

# Cancelling Coroutines

<https://medium.com/androiddevelopers/cancellation-in-coroutines-aa6b90163629>

1

## Two types of Coroutine Cancellation

1. Intentional cancellation
  - Call `cancel()` on `Job` or `CoroutineScope`.
2. Abnormal cancellation
  - Exception thrown

2

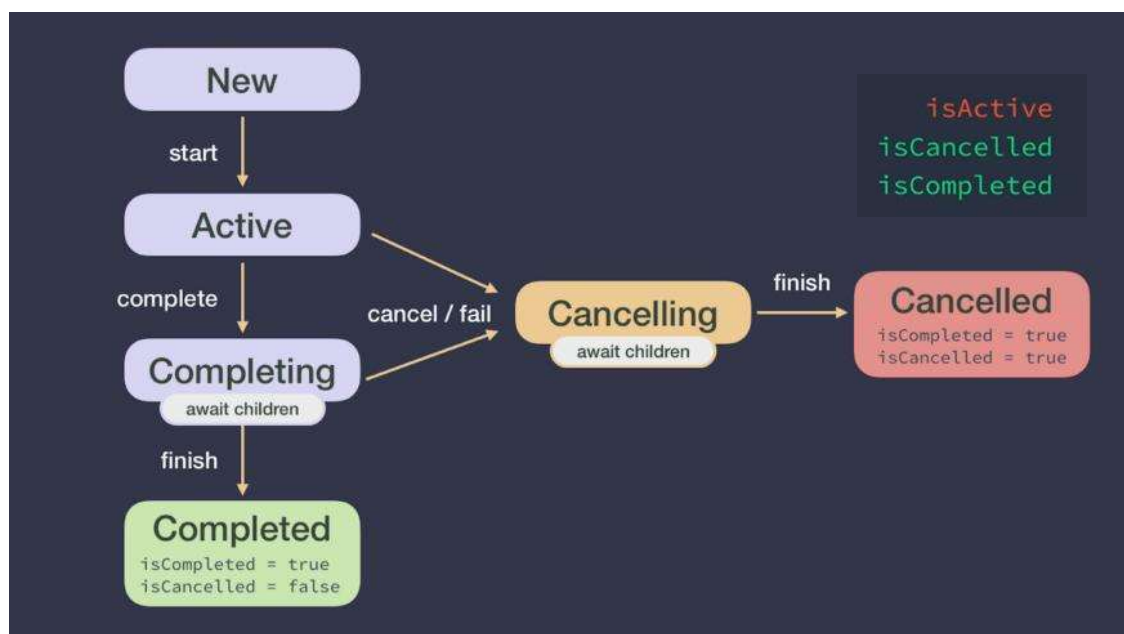
# Why cancel coroutines?

- Make sure to control the life of the coroutine, so that you can cancel it when it's no longer needed.
- Kotlin Coroutines offers is surprisingly simple, convenient, and safe, thanks to *structured concurrency*.



3

## Job lifecycle



4

State	<a href="#">isActive</a>	<a href="#">isCompleted</a>	<a href="#">isCancelled</a>
<i>New</i> (optional initial state)	false	false	false
<i>Active</i> (default initial state)	true	false	false
<i>Completing</i> (transient state)	true	false	false
<i>Cancelling</i> (transient state)	false	false	true
<i>Cancelled</i> (final state)	false	true	true
<i>Completed</i> (final state)	false	true	false

5

## Cancelling a coroutine

```
fun cancel(cause: CancellationException? = null)
```

```
val job = launch {
    // simulate IO or long-running computation
    ↪ delay(500L)
}

...

job.cancel()
```

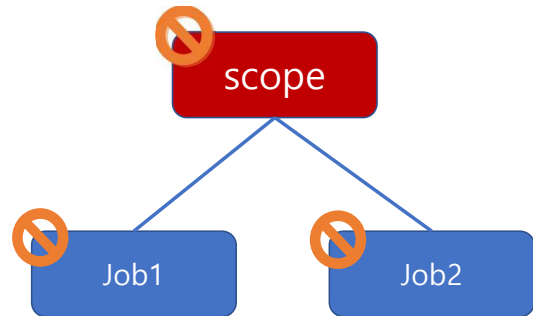
- When you cancel the parent coroutine, *all of its children are recursively cancelled, too.*

6

# Cancelling the scope cancels its children

- To cancel the entire coroutines recursively, cancel the root scope from which all the descendent coroutines are created.

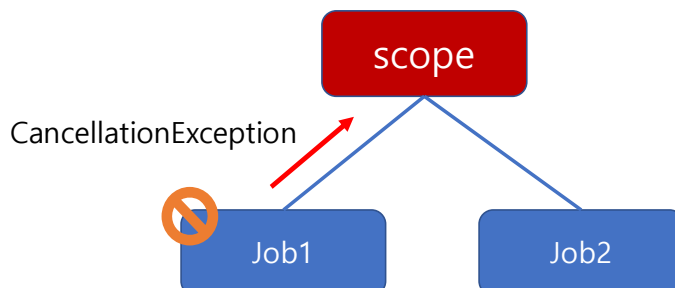
```
// assume we have a scope defined  
val job1 = scope.launch { ... }  
val job2 = scope.launch { ... }  
  
scope.cancel()
```



7

# Cancelling a child doesn't affect other siblings and its parent

```
// Assume we have a scope defined  
val job1 = scope.launch { ... }  
val job2 = scope.launch { ... }  
  
// First coroutine will be cancelled and the other one won't be affected  
job1.cancel()
```



8

# Cancellation Mechanism

- Coroutines handle cancellation by throwing a special exception: `CancellationException`.
- When a job is cancelled, a `CancellationException` exception is thrown *at the first suspension point*, notifying its parent about the cancellation.
- If the child was cancelled due to `CancellationException`, then no other action is required for the parent.

⚠ Once you cancel a scope, you won't be able to launch new coroutines in the cancelled scope.

9

## viewModelScope/lifecycleScope in Android

- Both `viewModelScope` and `lifecycleScope` are `CoroutineScope` objects that get cancelled at the right time.
- For example, when the `ViewModel` is cleared, it cancels the coroutines launched in its scope automatically.



10

# viewModelScope

```
class ProfileViewModel : ViewModel() {  
    val scope = CoroutineScope(  
        Dispatchers.Main + SupervisorJob()  
    )  
  
    fun onCreate() {  
        scope.launch { loadUserData() }  
    }  
  
    fun onDestroy() {  
        scope.cancel()  
    }  
  
    // ...  
}
```

```
class ProfileViewModel : ViewModel() {  
  
    fun onCreate() {  
        viewModelScope.launch { loadUserData() }  
    }  
  
    fun onDestroy() {  
        // viewModelScope.cancel()  
        // is automatically called  
    }  
    // ...  
}
```

11

## Why isn't my coroutine work stopping?

- The coroutine work doesn't just stop when `cancel` is called.
  - Rather, we need check if the coroutine is active periodically.

```
val startTime = System.currentTimeMillis()  
val job = launch {  
    var nextPrintTime = startTime  
    var i = 0  
    while (i < 5) {  
        // print a message twice a second  
        if (System.currentTimeMillis() >= nextPrintTime) {  
            println("Hello $i")  
            nextPrintTime += 500L  
        }  
    }  
}  
delay(1000)  
job.cancel()
```

```
Hello 0  
Hello 1  
Hello 2  
Hello 3  
Hello 4
```

*Cancellation of coroutine code needs to be **cooperative**!*

12

# Making your coroutine work cancellable

To make the coroutine code cooperative we have two options:

1. Checking `job.isActive` or `ensureActive()`
2. Let other work happen using `yield()`

```
val job = launch {  
    for(file in files) {  
        // TODO check for cancellation  
        readFile(file)  
    }  
}
```

13

## Making your coroutine work cancellable

To make your coroutine code cooperative we have two options:

1. **Checking `job.isActive` or `ensureActive()`**

```
val job = launch {  
    for(file in files) {  
        if (isActive()) {  
            readFile(file)  
        } else break  
    }  
    if (!isActive) { ... }  
}
```

```
val job = launch {  
    for(file in files) {  
        ensureActive()  
        readFile(file)  
    }  
}
```

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

14

# Making your coroutine work cancellable

To make your coroutine code cooperative we have two options:

## 2. Let other work happen using `yield()`

If the work you're doing is

- 1) CPU heavy,
- 2) may exhaust the thread pool and
- 3) you want to allow the thread to do work without having to add more to the pool, then use `yield()`.

```
val job = launch {  
    for(file in files) {  
        yield()  
        readFile(file)  
    }  
}
```

15

## Pro Tips

- All the suspending functions in `kotlinx.coroutines` are *cancellable*.

```
val job = launch {  
    repeat(4) {  
        println("Hello $it")  
        delay(500)  
    }  
}  
delay(1000)  
job.cancel()
```

```
Hello 0  
Hello 1
```

16



# Job.join vs Deferred.await cancellation

- `Job.join` suspends a coroutine until the work is completed. Together with `Job.cancel` it behaves as you'd expect:
  - If you're calling `job.cancel` then `job.join`, the coroutine will suspend until the job is completed.
  - Calling `job.cancel` after `job.join` has no effect.
- Calling `await` on a deferred

```
val deferred = async { ... }  
  
deferred.cancel()  
val result = deferred.await() // throws JobCancellationException!
```

- Calling `deferred.cancel` after `deferred.await` nothing happens.

17

## Handling cancellation side effects

Let's say that you want to execute a specific action when a coroutine is cancelled: closing any resources you might be using, logging the cancellation or some other cleanup code you want to execute.

- Check for `isActive`

```
while (i < 5 && isActive) {  
    // print a message twice a second  
    if (...) {  
        println("Hello ${i++}")  
        nextPrintTime += 500L  
    }  
}  
  
// the coroutine work is completed so we can cleanup  
if (!isActive) println("Clean up!")
```

18

# Handling cancellation side effects

- Use *try-catch-finally*

```
val job = launch {  
    try {  
        work()  
    } catch (e: CancellationException){  
        println("Work cancelled!")  
    } finally {  
        println("Clean up!")  
    }  
}
```

- But, what if the cleanup work we need to execute is suspending ...

19

## A coroutine in the cancelling state is not able to suspend!

- To be able to call suspend functions when a coroutine is cancelled, switch the cleanup work in a [NonCancellable CoroutineContext](#).

```
val job = launch {  
    try {  
        work()  
    } catch (e: CancellationException){  
        println("Work cancelled!")  
    } finally {  
        withContext(NonCancellable){  
            delay(1000L) // or some other suspend fun  
            println("Cleanup done!")  
        }  
    }  
}
```

20

## suspendCancellableCoroutine and invokeOnCancellation

```
suspend fun searchRecipes(query: String): List<Recipe> =
    suspendCancellableCoroutine { continuation ->
        val call = apiService.searchRecipes(query)
        call.enqueue(object : Callback<List<Recipe>> {
            override fun onResponse(call: Call<List<Recipe>>,
                                   response: Response<List<Recipe>>) {
                if (response.isSuccessful) {
                    val body = response.body()
                    if (body != null) {
                        continuation.resume(body)
                    }
                }
                // ...
            }
        })
        override fun onFailure(call: Call<List<Recipe>>, t: Throwable) {
            continuation.resumeWithException(t)
        }
    })
    continuation.invokeOnCancellation {
        call.cancel()
    }
}
```

21

## Summary of Cancelling Coroutines

- In order to cancel a coroutine, you simply need to call the `cancel` method on the Job object that was returned from the coroutine builder.
- Calling the `cancel` function on a `Job`, or on a `Deferred` instance, will stop the inner computation on a coroutine if the handling of the `isActive` flag is properly implemented.
- *Coroutine cancellation is cooperative*. This means that the suspending function has to cooperate in order to support cancelling.
- All suspending functions provided by the Kotlin coroutine library support cancellation already.
- Inside a cancelled coroutine, `CancellationException` is considered to be a normal reason for coroutine completion.

22