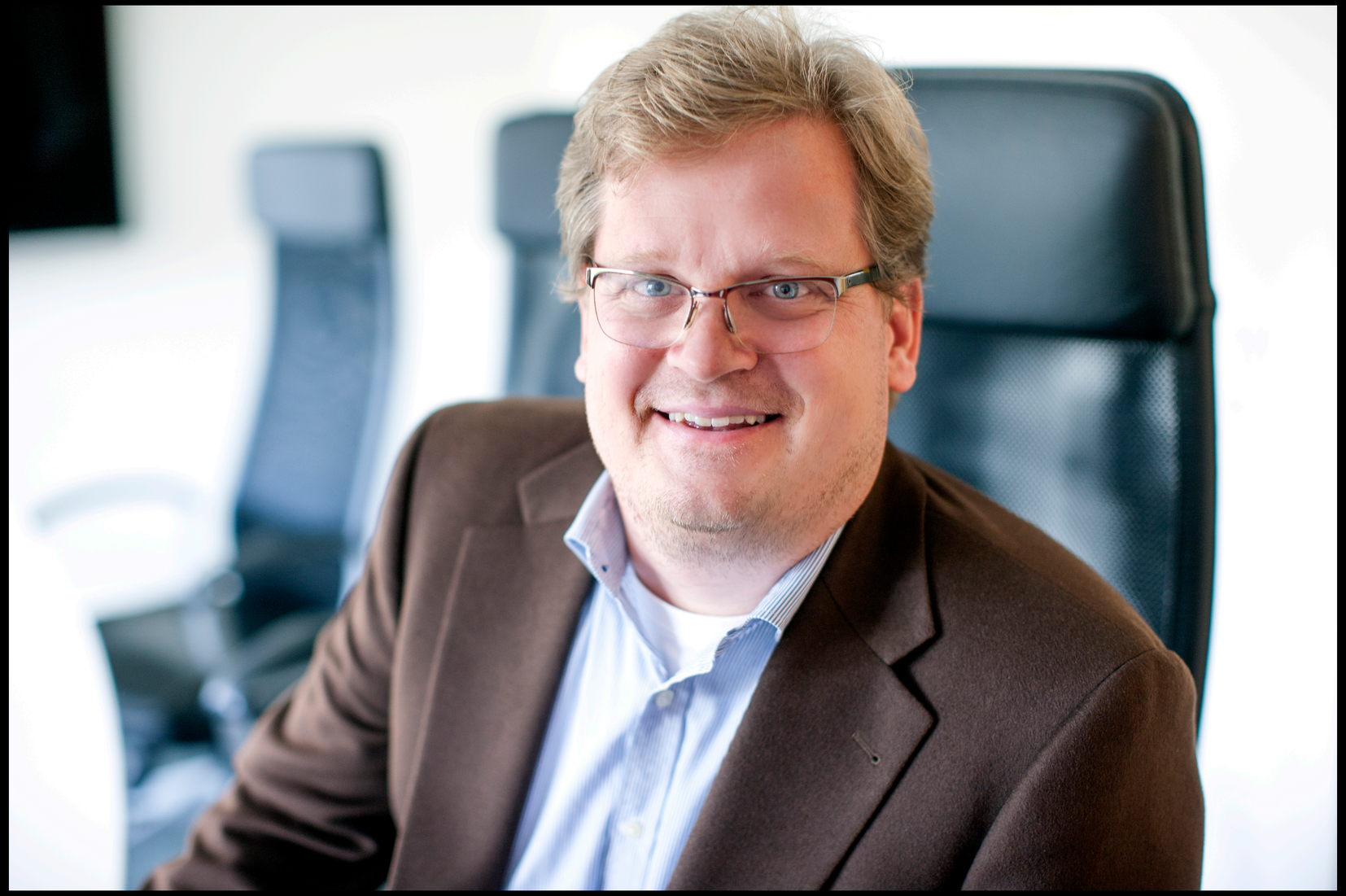


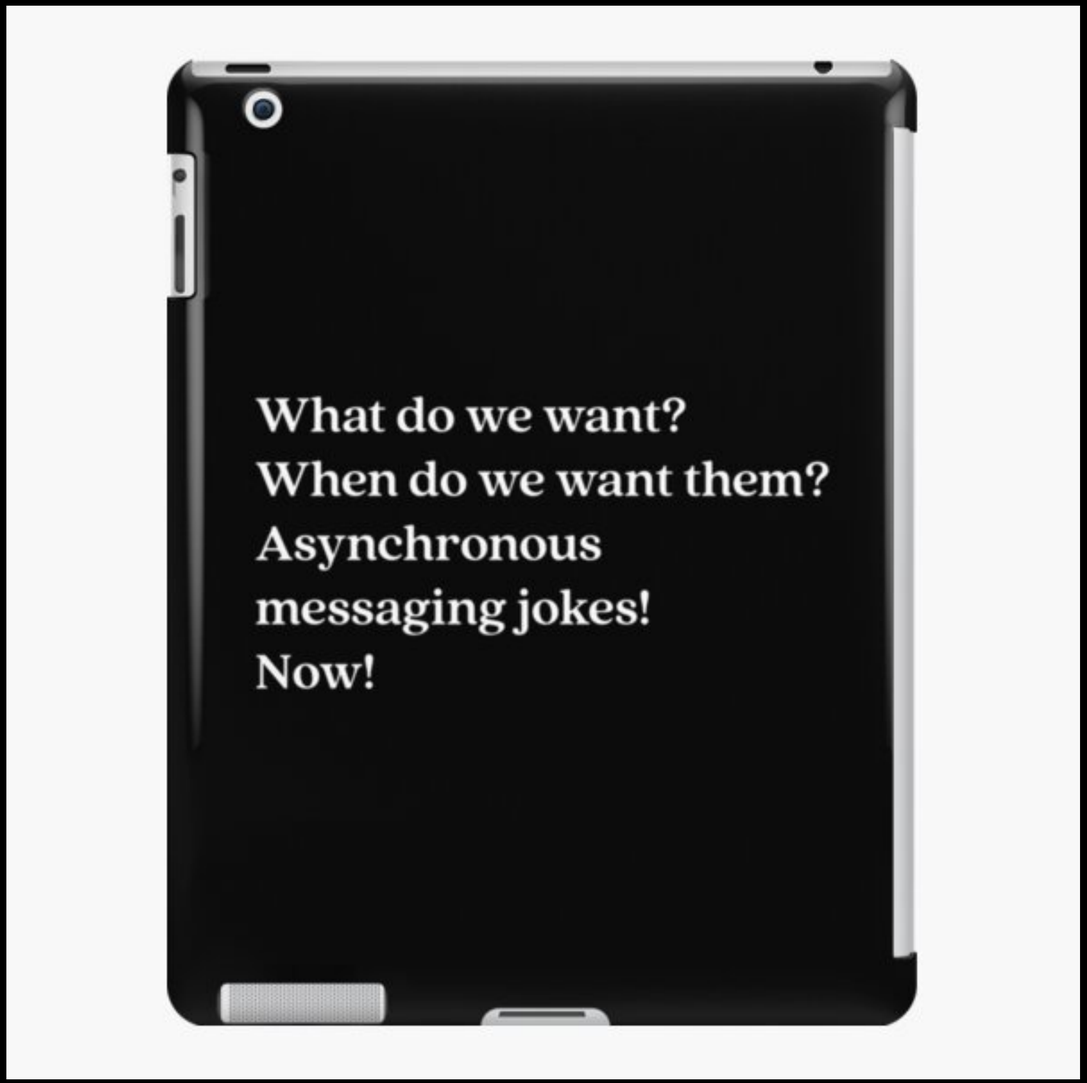
JESPER HOLMBERG

SOLUTION ARCHITECT & BACKEND SPECIALIST

CALLISTA

ASYNCHRONOUS PROGRAMMING

- Minimum number of threads: thread is relinquished as soon as a wait is encountered.
- Not necessarily faster, but more scalable.
- Problem: asynchronous programming is difficult.
- Kotlin coroutines were released in 2018.
- Similar to ‘async/await’.



FROM CALLBACKS TO COROUTINES

- Callbacks

```
fun requestId(arg: String,
              callback: (String) -> Unit) {
    // Create Id from 'arg'
    // Call 'callback' with Id
}

fun savePost(arg: String,
             callback: (String) -> Unit) {
    // Save 'arg' as new post
    // Call 'callback' with post
}

fun createArticle(arg: String) {
    requestId(arg) { id ->
        savePost(id) { result ->
            processResult(result)
        }
    }
}
```

- Future/Promise/Deferred

```
fun requestId(arg: String): Deferred<String> {
    // Create Id from 'arg'
    // Return Id in Deferred (future/promise)
}

fun savePost(arg: String): Deferred<String> {
    // Save 'arg' as new post
    // Return post in Deferred(future/promise)
}

fun createArticle(arg: String) {
    requestId(arg)
     .thenCompose { id -> savePostAsync(id) }
     .thenAccept { result ->
         processResult(result)
     }
}
```

- Coroutines

```
suspend fun requestId(arg: String): String {
    // Create Id from 'arg'
    // Return Id
}

suspend fun savePost(arg: String): String {
    // Save 'arg' as new post
    // Return post
}

fun createArticle(arg: String) {
    GlobalScope.launch {
        val id = requestId(arg)
        val result = savePost(id)
        processResult(result)
    }
}
```

REACTIVE STREAMS VS COROUTINES - 1

```
@RestController
class PostController() {
    @GetMapping("/{id}")
    fun findOne(id: Long?): Mono<Post> =
        repository.findById(id)

    @GetMapping
    fun findAll(): Flux<Post> =
        repository.findAll()
}
```

Spring does an implicit ‘subscribe’ on all reactive streams functions in the controller.

```
@RestController
class PostController() {
    @GetMapping("/{id}")
    suspend fun findOne(id: Long): Post? =
        repository.findById(id)

    @GetMapping
    fun findAll(): Flow<Post> =
        repository.findAll()
}
```

Spring creates an implicit coroutine context when calling all suspending functions in the controller.

REACTIVE STREAMS VS COROUTINES - 2

```
fun getUser(userId: Int): Mono<User>
fun getAccount(accountId: Int): Mono<Account> {}

fun getAccountNo(userId: Int): Mono<String> =
    getUser(userId).flatMap {
      getAccount(it.accountId)
        .map(Account::accountNo)
    }

fun getUserName(userId: Int): Mono<String> =
    getUser(userId)
      .map { it.name }
      .onErrorReturn("Unknown: $userId")
```

```
suspend fun getUser(userId: Int): User
suspend fun getAccount(accountId: Int): Account

suspend fun getAccountNo(userId: Int): String =
   getAccount(getUser(userId).accountId).accountNo

suspend fun getUserName(userId: Int): String =
    try {
      getUser(userId).name
    } catch (e: NotFoundException) {
      "Unknown: $userId”
    }
```

REACTIVE STREAMS VS COROUTINES - 3

```
fun processOrder(existingComponents: Set<Component>) :
        Mono<Product> =
    ensureAllRequired(requiredComponents,
                existingComponents)
        ?.filter { succeeded -> succeeded }
        ?.flatMap { _ -> combineFront() }
        ?.zipWith(prepareBackend())
        ?.map { top -> assemble(top.t1, top.t2) }
        ?.zipWith(prepareTop())
        ?.zipWith(registerProduct()) {
              contentAndVesselTuple, registered ->
            inventory(registered,
                  contentAndVesselTuple.t1,
                  contentAndVesselTuple.t2)
         }
        ?.zipWith(prepareDelivery()) {
            component, registry ->
             Product(component, registry)
         }
```

```
suspend fun processOrder(existingComponents: Set<Component>):
            Product =
    if (ensureAllRequired(requiredComponents,
                    existingComponents)) {
        val product = assemble(combineFront(),
                        prepareBackend())
        val inventory = inventory(registerProduct(),
                            product,
                            prepareTop())
        val delivery = prepareDelivery()
        Product(inventory, delivery)
    }
```

COROUTINES, CONTD

- Coroutines are converted into callbacks and state machines by the Kotlin compiler.
- Coroutines are more flexible than ‘async/await’ found in other languages – can be used also outside scope of asynchronous code.
- In most languages, ‘async/await’ is per default concurrent. This is not the case with Kotlin coroutines:

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

fun loadAndCombine(name1: String, name2: String): Image =
    coroutineScope {
        val image1 = loadImage(name1)
        val image2 = loadImage(name2)
        combineImages(image1, image2)
    }
```

STRUCTURED CONCURRENCY

- While running parallel solutions, many things can go wrong: exceptions, timeouts etc.
- How do you get a thread to cancel when another thread has timed out?
- How do you make sure that all resources and threads are cancelled and all resources are cleaned up?
- 'Structured concurrency' with the help of coroutines is powerful tool to ensure that these challenges can be addressed.

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

fun loadAndCombine(name1: String, name2: String): Image =
    coroutineScope {
        val image1 = async { loadImage(name1) }
        val image2 = async { loadImage(name2) }
        combineImages(image1, image2)
    }
```

COROUTINES FLOW

- Coroutines flow implements Reactive stream's Publisher (Flux in Project reactor).
- Publisher and subscriber implement suspending methods, which means that back pressure can be implemented in a very natural way.
- The publisher simply suspends when the receiver has not requested any more data.

```
// Publisher
fun emitter(): Flow<Int> = (1..5).asFlow()

// Subscriber
suspend fun receive() {
  emitter().collect {
    print("Collect $it")
    delay(3000)
  }
}
```

PROJECT LOOM

- Project Loom is Oracle's plan to create light-weight threads on the JVM.
- Creating threads becomes cheap. When a thread waits, it can be suspended automatically.
- The release of Project Loom will profoundly affect the foundations for both reactive streams and coroutines.
- Coroutines is not just threads, and includes concepts such as structured concurrency that will keep being highly relevant.
- The final release date for Project Loom is not yet known.

CONCLUSIONS

- Coroutines offer a nice tool which requires less of a mind shift for developers than alternative solutions.
- Kotlin code always plays nice with Java solutions, and can be used side-by-side with Java in existing code bases.