KOTLIN ELEMENTARY

1 2 3 4 5 6

Continuation & CPS

factorial

```
fun fact(n:Int) = _fact(n,1)
tailrec fun _fact(n:Int, a:Int):Int = if(n == 0) a else _fact(n - 1,n*a)
```

factorial

```
fun fact(n:Int) = _fact(n,1)
tailrec fun _fact(n:Int, a:Int):Int = if(n == 0) a else _fact(n - 1,n*a)
```

```
println(fact(3))
```

factorial CPS

```
fun factCPS(n:Int, block:(Int)->Unit) = _factCPS(n,1, block)
tailrec fun _factCPS(n:Int, a:Int, block:(Int)->Unit){
    if(n == 0) block(a) else _factCPS(n - 1,n*a, block)
}
```

factorial CPS

```
fun factCPS(n:Int, block:(Int)->Unit) = _factCPS(n,1, block)
tailrec fun _factCPS(n:Int, a:Int, block:(Int)->Unit){
    if(n == 0) block(a) else _factCPS(n - 1,n*a, block)
}
```

```
factCPS(3, ::println)
```

factorial Throw

```
fun factThrow(n:Int) = _factThrow(n,1)
fun _factThrow(n:Int, a:Int):Int = when {
    n < 0 -> throw Throwable("invalid value: $n")
    n == 0 -> a
    else -> _fact(n - 1, n * a)
}
```

factorial Throw

```
fun factThrow(n:Int) = _factThrow(n,1)
fun _factThrow(n:Int, a:Int):Int = when {
    n < 0 -> throw Throwable("invalid value: $n")
    n == 0 -> a
    else -> _fact(n - 1, n * a)
}
```

```
try{
    println(factThrow(-3))
}catch (e:Throwable){
    println(e)
}
```

factorial CPS Exception

```
fun factCPSEx(n:Int, block:(Int)->Unit, ex:(String)->Unit = {}) = _factCPSEx(n,1, block, ex)
tailrec fun _factCPSEx(n:Int, a:Int, block:(Int)->Unit, ex:(String)->Unit) {
    when {
        n < 0 -> ex("invalid value: $n")
        n == 0 -> block(a)
        else -> _factCPSEx(n - 1, n * a, block, ex)
    }
}
```

factorial CPS Exception

```
fun factCPSEx(n:Int, block:(Int)->Unit, ex:(String)->Unit = {}) = _factCPSEx(n,1, block, ex)
tailrec fun _factCPSEx(n:Int, a:Int, block:(Int)->Unit, ex:(String)->Unit) {
    when {
        n < 0 -> ex("invalid value: $n")
        n == 0 -> block(a)
        else -> _factCPSEx(n - 1, n * a, block, ex)
    }
}
```

```
factCPSEx(-3, ::println, ::println)
```

Continuation & Sequence

Continuation

```
class Cont<T>{
    var state = 0
    var isCompleted = false
    var result:T? = null
    fun resume(v:T){
        state++
        result = v
    }
    fun complete(v:T){
        isCompleted = true
        result = v
    }
}
```

Continuation

```
class Cont<T>{
    var state = 0
    var isCompleted = false
    var result:T? = null
    fun resume(v:T){
        state++
        result = v
    }
    fun complete(v:T){
        isCompleted = true
        result = v
    }
}
```

```
fun continuation1(a:Int, cont:Cont<Int>? = null) = run{
    var v:Int
    val c = if(cont == null){
        v = a
        Cont()
    }else{
        v = cont.result!!
        cont
    }
```

Continuation

```
class Cont<T>{
    var state = 0
    var isCompleted = false
    var result:T? = null
    fun resume(v:T){
        state++
        result = v
    }
    fun complete(v:T){
        isCompleted = true
        result = v
    }
}
```

```
fun continuation1(a:Int, cont:Cont<Int>? = null) = run{
    var v:Int
    val c = if(cont == null){
        v = a
        Cont()
    }else{
        v = cont.result!!
        cont
    }
    when(c.state){
        0->{
            v++
            println("state $v")
            c.resume(v)
        }
        1->{
            v++
            println("state $v")
            c.resume(v)
        }
        else->{
            v++
            println("state $v")
            c.complete(v)
        }
    }
    c
}
```

Continuation

```
class Cont<T>{
    var state = 0
    var isCompleted = false
    var result:T? = null
    fun resume(v:T){
        state++
        result = v
    }
    fun complete(v:T){
        isCompleted = true
        result = v
    }
}
```

```
var cont = continuation1(3)
while(!cont.isCompleted){
    cont = continuation1(3, cont)
}
println(cont.result)
```

```
fun continuation1(a:Int, cont:Cont<Int>? = null) = run{
    var v:Int
    val c = if(cont == null){
        v = a
        Cont()
    }else{
        v = cont.result!!
        cont
    }
    when(c.state){
        0->{
            v++
            println("state $v")
            c.resume(v)
        }
        1->{
            v++
            println("state $v")
            c.resume(v)
        }
        else->{
            v++
            println("state $v")
            c.complete(v)
        }
    }
    c
}
```

Sequence to Iteration

```
fun continuation1(a:Int, cont:Cont<Int>? = null) = run{
    var v:Int
    val c = if(cont == null){
        v = a
        Cont()
    }else{
        v = cont.result!!
        cont
    }
    when(c.state){
        0->{
            v++
            println("state $v")
            c.resume(v)
        }
        1->{
            v++
            println("state $v")
            c.resume(v)
        }
        else->{
            v++
            println("state $v")
            c.complete(v)
        }
    }
    c
}
```

Sequence to Iteration

```
val s = sequence {
    var v = 3
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
}
println(s.last())
```

```
fun continuation1(a:Int, cont:Cont<Int>? = null) = run{
    var v:Int
    val c = if(cont == null){
        v = a
        Cont()
    }else{
        v = cont.result!!
        cont
    }
    when(c.state){
        0->{
            v++
            println("state $v")
            c.resume(v)
        }
        1->{
            v++
            println("state $v")
            c.resume(v)
        }
        else->{
            v++
            println("state $v")
            c.complete(v)
        }
    }
    c
}
```

Sequence to Iteration

```
val s = sequence {
    var v = 3
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
}
println(s.last())
```

```
override suspend fun yield(value: T) {
    nextValue = value
    state = State_Ready
    return suspendCoroutineUninterceptedOrReturn { c ->
        nextStep = c
        COROUTINE_SUSPENDED
    }
}
```

```
                println("state $v")
                c.resume(v)
            }
            else->{
                v++
                println("state $v")
                c.complete(v)
            }
        }
        c
    }
```

Sequence to Iteration

```
val s = sequence {
    var v = 3
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
}
println(s.last())
```

```
override suspend fun yield(value: T) {
    nextValue = value
    state = State_Ready
    return suspendCoroutineUninterceptedOrReturn { c ->
        nextStep = c
        COROUTINE_SUSPENDED
    }
}
```

```
                println("state $v")
                c.resume(v)
            }
            else->{
                v++
                println("state $v")
                c.complete(v)
            }
        }
        c
    }
```

Sequence to Iteration

```
val s = sequence {
    var v = 3
    v++
    println("state $v")
    yield(v)
    v++
    println("state $v")
    yield(v)
    v++
    yield(v)
}
println(s.last())
```

```
override suspend fun yield(value: T) {
    nextValue = value
    state = State_Ready
    return suspendCoroutineUninterceptedOrReturn { c ->
        nextStep = c
        COROUTINE_SUSPENDED
    }
}
```

```
suspend fun <T> suspendCoroutineUninterceptedOrReturn(block: (Continuation<T>) -> Any?): T
```

```
                                                println("state $v")
                                            }
                                            else->{
                                                v++
                                                println("state $v")
                                                c.complete(v)
                                            }
                                        }
                                        c
                                    }
```

CO

```
class State{
    var result = ""
    lateinit var target:Promise<Response>
}
```

CO

```
class State{
    var result = ""
    lateinit var target:Promise<Response>
}
sequence{
    val s = State()
    s.target = window.fetch(Request("a.txt"))
    yield(s)
    s.target = window.fetch(Request(s.result))
    yield(s)
    println(s.result)
}
```

CO

```
class State{
    var result = ""
    lateinit var target:Promise<Response>
}
sequence{
    val s = State()
    s.target = window.fetch(Request("a.txt"))
    yield(s)
    s.target = window.fetch(Request(s.result))
    yield(s)
    println(s.result)
}
```

```
fun co(it:Iterator<State>? = null, sep:SequenceScope<State>? = null){
    val iter = it ?: sep?.iterator() ?: throw Throwable("invalid")
    if(iter.hasNext()) iter.next().let {st->
        st.target.then{it.text()}.then{
            st.result = it
            co(iter)
        }
    }
}
```

CO

```
class State{
    var result = ""
    lateinit var target:Promise<Response>
}
sequence{
    val s = State()
    s.target = window.fetch(Request("a.txt"))
    yield(s)
    s.target = window.fetch(Request(s.result))
    yield(s)
    println(s.result)
}
```

```
co(sequence{
    val s = State()
    s.target = window.fetch(Request("a.txt"))
    yield(s)
    s.target = window.fetch(Request(s.result))
    yield(s)
    println(s.result)
})
```

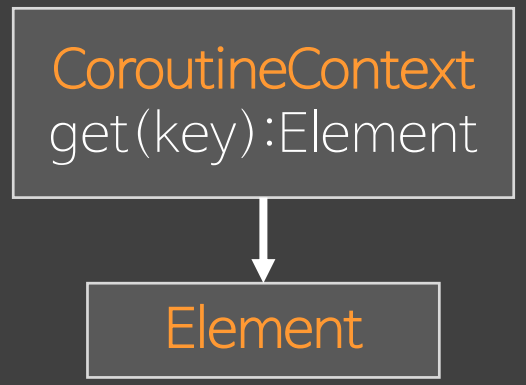
```
fun co(it:Iterator<State>? = null, sep:SequenceScope<State>? = null){
    val iter = it ?: sep?.iterator() ?: throw Throwable("invalid")
    if(iter.hasNext()) iter.next().let {st->
        st.target.then{it.text()}.then{
            st.result = it
            co(iter)
        }
    }
}
```

suspend & coroutine

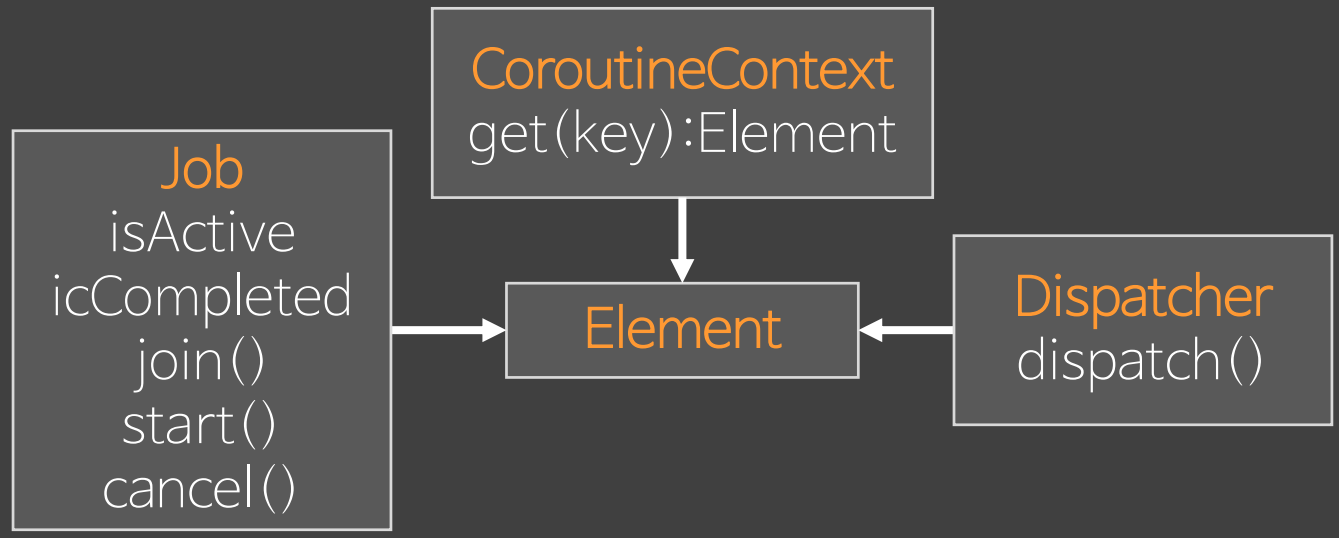
Kotlin Coroutines

CoroutineContext
get(key):Element

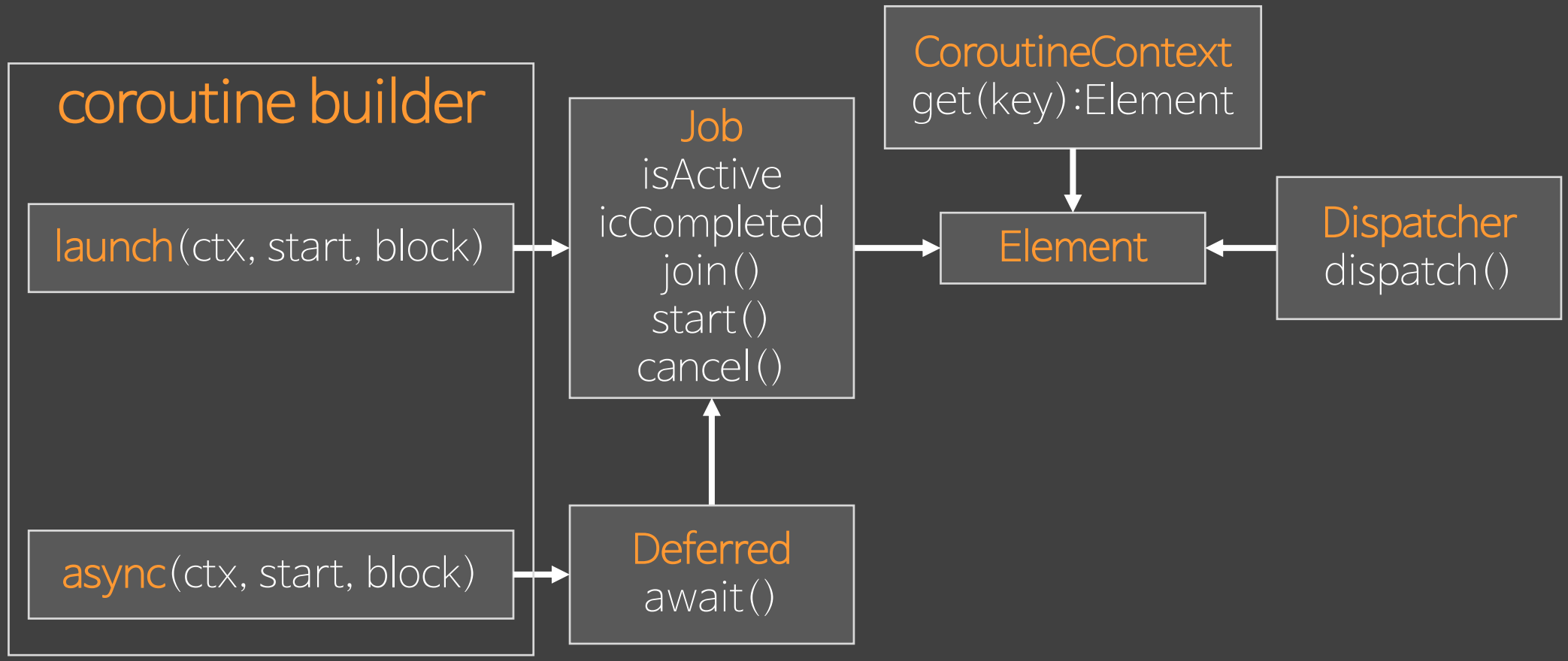
Kotlin Coroutines



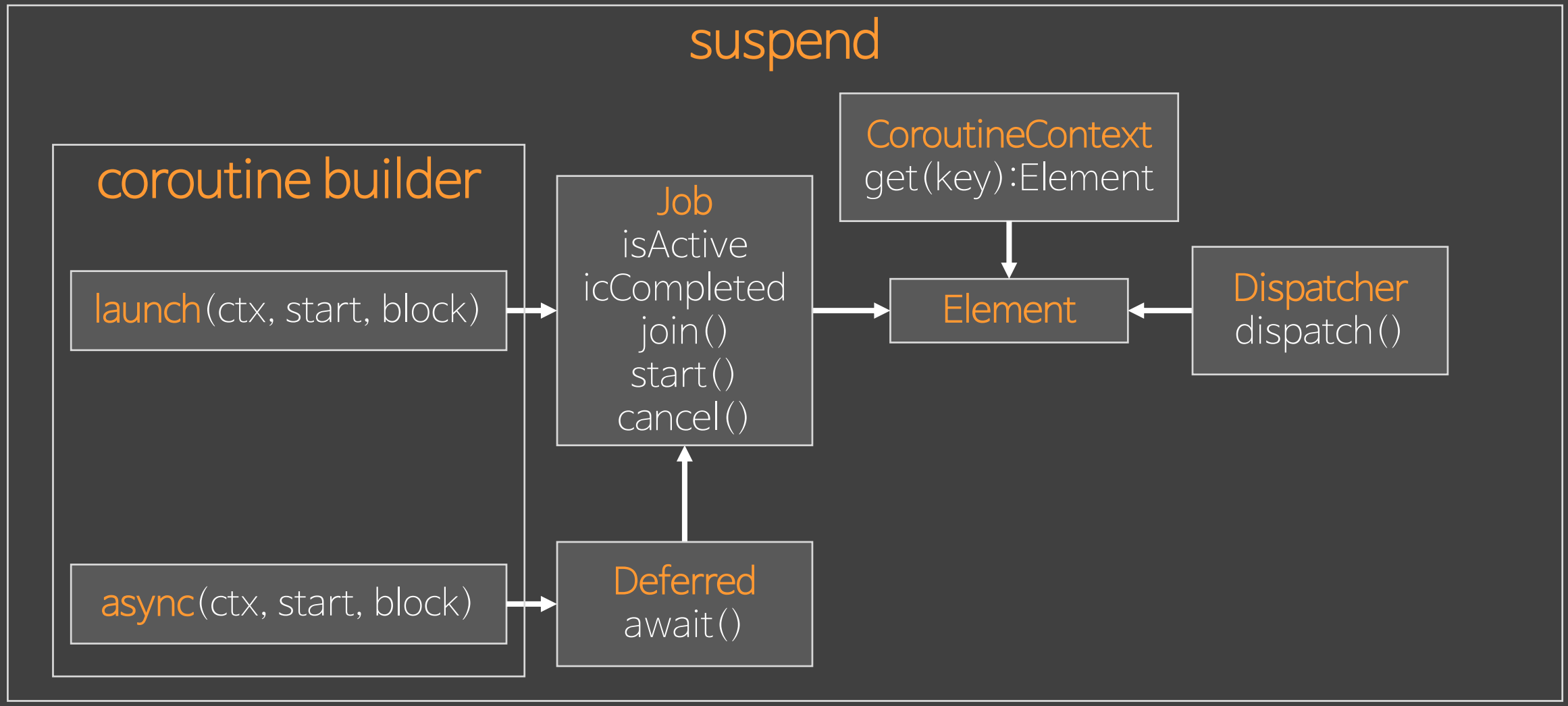
Kotlin Coroutines



Kotlin Coroutines



Kotlin Coroutines



task

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

task

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

```
suspend fun main(){
    println("a")
    println(task{it("b")})
    println("c")
}
```

task

```
suspend fun <T>task(block:
    cont:Continuation<T>-:
}
```

```
suspend fun main(){
    println("a")
    println(task{it("b")})
    println("c")
}
```

```
Coroutine$main.prototype.doResume = function () {
  do
    try {
      switch (this.state_0) {
        case 0:
          println('a');
          this.state_0 = 2;
          this.result_0 = task(main$lambda, this);
          if (this.result_0 === COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED;
          continue;
        case 1:
          throw this.exception_0;
        case 2:
          println(this.result_0);
          println('c');
          return;
        default:this.state_0 = 1;
          throw new Error('State Machine Unreachable execution');
      }
    } }
  while (true);
```

task

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

```
suspend fun main(){
    println("a")
    println(task{it("b")})
    println("c")
}
```

```
suspend fun timeout(t:Int):Unit = task{window.setTimeout({it(Unit)}, t)}
```

task

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

```
suspend fun main(){
    println("a")
    println(task{it("b")})
    println("c")
}
```

```
suspend fun timeout(t:Int):Unit = task{window.setTimeout({it(Unit)}, t)}
```

```
suspend fun main(){
    println("a")
    timeout(1000)
    println("b")
}
```

suspendCancellableCoroutine

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

suspendCancellableCoroutine

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

```
suspend fun <T> Promise<T>.await(): T = suspendCancellableCoroutine {
    cont: CancellableContinuation<T> ->
    this.then(
        onFulfilled = { cont.resume(it) },
        onRejected = { cont.resumeWithException(it) })
}
```

suspendCancellableCoroutine

```
suspend fun <T>task(block:((T)->Unit)->Unit):T = suspendCoroutine{
    cont:Continuation<T>->block{cont.resume(it)}
}
```

```
suspend fun <T> Promise<T>.await(): T = suspendCancellableCoroutine {
    cont: CancellableContinuation<T> ->
    this.then(
        onFulfilled = { cont.resume(it) },
        onRejected = { cont.resumeWithException(it) })
}
```

```
suspend fun main(){
    val response1 = window.fetch(Request("a.txt")).await()
    val text1 = response1.text().await()
    val response2 = window.fetch(Request(text1)).await()
    val text2 = response2.text().await()
    println(text2)
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("b")
    }
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }.start()
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }.join()
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }.start()
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }.join()
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }.start()
}
```

```
suspend fun main(){
    val a = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        "a"
    }
    val b = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        "b"
    }
    println(a.await())
    println(b.await())
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }.join()
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }.start()
}
```

```
suspend fun main(){
    val a = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        "a"
    }
    val b = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.LAZY
    ){
        timeout(1000)
        "b"
    }
    println(a.await())
    println(b.await())
}
```

launch & async

```
suspend fun main(){
    GlobalScope.launch(
        context = EmptyCoroutineContext,
        start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        println("a")
    }.join()
    GlobalScope.launch(
        context = Dispatchers.Default,
        start = CoroutineStart.LAZY
    ){
        timeout(1000)
        println("b")
    }.start()
}
```

```
suspend fun main(){
    val a = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.DEFAULT
    ){
        timeout(1000)
        "a"
    }
    val b = GlobalScope.async(
            context = EmptyCoroutineContext,
            start = CoroutineStart.LAZY
    ){
        timeout(1000)
        "b"
    }
    b.start()
    println(a.await())
    println(b.await())
}
```

launch & async

```
fun <T>async(block: suspend CoroutineScope.() -> T) = GlobalScope.async{block()}
fun launch(block: suspend CoroutineScope.() -> Unit) = GlobalScope.launch{block()}
```

launch & async

```
fun <T>async(block: suspend CoroutineScope.() -> T) = GlobalScope.async{block()}
fun launch(block: suspend CoroutineScope.() -> Unit) = GlobalScope.launch{block()}
```

```
suspend fun main(){
    val deferred = async{"b"}
    val job = launch{println("a")}
    println(deferred.await())
}
```

etc

gradle

```
compile 'org.jetbrains.kotlinx:kotlinx-coroutines-core-js:1.3.0-M1'
```

kotlinx.coroutine

https://github.com/Kotlin/kotlinx.coroutines/