

# **Advanced Kotlin Coroutines**

**Попов Матвей**

# План

- Вспомним что такое coroutines
- Базовые примитивы Kotlin Coroutines
- Как они устроены под капотом?
- Data race в Kotlin Coroutines
- Примитивы синхронизации
- Сравниваем корутины с другими языками

# Что же такое Coroutine?

**Корутины - легковесные  
потоки**

# Несколько фактов:

- Появились в Kotlin 1.1
- Стабильное API с версии 1.3
- Возможность писать асинхронный код в синхронном стиле
- Не являются частью стандартной библиотеки языка
- Всю магию делает компилятор

```
// Асинхронный подход
remoteCall { data →
    storeData(data) {
        println(data)
    }
}
```

```
// Синхронный подход
val data = remoteCall()
storeData(data)
println(data)
```

# **Сравним потоки и корутины**

# Почему корутины это легковесные потоки?

- Можно создать миллиард корутин
- При создании несколько тысяч потоков будет OutOfMemory
- Корутины не требуют системных ресурсов
- Корутины используют заданные пулы потоков
- Корутина может выполняться на разных потоках



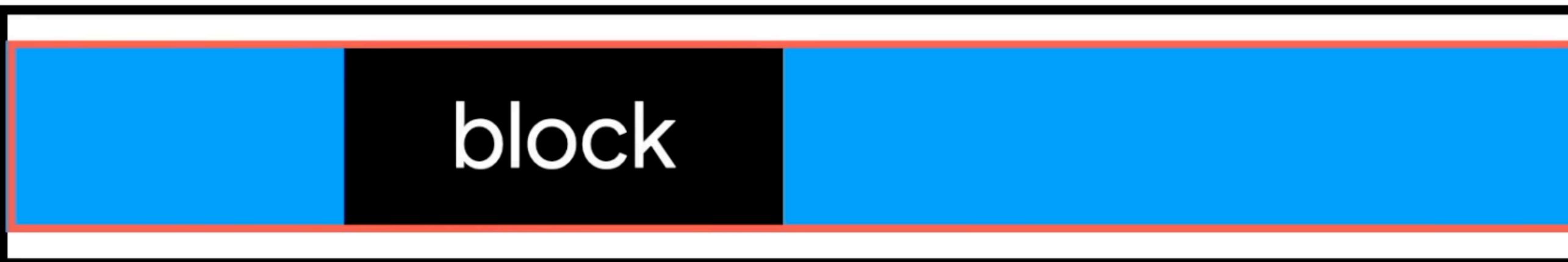
ПОТОК

Coroutine

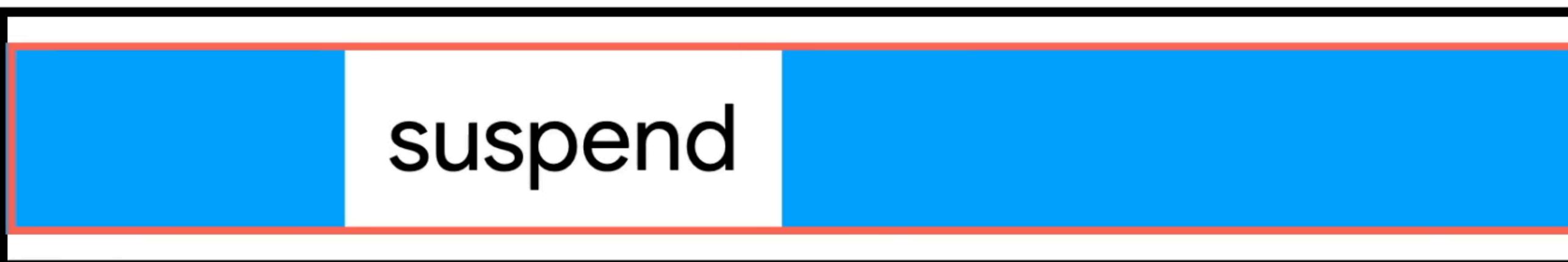
Dispatcher

**Так какая же разница?**

Поток



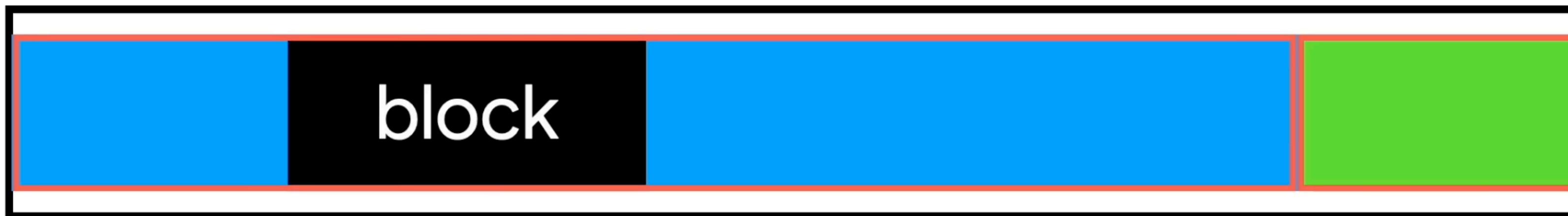
Coroutine



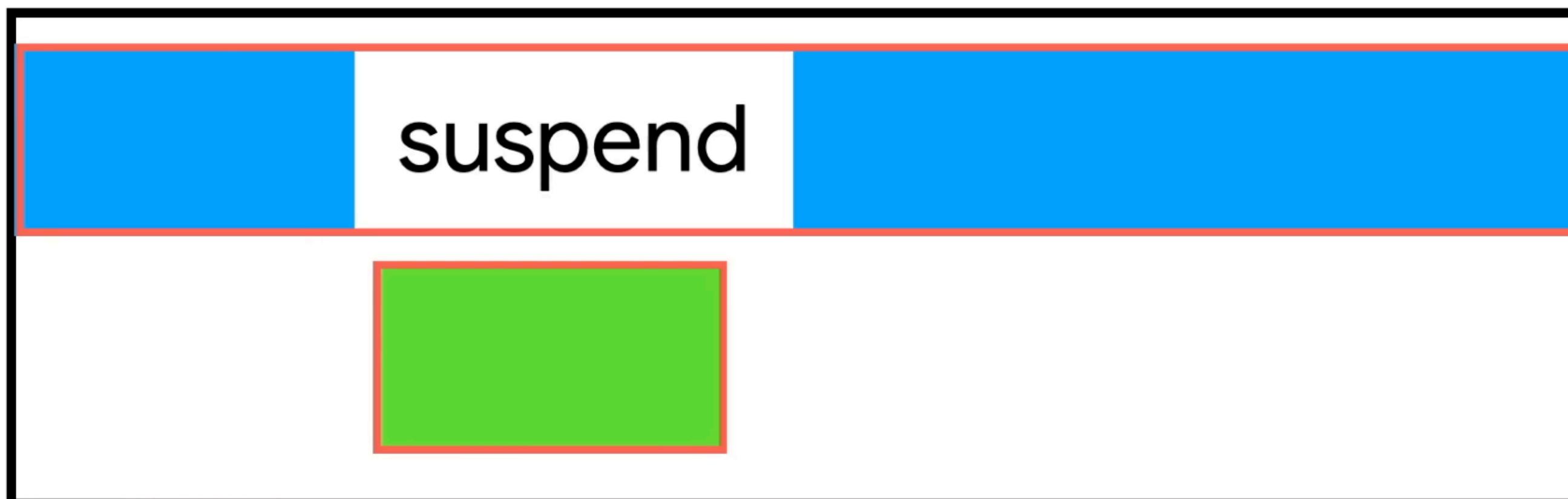
Поток

задача

## Поток



## Coroutine



Поток

Задача



Поток

Coroutine

Dispatcher

# **Базовые примитивы**

**Как следить за выполнением  
корутины?**

- Можно хранить на них ссылки(как в RxJava)
- Любая запускаемая корутина привязана к CoroutineScope
- CoroutineScope описывает жизненный цикл корутины/корутин
- CoroutineScope позволяет следить за ресурсами

# Как останавливать корутины?

- В Java Threads нужно проверять каждый такт цикла на interrupted()
- В KotlinCoroutines похожий функционал: нужно проверять isActive
- Важно проверять перед вызовом suspend операции

**Давайте на все это  
посмотрим!**

# Настраиваем корутины

# CoroutineContext

- Каждая корутина выполняется в каком-то контексте
- Контекст представлен классом CoroutineContext
- CoroutineContext это ассоциативный массив

`CoroutineContext ~ Map<Key<Element>, Element>`

```
public interface CoroutineContext {  
  
    operator fun <E : Element> get(key: Key<E>): E?  
  
    fun <R> fold(initial: R, operation: (R, Element) → R): R  
  
    operator fun plus(context: CoroutineContext): CoroutineContext  
  
    fun minusKey(key: Key<*>): CoroutineContext  
}
```

```
interface Key<E : Element>

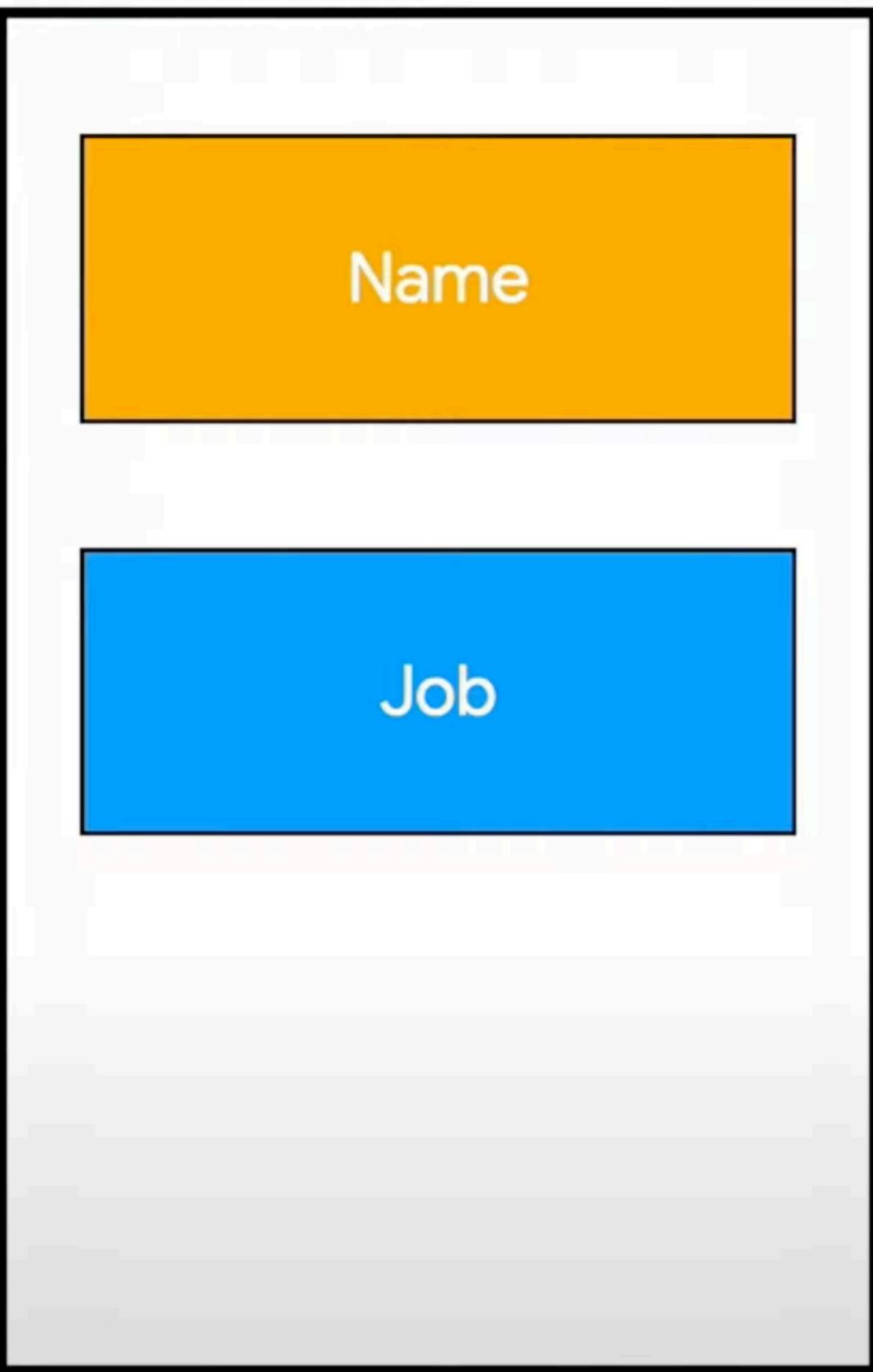
interface Element : CoroutineContext {

    val key: Key<*>
}
```

Context1



Context2



+

=

ContextSum

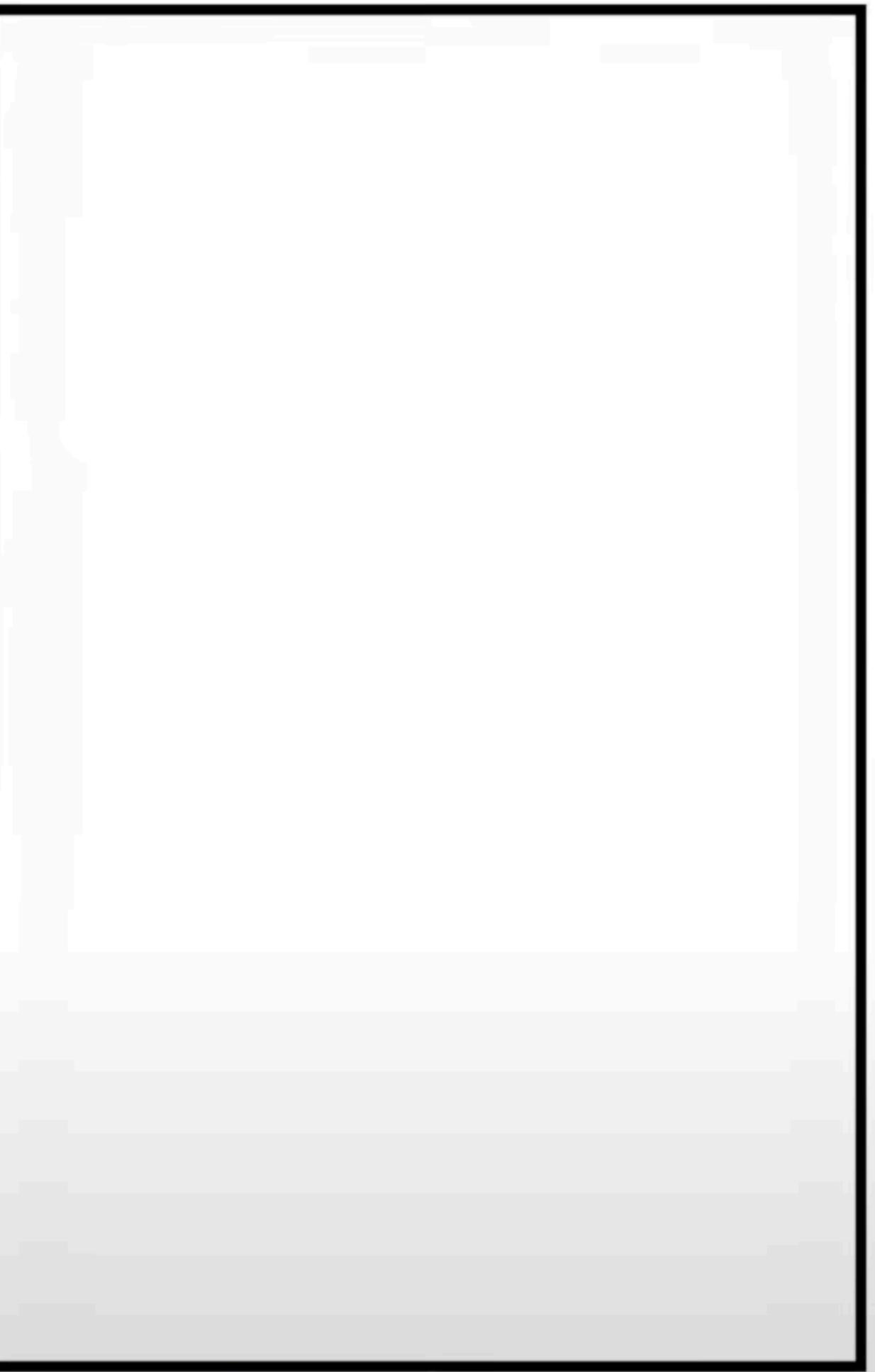


Context1



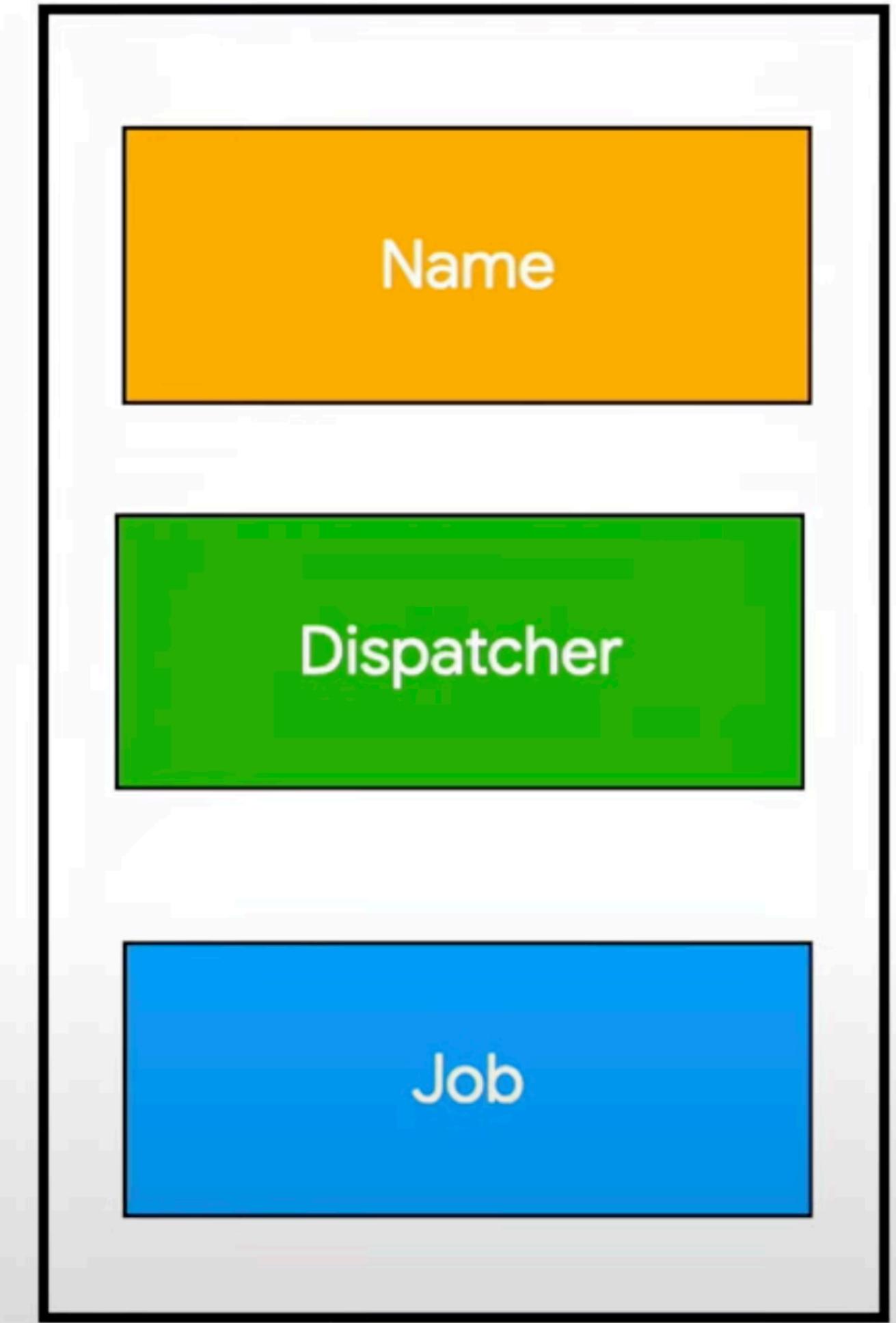
+

Context2



=

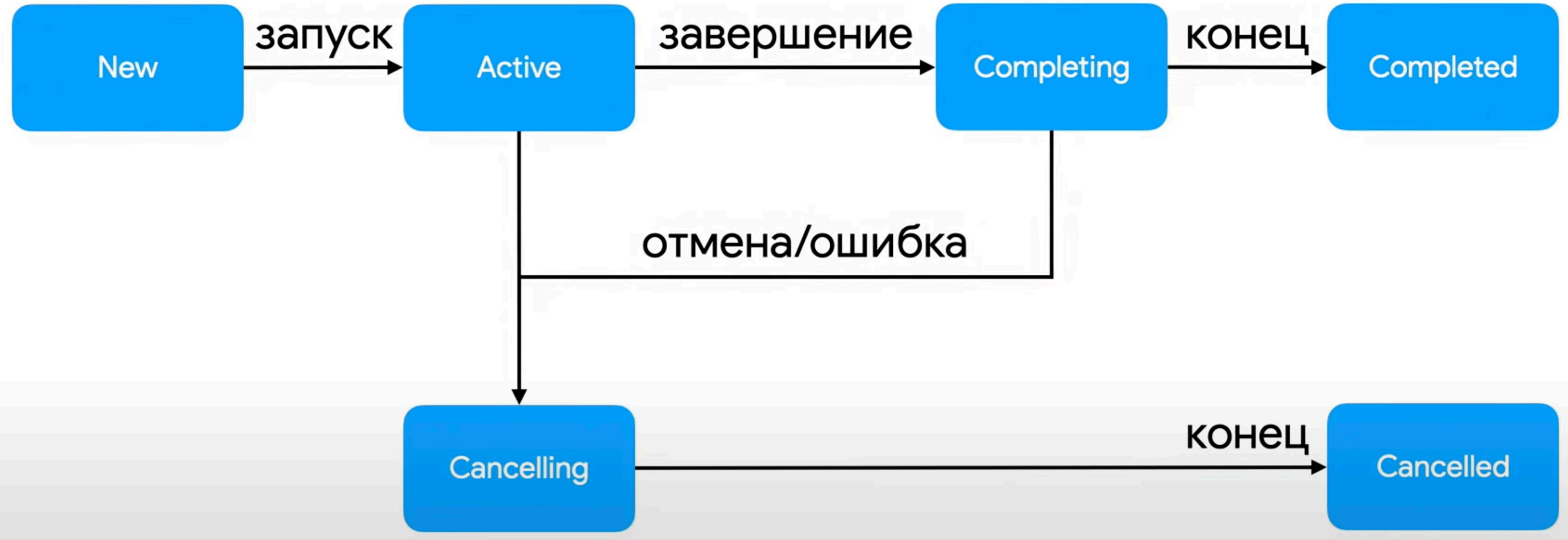
ContextSum

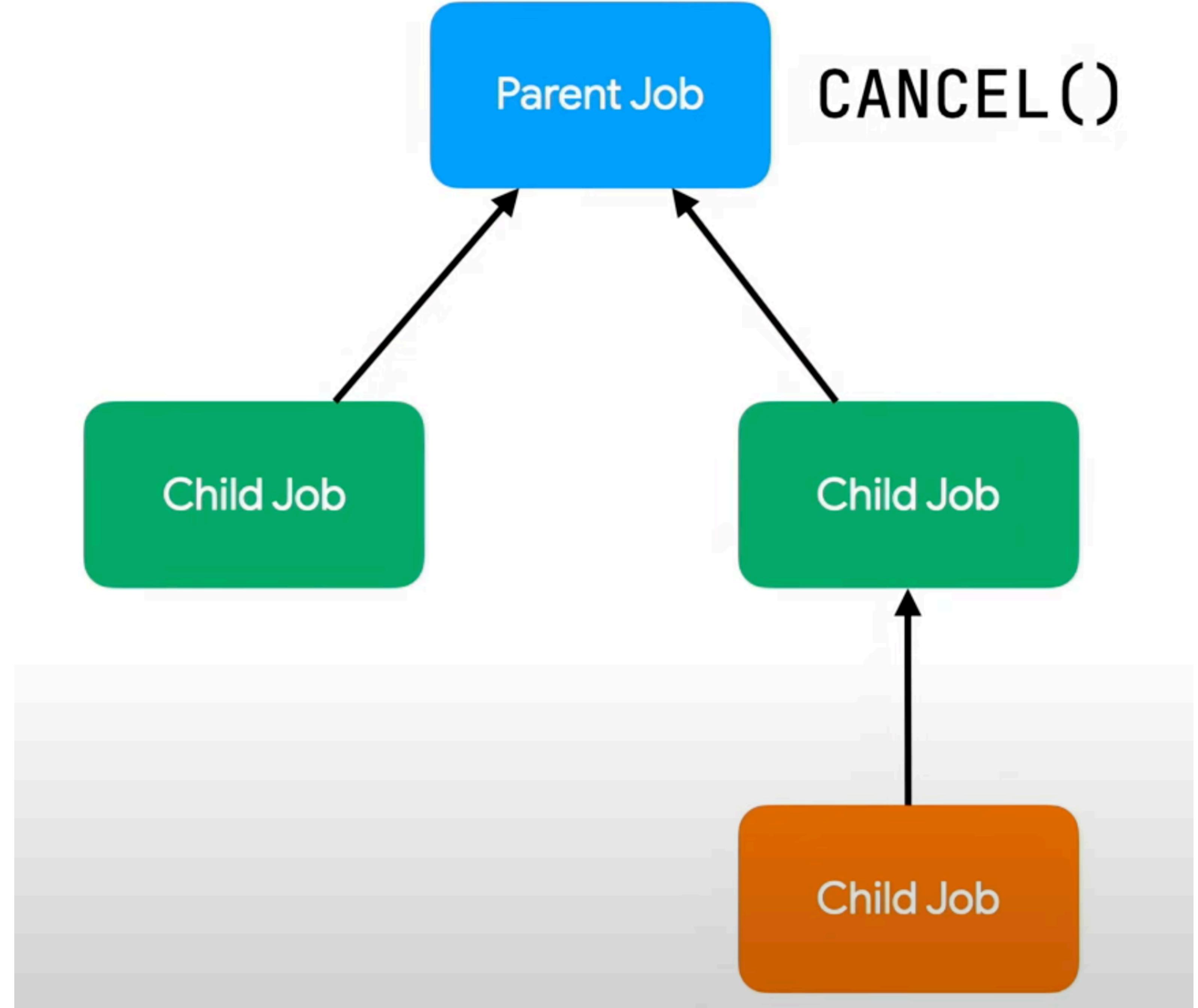


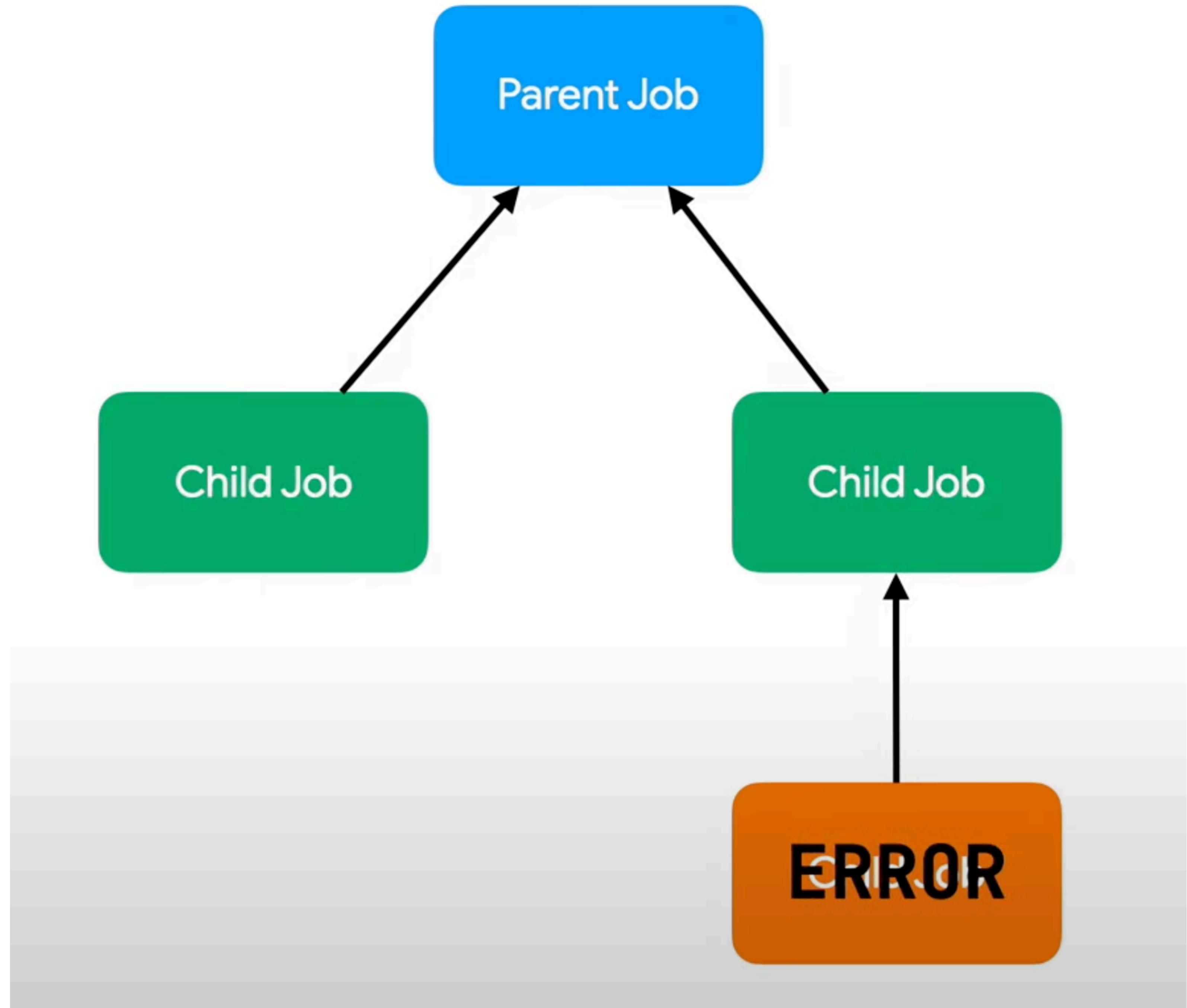
**Какие существую  
компоненты?**

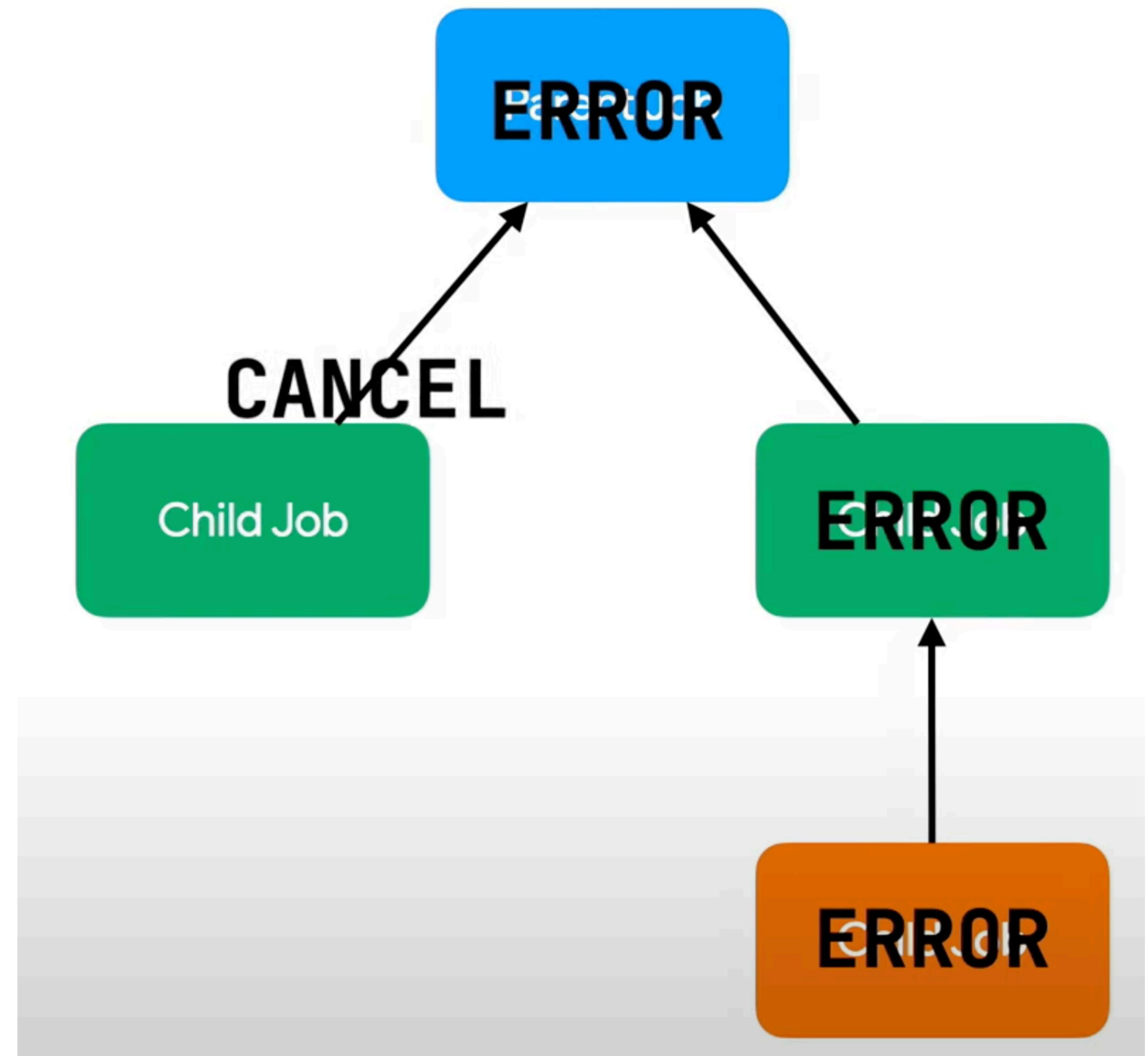
# Job

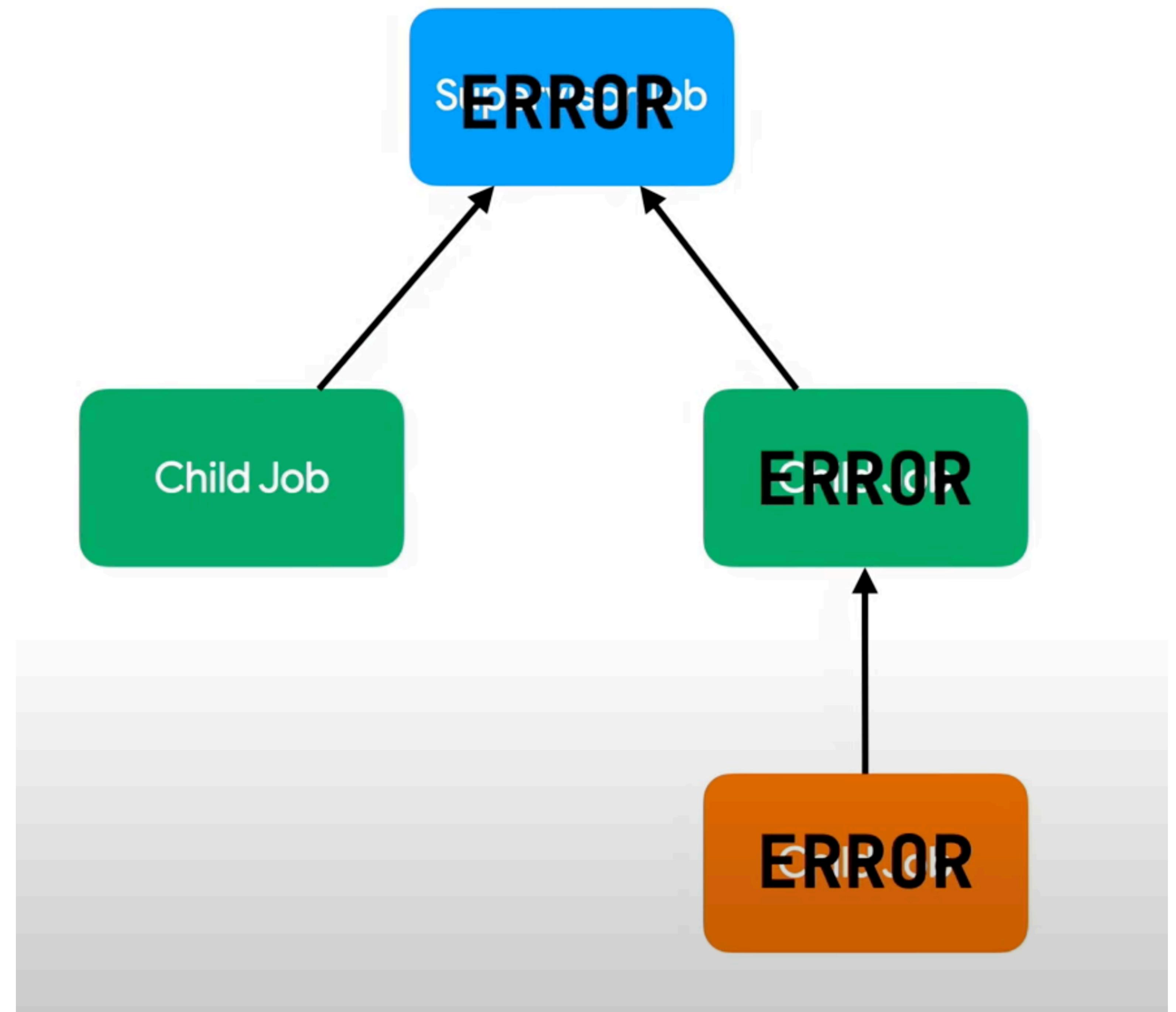
- Можно управлять работой корутины
- Job можно отменить
- Имеет свой жизненный цикл
- На основе ее можно организовать иерархию:  
родитель <-> ребенок











```
interface Job : CoroutineContext.Element {  
  
    val isActive: Boolean  
    val isCompleted: Boolean  
    val isCancelled: Boolean  
  
    fun cancel(cause: CancellationException? = null)  
  
    fun invokeOnCompletion(handler: CompletionHandler): DisposableHandle  
  
    suspend fun join()  
  
    fun start(): Boolean  
  
    val children: Sequence<Job>  
}  
  
fun Job.ensureActive()
```

# **CoroutineDispatcher**

- Отвечает за то, на каком потоке/  
потоках будет выполняться  
корутина
- Решает на каком пуле потоков  
выполнить корутину
-

```
object Dispatchers {  
    val Default: CoroutineDispatcher  
    val Main: MainCoroutineDispatcher  
    val Unconfined: CoroutineDispatcher  
    val IO: CoroutineDispatcher  
}
```

```
class MainCoroutineDispatcher : CoroutineDispatcher() {  
    val immediate: MainCoroutineDispatcher  
}
```

# **CoroutineExceptionHandler**

- Обработчик ошибок, которые происходят в корутинах
- Перехватывает ошибки в детях
- Когда ошибка захватывается, то родителям не пробрасывается
- Можно определить дефолтный через ServiceLoader

```
CoroutineExceptionHandler { context, error: Throwable →  
    logError(error)  
}
```

# CoroutineName

- Задает осознанное имя корутине

```
launch(CoroutineName("LongTask"))
```

# Жизненный цикл и Structured Concurrency

# **Structured Concurrency**

**Парадигма программирования, направленная на  
повышение ясности, качества и времени разработки  
программы за счет использования структурированного  
подхода к параллельному программированию**

# Проще говоря:

- Использование статической области видимости
- Позволяет легко рассуждать о коде и его потоке
- Упрощает понимание
- Читаете код как будто он синхронный
- Предоставляет иерархическую структуру

**Что значит жизненный цикл?**

- Асинхронная операция должны быть оставлена когда вам не нужен результат
- В Java Threads это Thread.interrupt()
- В RxJava это Disposable
- Нужно сохранять ссылки на объекты
- Отменять на уровне API не обязательно

**fun CoroutineScope.launch()**

**fun CoroutineScope.async()**

**fun runBlocking()**

# Принципы работы CoroutineScope

## Отмена Scope - отмена корутин

Scope может отменить выполнение всех дочерних корутин, если возникнет ошибка или операция будет отменена

## Scope знает про все корутины

Любая корутина, запускаемая в скоупе, будет хранится ссылкой в нём через отношение "родитель-ребенок" у Job

# **я тебя буду ждать**

scope автоматически ожидает выполнения всех дочерних корутин,  
но не обязательно завершается вместе с ними

# **CoroutineScope vs CoroutineContext**

```
interface CoroutineScope {  
    val coroutineContext: CoroutineContext  
}
```

# В чем же отличие?

- Разное целевое назначение использования
- CoroutineContext набор параметров для выполнения корутины
- CoroutineScope предназначен для объединения всех корутин
- CoroutineScope источник параметров по умолчанию

# **Создание CoroutineScope**

*CoroutineScope(Job() + Dispatchers.Default)*

```
suspend fun loadData() {  
    coroutineScope { this: CoroutineScope  
        // Параллельные операции внутри scope  
        launch(Dispatchers.IO) { this: CoroutineScope  
            loadRemote()  
        }  
        launch(Dispatchers.IO) { this: CoroutineScope  
            loadLocal()  
        }  
    }  
}
```

**Как это устроено?**

## Креш в coroutineScope

Если внутри coroutineScope произойдет креш, то он будет прорасываться родительскому scope

# Остановка родительского scope

Остановка родительского scope приведет к остановке scope, полученного в coroutineScope

# **coroutineScope** ждёт всех

Функция coroutineScope приостановит выполнение корутины, до тех пор пока все корутины и весь код внутри не будет выполнен

# Scope внутри корутины

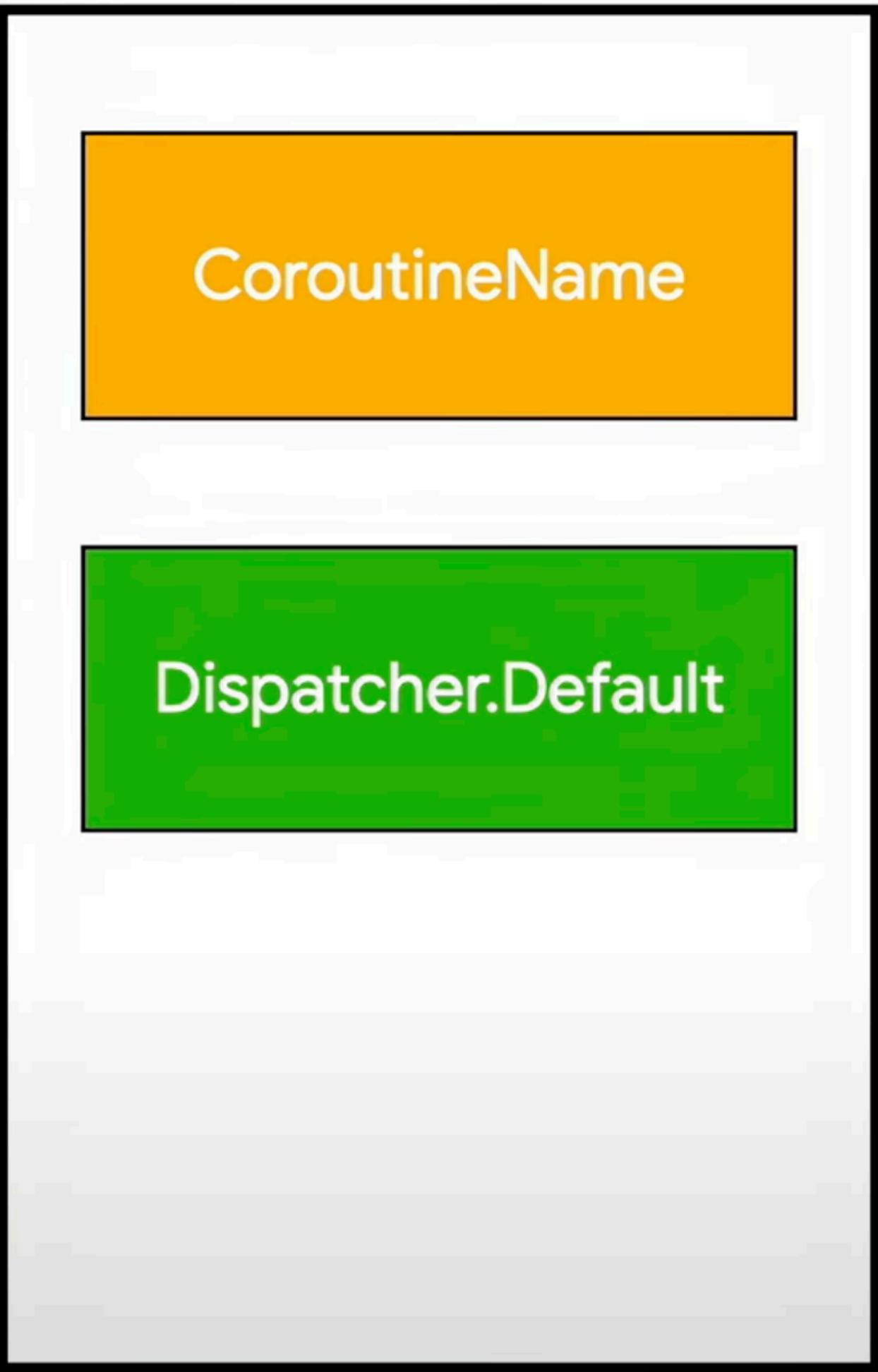
```
Launch() { this: CoroutineScope  
    ...  
}
```

**Как формируется вложенный  
CoroutineScope?**

## CoroutineScope



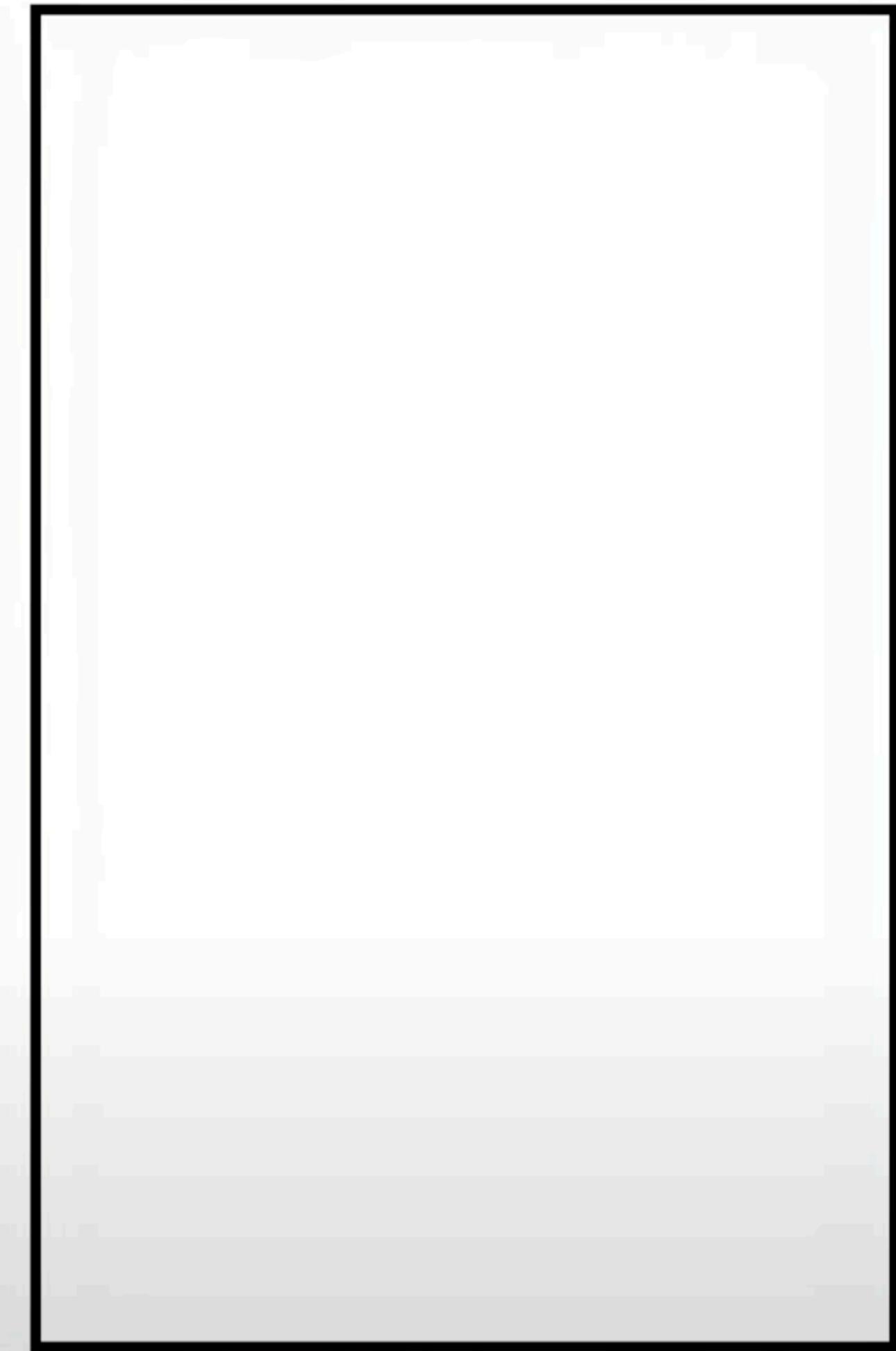
## Context in Coroutine



+

=

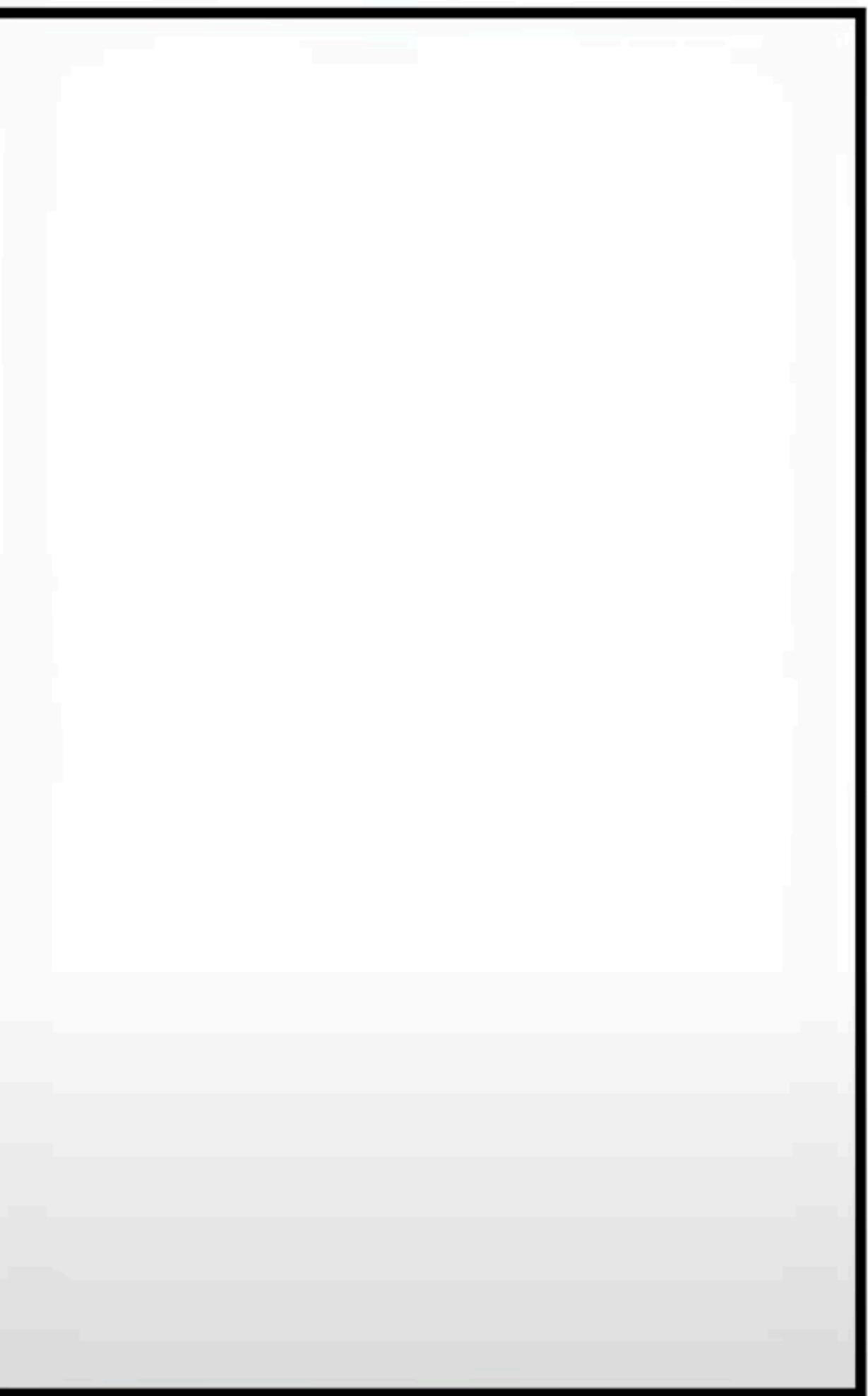
## Scope in Coroutine



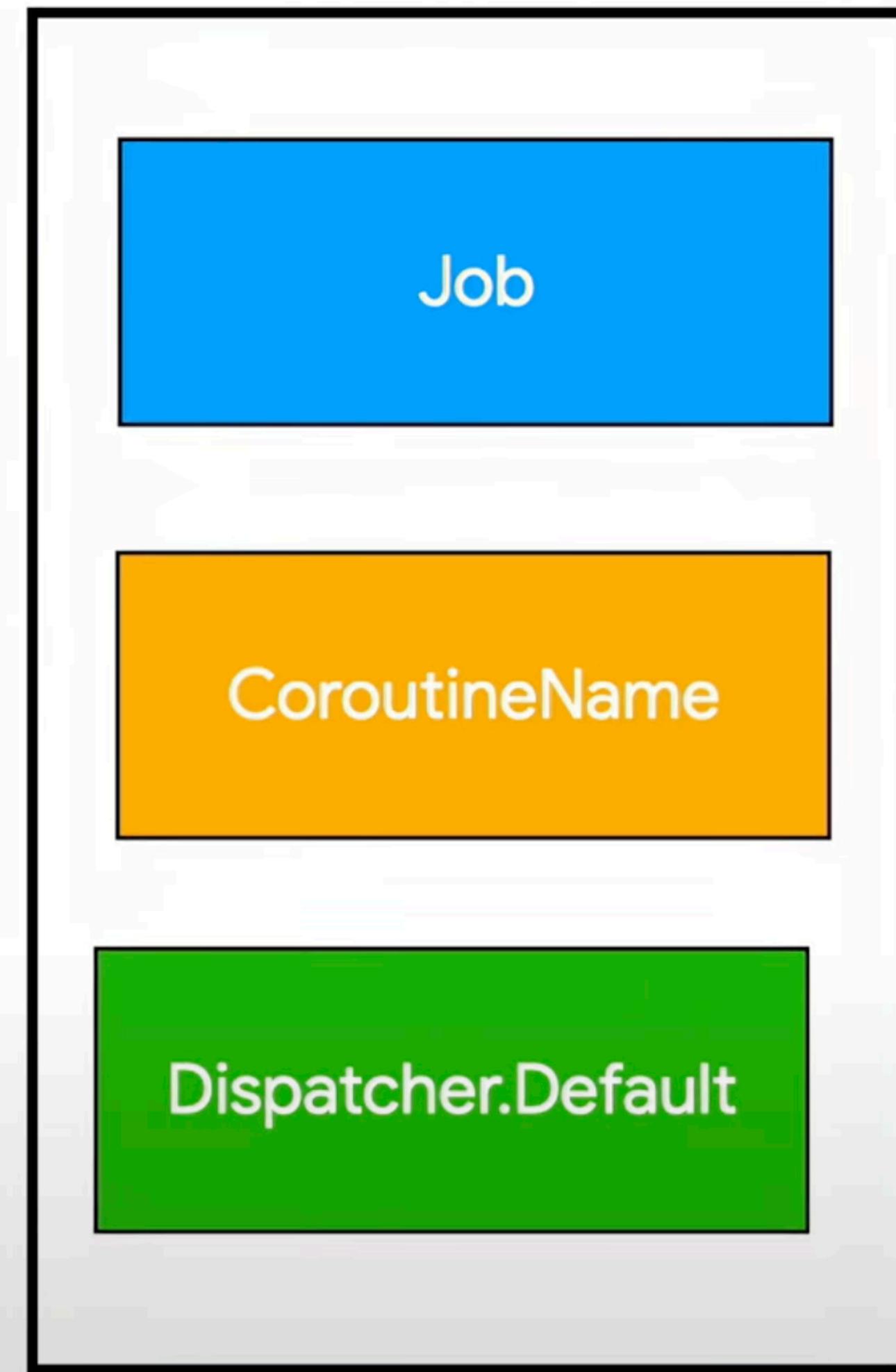
## CoroutineScope



## Context в Coroutine



## Scope in Coroutine



**А что там с отменой?**

```
CoroutineScope.cancel(cause: CancellationException? = null)
```

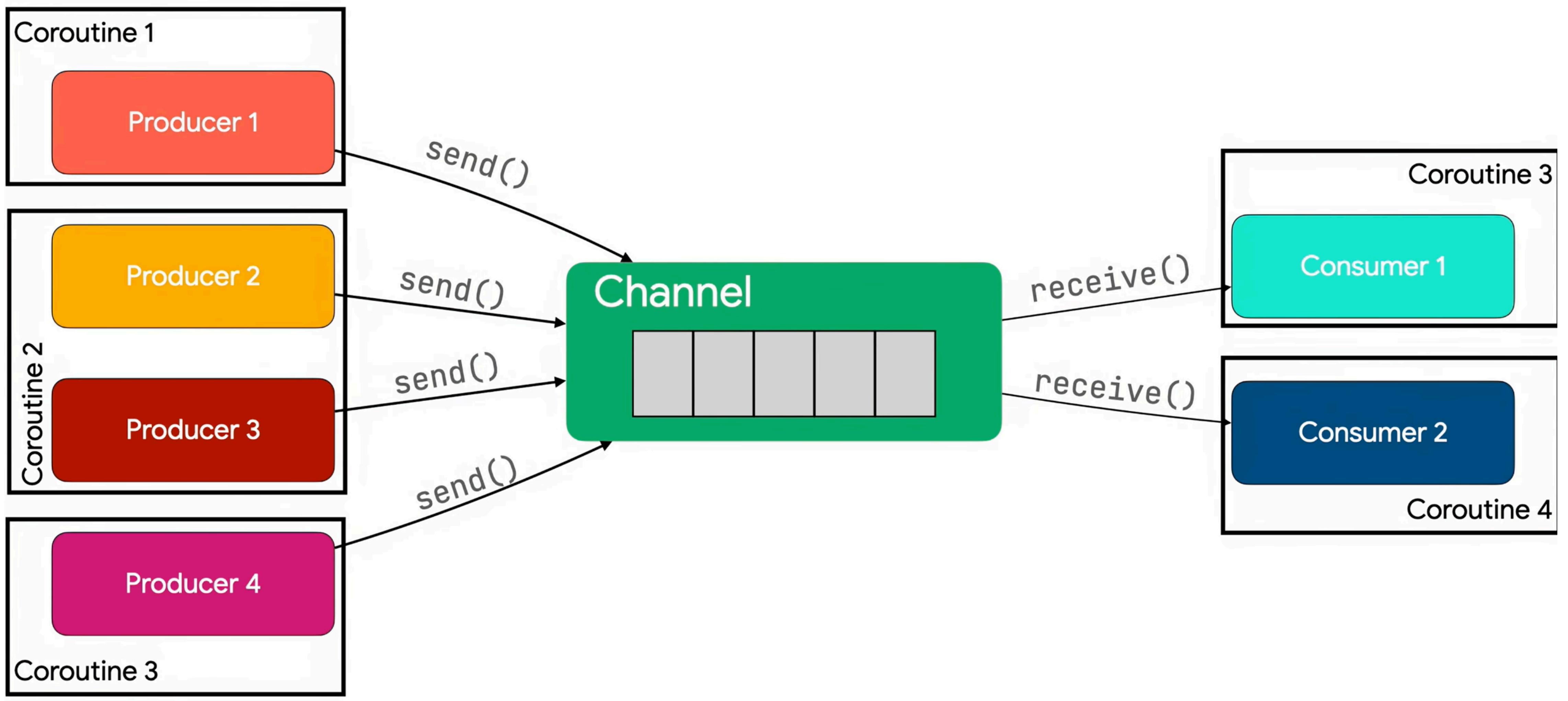
```
scope.coroutineContext[Job]?.  
    cancel(cause: CancellationException? = null)
```

# Смена контекста в CoroutineScope

```
suspend fun loadData() {  
    withContext(Dispatchers.IO) { this: CoroutineScope  
        loadRemote()  
    }  
    // Код выполнится после кода в withContext  
}
```

**Как можно обмениваться  
данным в Kotlin-Coroutines?**

# Channel

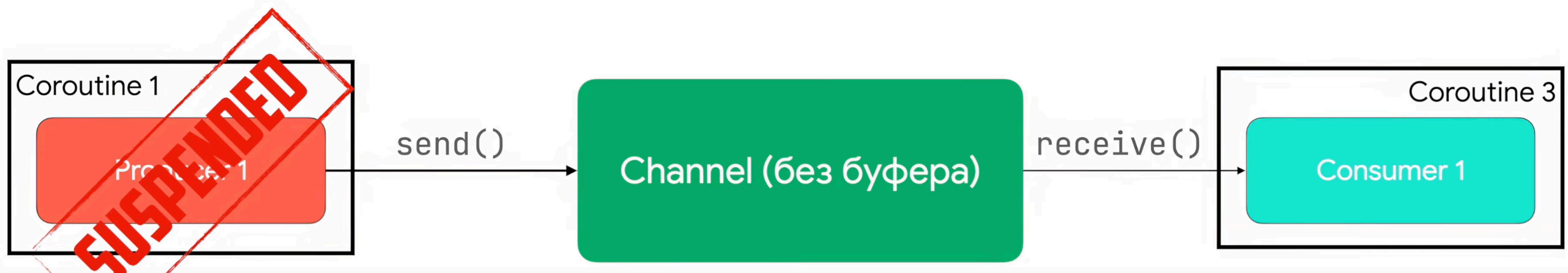


- Аналог блокирующих очередей из Java
- Приостановка при отправке
- Каналы можно закрывать
- Может быть много отправителей и получателей
- По умолчанию нет буфера
- Предназначены для обмена данными между несколькими корутинами

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

```
interface ReceiveChannel<out E> {  
    suspend fun receive(): E  
    fun tryReceive(): ChannelResult<E>  
    operator fun iterator(): ChannelIterator<E>  
    fun cancel(cause: CancellationException? = null)  
}
```

```
interface SendChannel<in E> {  
    suspend fun send(element: E)  
    fun trySend(element: E): ChannelResult<Unit>  
    fun cancel(cause: CancellationException? = null)  
}
```



```
fun <E> Channel<  
    // Размер буфера  
    capacity: Int = RENDEZVOUS,  
  
    // Что делать при переполнение буфера  
    onBufferOverflow: BufferOverflow = SUSPEND,  
  
    // Сюда попадут все значения, которые не доставили  
    onUndeliveredElement: ((E) → Unit)? = null  
): Channel<E>
```

# Стандартные размеры буфера Channel

## 👉 **RENDEZVOUS**

Без буфера. Значение для любого Channel по умолчанию

## 👉 **CONFLATED**

Размер буфера = 1. Хранит только последнее полученное значение, а предыдущее удаляется

## 👉 **BUFFERED**

Задает стандартный размер буфера, который определен в свойствах окружения (по умолчанию это 64)

## 👉 **UNLIMITED**

Максимально возможный размер буфера (`Int.MAX_SIZE`)

[Все константы находятся в интерфейсе Channel](#)

# Политика поведения при переполнении буфера

## 👉 **SUSPEND**

Корутина будет приостановлена, если принять значение сейчас некому, а буфер переполнен или отсутствует. Поведение по умолчанию.

## 👉 **DROP\_OLDEST, DROP\_LATEST**

Удаляет значения (самые старые/самые новые) в буфере при его переполнении. Вызов send() никогда не приостановит корутину, а trySend() всегда успешно выполнится, при условии наличия буфера в Channel

# **Actor и Producer**

```
val channel: ReceiveChannel<Int> =  
    scope.produce {  
        var x = 1  
        while (true) {  
            send(x++)  
        }  
    }
```

```
scope.launch {  
    val data = channel.receive()  
}
```

```
val actor: SendChannel<Any> = scope.actor {  
    →    val data = receive()  
        writeStringToFile(data)  
}  
  
scope.launch {  
    →    actor.send(data)  
}
```

# Синхронизация между корутинами

# Что можно использовать из JDK?

- Можно использовать потокобезопасные коллекции
- Нельзя использовать различные Lock и synchronized

**Снятие блокировки потока должно  
происходить на том же потоке,  
котором она и была захвачена**

```
var counter = 0
val lock = ReentrantLock()

val jobs = List(100) {
    scope.launch(start = CoroutineStart.LAZY) {
        repeat(1_000) {
            lock.withLock {
                counter += generateInt()
            }
        }
    }
}
```

**Критическая секция** – участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения.

**Что же делать?**

# Использовать Mutex

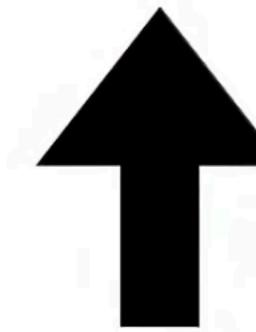
```
var counter = 0
val mutex = Mutex()

val jobs = List(100) {
    scope.launch(start = CoroutineStart.LAZY) {
        repeat(1_000) {
            mutex.withLock {
                counter += generateInt()
            }
        }
    }
}
jobs.joinAll()
```

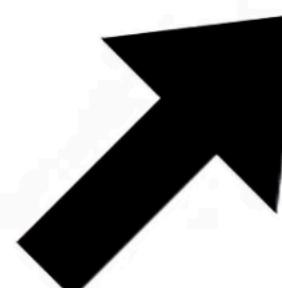
**Также можно использовать  
Channel**

# **Синхронизация через коммуникацию**

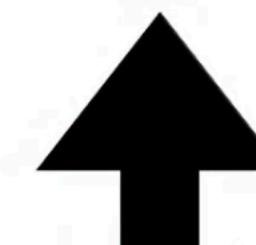
ДАННЫЕ



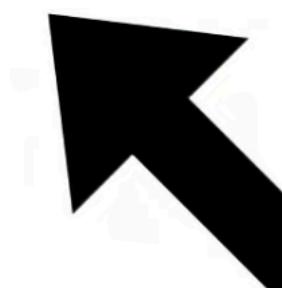
ACTOR



Coroutine 1



Coroutine 2



Coroutine 3

**Почему не java.util.concurrent?**

- Потому что это API Java
- Оно не умеет в мультиплатформость

# Flow

# Flow

Асинхронный поток данных, который последовательно выдает значения и завершается успешно либо с ошибкой

**Что значит холодный?**

**Холодный стрим выдаёт  
значения только при запуске**

**Горячий стрим начинает  
работать сразу при создании**

```
interface Flow<T> {  
  
    suspend fun collect(collector: FlowCollector<T>)  
}
```

```
fun interface FlowCollector<T> {  
  
    suspend fun emit(value: T)  
}
```

**А где же операторы?**

```
fun <T, R> Flow<T>.map(transform: suspend (value: T) → R): Flow<R>

fun <T> Flow<T>.onEach(action: suspend (T) → Unit): Flow<T>

fun <T> Flow<T>.filter(predicate: suspend (T) → Boolean): Flow<T>
```

# FlowBuilders

# Создание Flow

*flowOf(1)*

*flowOf("a", "b", "c", "d")*

*listOf("a", "b", "c", "d").asFlow()*

# Создание Flow

```
fun fibonacci(count: Int) = flow {
    require(count ≥ 1) { "count must be positive value" }
    ↪ emit(1)
    ↪ if (count ≥ 2) emit(1)

    var prev = 1
    var cur = 1
    for (i in 3..count) {
        val a = cur
        cur += prev
        prev = a
        ↪ emit(cur)
    }
}
```

# **Преобразование потока**

Входной поток (upstream)



`Flow<T>.map(transform: suspend (value: T) → R): Flow<R>`



Оператор

Выходной поток (downstream)



Входной и выходной поток - это относительные понятия касательно оператора

- map / mapNotNull
- switchMap
- combineLatest
- debounce / sample
- delayEach / delayFlow
- filter / filterNot / filterIsInstance / filterNotNull
- zip
- catch
- onEach / onCompletion
- flatMapConcat / flatMapMerge
- flattenConcat / flattenMerge

- collect
- collectIndexed
- collectFirst
- collectLatest
- first()
- firstOrNull()
- fold()
- last()
- lastOrNull()
- launchIn()
- reduce()
- single()
- singleOrNull()
- toList()
- toSet()

flow  
collector



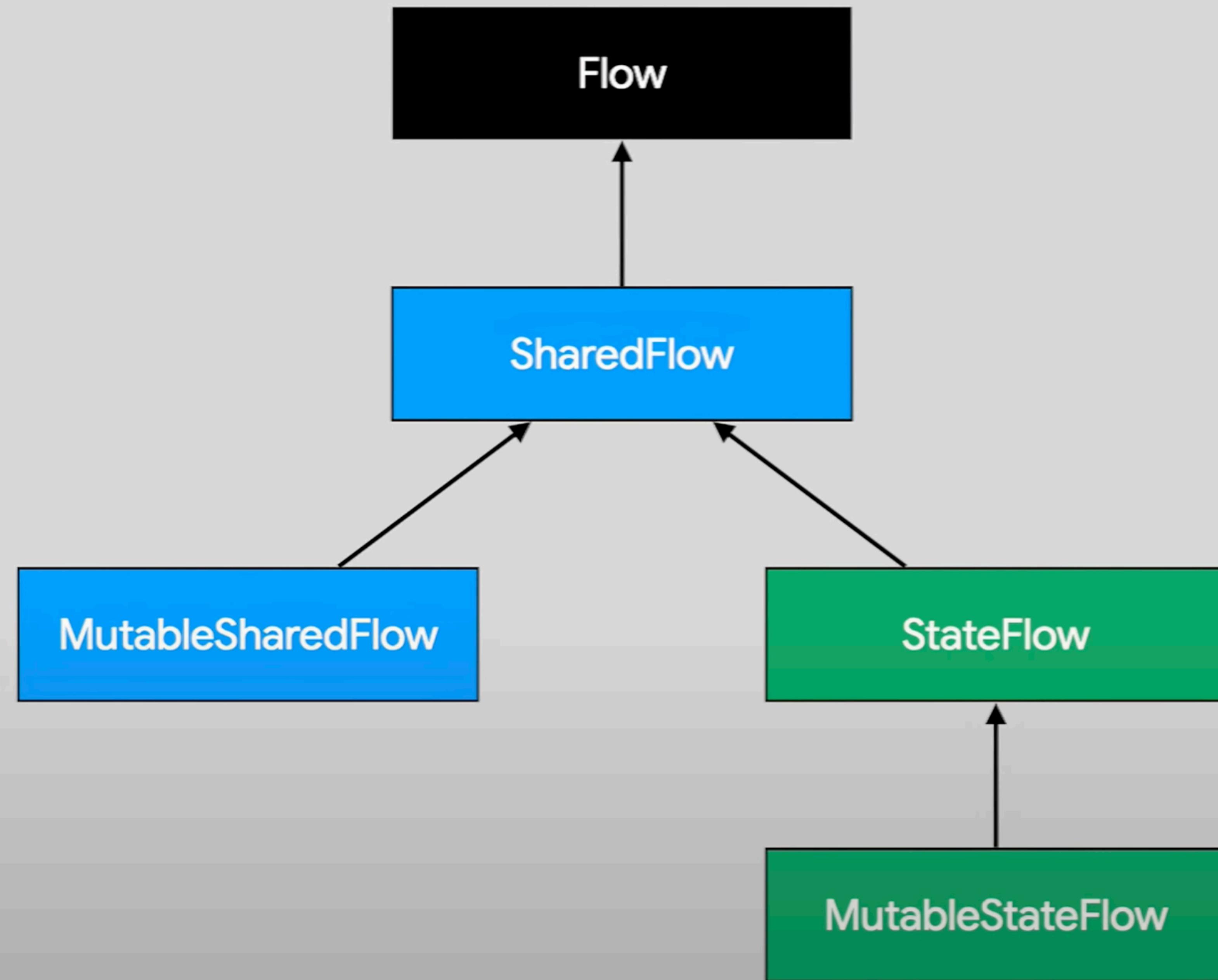
Flow эмитирует следующее значение только после того как выполняются все операторы для него и collector обработает его

**Данные можно кэшировать**

```
flow.buffer(  
    capacity = Channel.BUFFERED,  
    onBufferOverflow = BufferOverflow.SUSPEND  
)
```

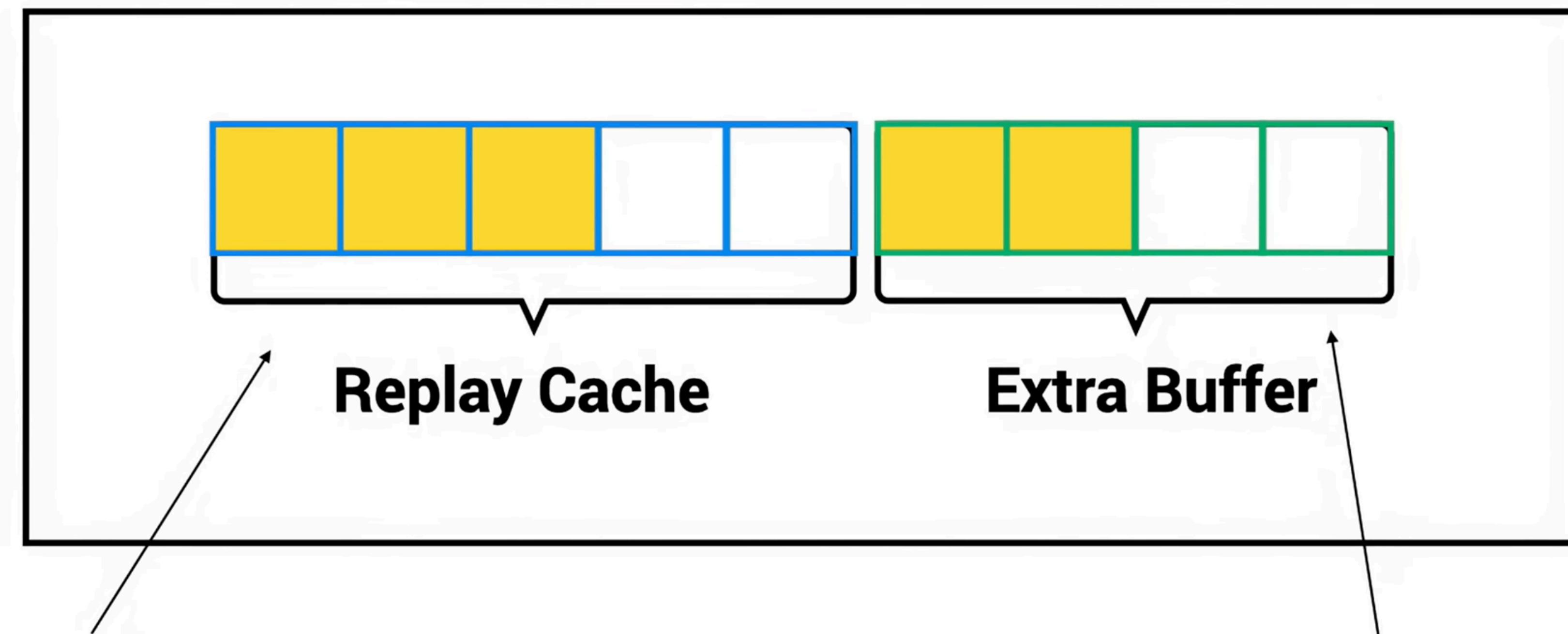
# **SharedFlow**

- Горячий Flow, который эмитит значения всем подписчикам
- Бесконечный Flow
- Эмитит сразу после создания
- Может содержать несколько подписчиков
- Не имеет контекста выполнения



**Существует буфер**

```
MutableSharedFlow(    replay = 5 , extraBufferCapacity = 4)
```



Элементы доставляются всем  
новым подписчикам

Элементы сохраняются при наличии  
подписчиков, когда они не могут быть  
доставлены сразу же.

Очищается при отсутствии подписчиков

# Политика поведения при переполнении буфера

## 👉 **SUSPEND**

Корутина будет приостановлена, если принять значение сейчас некому, а буфер переполнен или отсутствует. Поведение по умолчанию.

## 👉 **DROP\_OLDEST, DROP\_LATEST**

Удаляет значения (самые старые/самые новые) в буфере при его переполнении. Вызов send() никогда не приостановит корутину, а trySend() всегда успешно выполнится, при условии наличия буфера в Channel

# **StateFlow**

```
interface StateFlow<out T> : SharedFlow<T> {  
    val value: T  
}
```

```
fun <T> MutableStateFlow(value: T): MutableStateFlow<T>
```

```
interface MutableStateFlow<T> : StateFlow<T>, MutableSharedFlow<T> {  
    var value: T  
    fun compareAndSet(expect: T, update: T): Boolean  
}
```

# StateFlow<T>

~

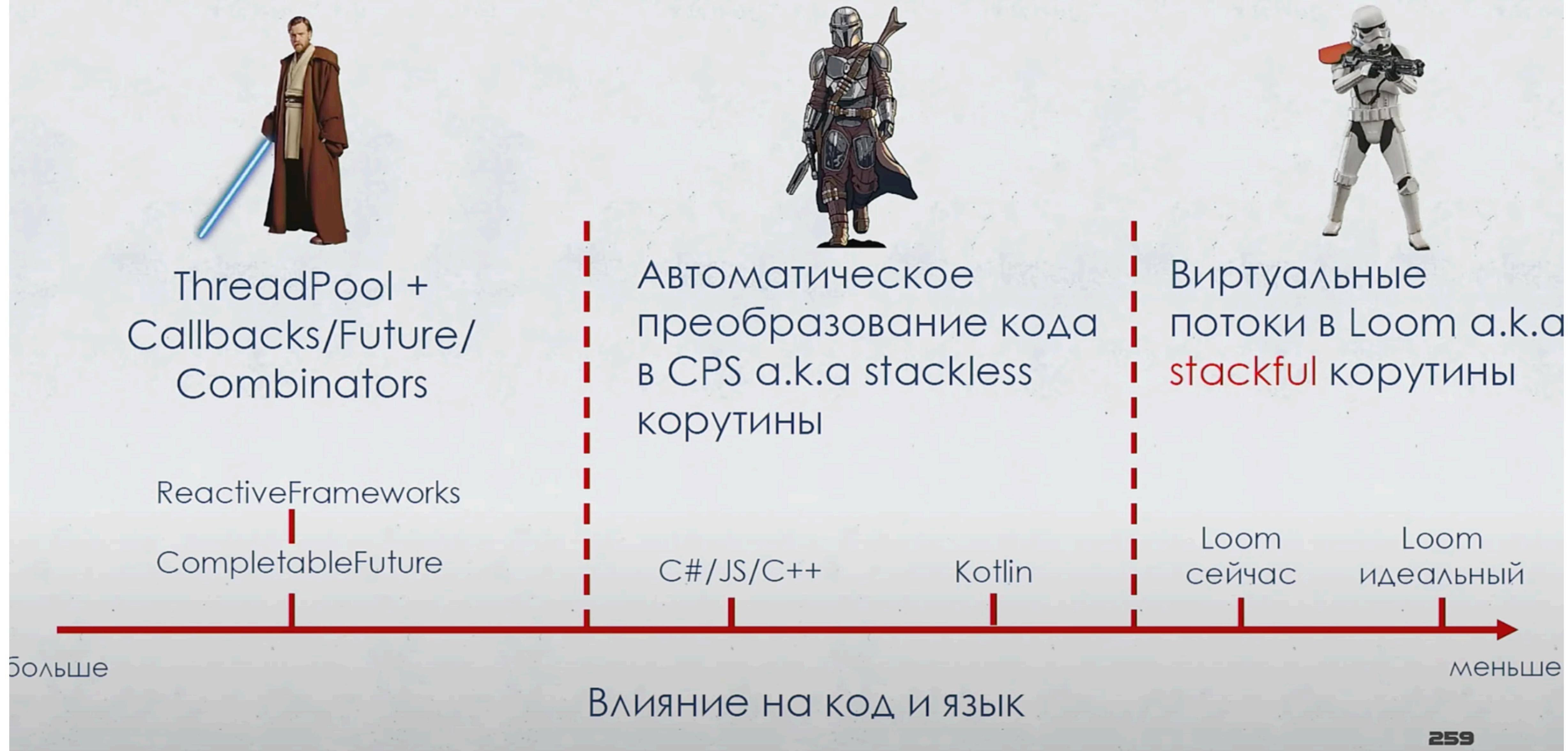
```
SharedFlow<T>(  
    replay = 1,  
    extraBufferCapacity = 0,  
    onBufferOverflow = BufferOverflow.DROP_OLDEST  
)
```

# Backpressure

Ситуация когда генерация новых событий происходит быстрее чем они успевают обрабатываться, в результате чего появляется очередь данных

**А что там с Coroutines в других языка?**

# РЕШЕНИЯ ПРОБЛЕМЫ ТРЕДОВ



**Получается идеальных  
корутин не существует?**



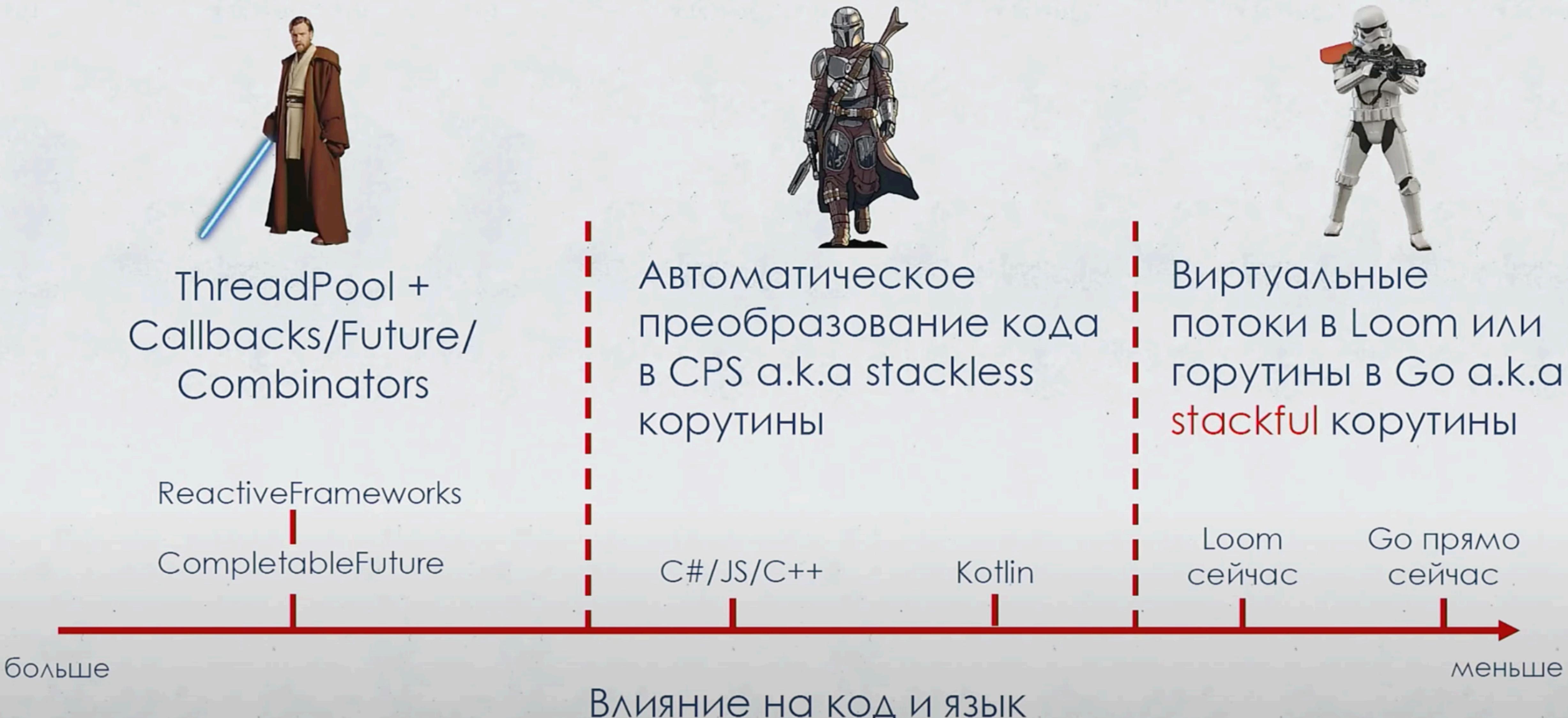
# BRAVE NEW WORLD

Горутины в Go:

- о Любую функцию можно запустить параллельно
- о Специальных модификаторов/раскраски – нет
- о Ограничений на код – нет
- о Миллион горутин создается и работает



# РЕШЕНИЯ ПРОБЛЕМЫ ТРЕДОВ



# **Итоги**

**Обмениваться данными между  
корутинаами легко и просто!**

**Материалы будут в README**

**Спасибо за внимание!**