Pro iPhone Development with Swift 4

Design and Manage Top Quality Apps

Molly Maskrey
Wallace Wang

Apress®

Pro iPhone Development with Swift 4

Design and Manage Top Quality Apps

Molly Maskrey
Wallace Wang

Apress®

Pro iPhone Development with Swift 4: Design and Manage Top Quality Apps

Molly Maskrey
Parker, Colorado, USA

Wallace Wang
San Diego, California, USA

ISBN-13 (pbk): 978-1-4842-3380-1
ISBN-13 (electronic): 978-1-4842-3381-8
https://doi.org/10.1007/978-1-4842-3381-8

Library of Congress Control Number: 2018932359

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Technical Reviewer: Bruce Wade
Coordinating Editor: Jessica Vakili
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3380-1. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to everyone who has an idea for an app but didn't know what to do first or how to get started. First, believe in your idea. Second, trust that you have intelligence to achieve your dream even if you don't know how you'll get there. Third, keep learning and improving your skills all the time. Fourth, stay focused. Success will come one day as long as you persist and never give up on yourself.

Table of Contents

About the Authors

Molly Maskrey started as an electrical engineer in her 20s working for various large Aerospace companies including IBM Federal Systems, TRW (now Northrup-Grumman), Loral Systems, Lockheed-Martin, and Boeing. After successfully navigating the first dot.com boom, she realized that a break was in order, took took several years off, moved to Maui and taught windsurfing at the beautiful Kanaha Beach Park.

She moved back to Colorado in 2005 and, with Jennifer, formed Global Tek Labs, an iOS development and accessory design services company that is now one of the leading consulting services for new designers looking to create smart attachments to Apple devices.

In 2014 Molly and Jennifer formed Quantitative Bioanalytics Laboratories, a wholly owned subsidiary of Global Tek to bring high-resolution mobile sensor technology to physical therapy, elder balance and fall prevention, sports performance quantification and instrumented gait analysis (IGA). In a pivot, Molly changed the direction of QB Labs to a platform-based predictive analytics company seeking to democratize data science for smaller companies.

Molly's background includes advanced degrees in Electrical Engineering, Applied Mathematics, Data Science, and business development. Molly generally speaks at a large number of conferences throughout the year including the Open Data Science Conference (ODSC) 2017 West advancing the topic of moving analytics from the cloud to the fog for smart city initiatives. What fuels her to succeed is the opportunity to bring justice and equality to everyone whether it's addressing food insecurity, with her business partner, to looking at options for better management of mental health using empirical data and tools such as natural language processing, speech pattern recognition using neural networks, or analyzing perfusion in brain physiology.

Wallace Wang has written dozens of computer books over the years beginning with ancient MS-DOS programs like WordPerfect and Turbo Pascal, migrating to writing books on Windows programs like Visual Basic and Microsoft Office, and finally switching to Swift programming for Apple products like the Macintosh and iPhone.

When he's not helping people discover the joys of programming, he performs stand-up comedy and appears on two radio shows on KNSJ in San Diego (http://knsj.org) called "Notes From the Underground" and "Laugh In Your Face Radio."

He also writes a screenwriting blog called "The 15 Minute Movie Method" (http://15minutemoviemethod.com), a blog about the latest cat news on the Internet called "Cat Daily News" (http://catdailynews.com), and a blog about the latest trends in technology called "Top Bananas" (http://www.topbananas.com).

About the Technical Reviewer

Bruce Wade is a software engineer from British Columbia, Canada. He started software development when he was 16 years old by coding his first web site. He went on to study Computer Information Systems ad DeVry Institute of Technology in Calgary, then to further enhance his skills he studied Visual & Game Programming at The Art Institute Vancouver. Over the years he has worked for large corporations as well as several start-ups. His software experience has led him to utilize many different technologies including C/C++, Python, Objective-C, Swift, Postgres, and JavaScript. In 2012 he started the company Warply Designed to focus on mobile 2D/3D and OS X development. Aside from hacking out new ideas, he enjoys spending time hiking with his Boxer Rasco, working out, and exploring new adventures.

CHAPTER 1

Multithreaded Programming Using Grand Central Dispatch

While the idea of programming multithreaded functions in any environment may seem daunting at first (see Figure 1-1), Apple came up with a new approach that makes multithreaded programming much easier. **Grand Central Dispatch** comprises language features, runtime libraries, and system enhancements that provide systemic, comprehensive improvements to the support for concurrent code execution on multicore hardware in iOS and macOS.

M. Maskrey and W. Wang, *Pro iPhone Development with Swift 4*,
https://doi.org/10.1007/978-1-4842-3381-8_1

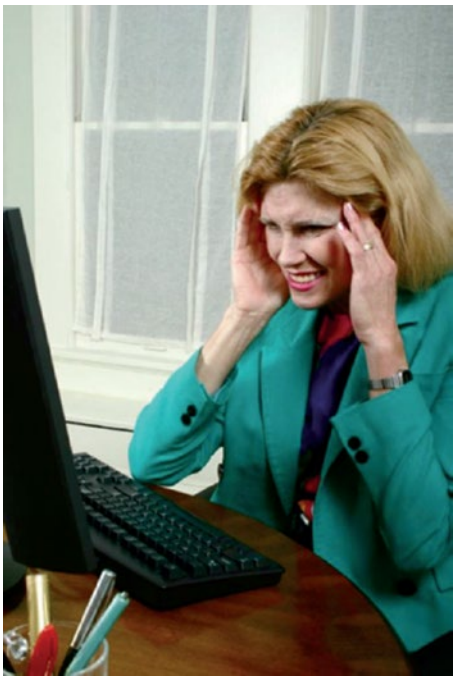

Figure 1-1. *Programming multithreaded applications can seem to be a disheartening experience*

A big challenge facing developers today is writing software able to perform complex actions in response to user input while remaining responsive, so that the user isn't constantly kept waiting while the processor does some behind-the-scenes task. That challenge has been with us all along; and in spite of the advances in computing technology that bring us faster CPUs, the problem persists. Look at the nearest computer screen; chances are that the last time you sat down to work at your computer, at some point, your workflow was interrupted by a spinning mouse cursor of some kind or another.

One of the reasons this has become so problematic is the way software is typically written: as a sequence of events to be performed sequentially. Such software can scale up as CPU speeds increase, but only to a certain point. As soon as the program gets stuck waiting for an external resource, such as a file or a network connection, the entire sequence of events is effectively paused. All modern operating systems now allow the use of multiple threads of execution within a program, so that even if a single thread is stuck waiting for a specific event, the other threads can keep going. Even so, many developers see multithreaded programming as a mystery and shy away from it.

Note A thread is a sequence of instructions managed independently by the operating system.

Apple provides Grand Central Dispatch (GCD) giving the developer an entirely new API for splitting up the work the application needs to do into smaller chunks that can be spread across multiple threads and, with the right hardware, multiple CPUs.

We access this API using Swift closures, providing a convenient way to structure interactions between different objects while keeping related code closer together in our methods.

Creating the SlowWorker Application

As a platform for demonstrating how GCD works, we'll create the SlowWorker application that consists of a simple interface driven by a single button and a text view. Click the button, and a synchronous task is immediately started, locking up the app for about ten seconds. Once the task completes, some text appears in the text view, as shown in Figure 1-2.

Figure 1-2. *The SlowWorker application hides its interface behind a single button. Click the button, and the interface hangs for about ten seconds while the application does its work.*

Start by using the Single View Application template to make a new application in Xcode, as you've done many times before. Name this one SlowWorker, set Devices to Universal, click Next to save your project, and so on. Next, make the changes to ViewController.swift, as shown in Listing 1-1.

Listing 1-1. Add These Methods to the ViewController.swift File

```
@IBOutlet var startButton: UIButton!
@IBOutlet var resultsTextView: UITextView!

func fetchSomethingFromServer() -> String {
```

```
    Thread.sleep(forTimeInterval: 1)
    return "Hi there"
}

func processData(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 2)
    return data.uppercased()
}

func calculateFirstResult(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 3)
    return "Number of chars: \(data.characters.count)"
}

func calculateSecondResult(_ data: String) -> String {
    Thread.sleep(forTimeInterval: 4)
    return data.replacingOccurrences(of: "E", with: "e")
}

@IBAction func doWork(_ sender: AnyObject) {
        let startTime = NSDate()
        self.resultsTextView.text = ""
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: [\(firstResult)]\nSecond: [\(secondResult)]"
        self.resultsTextView.text = resultsSummary
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSince(startTime as
        Date)) seconds")
}
```

As you can see, the work of this class (such as it is) is split up into a number of small pieces. This code simulates some slow activities, and none of those methods really do anything time consuming at all. To make things interesting, each method contains a call to the `sleep(forTimeInterval:)` class method in `Thread`, which simply makes the program

(specifically, the thread from which the method is called) effectively pause and do nothing at all for the given number of seconds. The `doWork()` method also contains code at the beginning and end to calculate the amount of time it took for all the work to be done.

Now open Main.storyboard and drag a Button and a Text View into the empty View window. Position the controls as shown in Figure 1-3. You'll see some default text. Clear the text in the Text View and change the button's title to Start Working. To set the auto layout constraints, start by selecting the Start Working button, and then click the Align button at the bottom right of the editor area. In the pop-up, check Horizontally in Container and Click Add 1 Constraint. Next, Control-drag from the button to the top of the View window, release the mouse, and select Vertical Spacing to Top Layout Guide. To complete the constraints for this button, Control-drag from the button down to the text view, release the mouse, and select Vertical Spacing. To fix the position and size of the text view, expand the View Controller Scene in the Document Outline and Control-drag from the text view in the storyboard to the View icon in the Document Outline. Release the mouse and, when the pop-up appears, hold down the Shift key and select Leading Space to Container Margin, Trailing Space to Container Margin, and Vertical Spacing to Bottom Layout Guide, and then click return to apply the constraints. That completes the auto layout constraints for this application.

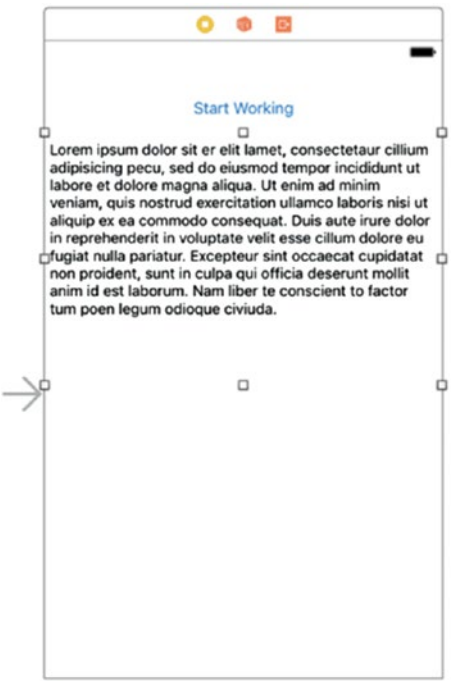

Figure 1-3. *The SlowWorker interface consists of a button and a text view*

Control-drag from the View Controller icon in the Document Outline to connect the view controller's two outlets (i.e., the `startButton` and `resultsTextView` instance variables) to the button and the text view.

Next, Control-drag from the button to the View Controller, release the mouse, and select the `doWork()` method in the pop-up so that it's called when the button is pressed. Finally, select the text view, use the Attributes Inspector to uncheck the Editable check box (it's in the upper-right corner), and delete the default text from the text view.

Save your work, and then select Run. Your app should start up, and pressing the button will make it work for about ten seconds (the sum of all those sleep amounts) before showing you the results. During your wait, you'll see that the Start Working button fades visibly, never turning back to its normal color until the "work" is done. Also, until the work is complete, the application's view is unresponsive. Tapping anywhere on the screen or rotating the device has no effect. In fact, the only way you can interact with your application during this time is by tapping the home button to switch away from it. This is exactly the state of affairs we want to avoid.

Threading Basics

Before we start implementing solutions, let's go over some concurrency basics. This is far from a complete description of threading in iOS or threading in general. I just want to explain enough for you to understand what we're doing in this chapter. Most modern operating systems (including, of course, iOS) support the notion of threads of execution. Each process can contain multiple threads, which all run concurrently. If there's just one processor core, the operating system will switch between all executing threads, much like it switches between all executing processes. If more than one core is available, the threads will be distributed among them, just as processes are.

All threads in a process share the same executable program code and the same global data. Each thread can also have some data that is exclusive to the thread. Threads can make use of a special structure called a **mutex** (short for **mutual exclusion**) or a lock, which can ensure that a particular chunk of code can't be run by multiple threads at once. This is useful for ensuring correct outcomes when multiple threads access the same data simultaneously, by locking out other threads when one thread is updating a value (in what's called a **critical section** of your code).

A common concern when dealing with threads is the idea of code being **thread-safe**. Some software libraries are written with thread concurrency in mind and have all their critical sections properly protected with mutexes. Some code libraries aren't thread-safe. For example, in Cocoa Touch, the Foundation framework is generally considered to be thread-safe. However, the UIKit framework (containing the classes specific to building GUI applications, such as `UIApplication`, `UIView`, and all its subclasses, and so on) is, for the most part, not thread-safe. (Some UIKit functionality such as drawing is considered thread-safe however.) This means that in running an iOS application, all method calls that deal with any `UIKit` objects should be executed from within the same thread, which is commonly known as the **main thread**. If you access `UIKit` objects from another thread, all bets are off. You are likely to encounter seemingly inexplicable bugs (or, even worse, you won't experience any problems, but some of your users will be affected by them after you ship your app).

Tip A lot has been written about thread safety. It's well worth your time to dig in and try to digest as much of it as you can. One great place to start is Apple's own documentation. Take a few minutes and read through this page (it will definitely help):

`https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/Multithreading/ThreadSafetySummary/ThreadSafetySummary.html`

Units of Work

The problem with the threading model described earlier is that, for the average programmer, writing error-free, multithreaded code is nearly impossible. This is not meant as a critique of our industry or of the average programmer's abilities; it's simply an observation. The complex interactions you must account for in your code when synchronizing data and actions across multiple threads are really just too much for most people to tackle. Imagine that 5% of all people have the capacity to write software at all. Only a small fraction of those 5% are really up to the task of writing heavy-duty multithreaded applications. Even people who have done it successfully will often advise others to not follow their example.

Fortunately, there are alternatives. It is possible to implement some concurrency without too much low-level detailed work. Just as we have the ability to display data on the screen without directly poking bits into video RAM and to read data from disk without interfacing directly with disk controllers, we can also leverage software abstractions that let us run our code on multiple threads without requiring us to do much directly with the threads.

The solutions Apple encourages us to use center around the idea of splitting up long-running tasks into units of work and putting those units into queues for execution. The system manages the queues for us, executing units of work on multiple threads. We don't need to start or manage the background threads directly, and we are freed from much of the bookkeeping that's usually involved in implementing multithreaded applications; the system takes care of that for us.

GCD: Low-Level Queuing

This idea of putting units of work into queues that can be executed in the background, with the system managing the threads for you, provides power and greatly simplifies many development situations where concurrency is needed. GCD made its debut on OS X (now macOS) several years ago, providing the infrastructure to do just that. A couple of years later, this technology came to the iOS platform as well. GCD puts some great concepts — units of work, painless background processing, and automatic thread management — into a C interface that can be used not only with Objective-C, but also with C, C++, and, of course, Swift. To top things off, Apple has made its implementation of GCD open source, so it can be ported to other Unix-like operating systems, as well.

One of the key concepts of GCD is the **queue**. The system provides a number of predefined queues, including a queue that's guaranteed to always do its work on the main thread. It's perfect for the non-thread-safe UIKit. You can also create your own queues — as many as you like. GCD queues are strictly first-in, first-out (FIFO). Units of work added to a GCD queue will always be started in the order they were placed in the queue. That said, they may not always finish in the same order, since a GCD queue will automatically distribute its work among multiple threads, if possible.

GCD accesses a pool of threads that are reused throughout the lifetime of the application. It tries to maintain a number of threads appropriate for the machine's architecture. It automatically takes advantage of a more powerful machine by utilizing more processor cores when it has work to do. Until a few years ago, iOS devices were all single-core, so this wasn't much of an issue. But now that all iOS devices released in the past few years feature multicore processors, GCD has become truly useful.

GCD uses closures to encapsulate the code to be added to a queue. Closures are first-class language citizens in Swift — you can assign a closure to a variable, pass one to a method, or return one as the result of a method call. Closures are the equivalent of Objective-C's blocks and similar features, sometimes referred to using the term **lambdas**, in other programming languages, such as Python. Much like a method or function, a closure can take one or more parameters and specify a return value, although closures used with GCD can neither accept arguments nor return a value. To declare a closure variable, you simply assign to it some code wrapped in curly braces, optionally with arguments:

```
// Declare a closure variable "loggerClosure" with no parameters
// and no return value.
let loggerClosure = {
    print("I'm just glad they didn't call it a lambda")
}
```

You can execute the closure in the same way as you call a function:

```
// Execute the closure, producing some output in the console.
loggerClosure()
```

Improving SlowWorker

To see how to use closures with GCD, let's revisit SlowWorker's `doWork()` method. It currently looks like this:

```
    @IBAction func doWork(_ sender: AnyObject) {
            let startTime = NSDate()
            self.resultsTextView.text = ""
            let fetchedData = self.fetchSomethingFromServer()
            let processedData = self.processData(fetchedData)
            let firstResult = self.calculateFirstResult(processedData)
            let secondResult = self.calculateSecondResult(processedData)
            let resultsSummary =
                "First: [\(firstResult)]\nSecond: [\(secondResult)]"
            self.resultsTextView.text = resultsSummary
```

```
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSince(startTime as
        Date)) seconds")
}
```

We can make this method run entirely in the background by wrapping all the code in a closure and passing it to a GCD function called `DispatchQueue`. This function takes two parameters: a GCD queue and the closure to assign to the queue. Make the changes in Listing 1-2 to your copy of `doWork()`.

Listing 1-2. Modifications to the doWork Method to Use GCD

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    let queue = DispatchQueue.global(qos: .default)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: [\(firstResult)]\nSecond: [\(secondResult)]"
        self.resultsTextView.text = resultsSummary
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSince(startTime as
        Date)) seconds")
    }
}
```

The first changed line grabs a preexisting global queue that's always available, using the `DispatchQueue.global()` function. That function takes one arguments letting you specify a priority. If you specify a different priority in the argument you will actually get a different global queue, which the system will prioritize differently. For now, we'll stick with the default global queue.

The queue is then passed to the `queue.async()` function, along with the closure. GCD takes the closure and puts it on the queue, from where it will be scheduled to run on a background thread and executed one step at a time, just as when it was running in the main thread.

Note that we defined a variable called `startTime` just before the closure is created, and then use its value at the end of the closure. Intuitively, this doesn't seem to make sense because, by the time the closure is executed, the `doWork()` method has returned, so the `NSDate` instance that the `startTime` variable is pointing to should already be released! This is a crucial point to understand about closures: if a closure accesses any variables from "the outside" during its execution, then some special setup happens when the closure is created, allowing it to continue to access to them. All of this is done automatically by the Swift compiler and runtime — you don't need to do anything special to make it happen.

Don't Forget That Main Thread

Getting back to the project at hand, there's one problem here: UIKit thread safety. Remember that messaging any GUI object from a background thread, including our `resultsTextView`, is a no-no. In fact, if you run the example now, you'll see an exception appear in the Xcode console after about ten seconds, when the closure tries to update the text view. Fortunately, GCD provides a way to deal with this, too. Inside the closure, we can call another dispatching function, passing work back to the main thread. Make one additional change to your version of `doWork()`, as shown in Listing 1-3.

Listing 1-3. The modified doWork Method

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    let queue = DispatchQueue.global(attributes: DispatchQueue.
    GlobalAttributes.qosDefault)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: [\(firstResult)]\nSecond: [\(secondResult)]"
```

```
            DispatchQueue.main.async {
                self.resultsTextView.text = resultsSummary
            }
            let endTime = NSDate()
            print("Completed in \(endTime.timeIntervalSince(startTime as
            Date)) seconds")
        }
    }
```

Giving Some Feedback

If you build and run your app at this point, you'll see that it now seems to work a bit more smoothly, at least in some sense. The button no longer gets stuck in a highlighted position after you touch it, which perhaps leads you to tap again, and again, and so on. If you look in the Xcode console log, you'll see the result of each of those taps, but only the results of the last tap will be shown in the text view. What we really want to do is enhance the GUI so that, after the user presses the button, the display is immediately updated in a way that indicates that an action is underway. We also want the button to be disabled while the work is in progress so that the user can't keep clicking it to spawn more and more work into background threads. We'll do this by adding a `UIActivityIndicatorView` to our display. This class provides the sort of spinner seen in many applications and web sites. Start by adding an outlet for it at the top of ViewController.swift:

```
@IBOutlet var spinner : UIActivityIndicatorView!
```

Next, open Main.Storyboard; locate an Activity Indicator View in the library; and drag it into our view, next to the button. You'll need to add layout constraints to fix the activity indicator's position relative to the button. One way to do this is to Control-drag from the button to the activity indicator and select Horizontal Spacing from the pop-up menu to fix the horizontal separation between them, and then Control-drag again and select Center Vertically to make sure that their centers remain vertically aligned.

With the activity indicator spinner selected, use the Attributes Inspector to check the Hides When Stopped check box so that our spinner will appear only when we tell it to start spinning (no one wants an unspinning spinner in their GUI). Next, Control-drag from the View Controller icon to the spinner and connect the spinner outlet. Save your changes.

Now open ViewController.swift. Here, we'll first work on the `doWork()` method a bit, adding a few lines to manage the appearance of the button and the spinner when the user taps the button and when the work is done. We'll first set the button's `enabled` property to `false`, which prevents it from registering any taps and also shows that the button is disabled by making its text gray and somewhat transparent. Next, we get the spinner moving by calling its **startAnimating**() method. At the end of the closure, we re-enable the button and stop the spinner, which causes it to disappear again, as shown in Listing 1-4.

Listing 1-4. Adding the Spinner Functions to our doWork Method

```
@IBAction func doWork(sender: AnyObject) {
    let startTime = NSDate()
    resultsTextView.text = ""
    startButton.isEnabled = false
    spinner.startAnimating()
    let queue = DispatchQueue.global(qos: .default)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        let firstResult = self.calculateFirstResult(processedData)
        let secondResult = self.calculateSecondResult(processedData)
        let resultsSummary =
            "First: [\(firstResult)]\nSecond: [\(secondResult)]"
        DispatchQueue.main.async {
            self.resultsTextView.text = resultsSummary
            self.startButton.isEnabled = true
            self.spinner.stopAnimating()
        }
        let endTime = NSDate()
        print("Completed in \(endTime.timeIntervalSince(startTime as
        Date)) seconds")
    }
}
```

Build and run the app, and press the button. Even though the work being done takes a few seconds, the user isn't just left hanging. The button is disabled and looks the part as well. Also, the animated spinner lets the user know that the app hasn't actually hung up and can be expected to return to normal at some point.

Concurrent Closures

The sharp-eyed among you will notice that, after going through these motions, we still haven't really changed the basic sequential layout of our algorithm (if you can even call this simple list of steps an algorithm). All that we're doing is moving a chunk of this method to a background thread and then finishing up in the main thread. The Xcode console output proves it: this work takes ten seconds to run, just as it did at the outset. The issue is that the `calculateFirstResult()` and `calculateSecondResult()` methods don't depend on each and therefore don't need to be called in sequence. Doing them concurrently gives us a substantial speedup.

Fortunately, GCD has a way to accomplish this by using what's called a **dispatch group**. All closures that are dispatched asynchronously within the context of a group, via the `dispatch_group_async()` function, are set loose to execute as fast as they can, including being distributed to multiple threads for concurrent execution, if possible. We can also use `dispatch_group_notify()` to specify an additional closure that will be executed when all the closures in the group have been run to completion.

Make these final changes to the `doWork` method, as shown in Listing 1-5.

Listing 1-5. The Final Version of Our doWork Method

```
@IBAction func doWork(_ sender: AnyObject) {
    let startTime = Date()
    self.resultsTextView.text = ""
    startButton.isEnabled = false
    spinner.startAnimating()
    let queue = DispatchQueue.global(qos: .default)
    queue.async {
        let fetchedData = self.fetchSomethingFromServer()
        let processedData = self.processData(fetchedData)
        var firstResult: String!
        var secondResult: String!
        let group = DispatchGroup()
```

```
            queue.async(group: group) {
                firstResult = self.calculateFirstResult(processedData)
            }
            queue.async(group: group) {
                secondResult = self.calculateSecondResult(processedData)
            }

            group.notify(queue: queue) {
                let resultsSummary = "First: [\(firstResult!)]\nSecond:
                [\(secondResult!)]"
                DispatchQueue.main.async {
                    self.resultsTextView.text = resultsSummary
                    self.startButton.isEnabled = true
                    self.spinner.stopAnimating()
                }
                let endTime = Date()
                print("Completed in \(endTime.timeIntervalSince(startTime))
                seconds")
            }
        }
    }
```

One complication here is that each of the `calculate` methods returns a value that we want to grab, so we need to make sure that the variables `firstResult` and `secondResult` can be assigned from the closures. To do this, we declare them using `var` instead of `let`. However, Swift requires a variable that's referenced from a closure to be initialized, so the following declarations don't work:

```
var firstResult: String
var secondResult: String
```

You can, of course, work around this problem by initializing both variables with an arbitrary value, but it's easier to make them implicitly unwrapped optionals by adding `!` to the declaration:

```
var firstResult: String!
var secondResult: String!
```

Now, Swift doesn't require an initialization, but we need to be sure that both variables will have a value when they are eventually read. In this case, we can be sure of that, because the variables are read in the completion closure for the async group, by which time they are certain to have been assigned a value. With this in place, build and run the app again. You'll see that your efforts have paid off. What was once a ten-second operation now takes just seven seconds, thanks to the fact that we're running both of the calculations simultaneously.

Obviously, our contrived example gets the maximum effect because these two "calculations" don't actually do anything but cause the thread they're running on to sleep. In a real application, the speedup would depend on what sort of work is being done and what resources are available. The performance of CPU-intensive calculations is helped by this technique only if multiple CPU cores are available. It will get better almost for free as more cores are added to future iOS devices. Other uses, such as fetching data from multiple network connections at once, would see a speed increase even with just one CPU.

As you can see, GCD is not a panacea. Using GCD won't automatically speed up every application. But by carefully applying these techniques at those points in your app where speed is essential, or where you find that your application feels like it's lagging in its responses to the user, you can easily provide a better user experience, even in situations where you can't improve the real performance.

Background Processing

Another important technology for handling concurrency is background processing. This allows your apps to run in the background — in some circumstances, even after the user has pressed the home button.

This functionality should not be confused with the true multitasking that modern desktop operating systems now feature, where all the programs you launch remain resident in the system RAM until you explicitly quit them (or until the operating system needs to free up some space and starts swapping them to disk). iOS devices still have too little RAM to be able to pull that off very well. Instead, this background processing is meant to allow applications that require specific kinds of system functionality to continue to run in a constrained manner when they are in the background. For instance, if you have an app that plays an audio stream from an Internet radio station, iOS will let that app continue to run, even if the user switches to another app. Beyond that, it