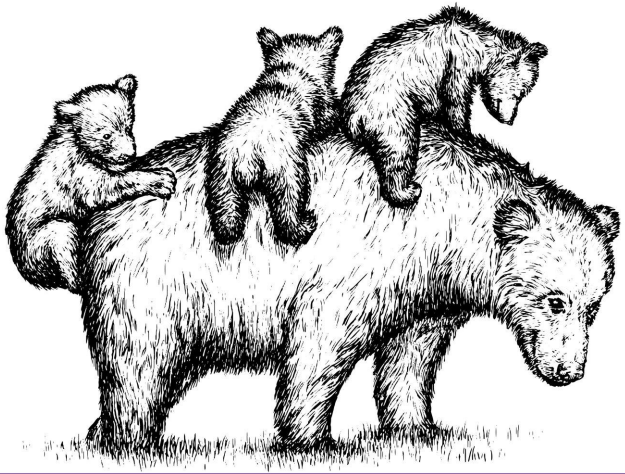
Kotlin Coroutines

Asynchronous Programming Made Simple

Problem

Getting the wrong idea from that conference talk you attended



Solving Imaginary Scaling Issues

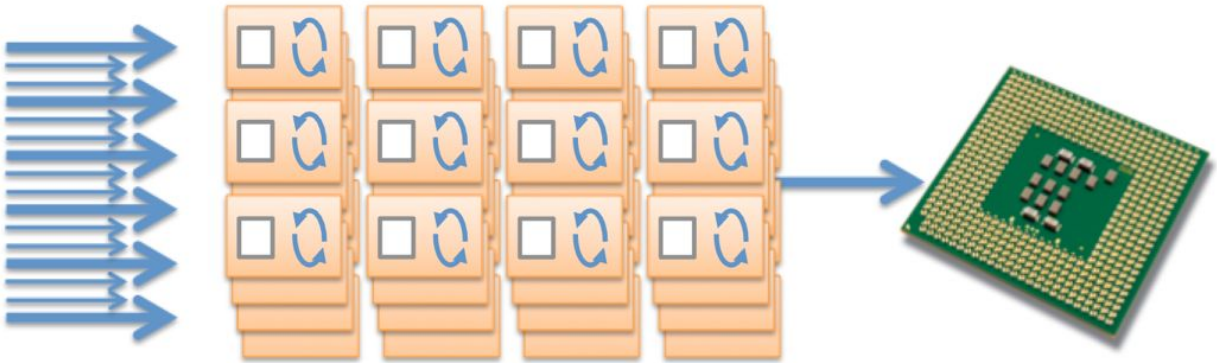
At Scale

O RLY?

@ThePracticalDev

Apache

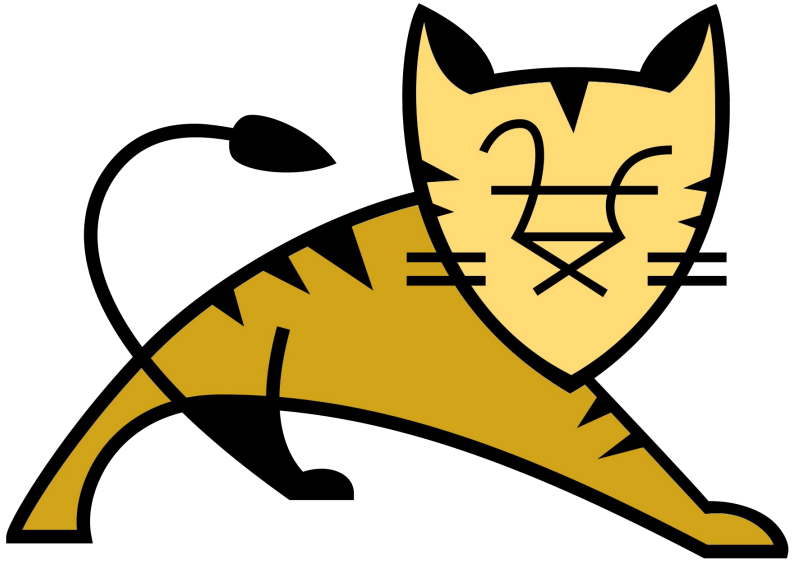
HTTP SERVER



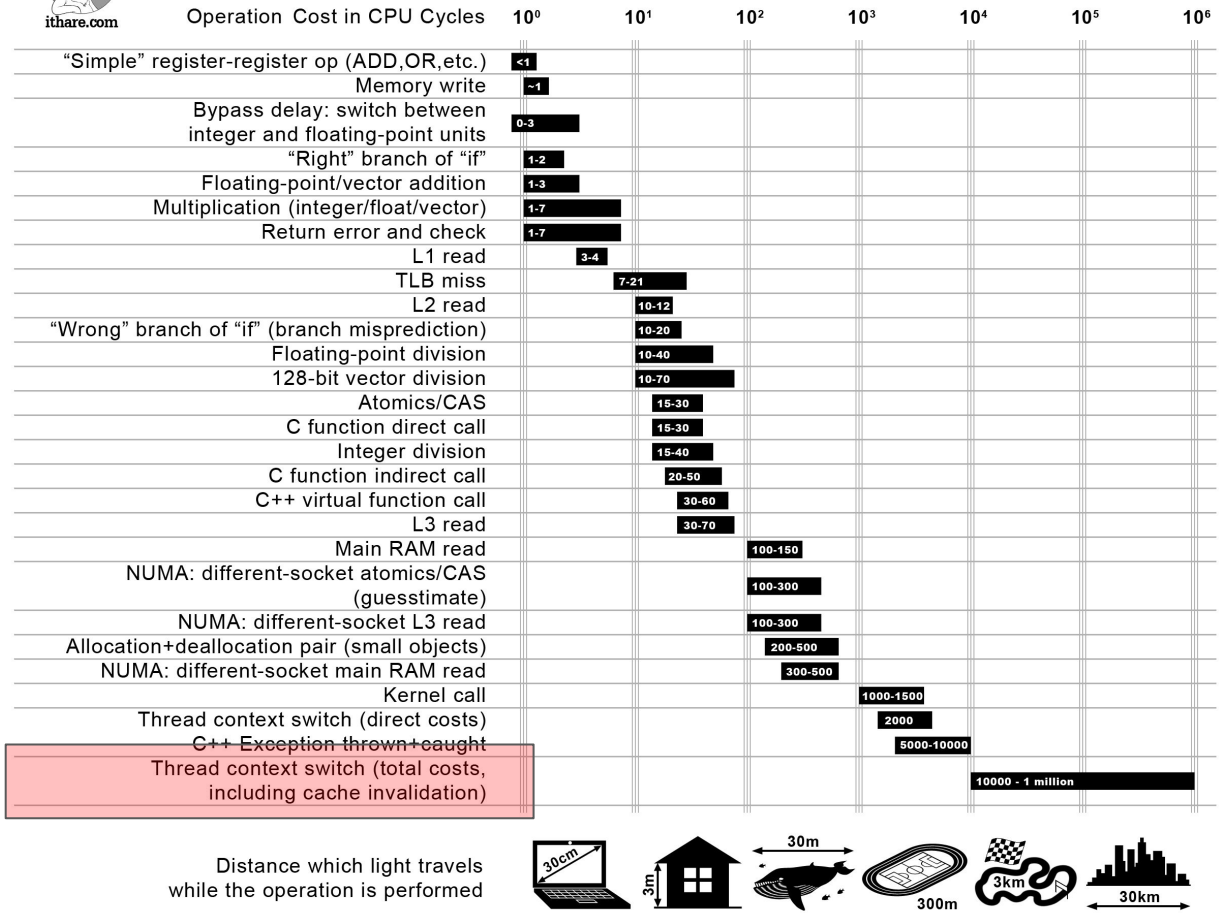
Hundreds of concurrent connections...

require hundreds of heavyweight threads or processes...

competing for limited CPU and memory



Not all CPU operations are created equal



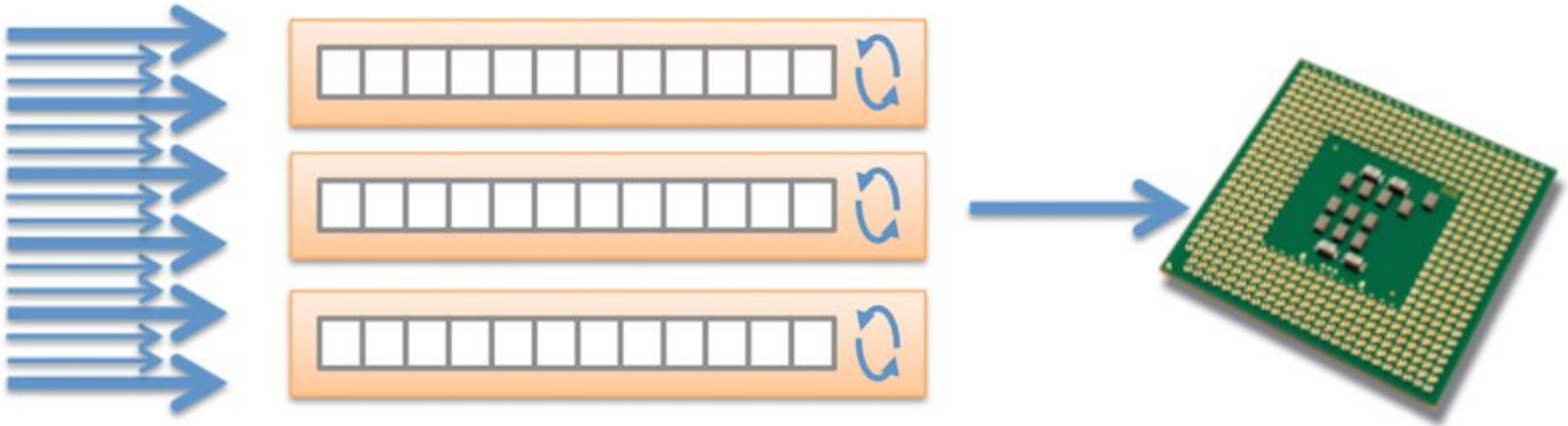
1M threads?

```
(1..1_000_000).forEach {
    thread(start = true) {
        println(it)
        Thread.sleep( millis: 1000L)
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:717)
    at kotlin.concurrent.ThreadsKt.thread(Thread.kt:30)
    at kotlin.concurrent.ThreadsKt.thread$default(Thread.kt:15)
    at by.heap.komodo.samples.coroutines.SuspendKt.main(Suspend.kt:40)
```

Intel Core i7-6700HQ, 32GB - **10k** Thread


NGINX



Hundreds of concurrent connections...

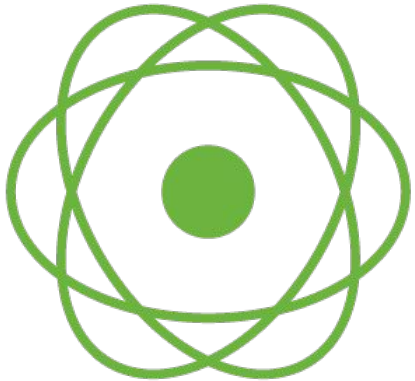
handled by a small number of multiplexing processes...

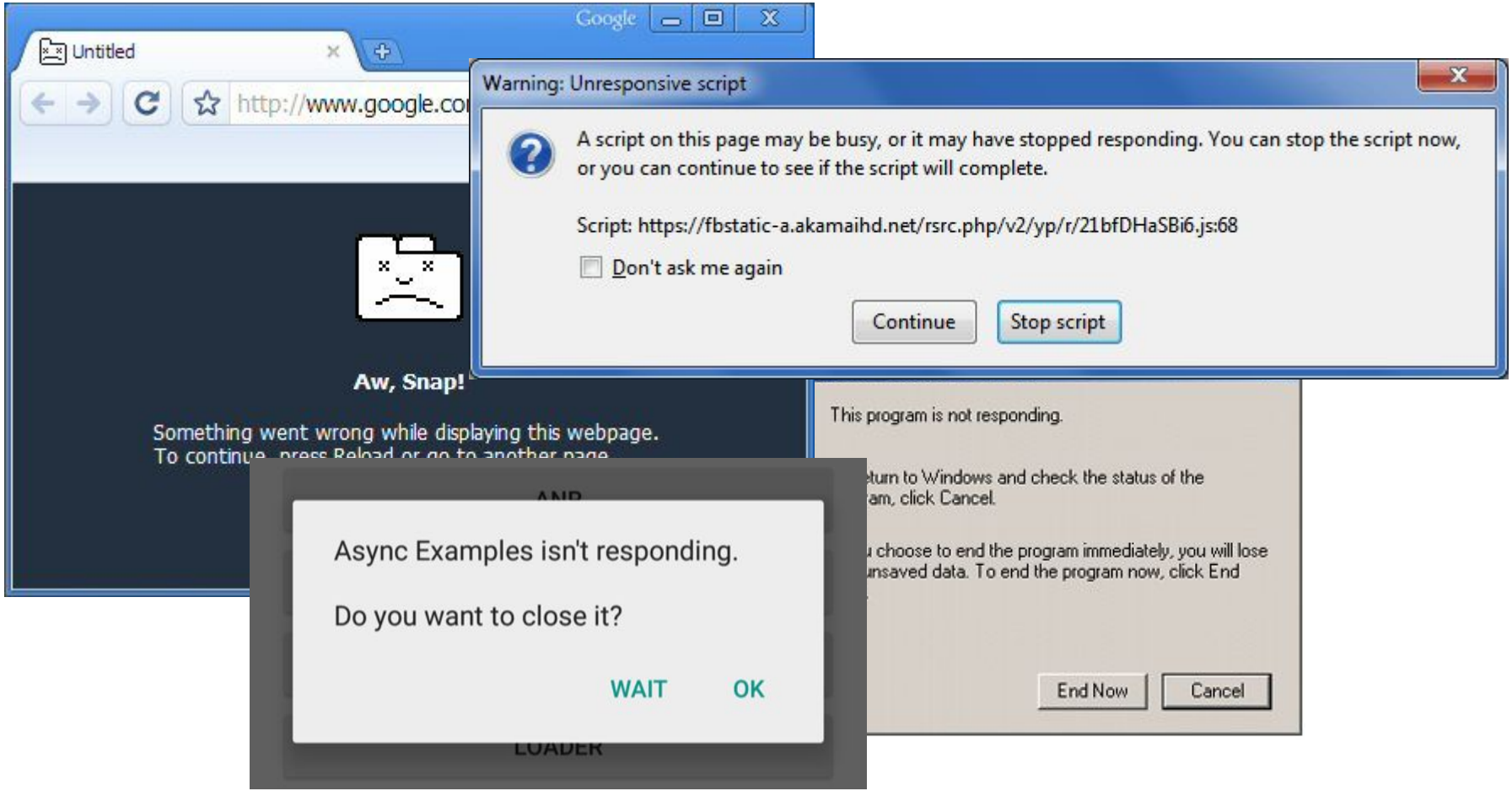
typically one process per core


VERT.X


undertow


Netty




Google
Untitled
http://www.google.co
Aw, Snap!
Something went wrong while displaying this webpage.
To continue, press Reload or go to another page.
Warning: Unresponsive script
A script on this page may be busy, or it may have stopped responding. You can stop the script now, or you can continue to see if the script will complete.
Script: https://fbstatic-a.akamaihd.net/rsrc.php/v2/yp/r/21bfDHaSBi6.js:68
Don't ask me again
Continue
Stop script
This program is not responding.
eturn to Windows and check the status of the am, click Cancel.
u choose to end the program immediately, you will lose unsaved data. To end the program now, click End
End Now
Cancel
Async Examples isn't responding.
Do you want to close it?
WAIT
OK

Android (UI Thread)

https://stackoverflow.com/questions/3652560/what-is-the-android-uithread-ui-thread#3653478:

The UIThread is the main thread of execution for your application. This is where **most of your application code** is run. All of your application components (Activities, Services, ContentProviders, BroadcastReceivers) are created in this thread, and any system calls to those components are performed in this thread.

Typical Backend

BlogService

```
fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

AuthenticationService

```
@Service
class AuthenticationService(
    private val httpClient: HttpClient
) {
    fun getUserId(token: String): String {
        val request = HttpGet(URI("http://authService:8080/"))
        return httpClient.execute(request).entity.content.reader().readText()
    }
}
```


This is Fine.

Callbacks

```
fun getUserId(token: String): String {
```

AuthenticationService

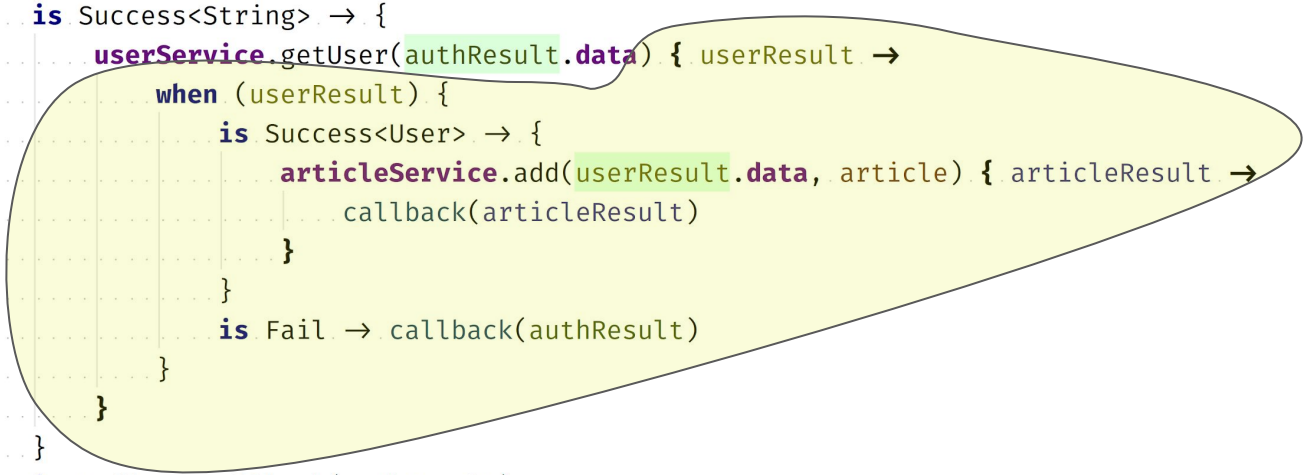
```
@Service
class AuthenticationService(
    private val httpClient: HttpAsyncClient
) {
    fun getUserId(token: String, callback: (Result) -> Unit) {
        val request = HttpGet(URI("https://auth:8080/"))
        httpClient.execute(request, object : FutureCallback<HttpResponse> {
            override fun completed(result: HttpResponse) {
                callback(Success(result.entity.content.reader().readText()))
            }

            override fun failed(ex: Exception) {
                callback(Fail(ex.message ?: "Error fetching auth data."))
            }

            override fun cancelled() {
                callback(Fail("Request canceled."))
            }
        })
    }
}
```

Callback

```
fun post(token: String, article: Article, callback: (Result<String>) → Unit) {
    authenticationService.getUserId(token) { authResult →
        when (authResult) {
            is Success<String> → {
                userService.getUser(authResult.data) { userResult →
                    when (userResult) {
                        is Success<User> → {
                            articleService.add(userResult.data, article) { articleResult →
                                callback(articleResult)
                            }
                        }
                        is Fail → callback(authResult)
                    }
                }
            }
            is Fail → callback(authResult)
        }
    }
}
```

Callback

```
fun post(token: String, article: Article, callback: (Result<String>) → Unit) {
    authenticationService.getUserId(token) { authResult →
        when (authResult) {
            is Success<String> → {
                userService.getUser(authResult.data) { userResult →
                    when (userResult) {
                        is Success<User> → {
                            articleService.add(userResult.data, article) { articleResult →
                                callback(articleResult)
                            }
                        }
                        is Fail → callback(authResult)
                    }
                }
            }
            is Fail → callback(authResult)
        }
    }
}
```



Callback

```
fun post(token: String, article: Article, callback: (Result<String>) → Unit) {
    authenticationService.getUserId(token) { authResult →
        when (authResult) {
            is Success<String> → {
                userService.getUser(authResult.data) { userResult →
                    when (userResult) {
                        is Success<User> → {
                            articleService.add(userResult.data, article) { articleResult →
                                callback(articleResult)
                            }
                        }
                        is Fail → callback(authResult)
                    }
                }
            }
            is Fail → callback(authResult)
        }
    }
}
```

Futures/Promises/Deferred

```
fun getUserId(token: String): String {
fun getUserId(token: String, callback: (Result) → Unit) {
```

AuthenticationService

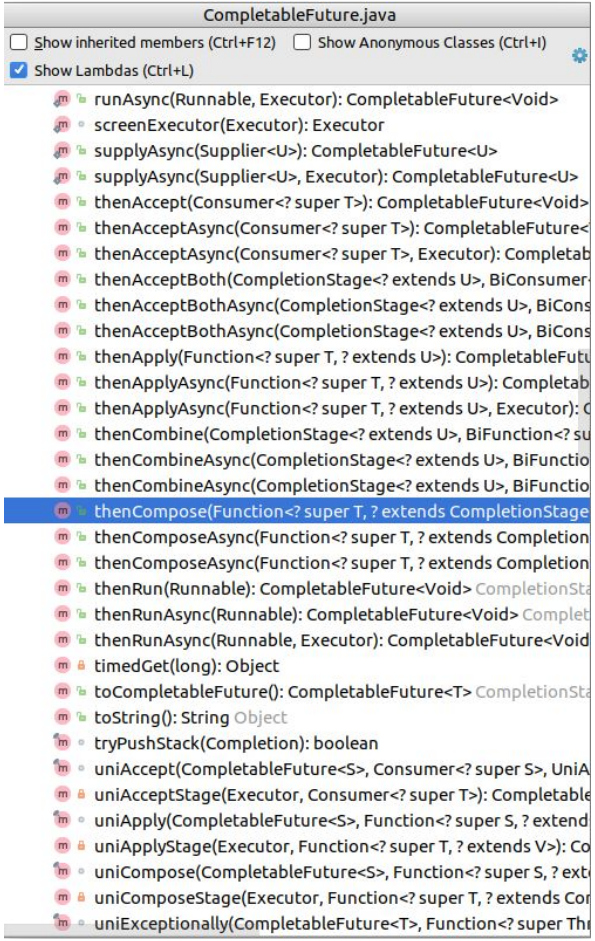
```
class AuthenticationService(
    private val httpClient: HttpAsyncClient
) {
    fun getUserId(token: String): CompletableFuture<String> {
        val request = HttpGet( uri: "https://auth:8080/")
        val future = CompletableFuture<HttpResponse>()

        httpClient.execute(request, object : FutureCallback<HttpResponse> {
            override fun completed(result: HttpResponse) {
                future.complete(result)
            }

            override fun cancelled() {
                future.cancel( mayInterruptIfRunning: false)
            }

            override fun failed(ex: Exception) {
                future.completeExceptionally(ex)
            }
        })

        return future.thenApply { it.entity.content.reader().readText() }
    }
}
```

Futures

```
fun post(token: String, article: Article): CompletableFuture<Result> {
    return authenticationService.getUserId(token)
        .thenCompose(userService::getUser)
        .thenCompose { user ->
            articleService.add(user, article)
        }
        .handle { u, e ->
            if (e != null) {
                Fail(e.message ?: "Can't create article.")
            } else {
                Success( data: "New article created.")
            }
        }
}
```

CompletableFuture

- compose;
- combine;
- handle;
- accept;
- apply;
- supply.



AWS Example

```
fun listS3Objects(name: String): List<ObjectListing> {
    val listings : MutableList<ObjectListing> = mutableListOf<ObjectListing>()

    var objects : ObjectListing! = s3.listObjects(name)
    listings.add(objects)
    do {
        objects = s3.listNextBatchOfObjects(objects)
        listings.add(objects)
    } while (objects.isTruncated)

    return listings
}
```

AWS Example

```
fun asyncListS3Objects(name: String): CompletableFuture<List<ObjectListing>> {
    return asyncS3.listObjects(name)
        .thenCompose { listing →
            if (listing.isTruncated) {
                fetchTruncated(listing, listOf(listing))
            } else {
                CompletableFuture.completedFuture(listOf(listing))
            }
        }
}

fun fetchTruncated(
    objectListing: ObjectListing,
    listings: List<ObjectListing>
): CompletableFuture<List<ObjectListing>> {
    return asyncS3.listNextBatchOfObjects(objectListing)
        .thenCompose { listing: ObjectListing →
            if (listing.isTruncated) {
                fetchTruncated(listing, listings: listings + listing)
            } else {
                CompletableFuture.completedFuture( value: listings + listing)
            }
        }
}
```

Coroutines

AWS Example

```
suspend fun coroutinesListS3Objects(name: String): List<ObjectListing> {
    val listings : MutableList<ObjectListing>  = mutableListOf<ObjectListing>()

    var objects : ObjectListing  = asyncS3.listObjects(name).await()
    listings.add(objects)

    do {
        objects = asyncS3.listNextBatchOfObjects(objects).await()
        listings.add(objects)
    } while (objects.isTruncated)

    return listings
}
```

BlogService

```
fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

Coroutines

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

Coroutines

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

0 callbacks!*

* explicit

Continuation

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

Continuation

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

Continuation

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

Continuation

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

```
suspend fun post(token: String,
               article: Article): Result {

fun post(token: String,
         article: Article,
         cont: Continuation<Result>): Result {

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

```
when(cont.label) {
    0 → {
        cont.label = 1
        authenticationService.getUserId(token)
    }
    1 → {
        cont.label = 2
        userService.getUser(userId)
    }
    2 → {
        cont.label = 3
        articleService.add(user, article)
    }
}
```

Coroutines

```
suspend fun post(token: String, article: Article): Result {
    return try {
        val userId = authenticationService.getUserId(token)
        val user = userService.getUser(userId)
        articleService.add(user, article)
        Success( data: "New article created.")
    } catch (e: Exception) {
        LOGGER.log(Level.SEVERE, msg: "Can't create article.", e)
        Fail(e.message ?: "Can't create article.")
    }
}
```

AuthenticationService

```
fun getUserId(token: String): String {
fun getUserId(token: String, callback: (Result) → Unit) {
fun getUserId(token: String): CompletableFuture<String> {
```

```
class AuthenticationService(
    private val httpClient: HttpAsyncClient
) {
    suspend fun getUserId(token: String): String {
        val request = HttpGet( uri: "https://auth:8080/")

        return httpClient.execute(request).entity.content.reader().readText()
    }
}
```

HttpAsyncClient (Callback API) -> Coroutines

```
suspend fun HttpAsyncClient.execute(request: HttpUriRequest): HttpResponse {
    return suspendCancellableCoroutine { cont: CancellableContinuation<HttpResponse> →
        val future :Future<HttpResponse!>! = this.execute(request, object : FutureCallback<HttpResponse> {
            override fun completed(result: HttpResponse) {
                cont.resume(result)
            }

            override fun cancelled() {
                // Nothing
            }

            override fun failed(ex: Exception) {
                cont.resumeWithException(ex)
            }
        })

        cont.cancelFutureOnCompletion(future);
        Unit
    }
}
```

CompletableFuture (Future API) -> Coroutines

```
suspend fun <T> CompletableFuture<T>.await(): T {
    return suspendCancellableCoroutine { cont: CancellableContinuation<T> →
        this.whenComplete { t, u →
            if (u == null) {
                cont.resume(t)
            } else {
                cont.resumeWithException(u)
            }
        }
    }
}

CompletableFuture.supplyAsync {
    // compute on common pool
}.await()
```

Coroutines: Hello, World!

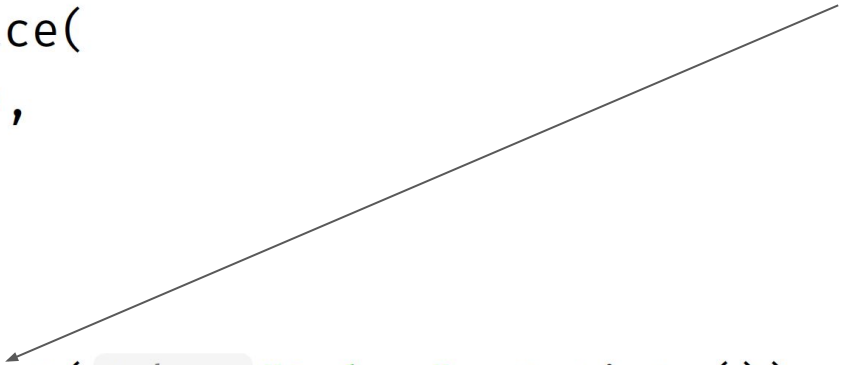
```
compile("org.jetbrains.kotlinx:kotlinx-coroutines-core:0.19.1")
```

```
fun main(args: Array<String>) {
    delay( time: 1000)
    println("Hello, World!")
}
```

Error:(56, 5) Kotlin: Suspend function **'delay'** should be called only from a **coroutine** or another **suspend function**

Coroutines: Hello, World!

```
fun main(args: Array<String>) {
    val blogService = BlogService(
        AuthenticationService(),
        UserService(),
        ArticleService()
    )
    val result = blogService.post( token: "token", Article())

    // work with result
}
```



Coroutines: Hello, World!

```
fun main(args: Array<String>)  {
    launch(CommonPool) {
        delay(1000)
        println("Hello, World!")
    }
}

// Nothing
```

Coroutines: Hello, World!

```
fun main(args: Array<String>)  {
    launch(CommonPool) {
        delay(1000)
        println("Hello, World!")
    }

    Thread.sleep(2000)
}

// Hello, World!
```

Coroutines: Hello, World!

```
fun main(args: Array<String>)  {
    runBlocking {
        delay(1000)
        print("Hello, ")
    }

    print("World!")
}

// Hello, World!
```

Coroutines: Hello, World!

```
fun main(args: Array<String>) = runBlocking {
    val blogService = BlogService(
        AuthenticationService(),
        UserService(),
        ArticleService()
    )
    val result = blogService.post( token: "token", Article())

    // work with result
}
```

Coroutines + Spring

```
@RestController
class BlogController(
    val blogService: BlogService
) {

    @PostMapping("/{token}")
    fun createArticle(@PathVariable token: String, article: Article): CompletableFuture<String> {
        return future {
            val result = blogService.post(token, article)

            when (result) {
                is Success → result.data
                is Fail → throw RuntimeException(result.message)
            }
        }
    }
}
```

Recap

Coroutines are:

- Way to pass callbacks(continuations), but write code that looks synchronous
- Can be considered as **lightweight threads**

Regular world -> Coroutines: **launch**, **runBlocking**, **future**, **async**, etc

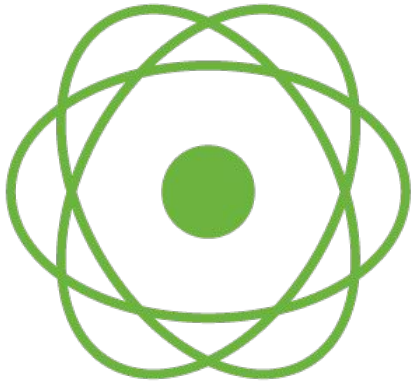
Coroutines -> Coroutines: **suspend** keyword

Async Api -> Coroutines: **suspendCoroutine**, **suspendCancellableCoroutine**

ktor

Anko

VERT.X



Q&A

Ruslan Ibragimov @HeapyHop

Belarus Kotlin User Group: https://bkug.by/

Java Professionals BY: http://jprof.by/

Awesome Kotlin: https://kotlin.link/

Slides: https://goo.gl/Ww2AsM

Learn Kotlin Coroutines

- Guide to kotlinx.coroutines by example
- Coroutines for Kotlin
- #coroutines Kotlin Slack
- Андрей Бреслав — Асинхронно, но понятно. Сопрограммы в Kotlin
- Andrey Breslav — Kotlin Coroutines (JVMLS 2016, old coroutines!)
- Корутины в Kotlin - Роман Елизаров, JetBrains

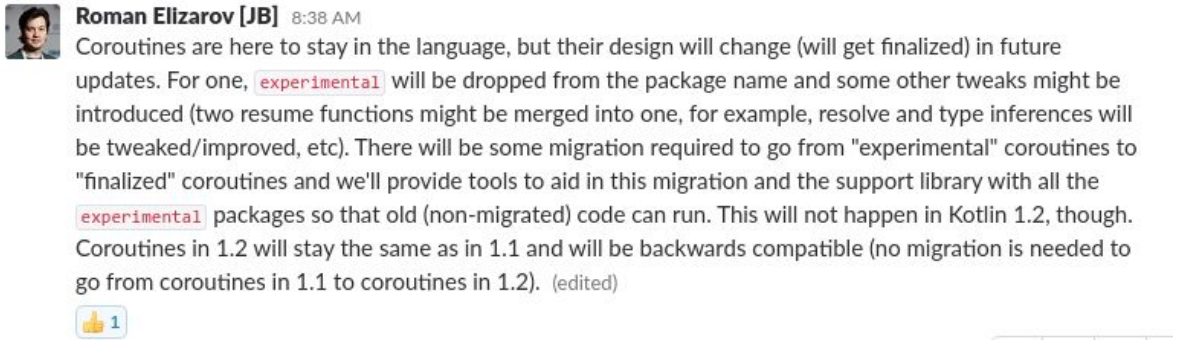
Kotlin Coroutines

Kotlin 1.1: Experimental

Kotlin 1.2: Experimental

```
// build.gradle
kotlin {
    experimental {
        coroutines 'enable'
    }
}
```

kotlin.coroutines.experimental -> kotlin.coroutines



Roman Elizarov [JB] 8:38 AM

Coroutines are here to stay in the language, but their design will change (will get finalized) in future updates. For one, `experimental` will be dropped from the package name and some other tweaks might be introduced (two resume functions might be merged into one, for example, resolve and type inferences will be tweaked/improved, etc). There will be some migration required to go from "experimental" coroutines to "finalized" coroutines and we'll provide tools to aid in this migration and the support library with all the `experimental` packages so that old (non-migrated) code can run. This will not happen in Kotlin 1.2, though. Coroutines in 1.2 will stay the same as in 1.1 and will be backwards compatible (no migration is needed to go from coroutines in 1.1 to coroutines in 1.2). (edited)

1

Suspend functions
Coroutines builders
kotlinx.coroutines

Coroutines & Kotlin

- suspend – language
- low-level core API: coroutine builders, etc – kotlin.coroutines (kotlin-stdlib)
- libraries – example: kotlinx.coroutines (kotlinx-coroutines-core)

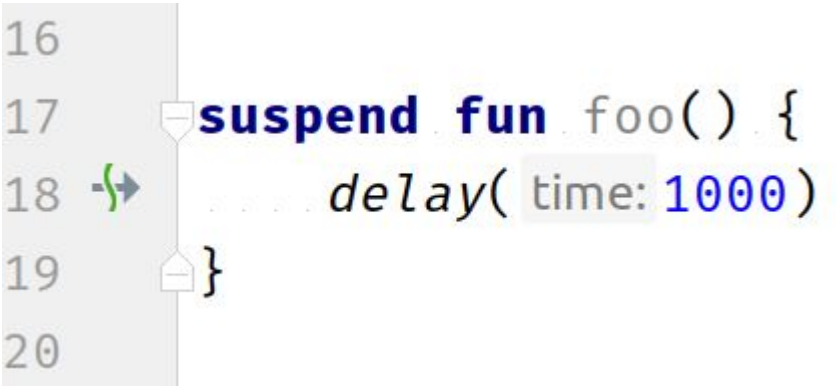
Suspend

Suspending Functions

```
suspend fun delay(
    time: Long,
    unit: TimeUnit = TimeUnit.MILLISECONDS
) {
    // ...
}
```

Suspending Functions

```
suspend fun foo() {
    delay(1000)
}
```



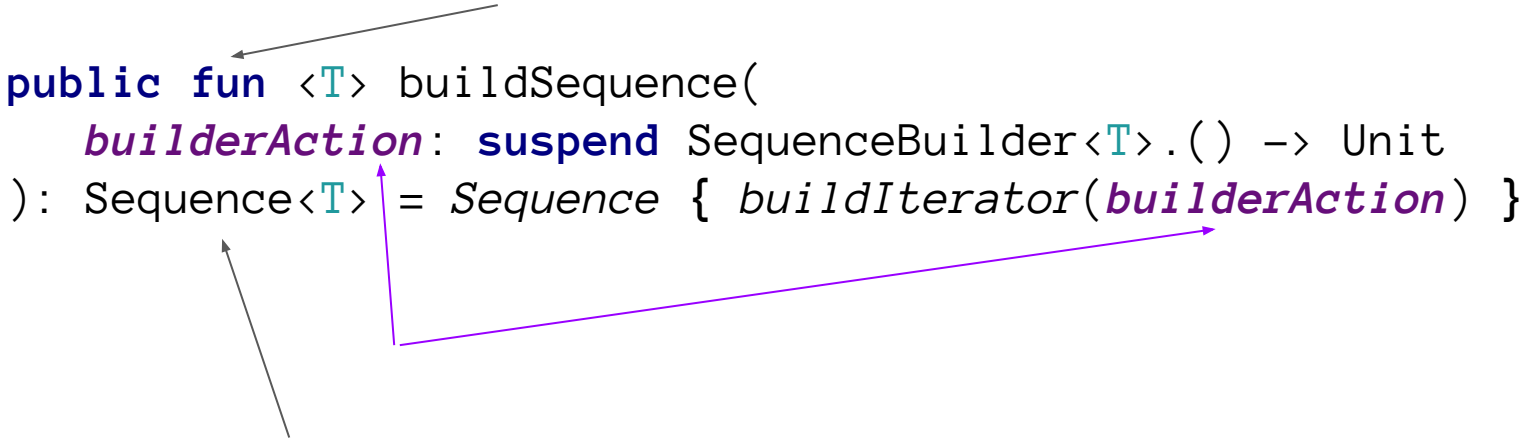
Suspending Lambda

```
public fun launch(
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    …
}
```

Generators

buildSequence

kotlin.coroutines.experimental.buildSequence



```
public fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> = Sequence { buildIterator(builderAction) }
```

kotlin.coroutines.experimental.buildSequence

```
val lazySeq: Sequence<Int> = buildSequence {
    for (i in 1..100) {
        yield(i)
    }
}

lazySeq.take(3).forEach { print(it) }
// 123
```

kotlin.coroutines.experimental.buildSequence

```
val lazySeq: Sequence<Int> = buildSequence {
    for (i in 1..100) {
        delay(1000)
        yield(i)
    }
}
```

Error:(22, 9) Kotlin: Restricted suspending functions can only invoke member or extension suspending functions on their restricted coroutine scope

```
public fun <T> buildSequence(
    builderAction: suspend SequenceBuilder<T>.() -> Unit
): Sequence<T> = Sequence { buildIterator(builderAction) }
```

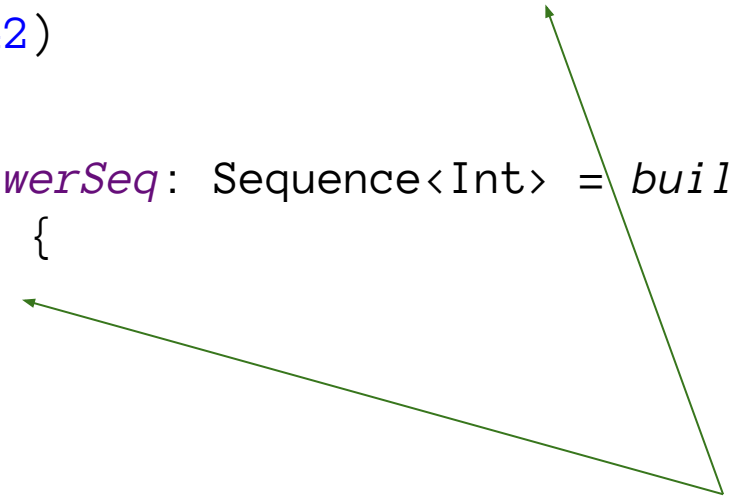
kotlin.coroutines.experimental.

SequenceBuilder

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> internal constructor() {
    public abstract suspend fun yield(value: T)
    public abstract suspend fun yieldAll(iterator: Iterator<T>)
    ...
}
```

kotlin.coroutines.experimental.SequenceBuilder

```
suspend fun SequenceBuilder<Int>.answer() {
    this.yield(42)
}

val ultimateAnswerSeq: Sequence<Int> = buildSequence {
    while (true) {
        answer()
    }
}
```



buildIterator

Iterator<T>

buildSequence
buildIterator
@RestrictsSuspension

kotlinx.coroutines

kotlin.coroutines.experimental.launch

```
public fun launch(
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() → Unit
): Job {
```

CoroutineContext

- Unconfined
- CommonPool
- newSingleThreadContext, newFixedThreadPoolContext
- Executor.asCoroutineDispatcher

CoroutineStart

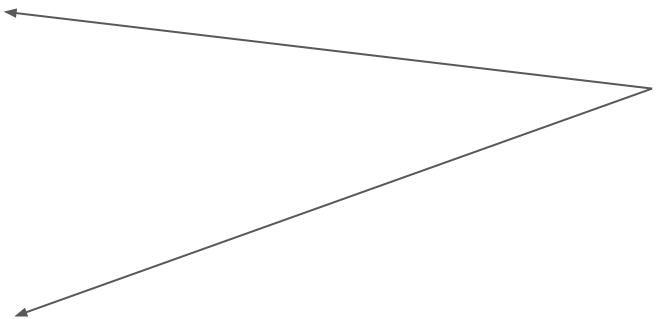
```
CoroutineStart.DEFAULT → block.startCoroutineCancellable(completion)
CoroutineStart.ATOMIC → block.startCoroutine(completion)
CoroutineStart.UNDISPATCHED → block.startCoroutineUndispatched(completion)
CoroutineStart.LAZY → Unit // will start lazily
```

CoroutineScope

```
public interface CoroutineScope {
    public val isActive: Boolean
    public val context: CoroutineContext
}
```

kotlin.coroutines.experimental.launch

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val jobs = List( size: 100_000) {
        launch(CommonPool) {
            delay( time: 1000L)
            print(".")
        }
    }
    jobs.forEach { it.join() }
}
```



kotlinx.coroutines.experimental.launch

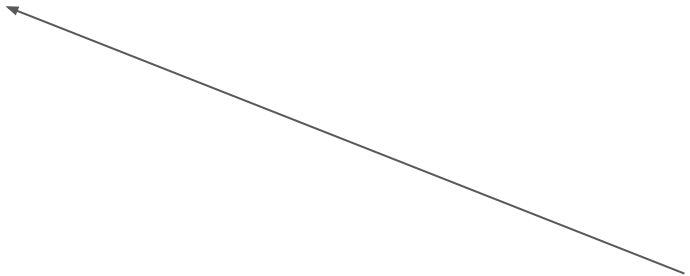
```
val job = launch(CommonPool) {
    while (isActive) {
        delay(100)
        println(42)
    }
}
job.cancel()
```

kotlinx.coroutines.experimental.NonCancellable

```
val job = launch(CommonPool) {
    try {
        // ...
    } finally {
        run(NonCancellable) {
            // this code isn't cancelled
        }
    }
}
job.cancel()
```

kotlin.coroutines.experimental.**async**

```
public fun <T> async(
    context: CoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() → T
): Deferred<T> {
```

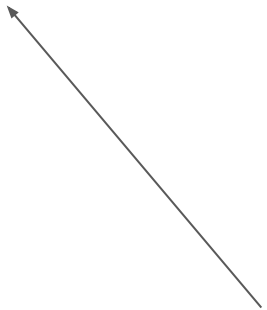


kotlin.coroutines.experimental.async

```
fun main(args: Array<String>) {
    val blogService = BlogService(
        AuthenticationService(),
        UserService(),
        ArticleService()
    )

    runBlocking {
        val result1 = async(Unconfined) { blogService.post( token: "token1", Article()) }
        val result2 = async(Unconfined) { blogService.post( token: "token2", Article()) }

        println("Result: ${result1.await()}, ${result2.await()}")
    }
}
```

kotlin.coroutines.experimental.yield

```
suspend fun foo() {
    list.forEach {
        /*
         ...
        compute relatively heavy task
         */
        yield()
    }
}
```



kotlinx.coroutines.experimental.(withTimeout/withTimeoutOrNull)

```
withTimeout(100) {
    request.await()
}

withTimeoutOrNull(100) {
    request.await()
}
```

Recursive Coroutines

```
suspend fun test() {
    println(Instant.now())
    test()
}

tailrec suspend fun test() {
    println(Instant.now())
    test()
}
```

Debug

-Dkotlinx.coroutines.debug

Thread.currentThread().name

[main @coroutine#2]
[main @coroutine#3]
[main @coroutine#1]

```
public fun newCoroutineContext(context: CoroutineContext):
CoroutineContext = if (DEBUG) context +
CoroutineId(COROUTINE_ID.incrementAndGet()) else context
```

Call Coroutines from Java

```
suspend fun foo(): Int {
    //...
}

fun fooJava(): CompletableFuture<Int> =
    future { foo() }
```

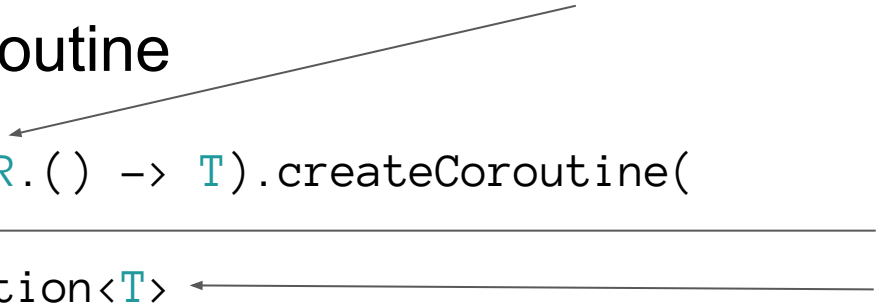
Core API

createCoroutine
startCoroutine
suspendCoroutine
suspendCancellableCoroutine

Core API

createCoroutine

kotlin.coroutines.experimental.createCoroutine



```
public fun <R, T> (suspend R.() -> T).createCoroutine(
        receiver: R,
        completion: Continuation<T>
): Continuation<Unit> = SafeContinuation(
        createCoroutineUnchecked(receiver, completion),
        COROUTINE_SUSPENDED
)

block.createCoroutine(receiver, completion)
```

```
launch(CommonPool) {
    delay(1000)
    println("Hello, World!")
}
```

Bytecode

```
package by.heap.komodo.samples.coroutines.bytecode

import kotlinx.coroutines.experimental.delay

suspend fun fetch() {
    delay(1000)
}
```

Bytecode

```
-rw-r--r-- 1 yoda yoda 1342 Jun  1 08:03 ExampleKt.class
-rw-r--r-- 1 yoda yoda 1833 Jun  1 08:03 ExampleKt$fetch$1.class
```

Bytecode

```
public final class ExampleKt {
  public static final Object fetch(
    Continuation<? super Unit>
  );
}
```

Bytecode

```
public final class ExampleKt {
    @Nullable
    public static final Object fetch(@NotNull final Continuation<? super
Unit> $continuation) {
    Intrinsics.checkParameterIsNotNull((Object)$continuation,
"$continuation");
    return new
ExampleKt$fetch.ExampleKt$fetch$1((Continuation)$continuation).doResume((Ob
ject)Unit.INSTANCE, (Throwable)null);
    }
}
```

Bytecode

```
final class ExampleKt$fetch$1 extends CoroutineImpl {
  public final Object doResume(Object, Throwable);
  ExampleKt$fetch$1(Continuation);
}
```

Bytecode

```
static final class ExampleKt$fetch$1 extends CoroutineImpl {
      @Nullable
      public final Object doResume(@Nullable final Object data, @Nullable final Throwable throwable) {
      final Object coroutine_SUSPENDED = IntrinsicsKt.getCOROUTINE_SUSPENDED();
      switch (super.label) {
            case 0: {
            ...
            break;
            }
            case 1: {
                  ...
                  break;
            }
            default: {
                  throw new IllegalStateException("call to 'resume' before 'invoke' with coroutine");
            }
      }
      return Unit.INSTANCE;
      }
}
```

startCoroutine

kotlin.coroutines.experimental.**startCoroutine**

```
public fun <R, T> (suspend R.() -> T).startCoroutine(
        receiver: R,
        completion: Continuation<T>
) {
    createCoroutineUnchecked(receiver, completion).resume(Unit)
}
```

```
block.startCoroutine(receiver, completion)
```

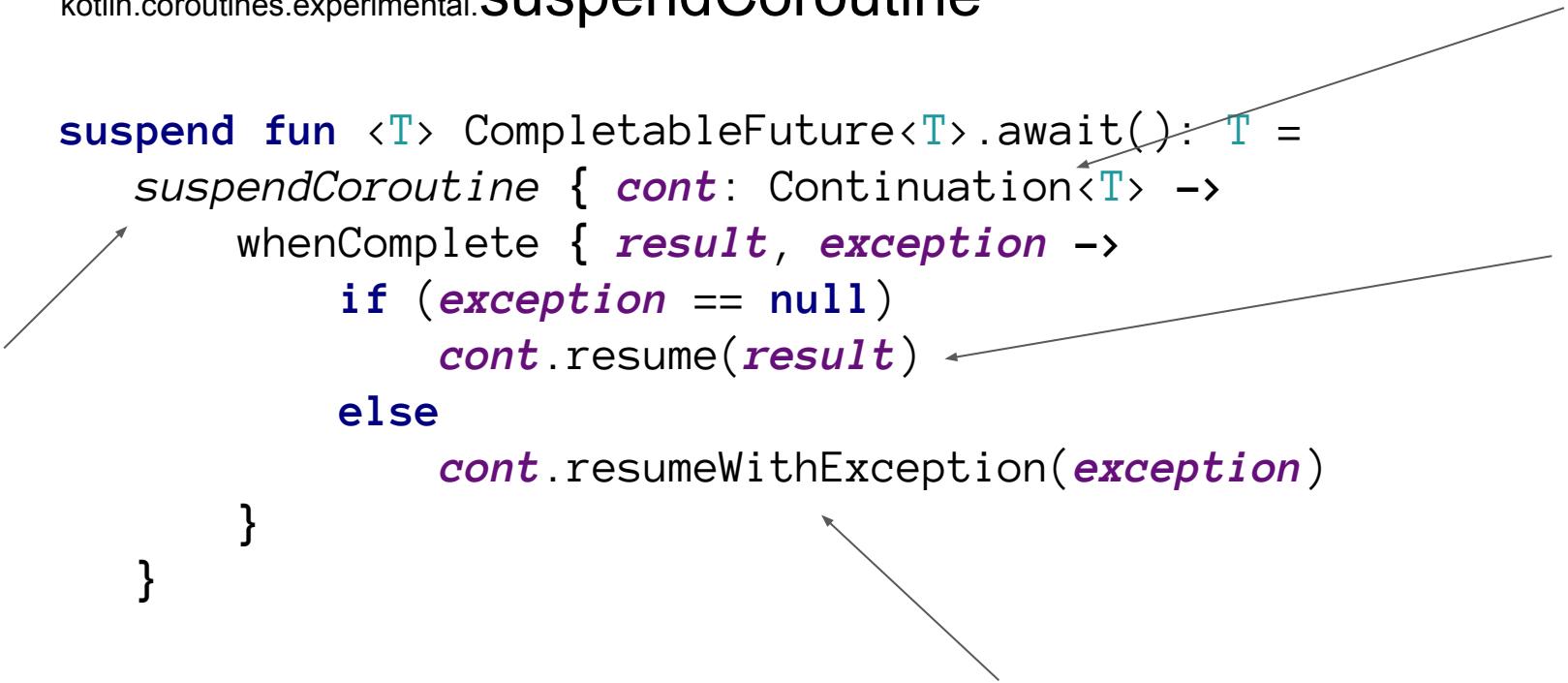
suspendCoroutine

kotlin.coroutines.experimental.**suspendCoroutine**

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

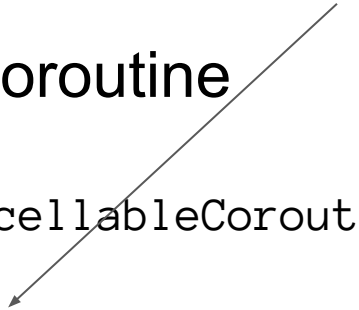
kotlin.coroutines.experimental.**suspendCoroutine**

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



suspendCancellableCoroutine

kotlin.coroutines.experimental.suspendCancellableCoroutine



```
public inline suspend fun <T> suspendCancellableCoroutine(
    holdCancellability: Boolean = false,
    crossinline block: (CancellableContinuation<T>) -> Unit
): T = suspendCoroutineOrReturn { cont ->
    val cancellable = CancellableContinuationImpl(cont, active = true)
    if (!holdCancellability) cancellable.initCancellability()
    block(cancellable)
    cancellable.getResult()
}
```

kotlin.coroutines.experimental.suspendCancellableCoroutine

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCancellableCoroutine { cont: CancellableContinuation<T> ->
        whenComplete { result, exception ->
            if (exception == null)
                cont.resume(result)
            else
                cont.resumeWithException(exception)
        }
        cont.invokeOnCompletion { this.cancel(false) }
    }
```

suspend
createCoroutine
startCoroutine
suspendCoroutine
suspendCancellableCoroutine

Not Covered

- Channels
- Select
- ...

Shared mutable state and concurrency

- Thread-safe data structures (Atomics)
- Thread confinement fine-grained
- Thread confinement coarse-grained
- Mutual exclusion (suspending)
- Actors
- Read more