Я тебя создал, я тебя и отменю

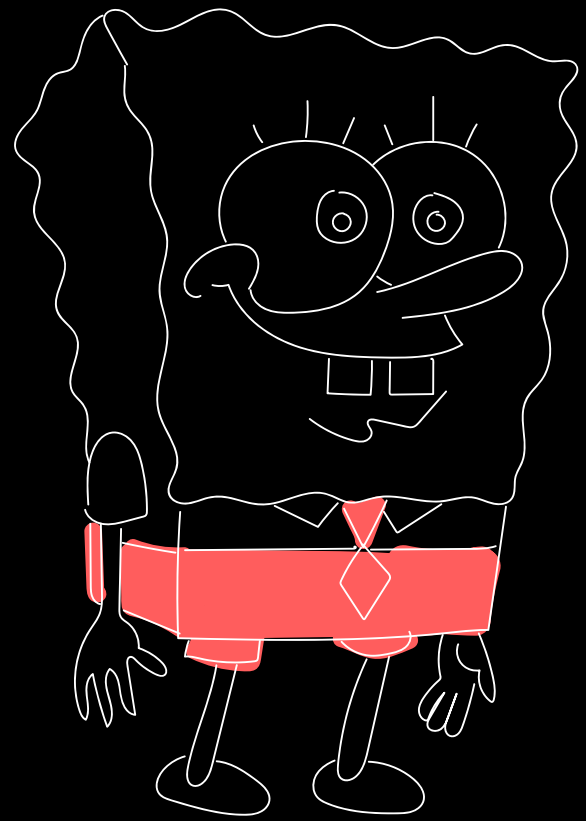
Рассмотрим как правильно отменять
корутины и для чего вообще это необходимо

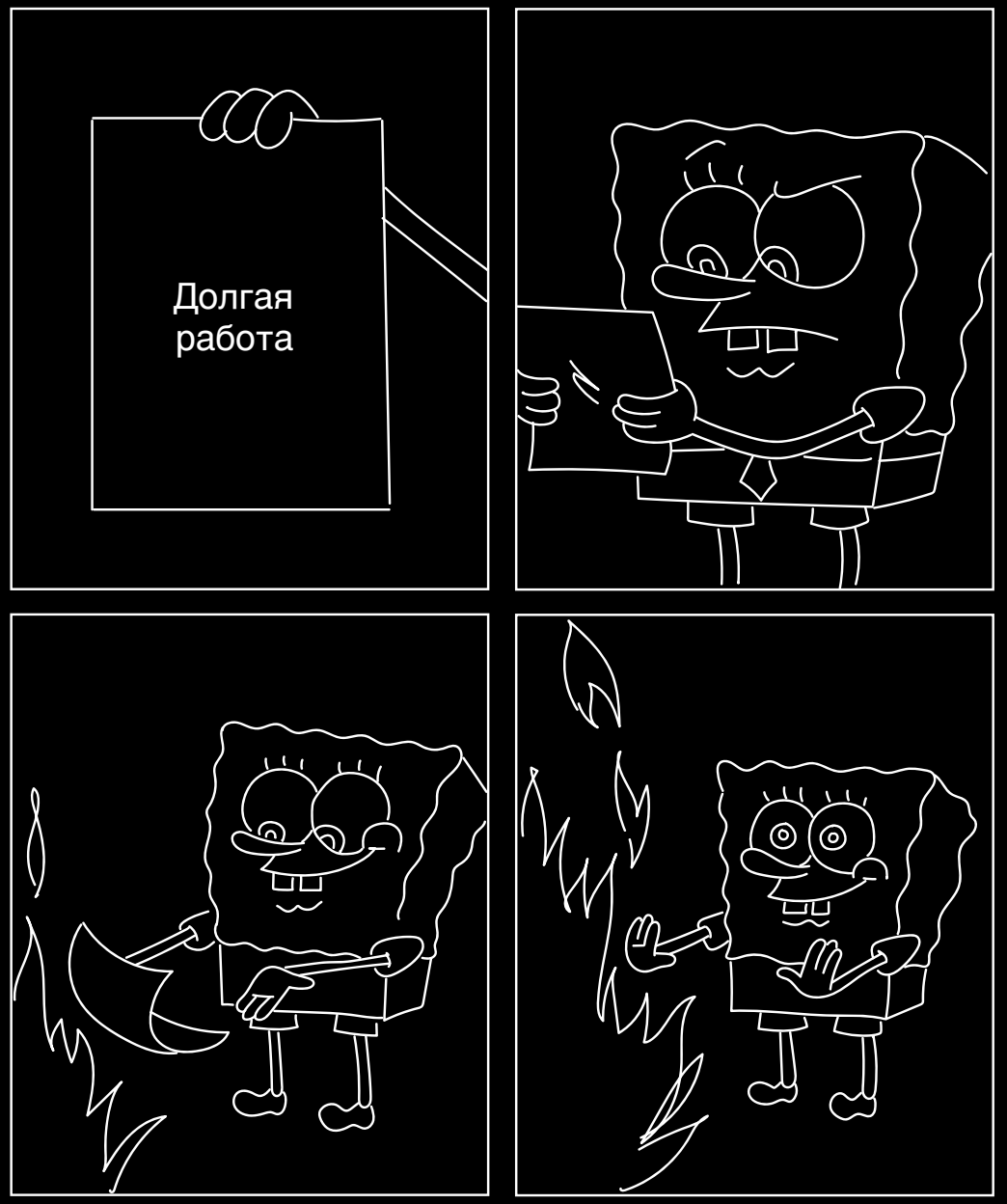
Ильичев Павел
Android developer

DELIVERY CLUB TECH

Программа

1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Что может пойти не так?




Долгая
работа

Программа

1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Взглянем на парочку примеров неправильной работы с корутинами.

Отменяем Thread()

```
fun main() {
    val thread = thread {
        while (true) {
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)



}
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
        while (true) {
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
        while (true) {
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

```
…
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I’m interrupted
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
…
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
      while(Thread.currentThread().isInterrupted.not()) {
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
      while(Thread.currentThread().isInterrupted.not()) {
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

```
…
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I’m interrupted


Process finished with exit code 0
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
    while(true) {
        Thread.sleep(100)
        println("I'm alive!")
      }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

Отменяем Thread()

```
…
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm alive!
I'm interrupted
Exception in thread "Thread-0" java.lang.InterruptedException: sleep interrupted
at java.lang.Thread.sleep(Native Method)
```

Отменяем Thread()

java.lang.Object#wait()
java.lang.Object#wait(long)
java.lang.Object#wait(long, int)
java.lang.Thread#sleep(long)

Отменяем Thread()

```
fun main() {
    val thread = thread {
        try {
            while (true) {
                Thread.sleep(100)
                println("I'm alive!")
            }
        } catch (error: InterruptedException) {
            // release resources
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.interrupt()
}
```

Отменяем Thread()

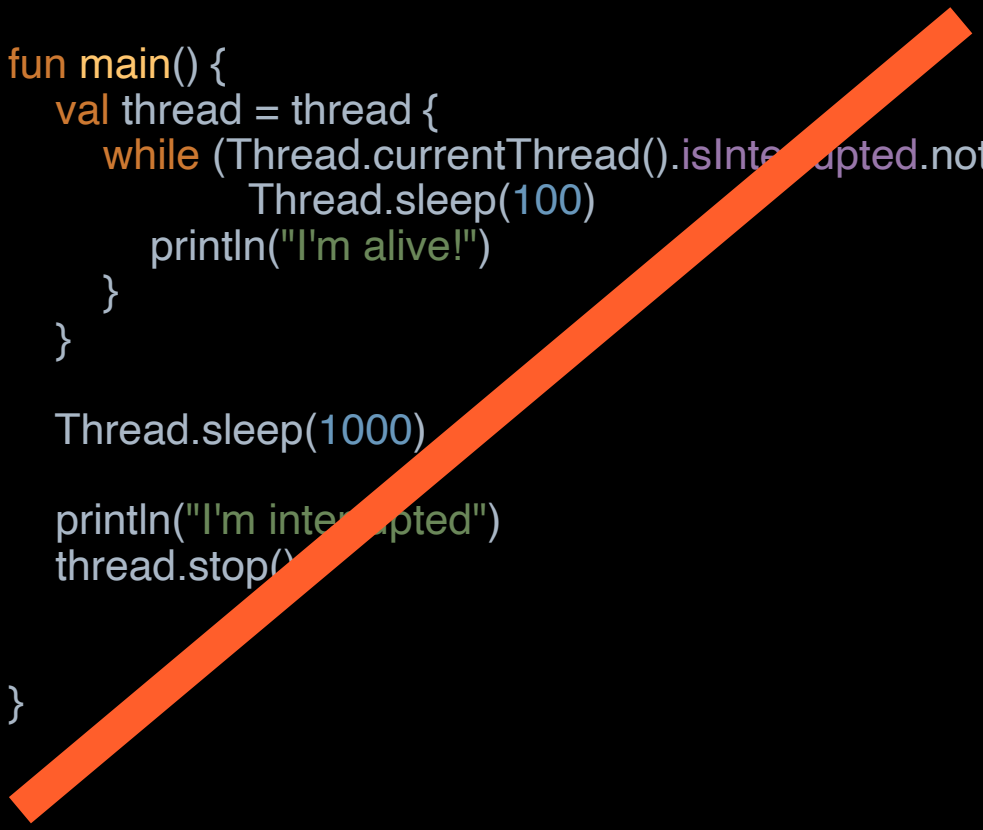
```
fun main() {
    val thread = thread {
      while(true) {
                    Thread.sleep(100)
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.stop()
}
```

Отменяем Thread()

```
@Deprecated
public final void stop() {
    /*
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        checkAccess();
        if (this != Thread.currentThread()) {
            security.checkPermission(SecurityConstants.STOP_THREAD_PERMISSION);
        }
    }
    // A zero status value corresponds to "NEW", it can't change to
    // not-NEW because we hold the lock.
    if (threadStatus != 0) {
        resume(); // Wake up thread if it was suspended; no-op otherwise
    }

    // The VM can handle all thread states
    stop0(new ThreadDeath());
    */
    throw new UnsupportedOperationException();
}
```

Отменяем Thread()

```
fun main() {
    val thread = thread {
        while (Thread.currentThread().isInterrupted.not()) {
                Thread.sleep(100)
            println("I'm alive!")
        }
    }

    Thread.sleep(1000)

    println("I'm interrupted")
    thread.stop()


}
```



Вывод

Главное правило:

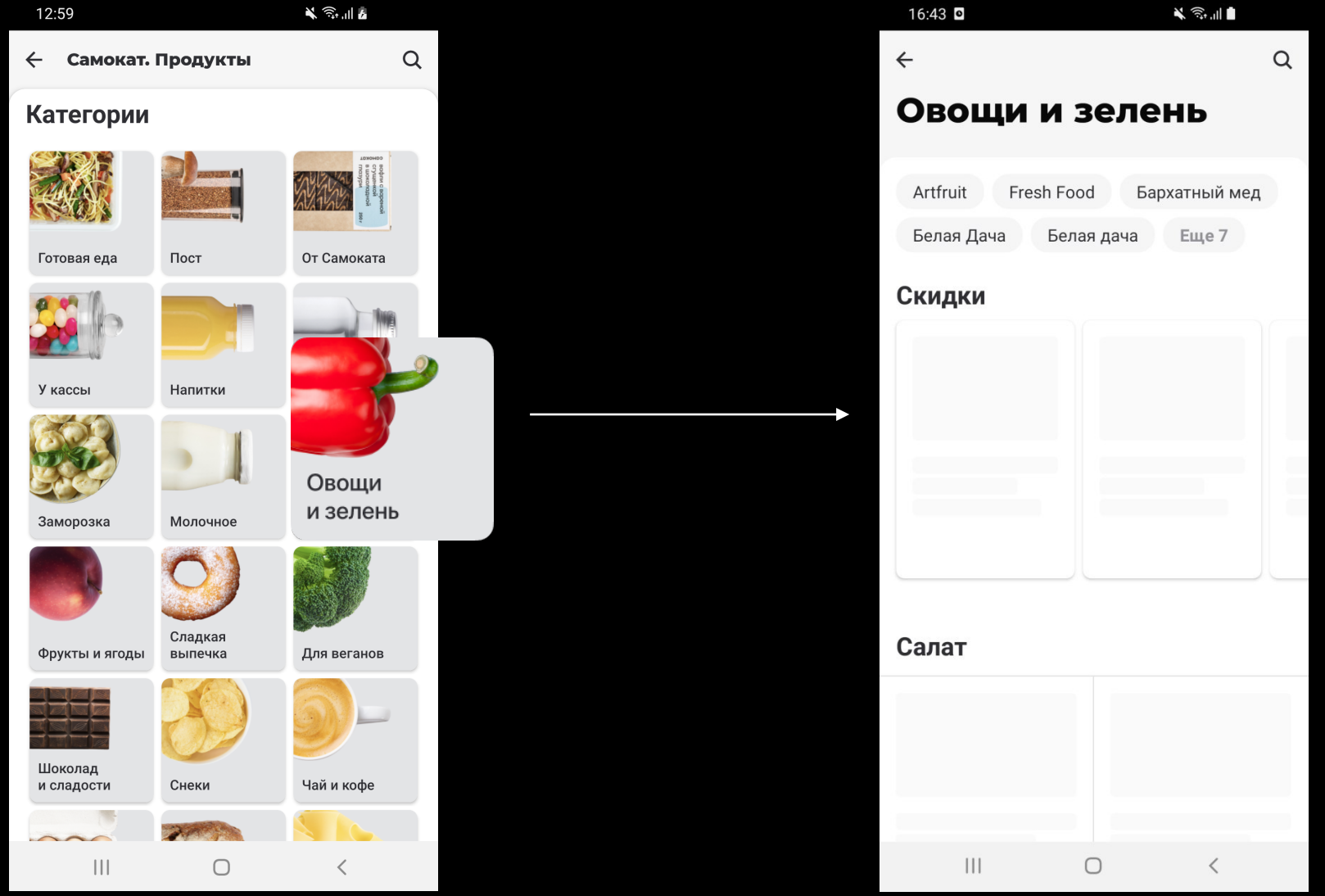
Процедура остановки потока должна быть управляемой.

Инструменты для управления:

1. Управление посредством специфичного флага.
2. Управление посредством специфичным исключением.

Программа

1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Что может пойти не так?


12:59
Самокат. Продукты
Категории
Готовая еда
Пост
От Самоката
У кассы
Напитки
Заморозка
Молочное
Овощи и зелень
Фрукты и ягоды
Сладкая выпечка
Для веганов
Шоколад и сладости
Снеки
Чай и кофе
16:43
Овощи и зелень
Artfruit
Fresh Food
Бархатный мед
Белая Дача
Белая дача
Еще 7
Скидки
Салат

scope.launch {}

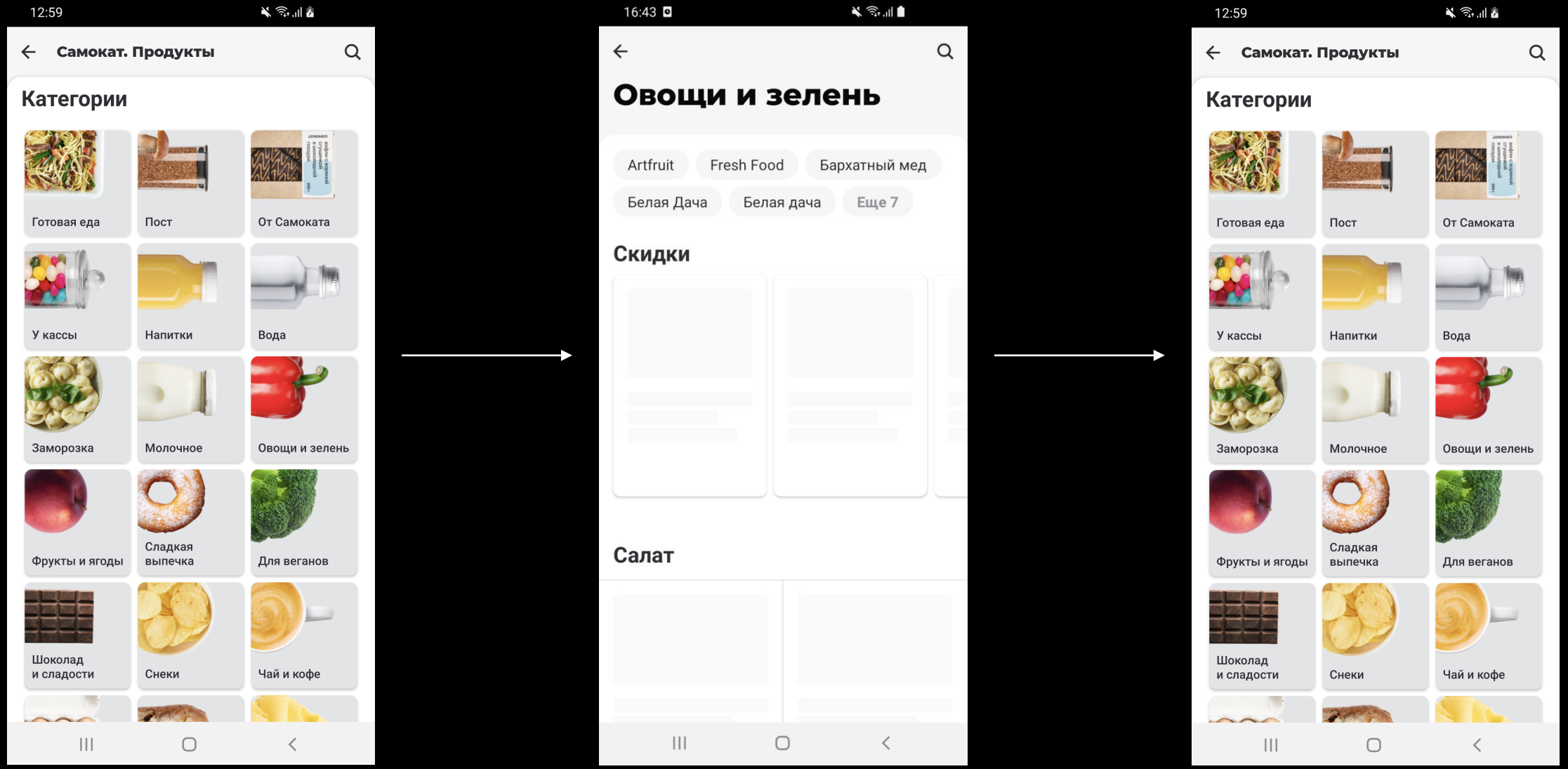
```
class CategoryViewModel : ViewModel() {

    init {
        viewModelScope.launch {
            productItems.forEach { product ->
                // do some heavy work
            }
        }
    }
}
```

scope.launch {}

```
class CategoryViewModel : ViewModel() {

    init {
        viewModelScope.launch {
            productItems.forEach { product ->
                // do some heavy work
            }
        }
    }
}
```

scope.launch {}

```
class CategoryViewModel : ViewModel() {

    init {
        viewModelScope.launch {
            productItems.forEach { product ->
                // do some heavy work
            }
        }
    }
}
```


12:59
Самокат. Продукты
Категории
Готовая еда
Пост
От Самоката
У кассы
Напитки
Вода
Заморозка
Молочное
Овощи и зелень
Фрукты и ягоды
Сладкая выпечка
Для веганов
Шоколад и сладости
Снеки
Чай и кофе
16:43
Овощи и зелень
Artfruit
Fresh Food
Бархатный мед
Белая Дача
Белая дача
Еще 7
Скидки
Салат
12:59
Самокат. Продукты
Категории
Готовая еда
Пост
От Самоката
У кассы
Напитки
Вода
Заморозка
Молочное
Овощи и зелень
Фрукты и ягоды
Сладкая выпечка
Для веганов
Шоколад и сладости
Снеки
Чай и кофе


Диагноз
Утечка памяти
memesmix.net

onDestroy()

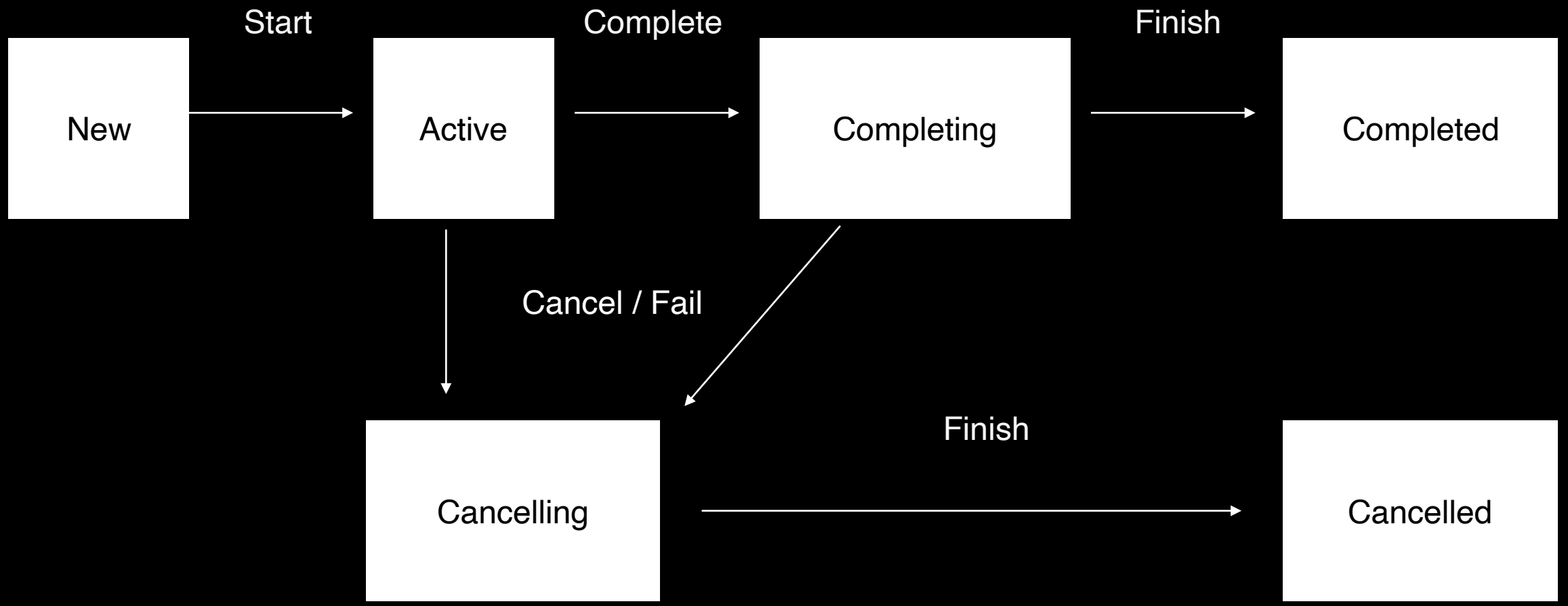
```
if (!isChangingConfigurations()) {

}
```

```
    getViewModelStore().clear()
```

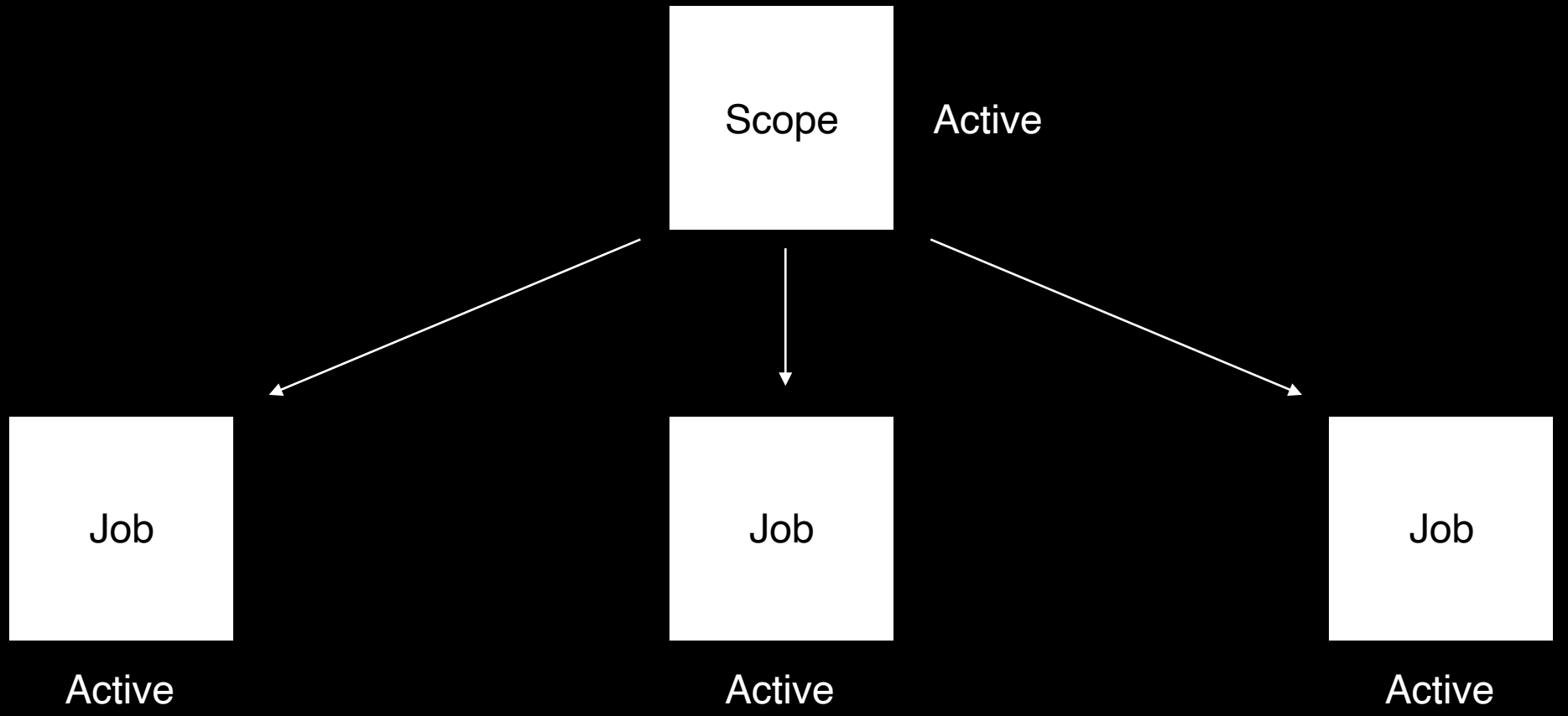
```
for (ViewModel vm : mMap.values()) {
    vm.clear();
}
```

```
coroutineContext[Job]?.cancel()
```

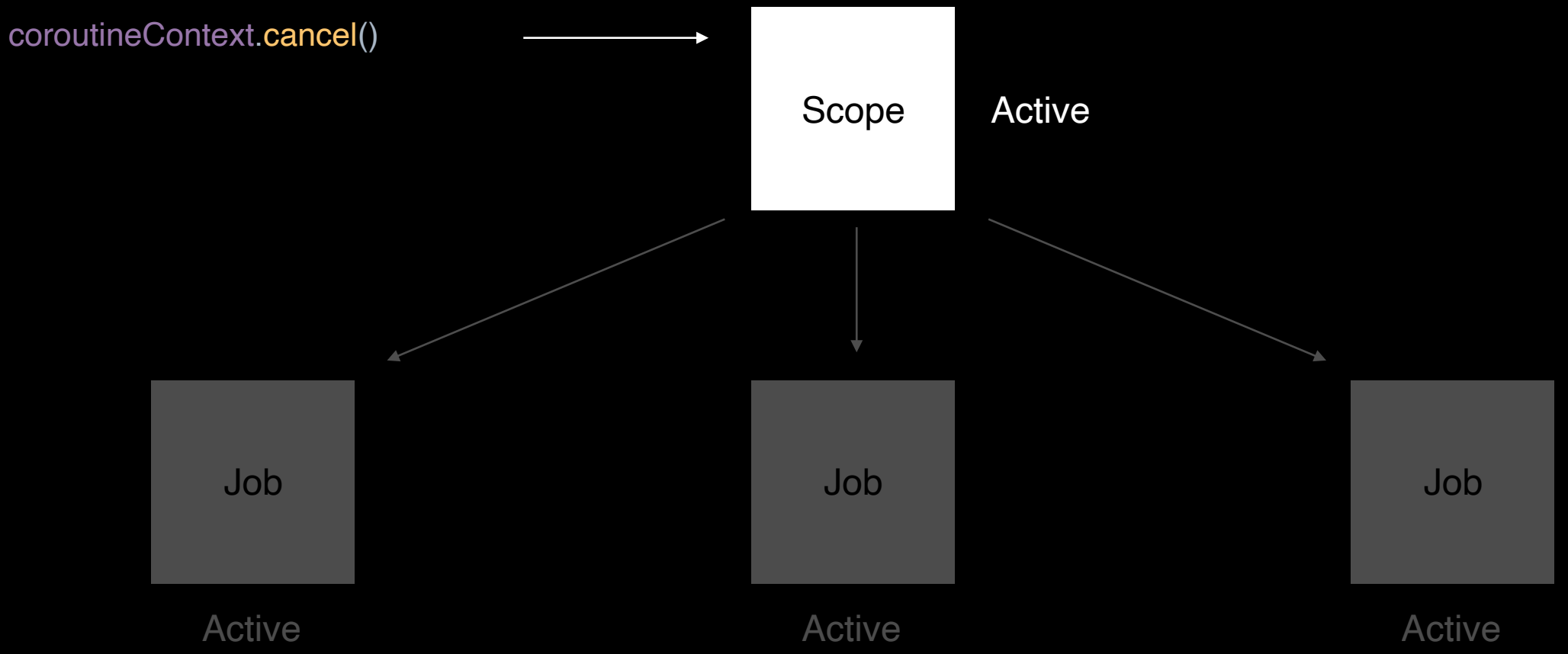
Job states


Start
Complete
Finish
New
Active
Completing
Completed
Cancel / Fail
Finish
Cancelling
Cancelled

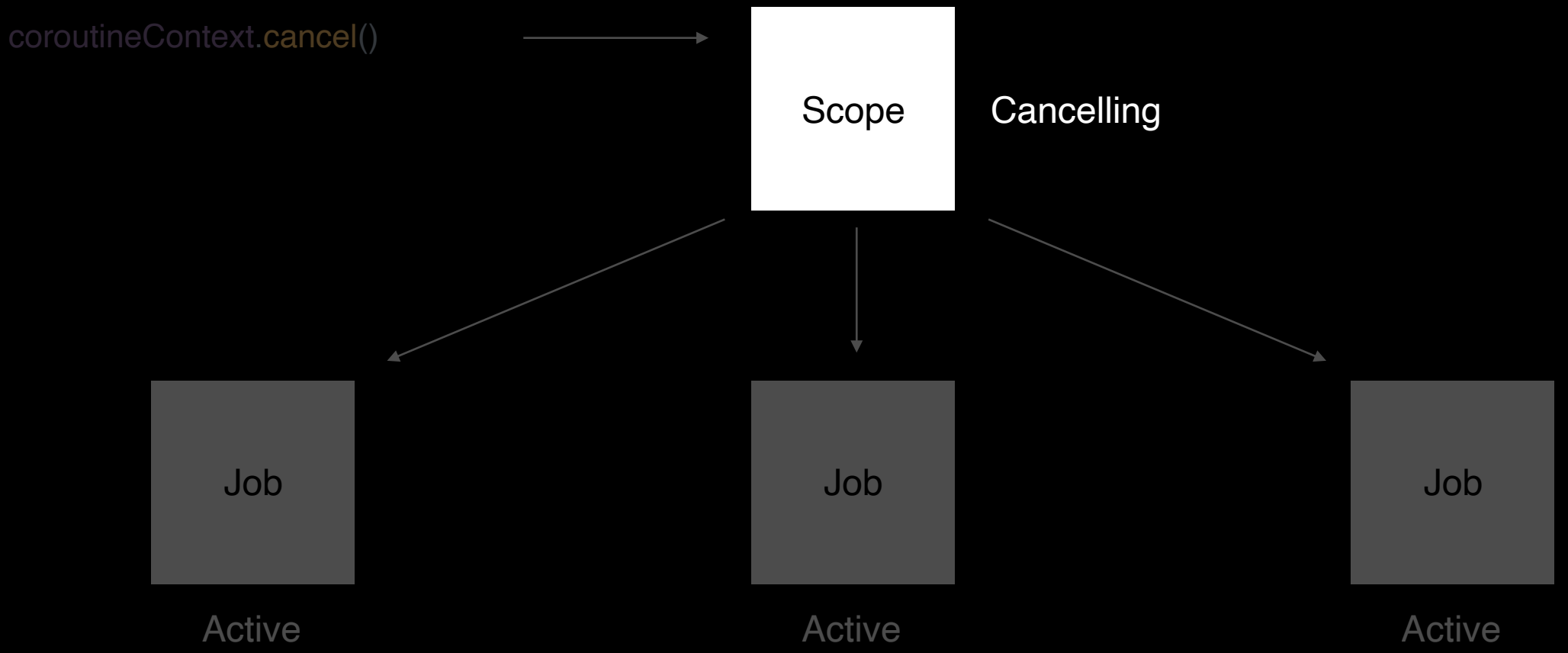
Job states



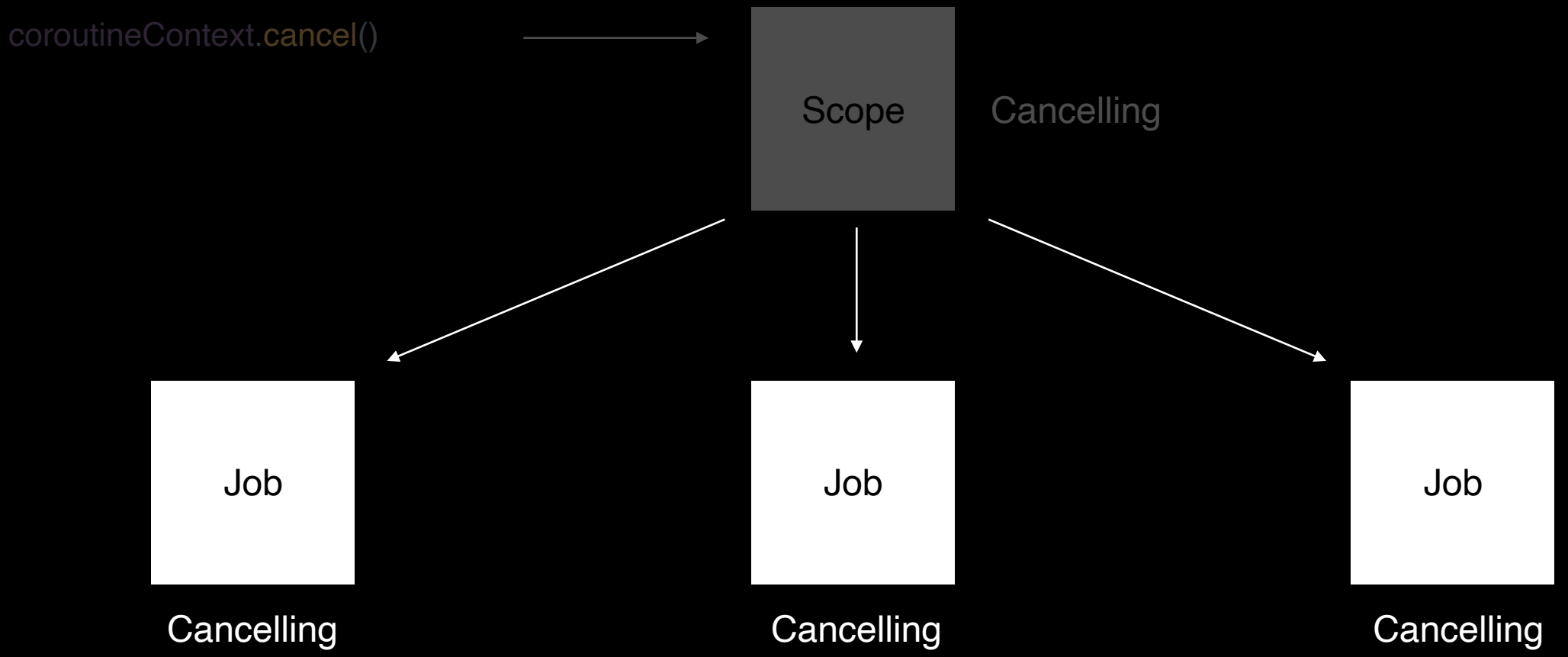
Job states


coroutineContext.cancel()
Scope
Active
Job
Job
Job
Active
Active
Active

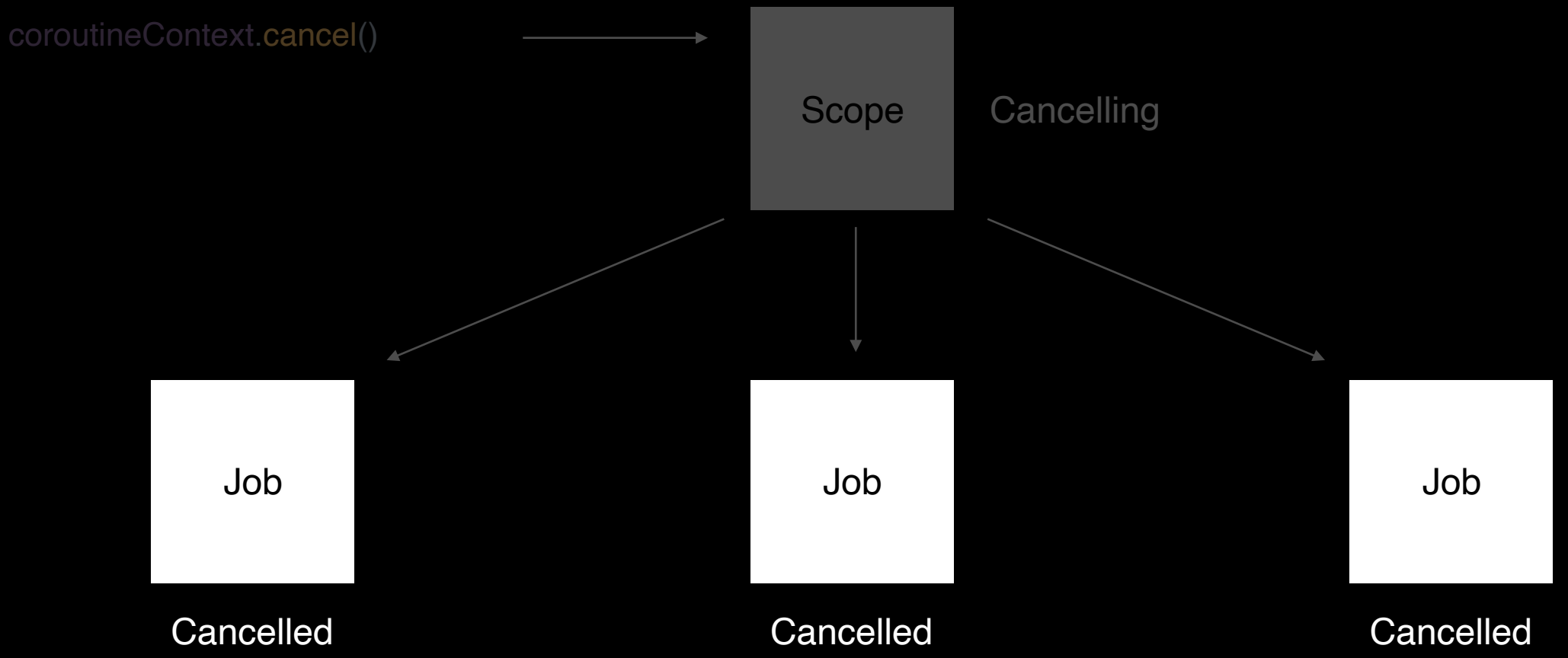
Job states


coroutineContext.cancel()
Scope
Cancelling
Job
Job
Job
Active
Active
Active

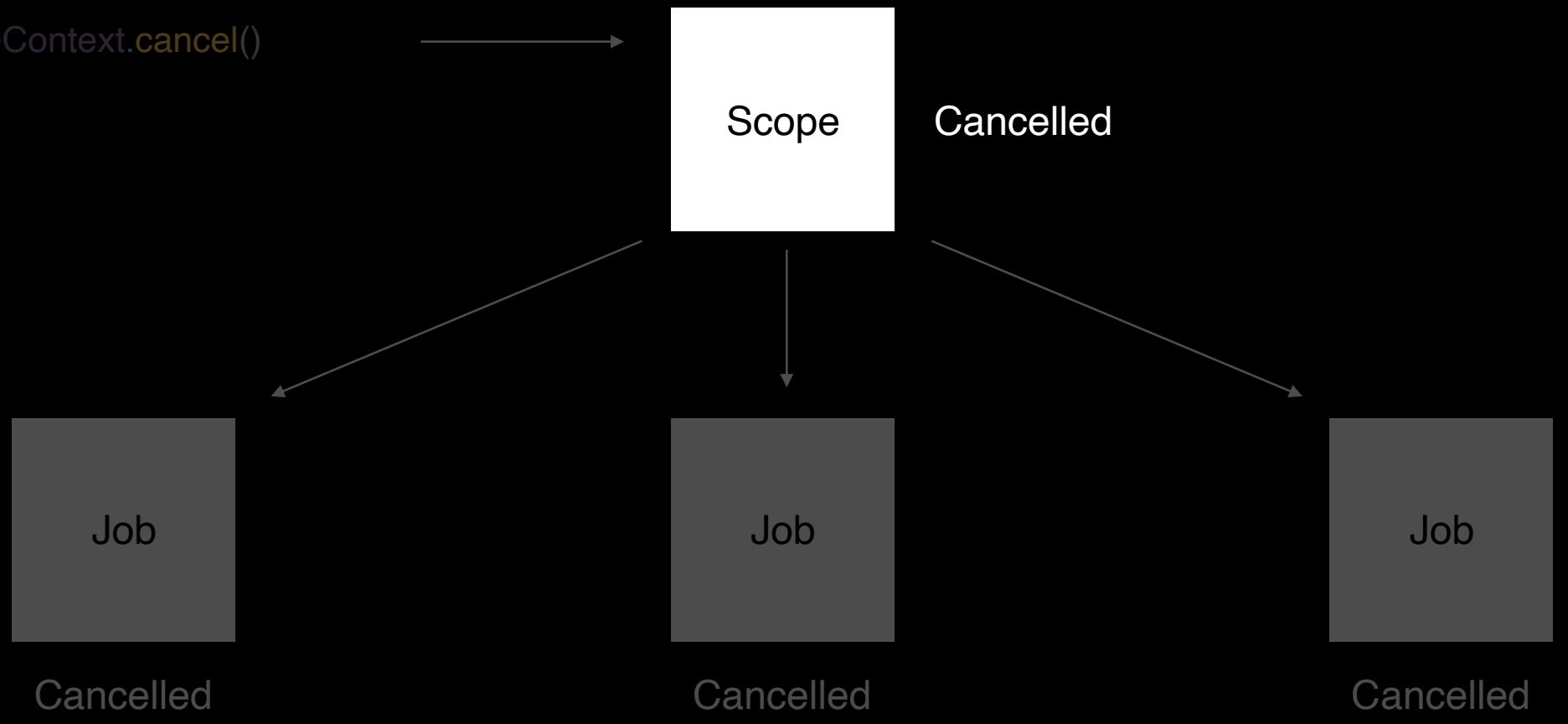
Job states


coroutineContext.cancel()
Scope
Cancelling
Job
Job
Job
Cancelling
Cancelling
Cancelling

Job states


coroutineContext.cancel()
Scope
Cancelling
Job
Job
Job
Cancelled
Cancelled
Cancelled

Job states


coroutineContext.cancel()
Scope
Cancelled
Job
Job
Job
Cancelled
Cancelled
Cancelled

Job flags

State	isActive	isCompleted	isCancelled
New (optional initial state)	`false`	`false`	`false`
Active (default initial state)	`true`	`false`	`false`
Completing (transient state)	`true`	`false`	`false`
Cancelling (transient state)	`false`	`false`	`true`
Cancelled (final state)	`false`	`true`	`true`
Completed (final state)	`false`	`true`	`false`

scope.launch {}

```
class CategoryViewModel : ViewModel() {

    init {
            viewModelScope.launch {
                productItems.forEach { product ->
                    if (isActive.not()) return@launch
                    // do some heavy work
                }
            }
        }
}
```

scope.launch {}

```
class CategoryViewModel : ViewModel() {

    init {
            viewModelScope.launch {
                productItems.forEach { product ->
                    if (isActive.not()) return@launch
                    // do some heavy work
                }
            }
        }
}
```

scope.cancel()

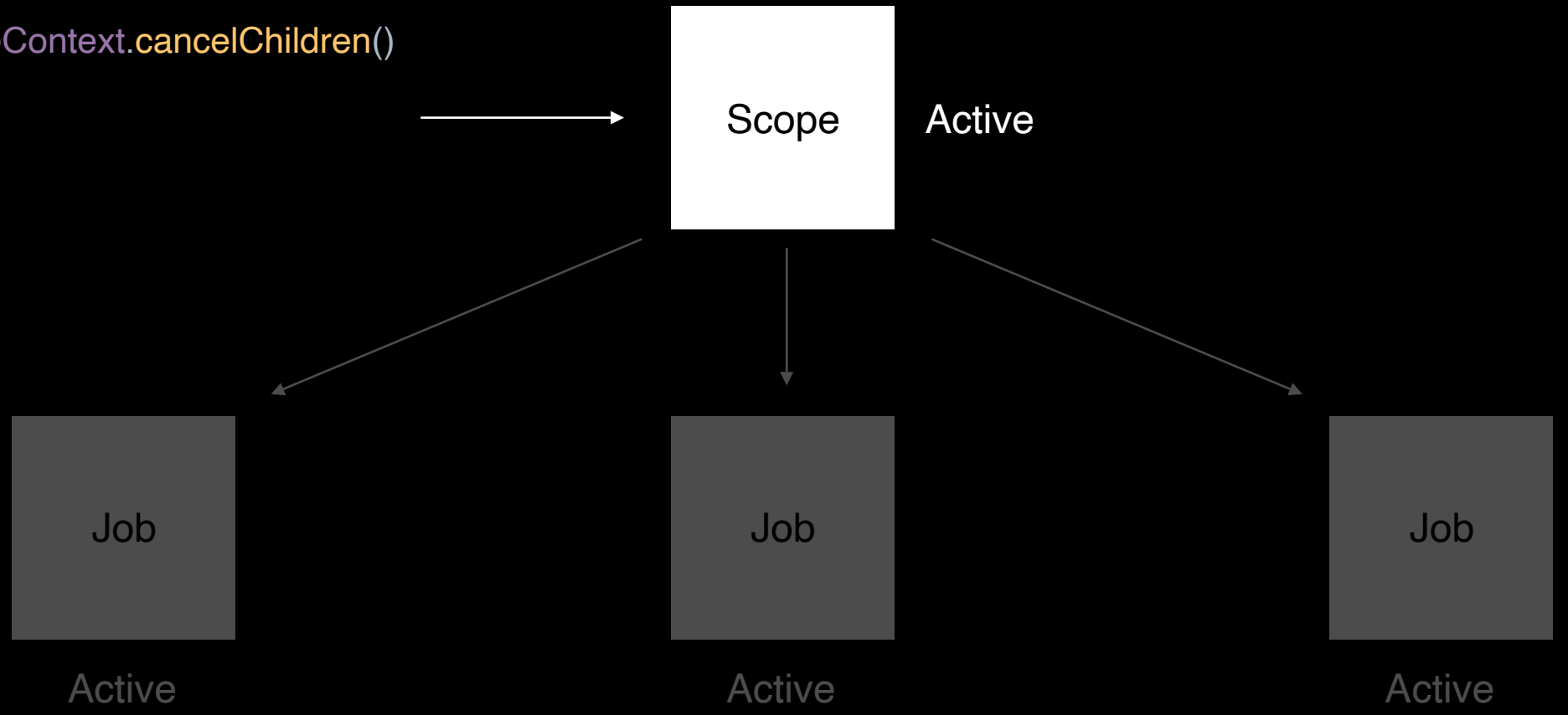
1. Отменяются все дочерние джобы.
2. Скоуп переходит в состояние Cancelled, что не дает возможности его дальнейшего использования.

CoroutineContext.cancelChildren()

```
class CategoryViewModel : ViewModel() {

    init {
        viewModelScope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
                // do some heavy work
            }
        }
    }

    fun cancelHandleData() {
        viewModelScope.coroutineContext.cancelChildren()
    }
}
```

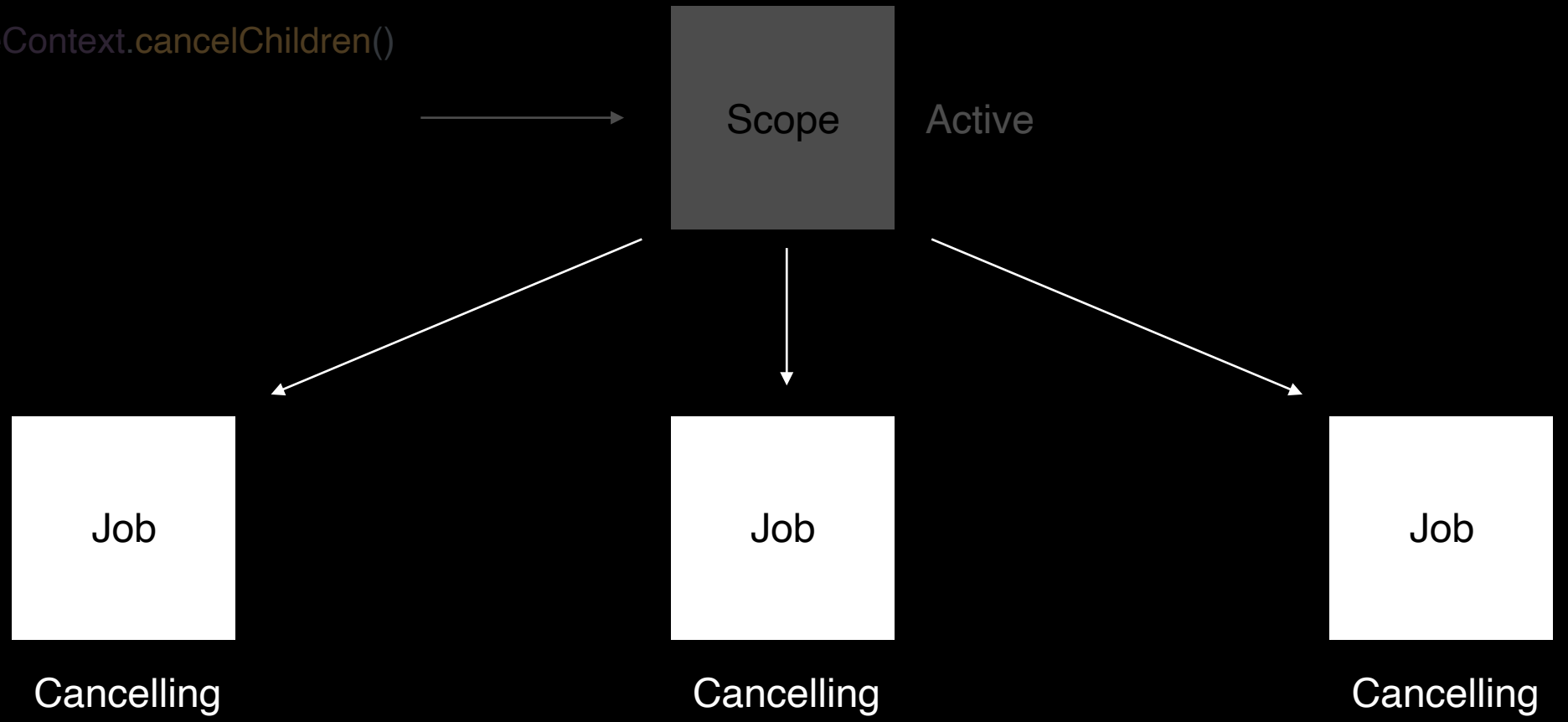
Job states

```
coroutineContext.cancelChildren()
```



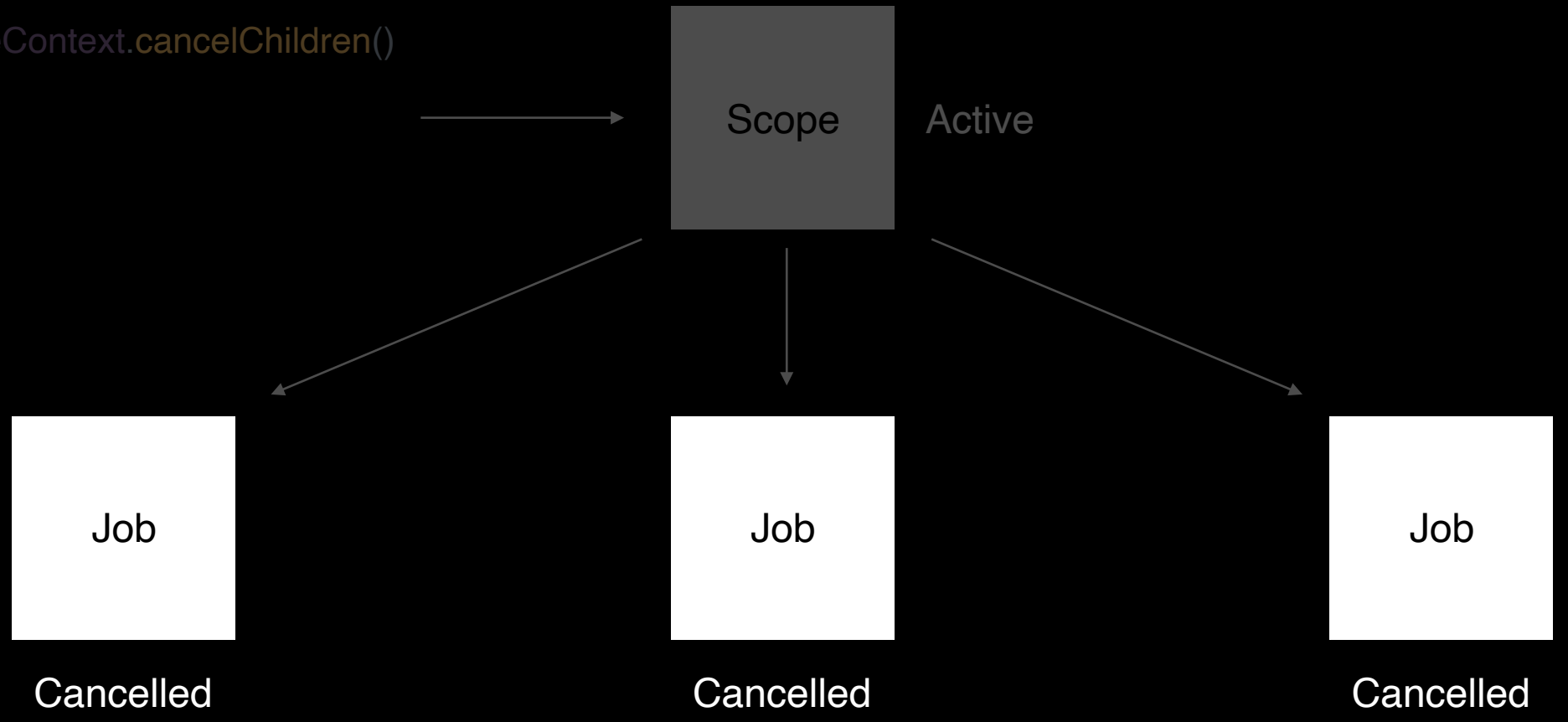
Job states

coroutineContext.cancelChildren()



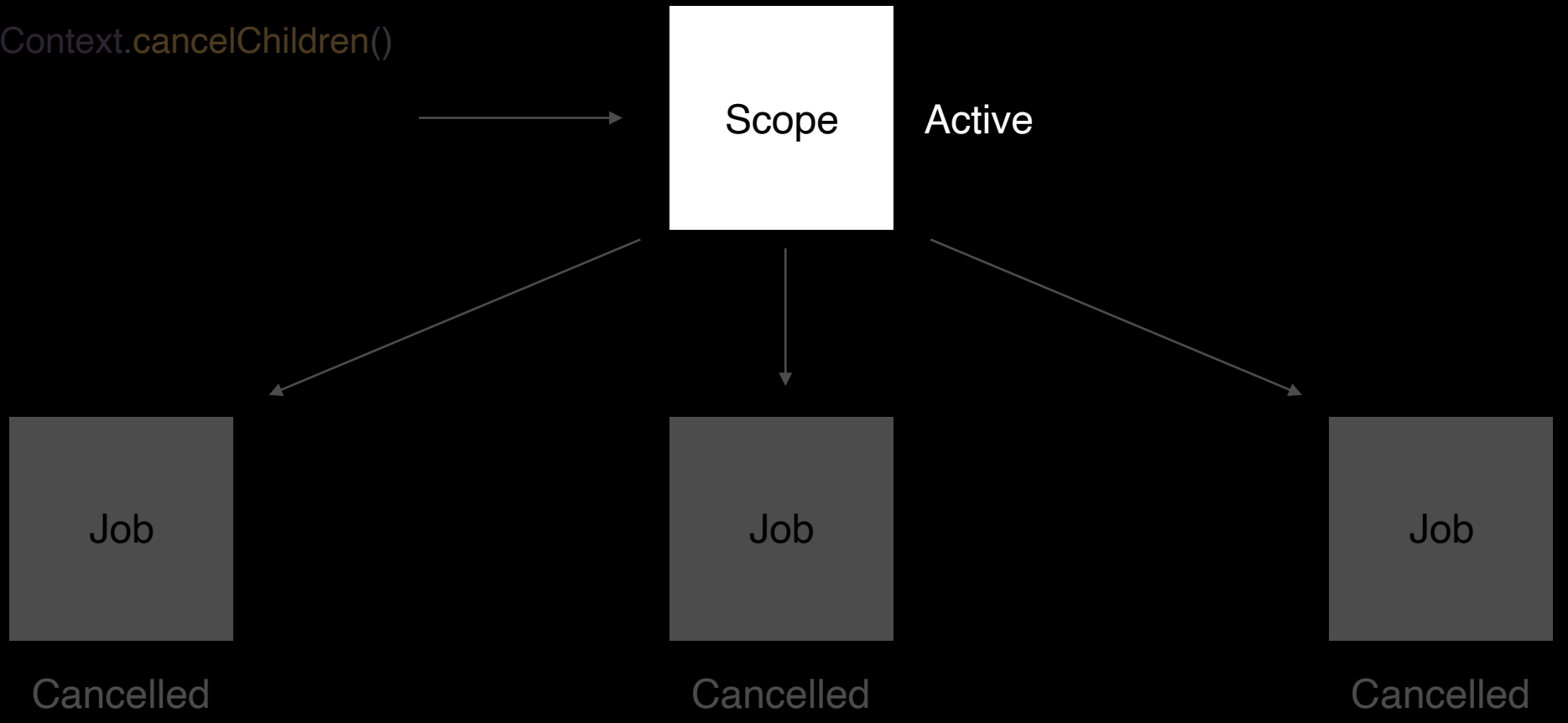
Job states

coroutineContext.cancelChildren()



Job states

coroutineContext.cancelChildren()



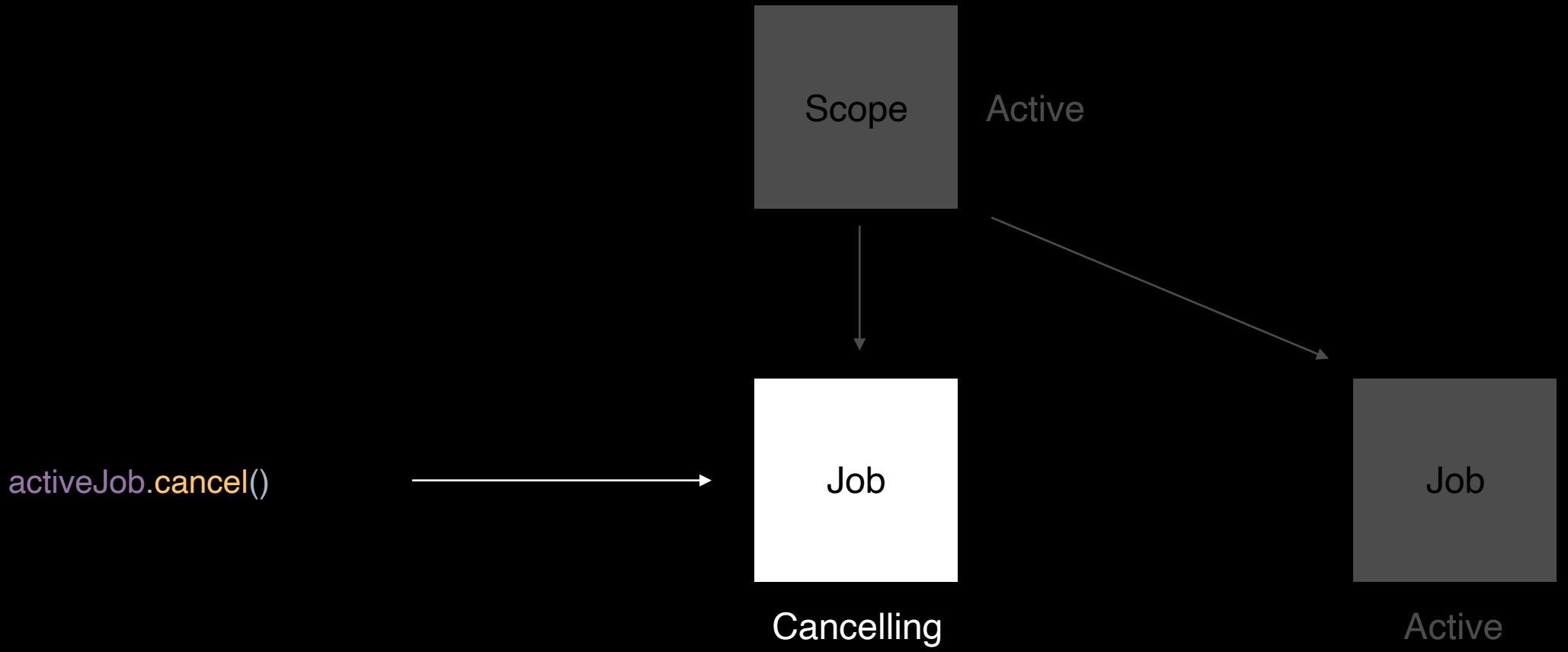
CoroutineContext.cancelChildren()

```
class CategoryViewModel : ViewModel() {

    init {
        viewModelScope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
                // do some heavy work
            }
        }
    }

    fun cancelHandleData() {
        viewModelScope.coroutineContext.cancelChildren()
    }
}
```

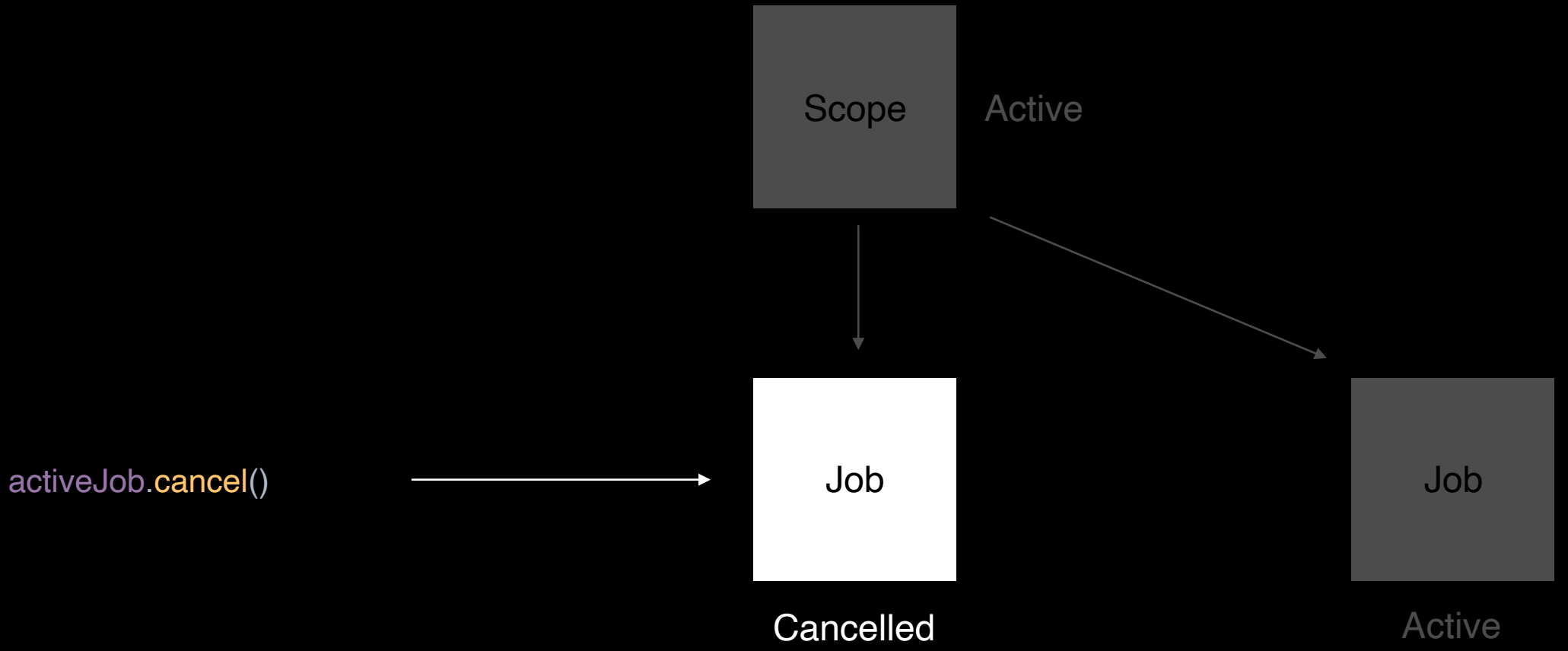
Job.cancel()

```
class CategoryViewModel : ViewModel() {

    private var activeJob: Job? = null

    init {
        activeJob = viewModelScope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
                // do some heavy work
            }
        }
    }

    fun cancelHandleData() {
        activeJob?.cancel()
        activeJob = null
    }
}
```

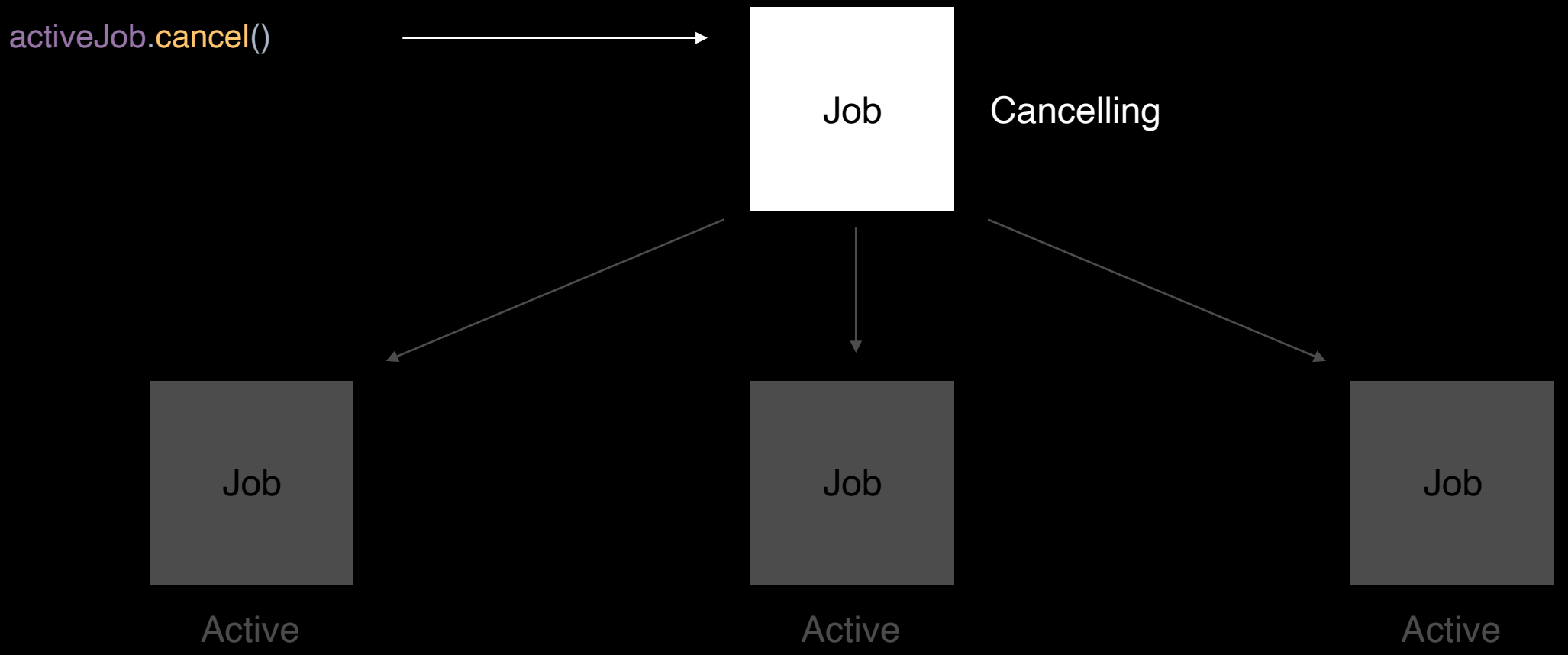
Job states



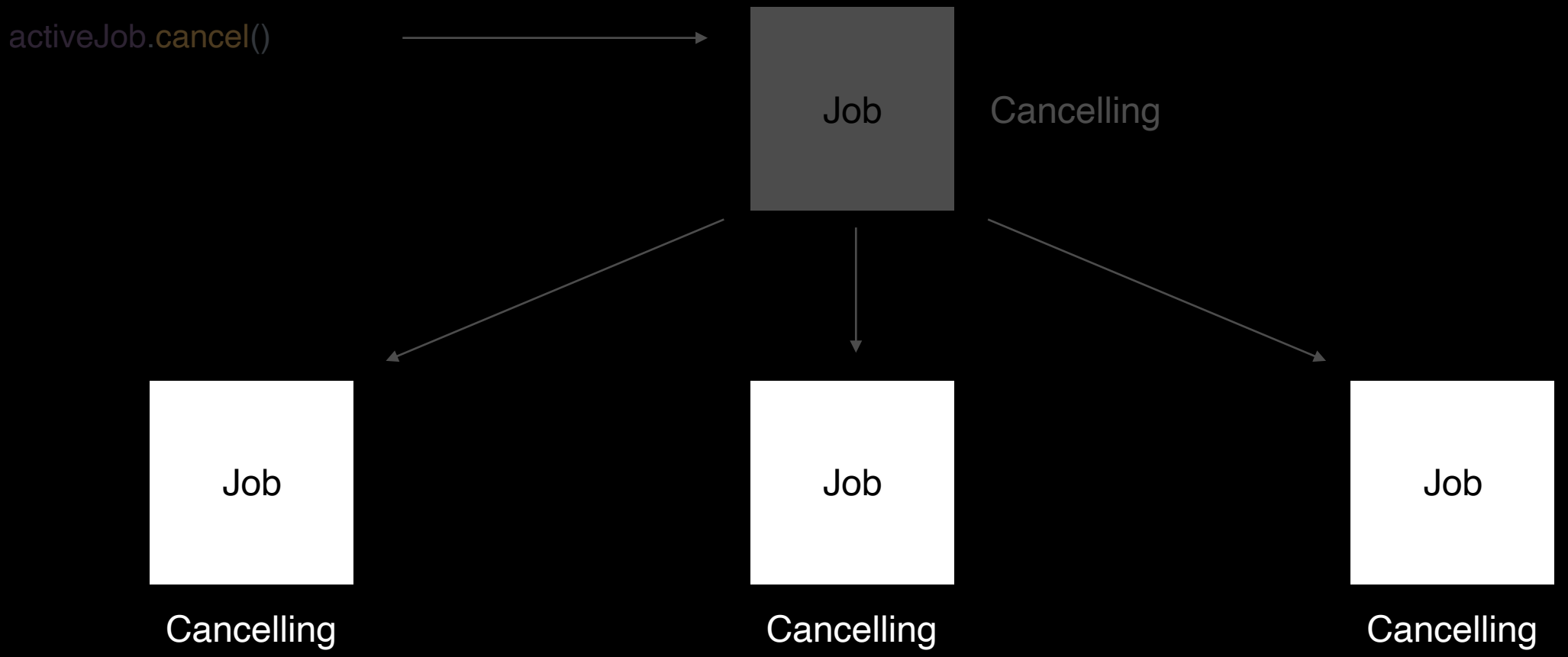
Job states



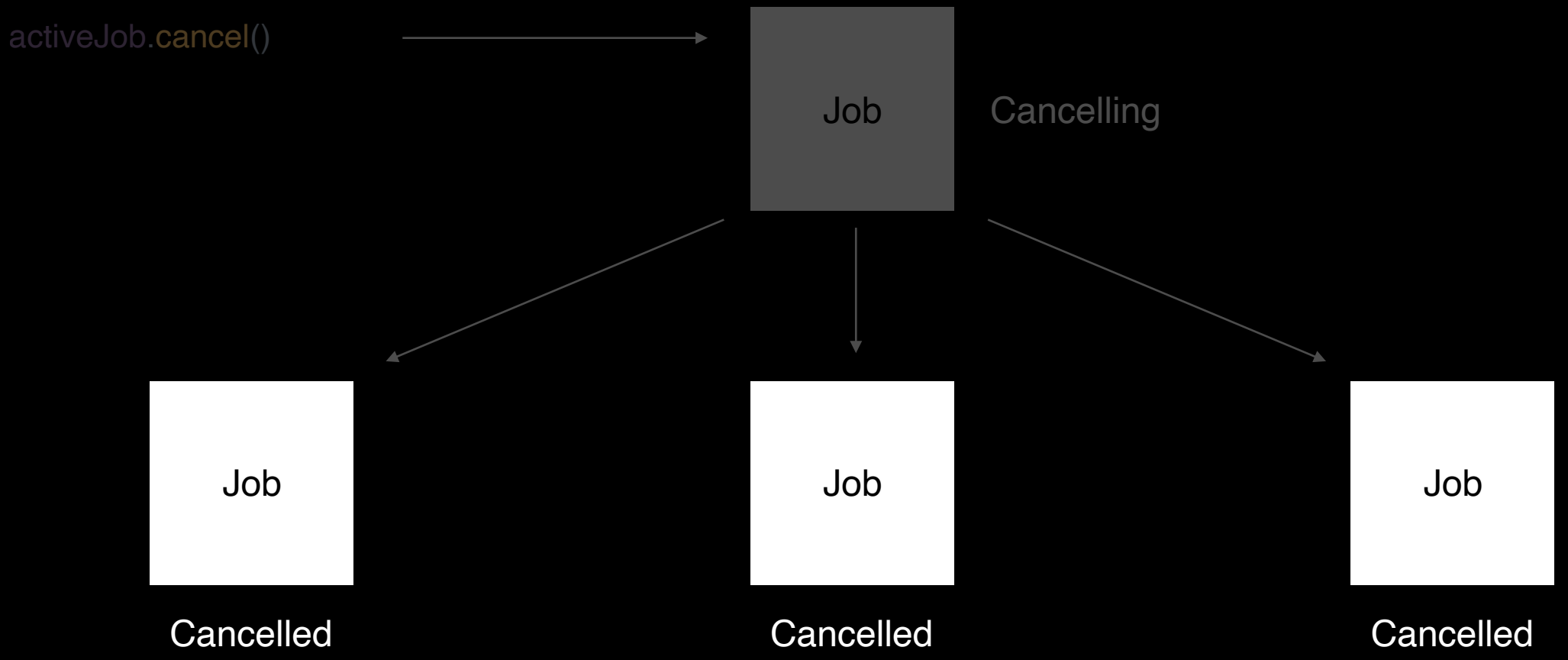
Job states


activeJob.cancel()
Job
Cancelling
Job
Job
Job
Active
Active
Active

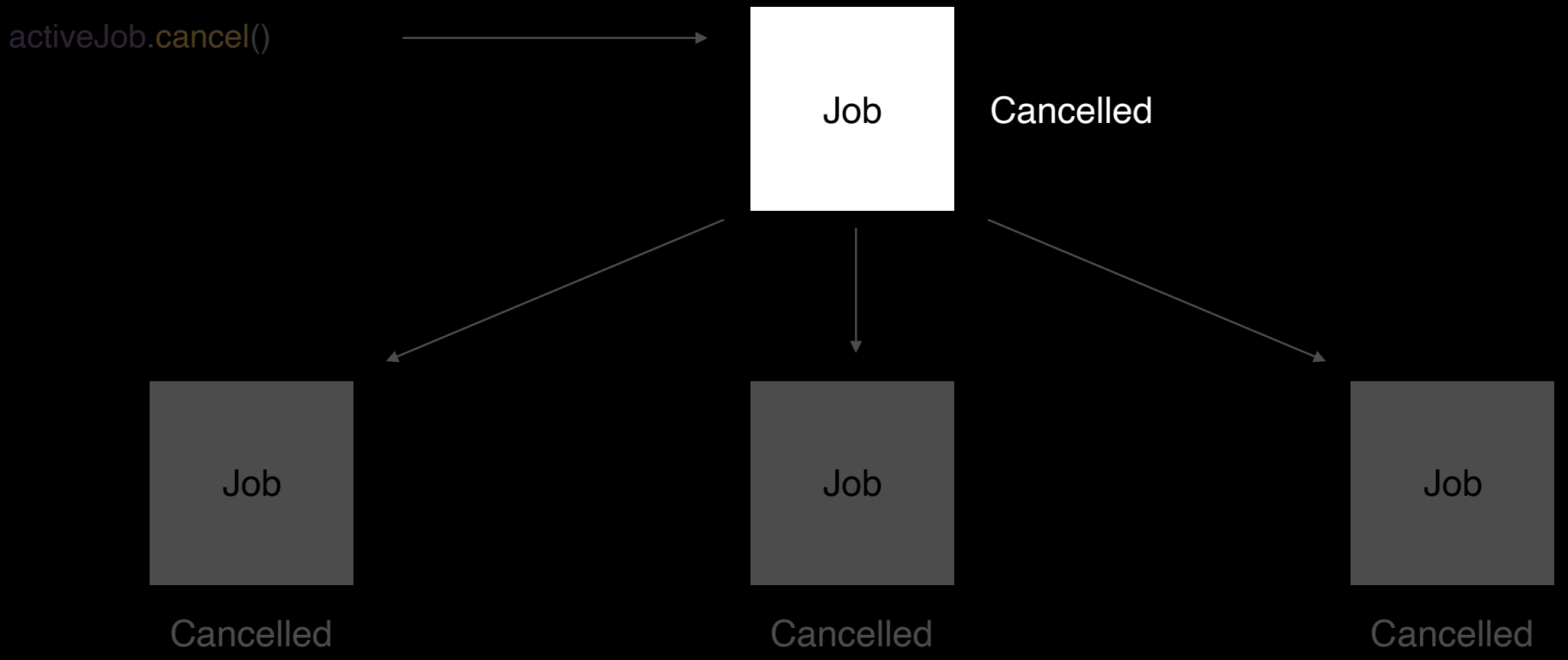
Job states


activeJob.cancel()
Job
Cancelling
Job
Job
Job
Cancelling
Cancelling
Cancelling

Job states


activeJob.cancel()
Job
Cancelling
Job
Job
Job
Cancelled
Cancelled
Cancelled

Job states


activeJob.cancel()
Job
Cancelled
Job
Job
Job
Cancelled
Cancelled
Cancelled

Job.cancel()

```
class StoreViewModel : ViewModel() {

    private var activeJob: Job? = null

    init {
        activeJob = viewModelScope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
                // do some heavy work
            }
        }
    }

    fun cancelHandleData() {
        activeJob?.cancel()
        activeJob = null
    }
}
```

ensureActive()

```
class StoreViewModel : ViewModel() {

    init {
            viewModelScope.launch {
                productItems.forEach { product ->
                    ensureActive()
                    // do some heavy work
                }
            }
        }
}
```

ensureActive()

```
public fun Job.ensureActive(): Unit {
    if (!isActive) throw getCancellationException()
}
```

ensureActive()

```
class StoreViewModel : ViewModel() {

    init {
            viewModelScope.launch {
                productItems.forEach { product ->
                    ensureActive()
                    // do some heavy work
                }
            }
        }
}
```

```
fun cancel() {
    scope.cancel()
}
```

ensureActive()

```
class StoreViewModel : ViewModel() {

    init {
            viewModelScope.launch {
                productItems.forEach { product ->
                    ensureActive()
                    // do some heavy work
                }
            }
        }
}
```

```
fun cancel() {
    scope.cancel()
}
```

thow CancellationException()

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
        productItems.forEach { product ->
            if (isActive.not()) {
                throw CancellationException()
            }
        }
    }
}
```

Как отменить корутину

1. Scope.cancel() - отменить скоуп и все дочерние элементы.
2. Scope.coroutineContext.cancelChildren() - отменить только дочерние элементы, скоуп остается активным.
3. Job.cancel() - отменить выбранную корутину.

Опции закончить работу корутины

1. Проверка на статус и локальный return.
2. Закончить работу с прокидыванием CancellationException().
3. Воспользоваться экстеншеном ensureActive().

try {} catch() {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
        try {
            productItems.forEach { product ->
                if (isLoadActual.not()) {
                    throw CancellationException()
                }
            }
        } catch (error: CancellationException) {
            // release resources
        }
    }
}
```

try {} catch() {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
            try {
                productItems.forEach { product ->
                    if (isLoadActual.not()) {
                        throw CancellationException()
                    }
                }
            } catch (error: CancellationException) {
                // release resources
            }
        }
}
```

try {} catch() {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
            try {
                productItems.forEach { product ->
                    if (isActive.not()) return@launch
                }
            } catch (error: CancellationException) {
                // release resources
            }
        }
}


 fun cancelWork() {
    scope.cancel()
}
```

try {} catch() {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
            try {
                productItems.forEach { product ->
                    if (isActive.not()) return@launch
                }
            } catch (error: CancellationException) {
                // release resources
            }
        }
}

 fun cancelWork() {
     scope.cancel()
 }
```

Job.cancel()

```
/**
 * Cancels this scope, including its job and all its children with an optional cancellation [cause].
 * A cause can be used to specify an error message or to provide other details on
 * a cancellation reason for debugging purposes.
 * Throws [IllegalStateException] if the scope does not have a job in it.
 */
public fun CoroutineScope.cancel(cause: CancellationException? = null)
```

CoroutineExceptionHandler {}

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    (error as? CancellationException)?.let {
        // release resources
    }
})

fun prepareData() {
    scope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
            }
        }
}



fun cancelWork() {
    scope.cancel()
}
```

CoroutineExceptionHandler {}

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    (error as? CancellationException)?.let {
        // release resources
    }
})

fun prepareData() {
    scope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
            }
        }
}



fun cancelWork() {
    scope.cancel()
}
```

CoroutineExceptionHandler {}

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    (error as? CancellationException)?.let {
        // release resources
    }
})

fun prepareData() {
    scope.launch {
            productItems.forEach { product ->
                if (isActive.not()) return@launch
            }
        }
}



fun cancelWork() {
    scope.cancel()
}
```

CoroutineExceptionHandler {}

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    (error as? CancellationException)?.let {
        // release resources
    }
})

fun prepareData() {
    scope.launch {
            productItems.forEach { product ->
                if (isLoadActual.not()) {
                    throw CancellationException()
                }
            }
        }
}
```

CoroutineExceptionHandler {}

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    (error as? CancellationException)?.let {
        // release resources
    }
})

fun prepareData() {
    scope.launch {
            productItems.forEach { product ->
                if (isLoadActual.not()) {
                    throw CancellationException()
                }
            }
        }
}
```

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
            // release resources
        }
    }
}
```

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
        scope.coroutineContext[Job]?.invokeOnCompletion { error ->
            // clear resources
        }

    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
                // clear resources
            }
    }
}
```

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
        scope.coroutineContext[Job]?.invokeOnCompletion { error ->
                (error as? CancellationException)?.let {
                    // clear resources
                    }
        }

    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
                    (error as? CancellationException)?.let {
                    // clear resources
                    }
                }
    }
}
```

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
        scope.coroutineContext[Job]?.invokeOnCompletion { error ->
                (error as? CancellationException)?.let {
                    // clear resources
                    }
        }

    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
                    (error as? CancellationException)?.let {
                    // clear resources
                    }
                }
    }
}
```

1

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
        scope.coroutineContext[Job]?.invokeOnCompletion { error ->
                (error as? CancellationException)?.let {
                    // clear resources
                    }
        }

    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
                    (error as? CancellationException)?.let {
                    // clear resources
                    }
                }
    }
}
```

2

1

Job.invokeOnCompleiton {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun prepareData() {
        scope.coroutineContext[Job]?.invokeOnCompletion { error ->
                (error as? CancellationException)?.let {
                    // clear resources
                    }
        }

    scope.launch {
        productItems.forEach { product ->
            // do some heavy work
        }
    }.also {
        it.invokeOnCompletion { error ->
                    (error as? CancellationException)?.let {
                    // clear resources
                    }
                }
    }
}
```

Вывод

Опции определить отмену корутины

1. Try catch – отловить отмену локально.
2. InvokeOnCompletion – отловить общую отмену корутины / скоупа.

Программа

1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Что может пойти не так?

delay()

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        while (true) {
            delay(5000)
            syncData()
        }
    }
}
```

delay()

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        while (true) {
            delay(5000)
            syncData()
        }
    }
}
```

delay()

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        while (true) {
            delay(5000)
            syncData()
        }
    }
}

private fun cancel() {
    scope.cancel()
}
```

delay()

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        while (true) {
            delay(5000)
            syncData()
        }
    }
}

private fun cancel() {
    scope.cancel()
}
```

delay()

```
public suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // don't delay
    return suspendCancellableCoroutine sc@ { cont: CancellableContinuation<Unit> ->
        . . .
    }
}
```

suspendCancellableCoroutine()

```
/**
 * Suspends the coroutine like [suspendCoroutine], but providing a [CancellableContinuation] to
 * the [block]. This function throws a [CancellationException] if the [Job] of the coroutine is
 * cancelled or completed while it is suspended.
 * . . .
 **/

public suspend inline fun <T> suspendCancellableCoroutine(
    crossinline block: (CancellableContinuation<T>) -> Unit
): T =
    suspendCoroutineUninterceptedOrReturn { uCont ->
            . . .
    }
```

suspendCancellableCoroutine()

[join][Job.join]
[await][Deferred.await]
[lock][Mutex.lock]
[delay]

try {} finally {}

```
val scope = CoroutineScope(Dispatchers.IO)

fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } catch (error: CancellationException) {
            // release resources
        }
    }
}
```

try {} finally {}

```
val scope = CoroutineScope(Dispatchers.IO)

fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } finally {
            // release resources
        }
    }
}
```

try {} finally {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } finally {
            delay(1000)
            // release resources
        }
    }
}
```

try {} finally {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } finally {
            delay(1000)
            // release resources
        }
    }
}
```

withContext(NonCancellable) {}

```
public object NonCancellable : AbstractCoroutineContextElement(Job), Job {
    override val isActive: Boolean = true
    override val isCompleted: Boolean = false
    override val isCancelled: Boolean = false
    // other properties and functions
}
```

withContext(NonCancellable) {}

```
public object NonCancellable : AbstractCoroutineContextElement(Job), Job {
    override val isActive: Boolean = true
    override val isCompleted: Boolean = false
    override val isCancelled: Boolean = false
    // other properties and functions
}
```

withContext(NonCancellable) {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } finally {
            withContext(NonCancellable) {
                delay(1000)
                // release resources
            }
        }
    }
}
```

withContext(NonCancellable) {}

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSyncData() {
    scope.launch {
        try {
            while (true) {
                delay(5000)
                syncData()
            }
        } finally {
            withContext(NonCancellable) {
                delay(1000)
                // release resources
            }
        }
    }
}
```

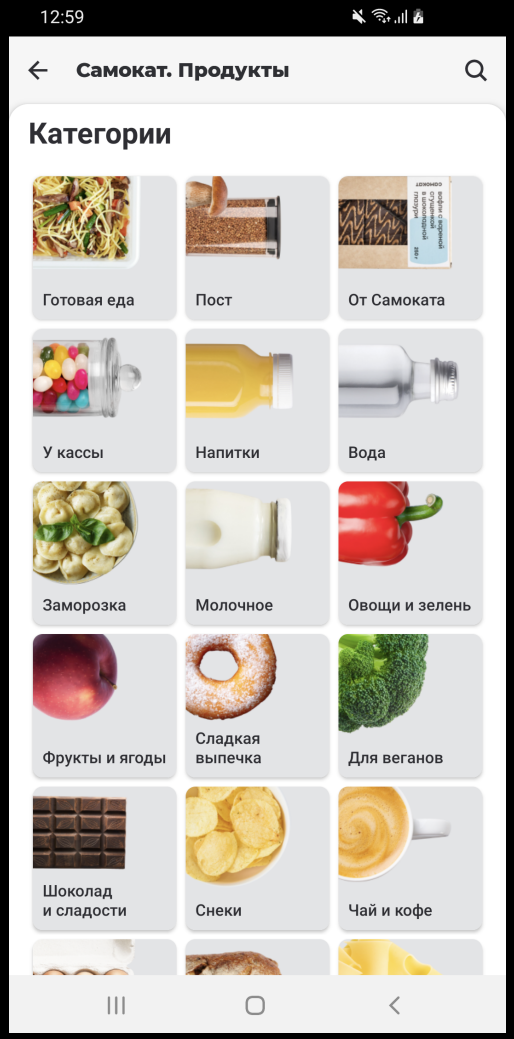
Вывод

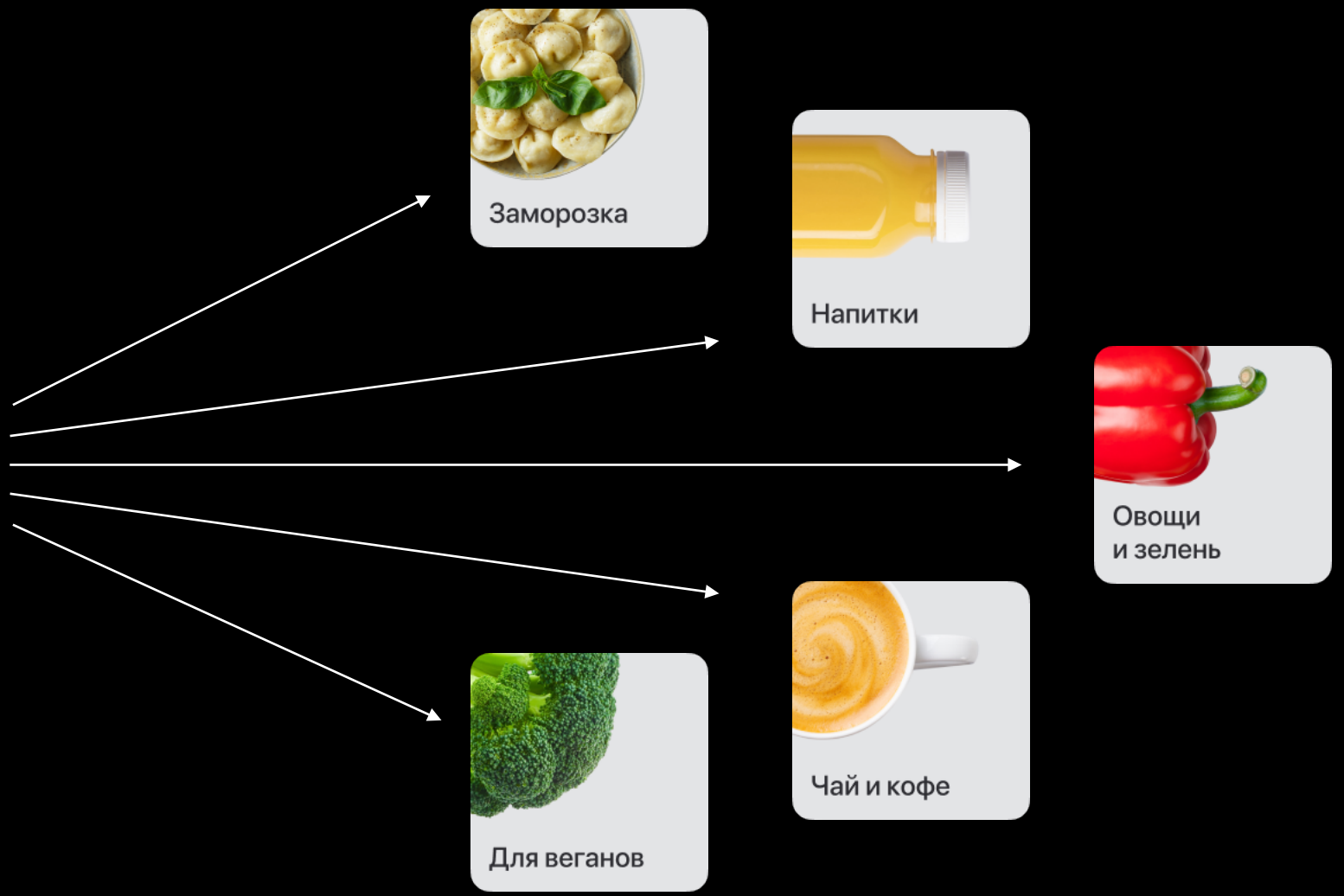
Отмены в прерываниях

1. Отмена в момент прерывания порождает CancellationException.
2. Используйте NonCancellable для блоков кода, которые не должны быть отменены.

Программа

1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Что может пойти не так?

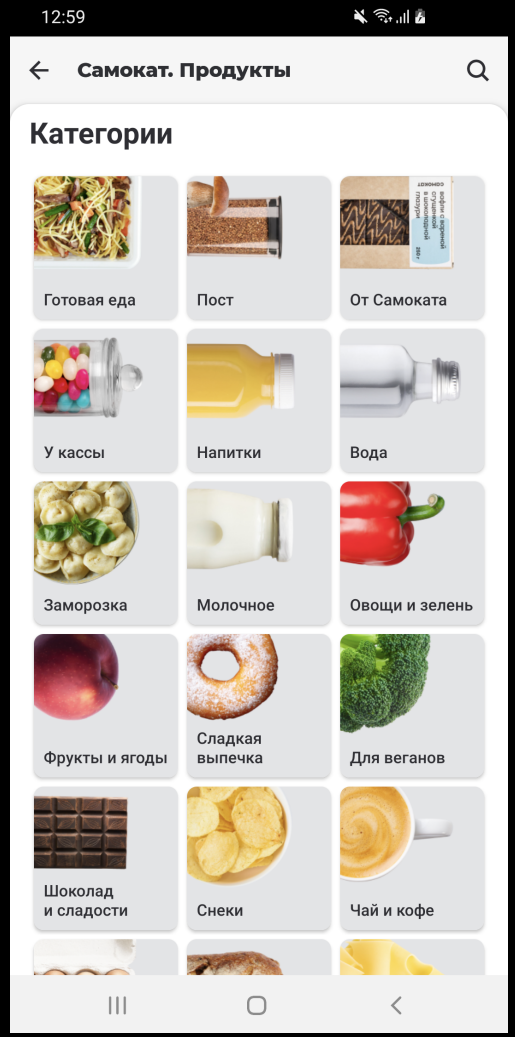

12:59
Самокат. Продукты
Категории
Готовая еда
Пост
От Самоката
У кассы
Напитки
Вода
Заморозка
Молочное
Овощи и зелень
Фрукты и ягоды
Сладкая выпечка
Для веганов
Шоколад и сладости
Снеки
Чай и кофе

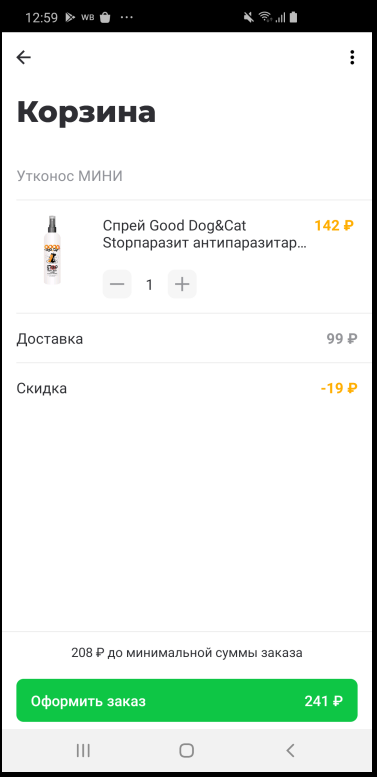

Заморозка
Напитки
Овощи и зелень
Для веганов
Чай и кофе

Retrofit suspend function

```
val scope = CoroutineScope(Dispatchers.IO)


fun loadData() {
    scope.launch {
        categoriesId.forEach { id ->
            apiService.getCategory(id)
            // some logic with saving data
        }
    }
}

@GET("URL")
suspend fun getCategory(id: String) : Any
```


12:59
Продукты
самокат
20-30 мин
Доставим бесплатно
МИРАТОРГ
45-60 мин
Первая доставка за 1 ₽ по коду FREE1
Быстроном
40-60 мин
Доставим бесплатно от 699 ₽


12:59
Самокат. Продукты
Категории
Готовая еда
Пост
От Самоката
У кассы
Напитки
Вода
Заморозка
Молочное
Овощи и зелень
Фрукты и ягоды
Сладкая выпечка
Для веганов
Шоколад и сладости
Снеки
Чай и кофе


12:59
Корзина
Утконос МИНИ
Спрей Good Dog&Cat Stopпаразит антипаразитар...
142 ₽
1
Доставка
99 ₽
Скидка
-19 ₽
208 ₽ до минимальной суммы заказа
Оформить заказ
241 ₽

Retrofit suspend function

```
val scope = CoroutineScope(Dispatchers.IO)


fun loadData() {
    scope.launch {
        categoriesId.forEach { id ->
            apiService.getCategory(id)
            // some logic with saving data
        }
    }
}

@GET("URL")
suspend fun getCategory(id: String) : Any

fun cancel() {
    scope.cancel()
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        }
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        }
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        }
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        }
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        }
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
}
```

Retrofit suspend function

```
suspend fun <T : Any> Call<T?>.await(): T? {
  return suspendCancellableCoroutine { continuation ->
    continuation.invokeOnCancellation {
      cancel()
    }
    enqueue(object : Callback<T?> {
      override fun onResponse(call: Call<T?>, response: Response<T?>) {
        if (response.isSuccessful) {
          continuation.resume(response.body())
        } else {
          continuation.resumeWithException(HttpException(response))
        } continuation.resumeWithException(HttpException(response))
      }

      override fun onFailure(call: Call<T?>, t: Throwable) {
        continuation.resumeWithException(t)
      }
    }) continuation.resumeWithException(t)
  }
}
```

Вывод

Кастомные прерывания

1. SuspendCoroutine {} vs SuspendCancellableCoroutine {}.
2. Высвобождайте ресурсы и производите необходимые отписки от колбэков в CancellableContinuation.invokeOnCancellation {}.

Программа

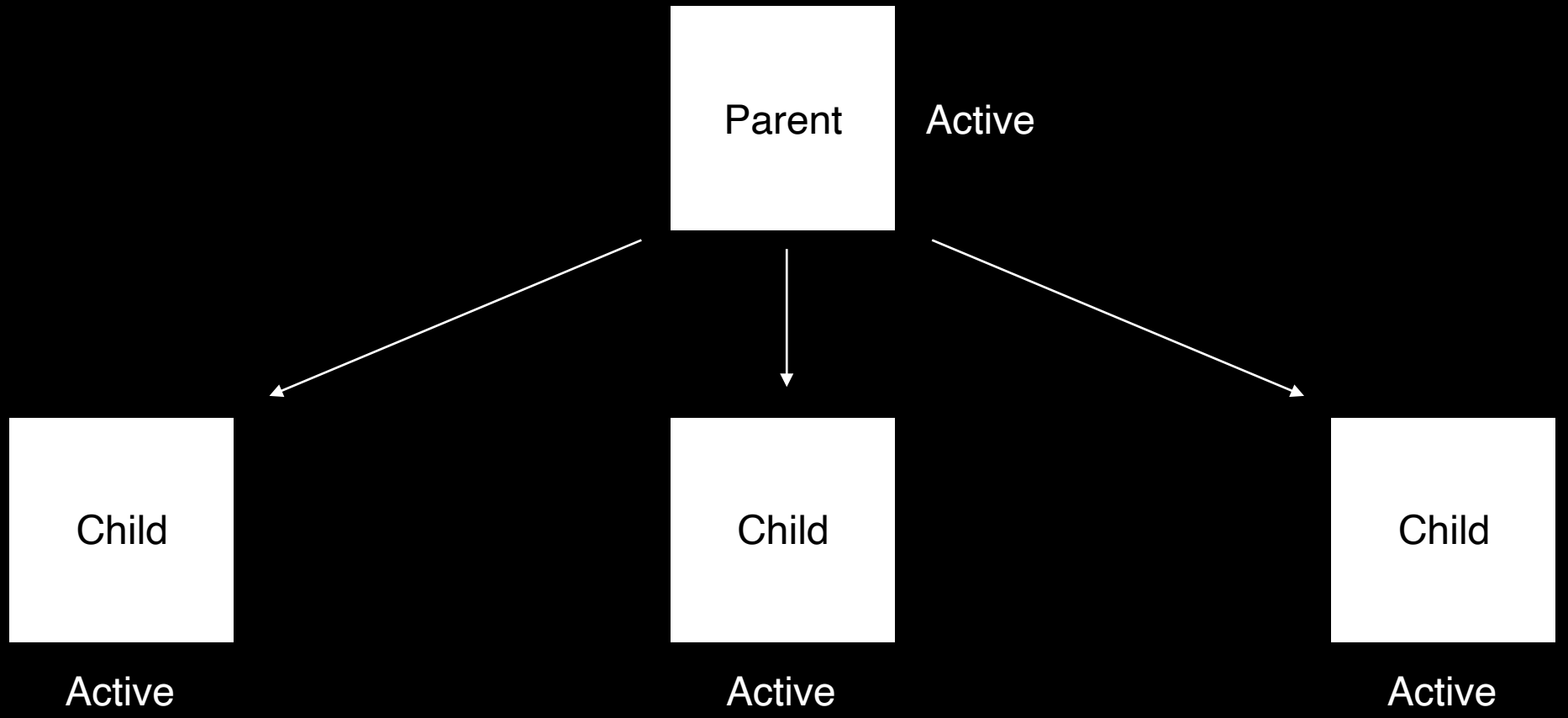
1. Вспомним как отменять Thread.
2. Рассмотрим отмену корутин на реальном примере загрузки данных.
3. Познакомимся с отменами в прерываниях.
4. Разберемся с кастомным прерыванием на примере Retrofit.
5. Что может пойти не так?

Job() vs SupervisorJob()

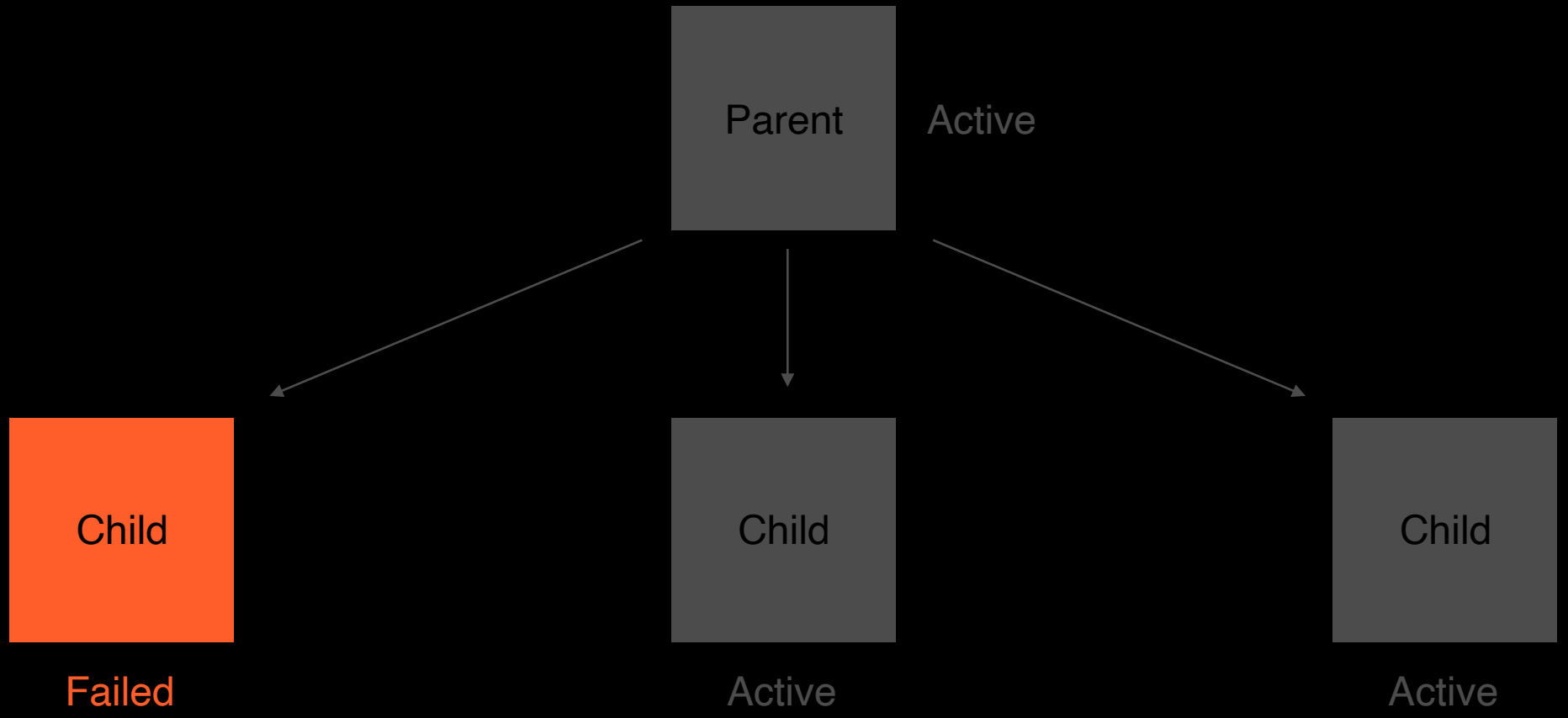
```
val scope = CoroutineScope(Job())


scope.launch(Job()) {

}
```

```
val scope = CoroutineScope(SupervisorJob())


scope.launch(SupervisorJob()) {

}
```

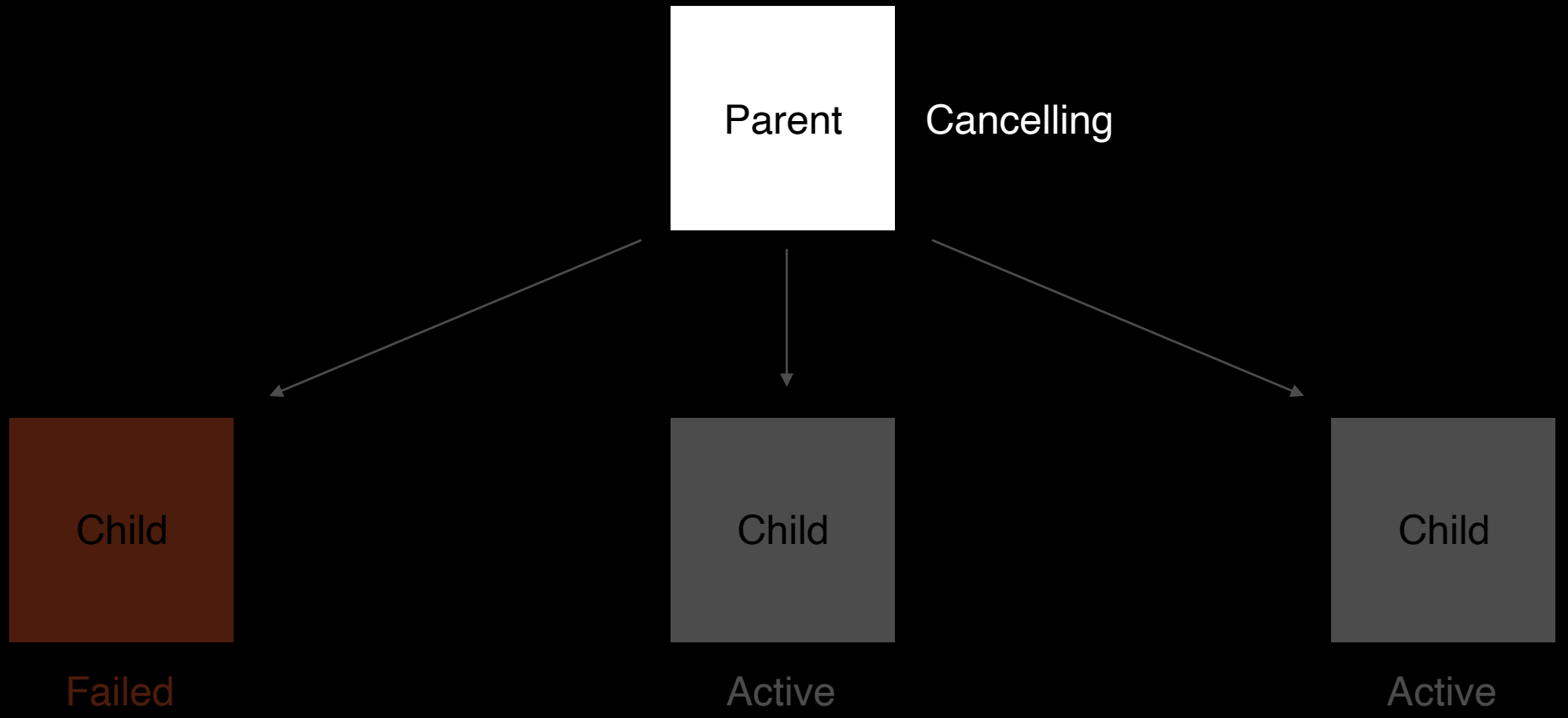
Job



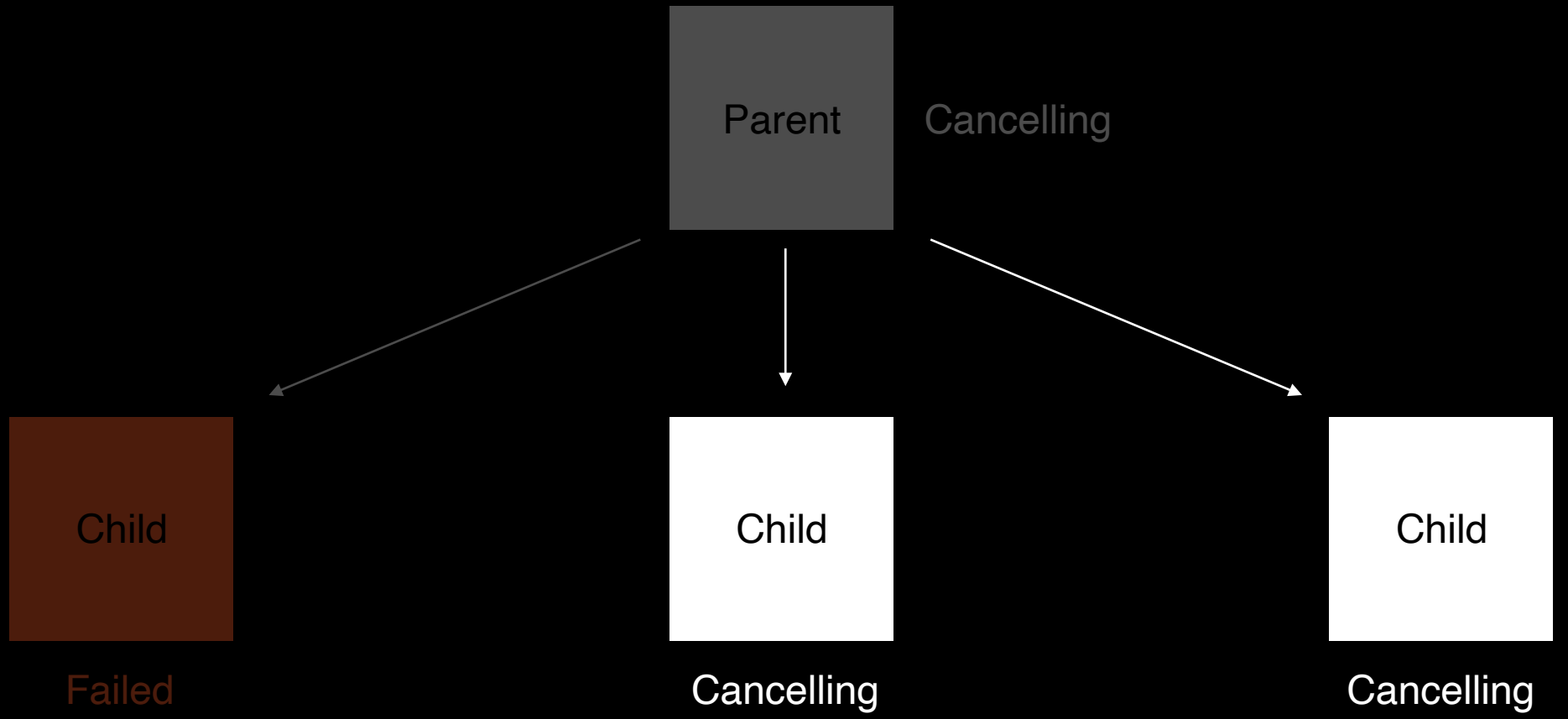
Job



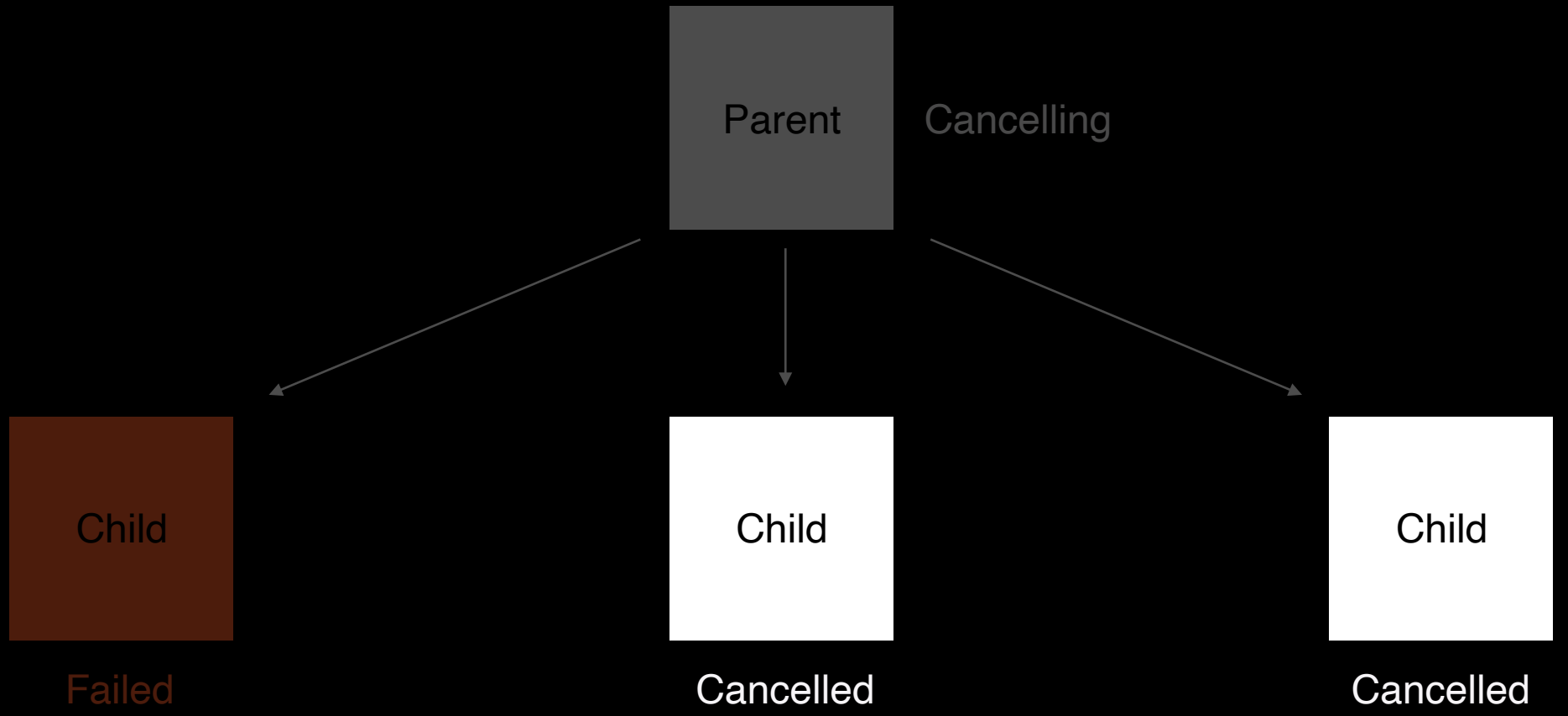
Job



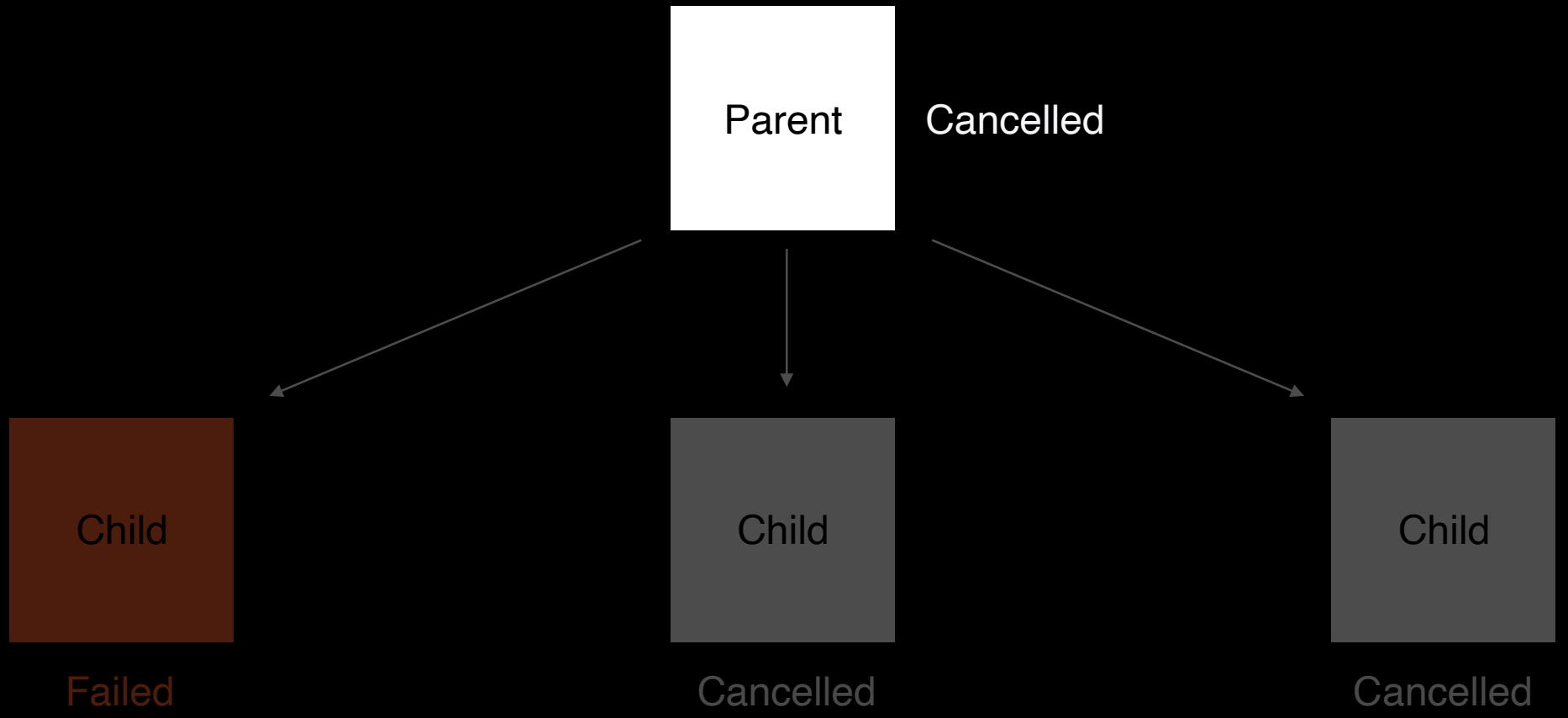
Job


Parent
Cancelling
Child
Child
Child
Failed
Cancelling
Cancelling

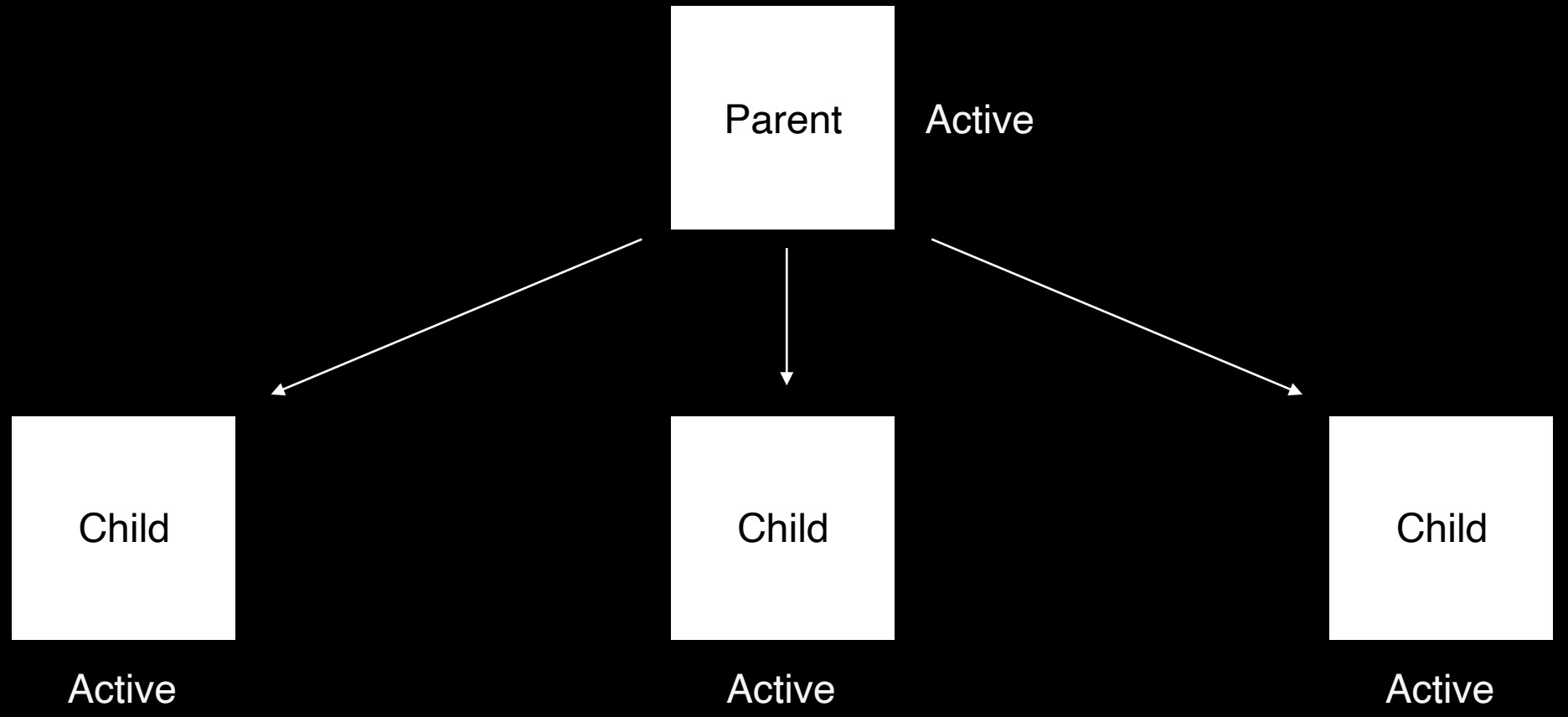
Job


Parent
Cancelling
Child
Child
Child
Failed
Cancelled
Cancelled

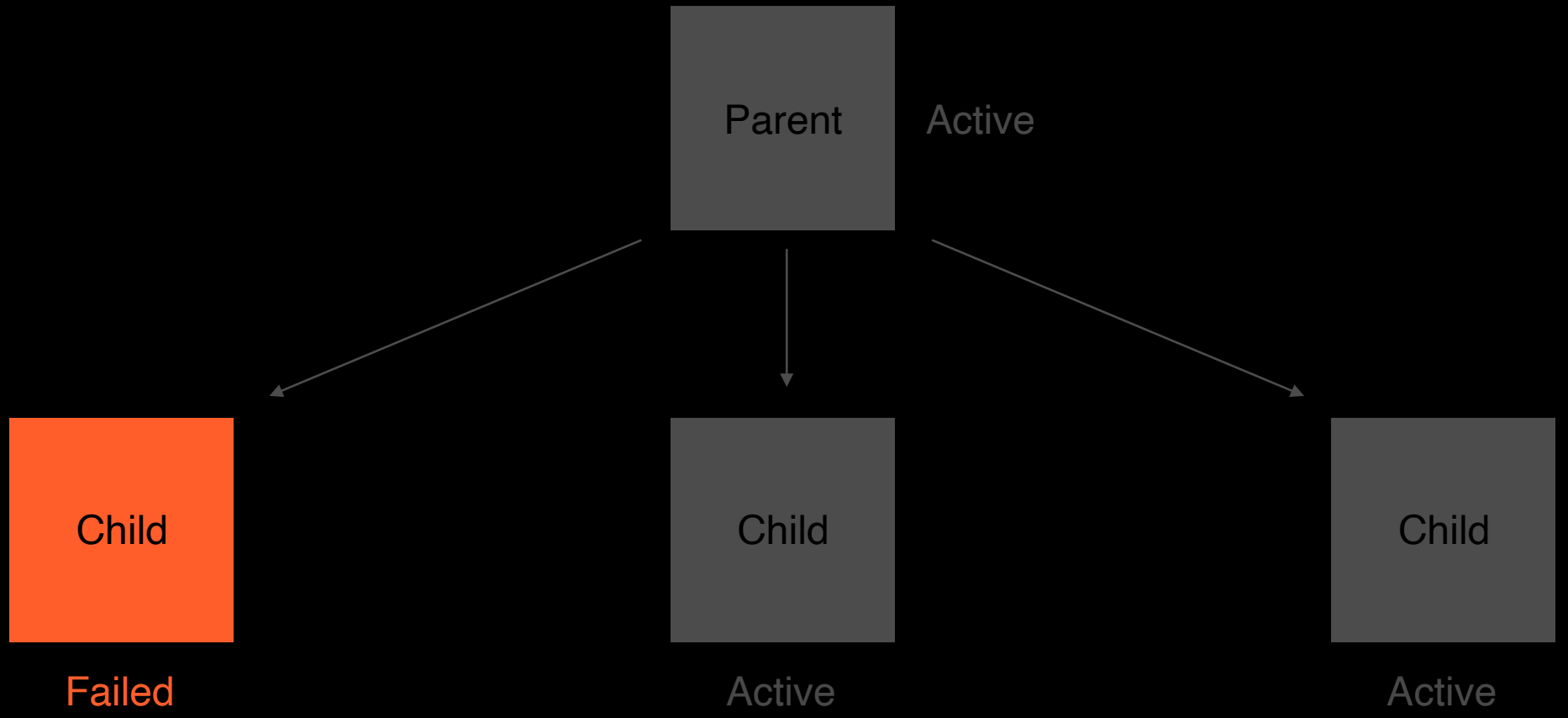
Job


Parent
Cancelled
Child
Child
Child
Failed
Cancelled
Cancelled

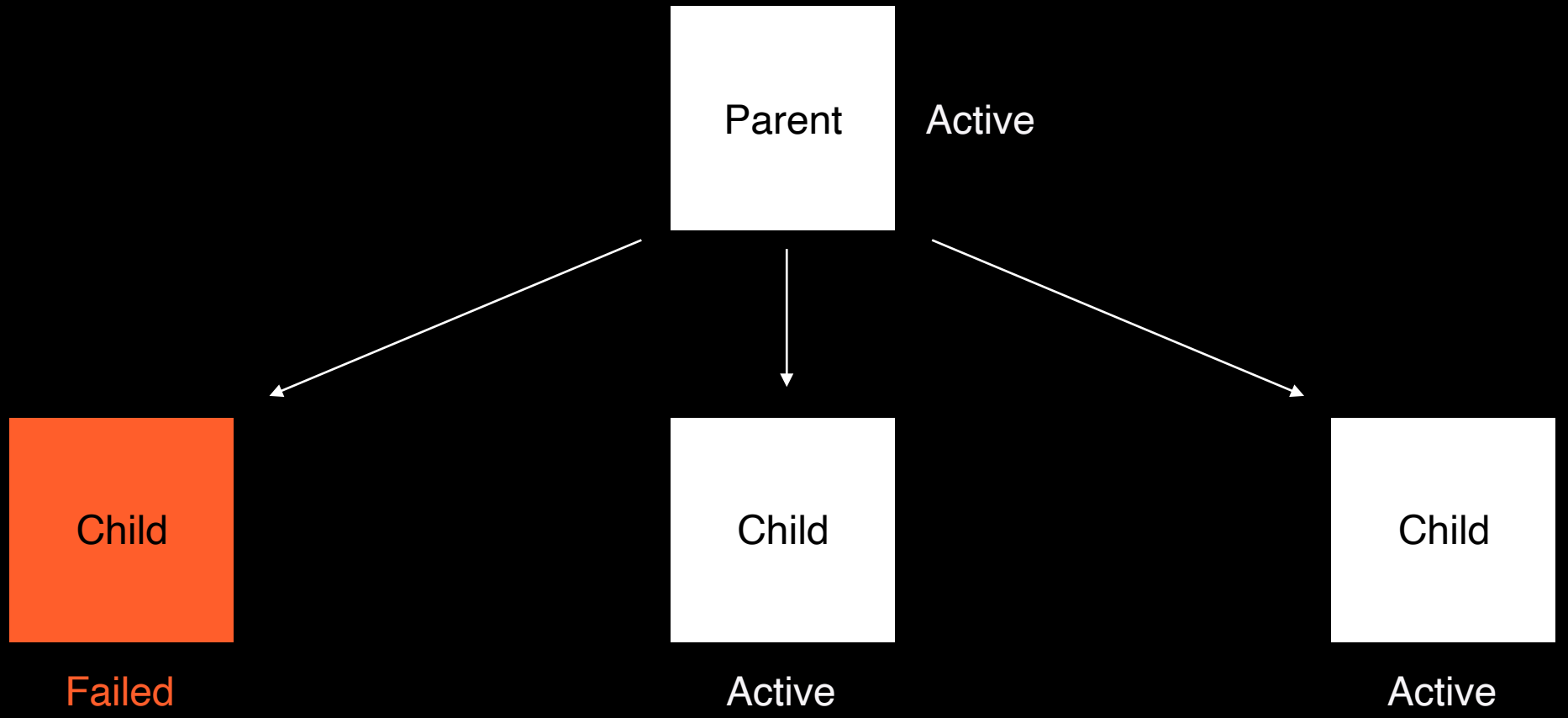
SupervisorJob



SupervisorJob



SupervisorJob



Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
        }
    }

    scope.launch {
        products.forEach { product ->
            // do some work
        }
    }
}
```

Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
            throw RuntimeException()
        }
    }

    scope.launch {
        products.forEach { product ->
            // do some work
        }
    }
}
```

Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
            throw RuntimeException()
        }
    }

    scope.launch {
        products.forEach { product ->
            // do some work
        }
    }
}
```

Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
            throw RuntimeException()
        }
    }

    scope.launch {
        products.forEach { product ->
            // do some work
        }
    }
}
```

Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
            throw RuntimeException()
        }
    }

    scope.launch {
        products.forEach { product ->
            if (isActive.not()) return@launch
            // do some work
        }
    }
}
```

Job() vs SupervisorJob()

```
val scope = CoroutineScope(Dispatchers.IO + CoroutineExceptionHandler { context, error ->
    // handle error
})

fun startSyncData() {
    scope.launch {
        products.forEach { product ->
            // do some work
            throw RuntimeException()
        }
    }

    scope.launch {
        products.forEach { product ->
            if (isActive.not()) return@launch
            // do some work
        }
    }
}
```

Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
    }
}
```

Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
    }
}
```

Nested functions

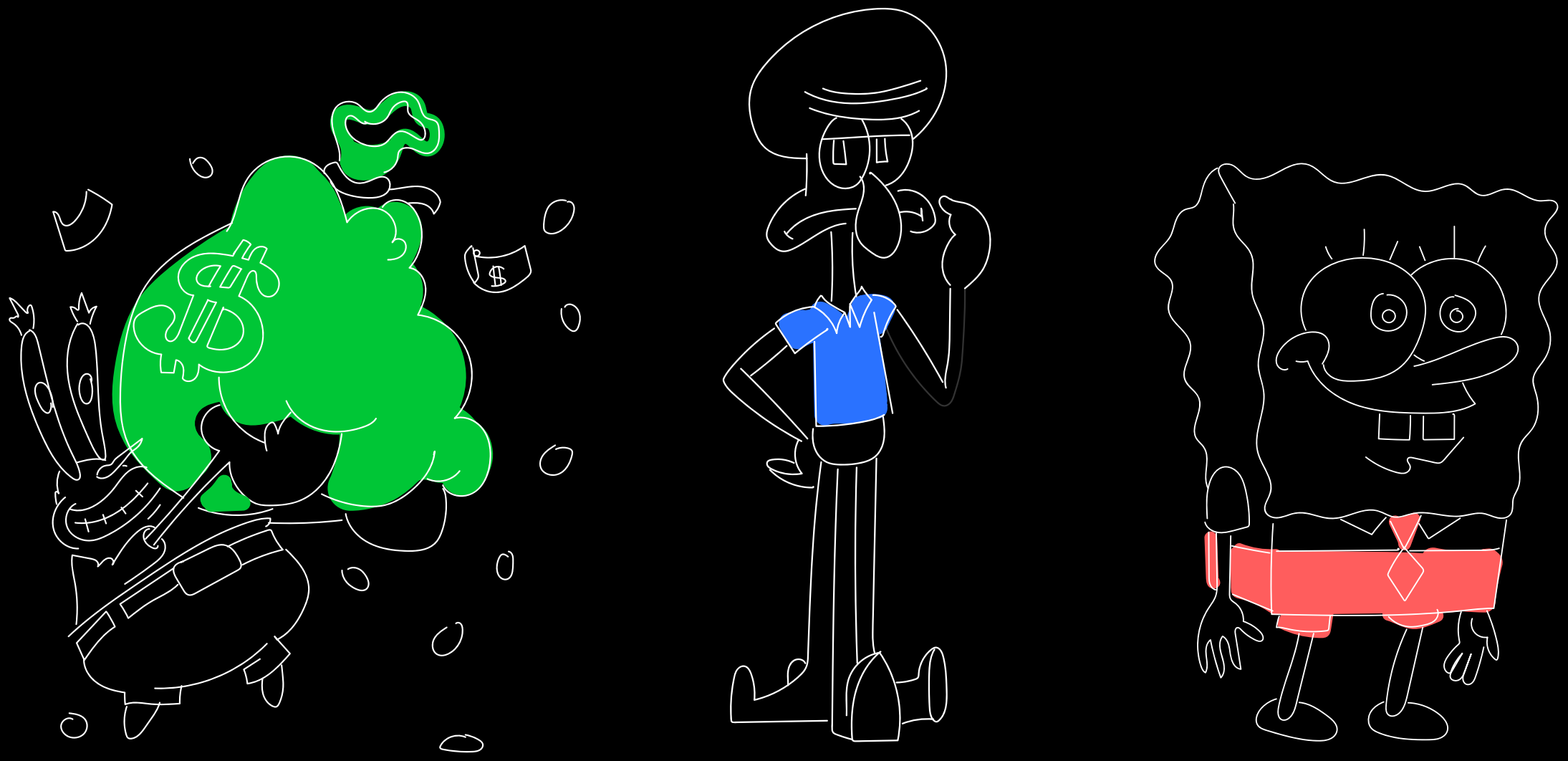
```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
    }
}
```

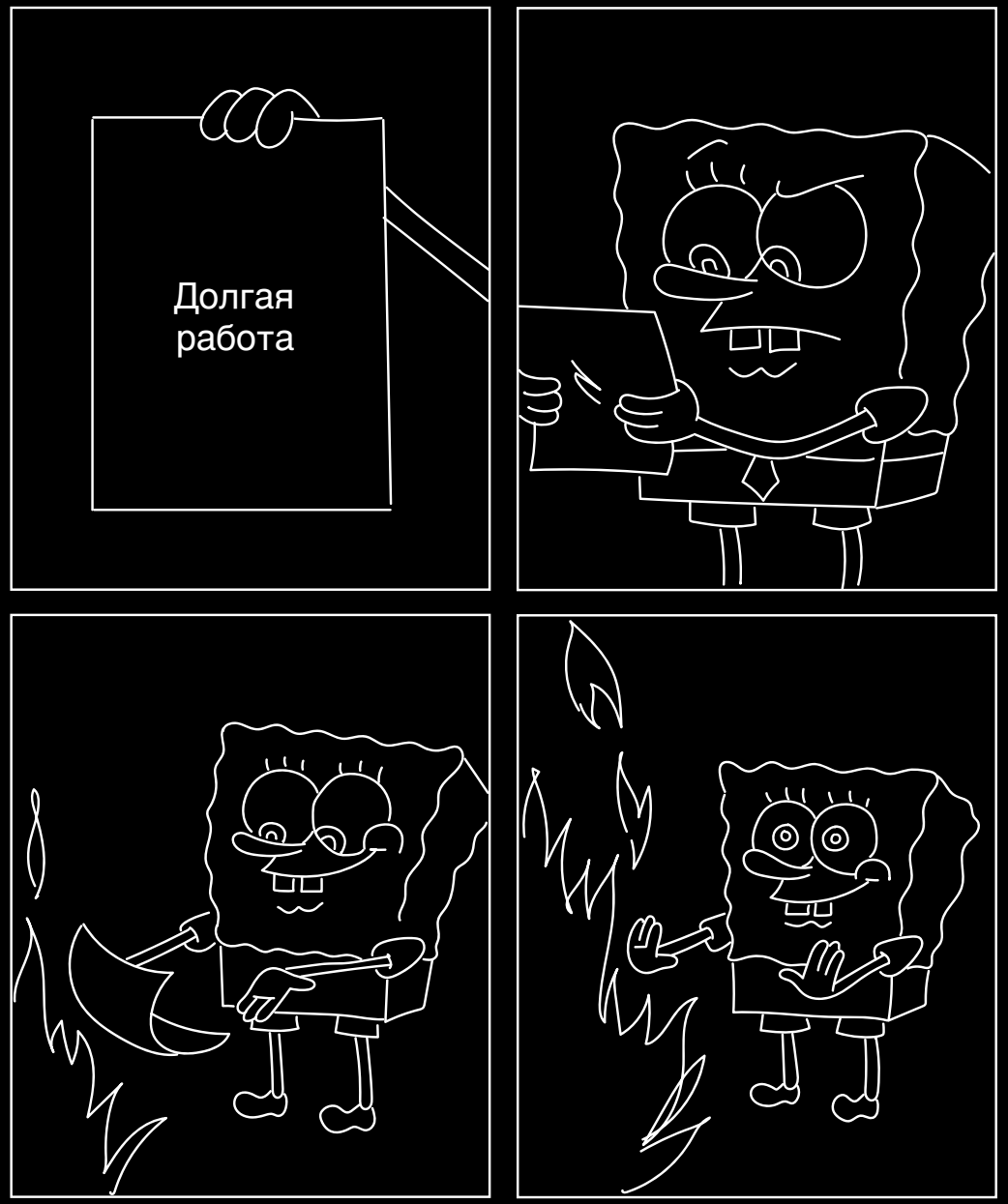
Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
    }
}



fun cancelWork() {
    scope.cancel()
}
```

Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
    }
}



fun cancelWork() {
    scope.cancel()
}
```




Долгая
работа

Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
        throw error
    }
}
```

Nested functions

```
val scope = CoroutineScope(Dispatchers.IO)


fun startSomeOperation() {
    scope.launch {
        startSuspendableOperation()
        // do something
    }
}

suspend fun startSuspendableOperation() {
    try {
        delay(3000)
        // do something
    } catch (error: CancellationException) {
        // release resources
        throw error
    }
}
```

Вывод

Не забыть

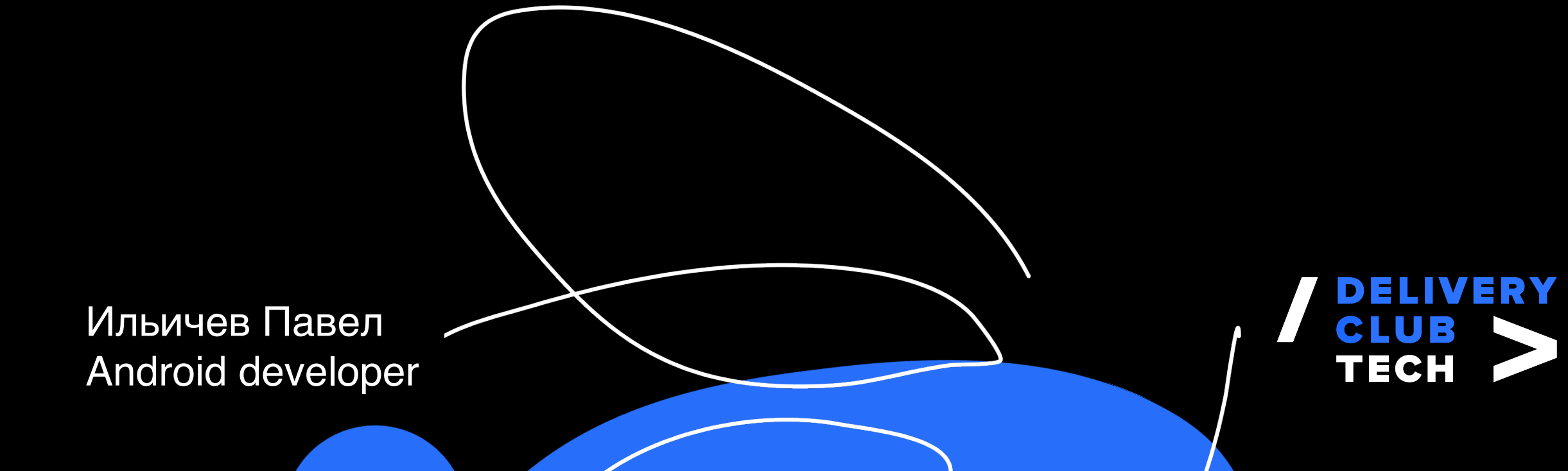
1. Не забывать делать отменяемыми корутины, особенно если работаете с Job в CoroutineContext.
2. Если локально отловили CancellationException, то необходимо пробросить его дальше.

Напутствие

1. Любые длительную операции делайте отменяемыми.
2. Контролируйте процесс отмен.
3. Консистентно комбинируйте механизмы окончания работы при отмене корутины с механизмами определения отмененного состояния.

Спасибо
за внимание

Ильичев Павел
Android developer



DELIVERY CLUB TECH