

A Curiosity rover is shown on the surface of Mars, its robotic arm extended towards a large rock. The background is a hazy, reddish-brown landscape. Overlaid on the image is large, bold, white text.

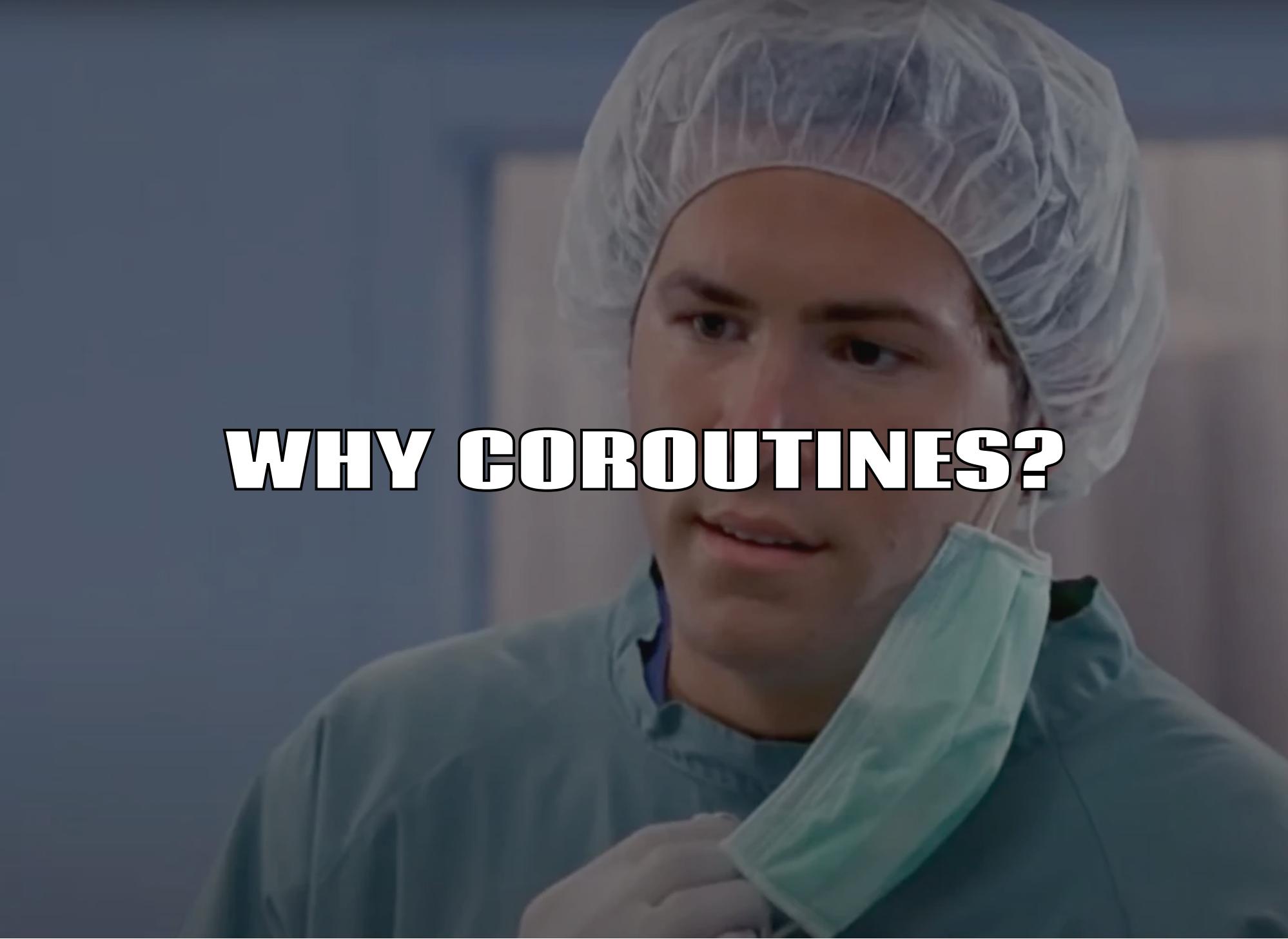
# **EXPLORING KOTLIN COROUTINES**

**Stefan Bissell**

# Examples and slides:

<https://github.com/weaselflink/exploring-kotlin-coroutines>



A close-up photograph of a man wearing a white surgical cap and a light blue surgical mask. He is looking slightly to his right with a neutral expression. The background is a plain, light-colored wall.

# WHY COROUTINES?

# WHY DO WE USE THREADS?

Threads are used to ...

# WHY DO WE USE THREADS?

Threads are used to ...

- utilize multiple processors

# WHY DO WE USE THREADS?

Threads are used to ...

- utilize multiple processors
- WAIT

# WHAT DOES A THREAD COST?

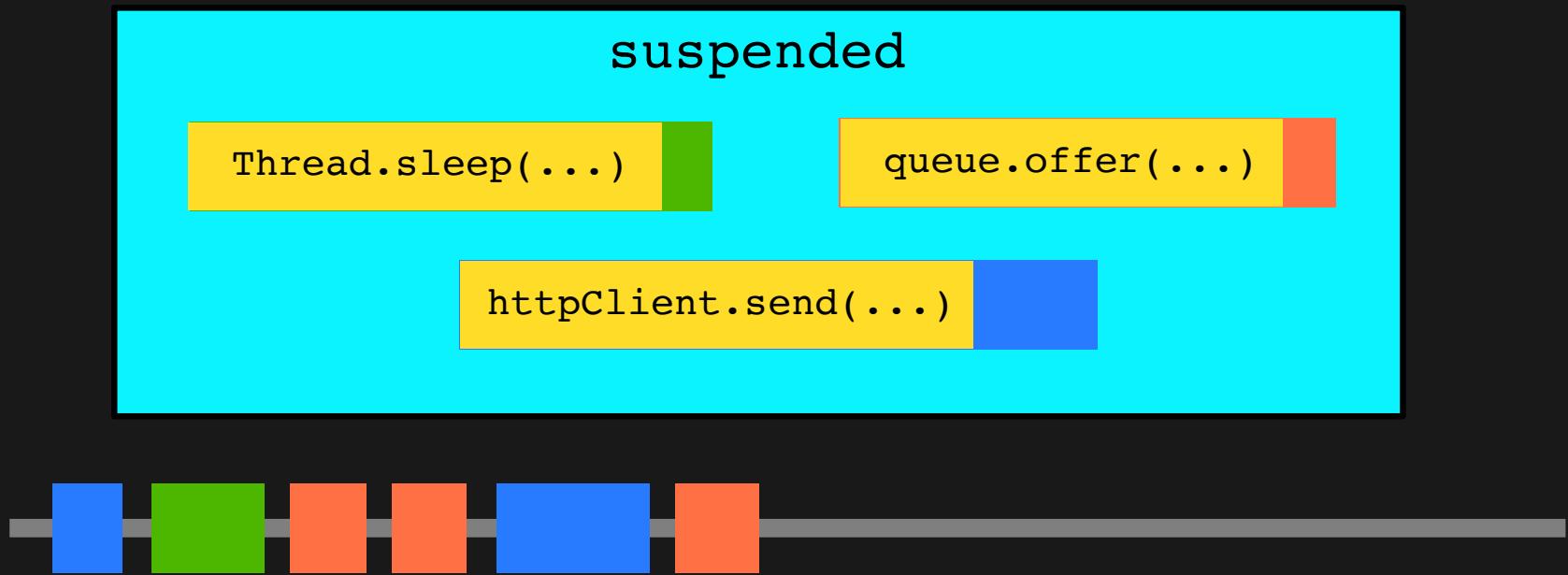
```
1 while (true) {  
2     thread {  
3         TimeUnit.MINUTES.sleep(1)  
4     }  
5 }
```

You will get a OutOfMemoryError relatively soon  
(a few thousand threads on my machine).

# THREADS MOSTLY WAIT



# ONE THREAD SHOULD BE ENOUGH



# COROUTINES ARE CHEAP

Coroutines love waiting for something.

```
1 runBlocking {  
2     repeat(1_000_000) {  
3         launch {  
4             delay(TimeUnit.MINUTES.toMillis(1))  
5         }  
6     }  
7 }
```

# NO MORE THREADS?

Coroutines still need threads to execute them.

```
1 val context = newSingleThreadContext("single thread")
2 runBlocking {
3     launch(context) {
4         println(Thread.currentThread().name)
5     }
6 }
```

# BREAKING COROUTINES

Blocking the thread inside a coroutine is a bad idea!

```
1 val context = newSingleThreadContext("single thread")
2 runBlocking {
3     launch(context) {
4         TimeUnit.MINUTES.sleep(1)
5     }
6     launch(context) {
7         println("I have to wait a minute")
8     }
9 }
```

```
READY
3 FOR X=1 TO 10
3 PRINT "HOLA WIKIPEDIA"
3 NEXT X
```

UN

```
HOLA WIKIPEDIA
```

# BASICS

READY

A photograph of a space shuttle launching from a launch pad. The shuttle is positioned vertically in the center of the frame, with its orange external fuel tank and white solid rocket boosters. A large plume of white smoke and fire erupts from the base of the shuttle, obscuring the lower portion of the image. In the background, the clear blue sky is filled with scattered white clouds. To the left of the shuttle, the metal lattice structure of the launch tower is visible. On the right, the dark silhouette of a building or hangar can be seen. The overall scene conveys a sense of powerful motion and ascent.

# LAUNCHING COROUTINES

# THE LAUNCH() METHOD

```
public fun CoroutineScope.launch(...): Job
```

# THE LAUNCH() METHOD

```
public fun CoroutineScope.launch(...): Job
```

- has no suspend keyword

# THE LAUNCH() METHOD

```
public fun CoroutineScope.launch(...): Job
```

- has no `suspend` keyword
- needs a `CoroutineScope`

# THE LAUNCH() METHOD

```
public fun CoroutineScope.launch(...): Job
```

- has no `suspend` keyword
- needs a `CoroutineScope`
- returns a `Job`

# NOT SUSPENDING

```
1 launch {  
2     println("I am second")  
3 }  
4 println("I am first")
```

# NOT SUSPENDING

```
1 launch {
2     println("I am first")
3 }
4 yield() // allows launch to run
5 println("I am second")
```

# NEEDS A SCOPE

```
1 CoroutineScope(someContext).launch {  
2     // ...  
3 }
```

# NEEDS A SCOPE

```
1 runBlocking {  
2     launch {  
3         // ...  
4     }  
5 }
```

# NEEDS A SCOPE

```
1 suspend fun bla() {
2     coroutineScope {
3         launch {
4             // ...
5         }
6     }
7 }
```

# NEEDS A SCOPE

```
1 suspend fun CoroutineScope.bla() {
2     launch {
3         // ...
4     }
5 }
```

# RETURNS A JOB

```
1 val job: Job = launch {  
2     // ...  
3 }  
4 if (job.isActive) {  
5     // react to coroutine state  
6 }  
7 job.cancel()  
8 job.join()
```

# DEFERRED IS ALSO A JOB

```
1 val result: Deferred<Int> = async {  
2     5  
3 }  
4 val value = result.await() // 5  
5 val job: Job = result  
6 job.cancel()
```

# CANCELLING COROUTINES



# EXCEPTION

Cancels whole scope.

```
1 launch {
2     launch {
3         // cancelled
4     }
5     throw Exception()
6 }
7 launch {
8     // cancelled
9 }
10 // cancelled
```

# MANUAL CANCELLATION

Cancels only self and children.

```
1 launch {
2     launch {
3         // cancelled
4     }
5     cancel()
6 }
7 launch {
8     // runs fine
9 }
10 // runs fine
```

# SUPERVISORJOB

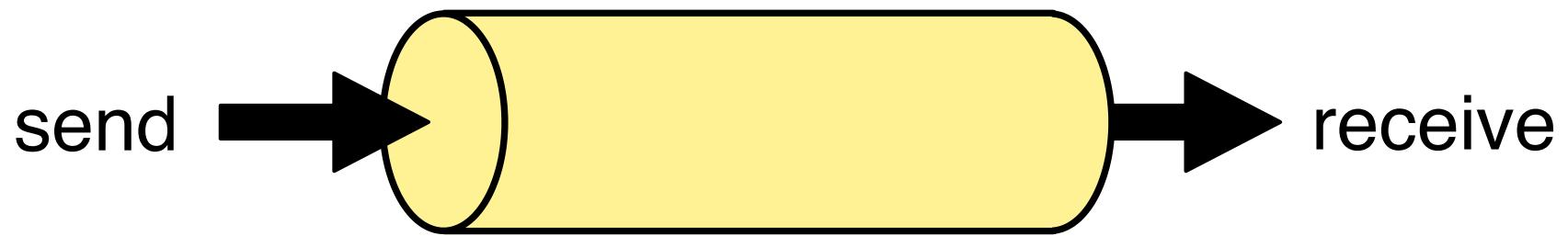
Exception cancellation is contained in children.

```
1 supervisorScope {  
2     launch {  
3         launch {  
4             // cancelled  
5         }  
6         throw Exception()  
7     }  
8     launch {  
9         // runs fine  
10    }  
11 }
```



# CHANNELS

# SUSPENDING METHODS



# SUSPENDING METHODS



```
suspend fun send(element: E)  
suspend fun receive(): E
```

# DEFAULT CAPACITY

```
1 runBlocking {  
2     val channel = Channel<Int>()  
3     channel.send(7)  
4     println(channel.receive())  
5 }
```

# DEFAULT CAPACITY

```
1 runBlocking {  
2     val channel = Channel<Int>()  
3     channel.send(7) // suspends forever  
4     println(channel.receive())  
5 }
```

# DEFAULT CAPACITY

```
1 runBlocking {  
2     val channel = Channel<Int>()  
3     launch {  
4         channel.send(7)  
5     }  
6     println(channel.receive())  
7 }
```

# DEFAULT CAPACITY

```
1 runBlocking {  
2     val channel = Channel<Int>(capacity = 1)  
3     channel.send(7)  
4     println(channel.receive())  
5 }
```

## CAN BRIDGE ACROSS CONTEXTS

Channel instances are not bound to coroutine scopes.

```
1 val channel = Channel<Int>()
```

# CAN BRIDGE ACROSS CONTEXTS

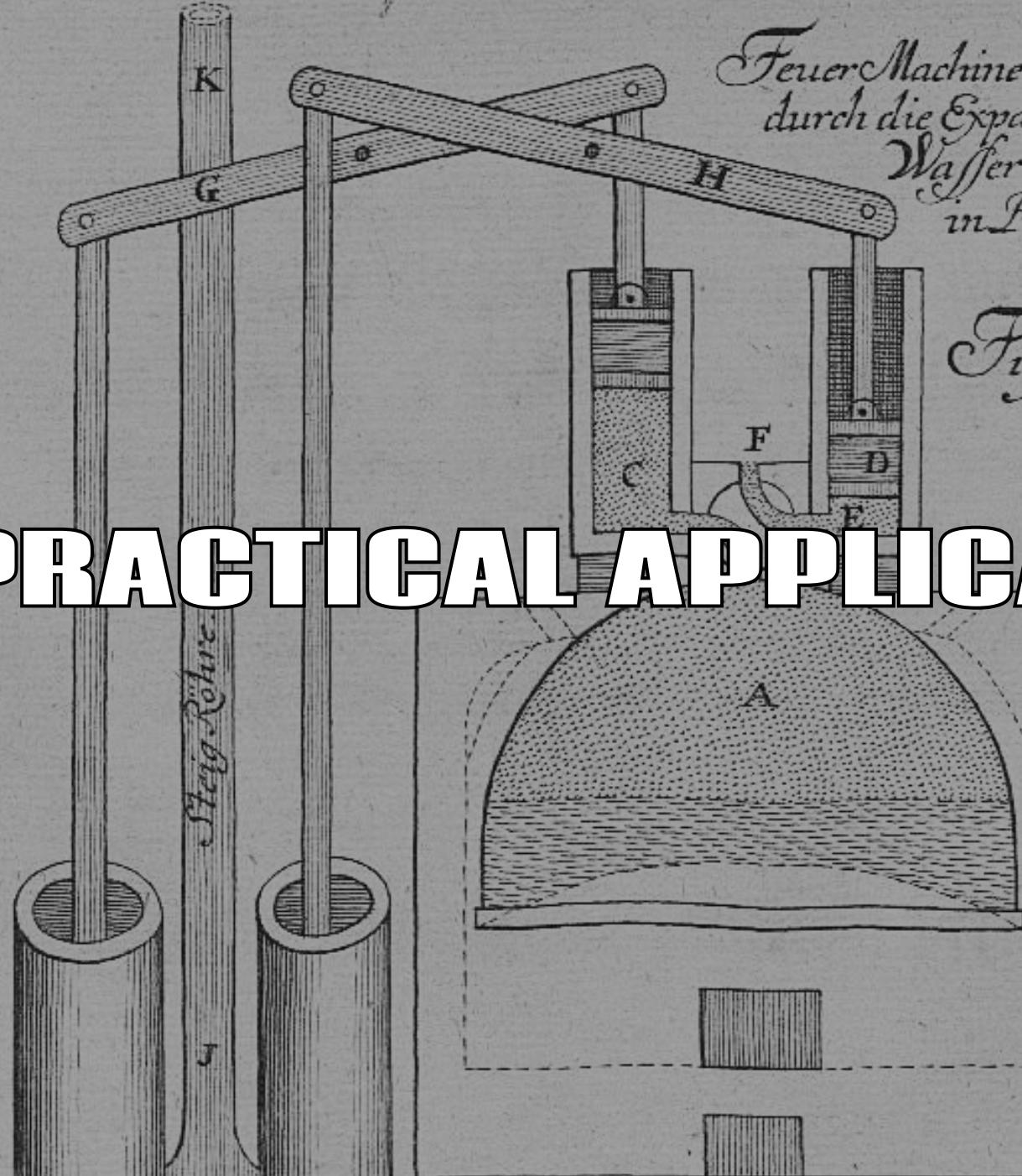
Channel instances are not bound to coroutine scopes.

```
1 val channel = Channel<Int>()
2 scope1.launch {
3     var sum = 0
4     for (message in channel) {
5         sum += message
6     }
7     println(sum)
8 }
```

# CAN BRIDGE ACROSS CONTEXTS

Channel instances are not bound to coroutine scopes.

```
1 val channel = Channel<Int>()
2 scope1.launch {
3     var sum = 0
4     for (message in channel) {
5         sum += message
6     }
7     println(sum)
8 }
9 scope2.launch {
10     channel.send(1)
11     channel.close()
12 }
```



# PRACTICAL APPLICATIONS

*Hijdraulie P. 2.*

SOCIALISTS IN UNION SQ. 51012

# WORKER POOLS

Agitate-  
Educate-  
Organize.

ONE  
BIG  
UI

THE WORKERS—  
FIGHT THE WARS—  
—The—  
BOSSSES REAP  
THE PROFITS.

40  
MILITIAMEN  
IN GUARD HOUSE  
WONT  
DO DUTY

# HTTP WORKERS

## HTTP WORKERS

**Problem:** calling slow services exhausts thread pool

## HTTP WORKERS

**Problem:** calling slow services exhausts thread pool

**Old solution:** separate thread pool per service

## HTTP WORKERS

**Problem:** calling slow services exhausts thread pool

**Old solution:** separate thread pool per service

**Problem:** a few slow calls can block service

# NON-BLOCKING HTTP

Non-blocking clients (e.g. `java.net.http`) can be easily used from coroutines.

```
1 suspend fun get(url: String): HttpResponse<String> =
2     HttpClient
3         .newHttpClient()
4         .sendAsync(
5             HttpRequest.newBuilder(URI(url)).GET().build(),
6             HttpResponse.BodyHandlers.ofString()
7         )
8         .await()
```

# NO MORE THREAD POOLS

```
1 supervisorScope {  
2     repeat(1_000) {  
3         launch {  
4             get("https://httpbin.org/delay/5")  
5                 .also {  
6                     println(it.statusCode())  
7                 }  
8             }  
9         }  
10    }
```

# WEB SERVERS



# ONE THREAD PER REQUEST

Classic Java web server way of handling requests.

More requests, more threads, more resources consumed.

# KTOR

One (or more) coroutines per request.

```
1 get("/balance/{customerId}") {
2     findAccounts(call.parameters["customerId"])
3     .map {
4         async { // launch + return value
5             getBalance(it)
6         }
7     }
8     .sumOf {
9         it.await()
10    }
11 }
```

# KTOR

Every call can take as much time as it likes.

```
1 get("/sales.csv") {
2     call.respondBytesWriter {
3         getSoldItemLines() // produce lines on demand
4             .forEach {
5                 writeAvailable(it.toByteBuffer())
6             }
7     }
8 }
```

A black and white photograph of a man and a woman in period clothing in an ornate room. The woman, on the left, wears a long, flowing gown with a high collar and a necklace; she is barefoot and has her right arm extended. The man, on the right, wears a light-colored waistcoat over a white shirt with a bow tie, dark trousers, and dark shoes; he is holding a cane and looking towards the woman.

**ACTORS**

## ACTOR MODEL

When an actor receives a message, it can (in any order)

- send messages to other actors
- create new actors
- modify its behaviour

# ENCAPSULATING STATE

```
1 actor<CounterAction> {
2     var counter = 0 // <- mutable state
3     for (message in channel) {
4         when (message) {
5             is CounterAction.Add -> {
6                 counter += message.value
7             }
8             is CounterAction.Get -> {
9                 message.answer.complete(counter)
10            }
11        }
12    }
13 }
```

# FAN OUT

```
1 actor<ShoppingCartPriceRequest> {
2     for (message in channel) {
3         val aggregator = aggregatorActor(message.result)
4         getShoppingCart(message.id)
5             .foreach {
6                 aggregator.send(ItemPriceRequest(it))
7             }
8         aggregator.close()
9     }
10 }
```

# Examples and slides:

<https://github.com/weaselflink/exploring-kotlin-coroutines>

