Kotlin Coroutines Reloaded

Presented at JVM Language Summit, 2017

/Roman Elizarov @ JetBrains

Speaker: Roman Elizarov

- 16+ years experience
- Previously developed high-perf trading software
  @ Devexperts
- Teach concurrent & distributed programming
  @ St. Petersburg ITMO University
- Chief judge
  @ Northeastern European Region of ACM ICPC
- Now work on Kotlin
  @ JetBrains

Agenda

- Recap of Kotlin coroutines prototype
  - Presented @ JVMLS, 2016 by Andrey Breslav
- The issues in the prototype design
  - and the solutions that were found
- Design released to public in Kotlin 1.1
- Evolved coroutines support libraries
- Challenges & future directions

Recap of Kotlin coroutines prototype

Presented @ JVMLS, 2016 by Andrey Breslav

Async the C# (JS, TS, Dart, etc) way

```
async Task<String> Work() { … }

async Task MoreWork()
{
    Console.WriteLine("Work started");
    var str = await Work();
    Console.WriteLine($"Work completed {str}");
}
```

Async the C# (JS, TS, Dart, etc) way

```
async Task<String> work() { … }

async Task MoreWork()
{
    Console.WriteLine("Work started");
    var str = await Work();
    Console.WriteLine($"Work completed {str}");
}
```

Async the Kotlin way (prototype)

```kotlin
fun work(): CompletableFuture<String> { … }

fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```

Async the Kotlin way (prototype)

```
fun work(): CompletableFuture<String> { … }

fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```

Functions vs Keywords

Extensibility

Runs on stock JVM 1.6+

Purely local -- no global bytecode transforms

Suspending functions

A grand unification between async/await and generate/yield

Suspending functions: use

```
val str = await(work())
```

Suspending functions: use

```
              CompletableFuture<String>

val str = await(work()) // String result
```

Suspending functions: declare (prototype)

```
          CompletableFuture<String>

val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Unit
```

Suspending functions: declare (prototype)

CompletableFuture<String>

```
val str = await(work()) // String result
```

Magic signature transformation

callback

void

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Unit
```

Suspending functions: declare (prototype)

```
        CompletableFuture<String>
val str = await(work()) // String result
```

callback

void

```kotlin
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Unit
```

```kotlin
interface Continuation<in T> {
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Continuation is a generic callback interface

Suspending functions: implement (prototype)

```
         CompletableFuture<String>
val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}
```

Suspending functions: implement (prototype)

```
        CompletableFuture<String>

val str = await(work()) // String result


suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}
```

Suspending functions: implement (prototype)

```
        CompletableFuture<String>

val str = await(work()) // String result


suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}
```

Suspending functions: implement (prototype)

```
        CompletableFuture<String>

val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                                 else c.resume(value)
    }
}
```

Suspending functions: implement (prototype)

```
                  CompletableFuture<String>

val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}
```

Simple, but wrong!

A problematic example

Where grand vision meets reality

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                                  else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

What if work() always returns a future that is *already complete*?

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

`problem$stateMachine`

A problematic example

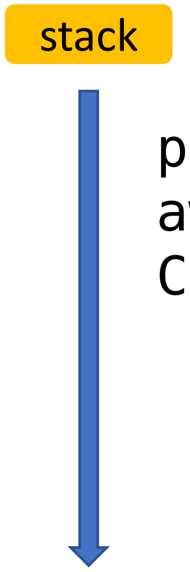
```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                         else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

problem$stateMachine
await

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```




```
problem$stateMachine
await
CompletableFuture.whenComplete
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

```
problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                             else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                             else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

```
problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda
ContinuationImpl.resume
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

```
problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda
ContinuationImpl.resume
problem$stateMachine
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

stack

```
problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda
ContinuationImpl.resume
problem$stateMachine
await
```

A problematic example

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>) {
    f.whenComplete { value, exception ->
        if (exception != null) c.resumeWithException(exception)
                          else c.resume(value)
    }
}

fun problem() = async {
    repeat(10_000) {
        await(work())
    }
}
```

StackOverflowError

stack

```
problem$stateMachine
await
CompletableFuture.whenComplete
await$lambda
ContinuationImpl.resume
problem$stateMachine
await
…
```

A solution

A difference between knowing the path & walking the path.

A solution

```
        CompletableFuture<String>

val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Unit
```

A solution (0): stack unwind convention

```
        CompletableFuture<String>

val str = await(work()) // String result

suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

A solution (0)

```
                CompletableFuture<String>
val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
T | COROUTINE_SUSPENDED
```

A solution (0)

```
val str = await(work()) // String result
```

$\overbrace{\texttt{work()}}^{\texttt{CompletableFuture<String>}}$

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
T | COROUTINE_SUSPENDED
```

Did not suspend -> Returns result

A solution (0)

```
        CompletableFuture<String>
val str = await(work()) // String result
```

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
T | COROUTINE_SUSPENDED
```

Did not suspend -> Returns result

Did suspend -> WILL invoke continuation

A solution (0)

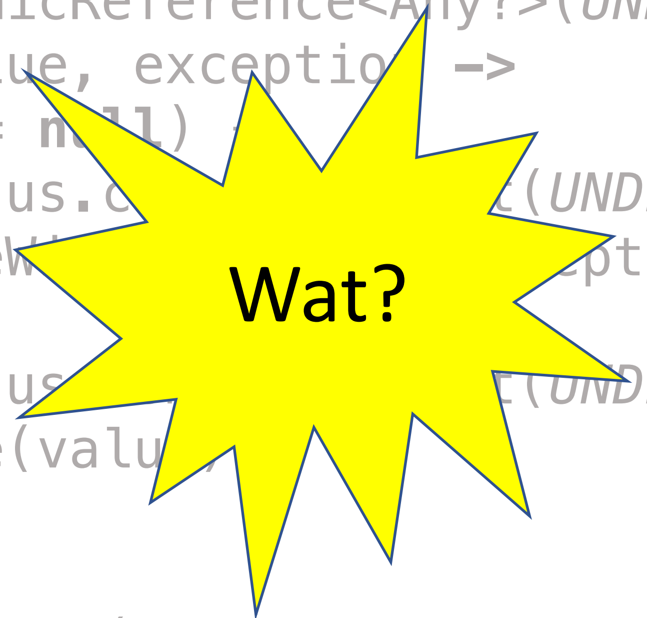
```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? {
    …
}
```

A solution?

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? {
    val consensus = AtomicReference<Any?>(UNDECIDED)
    f.whenComplete { value, exception ->
        if (exception != null) {
            if (!consensus.compareAndSet(UNDECIDED, Fail(exception)))
                c.resumeWithException(exception)
        } else {
            if (!consensus.compareAndSet(UNDECIDED, value))
                c.resume(value)
        }
    }
    consensus.compareAndSet(UNDECIDED, COROUTINE_SUSPENDED)
    val result = consensus.get()
    if (result is Fail) throw result.exception
    return result
}
```

A solution?

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? {
    val consensus = AtomicReference<Any?>(UNDECIDED)
    f.whenComplete { value, exceptio  ->
        if (exception != null)
            if (!consensus.c          t(UNDECIDED, Fail(exception)))
                c.resumeW           ption)
        } else {
            if (!consensus           t(UNDECIDED, value))
                c.resume(valu
        }
    }
    consensus.compareAndSet(UNDECIDED, COROUTINE_SUSPENDED)
    val result = consensus.get()
    if (result is Fail) throw result.exception
    return result
}
```



A solution (1): Call/declaration fidelity

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
suspend fun <T> await(f: CompletableFuture<T>): T
```

natural signature

A solution (1)

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

Compiles as (on JVM)

```
suspend fun <T> await(f: CompletableFuture<T>): T
```

A solution (1)

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

Compiles as (on JVM)

CPS Transformation

```
suspend fun <T> await(f: CompletableFuture<T>): T
```

A solution (1)

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
suspend fun <T> await(f: CompletableFuture<T>)
```

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Recover continuation

A solution (1)

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
suspend fun <T> await(f: CompletableFuture<T>)
```

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Recover continuation

Inspired by call/cc from Scheme

A solution (1)

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any?
```

```
suspend fun <T> await(f: CompletableFuture<T>)
```

Works as

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Inspired by call/cc from Scheme

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

```
inline suspend fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?): T
```

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

```
inline suspend fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?): T
```

T | COROUTINE_SUSPENDED

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Intrinsic

```
inline suspend fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?): T
```

```
T | COROUTINE_SUSPENDED
```

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Intrinsic

```
inline suspend fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?): T
```

Compiles as (on JVM)

CPS Transformation

```
fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?,
    c: Continuation<T>): Any? =
        block(c)
```

A solution (1)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Intrinsic

```
inline suspend fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?): T
```

Compiles as (on JVM)

CPS Transformation

```
fun <T> suspendCoroutineOrReturn(
    crossinline block: (Continuation<T>) -> Any?,
    c: Continuation<T>): Any? =
        block(c)
```

A solution (2): Tail suspension

A solution (2)

Tail suspend invocation:
Continuation pass-through

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

A solution (2)

Tail suspend invocation:
Continuation pass-through

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Compiles as (on JVM)

CPS Transformation

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? =
    suspendCoroutineOrReturn(c) { c ->
        …
    }
```

A solution (2)

Tail suspend invocation:
Continuation pass-through

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Compiles as (on JVM)

CPS Transformation

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? =
    suspendCoroutineOrReturn(c) { c ->
        …
    }
```

A solution (2)

Tail suspend invocation:
Continuation pass-through

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutineOrReturn { c ->
        …
    }
```

Compiles as (on JVM)

CPS Transformation

```
fun <T> await(f: CompletableFuture<T>, c: Continuation<T>): Any? {
    …
}
```

A solution (3): Abstraction

A solution (3)

```
public inline suspend fun <T> suspendCoroutine(
    crossinline block: (Continuation<T>) -> Unit): T =
        suspendCoroutineOrReturn { c: Continuation<T> ->
            val safe = SafeContinuation(c)
            block(safe)
            safe.getResult()
        }
```

A solution (3)

```
public inline suspend fun <T> suspendCoroutine(
    crossinline block: (Continuation<T>) -> Unit): T =
        suspendCoroutineOrReturn { c: Continuation<T> ->
            val safe = SafeContinuation(c)
            block(safe)
            safe.getResult()
        }
```

A solution (3)

Any? Is gone

```
public inline suspend fun <T> suspendCoroutine(
    crossinline block: (Continuation<T>) -> Unit): T =
        suspendCoroutineOrReturn { c: Continuation<T> ->
            val safe = SafeContinuation(c)
            block(safe)
            safe.getResult()
        }
```

A solution (3)

```
public inline suspend fun <T> suspendCoroutine(
    crossinline block: (Continuation<T>) -> Unit): T =
        suspendCoroutineOrReturn { c: Continuation<T> ->
            val safe = SafeContinuation(c)
            block(safe)
            safe.getResult()
        }
```

Encapsulates result consensus algorithm

A solution (4): Putting it all together

A solution (4)

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutine { c ->
        f.whenComplete { value, exception ->
            if (exception != null) c.resumeWithException(exception)
                                   else c.resume(value)
        }
    }
```

Looks simple, works correctly!

Recap steps to solution

- T | COROUTINE_SUSPENDED (Any?) to allow invocations that do not suspend and thus avoid StackOverflowError
- Call/declaration fidelity via CPS transformation
- Introduce call/cc (suspendCoroutineOrReturn) to recover hidden continuation
- Tail call invocation support to recover prototype semantics
- Use abstraction (suspendCoroutine) to hide implementation complexities from end-users

The final touch: await extension

```
suspend fun <T> await(f: CompletableFuture<T>): T =
    suspendCoroutine { c ->
        f.whenComplete { value, exception ->
            if (exception != null) c.resumeWithException(exception)
                                   else c.resume(value)
        }
    }
```

The final touch: await extension

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { c ->
        whenComplete { value, exception ->
            if (exception != null) c.resumeWithException(exception)
                                   else c.resume(value)
        }
    }
```

The final touch: await extension

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { c ->
        whenComplete { value, exception ->
            if (exception != null) c.resumeWithException(exception)
                                   else c.resume(value)
        }
    }

fun moreWork() = async {
    println("Work started")
    val str = await(work())
    println("Work completed: $str")
}
```

The final touch: await extension

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine { c ->
        whenComplete { value, exception ->
            if (exception != null) c.resumeWithException(exception)
                                   else c.resume(value)
        }
    }

fun moreWork() = async {
    println("Work started")
    val str = work().await()
    println("Work completed: $str")
}
```

Reads left-to-right just like it executes

Kotlin

Coroutine builders

The mystery of inception

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c(controller).resume(Unit)
    return controller.future
}
```

Coroutine builders (prototype)

A special modifier

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c(controller).resume(Unit)
    return controller.future
}
```

Coroutine builders (prototype)

Magic signature transformation

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c(controller).resume(Unit)
    return controller.future
}
```

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c(controller).resume(Unit)
    return controller.future
}
```

A boilerplate to **start coroutine**

Coroutine builders (prototype)

```kotlin
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T, c: Continuation<Nothing>) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable,
                                 c: Continuation<Nothing>) {
        future.completeExceptionally(exception)
    }
}
```

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T, c: Continuation<Nothing>) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable,
                                 c: Continuation<Nothing>) {
        future.completeExceptionally(exception)
    }
}
```

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T, c: Continuation<Nothing>) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable,
                                 c: Continuation<Nothing>) {
        future.completeExceptionally(exception)
    }
}
```

was never used

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable) {
        future.completeExceptionally(exception)
    }
}
```

Coroutine builders (prototype)

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable) {
        future.completeExceptionally(exception)
    }
}
```

Looks like Continuation<T>

Coroutine builders (prototype)

Something that takes Continuation<T>

```
fun <T> async(
    coroutine c: FutureController<T>.() -> Continuation<Unit>
): CompletableFuture<T> { … }

class FutureController<T> {
    val future = CompletableFuture<T>()

    operator fun handleResult(value: T) {
        future.complete(value)
    }

    operator fun handleException(exception: Throwable) {
        future.completeExceptionally(exception)
    }
}
```

Coroutine builders: evolved

Coroutine builders

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c.startCoroutine(completion = controller)
    return controller.future
}
```

Coroutine builders

No special **coroutine** keyword

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c.startCoroutine(completion = controller)
    return controller.future
}
```

natural signature

Coroutine builders

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> {
    val controller = FutureController<T>()
    c.startCoroutine(completion = controller)
    return controller.future
}
```

Provided by standard library

Coroutine builders

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> { … }

class FutureController<T> : Continuation<T> {
    val future = CompletableFuture<T>()

    override fun resume(value: T) {
        future.complete(value)
    }

    override fun resumeWithException(exception: Throwable) {
        future.completeExceptionally(exception)
    }
}
```

Coroutine builders

Serves as *completion* continuation for coroutine

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> { … }

class FutureController<T> : Continuation<T> {
    val future = CompletableFuture<T>()

    override fun resume(value: T) {
        future.complete(value)
    }

    override fun resumeWithException(exception: Throwable) {
        future.completeExceptionally(exception)
    }
}
```

Bonus features

Free as in cheese

Arbitrary suspending calls

```
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }
```

Arbitrary suspending calls

```
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }
```

Arbitrary suspending calls

```
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }
```

Arbitrary suspending calls

```
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }
```

Returns CompletableFuture

```
fun moreWork() = async {
    println("Work started")
    val str = work().await()
    println("Work completed: $str")
}
```

Arbitrary suspending calls

```
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }


suspend fun moreWork(): T = suspending {
    println("Work started")
    val str = work().await()
    println("Work completed: $str")
}
```

Arbitrary suspending calls

```kotlin
suspend fun <T> suspending(block: suspend () -> T): T =
    suspendCoroutine { continuation ->
        block.startCoroutine(completion = continuation)
    }


suspend fun moreWork(): T = suspending {
    println("Work started")
    val str = work().await()
    println("Work completed: $str")
}
```

Tail suspend invocation

Non-tail (arbitrary) suspend invocation

Arbitrary suspending calls

```
suspend fun moreWork() {
    println("Work started")
    val str = work().await()
    println("Work completed: $str")
}
```

Non-tail (arbitrary) suspend invocation

Stackless vs Stackful coroutines

The crucial distinction

Stackless vs Stackful coroutines

	Stackless	Stackful
Restrictions	Use in special ctx	Use anywhere
Implemented in	C#, Scala, Kotlin, …	Quasar, Javaflow, …

Stackless vs Stackful coroutines

	Stackless	Stackful
Restrictions	Use in special ctx	Use anywhere
Implemented in	C#, Scala, Kotlin, …	Quasar, Javaflow, …

Stackless vs Stackful coroutines

	Stackless	**Stackful**
Restrictions	Use in special ctx	Use anywhere
Implemented in	C#, Scala, Kotlin, …	Quasar, Javaflow, …
Can suspend in?	**suspend** functions	**throws** SuspendExecution / @Suspendable functions

Stackless vs Stackful coroutines

	Stackless	Stackful
Restrictions	Use in special ctx	Use anywhere
Implemented in	C#, Scala, …	Kotlin, Quasar, Javaflow, …

Stackless vs Stackful coroutines

	Stackless	Stackful
Restrictions	Use in special ctx	Use anywhere
Implemented in	C#, Scala, Kotlin, Quasar, JavaFlow, …	LISP, Go, …

Stackless vs Stackful coroutines

	Stackless	
Restrictions	Use in spe	where
Implemented in	C#, Sca	Quasar, Javaflow, …

False dichotomy

Async vs Suspending functions

The actual difference

```kotlin
fun work() = async { … }
```

VS

```kotlin
suspend fun work() { … }
```

```kotlin
fun work(): CompletableFuture<String> = async { … }
```

In Kotlin you *have* a choice

VS

```kotlin
suspend fun work(): String { … }
```

```
fun workAsync(): CompletableFuture<String> = async { … }
```

VS

```
suspend fun work(): String { … }
```

The problem with async

workAsync() **VALID** –> produces `CompletableFuture<String>`

concurrent & async behavior

workAsync().*await*() **VALID** –> produces `String`

sequential behavior

The most *needed* one, yet the most syntactically clumsy, esp. for suspend-heavy (CSP) code style

Kotlin **suspending functions**
imitate <u>sequential</u> behavior
by default

Concurrency is hard
Concurrency has to be explicit

Composability

Making those legos click

Builders

```
fun <T> async(
    c: suspend () -> T
): CompletableFuture<T> { … }
```

Builders

```
fun <T> async(
    c: suspend () -> T
): ListenableFuture<T> { … }
```

Builders

```
fun <T> async(
    c: suspend () -> T
): MyOwnFuture<T> { … }
```

Suspending functions

```
fun <T> async(
    c: suspend () -> T
): MyOwnFuture<T> { … }

suspend fun <T> CompletableFuture<T>.await(): T { … }
```

Suspending functions

```
fun <T> async(
    c: suspend () -> T
): MyOwnFuture<T> { … }

suspend fun <T> ListenableFuture<T>.await(): T { … }
```

Suspending functions

```
fun <T> async(
    c: suspend () -> T
): MyOwnFuture<T> { … }

suspend fun <T> MyOwnFuture<T>.await(): T { … }
```

Suspending functions

```
fun <T> async(
    c: suspend () -> T
): MyOwnFuture<T> { … }

suspend fun <T> MyOwnFuture<T>.await(): T { … }

fun moreWorkAsync() = async {
    println("Work started")
    val str = workAsync().await()
    println("Work completed: $str")
}
```

All combinations need to *compose*

Composability: evolved

- Kotlin **suspending functions** are composable *by default*
  - Asynchronous use-case
  - Define asynchronous suspending functions anywhere
  - Use them inside any asynchronous coroutine builder
    - Or inside other suspending functions

- **generate/yield** coroutines are synchronous
  - Restricted via a special @RestrictsSuspension annotation
  - Opt-in to define synchronous coroutines

Coroutine context

The last piece of composability puzzle

asyncUI (prototype)

```
asyncUI {
    val image = await(loadImage(url))
    myUI.updateImage(image)
}
```

Supposed to work in UI thread

asyncUI (prototype)

A special *coroutine builder*

```
asyncUI {
    val image = await(loadImage(url))
    myUI.updateImage(image)
}
```

asyncUI (prototype)

```
asyncUI {
    val image = await(loadImage(url))
    myUI.updateImage(image)
}
```

A special *suspending function* in its scope

Composability problem again!

asyncUI: evolved

```
async(UI) {
    val image = loadImageAsync(url).await()
    myUI.updateImage(image)
}
```

asyncUI: evolved

```
async(UI) {
    val image = loadImageAsync(url).await()
    myUI.updateImage(image)
}
```

Explicit context *convention* for all builders

Can *intercept* continuation to resume in the appropriate thread

The actual Continuation interface

```
async(UI) {
    val image = loadImageAsync(url).await()
    myUI.updateImage(image)
}
```

Is used to transparently lookup an interceptor for resumes

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

The actual Continuation interface

```
async(UI) {
    val image = loadImageAsync(url).await()
    myUI.updateImage(image)
}
```

Does not have to be aware – receives intercepted continuation to work with

```
interface Continuation<in T> {
    val context: CoroutineContext
    fun resume(value: T)
    fun resumeWithException(exception: Throwable)
}
```

Thread-safety

A million-dollar quest for correctly-synchronized (data-race free) code

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()
    val str = work().await()
    list.add(str)
}
```

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()   } One thread
    val str = work().await()
    list.add(str)
}
```

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()
    val str = work().await()
    list.add(str)
}
```

One thread

Suspends / resumes

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()
    val str = work().await()
    list.add(str)
}
```

One thread

Suspends / resumes

Another thread

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()
    val str = work().await()
    list.add(str)
}
```

One thread

Suspends / resumes

Another thread

Is there a data race?

Do we need volatile when spilling locals to a state machine?

A need for happens-before relation

```
fun moreWork() = async {
    val list = ArrayList<String>()
    val str = work().await()
    list.add(str)
}
```

happens-before

There is no data race here!

await establishes happens-before relation

Challenges

If it only was all that simple…

Thread confinement vs coroutines

```
fun moreWork() = async {
    synchronized(monitor) {
        val str = work().await()
    }
}
```

Thread confinement vs coroutines

```
fun moreWork() = async {
    synchronized(monitor) {
        val str = work().await()
    }
}
```

MONITORENTER in one thread
Suspends / resumes
MONITOREXIT in another thread

IllegalMonitorStateException

Thread confinement vs coroutines

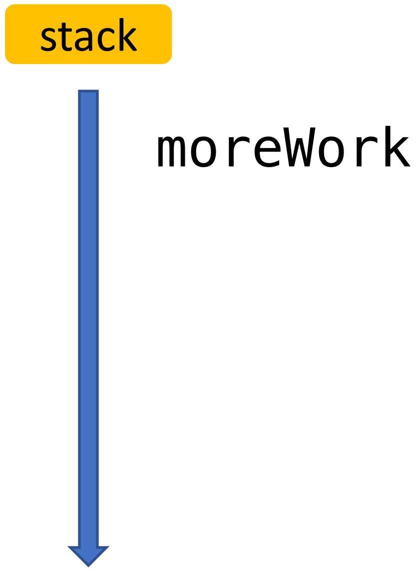
- Monitors
- Locks (j.u.c.l.ReentrantLock)
- Thread-locals
- …
- Thread.currentThread()

A **CoroutineContext** is a map of coroutine-local elements as replacement

Stack traces in exceptions

Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```



Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```

stack

```
moreWork
work
```

Stack traces in exceptions

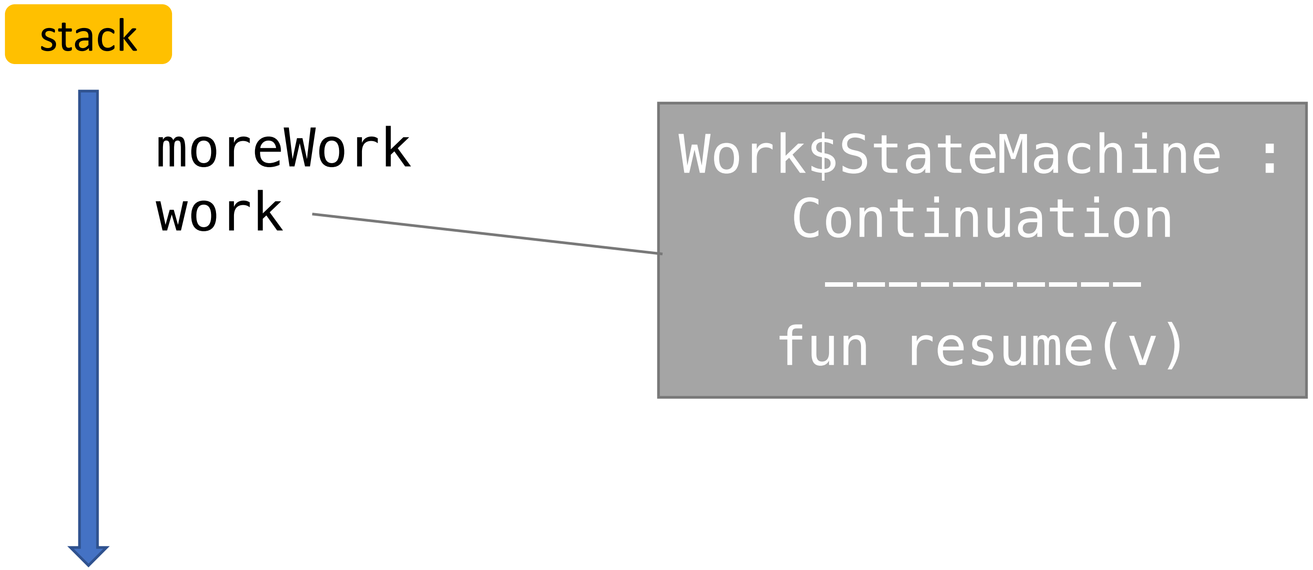
```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```

stack

```
moreWork
work
await
```

Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```



Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```

stack

```
Work$StateMachine.resume
work
```

Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```

JVM stack

```
Work$StateMachine.resume
work
```

Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```

JVM stack

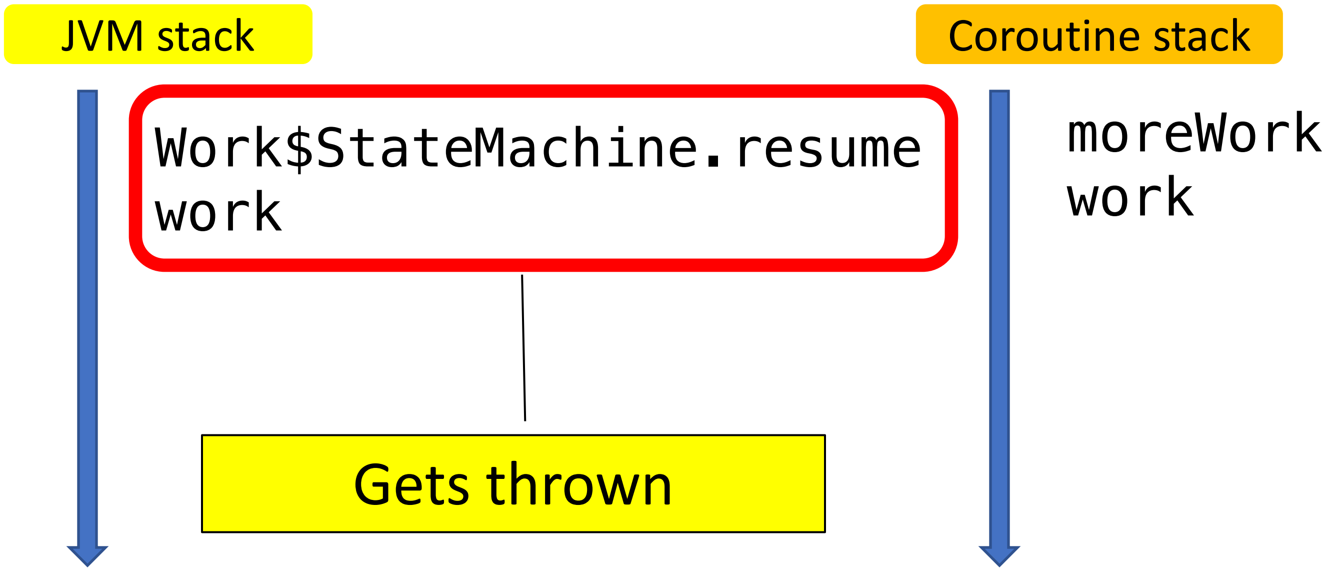
```
Work$StateMachine.resume
work
```

Coroutine stack

```
moreWork
work
```

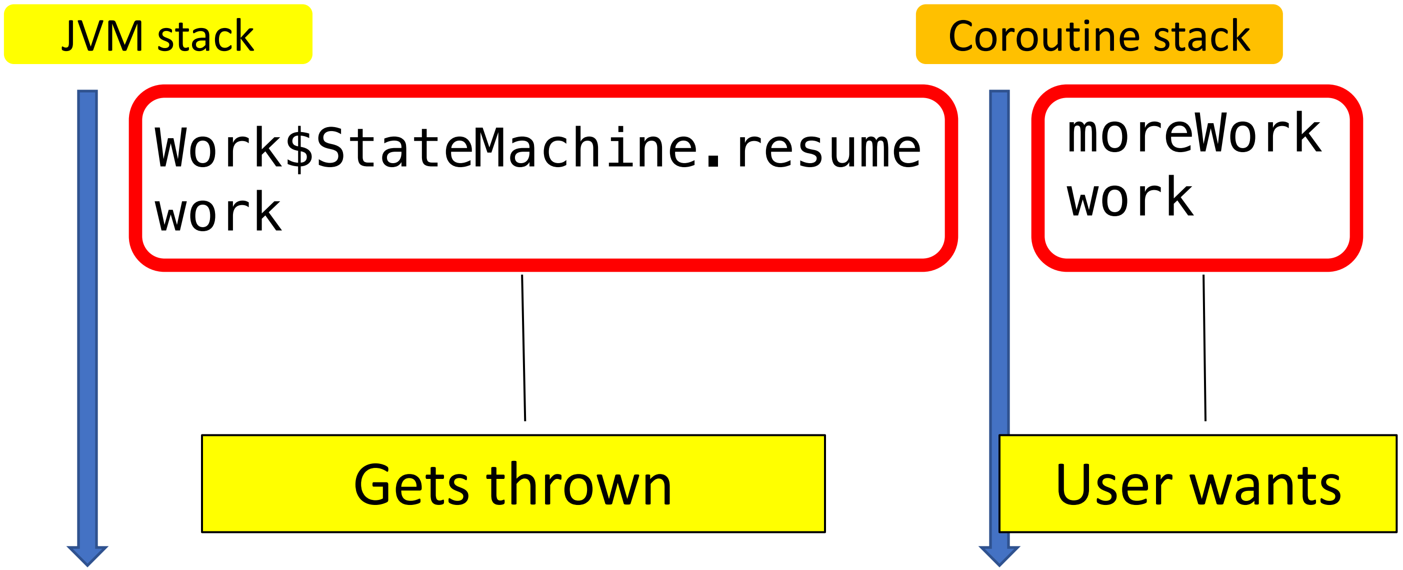
Stack traces in exceptions

```
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```



Stack traces in exceptions

```kotlin
suspend fun moreWork() {
    work()
}

suspend fun work() {
    someAsyncOp().await()
    throw Exception()
}
```



Library evolution

Going beyond the language & stdlib

Library evolution: already there

- Communication & synchronization primitives
  - **Job:** A completable & cancellable entity to represent a coroutine
  - **Deferred**: A future with **suspend** fun **await** (and *no* thread-blocking methods)
  - **Mutex**: with **suspend** fun **lock**
  - **Channel**: **SendChannel** & **ReceiveChannel**
    - **RendezvousChannel** – synchronous rendezvous
    - **ArrayChannel** – fixed size buffer
    - **LinkedListChannel** – unlimited buffer
    - **ConflatedChannel** – only the most recent sent element is kept
  - **BroadcastChannel**: multiple subscribes, all receive

Library evolution: already there

- Coroutine builders
  - **launch** (fire & forget, returns a **Job**)
  - **async** (returns **Deferred**)
  - **future** (integration with **CompletableFuture** & **ListenableFuture**)
  - **runBlocking** (to explicitly delimit code that blocks a thread)
  - **actor** / **produce** (consume & produce messages over channels)
  - **publish** (integration with reactive streams, returns **Publisher**)
  - **rxCompletable, rxSingle**, **rxObservable, rxFlowable** (RxJava 1/2)

Library evolution: already there

- Top-level functions
  - **select** expression to await multiple events for full CSP-style programming
  - **delay** for time-based logic in coroutines
- Extensions
  - **await** for all kinds of futures (JDK, Guava, RxJava)
  - **aRead**, **aWrite**, etc for **AsynchronousXxxChannel** in NIO
- Cancellation & job (coroutine/actor) hierarchies
  - **withTimeout** for a *composable* way for any suspend function run w/timeout

Library evolution: WIP

- Serialization / migration of coroutines
- Migrating libraries to Kotlin/JS & Kotlin/Native (lang support – done)
- Full scope of pipelining operators on channels (filter, map, etc)
- Optimizations for single-producer and/or single-consumer cases
- Allow 3rd party primitives to participate in **select** (alternatives)
- ByteChannel for suspendable IO (http client/server, websocket, etc)
  - See also ktor.io

A closing note on terminology

- We don’t use the term *fiber/strand/green threads/…*
- The term *coroutine* works just as well
  - ”Fibers describe essentially the same concept as coroutines” © Wikipedia

Kotlin Coroutines are ***very*** light-weight threads

Wrap up

Let’s call it a day

Experimental status of coroutines

- This design is *new* and unlike mainstream
  - For some very good reasons
- We want community to try it for real
  - So we released it as an ***experimental*** feature

opt-in flag

- We guarantee *backwards compatibility*
  - Old code compiled with coroutines continues to work
- We reserve the right to break *forward compatibility*
  - We may add things so new code may not run w/old RT
- Design will be finalized at a later point
  - Old code will continue to work via support library
  - Migration aids to the final design will be provided

Thank you

Any questions?

Slides are available at www.slideshare.net/elizarov

email **elizarov** at gmail

relizarov