Up to date for iOS 12, Xcode 10 & Swift 4.2

Concurrency by Tutorials

FIRST EDITION

Multithreading in Swift with GCD and Operations

By the raywenderlich.com Tutorial Team

Scott Grosch

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

Welcome to *Concurrency by Tutorials*!

Building a modern iOS, macOS or tvOS app is absolutely a bliss these days, especially with the prevalence of Swift as the recommended language by Apple. But as our apps become larger and performance becomes more and more critical to our app's consumers, learning how to efficiently utilize concurrency in your apps is one of the most important things you could do.

Concurrency is the concept of multiple things, or pieces of work, running at the same time. With the addition of CPU cores in our devices, knowing how to properly utilize your customer's hardware to the maximum is absolutely a must. Unfortunately, proper concurrency in iOS apps is one of the lesser-known, lower-level topics, that every developer wants to (and should) understand properly, but is usually intimidated by.

This is where *Concurrency by Tutorials* comes to the rescue! In this book, you'll learn everything there is to know about how to write performant and concurrent code for your iOS apps.

We are pleased to offer you this sample from the full *Concurrency by Tutorials* book that will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

This sample includes:

Chapter 1: Introduction: Get a quick overview of what concurrency is and why you might want to use it.

Chapter 2: GCD vs. Operations: GCD vs.Operations: Concurrency can be handled by either Grand Central Dispatch (GCD) or Operations. Learn about the differences between the two and why you might choose one over the other.

Chapter 3: Queues & Threads: This chapter teaches you how to use a GCD queue to offload work from the main thread. You'll also learn what a "thread" is.

The book is ready for purchase at:

- https://store.raywenderlich.com/products/concurrency-by-tutorials.

Enjoy!

The *Concurrency by Tutorials* Team

Concurrency by Tutorials

By Scott Grosch

Notice of Rights

Notice of Liability

Trademarks

Dedications

"This book is dedicated to my wife and daughter, as well as to my parents who always made sure a good education was a priority."

— *Scott Grosch*

About the Author

Scott Grosch is the author of this book. He has been involved with iOS app development since the first release of the public SDK from Apple. He mostly works with a small set of clients on a couple large apps. During the day, Scott is a Solutions Architect at a Fortune 500 company in the Pacific Northwest. At night, he's still working on figuring out how to be a good parent to a toddler with his wife.

About the Editors

Marin Bencevic is the tech editor of this book. He is a Swift and Unity developer who likes to work on cool iOS apps and games, nerd out about programming, learn new things and then blog about it. Mostly, though, he just causes SourceKit crashes. He also has a chubby cat.

Shai Mishali is the Final Pass Editor of this book. He's the iOS Tech Lead for Gett, the global on-demand mobility company; as well as an international speaker, and a highly active open-source contributor and maintainer on several high-profile projects - namely, the RxSwift Community and RxSwift projects. As an avid enthusiast of hackathons, Shai took 1st place at BattleHack Tel-Aviv 2014, BattleHack World Finals San Jose 2014, and Ford's Developer Challenge Tel-Aviv 2015. You can find him on GitHub and Twitter @freak4pc.

Manda Frederick is the editor of this book. She has been involved in publishing for over 10 years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

What You Need

To follow along with this book, you'll need the following:

- A Mac running **macOS Mojave** (10.14) or later. Earlier versions might work, but they're untested.
- **Xcode 10.1 or later**. Xcode is the main development tool for iOS. You'll need Xcode 10.1 or later for the tasks in this book. You can download the latest version of Xcode from Apple's developer site here: apple.co/2asi58y
- **An intermediate level knowledge of Swift**. This book teaches concurrency when building iOS applications using Swift. You could use the knowledge acquired in this book for your Objective-C codebase, but this book won't include any Objective-C examples. You could also use this knowledge for your macOS, tvOS and watchOS apps, but like Objective-C, this book won't include any examples for these platforms.

If you want to try things out on a physical iOS device, you’ll need a developer account with Apple, which you can obtain for free. However, all the sample projects in this book will work just fine in the iOS Simulator bundled with Xcode, so the paid developer account is completely optional.

Book License

By purchasing *Concurrency by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Concurrency by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Concurrency by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: “Artwork/images/designs: from *Concurrency by Tutorials*, available at www.raywenderlich.com”.
- The source code included in *Concurrency by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Concurrency by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an “as is” basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action or contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Chapter 1: Introduction

Performance. Responsiveness. They're not sexy tasks. When done properly, nobody is going to thank you. When done incorrectly, app retention is going to suffer and you'll be dinged during your next yearly performance review.

There are a multitude of ways in which an app can be optimized for speed, performance and overall responsiveness. This book will focus on the topic of **concurrency**.

What is concurrency?

Wikipedia defines concurrency as "the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units." What this means is looking at the logic of your app to determine which pieces can run at the same time, and possibly in a random order, yet still result in a correct implementation of your data flow.

Moderns devices almost always have more than a single CPU, and Apple's iPhones have been dual core since 2011. Having more than one core means they are capable of running more than a single task at the same time. By splitting your app into logical "chunks" of code you enable the iOS device to run multiple parts of your program at the same time, thus improving overall performance.

Why use concurrency?

It's critical to ensure that your app runs as smoothly as possible and that the end user is not ever forced to wait for something to happen. A second is a minuscule amount of time for most everything not related to a computer. However, if a human has to wait a second to see a response after taking an action on a device like an iPhone, it feels like an eternity. "It's too slow" is one of the main contributors to your app being uninstalled.

Scrolling through a table of images is one of the more common situations wherein the end user will be impacted by the lack of concurrency. If you need to download an image from the network, or perform some type of image processing before displaying it, the scrolling will stutter and you'll be forced to display multiple "busy" indicators instead of the expected image.

A beneficial side effect to using concurrency is that it helps you to spend a bit more time thinking about your app's overall architecture. Instead of just writing massive methods to "get the job done" you'll find yourself naturally writing smaller, more manageable methods that can run concurrently.

How to use concurrency

That's the focus of this book! At a high level you need to structure your app so that some tasks can run at the same time. Multiple tasks that modify the same resource (i.e., variable) can't run at the same time, unless you make them thread safe.

Tasks which access different resources, or *read-only* shared resources, can all be accessed via different threads to allow for much faster processing.

This book will focus on the two main ways that iOS provides you with the ability to run code concurrently. The first section on **Grand Central Dispatch** will cover the common scenarios where you will find yourself being able to implement concurrency. You'll learn how to run tasks in the background, how to group tasks together and how to handle restricting the number of tasks that can run at once. By the end of the first section you'll also have a strong grasp of the dangers of concurrency and how to avoid them.

In the second section you'll focus on the `Operation` class. Built on top of Grand Central Dispatch, operations allow for the handling of more complex scenarios such as reusable code to be run on a background thread, having one thread depend on another, and even canceling an operation before it's started or completed.

Most modern programming languages provide for some form of concurrency and Swift is of course no exception. Different languages use widely different mechanisms for handling concurrency. C# and Typescript, for example use an *async/await* pattern, whereas Swift uses closures to handle what runs on another thread. Swift 5 originally had plans to implement the more common *async/await* pattern but it was removed from the specification until the next release.

Where to go from here?

Well to the next page of course! Hopefully as you work through the following chapters you'll gain an appreciation for what concurrency can do for your app and why your end users will appreciate the extra effort you put into making the app perform as fast as possible. Knowing when to use Grand Central Dispatch as opposed to an `Operation` subclass early in the app lifecycle will save you hours of rework down the road.

Chapter 2: GCD & Operations

There are two APIs that you'll use when making your app concurrent: **Grand Central Dispatch**, commonly referred to as **GCD**, and **Operations**. These are neither competing technologies nor something that you have to exclusively pick between. In fact, Operations are built on top of GCD!

Grand Central Dispatch

GCD is Apple's implementation of C's **libdispatch** library. Its purpose is to queue up **tasks** — either a method or a closure — that can be run in parallel, depending on availability of resources; it then executes the tasks on an available processor core.

> **Note**: Apple's documentation sometimes refers to a **block** in lieu of **closure**, since that was the name used in Objective-C. You can consider them interchangeable in the context of concurrency.

While GCD uses threads in its implementation, you, as the developer, do not need to worry about managing them yourself. GCD's tasks are so lightweight to enqueue that Apple, in its 2009 technical brief on GCD, stated that only 15 instructions are required for implementation, whereas creating traditional threads could require several hundred instructions.

All of the tasks that GCD manages for you are placed into GCD-managed **first-in, first-out** (FIFO) queues. Each task that you submit to a queue is then executed against a pool of threads fully managed by the system.

> **Note**: There is no guarantee as to which thread your task will execute against.

Synchronous and asynchronous tasks

Work placed into the queue may either run **synchronously** or **asynchronously**. When running a task synchronously, your app will wait and block the current run loop until execution finishes before moving on to the next task. Alternatively, a task that is run asynchronously will start, but return execution to your app immediately. This way, the app is free to run other tasks while the first one is executing.

> **Note**: It's important to keep in mind that, while the queues are FIFO based, that does not ensure that tasks will finish in the order you submit them. The FIFO procedure applies to when the task *starts*, not when it finishes.

In general, you'll want to take any long-running non-UI task that you can find and make it run asynchronously in the background. GCD makes this very simple via closures with a few lines of code, like so:

```
// Class level variable
let queue = DispatchQueue(label: "com.raywenderlich.worker")

// Somewhere in your function
queue.async {
  // Call slow non-UI methods here

  DispatchQueue.main.async {
    // Update the UI here
  }
}
```

You'll learn all about `DispatchQueue` in Chapter 3, "Queues & Threads." In general, you create a queue, submit a task to it to run asynchronously on a background thread, and, when it's complete, you delegate the code back to the main thread to update the UI.

Serial and concurrent queues

The queue to which your task is submitted also has a characteristic of being either **serial** or **concurrent**. Serial queues only have a single thread associated with them and thus only allow a single task to be executed at any given time. A concurrent queue, on the other hand, is able to utilize as many threads as the system has resources for. Threads will be created and released as necessary on a concurrent queue.

> **Note**: While you can tell iOS that you'd like to use a concurrent queue, remember that there is no *guarantee* that more than one task will run at a time. If your iOS device is completely bogged down and your app is competing for resources, it may only be capable of running a single task.

Asynchronous doesn't mean concurrent

While the difference seems subtle at first, just because your tasks are *asynchronous* doesn't mean they will run *concurrently*. You're actually able to submit asynchronous tasks to either a serial queue or a concurrent queue. Being synchronous or asynchronous simply identifies whether or not the queue on which you're running the task must wait for the task to complete before it can spawn the next task.

On the other hand, categorizing something as *serial versus concurrent* identifies whether the queue has a *single* thread or *multiple* threads available to it. If you think about it, submitting three asynchronous tasks to a serial queue means that each task has to completely finish before the next task is able to start as there is only one thread available.

In other words, a task being synchronous or not speaks to the *source* of the task. Being serial or concurrent speaks to the *destination* of the task.

Operations

GCD is great for common tasks that need to be run a single time in the background. When you find yourself building functionality that should be reusable — such as image editing operations — you will likely want to encapsulate that functionality into a class. By subclassing `Operation`, you can accomplish that goal!

Operation subclassing

`Operations` are fully-functional classes that can be submitted to an `OperationQueue`, just like you'd submit a closure of work to a `DispatchQueue` for GCD. Because they're classes and can contain variables, you gain the ability to know what state the operation is in at any given point.

Operations can exist in any of the following states:

- `isReady`
- `isExecuting`

- `isCancelled`
- `isFinished`

Unlike GCD, an operation is run synchronously by default, and getting it to run asynchronously requires more work. While you can directly execute an operation yourself, that's almost never going to be a good idea due to its synchronous nature. You'll want to get it off of the main thread by submitting it to an `OperationQueue` so that your UI performance isn't impacted.

Bonus features

But wait, there's more! Operations provide greater control over your tasks as you can now handle such common needs as cancelling the task, reporting the state of the task, wrapping asynchronous tasks into an operation and specifying dependences between various tasks. Chapter 6, "Operations," will provide a more in-depth discussion of using operations in your app.

BlockOperation

Sometimes, you find yourself working on an app that heavily uses operations, but find that you have a need for a simpler, GCD-like, closure. If you don't want to also create a `DispatchQueue`, then you can instead utilize the `BlockOperation` class.

`BlockOperation` subclasses `Operation` for you and manages the concurrent execution of one or more closures on the default global queue. However, being an actual `Operation` subclass lets you take advantage of all the other features of an operation.

> **Note**: Block operations run concurrently. If you need them to run serially, you'll need to setup a dispatch queue instead.

Which should you use?

There's no clear-cut directive as to whether you should use GCD or Operations in your app. GCD tends to be simpler to work with for simple tasks you just need to execute and forget. Operations provide much more functionality when you need to keep track of a job or maintain the ability to cancel it.

If you're just working with methods or chunks of code that need to be executed, GCD is a fitting choice. If you're working with objects that need to encapsulate data and functionality then you're more likely to utilize Operations. Some developers even go to the extreme of saying that you should always use Operations because it's built on top of GCD, and Apple's guidance says to always use the highest level of abstraction provided.

At the end of the day, you should use whichever technology makes the most sense at the time and provides for the greatest long-term sustainability of your project, or specific use-case.

In the next chapter, you'll take a deep dive into how Grand Central Dispatch works, learn about the difference between threads and queues, and identify some of the complexities that can occur when implementing concurrency in your app.

Chapter 3: Queues & Threads

Dispatch queues and threads have been mentioned a couple of times now, and you're probably wondering what they are at this point. In this chapter, you'll get a much deeper understanding of what Dispatch queue and Threads are, and how to best incorporate them in your development workflow.

Threads

You've probably heard the term **multithreading** at some point, yes? A **thread** is really short for **thread of execution**, and it's how a running **process** splits tasks across resources on the system. Your iOS app is a process that runs multiple tasks by utilizing multiple threads. You can have as many threads executing at once as you have cores in your device's CPU.

There are many advantages to splitting your app's work into multiple threads:

- **Faster execution**: By running tasks on threads, it's possible for work to be done *concurrently*, which will allow it to finish faster than running everything serially.
- **Responsiveness**: If you only perform user-visible work on the main UI thread, then users won't notice that the app slows down or freezes up periodically due to work that could be performed on another thread.
- **Optimized resource consumption**: Threads are highly optimized by the OS.

Sounds great, right? More cores, more threads, faster app. I bet you're ready to learn how to create one, right? Too bad! In reality, you should never find yourself needing to create a thread explicitly. The OS will handle all thread creation for you using higher abstractions.

Apple provides the APIs necessary for thread management, but if you try to directly manage them yourself, you could in fact degrade, rather than improve, performance. The OS keeps track of many statistics to know when it should and should not allocate or destroy threads. Don't fool yourself into thinking it's as simple as spinning up a thread when you want one. For those reasons, this book will not cover direct thread management.

Dispatch queues

The way you work with threads is by creating a `DispatchQueue`. When you create a queue, the OS will potentially create and assign one or more threads to the queue. If existing threads are available, they can be reused; if not, then the OS will create them as necessary.

Creating a dispatch queue is pretty simple on your part, as you can see in the example below:

```
let label = "com.razeware.mycoolapp.networking"
let queue = DispatchQueue(label: label)
```

Phew, fairly easy, eh? Normally, you'd put the text of the label directly inside the initializer, but it's broken into separate statements for the sake of brevity.

The `label` argument simply needs to be any unique value for identification purposes. While you *could* simply use a UUID to guarantee uniqueness, it's best to use a reverse-DNS style name, as shown above (e.g. `com.company.app`), since the label is what you'll see when debugging and it's helpful to assign it meaningful text.

The main queue

When your app starts, a `main` dispatch queue is automatically created for you. It's a serial queue that's responsible for your UI. Because it's used so often, Apple has made it available as a class variable, which you access via `DispatchQueue.main`. You *never* want to execute something synchronously against the main queue, unless it's related to actual UI work. Otherwise, you'll lock up your UI which could potentially degrade your app performance.

If you recall from the previous chapter, there are two types of dispatch queues: *serial* or *concurrent*. The default initializer, as shown in the code above, will create a serial queue wherein each task must complete before the next task is able to start.

In order to create a concurrent queue, simply pass in the `.concurrent` attribute, like so:

```
let label = "com.razeware.mycoolapp.networking"
let queue = DispatchQueue(label: label, attributes: .concurrent)
```

Concurrent queues are so common that Apple has provided six different global concurrent queues, depending on the **Quality of service (QoS)** the queue should have.

Quality of service

When using a concurrent dispatch queue, you'll need to tell iOS how important the tasks are that get sent to the queue so that it can properly prioritize the work that needs to be done against all the other tasks that are clamoring for resources. Remember that higher-priority work has to be performed faster, likely taking more system resources to complete and requiring more energy than lower-priority work.

If you just need a concurrent queue but don't want to manage your own, you can use the `global` class method on `DispatchQueue` to get one of the pre-defined global queues:

```
let queue = DispatchQueue.global(qos: .userInteractive)
```

As mentioned above, Apple offers six quality of service classes:

.userInteractive

The `.userInteractive` QoS is recommended for tasks that the user *directly interacts* with. UI-updating calculations, animations or anything needed to keep the UI responsive and fast. If the work doesn't happen quickly, things may appear to freeze. Tasks submitted to this queue should complete virtually instantaneously.

.userInitiated

The `.userInitiated` queue should be used when the *user* kicks off a task from the UI that needs to happen immediately, but can be done asynchronously. For example, you may need to open a document or read from a local database. If the user clicked a button, this is probably the queue you want. Tasks performed in this queue should take a few seconds or less to complete.

.utility

You'll want to use the `.utility` dispatch queue for tasks that would typically include a progress indicator such as long-running computations, I/O, networking or continuous data feeds. The system tries to balance responsiveness and performance with energy efficiency. Tasks can take a few seconds to a few minutes in this queue.

.background

For tasks that the user is not directly aware of you should use the `.background` queue. They don't require user interaction and aren't time sensitive. Prefetching, database maintenance, synchronizing remote servers and performing backups are all great examples. The OS will focus on energy efficiency instead of speed. You'll want to use this queue for work that will take significant time, on the order of minutes or more.

.default and .unspecified

There are two other possible choices that exist, but you should not use explicitly. There's a `.default` option, which falls between `.userInitiated` and `.utility` and is the default value of the `qos` argument. It's not intended for you to directly use. The second option is `.unspecified`, and exists to support legacy APIs that may opt the thread out of a quality of service. It's good to know they exist, but if you're using them, you're almost certainly doing something wrong.

> **Note**: Global queues are always concurrent and first-in, first-out.

Inferring QoS

If you create your own concurrent dispatch queue, you can tell the system what the QoS is via its initializer:

```
let queue = DispatchQueue(label: label,
                          qos: .userInitiated,
                          attributes: .concurrent)
```

However, this is like arguing with your spouse/kids/dogs/pet rock: Just because you say it doesn't make it so! The OS will pay attention to what type of tasks are being submitted to the queue and make changes as necessary.

If you submit a task with a higher quality of service than the queue has, the queue's level will increase. Not only that, but all the operations enqueued will also have their priority raised as well.

If the current context is the main thread, the inferred QoS is `.userInitiated`. You can specify a QoS yourself, but as soon as you'll add a task with a higher QoS, your queue's QoS service will be increased to match it.

Adding task to queues

Dispatch queues provide both `sync` and `async` methods to add a task to a queue. Remember that, by **task**, I simply mean, "*Whatever block of code you need to run.*" When your app starts, for example, you may need to contact your server to update the app's state. That's not user initiated, doesn't need to happen immediately and depends on networking I/O, so you should send it to the global utility queue:

```
DispatchQueue.global(qos: .utility).async { [weak self] in
  guard let self = self else { return }

  // Perform your work here
  // ...

  // Switch back to the main queue to
  // update your UI
  DispatchQueue.main.async {
    self.textLabel.text = "New articles available!"
  }
}
```

There are two key points you should take away from the above code sample. First, there's nothing special about a `DispatchQueue` that nullifies the closure rules. You still need to make sure that you're properly handling the closure's captured variables, such as `self`, if you plan to utilize them.

Strongly capturing `self` in a GCD `async` closure will not cause a reference cycle (e.g. a retain cycle) since the whole closure will be deallocated once it's completed, but it *will* extend the lifetime of `self`. For instance, if you make a network request from a view controller that has been dismissed in the meantime, the closure will still get called. If you capture the view controller weakly, it will be `nil`. However, if you capture it strongly, the view controller will remain alive until the closure finishes its work. Keep that in mind and capture weakly or strongly based on your needs.

Second, notice how updates to the UI are dispatched to the `main` queue *inside* the dispatch to the background queue. It's not only OK, but very common, to nest `async` type calls inside others.

> **Note**: You should *never* perform UI updates on any queue other than the main queue. If it's not documented what queue an API callback uses, dispatch it to the main queue!

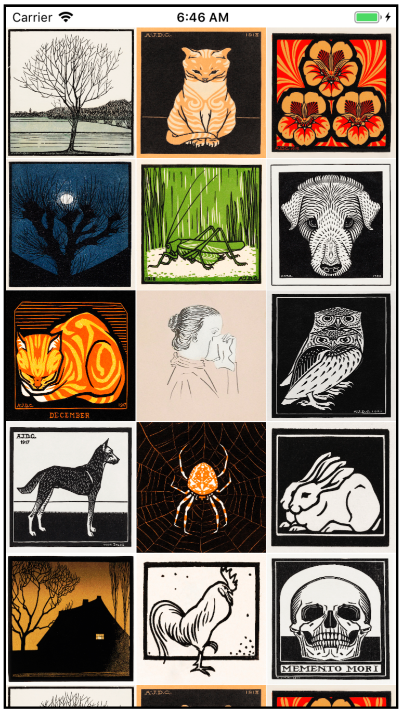
Use extreme caution when submitting a task to a dispatch queue synchronously. If you find yourself calling the `sync` method, instead of the `async` method, think once or twice whether that's really what you should be doing. If you submit a task synchronously to the current queue, which blocks the current queue, and your task tries to access a resource in the current queue, then your app will **deadlock**, which is explained more in Chapter 5, "Concurrency Problems." Similarly, if you call `sync` from the `main` queue, you'll block the thread that updates the UI and your app will appear to freeze up.

> **Note**: Never call `sync` from the main thread, since it would block your main thread and could even potentially cause a deadlock.

Image loading example

You've been inundated with quite a bit of theoretical concepts at this point. Time to see an actual example!

In the downloadable materials for this book, you'll find a starter project for this chapter. Open up the **Concurrency.xcodeproj** project. Build and run the app. You'll see some images slowly load from the network into a `UICollectionView`. If you try to scroll the screen while the images are loading, either nothing will happen or the scrolling will be very slow and choppy, depending on the speed of the device you are using.



Open up **CollectionViewController.swift** and take a look at what's going on. When the view loads, it just grabs a static list of image URLs to be displayed. In a production app, of course, you'd likely be making a network call at this point to generate a list of items to display, but for this example it's easier to hardcode a list of images.

The `collectionView(_:cellForItemAt:)` method is where the trouble happens. You can see that when a cell is ready to be displayed a call is made via one of `Data`'s constructors to download the image and then it's assigned to the cell. The code looks simple enough, and it is what most starting iOS developers would do to download an image, but you saw the results: a choppy, underperforming UI experience!

Unless you slept through the previous pages of explanation, you know by now that the work to download the image, which is a network call, needs to be done on a separate thread from the UI.

> **Mini-challenge**: Which queue do you think should handle the image download? Take a look back a few pages and make your decision.

Did you pick either `.userInteractive` or `.userInitiated`? It's tempting to do because the end result is directly visible to the user but the reality is if you used that logic then you'd never use any other queue. The proper choice here is to use the `.utility` queue. You've got no control over how long a network call will take to complete and you want the OS to properly balance the speed vs. battery life of the device.

Using a global queue

Create a new method in `CollectionViewController` that starts off like so:

```
private func downloadWithGlobalQueue(at indexPath: IndexPath) {
  DispatchQueue.global(qos: .utility).async { [weak self] in
  }
}
```

You'll eventually call this from `collectionView(_:cellForItemAt:)` to perform the actual image processing. Begin by determining which URL should be loaded. Since the list of URLs are part of `self`, you'll need to handle normal closure capture semantics. Add the following code *inside* the `async` closure:

```
guard let self = self else {
  return
}

let url = self.urls[indexPath.item]
```

Once you know the URL to load, you can use the same `Data` initializer you previously used. Even though it's an synchronous operation that's being performed, it is running on a separate thread and thus the UI isn't impacted. Add the following to the end of the closure:

```
guard let data = try? Data(contentsOf: url),
      let image = UIImage(data: data) else {
  return
}
```

Now that you've successfully downloaded the contents of the URL and turned it into a `UIImage`, it's time to apply it to the collection view's cell. Remember that updates to the UI can only happen on the main thread! Add this `async` call to the end of the closure:

```
DispatchQueue.main.async {
  if let cell = self.collectionView.cellForItem(at: indexPath) as?
PhotoCell {
    cell.display(image: image)
  }
}
```

Notice that the bare minimum of code is being sent back to the main thread. Do every bit of work that you can before dispatching to the main queue so that your UI remains as responsive as possible. Is the cell assignment confusing you? Why not just pass the actual `PhotoCell` to this method instead of an `IndexPath`?

Consider the nature of what you're doing here. You've offloaded the configuration of the cell to an asynchronous process. While the network download is occurring, the user is very likely doing *something* with your app. In the case of a `UITableView` or `UICollectionView`, that probably means that they're doing some scrolling. By the time the network call finishes, the cell might have been reused for another image, or it might have been disposed of completely. By calling `cellForItem(at:)`, you're grabbing the cell at the time you're ready to update it. If it still exists and if it's still on the screen, then you'll update the display. If it's not, then `nil` will be returned.

Had you instead simply passed in a `PhotoCell` and directly interacted with that object, you'd have discovered that random images are placed in random cells, and you'll see the same image repeated multiple times as you scroll around.

Now that you've got a proper image download and cell configuration method, update `collectionView(_:cellForItemAt:)` to call it. Replace everything in-between creating and returning the cell with these two lines of code:

```
cell.display(image: nil)
downloadWithGlobalQueue(at: indexPath)
```

Using built-in methods

You can see how simple the above changes were to vastly improve the performance of your app. However, it's not always necessary to grab a dispatch queue yourself. Many of the standard iOS libraries handle that for you. Add the following method to `CollectionViewController`:

```
private func downloadWithUrlSession(at indexPath: IndexPath) {
  URLSession.shared.dataTask(with: urls[indexPath.item]) {
    [weak self] data, response, error in

    guard let self = self,
          let data = data,
          let image = UIImage(data: data) else {
      return
    }

    DispatchQueue.main.async {
      if let cell = self.collectionView
        .cellForItem(at: indexPath) as? PhotoCell {
        cell.display(image: image)
      }
    }
  }.resume()
}
```

Notice how, this time, instead of getting a dispatch queue, you directly used the `dataTask` method on `URLSession`. The code is *almost* the same, but it handles the download of the data for you so that you don't have to do it yourself, nor do you need to grab a dispatch queue. Always prefer to use the system provided methods when they are available as it will make your code not only more future-proof but easier to read for other developers. A junior programmer might not understand what the dispatch queues are, but they understand making a network call.

If you call `downloadWithUrlSession(at:)` instead of `downloadWithGlobalQueue(at:)` in `collectionView(_:cellForItemAt:)` you should see the exact same result after building and running your app again.

Where to go from here?

At this point, you should have a good grasp of what dispatch queues are, what they're used for and how to use them. Play around with the code samples from above to ensure you understand how they work.

Consider passing the `PhotoCell` into the download methods instead of just passing in the `IndexPath` to see a common type of bug in practice.

The sample app is of course somewhat contrived so as to easily showcase how a `DispatchQueue` works. There are many other performance improvements that could be made to the sample app but those will have to wait for Chapter 7, "Operation Queues."

Now that you've seen the benefits, the next chapter will introduce you to the dangers of implementing concurrency in your app.

Where to Go From Here?

We hope you enjoyed this sample of *Concurrency by Tutorials*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

- Chapter 1: Introdction: Get a quick overview of what concurrency is and why you might want to use it.
- Chapter 2: GCD vs.Operations: Concurrency can be handled by either Grand Central Dispatch (GCD) or Operations. Learn about the differences between the two and why you might choose one over the other.
- Chapter 3: Queues & Threads: This chapter teaches you how to use a GCD queue to offload work from the main thread. You'll also learn what a "thread" is.
- Chapter 4: Groups & Semaphores: In the previous chapter you learned about how queues work. In this chapter you'll expand that knowledge to learn how to submit multiple tasks to a queue which need to run together as a "group" so that you can be notified when they have all completed. You'll also learn how to wrap an existing API so that you can call it asynchronously.
- Chapter 5: Concurrency Problems: By now you know how GCD can make your app so much fater. This chapter will show you some of the dangers of concurrency if you're not careful, and how to avoid them.
- Chapter 6: Operations: In this chapter you'll switch gears and start learning about the Operations class, which allows for much more powerful control over your concurrent tasks.

- Chapter 7: Operation Queues: Similar to the Dispatch Queues you learned about back in chapter 3, the Operation class uses an OperationQueue to perform a similar function.
- Chapter 8: Asynchronous Operations: Now that you can create an Operation and submit it to a queue, you'll learn how to make the operation itself asynchronous. While not something you'll do regularly, it's important to know that it's possible.
- Chapter 9: Operation Dependencies: The "killer feature" of Operations is being able to tell the OS that one operation is dependant on another and shouldn't being until the dependency has finished.
- Chapter 10: Canceling Operations: There are times when you need to stop an operation that is running, or has yet to start. This chapter will teach you the concepts that you need to be aware of to support cancelation.
- Chapter 11: Core Data: Core Data works well with concurrency as long as you keep a few simple rules in mind. This chapter will teach you how to make sure that your Core Data app is able to handle concurrency properly.
- Chapter 12: Thread Sanitizer: Data races occur when multiple threads access the same memory without synchronization and at least one access is a write. This chapter will teach you how to use Apple's Thread Sanitizer to detect data races.

We hope you enjoy the book!

— The *Concurrency by Tutorials* team

Dive into Concurrency in Cocoa apps!

Concurrency is the concept of multiple things, or pieces of work, running at the same time. With the addition of CPU cores in our devices, knowing how to properly utilize your customer's hardware to the maximum is absolutely a must. However, proper concurrency in Cocoa apps is one of the lesser-known topics that every developer wants to (and should) understand properly, but is usually intimidated by.

This is where Concurrency by Tutorials comes to the rescue! In this book, you'll learn everything there is to know about how to write performant and concurrent code for your Cocoa apps.

Who This Book Is For

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make their app efficiently perform tasks without affecting performance, and how to properly divide work to utilize hardware to the fullest extent.

Topics Covered in Concurrency by Tutorials:

- **What & Why:** Learn what is Concurrency and why would you even want to utilize it in your apps?
- **Grand Central Dispatch:** Learn about Apple's implementation of C's libdispatch, also known as GCD, it's one of the simplest ways to queue up tasks to be run in parallel.
- **Operations & Operation Queues:** When GCD doesn't quite cut it, you'll learn how to further customize and reuse your concurrent work using Operations and Operation Queues.
- **Common Concurrency Problems:** Learn about some of the problems you could face while developing concurrent applications, such as Race Conditions, Deadlocks, and more.
- **Threads & Thread Sanitizer:** Understand various threading-related concepts and how these connect to the knowledge you've accumulated throughout this book. You'll also learn how to use Thread Sanitizer to ease your debugging when things go wrong.

This book is sure to make you a pro in building concurrent and performant appllications, and finally understanding how these lower-level APIs work to the fullest, pushing your app to the top!

About Scott Grosch

Scott Grosch has been involved with iOS app development since the first release of the public SDK from Apple, and spends his days as a Solutions Architect at a Fortune 500 company in the Pacific Northwest.