Kotlin Coroutines

Deep Dive into Bytecode

#whoami

- Kotlin compiler engineer @ JetBrains
- Mostly working on JVM back-end
- Responsible for (almost) all bugs in coroutines code

Agenda

I will talk about

- State machines
- Continuations
- Suspend and resume

I won’t talk about

- Structured concurrency and cancellation
- async vs launch vs withContext
- and other library related stuff

Agenda

I will talk about

- State machines
- Continuations
- Suspend and resume

Beware, there will be code.
A lot of code.

I won’t talk about

- Structured concurrency and cancellation
- async vs launch vs withContext
- and other library related stuff

Why coroutines

- No dependency on a particular implementation of Futures or other such rich library;
- Cover equally the "async/await" use case and "generator blocks";
- Make it possible to utilize Kotlin coroutines as wrappers for different existing asynchronous APIs (such as Java NIO, different implementations of Futures, etc).

via coroutines KEEP

Getting Lazy With Kotlin

Pythagorean Triples

```
fun printPythagoreanTriples() {
   for (i in 1 until 100) {
       for (j in 1 until i) {
           for (k in i until 100) {
               if (i * i + j * j < k * k) {
                   break
               }
               if (i * i + j * j == k * k) {
                   println("$i^2 + $j^2 == $k^2")
               }
           }
       }
   }
}
```

Pythagorean Triples

```
fun printPythagoreanTriples() {
   for (i in 1 until 100) {
       for (j in 1 until i) {
           for (k in i until 100) {
               if (i * i + j * j < k * k) {
                   break
               }
               if (i * i + j * j == k * k) {
                   println("$i^2 + $j^2 == $k^2")
               }
           }
       }
   }
}
```

Pythagorean Triples

```
fun printPythagoreanTriples() {
    for (i in 1 until 100) {
        for (j in 1 until i) {
            for (k in i until 100) {
                if (i * i + j * j < k * k) {
                    break
                }
                if (i * i + j * j == k * k) {
                    println("$i^2 + $j^2 == $k^2")
                }
            }
        }
    }
}
```

Pythagorean Triples (with callback)

```
fun limitedPythagoreanTriples(limit: Int, action: (Int, Int, Int) -> Unit) {
   for (i in 1 until limit) {
       for (j in 1 until i) {
           for (k in i until limit) {
               if (i * i + j * j < k * k) {
                   break
               }
               if (i * i + j * j == k * k) {
                   action(i, j, k)
               }
           }
       }
   }
}
```

Pythagorean Triples (with callback)

```
fun limitedPythagoreanTriples(limit: Int, action: (Int, Int, Int) -> Unit) {
    for (i in 1 until limit) {
        for (j in 1 until i) {
            for (k in i until limit) {
                if (i * i + j * j < k * k) {
                    break
                }
                if (i * i + j * j == k * k) {
                    action(i, j, k)
                }
            }
        }
    }
}
```

Not what we wanted

Pythagorean Triples (with callback and proper limit)

```
class Iota(start: Int): Iterator<Int> {
   private var current = start
   override fun next() = current++
   override fun hasNext() = true
}

fun limitedPythagoreanTriples2(limit: Int, action: (Int, Int, Int) -> Unit) {
   var processed = 0
   for (i in Sequence { Iota(1) }) {
       for (j in 1 until i) {
           for (k in Sequence { Iota(i) }) {
               if (i * i + j * j < k * k) break
               if (i * i + j * j == k * k) {
                   if (processed++ >= limit) return
                   action(i, j, k)
               }
           }
       }
   }
}
```

Pythagorean Triples (with callback and proper limit)

```
class Iota(start: Int): Iterator<Int> {
   private var current = start
   override fun next() = current++
   override fun hasNext() = true
}

fun limitedPythagoreanTriples2(limit: Int, action: (Int, Int, Int) -> Unit) {
   var processed = 0
   for (i in Sequence { Iota(1) }) {
       for (j in 1 until i) {
           for (k in Sequence { Iota(i) }) {
               if (i * i + j * j < k * k) break
               if (i * i + j * j == k * k) {
                   if (processed++ >= limit) return
                   action(i, j, k)
               }
           }
       }
   }
}
```

Pythagorean Triples (lazy)

```
class PythagoreanTriples: Iterator<Triple<Int, Int, Int>> {
    private var i = 0
    private var j = 0
    private var k = 0
    override fun next(): Triple<Int, Int, Int> {
        while (true) {
            i++
            while (true) {
                j++
                while (true) {
                    k++
                    if (i * i + j * j < k * k) break
                    if (i * i + j * j == k * k) {
                        return Triple(i, j, k)
                    }
                }
            }
        }
    }
    override fun hasNext() = true
}
```

```
Sequence { PythagoreanTriples() }.take(100).forEach { (i, j, k) ->
    println("$i^2 + $j^2 == $k^2")
}
```

Pythagorean Triples (lazy)

```
class PythagoreanTriples: Iterator<Triple<Int, Int, Int>> {
    private var i = 0
    private var j = 0
    private var k = 0
    override fun next(): Triple<Int, Int, Int> {
        while (true) {
            i++
            while (true) {
                j++
                while (true) {
                    k++
                    if (i * i + j * j < k * k) break
                    if (i * i + j * j == k * k) {
                        return Triple(i, j, k)
                    }
                }
            }
        }
    }
    override fun hasNext() = true
}
```

```
Sequence { PythagoreanTriples() }.take(100).forEach { (i, j, k) ->
    println("$i^2 + $j^2 == $k^2")
}
```

Pythagorean Triples (with coroutines)

```
fun pythagoreanTriples() = sequence {
   for (i in Sequence { Iota(1) }) {
       for (j in 1 until i) {
           for (k in Sequence { Iota(i) }) {
               if (i * i + j * j < k * k) {
                   break
               }
               if (i * i + j * j == k * k) {
                   yield(Triple(i, j, k))
               }
           }
       }
   }
}
```

Pythagorean Triples (with coroutines)

```
fun pythagoreanTriples() = sequence {
   for (i in Sequence { Iota(1) }) {
       for (j in 1 until i) {
           for (k in Sequence { Iota(i) }) {
               if (i * i + j * j < k * k) {
                   break
               }
               if (i * i + j * j == k * k) {
                   yield(Triple(i, j, k))
               }
           }
       }
   }
}
```

~300 lines of bytecode
Cannot represent in kotlin

Simpler Version

```
fun firstThree() = sequence {
   yield(1)
   yield(2)
   yield(3)
}
```

Simpler Version

```
fun firstThree() = sequence {
    yield(1)
    yield(2)
    yield(3)
}
```

Simpler Version: stdlib

```
interface Continuation {
   fun resume()
}

class SequenceIterator<T>: Iterator<T>, Continuation {
   private var nextValue: Any? = null
   private var ready = false
   private var done = false
   internal var nextStep: Continuation? = null

   /* suspend */ fun yield(value: T, continuation: Continuation) {
       nextValue = value
       nextStep = continuation
       ready = true
   }

   override fun resume() {
       done = true
   }

   override fun hasNext(): Boolean {
       while (true) {
           if (ready) return true
           if (done) return false
           val step = nextStep!!
           nextStep = null
           step.resume()
       }
   }

   override fun next(): T {
       if (!hasNext()) error("already done")
       ready = false
       val result = nextValue as T
       nextValue = null
       return result
   }
}
```

Simpler Version: Compiler Generated

```
class FirstThreeContinuation(var completion: SequenceIterator<Int>): Continuation {
    private var index = 0
    override fun resume() {
        when (index) {
            0 -> {
                index = 1
                completion.yield(1, this)
            }
            1 -> {
                index = 2
                completion.yield(2, this)
            }
            2 -> {
                index = 3
                completion.yield(3, this)
            }
        }
    }
}
```

Continuations

Continuations

In computer science and computer programming, a **continuation** is an abstract representation of the control state of a computer program. via Wikipedia

Continuation is a state of coroutine.

Suspend Functions

```
suspend fun dummy() {}

suspend fun dummy2() {
    dummy()
    dummy()
}
```

Suspend Functions: Continuation

```
suspend fun dummy() {}

suspend fun dummy2() {
    dummy()
    dummy()
}
```

```
class Dummy2Continuation(completion: Continuation<*>):
        ContinuationImpl(completion) {
    var result: Result<Any?>? = null
    var label = 0
    override fun invokeSuspend(result: Result<Any?>): Any? {
        this.result = result
        return dummy2(this)
    }

    /* parent */
    override fun resumeWith(result: Result<Any?>) {
        try {
            val outcome = invokeSuspend(result)
            if (outcome == COROUTINE_SUSPENDED) return
            completion.resumeWith(Result.success(outcome))
        } catch (e: Throwable) {
             completion.resumeWith(Result.failure<Any?>(e))
        }
    }
}
```

Suspend Functions: State Machine

```
suspend fun dummy2() {
    dummy()
    dummy()
}

fun dummy2(continuation: Continuation<Unit>): Any? {
    val myContinuation: Dummy2Continuation =
        if (continuation is Dummy2Continuation)
            continuation
        else
            Dummy2Continuation(continuation)
    val result = myContinuation.result
    while (true) {
        when (myContinuation.label) {
            0 -> {
                if (result.value is Result.Failure)
                    throw result.value.exception
                myContinuation.label++
                val outcome = dummy(myContinuation)
                if (outcome == COROUTINE_SUSPENDED) return outcome
            }
            1 -> {
                if (result.value is Result.Failure)
                    throw result.value.exception
                myContinuation.label++
                val outcome = dummy(myContinuation)
                if (outcome == COROUTINE_SUSPENDED) return outcome
            }
            2 -> {
                if (result.value is Result.Failure)
                    throw result.value.exception
                return Unit
            }
            else -> throw IllegalStateException("call to 'resume'
before 'invoke' with coroutine")
        }
    }
}
```

WOW

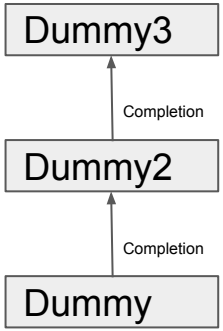
Such state

Much state-machine

Very allocation

Continuations: Completion

```
suspend fun dummy() {}

suspend fun dummy2() {
    dummy()
    dummy()
}

suspend fun dummy3() {
    dummy2()
    dummy2()
}
/* MEMBER ME? */
override fun resumeWith(result: Result<Any?>) {
    try {
        val outcome = invokeSuspend(result)
        if (outcome == COROUTINE_SUSPENDED) return
        completion.resumeWith(Result.success(outcome))
    } catch (e: Throwable) {
        completion.resumeWith(Result.failure<Any?>(e))
    }
}
```



SUSPEND/RESUME

JUST PRESS THE POWER BUTTON

Sequence

```
suspend fun yield(value: T) = suspendCoroutineUninterceptedOrReturn<Unit> {
    nextValue = value
    nextStep = it
    ready = true
    COROUTINE_SUSPENDED
}

override fun hasNext(): Boolean {
    while (true) {
        if (ready) return true
        if (done) return false
        val step = nextStep!!
        nextStep = null
        step.resume()
    }
}
```

Threads

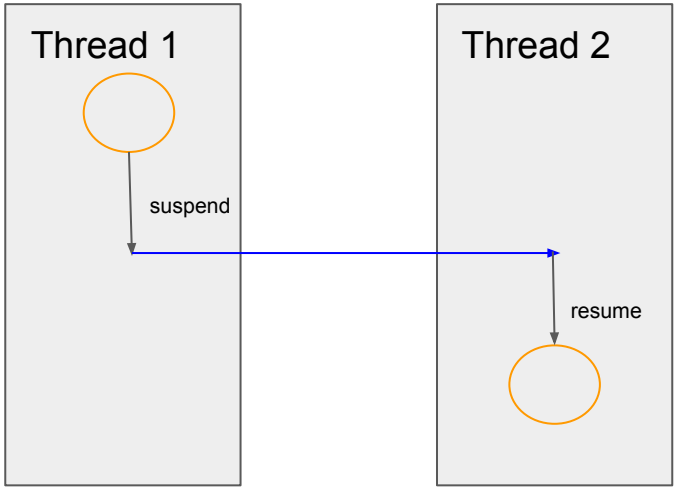
```
var proceed = suspend {
    printThreadId()
}

suspend fun suspendHere() = suspendCoroutine<Unit> {
    proceed = { it.resume(Unit) }
}

suspend fun printThreadId() {
    println(Thread.currentThread().id)
    suspendHere()
    println(Thread.currentThread().id)
}

fun builder(c: suspend () -> Unit) {
    c.startCoroutine(object : Continuation<Unit>{
        override val context = EmptyCoroutineContext
        override fun resumeWith(result: Result<Unit>) {
            result.getOrThrow()
        }
    })
}
```

```
fun main() {
    builder { proceed() }
    thread {
        builder { proceed() }
    }.join()
}
```

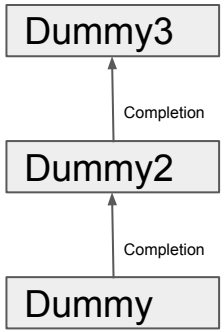
Threads

```
var proceed = suspend {
    printThreadId()
}

suspend fun suspendHere() = suspendCoroutine<Unit> {
    proceed = { it.resume(Unit) }
}

suspend fun printThreadId() {
    println(Thread.currentThread().id)
    suspendHere()
    println(Thread.currentThread().id)
}

fun builder(c: suspend () -> Unit) {
    c.startCoroutine(object : Continuation<Unit>{
        override val context = EmptyCoroutineContext
        override fun resumeWith(result: Result<Unit>) {
            result.getOrThrow()
        }
    })
}
```

```
fun main() {
    builder { proceed() }
    thread {
        builder { proceed() }
    }.join()
}
```



Completions Revised



If coroutine suspends, its continuation

1) Must be saved or coroutine is gone
2) Can be used to resume coroutine wherever we want

Completions form linked list, that represents call stack* of the coroutine

* And kotlinx.coroutines uses the these to recover stack trace

Suspend Lambdas

```
fun firstThree() = sequence {
    yield(1)
    yield(2)
    yield(3)
}
```

Wtf Is Lambda, Anyway

```
var i = 0
val c = { i++ }
```

```
class MyLamba(val i: kotlin.jvm.internal.Ref.IntRef):
        kotlin.jvm.internal.Lambda<Int>(arity = 0),
        kotlin.jvm.functions.Function0<Int> {
    override fun invoke(): Int {
        return i.element++
    }
}
```

Wtf Is Lambda, Anyway

```
var i = 0
val c = { i++ }
```

```
class MyLamba(val i: kotlin.jvm.internal.Ref.IntRef):
        kotlin.jvm.internal.Lambda<Int>(arity = 0),
        kotlin.jvm.functions.Function0<Int> {
    override fun invoke(): Int {
        return i.element++
    }
}
```

Wtf Is Suspend Lambda

```
var i = 0
val c : suspend () -> Int = { i++ }
```

```
class MySuspendLamba(i: IntRef, val completion: Continuation<*>):
        SuspendLambda(arity = 1, completion = completion),
        Function1<Continuation<Int>, Any?> {
    var label: Int = 0
    val i = i
    override fun invokeSuspend(result: Result<Any?>): Any? {
        when(label) {
            0 -> {
                if (result.value is Result.Failure) throw result.value.exception
                return i.element++
            }
            else -> throw IllegalStateException("call to 'resume' before 'invoke' with
coroutine")
        }
    }
    override fun create(completion: Continuation<*>): Continuation<Unit> {
        return MySuspendLamba(i, completion)
    }
    override fun invoke(completion: Any?): Any? {
        return (create(completion as Continuation<*>) as
MySuspendLamba).invokeSuspend(Result.success(Unit))
    }
}
```

Wtf Is Suspend Lambda

```
var i = 0
val c : suspend () -> Int = { i++ }
```

```
class MySuspendLamba(i: IntRef, completion: Continuation<*>):
        SuspendLambda(arity = 1, completion = completion),
        Function1<Continuation<Int>, Any?> {
    var label: Int = 0
    val i = i
    override fun invokeSuspend(result: Result<Any?>): Any? {
        when(label) {
            0 -> {
                if (result.value is Result.Failure) throw result.value.exception
                return i.element++
            }
            else -> throw IllegalStateException("call to 'resume' before 'invoke' with
coroutine")
        }
    }
    override fun create(completion: Continuation<*>): Continuation<Unit> {
        return MySuspendLamba(i, completion)
    }
    override fun invoke(completion: Any?): Any? {
        return (create(completion as Continuation<*>) as
MySuspendLamba).invokeSuspend(Result.success(Unit))
    }
}
```

Wtf Is Suspend Lambda

Suspend lambdas:

- Incapsulate state => suspend lambda is a continuation
- Do work in state-machine => suspend lambda is a coroutine
- Allocates itself upon invocation => because it is continuation

suspendCoroutineUninterceptedOrReturn

```
public suspend inline fun <T> suspendCoroutineUninterceptedOrReturn(crossinline block: (Continuation<T>) -> Any?): T =
    throw NotImplementedError("Implementation of suspendCoroutineUninterceptedOrReturn is intrinsic")
```

```
suspend fun dummy() = suspendCoroutineUninterceptedOrReturn<Unit> { println(it) }

fun dummy(it: Continuation<Unit>): Any? = println(it)
```

Why suspendCoroutine

```
fun builder(c: suspend () -> Unit) {
    c.startCoroutine(object : Continuation<Unit> {
        override val context = EmptyCoroutineContext
        override fun resumeWith(result: Result<Unit>) {
            result.getOrThrow()
        }
    })
}

fun main() {
    try {
        builder {
            suspendCoroutineUninterceptedOrReturn<Unit> {
                it.resumeWithException(IllegalStateException("boo"))
                Unit
            }
        }
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

Why suspendCoroutine

```
kotlin.KotlinNullPointerException
    at kotlin.coroutines.jvm.internal.ContinuationImpl.releaseIntercepted(ContinuationImpl.kt:117)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:38)
    at kotlin.coroutines.ContinuationKt.startCoroutine(Continuation.kt:113)
    at threads.TestKt.builder(test.kt:9)
    at threads.TestKt.main(test.kt:19)
    at threads.TestKt.main(test.kt)
```

Why suspendCoroutine (Kotlin/JS)

```
Unhandled JavaScript exception:
captureStack@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:24806:27
Exception@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:26677:7
RuntimeException@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:26706:7
RuntimeException_init_0@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:26717:7
NullPointerException@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:26826:7
throwNPE@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:26626:13
Kotlin.ensureNotNull@https://try.kotlinlang.org/static/kotlin/1.3.11/kotlin.js:819:28
main@https://try.kotlinlang.org/static/index.js line 56906 > eval:102:17
moduleId<@https://try.kotlinlang.org/static/index.js line 56906 > eval:110:3
@https://try.kotlinlang.org/static/index.js line 56906 > eval:5:16
RunProvider$loadJsFromServer$lambda/<@https://try.kotlinlang.org/static/index.js:56906:71
j@https://try.kotlinlang.org/static/lib/jquery/dist/jquery.min.js:2:27239
fireWith@https://try.kotlinlang.org/static/lib/jquery/dist/jquery.min.js:2:28057
x@https://try.kotlinlang.org/static/lib/jquery/dist/jquery.min.js:4:21841
b@https://try.kotlinlang.org/static/lib/jquery/dist/jquery.min.js:4:25897
```

Why suspendCoroutine

Why suspendCoroutine: correct way

```
fun builder(c: suspend () -> Unit) {
    c.startCoroutine(object : Continuation<Unit> {
        override val context = EmptyCoroutineContext
        override fun resumeWith(result: Result<Unit>) {
            result.getOrThrow()
        }
    })
}

fun main() {
    try {
        builder {
            suspendCoroutine<Unit> {
                it.resumeWithException(IllegalStateException("boo"))
            }
        }
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

Why suspendCoroutine

```
java.lang.IllegalStateException: boo
    at threads.TestKt$main$1.invokeSuspend(test.kt:21)
    at kotlin.coroutines.jvm.internal.BaseContinuationImpl.resumeWith(ContinuationImpl.kt:32)
    at kotlin.coroutines.ContinuationKt.startCoroutine(Continuation.kt:113)
    at threads.TestKt.builder(test.kt:9)
    at threads.TestKt.main(test.kt:19)
    at threads.TestKt.main(test.kt)
```

Further Read

- Coroutines KEEP
  https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md
- Coroutines Guide
  https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html
- Structured Concurrency
  https://medium.com/@elizarov/structured-concurrency-722d765aa952
- Explicit Concurrency
  https://medium.com/@elizarov/explicit-concurrency-67a8e8fd9b25


THANK
YOU!