Structured Concurrency and the Myth of Multithreading

by Gustavo Stingelin

1. IO vs CPU Bound
2. Is Node.js Multi-thread?
3. Structured Concurrency
4. Pizzeria

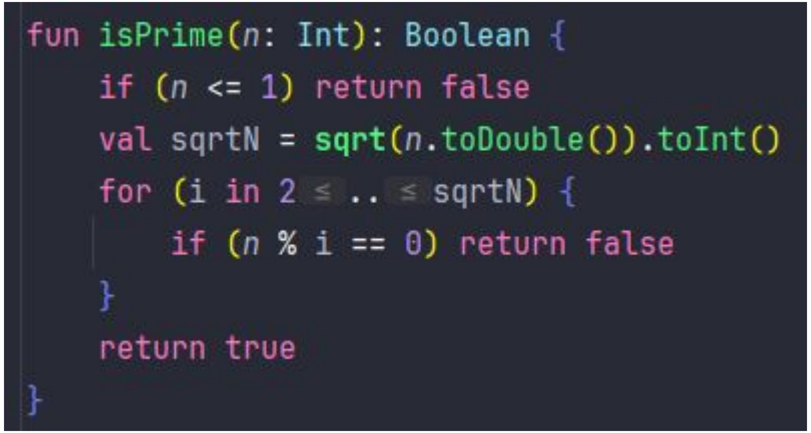
-30
01:24

IO Bound vs CPU Bound

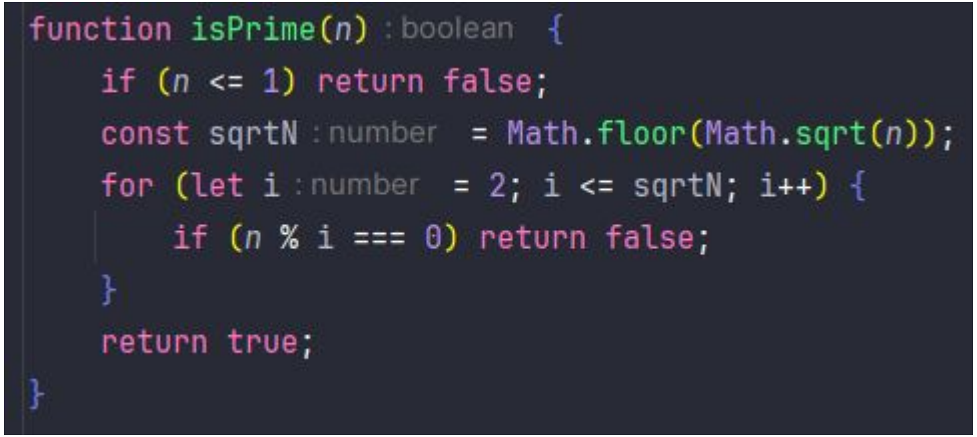
CPU Bound - wash dishes

- Image and video processing
- Encryption and Hashing
- Financial and scientific calculations
- Sorting and searching algorithms

CPU Bound - prime numbers



```
fun isPrime(n: Int): Boolean {
    if (n <= 1) return false
    val sqrtN = sqrt(n.toDouble()).toInt()
    for (i in 2 ≤ .. ≤ sqrtN) {
        if (n % i == 0) return false
    }
    return true
}
```

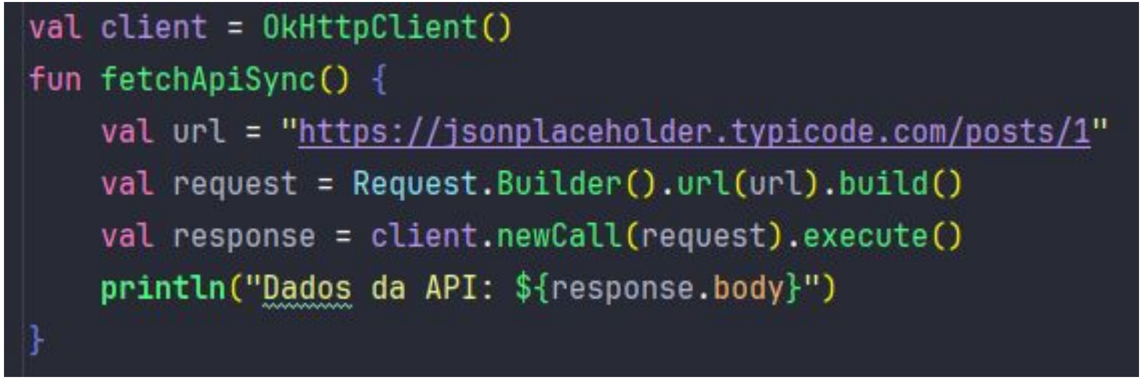


```
function isPrime(n) : boolean  {
    if (n <= 1) return false;
    const sqrtN : number  = Math.floor(Math.sqrt(n));
    for (let i : number  = 2; i <= sqrtN; i++) {
        if (n % i === 0) return false;
    }
    return true;
}
```

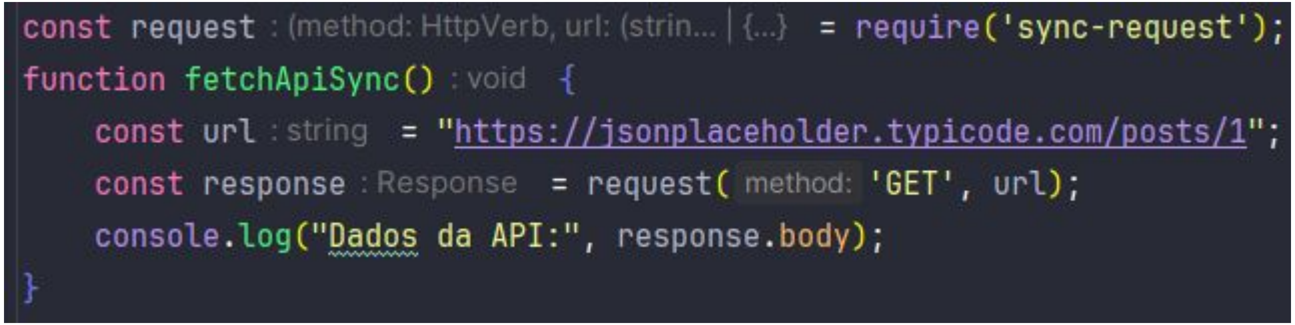
IO Bound - await the cook

- Reading and writing to disk
- Calls to external APIs
- Database queries
- Streaming media

IO Bound - await the cook



```kotlin
val client = OkHttpClient()
fun fetchApiSync() {
    val url = "https://jsonplaceholder.typicode.com/posts/1"
    val request = Request.Builder().url(url).build()
    val response = client.newCall(request).execute()
    println("Dados da API: ${response.body}")
}
```



```javascript
const request : (method: HttpVerb, url: (strin... | {...}  = require('sync-request');
function fetchApiSync() : void  {
    const url : string  = "https://jsonplaceholder.typicode.com/posts/1";
    const response : Response  = request( method: 'GET', url);
    console.log("Dados da API:", response.body);
}
```

IO Bound at Scale

blocking IO - multi thread/process 1990~2005

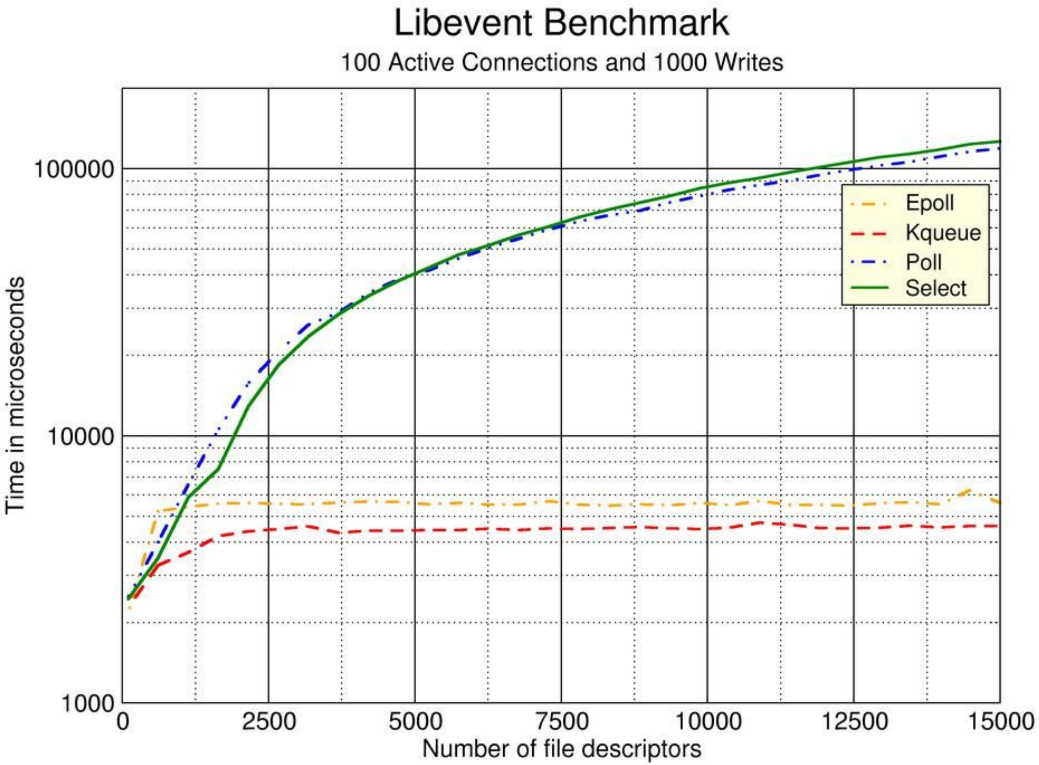
- New thread for each request
- High context switch
- Low efficiency
- High cost per thread

blocking IO - multi thread/process 1990~2005

APACHE
HTTP SERVER PROJECT
Java
Apache Tomcat
php
Microsoft
ASP
JBoss
by Red Hat
GlassFish

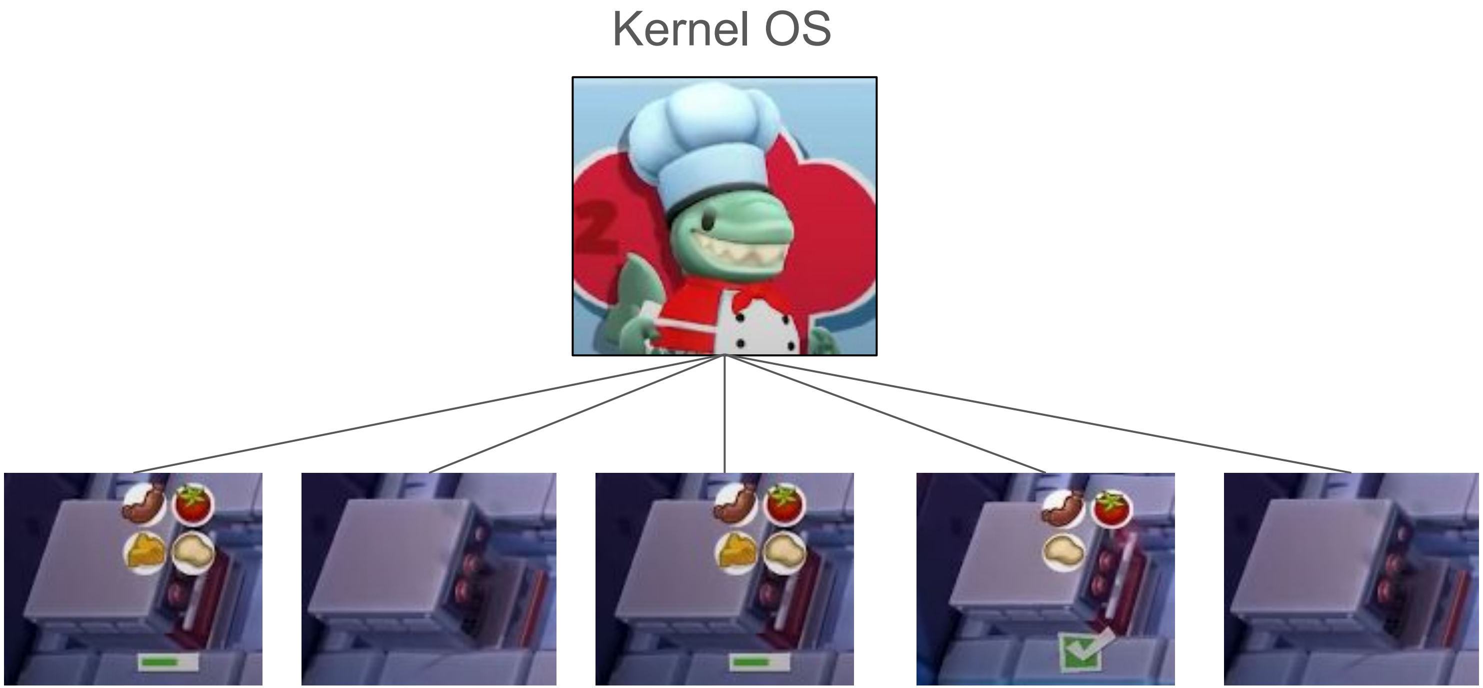
Non-blocking IO

- 80s - Unix - select()
- 90s - Unix - poll()
- 00s - Unix - epoll()



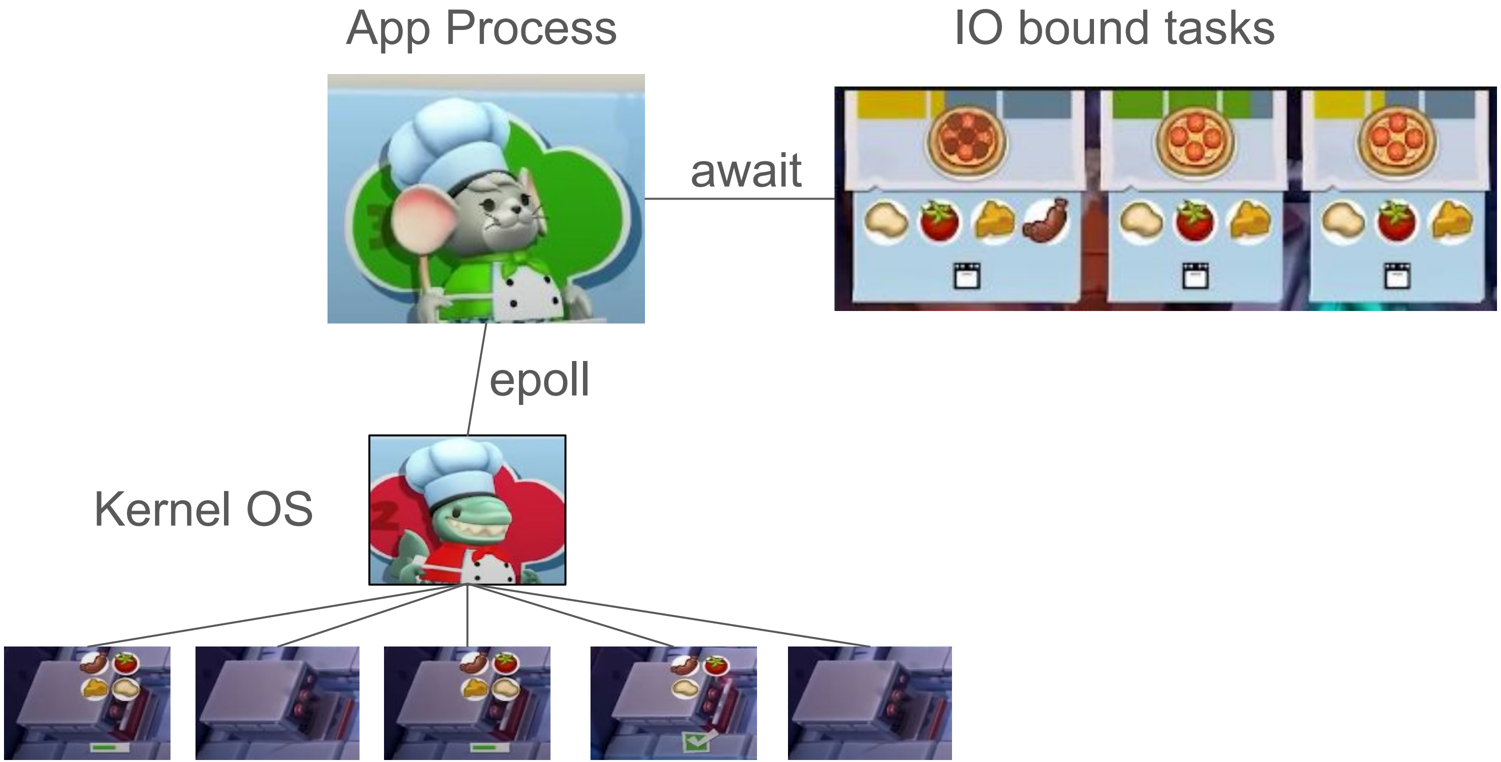
Source: cnblogs.com

Non-blocking IO



Kernel OS

Non-blocking IO



App Process

IO bound tasks

await

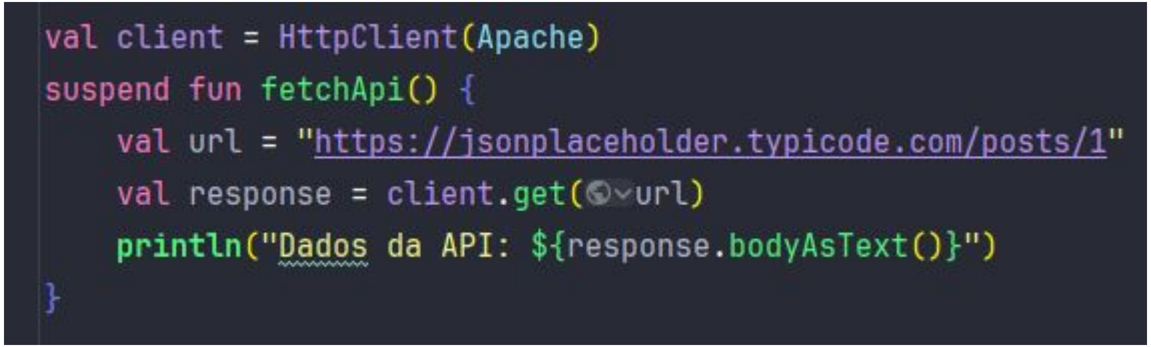
epoll

Kernel OS

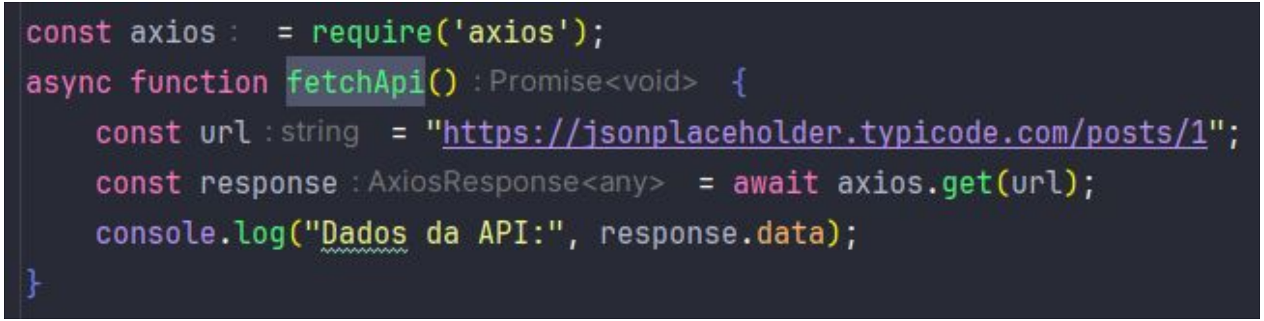
Non-blocking IO

- 2002 - Java NIO
- 2004 - Nginx, Jetty/Netty (Java)
- 2007 - Apache Tomcat (Java)
- 2009 - Node.js, Akka (Scala/Java), Vert.x (Java)
- 2012 - Golang
- 2014 - asyncio (Python)
- 2015 - CIO (Kotlin)
- 2016 - ASP.NET Core (C#)

Non-blocking IO

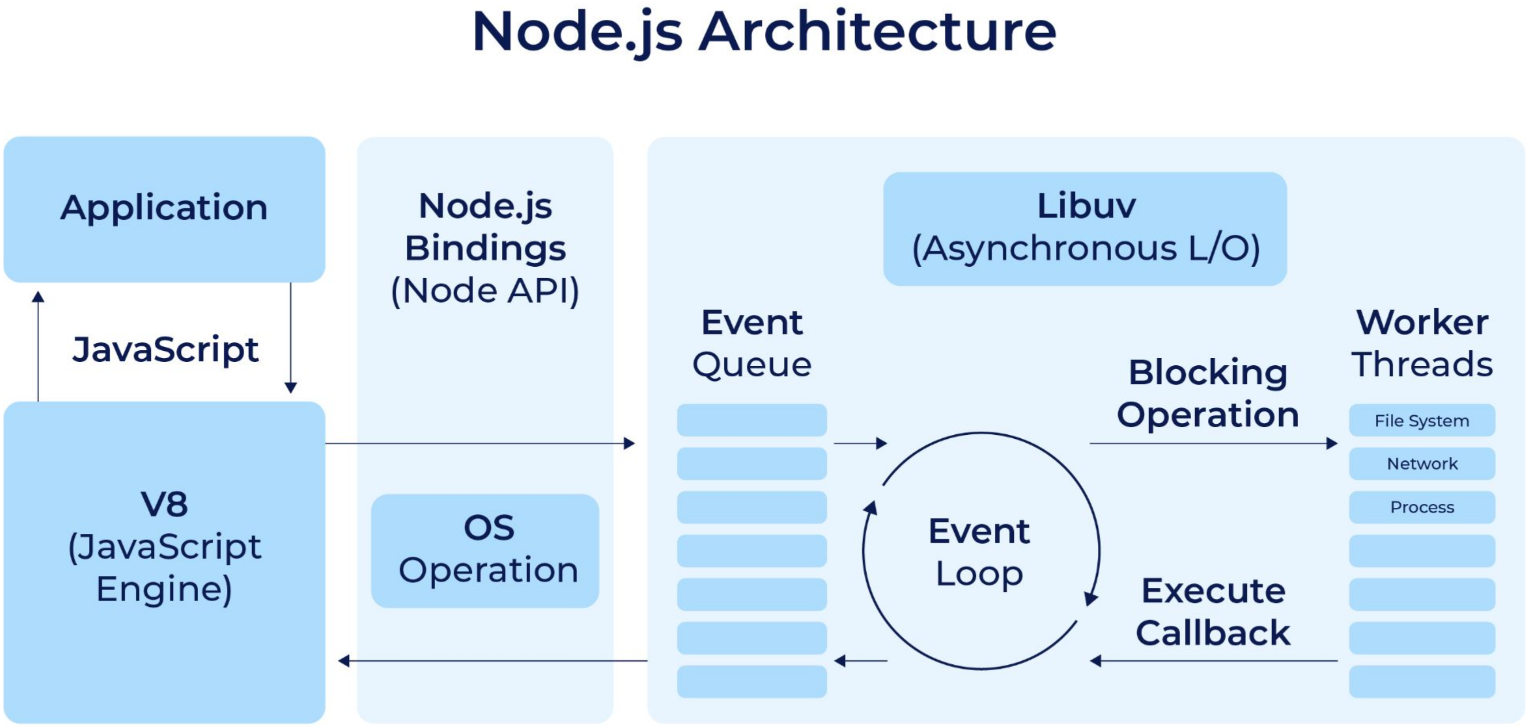


```kotlin
val client = HttpClient(Apache)
suspend fun fetchApi() {
    val url = "https://jsonplaceholder.typicode.com/posts/1"
    val response = client.get(url)
    println("Dados da API: ${response.bodyAsText()}")
}
```

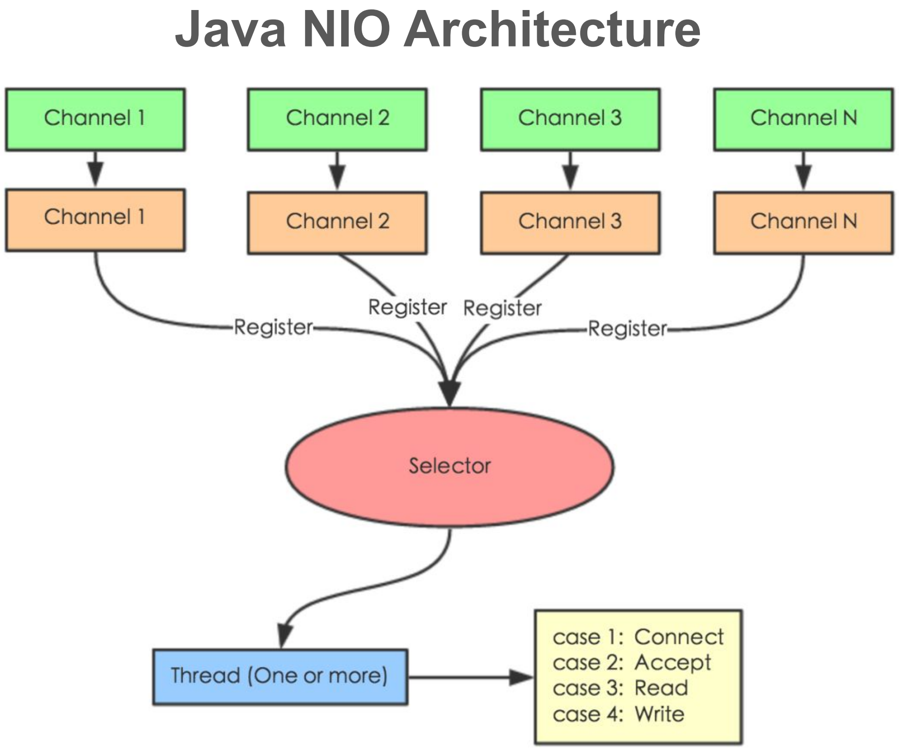


```javascript
const axios :  = require('axios');
async function fetchApi() : Promise<void>  {
    const url : string  = "https://jsonplaceholder.typicode.com/posts/1";
    const response : AxiosResponse<any>  = await axios.get(url);
    console.log("Dados da API:", response.data);
}
```

Non-blocking IO



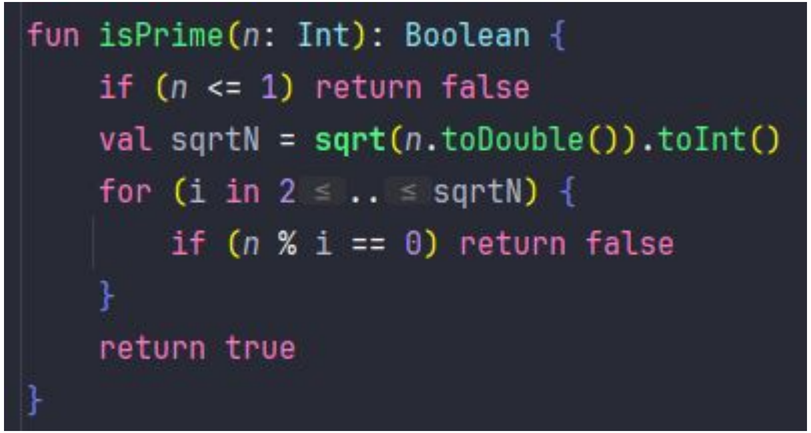
Non-blocking IO



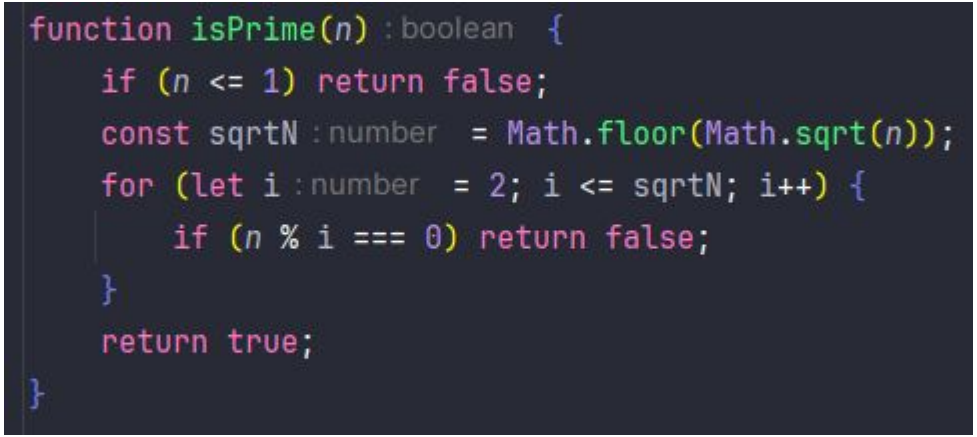
NIO Selector Model

IO Bound + CPU Bound

CPU Bound - prime numbers

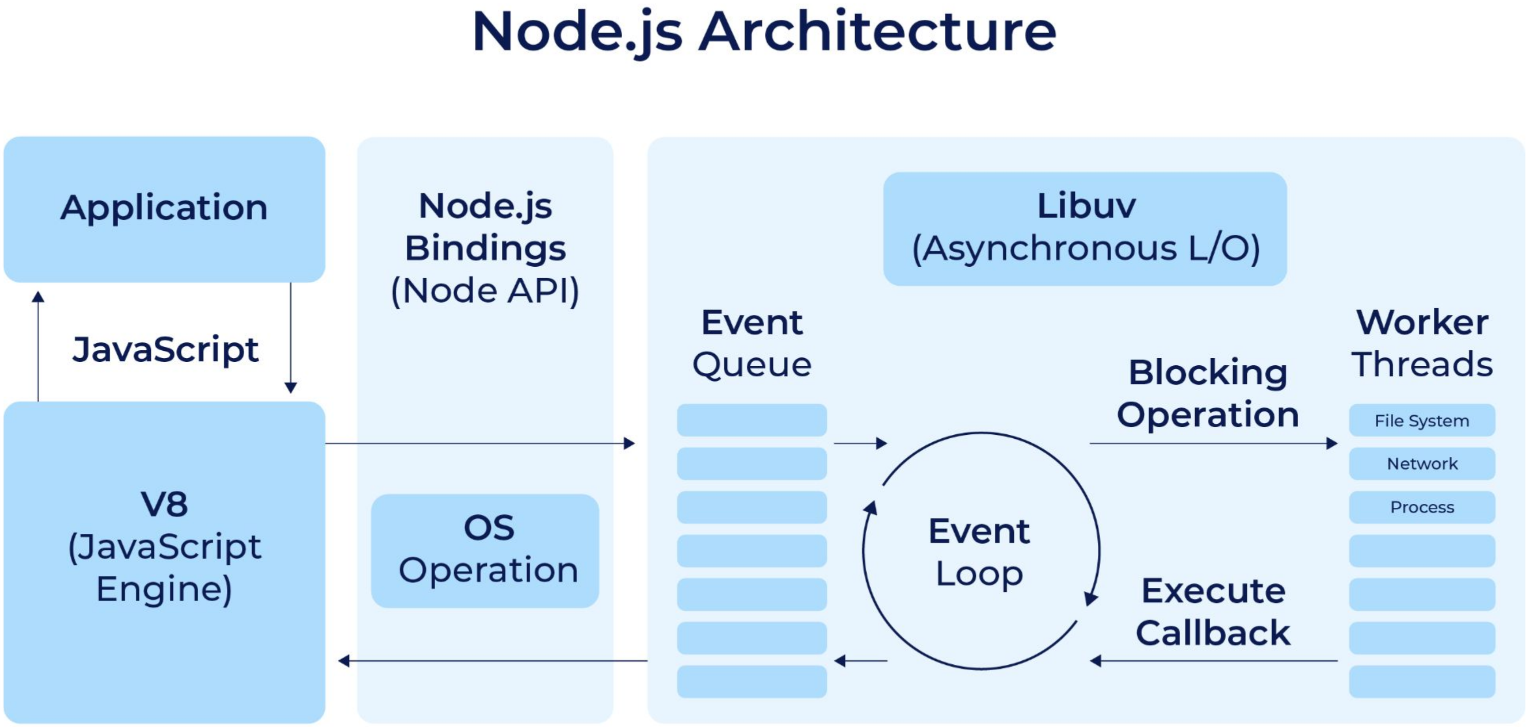


```kotlin
fun isPrime(n: Int): Boolean {
    if (n <= 1) return false
    val sqrtN = sqrt(n.toDouble()).toInt()
    for (i in 2 ≤ .. ≤ sqrtN) {
        if (n % i == 0) return false
    }
    return true
}
```

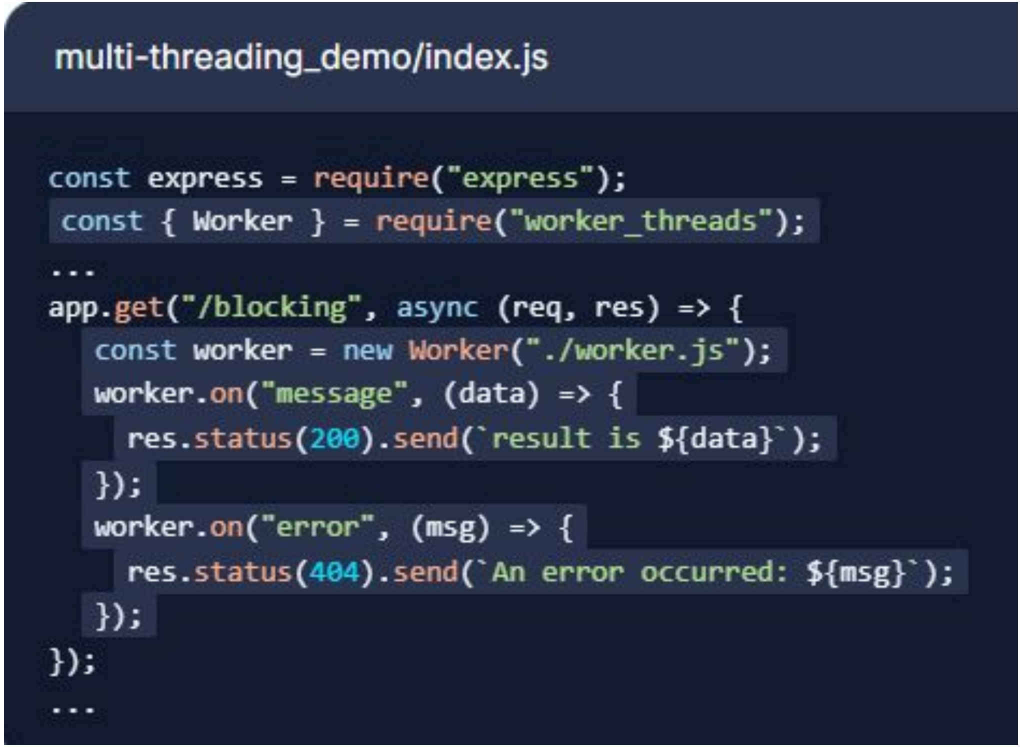


```javascript
function isPrime(n) : boolean  {
    if (n <= 1) return false;
    const sqrtN : number  = Math.floor(Math.sqrt(n));
    for (let i : number  = 2; i <= sqrtN; i++) {
        if (n % i === 0) return false;
    }
    return true;
}
```

CPU Bound on Node.js



CPU Bound on Node.js

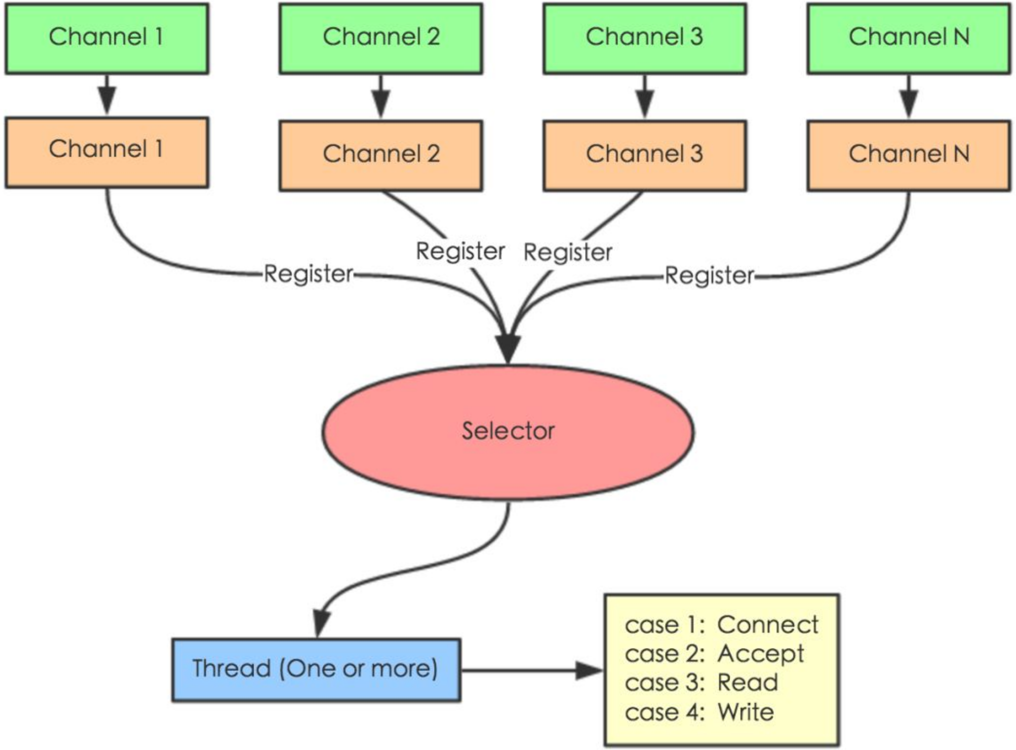


multi-threading_demo/index.js

```
const express = require("express");
const { Worker } = require("worker_threads");
...
app.get("/blocking", async (req, res) => {
  const worker = new Worker("./worker.js");
  worker.on("message", (data) => {
    res.status(200).send(`result is ${data}`);
  });
  worker.on("error", (msg) => {
    res.status(404).send(`An error occurred: ${msg}`);
  });
});
...
```

How To Use Multithreading in Node.js | DigitalOcean

CPU Bound on Java NIO



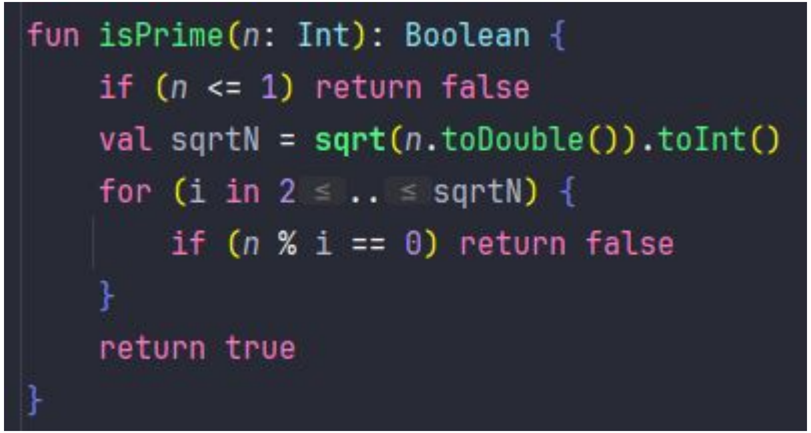
case 1: Connect
case 2: Accept
case 3: Read
case 4: Write

IO Bound + CPU Bound
at Scale

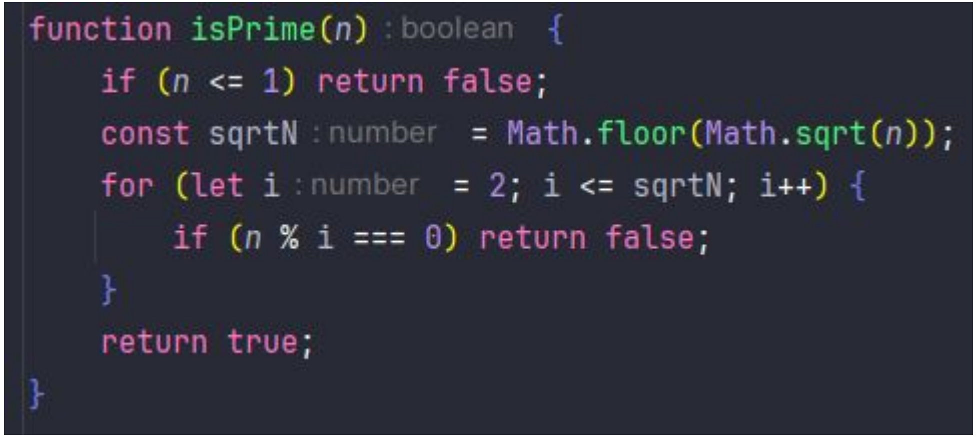
a.k.a.: Structured Concurrency

1233
Tip x 4
00:44

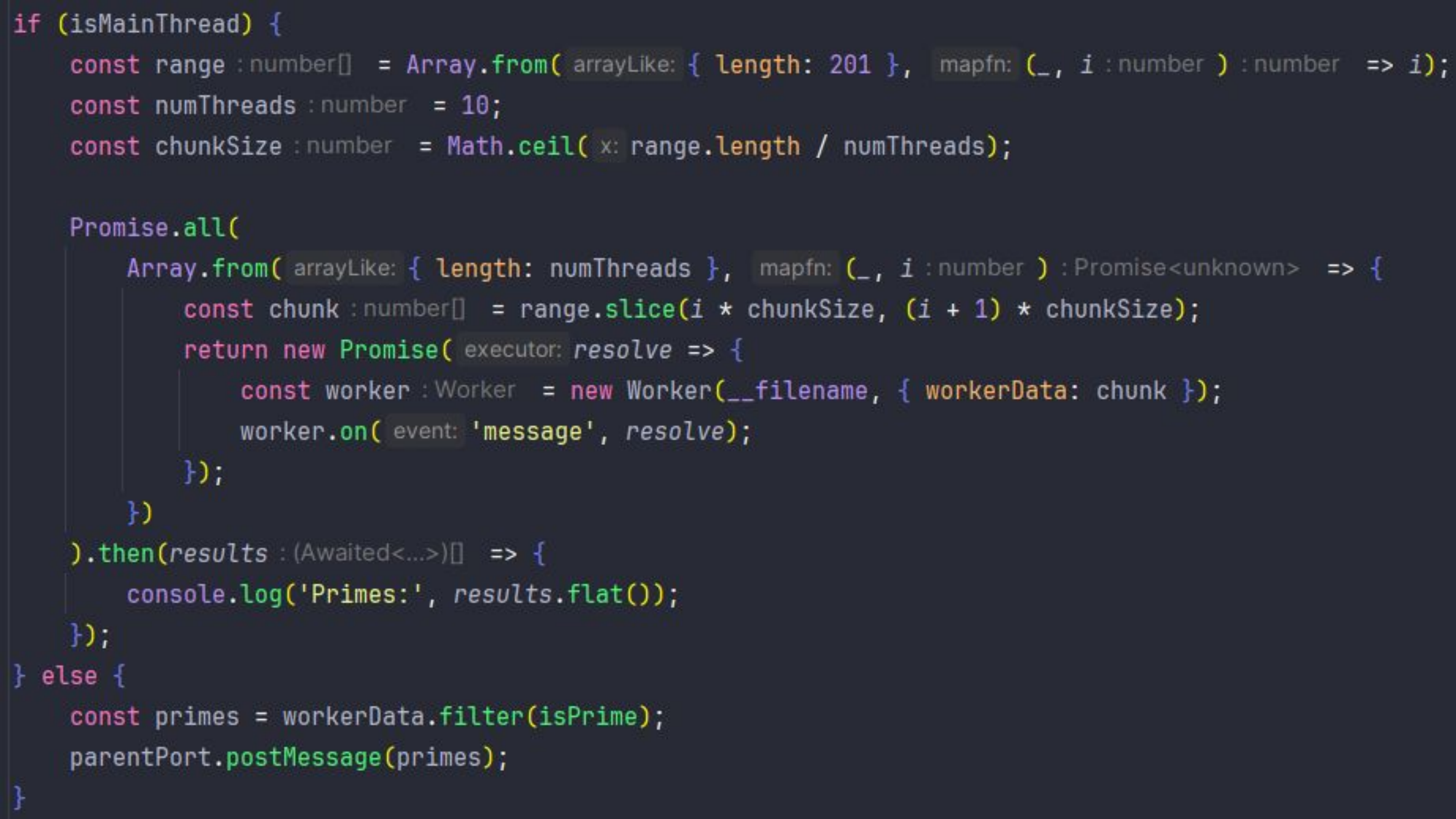
CPU Bound - prime numbers



```kotlin
fun isPrime(n: Int): Boolean {
    if (n <= 1) return false
    val sqrtN = sqrt(n.toDouble()).toInt()
    for (i in 2 ≤ .. ≤ sqrtN) {
        if (n % i == 0) return false
    }
    return true
}
```



```javascript
function isPrime(n) : boolean  {
    if (n <= 1) return false;
    const sqrtN : number  = Math.floor(Math.sqrt(n));
    for (let i : number  = 2; i <= sqrtN; i++) {
        if (n % i === 0) return false;
    }
    return true;
}
```



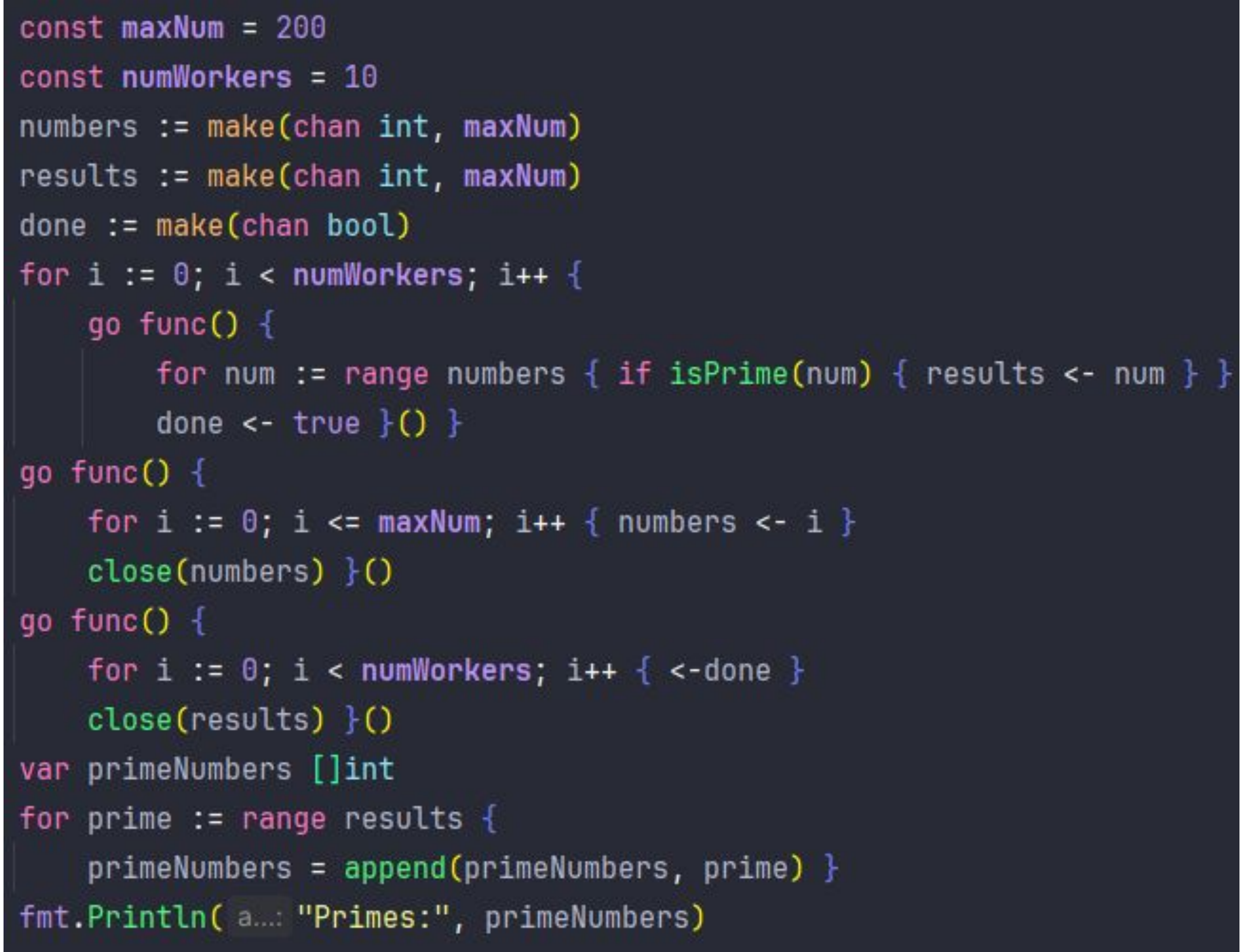
```
if (isMainThread) {
    const range : number[] = Array.from( arrayLike: { length: 201 }, mapfn: (_, i : number ) : number  => i);
    const numThreads : number = 10;
    const chunkSize : number = Math.ceil( x: range.length / numThreads);

    Promise.all(
        Array.from( arrayLike: { length: numThreads }, mapfn: (_, i : number ) : Promise<unknown>  => {
            const chunk : number[] = range.slice(i * chunkSize, (i + 1) * chunkSize);
            return new Promise( executor: resolve => {
                const worker : Worker = new Worker(__filename, { workerData: chunk });
                worker.on( event: 'message', resolve);
            });
        })
    ).then(results : (Awaited<...>)[]  => {
        console.log('Primes:', results.flat());
    });
} else {
    const primes = workerData.filter(isPrime);
    parentPort.postMessage(primes);
}
```

Ryan Dahl, the creator of Node.js:

> *“ I think Node is not the best system to build a massive server web. I would use Go for that. And honestly, that’s the reason why I left Node. It was the realization that: oh, actually, this is not the best server-side system ever”*

Source



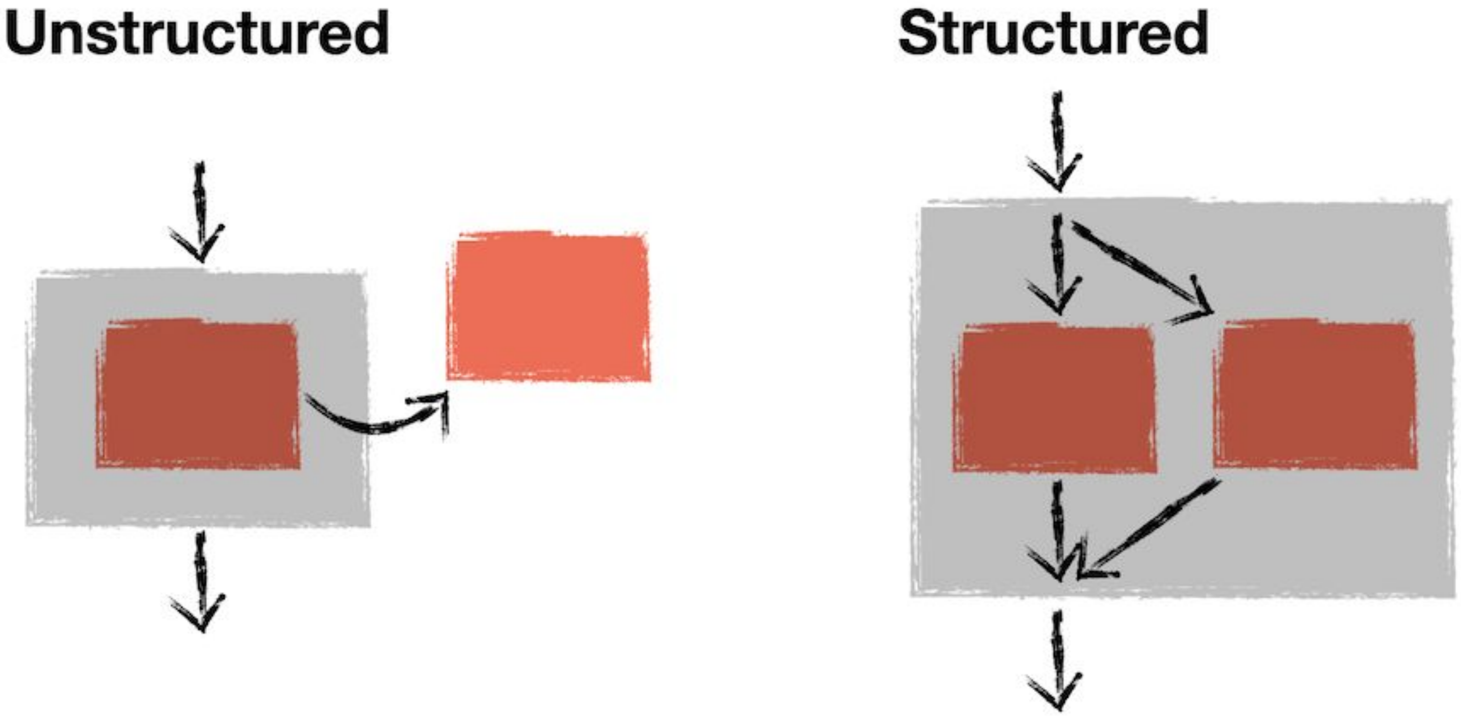
```go
const maxNum = 200
const numWorkers = 10
numbers := make(chan int, maxNum)
results := make(chan int, maxNum)
done := make(chan bool)
for i := 0; i < numWorkers; i++ {
    go func() {
        for num := range numbers { if isPrime(num) { results <- num } }
        done <- true }() }
go func() {
    for i := 0; i <= maxNum; i++ { numbers <- i }
    close(numbers) }()
go func() {
    for i := 0; i < numWorkers; i++ { <-done }
    close(results) }()
var primeNumbers []int
for prime := range results {
    primeNumbers = append(primeNumbers, prime) }
fmt.Println( a...: "Primes:", primeNumbers)
```

JET
BRAINS

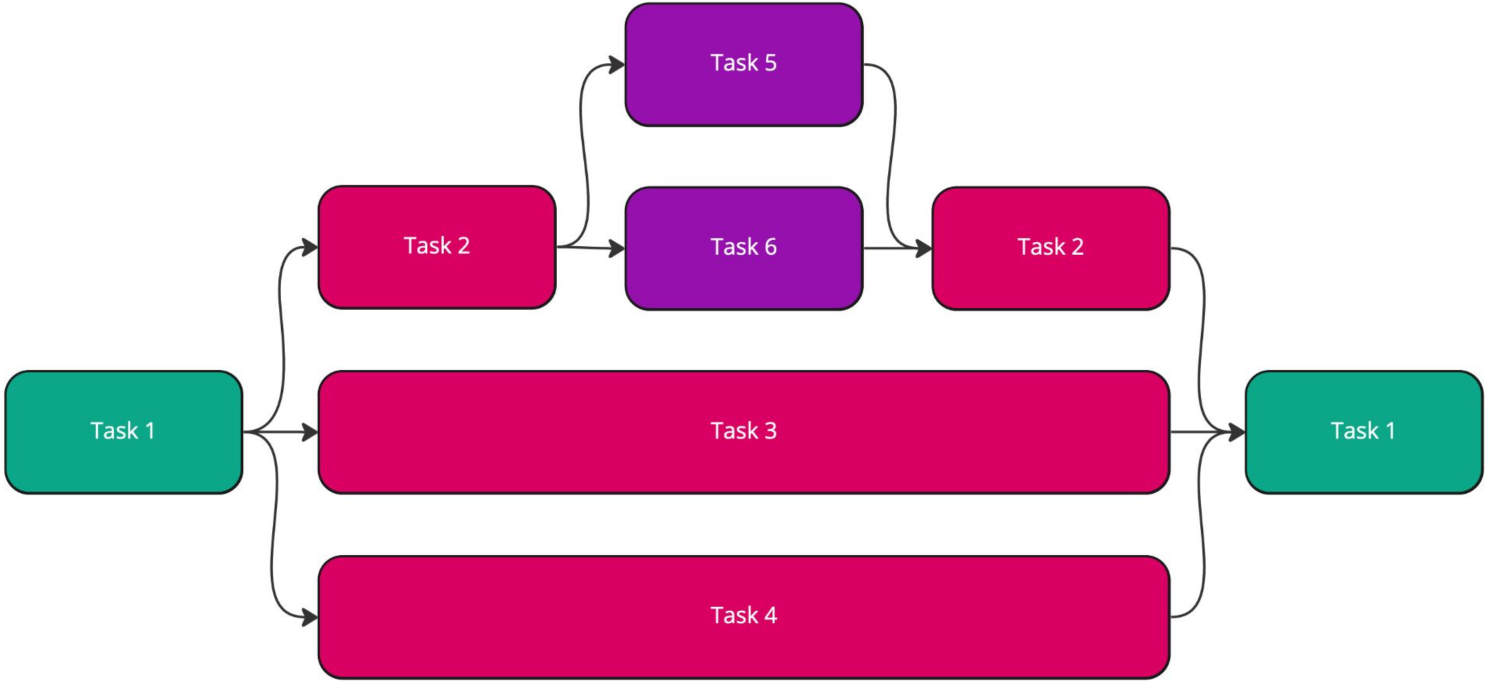
Kotlin 2.0

Fast Smart Multiplatform

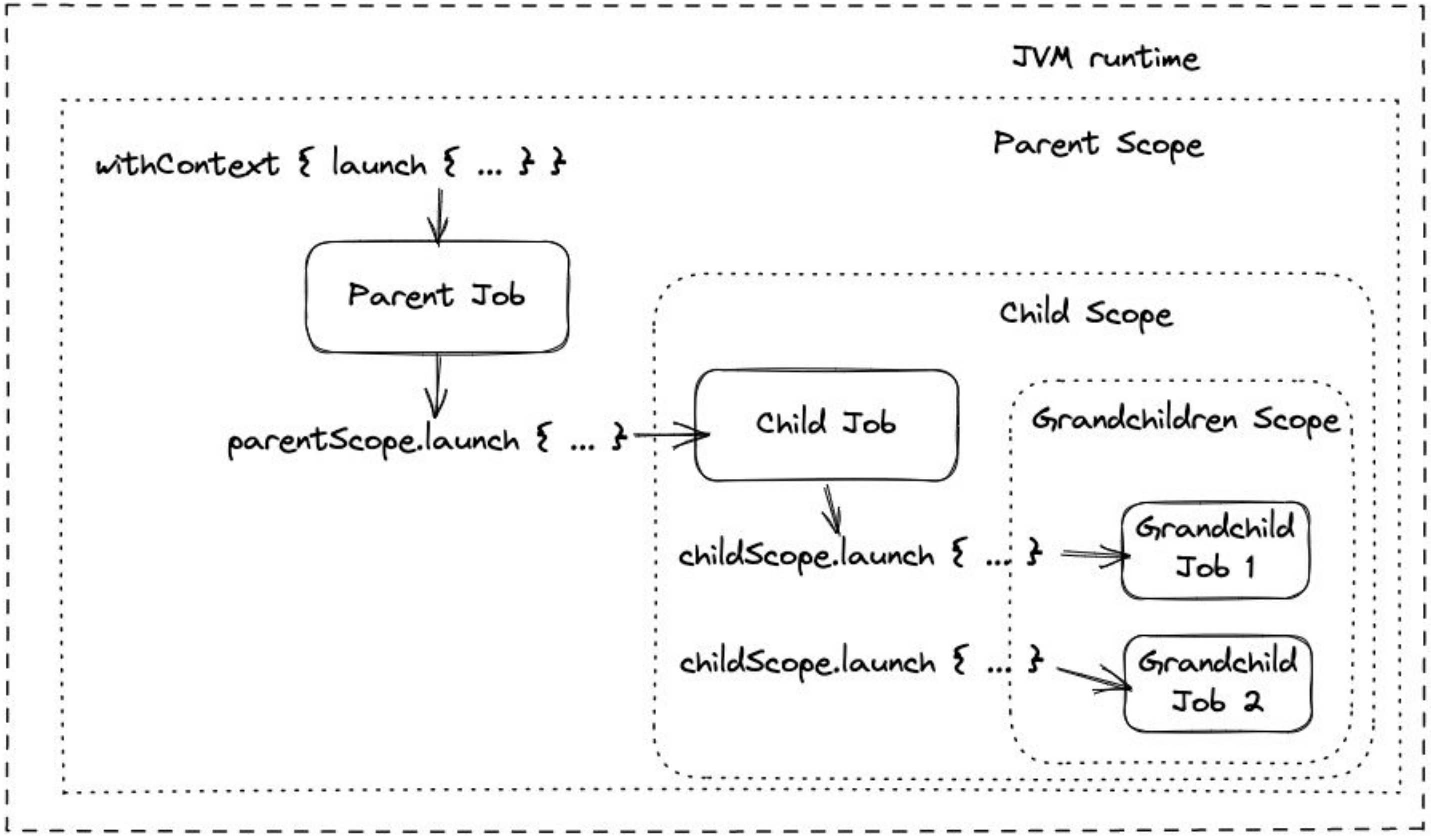
Concurrency



Structured Concurrency

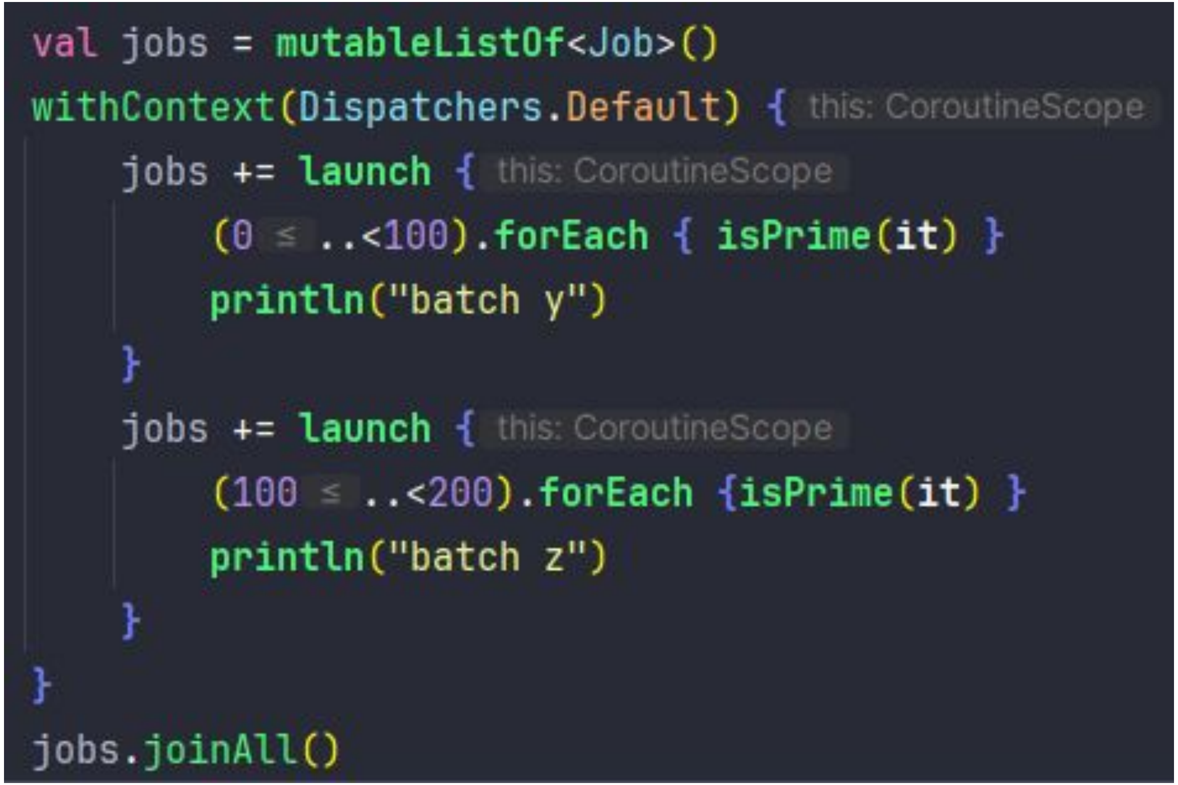


Structured Concurrency



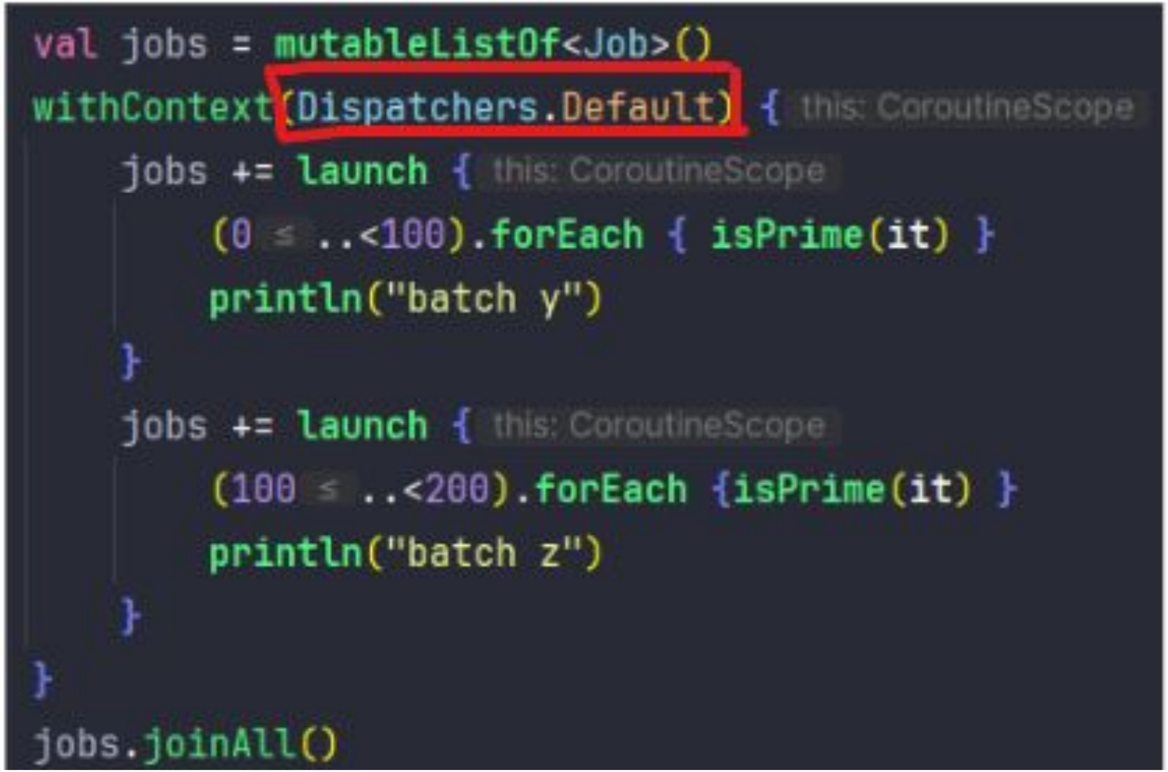
JVM runtime
Parent Scope
withContext { launch { ... } }
Parent Job
parentScope.launch { ... }
Child Scope
Child Job
Grandchildren Scope
childScope.launch { ... }
Grandchild Job 1
childScope.launch { ... }
Grandchild Job 2

Structured Concurrency



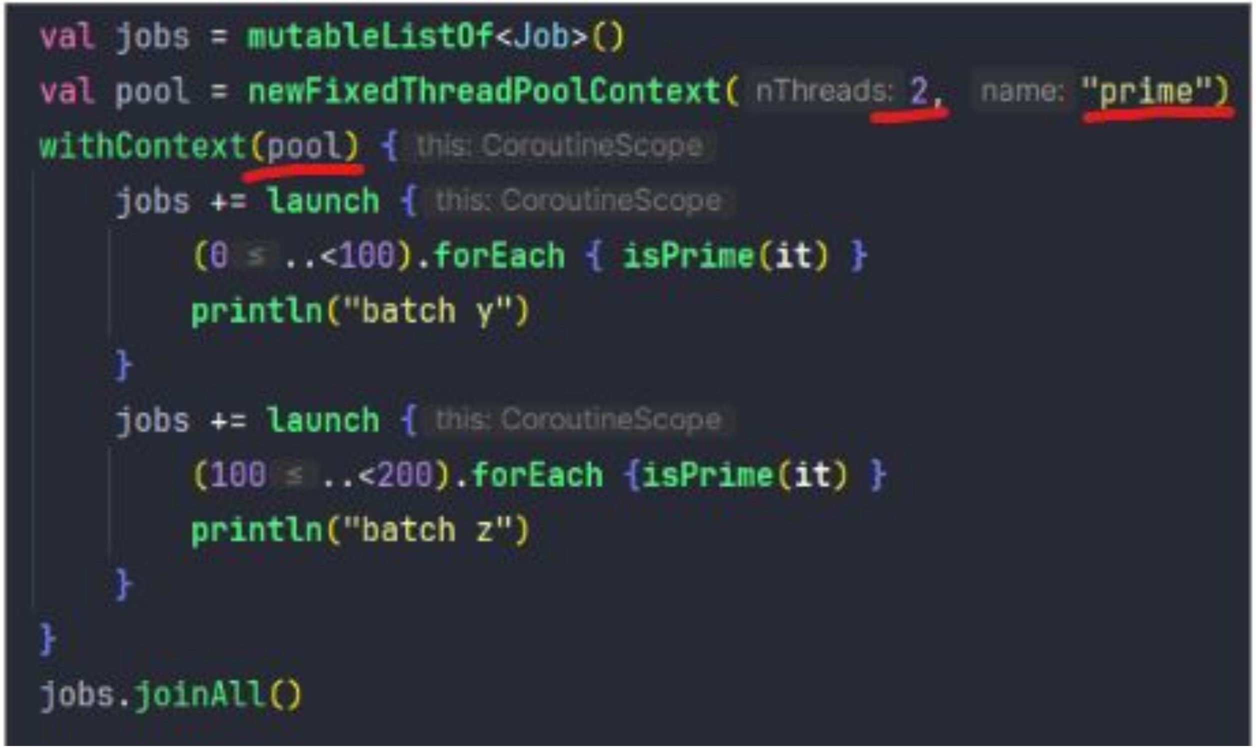
```
val jobs = mutableListOf<Job>()
withContext(Dispatchers.Default) { this: CoroutineScope
    jobs += launch { this: CoroutineScope
        (0 ≤ ..<100).forEach { isPrime(it) }
        println("batch y")
    }
    jobs += launch { this: CoroutineScope
        (100 ≤ ..<200).forEach {isPrime(it) }
        println("batch z")
    }
}
jobs.joinAll()
```

Structured Concurrency



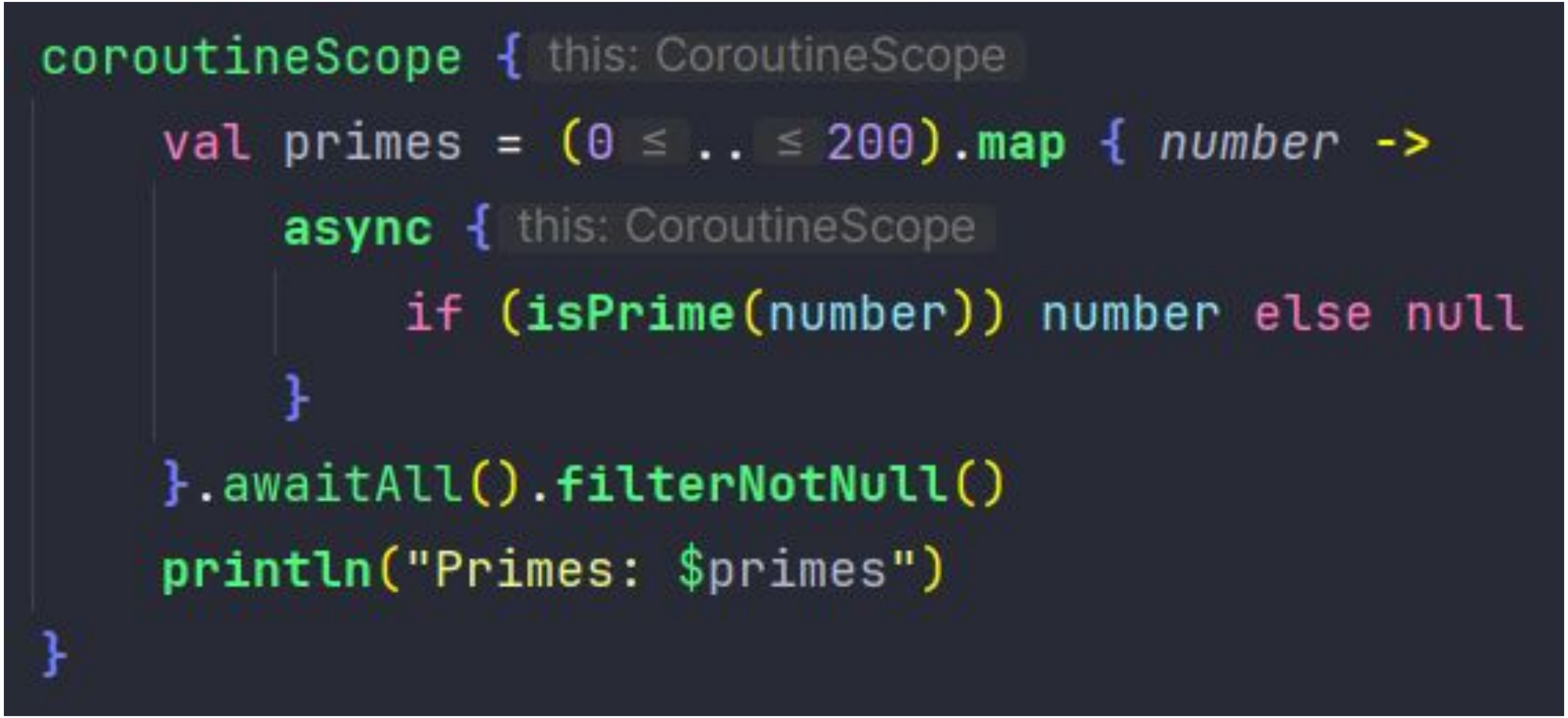
```
val jobs = mutableListOf<Job>()
withContext(Dispatchers.Default) { this: CoroutineScope
    jobs += launch { this: CoroutineScope
        (0 ≤ ..<100).forEach { isPrime(it) }
        println("batch y")
    }
    jobs += launch { this: CoroutineScope
        (100 ≤ ..<200).forEach {isPrime(it) }
        println("batch z")
    }
}
jobs.joinAll()
```

Structured Concurrency



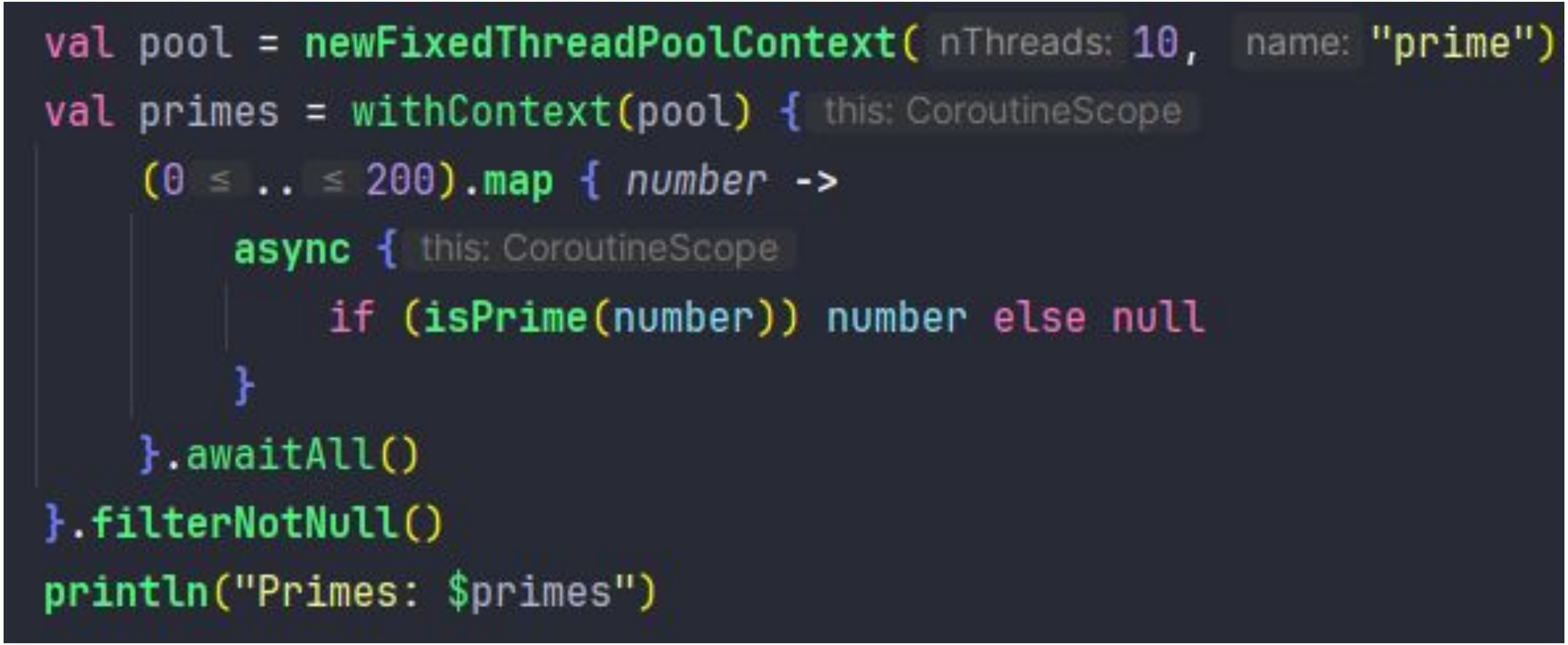
```
val jobs = mutableListOf<Job>()
val pool = newFixedThreadPoolContext( nThreads: 2,  name: "prime")
withContext(pool) { this: CoroutineScope
    jobs += launch { this: CoroutineScope
        (0 ≤ ..<100).forEach { isPrime(it) }
        println("batch y")
    }
    jobs += launch { this: CoroutineScope
        (100 ≤ ..<200).forEach {isPrime(it) }
        println("batch z")
    }
}
jobs.joinAll()
```

Structured Concurrency



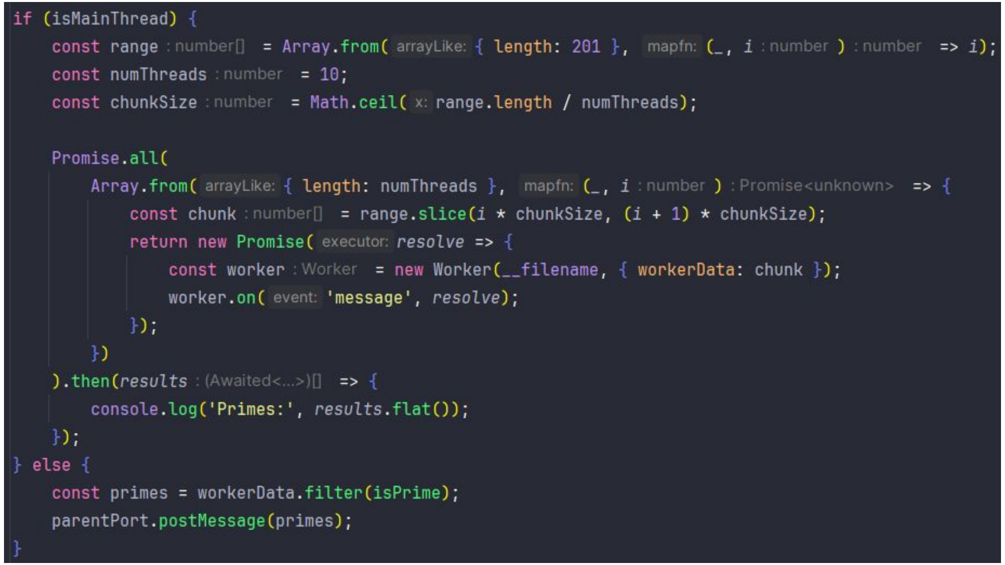
```
coroutineScope { this: CoroutineScope
    val primes = (0 ≤ .. ≤ 200).map { number ->
        async { this: CoroutineScope
            if (isPrime(number)) number else null
        }
    }.awaitAll().filterNotNull()
    println("Primes: $primes")
}
```

Structured Concurrency



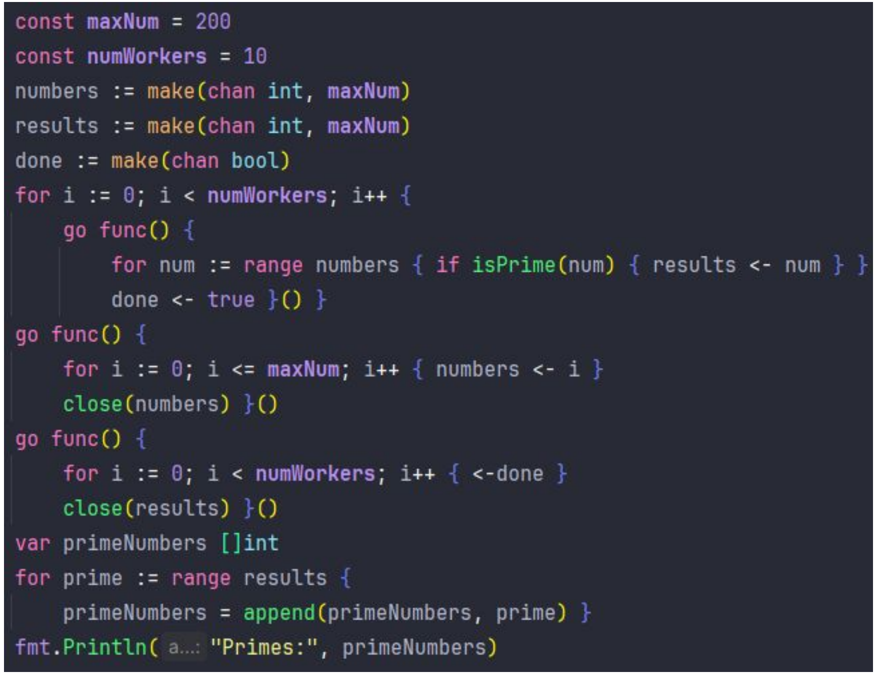
```
val pool = newFixedThreadPoolContext( nThreads: 10,  name: "prime")
val primes = withContext(pool) { this: CoroutineScope
    (0 ≤ .. ≤ 200).map { number ->
        async { this: CoroutineScope
            if (isPrime(number)) number else null
        }
    }.awaitAll()
}.filterNotNull()
println("Primes: $primes")
```

Structured Concurrency

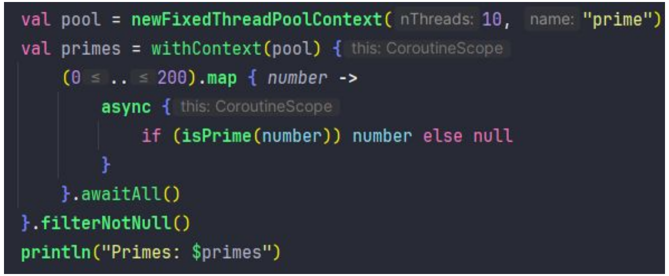


```
if (isMainThread) {
    const range :number[]  = Array.from( arrayLike: { length: 201 },  mapfn: (_, i :number ) :number  => i);
    const numThreads :number  = 10;
    const chunkSize :number  = Math.ceil( x: range.length / numThreads);

    Promise.all(
        Array.from( arrayLike: { length: numThreads },  mapfn: (_, i :number ) :Promise<unknown>  => {
            const chunk :number[]  = range.slice(i * chunkSize, (i + 1) * chunkSize);
            return new Promise( executor: resolve => {
                const worker :Worker  = new Worker(__filename, { workerData: chunk });
                worker.on( event: 'message', resolve);
            });
        })
    ).then(results :(Awaited<...>)[]  => {
        console.log('Primes:', results.flat());
    });
} else {
    const primes = workerData.filter(isPrime);
    parentPort.postMessage(primes);
}
```



```
const maxNum = 200
const numWorkers = 10
numbers := make(chan int, maxNum)
results := make(chan int, maxNum)
done := make(chan bool)
for i := 0; i < numWorkers; i++ {
    go func() {
        for num := range numbers { if isPrime(num) { results <- num } }
        done <- true }() }
go func() {
    for i := 0; i <= maxNum; i++ { numbers <- i }
    close(numbers) }()
go func() {
    for i := 0; i < numWorkers; i++ { <-done }
    close(results) }()
var primeNumbers []int
for prime := range results {
    primeNumbers = append(primeNumbers, prime) }
fmt.Println( a...: "Primes:", primeNumbers)
```

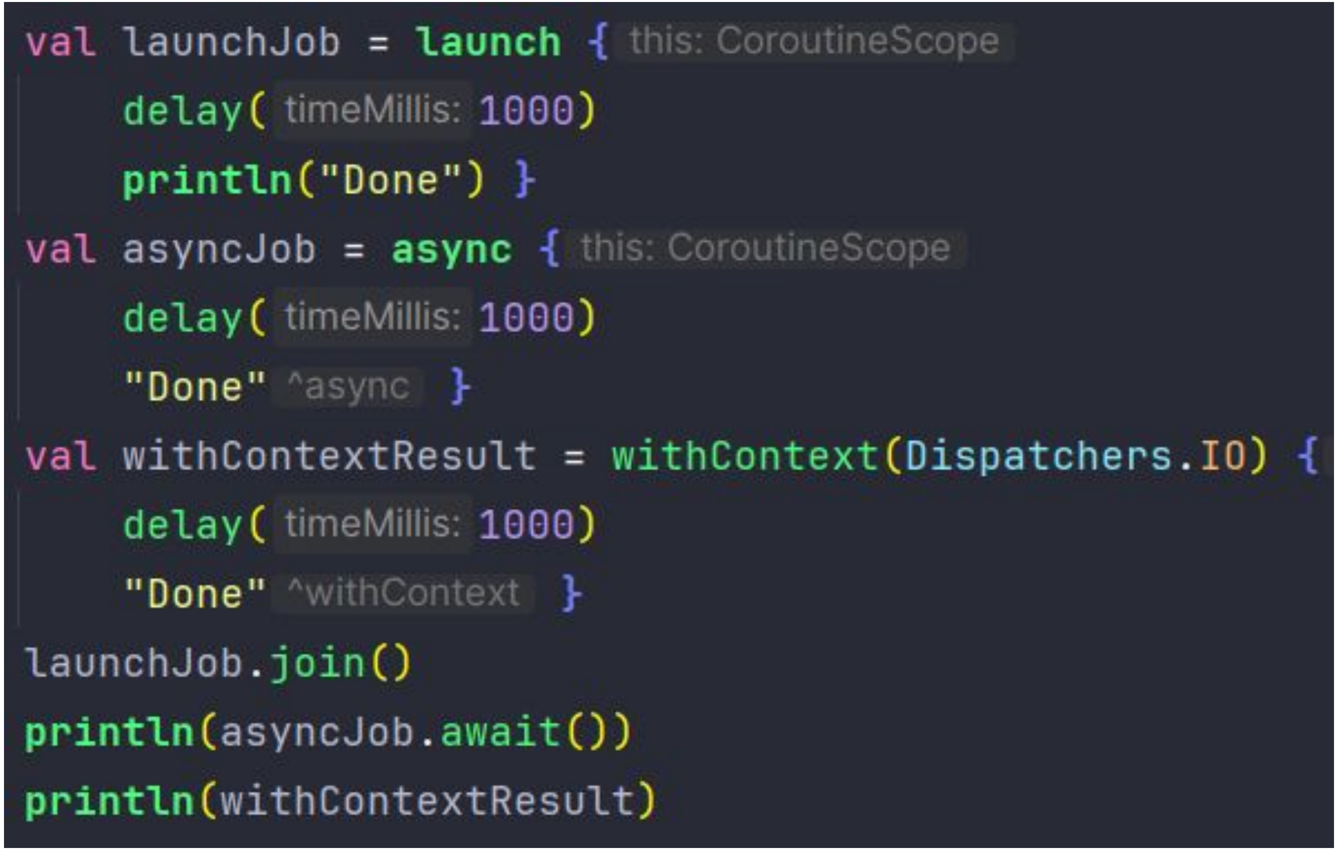


```
val pool = newFixedThreadPoolContext( nThreads: 10,  name: "prime")
val primes = withContext(pool) { this: CoroutineScope
    (0 ≤ .. ≤ 200).map { number ->
        async { this: CoroutineScope
            if (isPrime(number)) number else null
        }
    }.awaitAll()
}.filterNotNull()
println("Primes: $primes")
```

Structured Concurrency - Advanced Topics

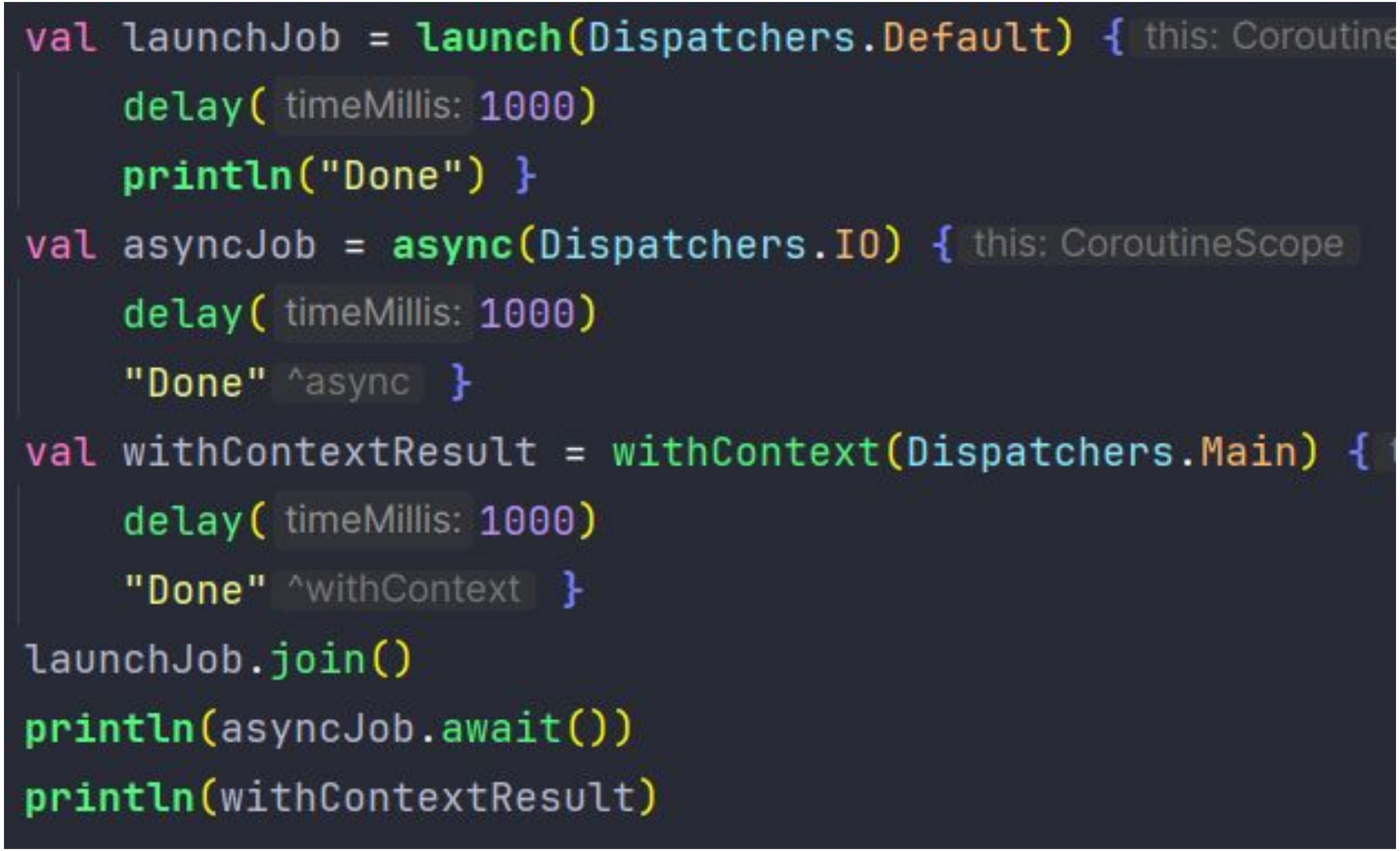
- diffs from async/launch/withContext
- dispatchers and thread pool
- channels
- selects
- flows and sequences
- mutex and semaphore
- suspendable and yield

async/launch/withContext



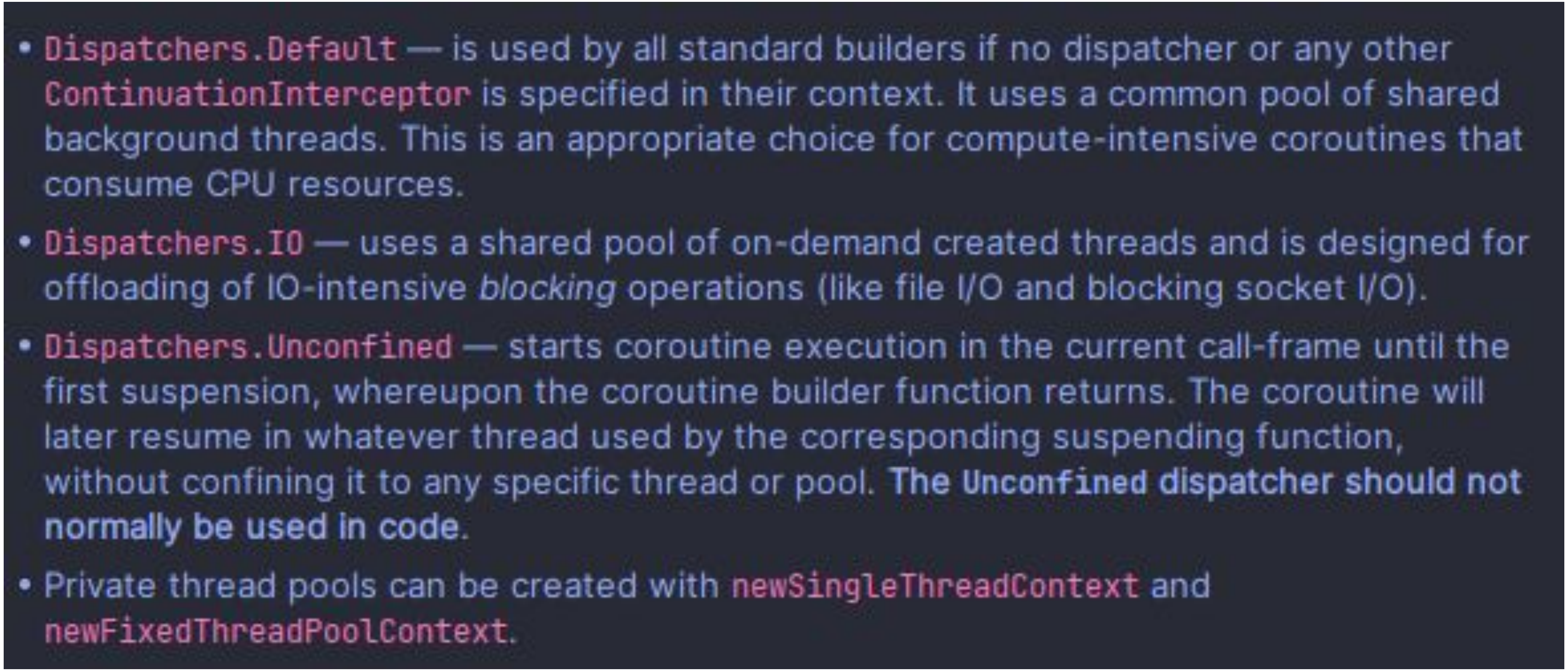
```
val launchJob = launch { this: CoroutineScope
    delay( timeMillis: 1000)
    println("Done") }
val asyncJob = async { this: CoroutineScope
    delay( timeMillis: 1000)
    "Done" ^async }
val withContextResult = withContext(Dispatchers.IO) {
    delay( timeMillis: 1000)
    "Done" ^withContext }
launchJob.join()
println(asyncJob.await())
println(withContextResult)
```

async/launch/withContext



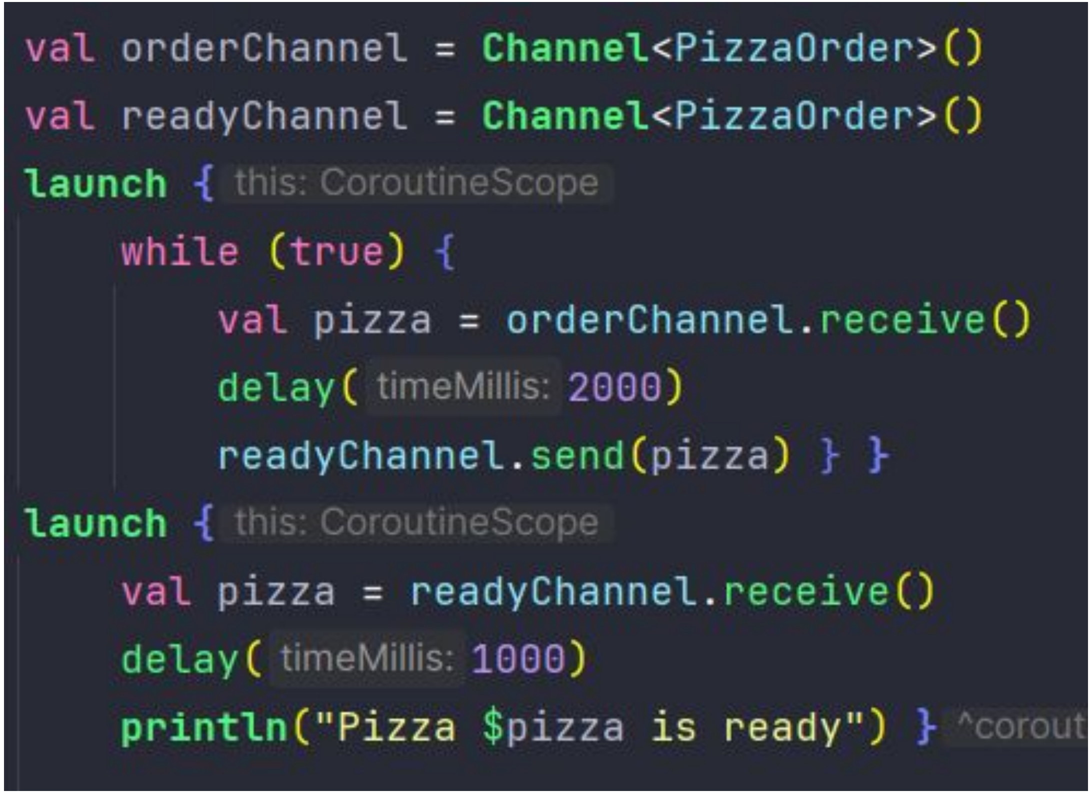
```
val launchJob = launch(Dispatchers.Default) { this: Coroutine
    delay( timeMillis: 1000)
    println("Done") }
val asyncJob = async(Dispatchers.IO) { this: CoroutineScope
    delay( timeMillis: 1000)
    "Done" ^async  }
val withContextResult = withContext(Dispatchers.Main) {
    delay( timeMillis: 1000)
    "Done" ^withContext  }
launchJob.join()
println(asyncJob.await())
println(withContextResult)
```

dispatchers and thread pool



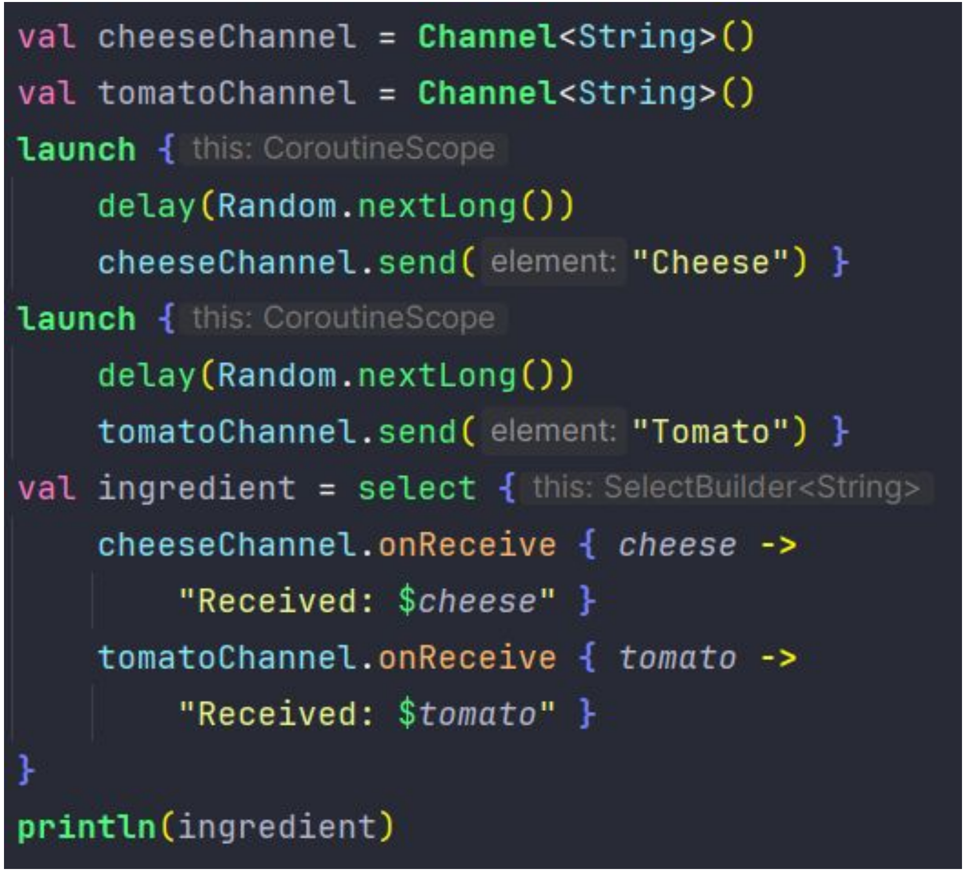
- `Dispatchers.Default` — is used by all standard builders if no dispatcher or any other `ContinuationInterceptor` is specified in their context. It uses a common pool of shared background threads. This is an appropriate choice for compute-intensive coroutines that consume CPU resources.
- `Dispatchers.IO` — uses a shared pool of on-demand created threads and is designed for offloading of IO-intensive *blocking* operations (like file I/O and blocking socket I/O).
- `Dispatchers.Unconfined` — starts coroutine execution in the current call-frame until the first suspension, whereupon the coroutine builder function returns. The coroutine will later resume in whatever thread used by the corresponding suspending function, without confining it to any specific thread or pool. **The `Unconfined` dispatcher should not normally be used in code**.
- Private thread pools can be created with `newSingleThreadContext` and `newFixedThreadPoolContext`.

channels



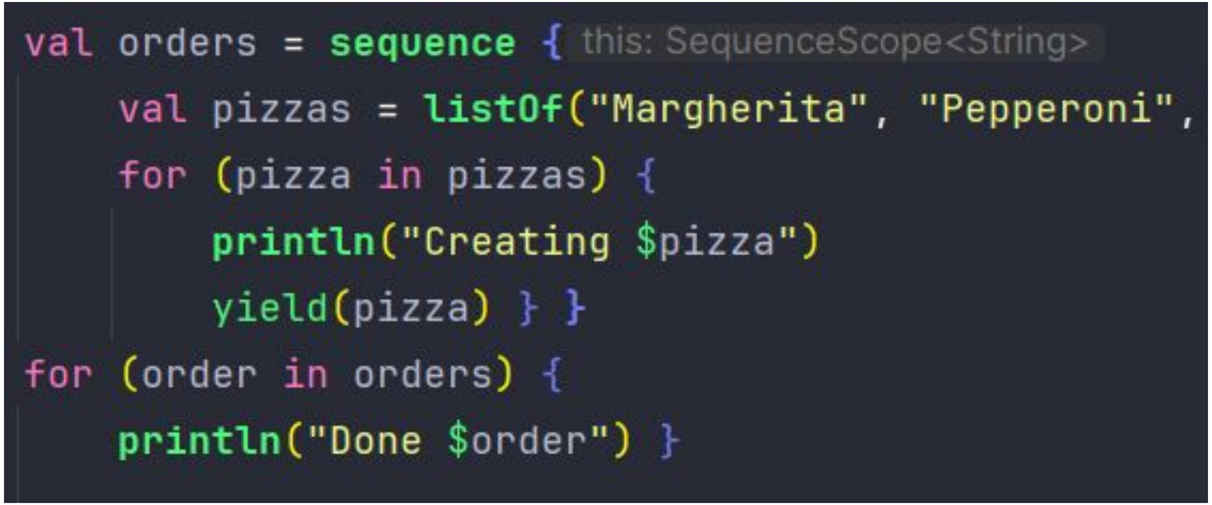
```
val orderChannel = Channel<PizzaOrder>()
val readyChannel = Channel<PizzaOrder>()
launch { this: CoroutineScope
    while (true) {
        val pizza = orderChannel.receive()
        delay( timeMillis: 2000)
        readyChannel.send(pizza) } }
launch { this: CoroutineScope
    val pizza = readyChannel.receive()
    delay( timeMillis: 1000)
    println("Pizza $pizza is ready") } ^corout
```

selects

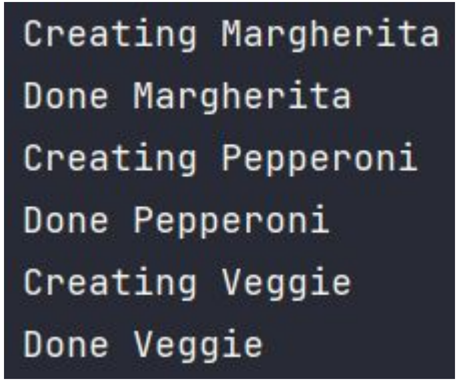


```
val cheeseChannel = Channel<String>()
val tomatoChannel = Channel<String>()
launch { this: CoroutineScope
    delay(Random.nextLong())
    cheeseChannel.send( element: "Cheese") }
launch { this: CoroutineScope
    delay(Random.nextLong())
    tomatoChannel.send( element: "Tomato") }
val ingredient = select { this: SelectBuilder<String>
    cheeseChannel.onReceive { cheese ->
        "Received: $cheese" }
    tomatoChannel.onReceive { tomato ->
        "Received: $tomato" }
}
println(ingredient)
```

flows and sequences

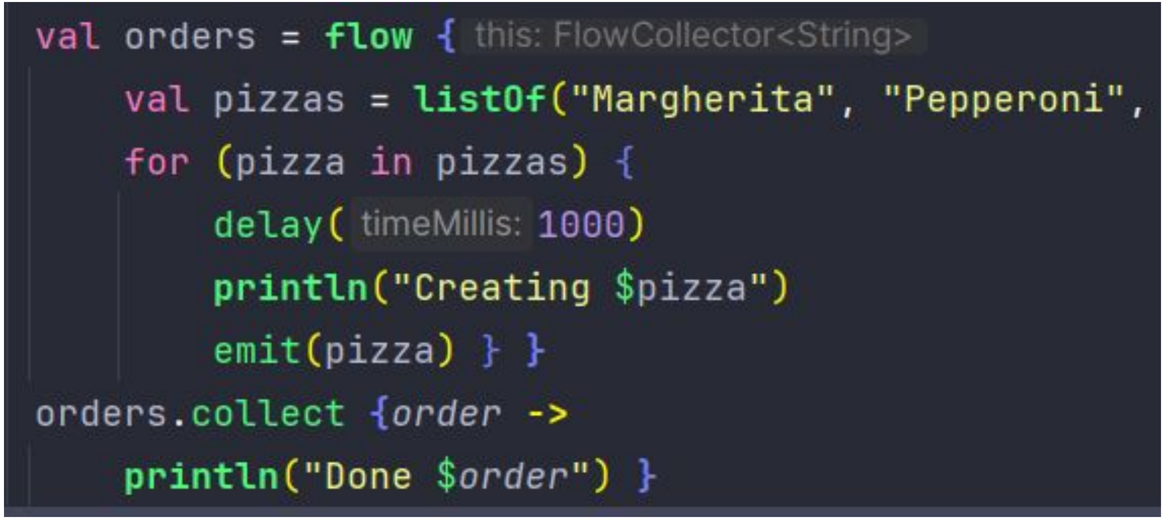


```kotlin
val orders = sequence { this: SequenceScope<String>
    val pizzas = listOf("Margherita", "Pepperoni",
    for (pizza in pizzas) {
        println("Creating $pizza")
        yield(pizza) } }
for (order in orders) {
    println("Done $order") }
```

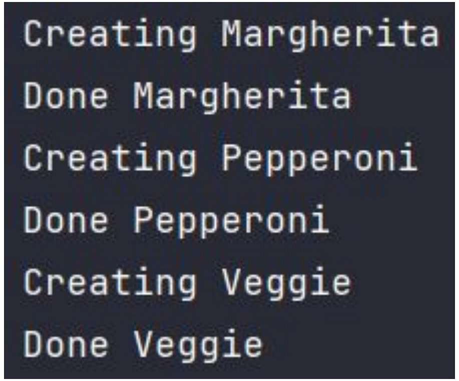


```
Creating Margherita
Done Margherita
Creating Pepperoni
Done Pepperoni
Creating Veggie
Done Veggie
```

flows and sequences

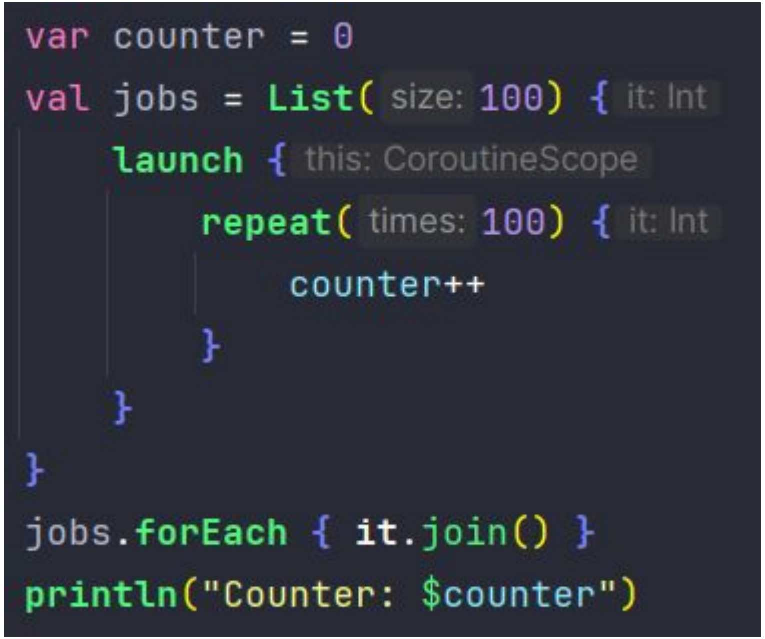


```
val orders = flow { this: FlowCollector<String>
    val pizzas = listOf("Margherita", "Pepperoni",
    for (pizza in pizzas) {
        delay( timeMillis: 1000)
        println("Creating $pizza")
        emit(pizza) } }
orders.collect {order ->
    println("Done $order") }
```

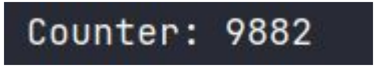


```
Creating Margherita
Done Margherita
Creating Pepperoni
Done Pepperoni
Creating Veggie
Done Veggie
```

mutex and semaphore

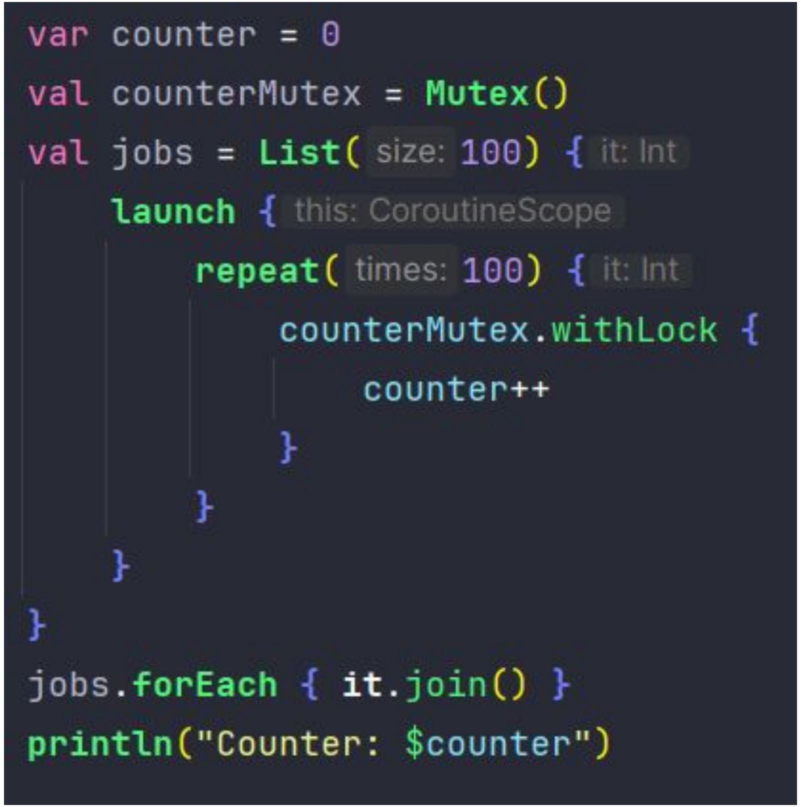


```kotlin
var counter = 0
val jobs = List( size: 100) { it: Int
    launch { this: CoroutineScope
        repeat( times: 100) { it: Int
            counter++
        }
    }
}
jobs.forEach { it.join() }
println("Counter: $counter")
```

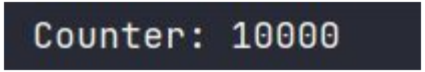


```
Counter: 9882
```

mutex and semaphore

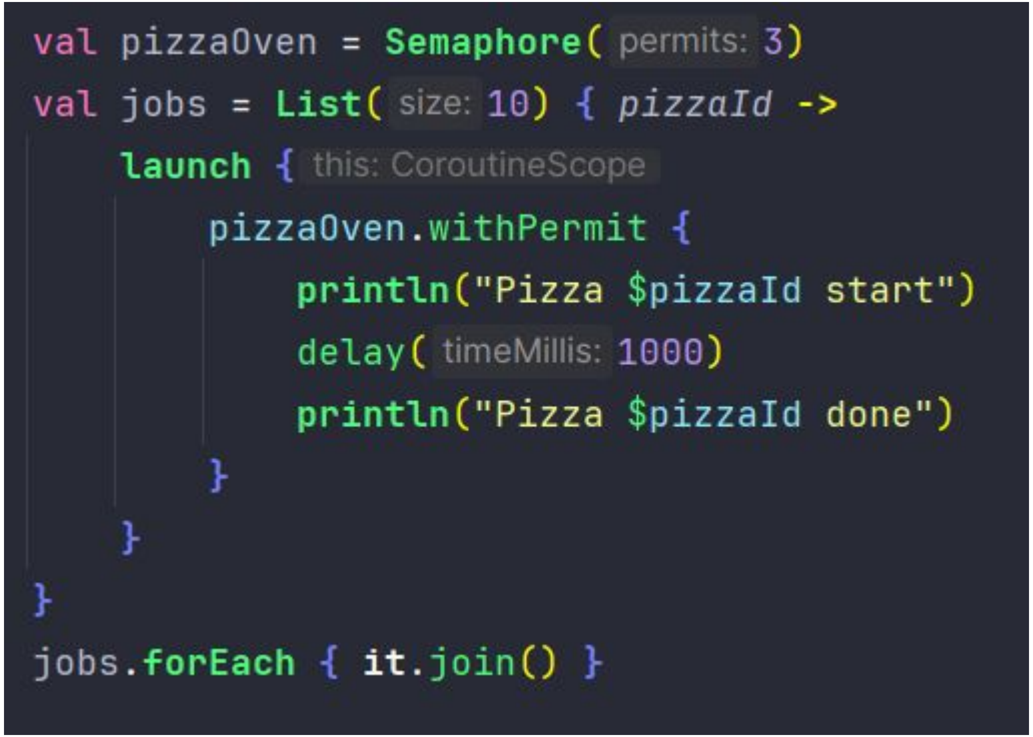


```
var counter = 0
val counterMutex = Mutex()
val jobs = List( size: 100) { it: Int
    launch { this: CoroutineScope
        repeat( times: 100) { it: Int
            counterMutex.withLock {
                counter++
            }
        }
    }
}
jobs.forEach { it.join() }
println("Counter: $counter")
```

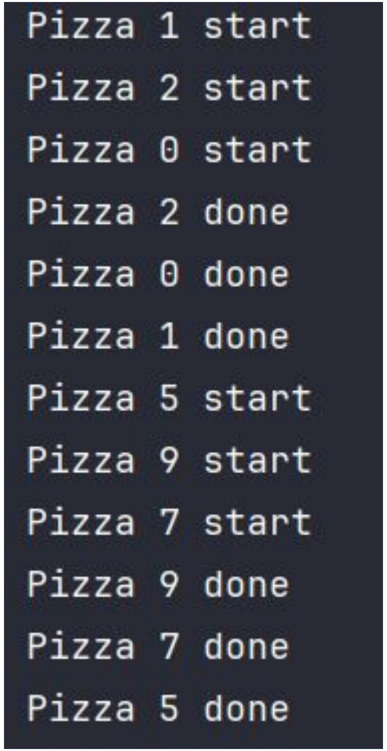


```
Counter: 10000
```

mutex and semaphore

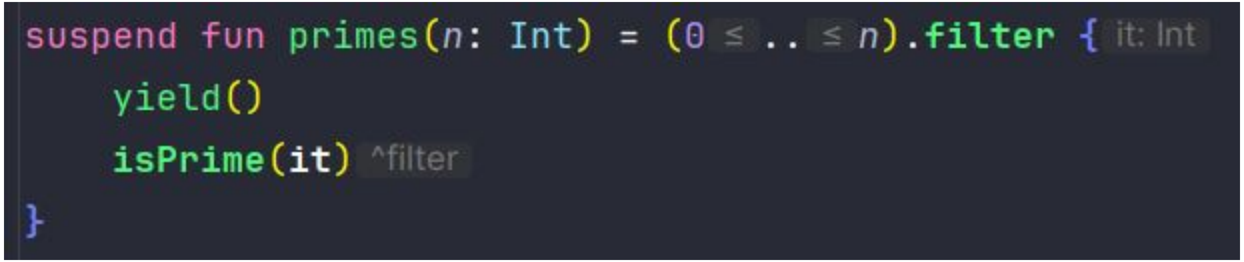


```
val pizzaOven = Semaphore( permits: 3)
val jobs = List( size: 10) { pizzaId ->
    launch { this: CoroutineScope
        pizzaOven.withPermit {
            println("Pizza $pizzaId start")
            delay( timeMillis: 1000)
            println("Pizza $pizzaId done")
        }
    }
}
jobs.forEach { it.join() }
```

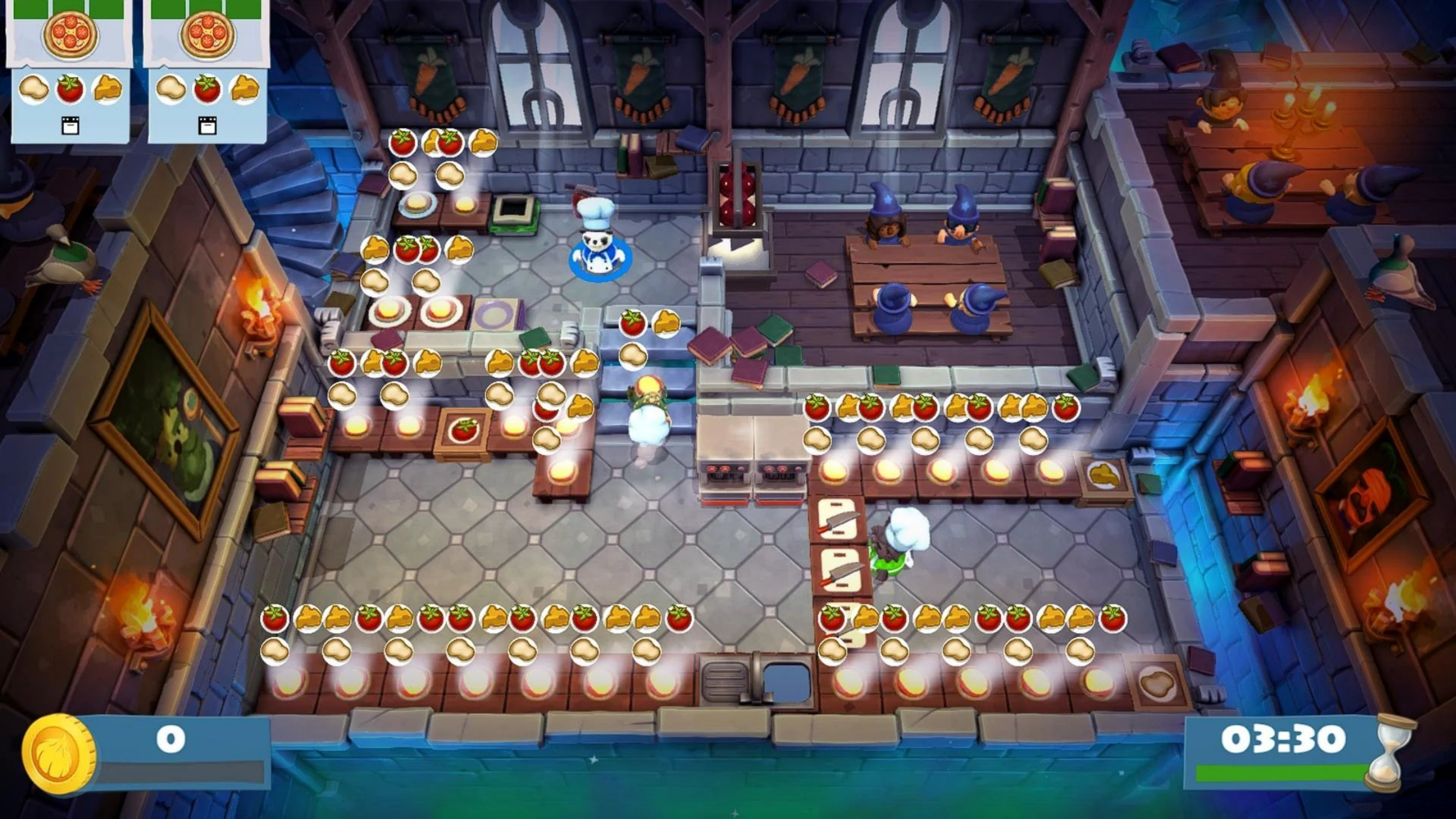


```
Pizza 1 start
Pizza 2 start
Pizza 0 start
Pizza 2 done
Pizza 0 done
Pizza 1 done
Pizza 5 start
Pizza 9 start
Pizza 7 start
Pizza 9 done
Pizza 7 done
Pizza 5 done
```

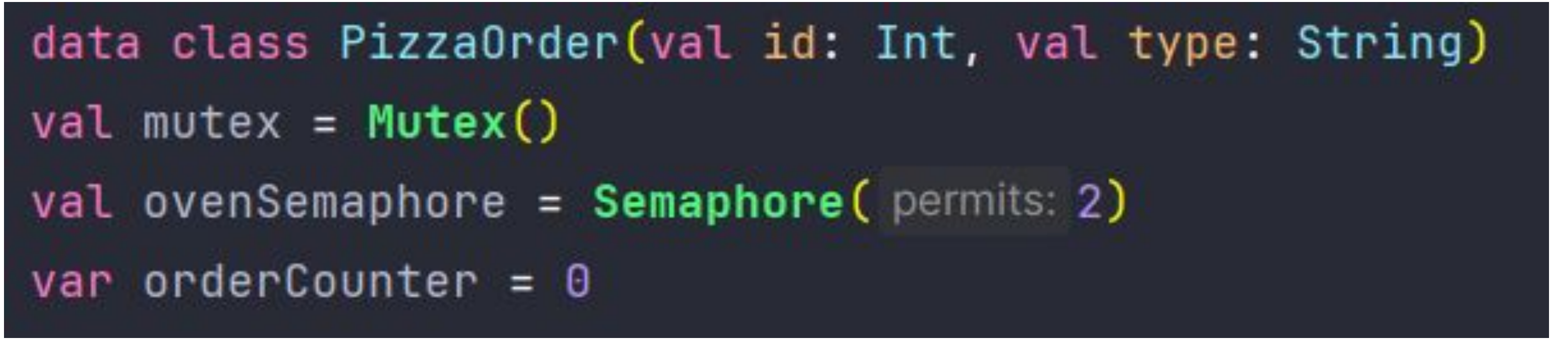
suspendable and yield



```
suspend fun primes(n: Int) = (0 ≤ .. ≤ n).filter { it: Int
    yield()
    isPrime(it) ^filter
}
```

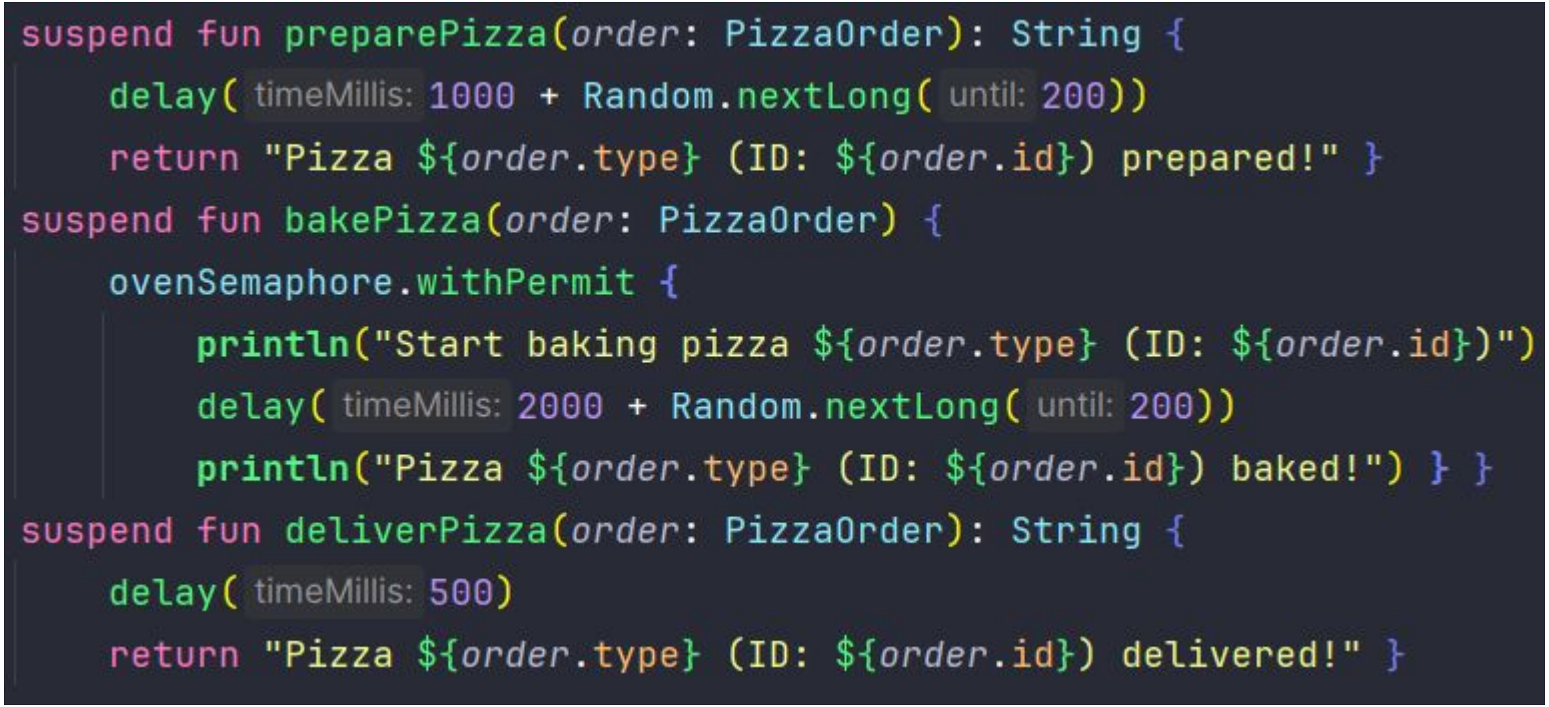

0
03:30

pizzeria



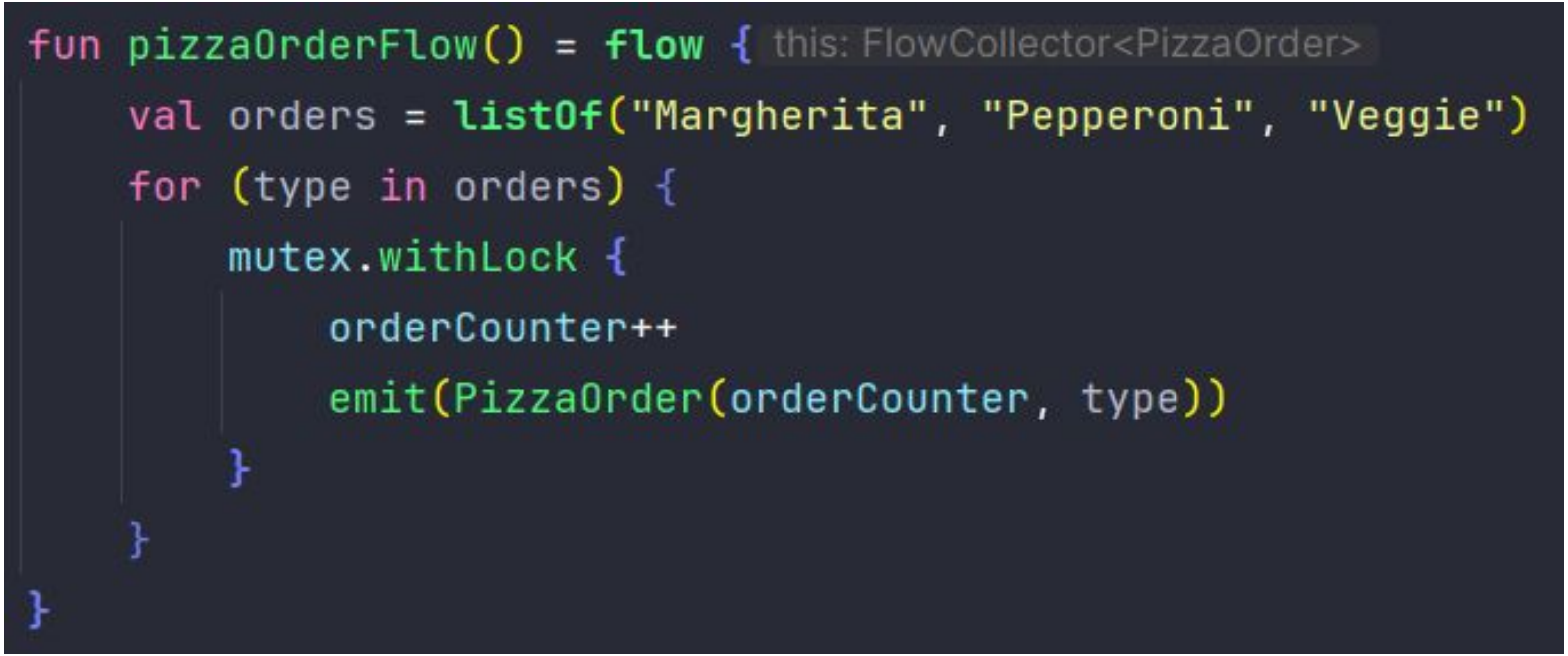
```
data class PizzaOrder(val id: Int, val type: String)
val mutex = Mutex()
val ovenSemaphore = Semaphore( permits: 2)
var orderCounter = 0
```

pizzeria



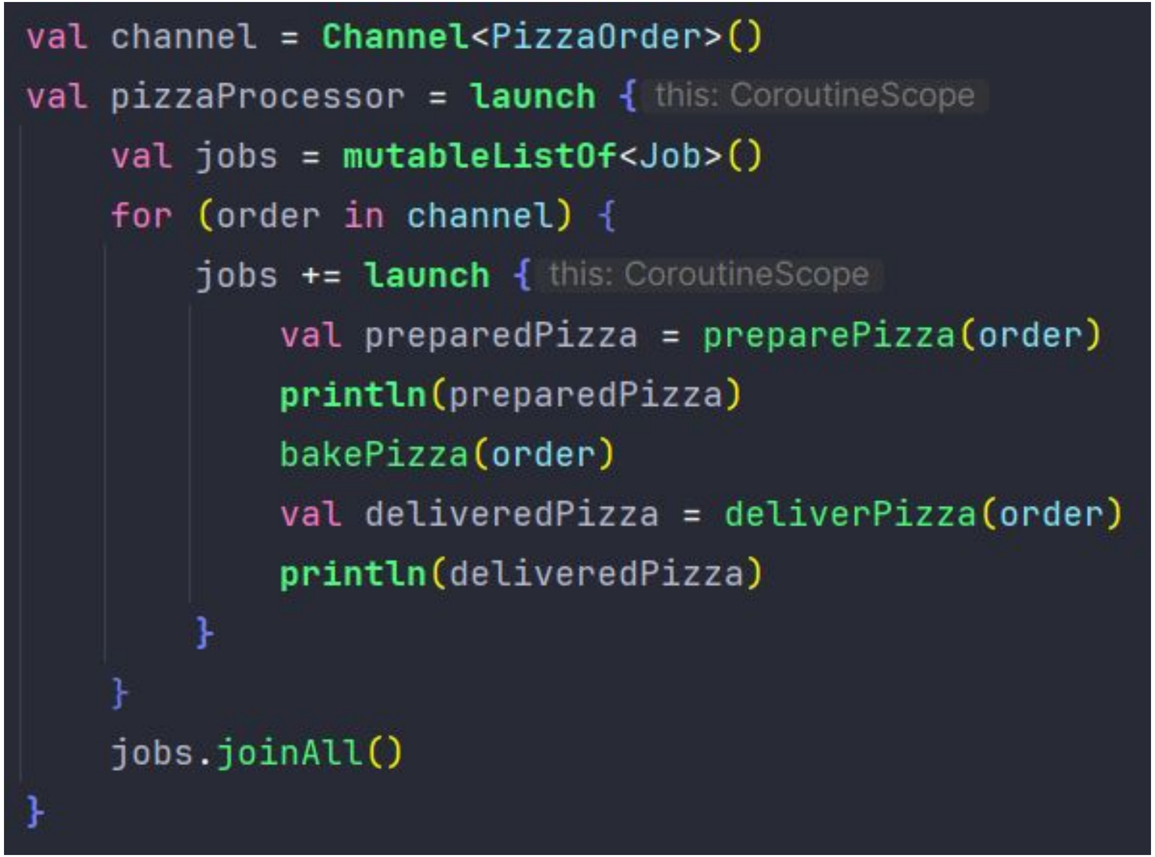
```
suspend fun preparePizza(order: PizzaOrder): String {
    delay( timeMillis: 1000 + Random.nextLong( until: 200))
    return "Pizza ${order.type} (ID: ${order.id}) prepared!" }
suspend fun bakePizza(order: PizzaOrder) {
    ovenSemaphore.withPermit {
        println("Start baking pizza ${order.type} (ID: ${order.id})")
        delay( timeMillis: 2000 + Random.nextLong( until: 200))
        println("Pizza ${order.type} (ID: ${order.id}) baked!") } }
suspend fun deliverPizza(order: PizzaOrder): String {
    delay( timeMillis: 500)
    return "Pizza ${order.type} (ID: ${order.id}) delivered!" }
```

pizzeria



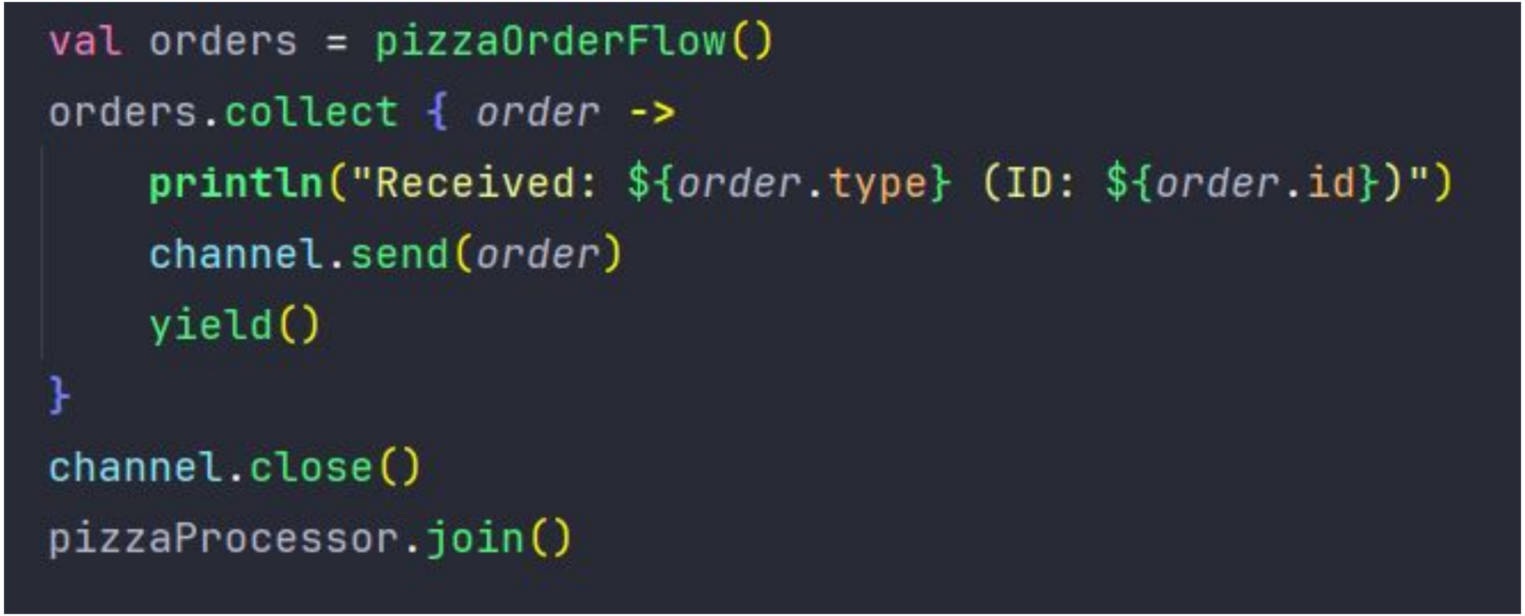
```
fun pizzaOrderFlow() = flow { this: FlowCollector<PizzaOrder>
    val orders = listOf("Margherita", "Pepperoni", "Veggie")
    for (type in orders) {
        mutex.withLock {
            orderCounter++
            emit(PizzaOrder(orderCounter, type))
        }
    }
}
```

pizzeria



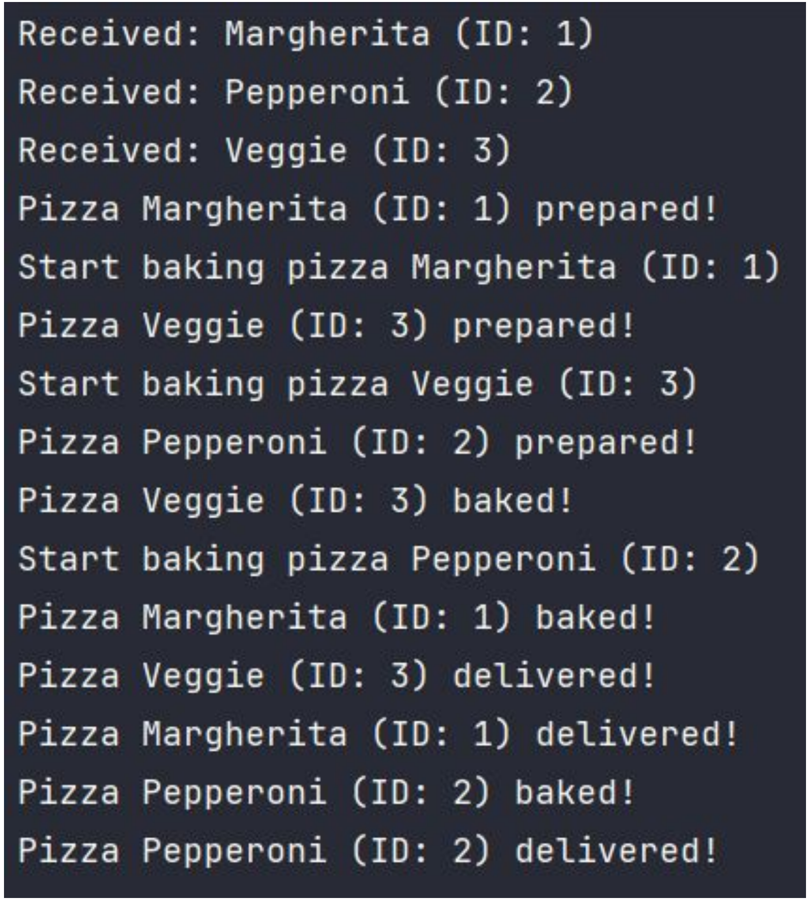
```
val channel = Channel<PizzaOrder>()
val pizzaProcessor = launch { this: CoroutineScope
    val jobs = mutableListOf<Job>()
    for (order in channel) {
        jobs += launch { this: CoroutineScope
            val preparedPizza = preparePizza(order)
            println(preparedPizza)
            bakePizza(order)
            val deliveredPizza = deliverPizza(order)
            println(deliveredPizza)
        }
    }
    jobs.joinAll()
}
```

pizzeria



```kotlin
val orders = pizzaOrderFlow()
orders.collect { order ->
    println("Received: ${order.type} (ID: ${order.id})")
    channel.send(order)
    yield()
}
channel.close()
pizzaProcessor.join()
```

pizzeria



```
Received: Margherita (ID: 1)
Received: Pepperoni (ID: 2)
Received: Veggie (ID: 3)
Pizza Margherita (ID: 1) prepared!
Start baking pizza Margherita (ID: 1)
Pizza Veggie (ID: 3) prepared!
Start baking pizza Veggie (ID: 3)
Pizza Pepperoni (ID: 2) prepared!
Pizza Veggie (ID: 3) baked!
Start baking pizza Pepperoni (ID: 2)
Pizza Margherita (ID: 1) baked!
Pizza Veggie (ID: 3) delivered!
Pizza Margherita (ID: 1) delivered!
Pizza Pepperoni (ID: 2) baked!
Pizza Pepperoni (ID: 2) delivered!
```

That's all Folks!

GitHub/LinkedIn

Vaga iFood

GitHub/LinkedIn

Vaga iFood