2 GCD vs. Operations

Written by Scott Grosch

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

`Operation`s 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 dependencies 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.

Where to go from here?

To the next chapter, of course! The rest of the book will examine, in detail, both Grand Central Dispatch and Operations. By the time you've completed the book you'll have a solid grasp on what both options provide, as well as a better idea on how to choose one over the other.

Completed

3. Queues & Threads →

← 1. Introduction