7 Operation Queues

Written by Scott Grosch

The real power of operations begins to appear when you let an `OperationQueue` handle your operations. Just like with GCD's `DispatchQueue`, the `OperationQueue` class is what you use to manage the scheduling of an `Operation` and the maximum number of operations that can run simultaneously.

`OperationQueue` allows you to add work in three separate ways:

- Pass an `Operation`.
- Pass a closure.
- Pass an array of `Operation`s.

If you implemented the project from the previous chapter, you saw that an operation by itself is a synchronous task. While you could dispatch it asynchronously to a GCD queue to move it off the main thread, you'll want, instead, to add it to an `OperationQueue` to gain the full concurrency benefits of operations.

OperationQueue management

The operation queue executes operations that are **ready**, according to quality of service values and any dependencies the operation has. Once you've added an `Operation` to the queue, it will run until it has completed or been canceled. You'll learn about dependencies and canceling operations in future chapters.

Once you've added an `Operation` to an `OperationQueue`, you can't add that same `Operation` to any other `OperationQueue`. `Operation` instances are **once and done** tasks, which is why you must have control into subclasses so that you can execute them multiple times, if necessary.

Waiting for completion

If you look under the hood of `OperationQueue`, you'll notice a method called `waitUntilAllOperationsAreFinished`. It does exactly what its name suggests: Whenever you find yourself wanting to call that method, in your head, replace the word **wait** with **block** in the method name. Calling it blocks the current thread, meaning that you must never call this method on the main UI thread.

If you find yourself needing this method, then you should set up a private serial `DispatchQueue` wherein you can safely call this blocking method. If you don't need to wait for all operations to complete, but just a set of operations, then you can, instead, use the `addOperations(_:waitUntilFinished:)` method on `OperationQueue`.

Quality of service

An `OperationQueue` behaves like a `DispatchGroup` in that you can add operations with different quality of service values and they'll run according to the corresponding priority. If you need a refresher on the different quality of service levels, refer back to Chapter 3, "Queues & Threads."

The default quality of service level of an operation queue is `.background`. While you can set the `qualityOfService` property on the operation queue, keep in mind that it might be overridden by the quality of service that you've set on the individual operations managed by the queue.

Pausing the queue

You can pause the operation queue by setting the `isSuspended` property to `true`. In-flight operations will continue to run but newly added operations will not be scheduled until you change `isSuspended` back to `false`.

Maximum number of operations

Sometimes you'll want to limit the number of operations which are running at a single time. By default, the dispatch queue will run as many jobs as your device is capable of handling at once. If you wish to limit that number, simply set the `maxConcurrentOperationCount` property on the dispatch queue. If you set the `maxConcurrentOperationCount` to `1`, then you've effectively created a serial queue.

Underlying DispatchQueue

Before you add any operations to an `OperationQueue`, you can specify an existing `DispatchQueue` as the `underlyingQueue`. If you do so, keep in mind that the quality of service of the dispatch queue will override any value you set for the operation queue's quality of service.

> **Note:** Do not specify the main queue as the underlying queue!

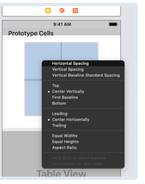
Fix the previous project

In the previous chapter, you set up an operation to handle the tilt shift, but it ran synchronously. Now that you're familiar with `OperationQueue`, you'll modify that project to work properly. You can either continue with your existing project or open up **Concurrency.xcodeproj** from this chapter's starter materials.

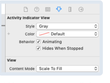
UIActivityIndicator

The first change you'll make is to add a `UIActivityIndicator` to clue the user that something is happening. Open up the **Main.storyboard** and choose the **Tilt Shift Table View Controller Scene**. Drag an activity indicator to the center of the image so that the crosshairs appear in both directions and place it there.

Once you've done that, Control-drag from the activity indicator onto the image view in a diagonal manner. On the pop-up that appears hold down the Shift key and select both **Center Vertically** and **Center Horizontally**.



On the **Attributes inspector**, check the **Animating** and **Hides When Stopped** behaviors.



Open up **PhotoCell.swift**. Add a new `@IBOutlet`, called `activityIndicator`, and link it to the newly added activity indicator in the storyboard:

```
@IBOutlet private weak var activityIndicator: UIActivityIndicatorView!
```

Next, add the following computed property to `PhotoCell`:

```
var isLoading: Bool {
  get { return activityIndicator.isAnimating }
  set {
    if newValue {
      activityIndicator.startAnimating()
    } else {
      activityIndicator.stopAnimating()
    }
  }
}
```

While you could make the `activityIndicator` property public and call the methods directly, it's suggested to not expose UI elements and outlets to avoid leaking UIKit-specific logic to higher layers of abstraction. For example, you may wish to replace this indicator with a custom component at some point down the road.

Updating the table

Head over to **TiltShiftTableViewController.swift**. In order to add operations to a queue, you need to create one. Add the following property to the top of the class:

```
private let queue = OperationQueue()
```

Next, replace everything in `tableView(_:cellForRowAt:)` between declaring `image` and returning the cell with the following:

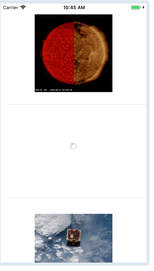
```
let op = TiltShiftOperation(image: image)
op.completionBlock = {
  DispatchQueue.main.async {
    guard let cell = tableView.cellForRow(at: indexPath)
      as? PhotoCell else { return }

    cell.isLoading = false
    cell.display(image: op.outputImage)
  }
}

queue.addOperation(op)
```

Instead of manually calling `start` on the operation, you'll now add the operation to a queue that will be starting and completing it for you. Additionally, the queue runs in the background by default, so you won't be blocking the main thread anymore.

When the operation completes, the `completionBlock` is called with no arguments, and it expects no return value. You'll immediately want to dispatch your code back to the main UI thread. If you are able to get a reference to the table view cell (if it wasn't scrolled away), then you're simply turning off the activity indicator and updating the image.

As soon as you add an operation to an operation queue, the job will be scheduled. Now there's no longer a reason to call `op.start()`.

Build and run the app and try scrolling the table again.



Now that your code is running in an asynchronous manner, the table scrolling is much smoother. While making such a change doesn't do anything to improve the performance of the code being run via the operation, it does ensure that the UI isn't locked up or choppy.

You may be thinking to yourself, "How is this any different than just doing it with GCD?" Right now, there really isn't a difference. But, in the next couple of chapters, you'll expand on the power of operations, and the reason for the changes will become clear.

Where to go from here?

The table currently loads and filters every image every time the cell is displayed. Think about how you might implement a caching solution so that the performance is even better.

The project is coming along nicely, but it is using static images, which usually won't be the case for production-worthy projects. In the next chapter, you'll take a look at network operations.

Mark Complete

8. Asynchronous Operations →

← 6. Operations