10 Canceling Operations

Written by Scott Grosch

With an operation, you have the capability of canceling a running operation *as long as it's written properly*. This is very useful for long operations that can become irrelevant over time. For instance, the user might leave the screen or scroll away from a cell in a table view. There's no sense in continuing to load data or make complex calculations if the user isn't going to see the result.

The magic of cancel

Once you schedule an operation into an operation queue, you no longer have any control over it. The queue will schedule and manage the operation from then on. The one and only change you can make, once it's been added to the queue, is to call the `cancel` method of `Operation`.

There's nothing magical about how canceling an operation works. If you send a request to an operation to stop running, then the `isCancelled` computed property will return `true`. Nothing else happens automatically! At first, it may seem strange that iOS doesn't stop the operation automatically, but it's really not.

What does *canceling* an operation mean to the OS?

- Should the operation simply throw an exception?
- Is there cleanup that needs to take place?
- Can a running network call be canceled?
- Is there a message to send server-side to let something else know the task stopped?
- If the operation stops, will data be corrupted?

With just the small list of issues presented in the bullets above, you can see why setting a flag identifying that cancellation has been *requested* is all that's possible automatically.

The default `start` implementation of `Operation` will first check to see whether the `isCancelled` flag is `true`, and exit immediately if it is.

Cancel and cancelAllOperations

The interface to cancel an operation is quite simple. If you just want to cancel a specific `Operation`, then you can call the `cancel` method. If, on the other hand, you wish to cancel all operations that are in an operation queue, then you should call the `cancelAllOperations` method defined on `OperationQueue`.

Updating AsyncOperation

In this chapter, you'll update the app you've been working on so that a cell's operations are canceled when the user scrolls away from that cell.

In Chapter 8, "Asynchronous Operations," you built the `AsyncOperation` base class. If you recall, there was a note with that code warning you that the provided implementation wasn't entirely complete. It's time to fix that!

The start class provided was written like so:

```
override func start() {
  main()
  state = .executing
}
```

If you're going to allow your operation to be cancelable — which you should always do unless you have a very good reason not to — then you need to check the `isCancelled` variable at appropriate locations. Open up the starter project from this chapter's download materials and edit **AsyncOperation.swift** to update the `start` method:

```
override func start() {
  if isCancelled {
    state = .finished
    return
  }

  main()
  state = .executing
}
```

After the above changes, it's now possible for your operation to be canceled before it's started.

Canceling a running operation

To support the cancellation of a running operation, you'll need to sprinkle checks for `isCancelled` throughout the operation's code. Obviously, more complex operations are going to be able to check in more places than something simple like your `NetworkImageOperation`.

Open up **NetworkImageOperation.swift** and in `main`, add a new `guard` statement right after the `defer`:

```
guard !self.isCancelled else { return }
```

For the network download, there's really no other location that you'd need to make the check. In fact, it's questionable whether or not you'd really want to make the check at all.

You've already spent the time to download the image from the network. Is it better to cancel and return no image, or let the image get created? There's no right or wrong answer. It's simply an architectural decision that you'll have to make based on the requirements of the project.

Next, add a way to cancel the network request while it's in progress. First, add a new property to the class:

```
private var task: URLSessionDataTask?
```

This will hold the network task while it's being run. Next, in `main`, replace the line where you create the data task with the following line:

```
task = URLSession.shared.dataTask(with: url) { [weak self]
```

Next, remove the call to `resume` at the end of that block. Instead, you're going to call `resume` on the `task` by adding the following at the end of `main`:

```
task?.resume()
```

Finally, you need to override `cancel` to make sure the task is canceled. Add the following method to the class:

```
override func cancel() {
  super.cancel()
  task?.cancel()
}
```

Now, the downloading can be canceled at any time.

It's time to allow canceling in **TiltShiftOperation.swift**. You'll probably want to place two checks in the `main` method. Just before setting the `fromRect` variable, make the first check:

```
guard !isCancelled else { return }
```

Once you've applied the tilt shift and grabbed the output image, that's a good point to stop before you then create the `CGImage`.

Next, just before setting `outputImage`, add the same check again.

```
guard !isCancelled else { return }
```

You've got a `CGImage` at this point but there's no value in converting it to a `UIImage` if a cancellation was requested. Some would argue for a third check, right after creating the `outputImage`, but that leads to the same question posed during the network operation: You've already done all the work, do you really want to stop now?

Now that you have a way to cancel the operation, it's time to hook this up to the table view so that the operations for a cell are canceled when the user scrolls away.

Open up **TiltShiftTableViewController.swift** and add the following property to the class:

```
private var operations: [IndexPath :[Operation]] = [:]
```

This is a dictionary that will hold the operations for a specific cell (both the downloading and tilt shifting). You need to store the operations because canceling is a method on the actual operation, so you need a way to grab it to cancel it.

Add the following lines to `tableView(_:cellForRowAt:)`, right before `return cell`, to store the operations:

```
if let existingOperations = operations[indexPath] {
  for operation in existingOperations {
    operation.cancel()
  }
}

operations[indexPath] = [tiltShiftOp, downloadOp]
```

If an operation for this index path already exists, cancel it, and store the new operations for that index path.

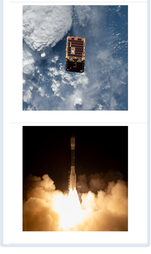
Next, add the following method to the end of the class:

```
override func tableView(
  _ tableView: UITableView,
  didEndDisplaying cell: UITableViewCell,
  forRowAt indexPath: IndexPath) {

  if let operations = operations[indexPath] {
    for operation in operations {
      operation.cancel()
    }
  }
}
```

This implements a table view delegate method that gets called when a cell goes offscreen. At that point, you'll cancel the operations for that cell, making sure the phone's resources are only used for visible cells.

Build and run the app.



You probably won't notice a big difference, but now when you quickly scroll through the table view the app won't load and filter an image for each cell that quickly went past the screen. The downloads for the ones that went offscreen are canceled, saving the user's network traffic and battery life and making your app run faster.

Where to go from here?

Having to cancel an operation doesn't necessarily mean something negative happened. At times you cancel an operation because it's simply no longer necessary. If you're working with a `UITableView` or a `UICollectionView` you may want to implement the prefetching delegate methods introduced in iOS 10. When the controller is going to prefetch, you'd create the operations. If the controller cancels prefetching, you'd cancel the operations as well.

Mark Complete

11. Core Data →

← 9. Operation Dependencies