Operation Queues

Quick Guides for Masterminds

J.D Gauchat
www.jdgauchat.com

Cover Illustration by **Patrice Garden**
www.smartcreativz.com

The content of this guide is a collection of excerpts from the book iOS Apps for Masterminds. For more information, visit www.formasterminds.com.

The source code for this eBook is available at **www.formasterminds.com**

Copyright Registration Number: 1140725

1st Edition 2018

What is Inside

This guide will teach you how to use operations in your iOS applications. After reading this guide, you will know how to generate operations using Foundation's operations and how to define concurrent tasks using Grand Central Dispatch.

About this Guide

This guide is a collection of excerpts from the book iOS Apps for Masterminds. The information included in this guide will help you understand a particular aspect of app development in iOS, but it will not teach you everything you need to know to develop an app for Apple devices. If you need a complete course on app development for iOS, read our book iOS Apps for Masterminds. For more information, visit our website at www.formasterminds.com.

What Do You Need

This guide assumes that you have a basic knowledge of app development, Xcode, and the Swift language. If you don't know how to program in Swift or how to create an application with Xcode, download our guides Introduction to Swift and Interface Builder. For a complete course on app development for iOS, read our book iOS Apps for Masterminds.

IMPORTANT: Supporting links, examples, projects, videos, and resources are available at www.formasterminds.com. Apple's official documentation is available at developer.apple.com. Frameworks and APIs references are available at developer.apple.com/reference.

The Basics: To create the files for the examples of this guide, you need an Apple computer and Xcode. Xcode is a development suite available for free at developer.apple.com. For more information on the requirements for app development, read our free guide App Development.

Table of Contents

Operation Queues

Operations

iOS can separate pieces of code and execute them in simultaneous operations. Operations are objects used to wrap a set of statements that perform a task. These operations can then be put in a queue and executed in order, one after another, according to a level of priority assigned by the system or explicitly by the application. The advantage of this process is that the operations can be performed concurrently, which means they can take advantage of the hardware capacity to execute multiple instructions at the same time. For example, we may have a code that downloads a big file from the Internet and another code that shows the progress on the screen. In cases like this, we cannot wait for one code to be over to execute the other; we need the two pieces of code to work at the same time. Using operations, we can wrap the code to download the file in one operation and the code that shows a progress bar in another, and then make them work simultaneously.

Block Operations

Foundation includes the **Operation** class to create operation objects. The class provides all the necessary functionality to configure an operation, but we must create a subclass to include the statements the operation is required to perform. To simplify the process, the framework defines a standard subclass of **Operation** called **BlockOperation**. This subclass can create an operation from the statements provided by a closure. The following is the class' initializer.

BlockOperation(block: Block**)**—This initializer creates an operation with the statements specified by the closure assigned to the **block** attribute.

Because the **BlockOperation** class is a subclass of **Operation**, it has access to its properties and methods. The following are the most useful.

completionBlock—This property sets a closure with the statements we want to execute after the operation is completed.

queuePriority—This property sets the operation's priority. It is an enumeration of type **QueuePriority** with the values **veryLow**, **low**, **normal**, **high**, and **veryHigh**.

cancel()—This method asks the system to cancel the operation.

The operations have to be added to a queue to be scheduled for execution. For this purpose, Foundation includes the **OperationQueue** class. Once an instance of this class is created, we can use the following methods to manage the operations in the queue.

addOperation(Operation**)**—This method adds the operation specified by the operation attribute to the queue. The attribute is an object created from an **Operation** subclass.

addOperation(Block**)**—This method creates an **Operation** object from the statements provided by the attribute and adds it to the queue. The attribute is a closure with the statements we want to include in the operation.

cancelAllOperations()—This method cancels all the operations in the queue.

By default, iOS places the application's code in a single queue called *Main Queue*. If we want some code to be executed in a different queue, we have to create an operation and add it to a new queue.

Listing 1: *Adding operations to a queue*

```
import UIKit

class ViewController: UIViewController {
   override func viewDidLoad() {
      super.viewDidLoad()
      let operation = BlockOperation(block: {
         var total: Double = 1
         for f in 1..<100 {
            total = log(total + Double(f))
         }
         print("Total: \(total)")
      })
      let queue = OperationQueue()
      queue.addOperation(operation)
      print("Printed in the Main Queue")
   }
}
```

The operation is created by the **BlockOperation()** initializer. This initializer takes a closure with the statements we want to execute (in this case, a loop to calculate the logarithm of a sequence of values) and creates the operation. With the operation ready, we initialize a new **OperationQueue** object to represent the queue and add the operation to it.

When the code of Listing 1 is executed, the system creates the operation, adds it to a new queue, and runs it as soon as it can. At this point, the system is running two codes at the same time, the code in the main queue and the code in the new queue we have just created. As a result, we will see the message "Printed in the Main Queue" on the console first, and then the message produced by the operation, because the operation takes more time to complete.

This code guarantees that the code in the main queue is executed, no matter how much time the statements in the operation take to finish the process. If we wouldn't have created the operation and added it to a new queue, the system would have had to wait until the execution of the loop was over to execute the last statement and print the message on the console.

Do It Yourself: Create a new Single View Application project. Update the **ViewController** class of the initial view with the code of Listing 1 and run the application. The Xcode's console should print the message "Printed in the Main Queue" first and then another message with the result of the operation.

Main Queue

Some frameworks, like UIKit, can only work in the main queue. If we try to access the elements of the interface from a different queue, the application will produce unexpected results. The **OperationQueue** class offers the following type property to get a reference to the main queue when we need to add operations to it.

main—This type property returns a reference to the **OperationQueue** object representing the main queue.

Code inside a closure may sometimes be executed in a different queue. Every time we have to execute code inside an operation or a block of code, we have to consider whether that code can be executed in any queue or it only works properly in the main queue. If the code is required to be executed in the main queue, we have to insert it in an operation, get a reference to the main queue from the **main** property, and add the operation to it, as in the following example.

Listing 2: *Adding operations to the main queue*

```
import UIKit

class ViewController: UIViewController {
   @IBOutlet weak var totalLabel: UILabel!

   override func viewDidLoad() {
      super.viewDidLoad()
      let operation = BlockOperation(block: {
         var total: Double = 1
         for f in 1..<100 {
            total = log(total + Double(f))
         }
         let main = OperationQueue.main
         main.addOperation({
            self.totalLabel.text = "Total: \(total)"
            })
        })
        let queue = OperationQueue()
        queue.addOperation(operation)
        print("Printed in the Main Queue")
    }
}
```

In Listing 2, instead of printing a message to the console we update a label with the result produced by the loop. The elements of the interface can only be modified from the main queue, so we get a reference to this queue and then update the label from inside an operation added with the **addOperation()** method. This method creates a new operation on the fly with the code inside the block and adds it to the queue, simplifying the process. At the end, we get a similar result to the previous application; the loop is executed along with the rest of the code, but when it is over, the operation that updates the label is executed, showing the result on the screen.

Do It Yourself: Add a label to the initial view and connect it to the view controller with an Outlet called **totalLabel**. Update the **ViewController** class of the previous example with the code of Listing 2. Run the application. You should see the result produced by the loop on the screen.

Grand Central Dispatch

Apple has been developing frameworks since the introduction of its first personal computer. Over time, some of these frameworks were replaced and others improved, creating a huge SDK with some of the frameworks performing similar tasks but at a different technical level. This means that for some tasks we can select the framework we want to work with depending on the level of customization our app needs. Among these duplicated frameworks are two dedicated to performing simultaneous operations. These are the Foundation's operations studied before and a system called Grand Central Dispatch. GCD is a framework programmed in the C language that works like Operations in Foundation but at a lower level, offering a deeper access to the system.

Although Foundation's operations are sufficient for most of the scenarios we may find in a professional application, a few old frameworks programmed in Objective-C require the use of GCD to perform concurrent tasks and therefore both systems remain useful. For this reason, Apple decided to introduce an additional framework called *Dispatch* to make GCD easier to implement.

Dispatch Framework

The Dispatch framework includes an extensive list of classes to provide access to all the original features of GCD, but with a friendly API. To schedule an operation, all we need to do is to create an instance of the **DispatchQueue** class. The following is the class initializer.

DispatchQueue(label: String**)**—This initializer creates a queue identified with the value specified by the **label** attribute.

Once the **DispatchQueue** instance is created, we have to call its methods to add a task to the queue. The class includes two methods to dispatch tasks synchronously (the operation must finish before others are initiated) or asynchronously (multiple operations are executed simultaneously).

async(execute: Block**)**—This method asynchronously executes the statements in the closure specified by the **execute** attribute.

sync(execute: Block**)**—This method synchronously executes the statements in the closure specified by the **execute** attribute.

The Dispatch framework is just another way to perform multiple tasks simultaneously. The following example repeats the code we used before with Operations, but this time the task is performed with a **DispatchQueue** object.

Listing 3: *Dispatching tasks with GCD*

```
import UIKit

class ViewController: UIViewController {
   override func viewDidLoad() {
      super.viewDidLoad()
      let queue = DispatchQueue(label: "myqueue")
      queue.async(execute: {
         var total: Double = 1
         for f in 1..<100 {
            total = log(total + Double(f))
         }
         print("Total: \(total)")
      })
      print("Printed in the Main Queue")
   }
}
```

The **DispatchQueue** class also offers a type property to access the main queue.

main—This type property returns the **DispatchQueue** object referencing the main queue.

As we did before, the following example updates a previous code to perform a task in the main queue inside an asynchronous operation.

Listing 4: *Working on the main queue with GCD*

```
import UIKit

class ViewController: UIViewController {
   @IBOutlet weak var totalLabel: UILabel!

   override func viewDidLoad() {
      super.viewDidLoad()
      let queue = DispatchQueue(label: "myqueue")
      queue.async(execute: {
         var total: Double = 1
         for f in 1..<100 {
            total = log(total + Double(f))
         }
         let main = DispatchQueue.main
         main.sync(execute: {
            self.totalLabel.text = "Total: \(total)"
         })
      })
      print("Printed in the Main Queue")
   }
}
```

When the code of Listing 4 is executed, it creates a queue with a closure to calculate the logarithm of a series of values. When the operation is over, the task gets a reference to the main queue and performs a second task to update a label on the interface with the result. This last operation is performed synchronously since we are already inside an asynchronous operation.

Do It Yourself: These examples work exactly the same way as the examples introduced for Foundation's operations. As well as the example of Listing 2, the example of Listing 4 requires the interface to include a label connected to an Outlet called **totalLabel**.

Quick Reference

BlockOperation

BlockOperation(block: Block**)**—This initializer creates an operation with the statements specified by the closure assigned to the **block** attribute.

completionBlock—This property sets a closure with the statements we want to execute after the operation is completed.

queuePriority—This property sets the operation's priority. It is an enumeration of type **QueuePriority** with the values **veryLow**, **low**, **normal**, **high**, and **veryHigh**.

cancel()—This method asks the system to cancel the operation.

OperationQueue

main—This type property returns a reference to the **OperationQueue** object representing the main queue.

addOperation(Operation**)**—This method adds the operation specified by the operation attribute to the queue. The attribute is an object created from an **Operation** subclass.

addOperation(Block**)**—This method creates an **Operation** object from the statements provided by the attribute and adds it to the queue. The attribute is a closure with the statements we want to include in the operation.

cancelAllOperations()—This method cancels all the operations in the queue.

DispatchQueue

main—This type property returns the **DispatchQueue** object referencing the main queue.

DispatchQueue(label: String**)**—This initializer creates a queue identified with the value specified by the **label** attribute.

async(execute: Block**)**—This method asynchronously executes the statements in the closure specified by the **execute** attribute.

sync(execute: Block**)**—This method synchronously executes the statements in the closure specified by the **execute** attribute.