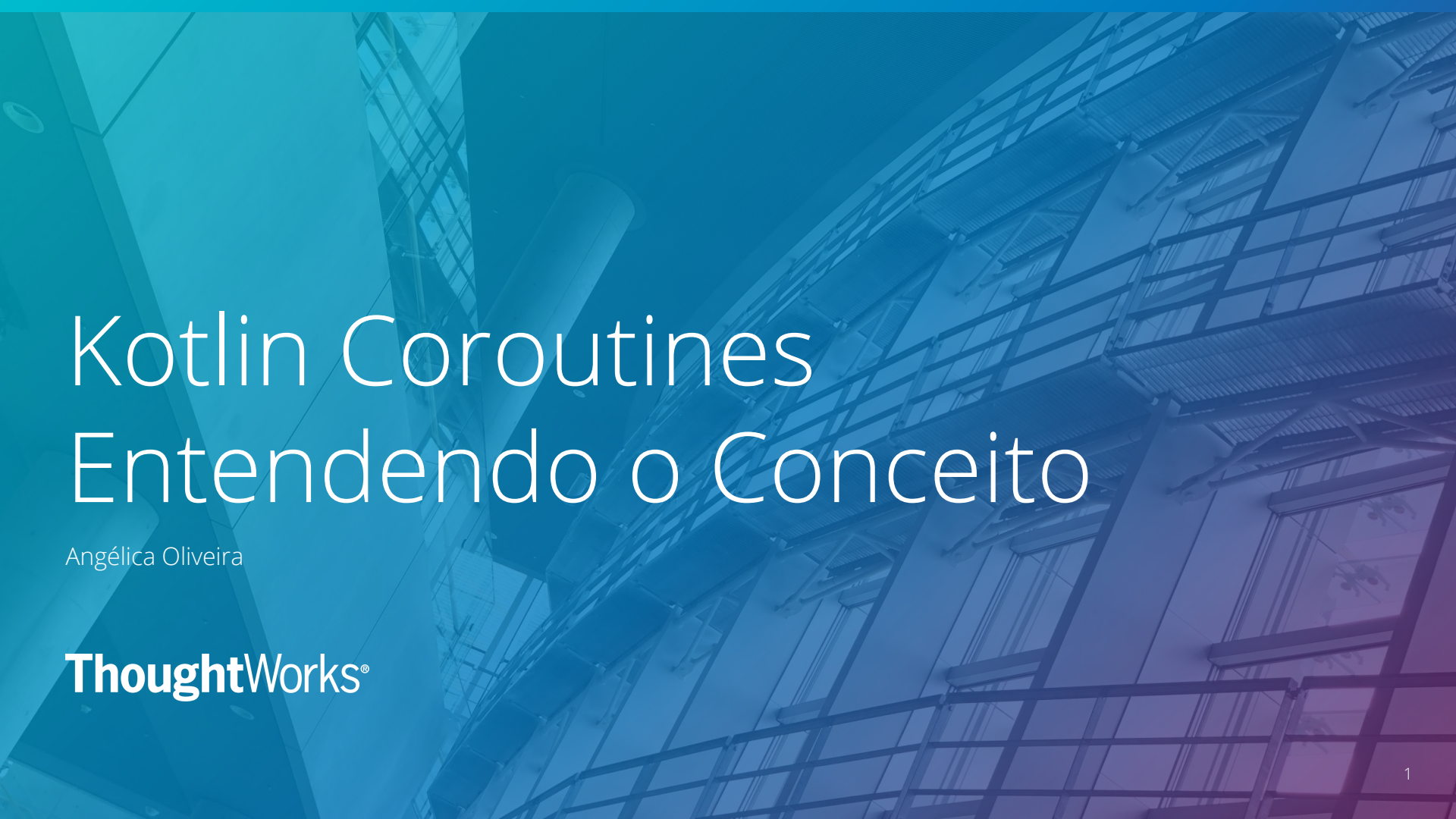


Kotlin Coroutines Entendendo o Conceito

Angélica Oliveira

ThoughtWorks®

ANGÉLICA OLIVEIRA

MOBILE DEVELOPER @THOUGHTWORKS

LinkedIn: angelica-oliv

Sobre o que vamos falar?

- Por que Coroutines?
- Coroutine Builders
- Suspending Functions
- Coroutine Context e Dispatchers
- Testes com Coroutines

Por que Coroutines?

COROUTINE BUILDERS

Coroutines Builders

runBlocking

```
fun main() {
    println("Hello,")

    // block the main thread
    runBlocking {
        // now we are inside a coroutine
        delay(2000L) // suspends for 2 seconds
    }

    // will be executed after 2 seconds
    println("Coroutines!")
}
```

Coroutines Builders

launch

```
fun main() {
    GlobalScope.launch { // launch new coroutine in background and continue
        delay(1000L)
        println("Coroutines!")
    }

    println("Hello,") // main thread continues here immediately

    runBlocking {
        delay(2000L)  // ... 2 seconds to keep JVM alive
    }
}
```

Coroutines Builders

async

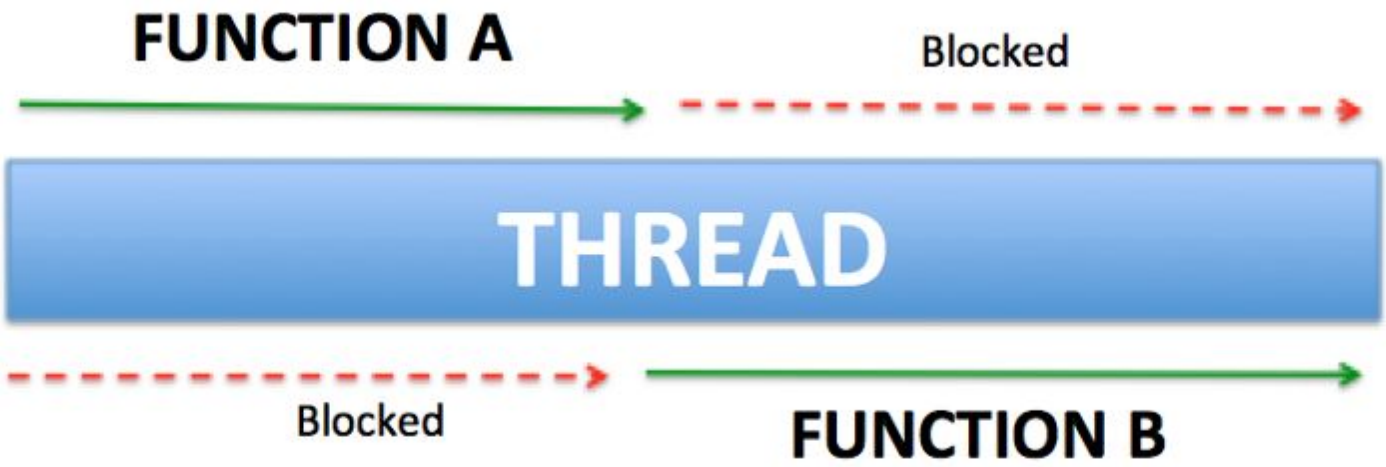
```
fun main() {
    val deferredResult: Deferred<String> = GlobalScope.async {
        // suspend for 1 second
        delay(1000L)
        // this is the result
        "Coroutine!"
    }

    runBlocking {
        println("Hello")
        // wait for the result
        println(deferredResult.await())
    }
}
```

SUSPENDING FUNCTIONS

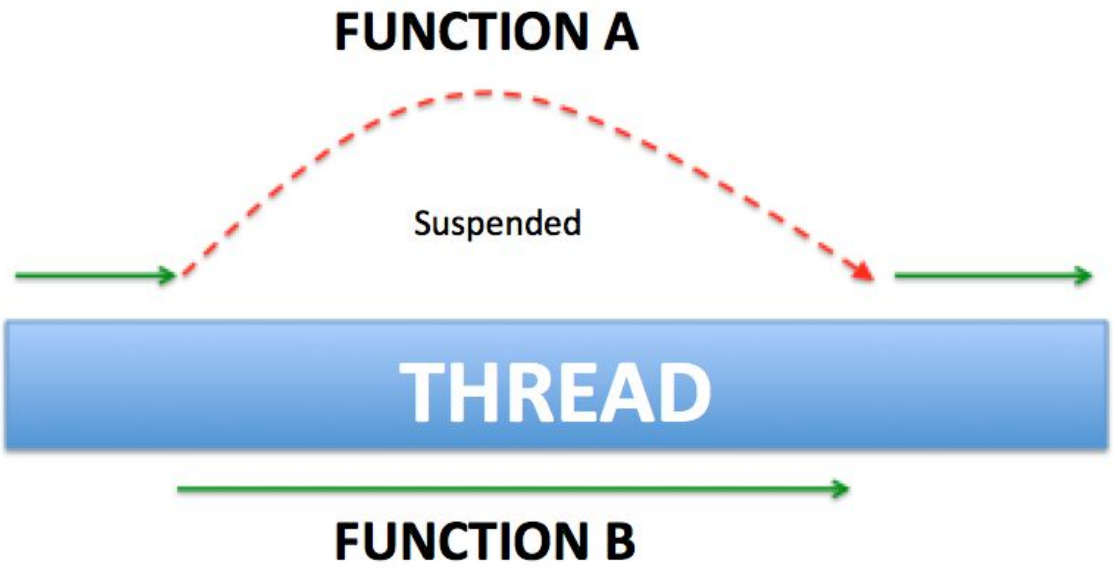
Suspending Functions

Blocking x Suspending



Suspending Functions

Blocking x Suspending



Suspending Functions

Por default coroutines são sequenciais

```
private suspend fun doSomethingUsefulOne(): Int {
    delay(1000L) // pretend we are doing something useful here
    return 13
}

private suspend fun doSomethingUsefulTwo(): Int {
    delay(1000L) // pretend we are doing something useful here, too
    return 29
}
```

Suspending Functions

Por default coroutines são sequenciais

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        val one = doSomethingUsefulOne()
        val two = doSomethingUsefulTwo()
        println("The answer is ${one + two}")
    }
    println("Completed in $time ms")
}
```

```
The answer is 42
Completed in 2014 ms
```

Suspending Functions

Por default coroutines são sequenciais

```
fun main() = runBlocking {
    val time = measureTimeMillis {
        val one = async { doSomethingUsefulOne() }
        val two = async { doSomethingUsefulTwo() }
        println("The answer is ${one.await() + two.await()}")
    }
    println("Completed in $time ms")
}
```

```
The answer is 42
Completed in 1020 ms
```

COROUTINE CONTEXT E DISPATCHERS

Coroutines Context

- Coroutines sempre executam em um determinado contexto definido pelo tipo CoroutineContext

- O contexto da Coroutine é um conjunto de vários elementos, os principais são:
  - Job
  - Dispatcher

Coroutines Context

Job

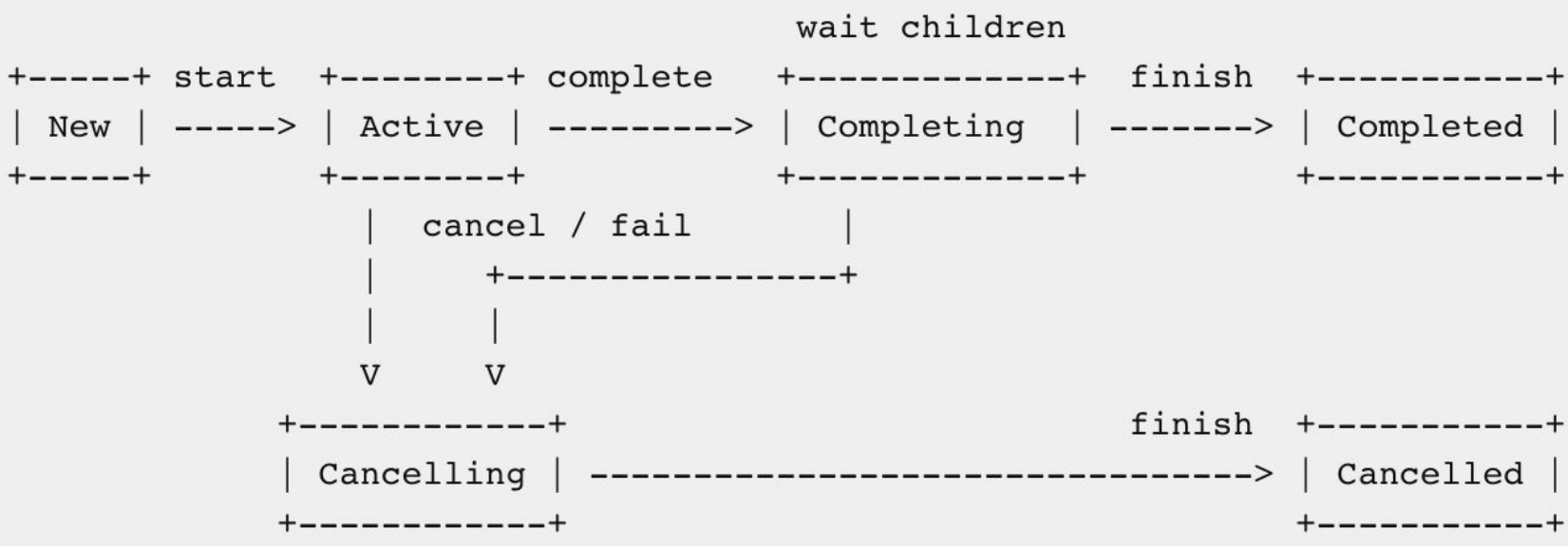
- Jobs são executáveis que possuem um ciclo de vida

- Podem possuir uma hierarquia parent-child

- Os estados de um Job são definidos pelas flags:
  - isActive
  - isCompleted
  - isCancelled

Coroutines Context

Job



Coroutines Context

Dispatcher

- Responsável por determinar que Thread ou Threads a Coroutine usa para sua execução

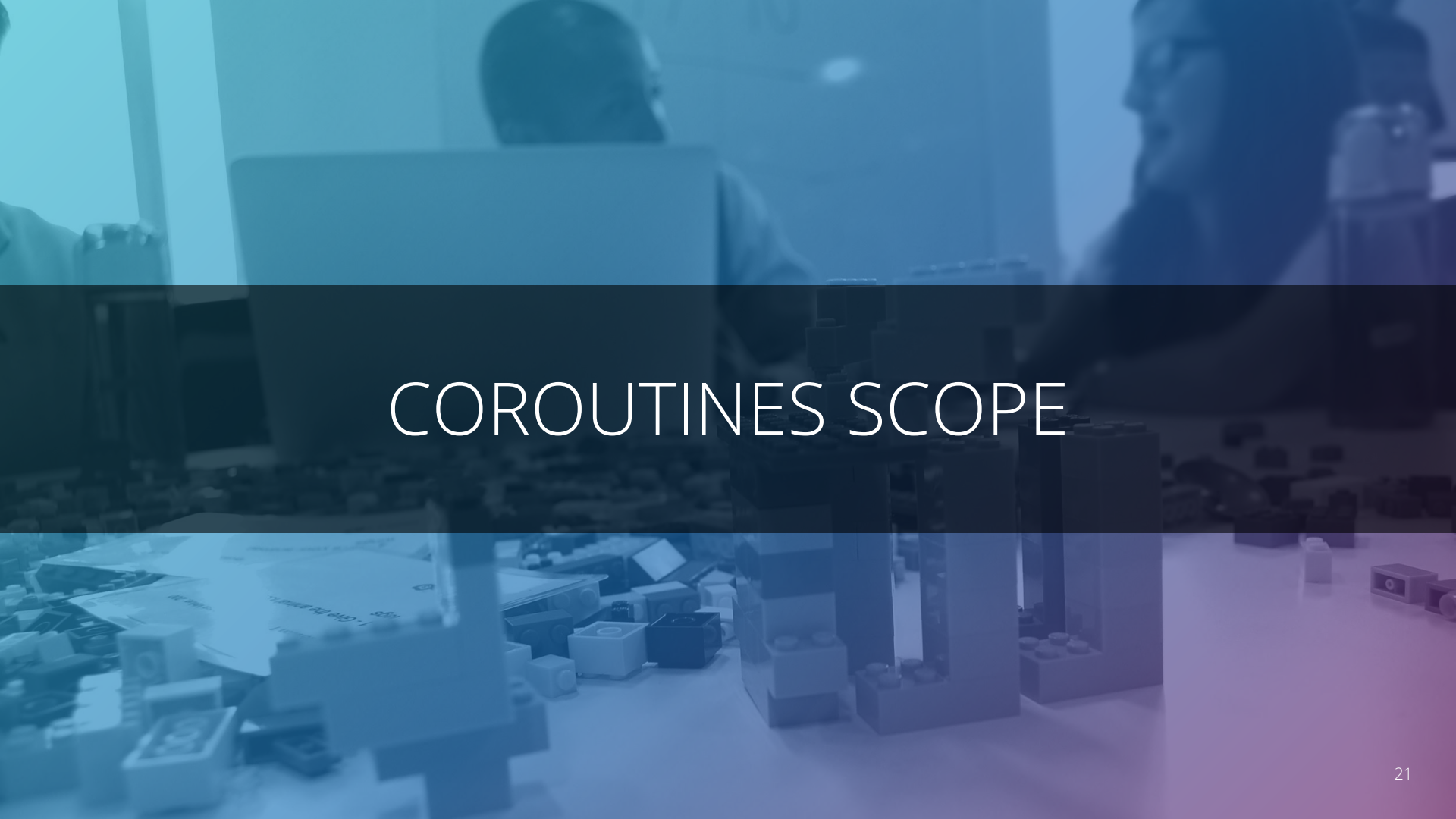
- Todos os builders de Coroutine aceitam um CoroutineContext como parâmetro, para explicitamente especificar o Dispatcher e outros elementos do contexto

Coroutines Context

Dispatcher

```
launch { ... }
launch(Dispatchers.Unconfined) { ... }
launch(Dispatchers.Default) { ... }
launch(newSingleThreadContext("MyOwnThread")) { ... }
```

```
Unconfined            : I'm working in thread main
Default               : I'm working in thread DefaultDispatcher-worker-2
newSingleThreadContext: I'm working in thread MyOwnThread
main runBlocking      : I'm working in thread main
```



COROUTINES SCOPE

Coroutines Scope

Global Scope

```
fun main() = runBlocking {
    GlobalScope.launch {
        repeat(1000) { i ->
            println("I'm sleeping $i ...")
            delay(500L)
        }
    }
    delay(1300L) // just quit after delay
}
```

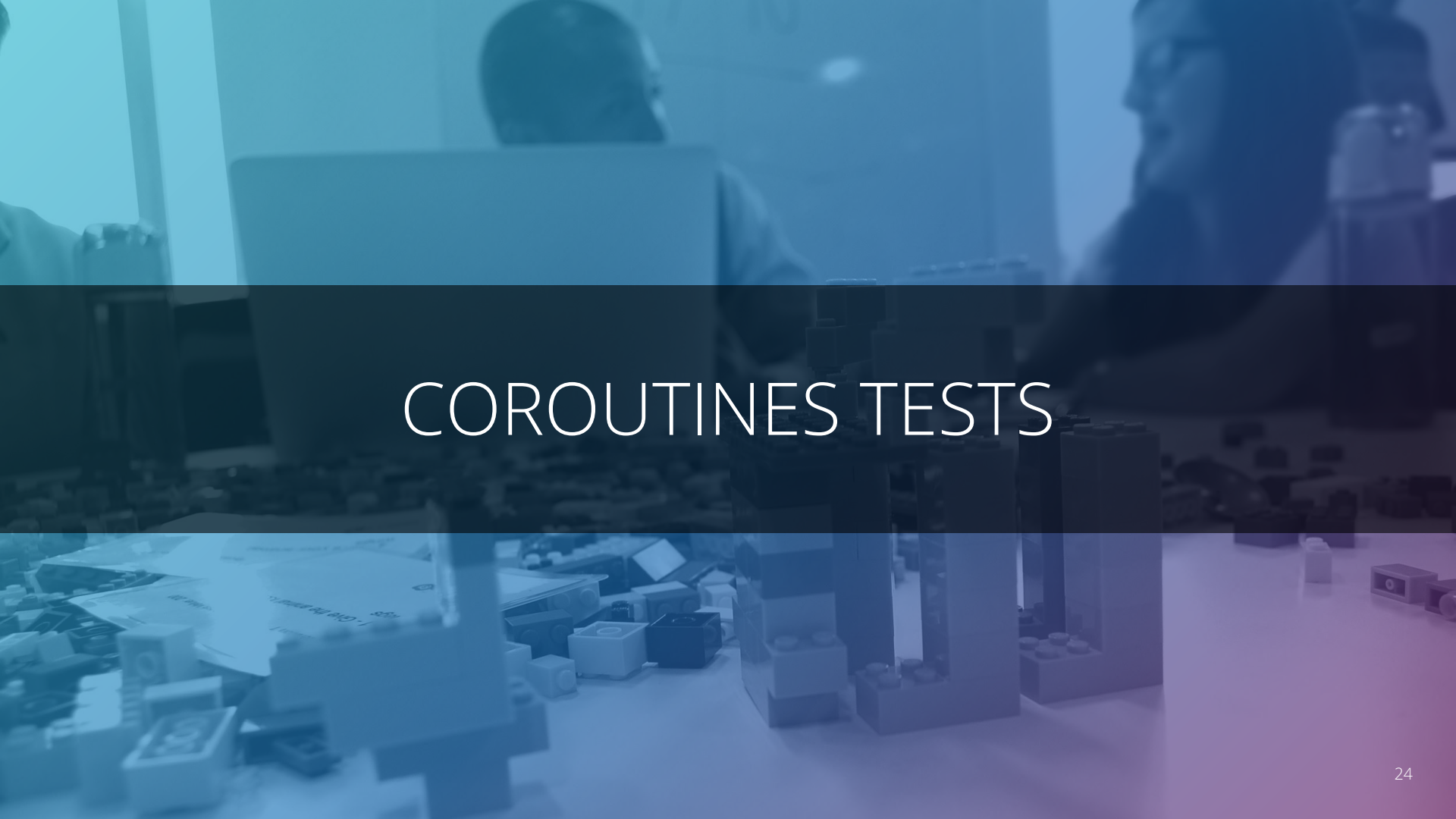
```
I'm sleeping 0 ...
I'm sleeping 1 ...
I'm sleeping 2 ...
```

Coroutines Scope

Scope builder

```
fun main() = runBlocking { // this: CoroutineScope
    coroutineScope { // Creates a coroutine scope
        launch {
            delay(500L)
            println("Task from nested launch")
        }

        delay(100L)
        println("Task from coroutine scope")
    }
}
```



COROUTINES TESTS

Coroutines Tests

Dispatcher.setMain

- Chamada que sobrescreve o Main Dispatcher em situações de testes

- Pode ser utilizado quando:
  - Main dispatcher não está disponível
  - É necessário sobrescrever por questões de sincronia dos testes (com uma implementação de um Dispatcher na classe de testes)

Coroutines Tests

Dispatcher.setMain

```
// overrides Main Dispatcher with a test implementation
Dispatchers.setMain(dispatcherImplementation)


// reset Main Dispatcher with original implementation
Dispatchers.resetMain()
```

Coroutines Tests

runBlockingTest

Provê controle extra para testar as coroutines:

- Avanço do tempo automático para suspend functions
- Avanço do tempo manual para testar coroutines múltiplas
- Execução pausada de chamadas *async* e *launch*
- Pausa, avanço ou recomeço das coroutines em um teste
- Reportar exceções como falhas de testes

Coroutines Tests

TestCoroutineScope e TestCoroutineDispatcher

- *TestCoroutineScope* provê controle detalhado da execução das coroutines nos testes

- *TestCoroutineDispatcher* é o dispatcher utilizado quando se executa uma coroutine com *TestCoroutineScope*

- Ambos se integram com *runBlockingTest*

CÓDIGO UTILIZADO NA PALESTRA ;)

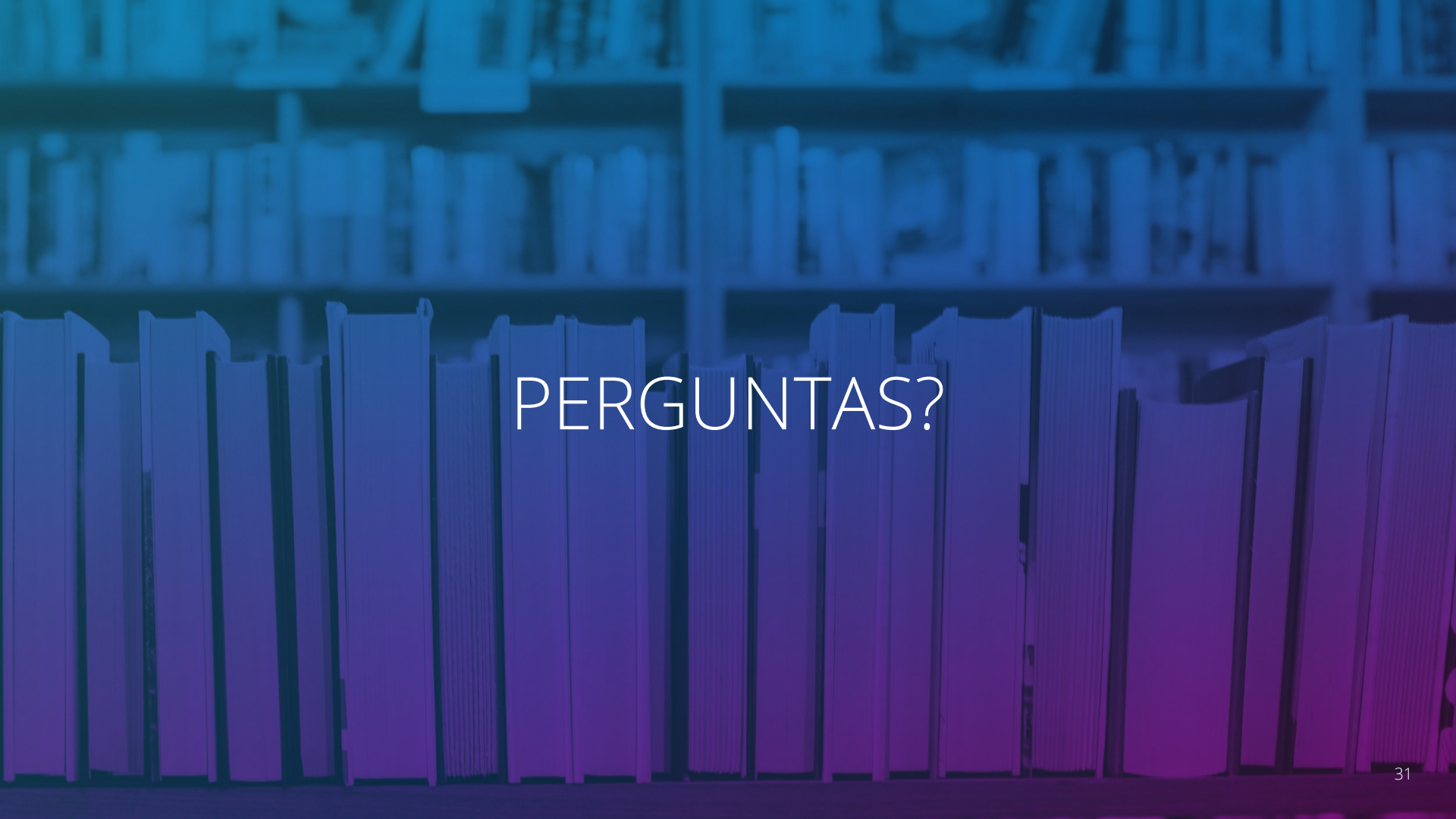
https://github.com/angelica-oliv/coroutines-basics

Quer se tornar uma ThoughtWorker?

http://bit.ly/tw-tdcsp-vagas

ThoughtWorks® /careers

Crie sua história


PERGUNTAS?

OBRIGADA

ANGÉLICA OLIVEIRA
SENIOR CONSULTANT DEVELOPER
angelica.deoliveira@thoughtworks.com |
thoughtworks.com

ThoughtWorks®