```
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())
```

```
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())
```

```
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
```

```
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())
```

```
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())
```

```
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
```

```
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
println(iter.next()) // 2
```

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



Synchronous with invoker

```
val iter = fibonacci.iterator()
println(iter.next()) // 1
println(iter.next()) // 1
println(iter.next()) // 2
```

Library vs language

Classic async

async/await
generate/yield } Keywords

Kotlin coroutines

suspend

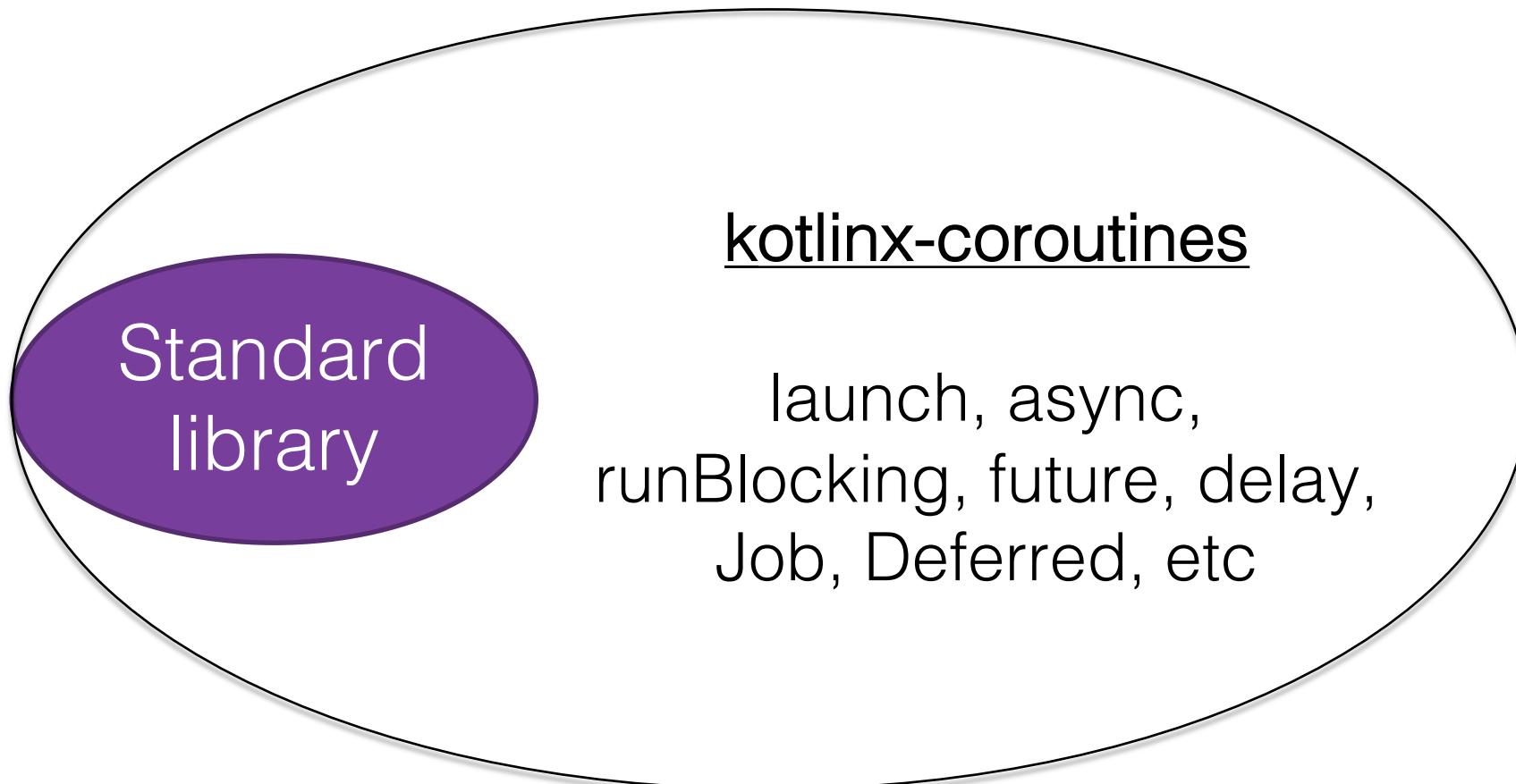
} Modifier

Kotlin coroutines



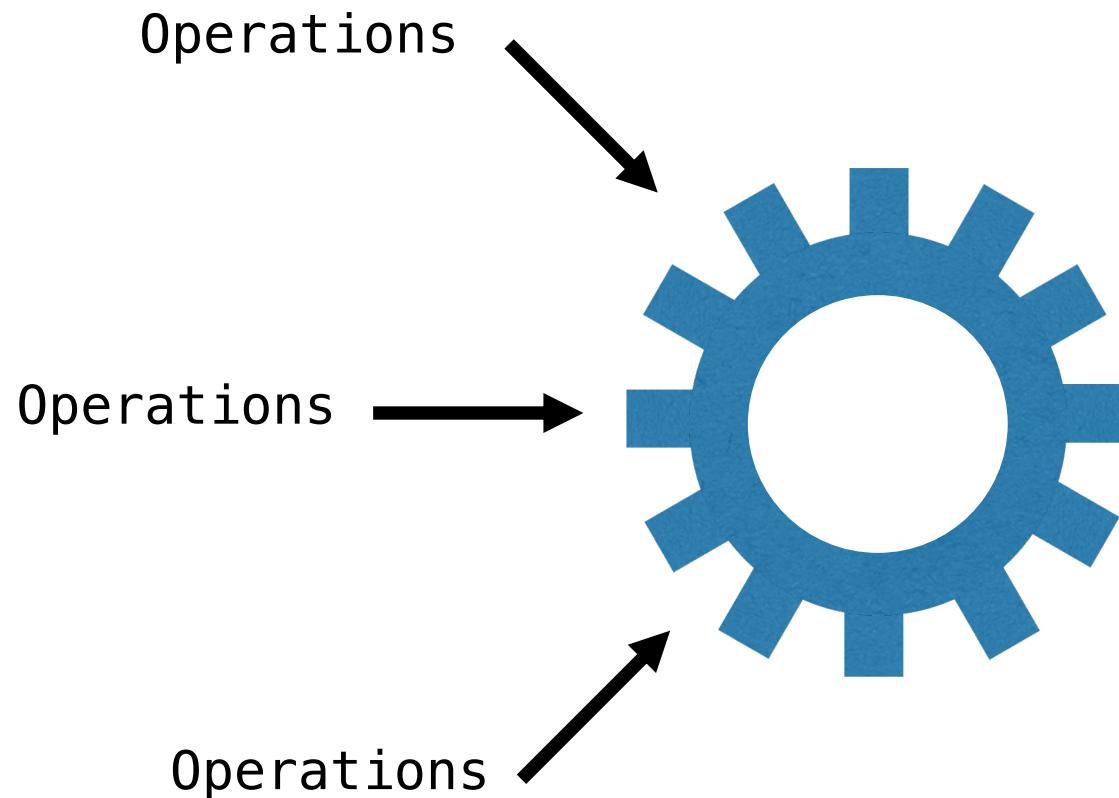
Standard
library

Kotlin coroutines

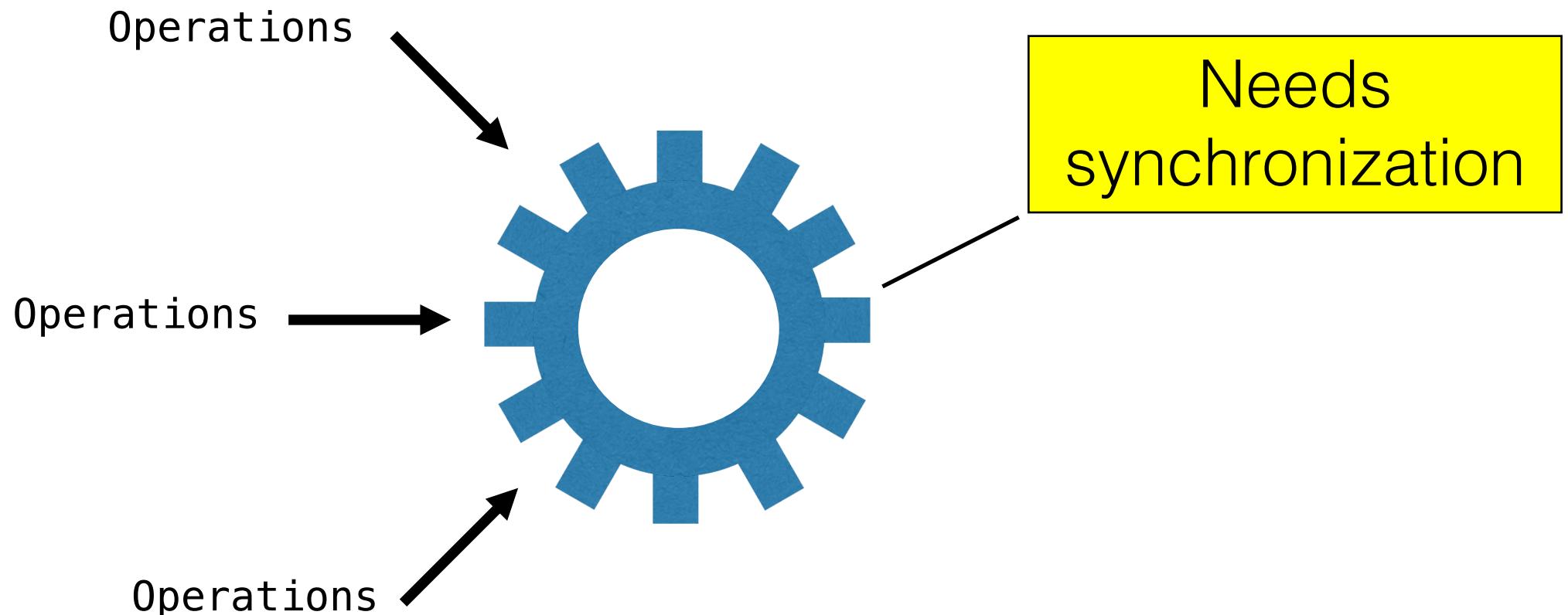


Part 4: CSP and Channels

Shared Mutable State



Shared Mutable State

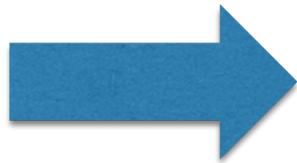


Shared Mutable State



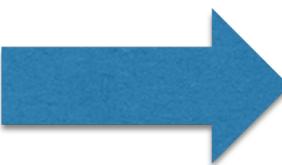
Communicating
Sequential
Processes

Shared
Mutable State



Share by
Communicating

Synchronization
Primitives



Communication
Primitives

Channel

Channel



Demo: Channel

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) {
            channel.send(x * x)
        }
    }
    repeat(5) {
        println(channel.receive())
    }
    println("Done!")
}
```

Demo

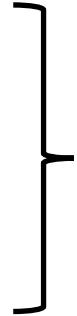
Demo: Channel with Delay

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) {
            delay(500)
            channel.send(x * x)
        }
    }
    repeat(5) {
        println(channel.receive())
    }
    println("Done!")
}
```

Demo

CSP

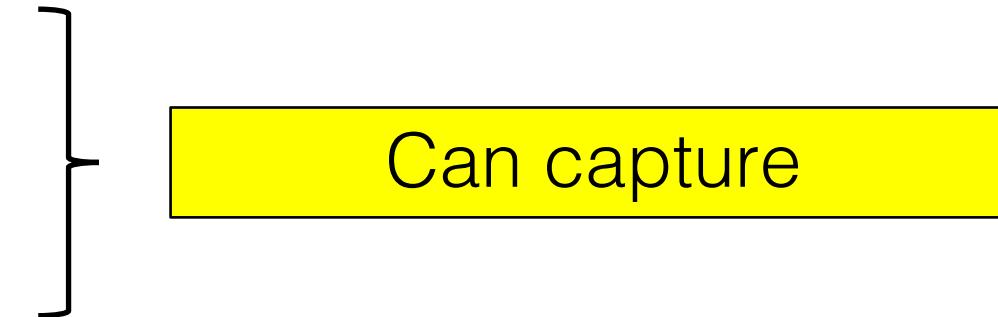
```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    launch {
        for (x in 1..5) {
            delay(500)
            channel.send(x * x)
        }
    }
    repeat(5) {
        println(channel.receive())
    }
    println("Done!")
}
```



Sequential code

Accidentally share?

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    // !!! outer scope is accessible !!!
    launch {
        for (x in 1..5) {
            delay(500)
            channel.send(x * x)
        }
    }
    repeat(5) {
        println(channel.receive())
    }
    println("Done!")
}
```



The code demonstrates a common mistake in Kotlin coroutines where a variable from the outer scope is captured by a coroutine. In this example, the variable `channel` is captured by both the `launch` block and the `repeat` loop. A brace groups the `launch` block and the `repeat` loop, and a yellow box to its right contains the text "Can capture".

Encapsulate

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            delay(500)  
            channel.send(x * x)  
        }  
    }  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Encapsulate

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            delay(500)  
            channel.send(x * x)  
        }  
    }  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Encapsulate

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            delay(500)  
            channel.send(x * x)  
        }  
    }
```



Sequential code!

```
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Encapsulate

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            delay(500)  
            channel.send(x * x)  
        }  
    }  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Demo

Demo: Thread?

```
val curThread: String get() = Thread.currentThread().name

fun CoroutineScope.producer(channel: Channel<Int>) =
    launch {
        for (x in 1..5) {
            println("Sending $x from $curThread")
            channel.send(x)
        }
    }
```

Demo: Thread?

```
val curThread: String get() = Thread.currentThread().name

fun CoroutineScope.producer(channel: Channel<Int>) = ...

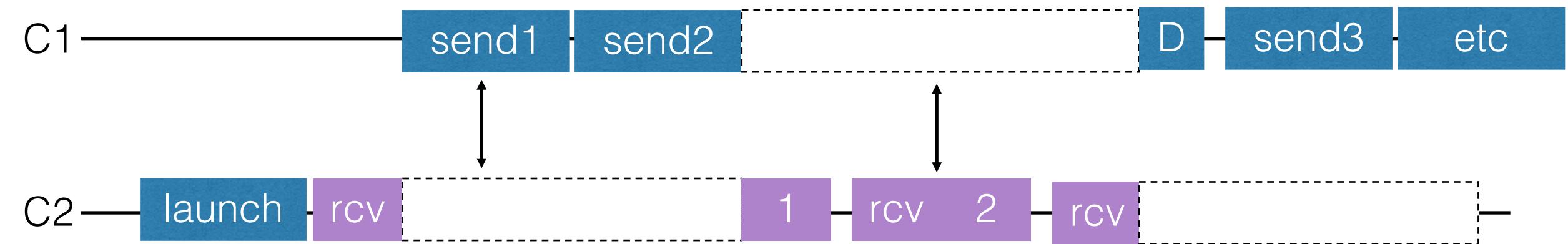
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>()
    producer(channel)
    repeat(5) {
        println("${channel.receive()} receive from $curThread")
    }
    println("Done!")
}
```

Demo

Peculiar pattern

```
Sending 1 from main
Sending 2 from main
1 receive from main
2 receive from main
Sending 3 from main
Sending 4 from main
3 receive from main
4 receive from main
Sending 5 from main
5 receive from main
Done!
```

Rendezvous channel



Demo: Background thread

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch(Dispatchers.Default) {  
        for (x in 1..5) {  
            println("Sending $x from $curThread")  
            channel.send(x)  
        }  
    }  
  
fun main() = ...
```

Demo

Channels are concurrent
communication primitives

Project: Better Concurrency

Variant

CONCURRENT

listOrgRepos

listRepoContributors1

listRepoContributors2

...

aggregate

Variant

CONCURRENT

listOrgRepos

listRepoContributors1

listRepoContributors2

...

aggregate

SLOW

Variant

GATHER

listOrgRepos

listRepoContributors1

listRepoContributors2

...

channel

aggregate

SLOW

update UI

aggregate

SLOW

update UI

...

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = ...
        // ...
        channel.send(users)
    }
}
```

Request8Gather.kt

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = ...
        // ...
        channel.send(users)
    }
}

var contribs = emptyList<User>()
repeat(repos.size) {
    val users = channel.receive()
    contribs = (contribs + users).aggregateSlow()
    callback(contribs)
}
```

State

Request8Gather.kt

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = ...
        // ...
        channel.send(users)
    }
}
```

State

```
var contribs = emptyList<User>()
repeat(repos.size) {
    val users = channel.receive()
    contribs = (contribs + users).aggregateSlow()
    callback(contribs)
}
```

Sequential

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = ...
        // ...
        channel.send(users)
    }
}

var contribs = emptyList<User>()
repeat(repos.size) {
    val users = channel.receive()
    contribs = (contribs + users).aggregateSlow()
    callback(contribs)
}
```

- File **src/project/Request8Gather.kt**
- Use **src/project/Request6Concurrent.kt**

Hands on!

Closing & iterating channels

How to determine the end?

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            channel.send(x * x)  
        }  
    }  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println(channel.receive())  
    }  
    println("Done!")  
}
```

Close channel

```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            channel.send(x * x)  
        }  
        channel.close()  
    }  
  
fun main() = ...
```

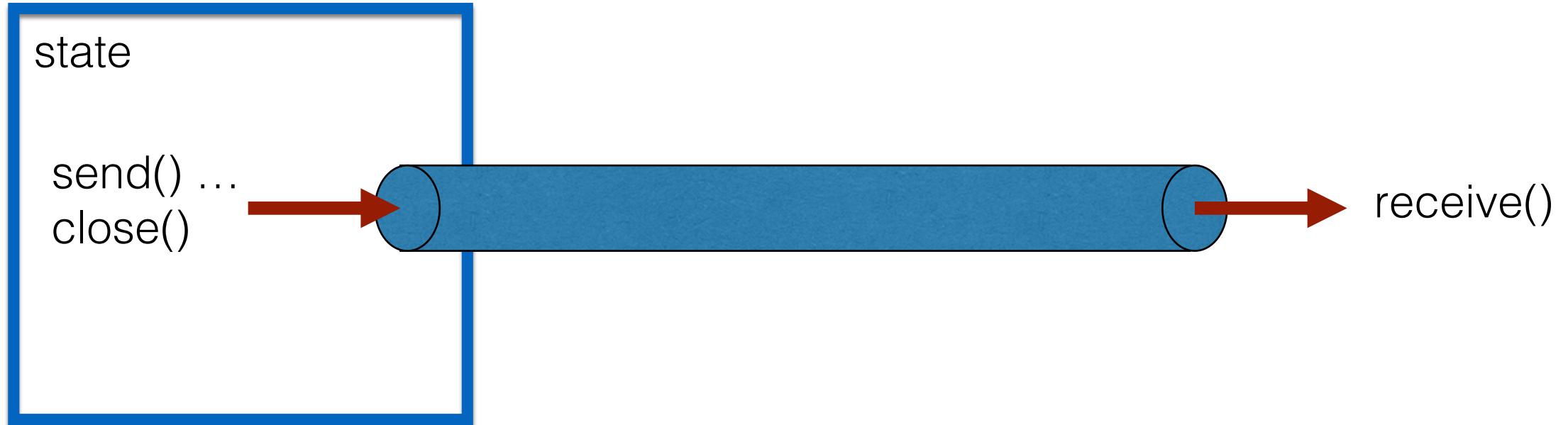
Iterate over channel

```
fun CoroutineScope.producer(channel: Channel<Int>) = ...  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    for (element in channel) {  
        println(element)  
    }  
    println("Done!")  
}
```

Demo

Pattern: Producer

Producer



```
fun CoroutineScope.producer(channel: Channel<Int>) =  
    launch {  
        for (x in 1..5) {  
            channel.send(x * x)  
        }  
        channel.close()  
    }  
  
fun main() = ...
```

```
fun CoroutineScope.producer(): ReceiveChannel<Int> =  
    produce {  
        for (x in 1..5) {  
            send(x * x)  
        }  
    }  
  
fun main() = ...
```

Implicit close()

```
fun CoroutineScope.producer(): ReceiveChannel<Int> =  
    produce {  
        for (x in 1..5) {  
            send(x * x)  
        }  
    }
```

```
fun main() = ...
```

Returns a channel

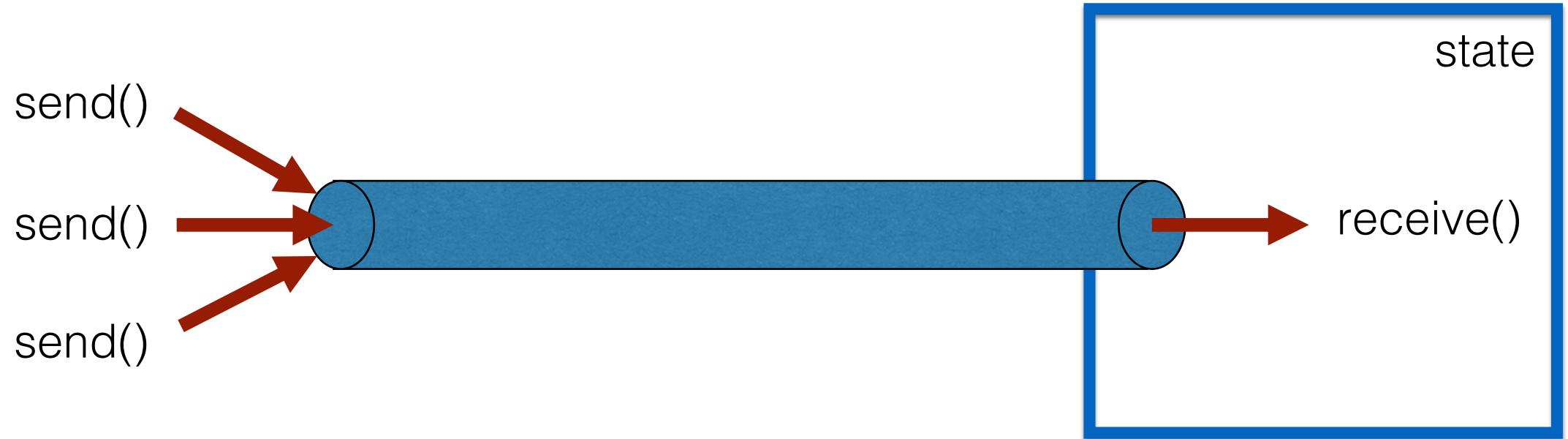
Implicit close()

```
fun CoroutineScope.producer(): ReceiveChannel<Int> = ...  
  
fun main() = runBlocking<Unit> {  
    val channel = producer()  
    for (element in channel) {  
        println(element)  
    }  
    println("Done!")  
}
```

Demo

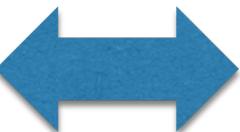
Pattern: Actor

Actor



CSP

Named
channels



Actor Model

Named
coroutines

Actor == named coroutine & inbox channel

Demo: Actor

```
fun CoroutineScope.summer(): SendChannel<Int> = actor {
    var sum = 0 // State!
    for (i in channel) {
        sum += i
        println("Sum = $sum")
    }
}
```

Demo: Actor

```
fun CoroutineScope.summer(): SendChannel<Int> = actor {
    var sum = 0 // State!
    for (i in channel) {
        sum += i
        println("Sum = $sum")
    }
}
```

}

Sequential!

Demo: Actor

Encapsulates state

```
fun CoroutineScope.summer(): SendChannel<Int> = actor {  
    var sum = 0 // State!  
    for (i in channel) {  
        sum += i  
        println("Sum = $sum")  
    }  
}
```



Sequential!

Demo: Actor

```
fun CoroutineScope.summer(): SendChannel<Int> = ...  
  
fun main() = runBlocking<Unit> {  
    val summer = summer()  
    for (x in 1..5) {  
        summer.send(x)  
    }  
    summer.close()  
}
```

Demo

Project: Actors

UI Update Actor

```
private val uiUpdateActor =  
    actor<List<User>> {  
        for (users in channel) {  
            updateResults(users)  
        }  
    }
```

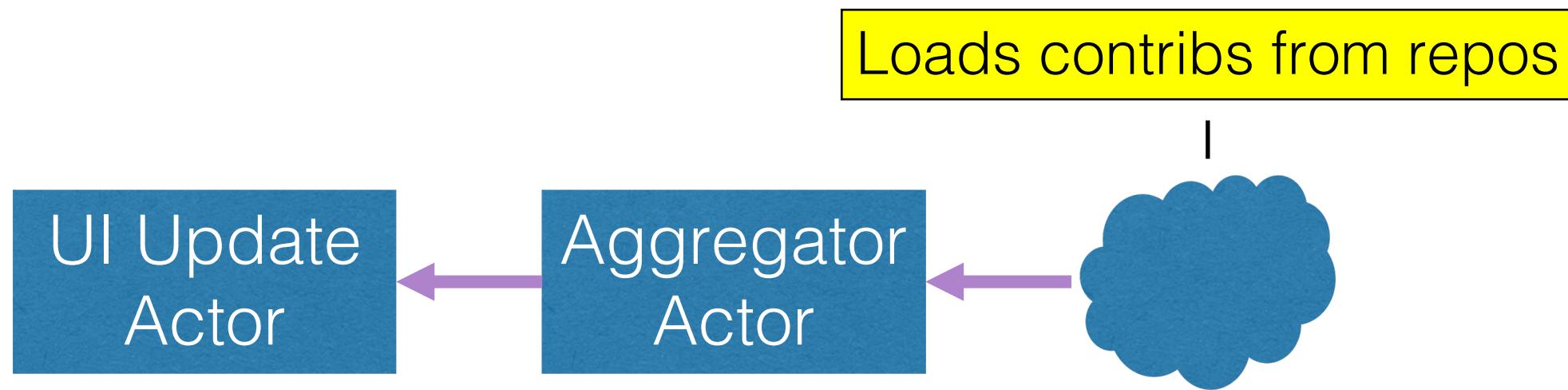
UI Update Actor

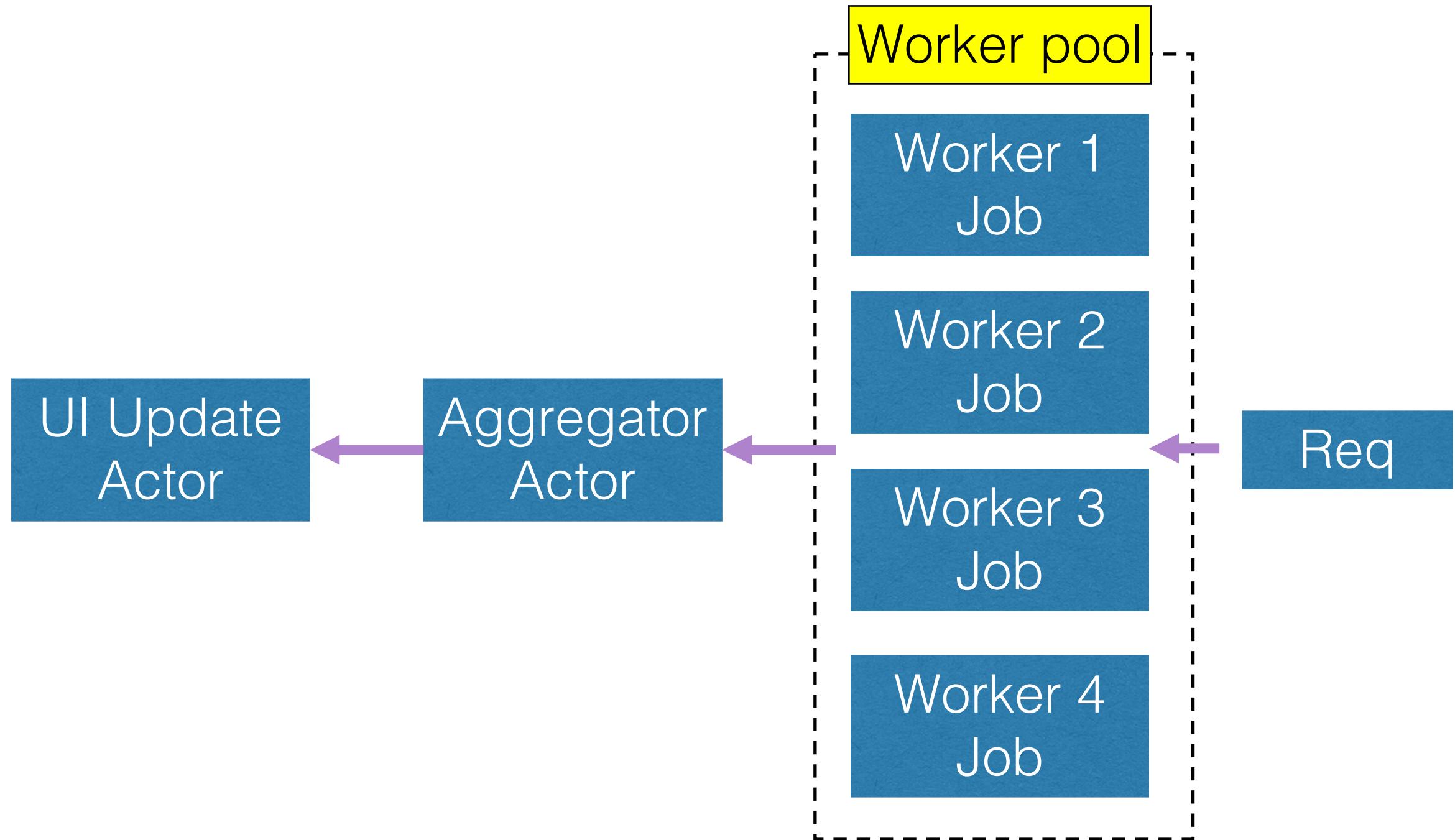
```
private val uiUpdateActor =  
    actor<List<User>> {  
        for (users in channel) {  
            updateResults(users)  
        }  
    }  
  
launch {  
    loadContributorsActor(req, uiUpdateActor)  
}
```



Aggregator Actor

```
fun CoroutineScope.aggregatorActor(  
    uiUpdateActor: SendChannel<List<User>>  
) =  
    actor<List<User>> {  
        var contribs: List<User> = emptyList() // STATE  
        for (users in channel) {  
            // TODO: :UPDATE STATE:  
            uiUpdateActor.send(contribs)  
        }  
    }
```





Worker request

```
class WorkerRequest(  
    val service: GitHubService,  
    val org: String,  
    val repo: String  
)
```

Worker Job

```
fun CoroutineScope.workerJob(  
    requests: ReceiveChannel<WorkerRequest>,  
    aggregator: SendChannel<List<User>>  
) =  
    launch {  
        for (req in requests) {  
            val users = TODO()  
            aggregator.send(users)  
        }  
    }
```

Launch workers

```
val aggregator = aggregatorActor(uiUpdateActor)
val requests = Channel<WorkerRequest>()
val workers = List(4) {
    workerJob(requests, aggregator)
}
```

Send requests to workers

```
for (repo in repos) {  
    requests.send(WorkerRequest(service, req.org, repo.name))  
}  
requests.close()
```

Join workers & complete

```
workers.joinAll()  
aggregator.close()
```

- File **src/project/Request9Actor.kt**
- Fill in the blanks (TODOs)
- Use **src/project/Request8Gather.kt**

Hands on!

Mutex vs Actor

Demo: Counter actor

```
sealed class CounterMsg
object IncCounter : CounterMsg()
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg()
```

Demo: Counter actor

```
sealed class CounterMsg
object IncCounter : CounterMsg()
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg()

fun counterActor() = actor<CounterMsg> {
    var counter = 0
    for (msg in channel) {
        when (msg) {
            is IncCounter -> counter++
            is GetCounter -> msg.response.complete(counter)
        }
    }
}
```

Demo: Counter actor

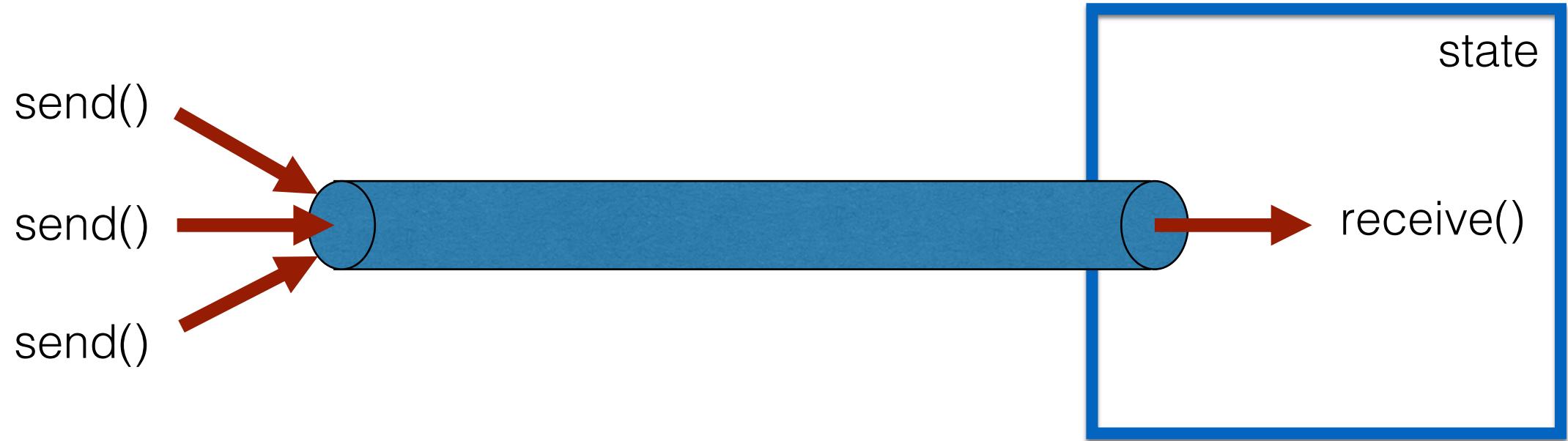
```
sealed class CounterMsg
object IncCounter : CounterMsg()
class GetCounter(val response: CompletableDeferred<Int>) : CounterMsg()

fun counterActor() = ...

fun main() = runBlocking<Unit> {
    val counter = counterActor()
    repeat(100) {
        counter.send(IncCounter)
    }
    val response = CompletableDeferred<Int>()
    counter.send(GetCounter(response))
    println("Counter = ${response.await()}")
    counter.close() // shutdown the actor
}
```

Demo

Actor



Demo: Counter with Mutex

```
class Counter {  
    private var counter = 0  
    private val mutex = Mutex()  
  
    suspend fun inc() = mutex.withLock {  
        counter++  
    }  
  
    suspend fun get() = mutex.withLock {  
        counter  
    }  
}
```

Demo: Counter with Mutex

```
class Counter { ... }

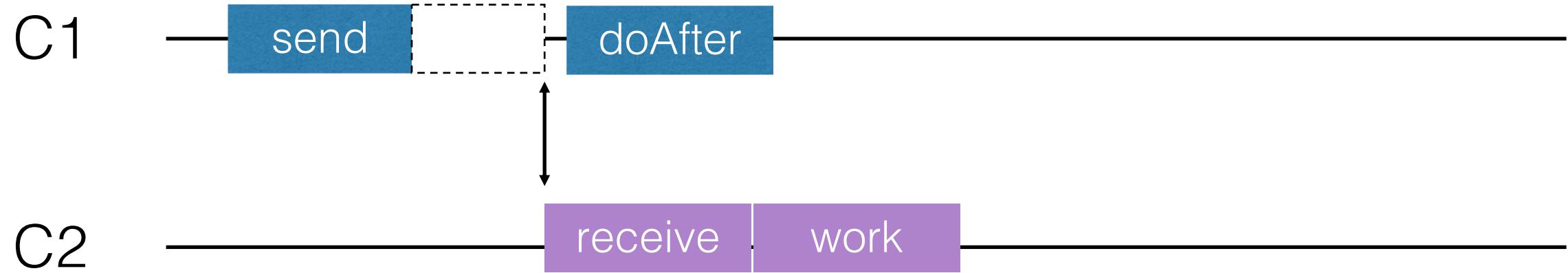
fun main() = runBlocking<Unit> {
    val counter = Counter()
    repeat(100) {
        counter.inc()
    }
    println("Counter = ${counter.get()}")
}
```

Demo

Mutex



Actor



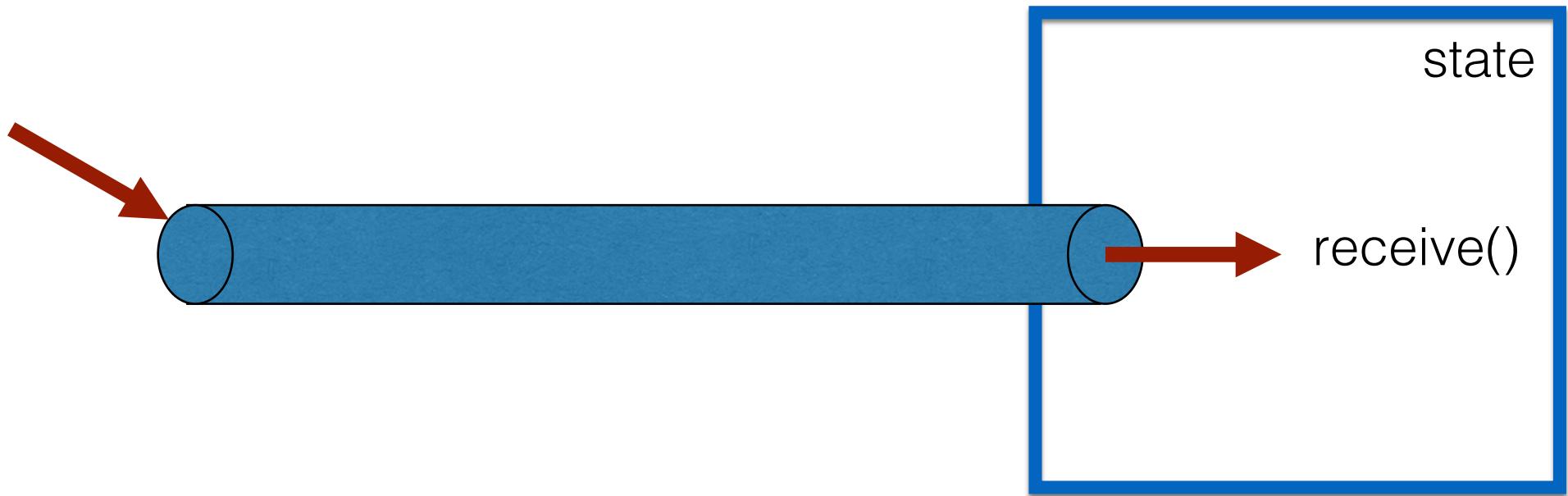
Mutex



Buffered channels

Motivation: Actors

doBefore()
send()
doAfter



Recap: Redezvous channels

```
fun CoroutineScope.producer(channel: Channel<Int>) = ...  
  
fun main() = runBlocking<Unit> {  
    val channel = Channel<Int>()  
    producer(channel)  
    repeat(5) {  
        println("${channel.receive()} receive")  
    }  
    println("Done!")  
}
```

Demo

Demo: Buffered channels

```
fun CoroutineScope.producer(channel: Channel<Int>) = ...
```

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>(5)
    producer(channel)
    repeat(5) {
        println("${channel.receive()} receive")
    }
    println("Done!")
}
```

Capacity

Demo

References and further reading

Guide to kotlinx.coroutines by example

- Basics
- Cancellation and Timeouts
- Composition
- Coroutine contexts
- Channels
- Shared Mutable State and Concurrency
- Select expressions

<https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md>

Thank you

Any questions?



Roman Elizarov

 elizarov @
relizarov

