Communication
Break Down

Bob Dahlberg
Mobile Lead Developer

QVIK

Coroutines

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

```
fun main() = runBlocking<Unit> {
    repeat(100_000) {
        launch { // creates a coroutine
            println(“On thread -> $thread”)
        }
    }
}
```

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

```
repeat(100_000) {
    launch { // creates a coroutine
        println(“On thread -> $thread”)
    }
}
```

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

```
repeat(100_000) {
    thread { // creates a thread
        println(“On thread -> $thread”)
    }
}
```

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

```
repeat(100_000) {
    launch(Dispatchers.Default) { // 8 threads
        println(“On thread -> $thread”)
    }
}
```

Coroutines

Lightweight threads

“Think of them as lightweight threads”

What do we mean by lightweight?

Should we treat them as threads?

Because they ***might*** be.

```
repeat(100_000) {
    launch(Dispatchers.IO) { // 84 threads
        println(“On thread -> $thread”)
    }
}
```

Coroutines

Lightweight threads

How about thread safety?

Coroutines

Lightweight threads

How about thread safety?

```
var i = 0
repeat(100_000) {
    launch(Dispatchers.Default) {
        i += it
    }
}
println("Result → $i”)
```

Coroutines

Lightweight threads

How about thread safety?

```
var i = 0
repeat(100_000) {
    launch(Dispatchers.IO) {
        i += it
    }
}
println("Result → $i”)
```

Coroutines

Lightweight threads

How about thread safety?

```
var i = 0
repeat(100_000) {
    launch {
        i += it
    }
}
println("Result → $i”)
```

Coroutines

Lightweight threads

How about thread safety?

```
var i = 0
launch(Dispatchers.Default) {
    repeat(100_000) {
        launch {
            i += it
        }
    }
    println("Result → $i”)
}
```

Coroutines

Lightweight threads

How about thread safety?

```
@Volatile var i = 0
launch(Dispatchers.Default) {
    repeat(100_000) {
        launch {
            i += it
        }
    }
    println("Result -> $i”)
}
```

Coroutines

Treat them as threads?

So treat them as threads and we are fine?

Coroutines

Treat them as threads?

```
A1. On thread main
A2. On thread worker-1
```

So treat them as threads and we are fine?

```
launch(Dispatchers.Unconfined) {
    println(“A1. On thread $thread”)
    delay(200)
    println(“A2. On thread $thread”)
}
```

Coroutines

Treat them as threads?

```
A1. On thread worker-1
B1. Switching worker-3
A2. On thread worker-3
```

```
fun main() = runBlocking<Unit>{
    launch(Dispatchers.IO) {
        println(“A1. On thread $thread”)
        switchContext()
        println(“A2. On thread $thread”)
    }
}

suspend fun switchContext() {
    withContext(Dispatchers.Default) {
        println(“B1. Switching $thread”)
    }
}
```

Coroutines

Treat them as threads?

```
A1. IO
B1. null
A2. Default
```

```
val local = ThreadLocal<String?>()
fun main() = runBlocking<Unit>{
    launch(Dispatchers.IO) {
        local.set("IO")
        println(“A1. ${local.get()}”)
        switchContext()
        println(“A2. ${local.get()}”)
    }
}
suspend fun switchContext() {
    withContext(Dispatchers.Default) {
        println(“B1. ${local.get()}”)
        local.set("Default")
    }
}
```

Coroutines

Treat them as threads?

So treat them as threads and we are fine?

Coroutines

Treat them as coroutines!

So treat them as threads and we are fine?

Nope, treat them as coroutines!

Coroutines

Treat them as coroutines!

```
Starting!
Ending!
Starting!
Ending!
```

Excellent example from Dan Lew (blog.danlew.net)

```
@Synchronized
fun criticalSection() {
    println(“Starting!")
    Thread.sleep(10)
    println("Ending!")
}

repeat(2) {
    thread { criticalSection() }
}
```

Coroutines

Treat them as coroutines!

```
Starting!
Starting!
Ending!
Ending!
```

Excellent example from Dan Lew (blog.danlew.net)

```
@Synchronized
suspend fun criticalSection() {
    println("Starting!")
    delay(10)
    println("Ending!")
}

repeat(2) {
    launch(Dispatchers.Default) {
        criticalSection()
    }
}
```

Coroutines

Treat them as coroutines!

Excellent example from Dan Lew (blog.danlew.net)

```
@Synchronized
fun criticalSection() {
    println("Starting!")
    Thread.sleep(10)
    println("Ending!")
}

repeat(2) {
    launch(Dispatchers.Default) {
        criticalSection()
    }
}
```

Let’s communicate coroutine-style

Communication

Deferred

Deferred is a non-blocking cancelable future.

Communication

Deferred

```
Val result: Deferred<Response> = async {
    fetchChannels()
}

println(“Deferred → ${result.await()}”)
```

Communication

Deferred

```
val result = async { delay(2000) }
val result2 = async { delay(1000) }

println(“Deferred → ${result.await()}”)
println(“Deferred → ${result2.await()}”)
```

Communication

Deferred

```
Deferred → Temp(name=“Bob”)
or
Deferred → Temp(name=“Charlie”)
```

```
data class Temp(var name: String)

val result = CompletableDeferred<Temp>()
launch(Dispatchers.Default) {
    val temp = Temp("Bob")
    result.complete(temp)
    temp.name = "Charlie"
}

val temp = result.await()
println("Deferred → $temp")
```

Communication

Deferred

```
Deferred → Temp(name=“Bob”)
```

```kotlin
data class Temp(val name: String)

val result = CompletableDeferred<Temp>()
launch(Dispatchers.Default) {
    result.complete(Temp("Bob"))
    delay(1)
    result.complete(Temp("Charlie"))
}


val temp = result.await()
println("Deferred → $temp”)
```

Communication

Deferred

```
data class Temp(var name: String)

val result = CompletableDeferred<Temp>()
launch(Dispatchers.Default) {
    result.complete(Temp("Bob"))
    delay(1)
    result.complete(Temp("Charlie"))
}

launch(Dispatchers.IO) {
    val temp = result.await()
    println("Deferred → $temp”)
}
```

Communication

Channels

Channels provide a way to transfer a stream of values.

Communication

Channels

```
val channel = Channel<String>()
launch(Dispatchers.Default) {
    channel.send("Bob")
    channel.send("Charlie")
}

println(“Get → ${channel.receive()}”)
println(“Get → ${channel.receive()}”)
```

Communication

Channels

```
val channel = Channel<String>()
launch(Dispatchers.Default) {
    channel.send("Bob")
    println(“Get -> ${channel.receive()}”)
}
```

Communication

Channels

```
val channel = Channel<String>(1)
launch(Dispatchers.Default) {
    channel.send("Bob")
    println(“Get -> ${channel.receive()}”)
}
```

Communication

Channels

```
Channel<String>(7) // BUFFERED
Channel<String>(Channel.UNLIMITED)
Channel<String>(Channel.CONFLATED)
Channel<String>(Channel.RENDEZVOUS)
```

Communication

Channels

```
val channel = Channel<String>()
launch(Dispatchers.Default) {
    channel.send("Bob")
    channel.send("Charlie")
}

println(“Get -> ${channel.toList()}”)
```

Communication

Channels

```
val channel = Channel<String>()
launch(Dispatchers.Default) {
    channel.send("Bob")
    channel.send(“Charlie”)
    channel.close()
}

println(“Get -> ${channel.toList()}”)
```

Communication

Channels

Channels are synchronization primitives

Let’s see where they excel

Communication

Channels

```
Name: Charlie | Name: Bob
Name: Bob     | Name: Charlie
```

```
suspend fun race(name:String): String {
    delay(nextLong(5000))
    return name
}

val ch = Channel<String>()
launch(…) { ch.send(race("Bob")) }
launch(…) { ch.send(race(“Charlie")) }

launch(Dispatchers.Default) {
    repeat(2) {
        println(“Name: ${ch.receive()}")
    }
    ch.close()
}
```

Communication

Channels

```
0 → 0
2 → 1
1 → 2
......
```

```
val ch = Channel<Int>()
launch(Dispatchers.Default) {
    repeat(30) { ch.send(it) }
    ch.close()
}

repeat(3) { id →
    launch(Dispatchers.Default) {
        for(msg in ch) {
            println(“$id → $msg”)
        }
    }
}
```

Communication

Channels

```
0 -> 0
2 -> 1
1 -> 2
......
```

```kotlin
val ch = produce {
    repeat(30) { send(it) }
}

repeat(3) { id ->
    launch(Dispatchers.Default) {
        ch.consumeEach {
            println("$id -> $it")
        }
    }
}
```

Communication

Mutex

Mutex - Kotlins mutual exclusion

Communication

Mutex

```
var i = 0
repeat(100_000) {
    launch(Dispatchers.Default) {
        i += it
    }
}
println("Result -> $i”)
```

Communication

Mutex

```
val mutex = Mutex()
var i = 0
repeat(100_000) {
    launch(Dispatchers.Default) {
    mutex.withLock {
        i += it
    }
}
println("Result → $i”)
```

Communication

Mutex

```
val mutex = Mutex()
suspend fun criticalSection() {
    mutex.withLock {
        println("Starting!")
        delay(10)
        println("Ending!")
    }
}

repeat(2) {
    launch { criticalSection() }
}
```

Communication

Flow

Flow - reactive streams contender

Communication

Flow

```
Value → 1
Value → 2
…
Value → 10
```

```
val example: Flow<Int> = flow {
    for(i in 1..10) {
        emit(i)
    }
}

example.collect {
    println("Value → $it")
}
```

Communication

Flow

```
Value -> 10
Value -> 12
…
Value -> 20
```

```kotlin
val example = flow {
    for(i in 1..10) {
        emit(i)
    }
}

example.filter { it >= 5 }
    .map { it * 2 }
    .collect {
        println("Value -> $it")
    }
```

Communication

Flow

```
Value -> 10
Value -> 12
…
Value -> 20
```

```kotlin
val example = (1..10).asFlow()

example.filter { it >= 5 }
    .map { it * 2 }
    .collect {
        println("Value -> $it")
    }
```

Communication

Flow

```
Flow on → main
Collect on → main
```

```
val example = flow {
    for(i in 1..10) {
        println(“Flow on → ${thread()}”)
        emit(i)
    }
}

example.filter { it ⩾ 5 }
    .map { it * 2 }
    .collect {
        println(“Collect on → ${thread()}”)
    }
```

Communication

Flow

```
Flow on → worker-1
Collect on → main
```

```
val example = flow {
    for(i in 1..10) {
        println(“Flow on → ${thread()}”)
        emit(i)
    }
}.flowOn(Dispatchers.Default)

example.filter { it ≥ 5 }
    .map { it * 2 }
    .collect {
        println(“Collect on → ${thread()}”)
    }
```

Communication

Flow

```
Flow on → worker-1
Map on → main
Collect on → main
```

```
val example = flow {
    for(i in 1..10) {
        println(“Flow on → ${thread()}”)
        emit(i)
    }
}.flowOn(Dispatchers.Default)

example.filter { it ⩾ 5 }
    .map {
        println(“Map on → ${thread()}”)
        it * 2
    }
    .collect {
        println(“Collect on → ${thread()}”)
    }
```

Communication

Flow

```
Flow on -> worker-1
Map on -> worker-1
Collect on -> main
```

```
val example = flow {
    for(i in 1..10) {
        println(“Flow on -> ${thread()}”)
        emit(i)
    }
}

example.filter { it >= 5 }
    .map {
        println(“Map on -> ${thread()}”)
        it * 2
    }
    .flowOn(Dispatchers.Default)
    .collect {
        println(“Collect on -> ${thread()}”)
    }
```

Communication

Flow

```
val example = flow {
    for(i in 1..10) {
        println(“Flow on → ${thread()}”)
        emit(i)
    }
}

example.filter { it ⩾ 5 }
    .flowOn(Dispatchers.IO)
    .map { it * 2 }
    .flowOn(Dispatchers.Default)
    .collect {
        println(“Collect on → ${thread()}”)
    }
```

Communication

Flow

```
Exception in thread "main"
java.lang.IllegalStateException:
Flow invariant is violated:
```

```
val example = flow {
    withContext(Dispatchers.Default) {
        for(i in 1..10) { emit(i) }
    }
}

example.filter { it >= 5 }
    .map { it * 2 }
    .collect {
        println(“Collect on -> ${thread()}”)
    }
```

Thank you!

The deck is available on: https://speakerdeck.com/bobdahlberg

Questions?

Bob Dahlberg
bob@qvik.com
medium.com/dahlbergbob
@mr_bob

QVIK