9 Operation Dependencies

Written by Scott Grosch

In this chapter, you're going to learn about **dependencies** between operations. Making one operation dependent on another provides two specific benefits for the interactions between operations:

1. Ensures that the dependent operation does not begin before the prerequisite operation has completed.
2. Provides a clean way to pass data from the first operation to the second operation automatically.

Enabling dependencies between operations is one of the primary reasons you'll find yourself choosing to use an `Operation` over GCD.

Modular design

Consider the tilt shift project you've been creating. You now have an operation that will download from the network, as well as an operation that will perform the tilt shift. You could instead create a single operation that performs both tasks, but that's not a good architectural design.

Classes should ideally perform a single task, enabling reuse within and across projects. If you had built the networking code into the tilt shift operation directly, then it wouldn't be usable for an already-bundled image. While you could add many initialization parameters specifying whether or not the image would be provided or downloaded from the network, that bloats the class. Not only does it increase the long-term maintenance of the class — imagine switching from `URLSession` to Alamofire — it also increases the number of test cases which have to be designed.

Specifying dependencies

Adding or removing a dependency requires just a single method call on the **dependent** operation. Consider a fictitious example in which you'd download an image, decrypt it and then run the resultant image through a tilt shift:

```
let networkOp = NetworkImageOperation()
let decryptOp = DecryptOperation()
let tiltShiftOp = TiltShiftOperation()

decryptOp.addDependency(op: networkOp)
tiltShiftOp.addDependency(op: decryptOp)
```

If you needed to remove a dependency for some reason, you'd simply call the obviously named method, `removeDependency(op:)`:

```
tiltShiftOp.removeDependency(op: decryptOp)
```

The `Operation` class also provides a read-only property, `dependencies`, which will return an array of `Operation`s, which are marked as dependencies for the given operation.

Avoiding the pyramid of doom

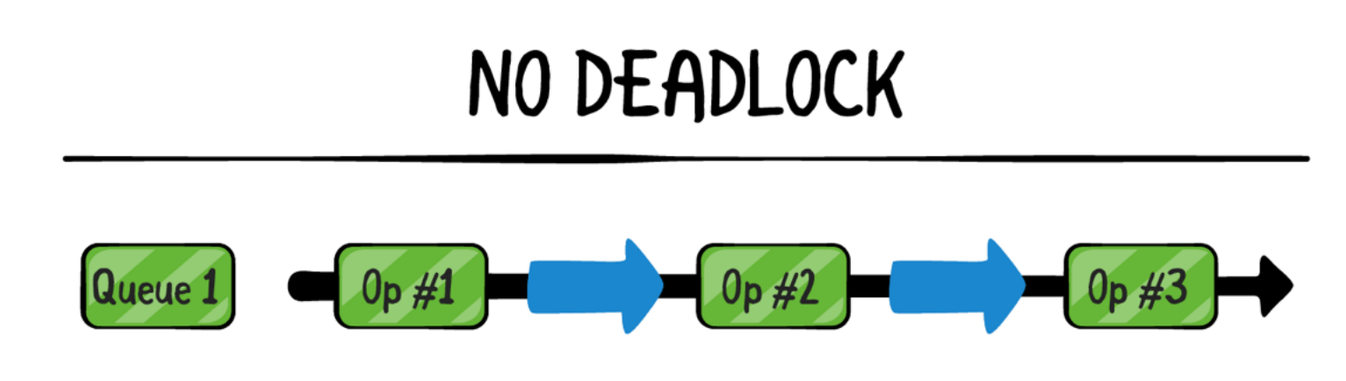
Dependencies have the added side effect of making the code much simpler to read. If you tried to write three chained operations together using GCD, you'd end up with a pyramid of doom. Consider the following **pseudo-code** for how you might have to represent the previous example with GCD:

```
let network = NetworkClass()
network.onDownloaded { raw in
  guard let raw = raw else { return }

  let decrypt = DecryptClass(raw)
  decrypt.onDecrypted { decrypted in
    guard let decrypted = decrypted else { return }

    let tilt = TiltShiftClass(decrypted)
    tilt.onTiltShifted { tilted in
      guard let tilted = tilted else { return }
    }
  }
}
```

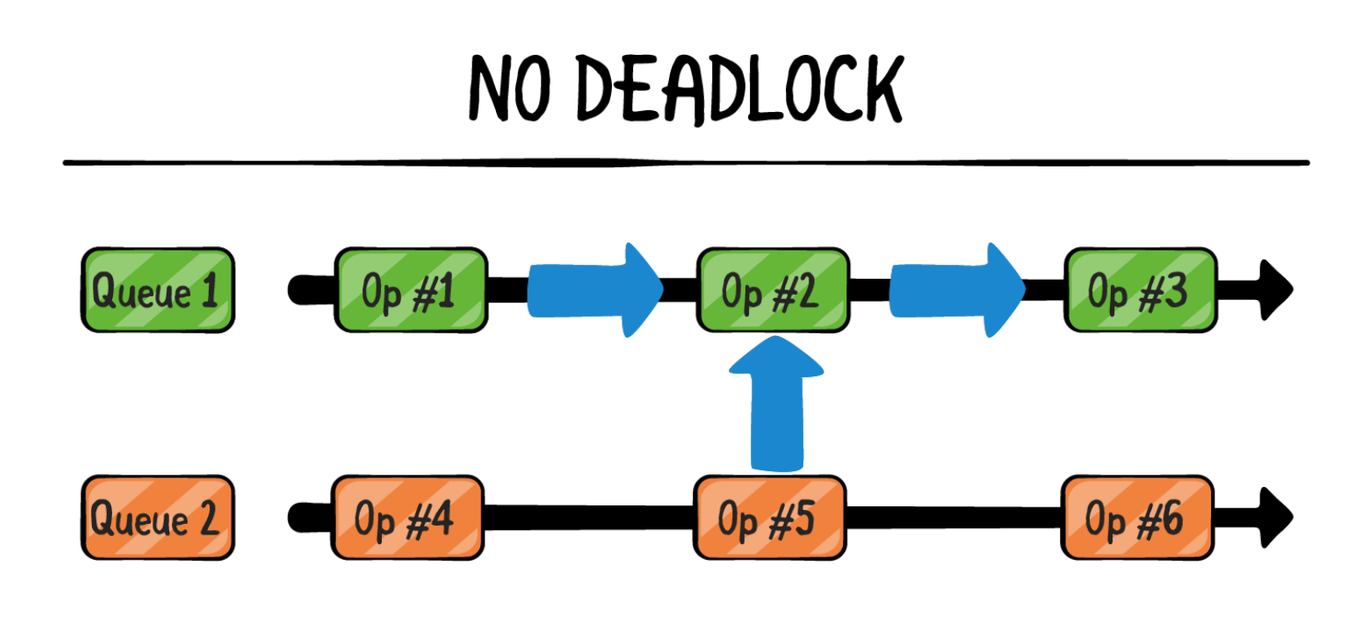
Which one is going to be easier to understand and maintain for the junior developer who takes over your project once you move on to bigger and better things? Consider also that the example provided doesn't take into account the retain cycles or error checking that real code would have to handle properly.

Watch out for deadlock

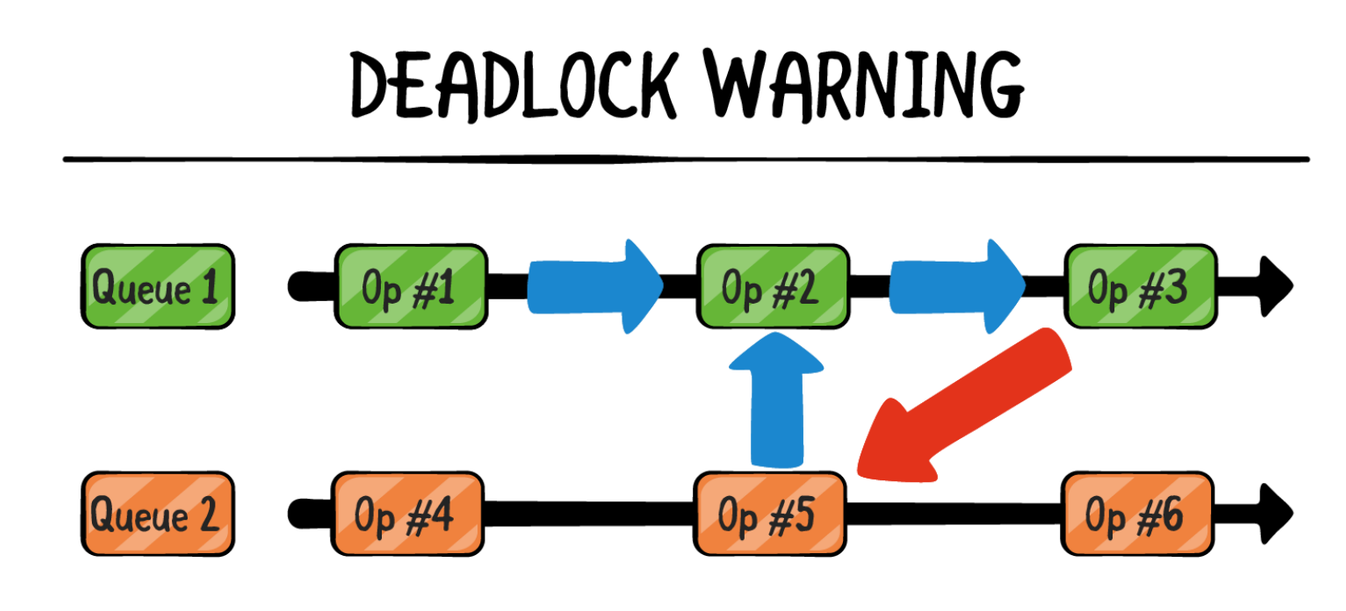
In Chapter 5, "Concurrency Problems," you learned about deadlock. Any time a task is dependent on another, you introduce the possibility of deadlock, if you aren't careful. Picture in your mind — better yet graph out — the dependency chain. If the graph draws a straight line, then there's no possibility of deadlock.



It's completely valid to have the operations from one operation queue depend on an operation from another operation queue. Even when you do that, as long as there are no loops, you're still safe from deadlock.



If, however, you start seeing loops, you've almost certainly run into a deadlock situation.



In the previous image, you can see where the problem lies:

- Operation 2 can't start until operation 5 is done.
- Operation 5 can't start until operation 3 is done.
- Operation 3 can't start until operation 2 is done.

If you start and end with the same operation number in a cycle, you've hit deadlock. None of the operations will ever be executed. There's no silver-bullet solution to resolve a deadlock situation, and they can be hard to find if you don't map out your dependencies. If you run into such a situation, you have no choice but to re-architect the solution you've designed.

Passing data between operations

Now that you've got a way to safely make one operation depend on another, there has to be a way to pass data between them. Enter the power of protocols. The `NetworkImageOperation` has an output property called `image`. What about the case, though, in which the property is called something else?

Part of the benefit to operations is the encapsulation and reusability they provide. You can't expect every person who writes an operation to call the output property `image`. Internally, to the class, there might have been a good reason to call it `foodImage`, for example.

Using protocols

Here's what you're really saying: "When this operation finishes, if everything went well, I will provide you with an image of type `UIImage`."

As usual, please open **Concurrency.xcodeproj** in the starter project folder that comes with this chapter's download materials and then create a new Swift file called **ImageDataProvider.swift**. Add the following code to the file:

```
import UIKit

protocol ImageDataProvider {
  var image: UIImage? { get }
}
```

Any operation that has an output of a `UIImage` should implement that protocol. In this case, the property names match one-to-one, which makes life easier. Think about your `TiltShiftOperation`, though. Following `CIFilter` naming conventions you called that one `outputImage`. Both classes should conform to the `ImageDataProvider`.

Adding extensions

Open up **NetworkImageOperation.swift** and add this code to the very bottom of the file:

```
extension NetworkImageOperation: ImageDataProvider {}
```

Since the class already contains the property exactly as defined by the protocol, there's nothing else you need to do. While you could have simply added `ImageDataProvider` to the class definition, the Swift Style Guide recommends the `extension` approach instead.

For `TiltShiftOperation`, there's a tiny bit more work to do. While you already have an output image, the name of the property isn't `image` as defined by the protocol.

Add the following code at the end of **TiltShiftOperation.swift**:

```
extension TiltShiftOperation: ImageDataProvider {
  var image: UIImage? { return outputImage }
}
```

Remember that an extension can be placed anywhere, in any file. Since you created both operations, it makes sense of course to place the extension right alongside the class. You might be using a third-party framework, however, wherein you can't edit the source. If the operation it provides gives you an image, you can add the extension to it yourself in a file within your project, like **ThirdPartyOperation+Extension.swift**.

Searching for the protocol

The `TiltShiftOperation` needs a `UIImage` as its input. Instead of just requiring the `inputImage` property be set, it can now check whether any of its dependencies provides a `UIImage` as output.

In **TiltShiftOperation.swift**, in `main()`, change the first `guard` statement (e.g. the first line) to this:

```
let dependencyImage = dependencies
  .compactMap { ($0 as? ImageDataProvider)?.image }
  .first

guard let inputImage = inputImage ?? dependencyImage else {
  return
}
```

In the above code, you try to unwrap either the input image directly provided to the operation or the dependency chain for something that will provide us an image, making sure it gave a non-`nil` image.

If neither of those worked, simply return without performing any work.

There's one last piece to making this all work. Because you now check the dependency chain for an image, there has to be a way to initialize a `TiltShiftOperation` without providing an input image. The simplest way to handle no input is by making the current constructor default the input image to `nil`.

Adjust your initializer to look as follows:

```
init(image: UIImage? = nil) {
  inputImage = image
  super.init()
}
```

Updating the table view controller

Head back over to **TiltShiftTableViewController.swift** and see if you can update it to download the image, tilt shift it, and then assign it to the table view cell.

For this to work, you'll need to add the download operation as a dependency of the tilt shift operation. In other words, the tilt shift *depends* on the download operation to get the image.

Replace the line in `tableView(_:cellForRowAt:)` where you set and declare `op` with the following code:

```
let downloadOp = NetworkImageOperation(url: urls[indexPath.row])
let tiltShiftOp = TiltShiftOperation()
tiltShiftOp.addDependency(downloadOp)
```

Instead of having a single operation, you now have two operations and a dependency between them.

Next, instead of setting `completionBlock` on the `op`, set it on the `tiltShiftOp`, because it will provide you with the image.

Replace the entire completion block with the following:

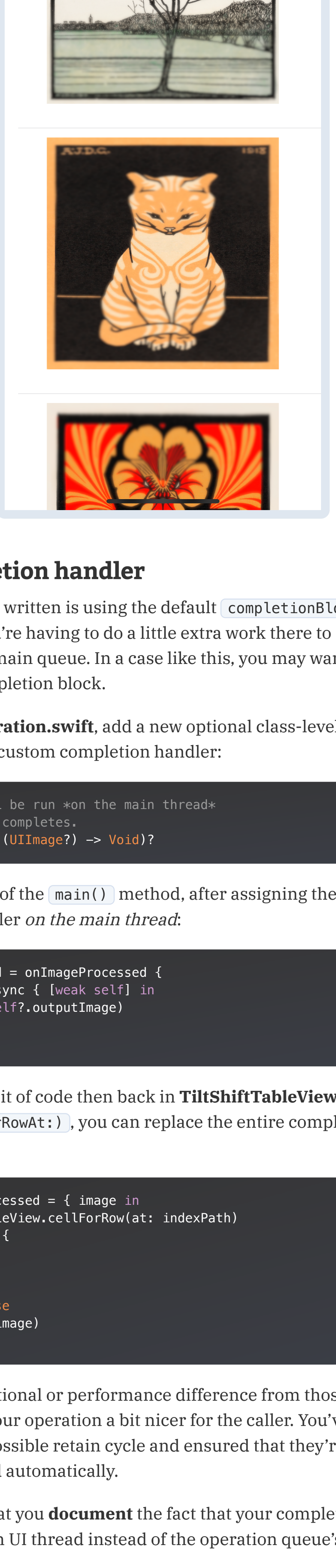
```
tiltShiftOp.completionBlock = {
  DispatchQueue.main.async {
    guard let cell = tableView.cellForRow(at: indexPath)
      as? PhotoCell else { return }

    cell.isLoading = false
    cell.display(image: tiltShiftOp.image)
  }
}
```

Finally, replace the line where you add `op` to the queue with these two lines:

```
queue.addOperation(downloadOp)
queue.addOperation(tiltShiftOp)
```

Even though you said that the tilt shift depends on the download, you still need to add both operations to the queue. The queue will keep track of dependencies and only start the tilt shift operation once the download is complete.

Build and run the app. You should see a list of tilt shifted images!



Custom completion handler

The code as currently written is using the default `completionBlock` provided by the `Operation` class. You're having to do a little extra work there to grab the image and dispatch back to the main queue. In a case like this, you may want to consider adding a custom completion block.

Back in **TiltShiftOperation.swift**, add a new optional class-level property at the top of the class to store a custom completion handler:

```
/// Callback which will be run *on the main thread*
/// when the operation completes.
var onImageProcessed: ((UIImage?) -> Void)?
```

Then, at the very end of the `main()` method, after assigning the `outputImage`, call that completion handler *on the main thread*:

```
if let onImageProcessed = onImageProcessed {
  DispatchQueue.main.async { [weak self] in
    onImageProcessed(self?.outputImage)
  }
}
```

If you add that extra bit of code then back in **TiltShiftTableViewController.swift**, in `tableView(_:cellForRowAt:)`, you can replace the entire completion block code with this:

```
tiltShiftOp.onImageProcessed = { image in
  guard let cell = tableView.cellForRow(at: indexPath)
    as? PhotoCell else {
    return
  }

  cell.isLoading = false
  cell.display(image: image)
}
```

While there's no functional or performance difference from those changes, it does make working with your operation a bit nicer for the caller. You've removed confusion over any possible retain cycle and ensured that they're properly working on the main UI thread automatically.

It's very important that you **document** the fact that your completion handler is being run on the main UI thread instead of the operation queue's provided thread.

The end user needs to know you're switching threads on them so they don't do something that could impact the application.

Notice how there are three `/` characters in the comment shown. If you use that syntax, then Xcode will display that comment as the summary of the property in the Quick Help Inspector. Xcode supports limited styling as well, which means that the text *on the main thread* will actually be italicized in the help display.

Where to go from here?

Throughout this chapter, you've learned how to tie the start of one operation to the completion of another. Consider where, in your existing apps, you could implement operations and operation dependencies to better modularize your app and remove the Pyramid of Doom indentation that you've likely implemented.

Mark Complete

10. Canceling Operations →

← 8. Asynchronous Operations