Cancelling Coroutines

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

Two types of Coroutine Cancellation

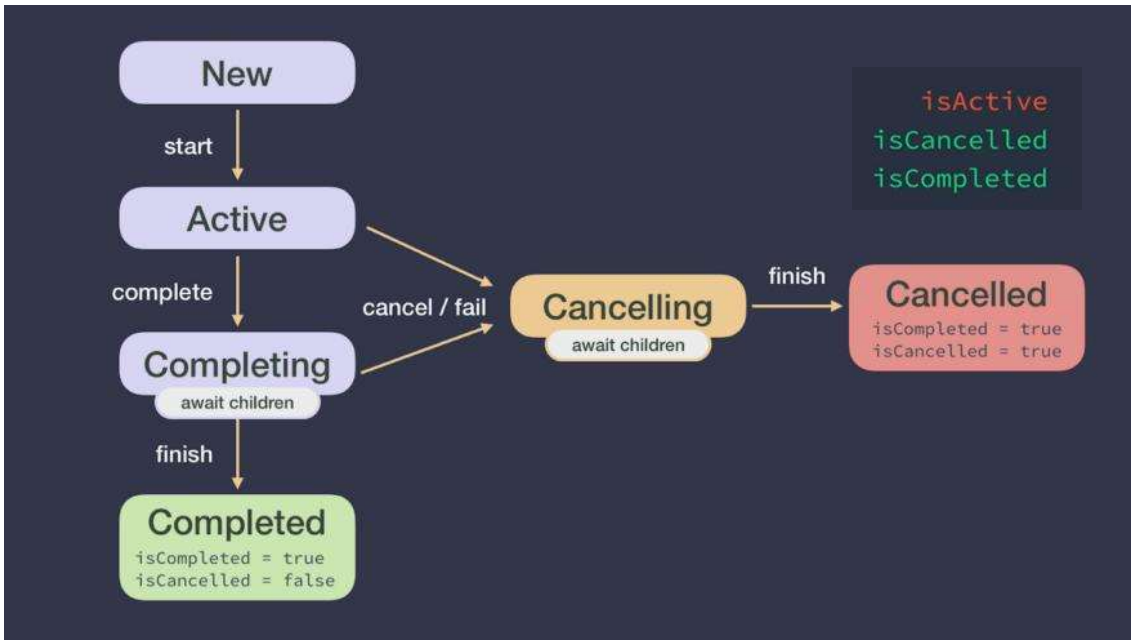
1. Intentional cancellation
   – Call `cancel()` on `Job` or `CoroutineScope`.

2. Abnormal cancellation
   – Exception thrown

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*.

Job lifecycle



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`

Cancelling a coroutine

fun cancel(cause: CancellationException? = null)

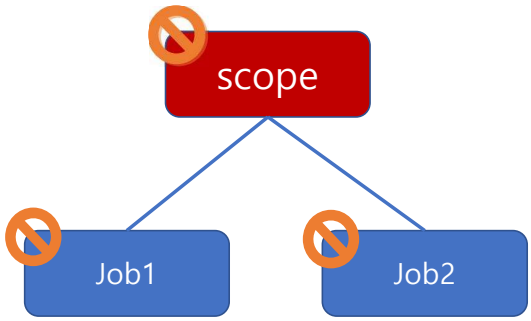
```
val job = launch {
    // simulate IO or long-running computation
    delay(500L)
}

...

job.cancel()
```

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

Cancelling the scope cancels its children

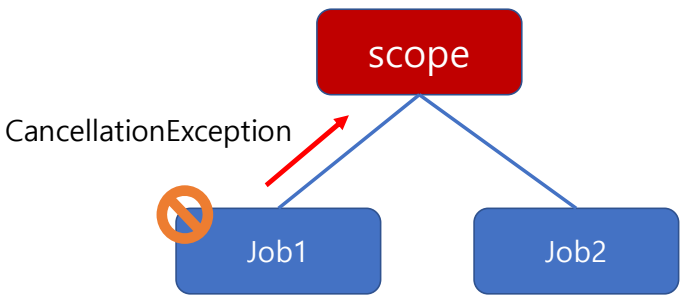
- To cancel the entire coroutines recursively, cancel the root scope from which all the descendent coroutines are created.

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

scope.cancel()
```



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()
```



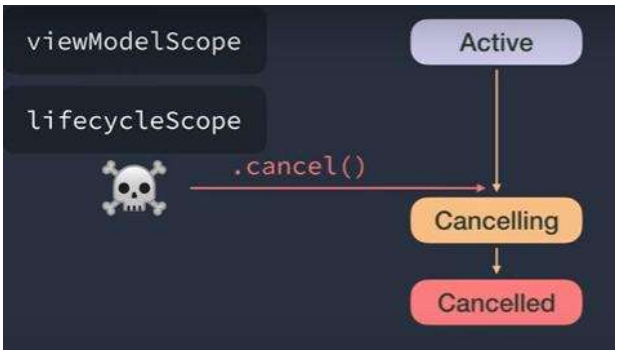
Cancellation Mechanism

- Coroutines handle cancellation by throwing a special exception: `CancellationException`.
- When a job is cancelled, a `CancellationException` 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 or call suspend functions in the cancelled scope.

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.



viewModelScope

```
class ProfileViewModel : ViewModel() {
  val scope = CoroutineScope(
      Dispatchers.Main + SupervisorJob()
  )

  fun onCreate() {
        scope.launch { loadUserData() }
  }

  fun onCleared() {
    scope.cancel()
  }

    // ...
}
```

```
class ProfileViewModel : ViewModel() {

  fun onCreate() {
    viewModelScope.launch { loadUserData() }
  }

  fun onCleared() {
    // viewModelScope.cancel()
    // is automatically called
  }
    // ...
}
```


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!***

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)
    }
}
```


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()
  }
}
```

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)
    }
}
```


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
```

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` (or `job.cancelAndJoin`), 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.

Handling cancellation side effects

Let's say that you want to execute a specific action when a coroutine is cancelled:

- closing any resources,
- 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!")
```

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 …

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` coroutine context.

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

CancellationException (Cont'd)

- We consume the `CancellationException` and prevent the coroutine from being cancelled properly.

```
viewModelScope.launch {
    fetchData<Data> { api.request1() }
    // do something
    fetchData<Data> { api.request2() }
}
```

This coroutine cancelled for some reason

First fetchData will be cancelled

But, second fetchData will be executed, too

```
private suspend fun <T> fetchData(action: suspend () -> T) =
    try {
        liveData.value = Resource.Success(action())
    } catch (ex: Exception) {
        liveData.value = Resource.Error(ex.message)
        if (ex is CancellationException) {
            throw ex
        }
    }
```


Migration from Callback to Suspend Function

- Use `suspendCoroutine` or `suspendCancellableCoroutine`.

```
fun searchRecipes(query: String, callback: MyCallback<List<Recipes>>) {
    val call = apiService.searchRecipes(query)
    call.enqueue(object : Callback<List<Recipe>> {
        override fun onResponse(call: Call<List<Recipe>>, …) {
            if (response.isSuccessful) {
                callback.onSuccess(body = response.body()!!)
            }
            // ...
        }

        override fun onFailure(call: Call<List<Recipe>>, t: Throwable) {
            callback.onError(t)
        }
    })
}
```

suspendCancellableCoroutine and invokeOnCancellation

```
suspend fun searchRecipes(query: String): List<Recipes> =
    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) {
                    continuation.resume(response.body()!!)
                }
                // ...
            }

            override fun onFailure(call: Call<List<Recipe>>, t: Throwable) {
                continuation.resumeWithException(t)
            }
        })
        continuation.invokeOnCancellation {
            call.cancel()
        }
    }
```


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 cancelation 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.