Modern Concurrency on Apple Platforms

Using async/await with Swift

Andrés Ibañez Kautsch

Apress®

Modern Concurrency on Apple Platforms

Using async/await with Swift

Andrés Ibañez Kautsch

Apress®

Modern Concurrency on Apple Platforms: Using async/await with Swift

Andrés Ibañez Kautsch
La Paz, Bolivia

ISBN-13 (pbk): 978-1-4842-8694-4 ISBN-13 (electronic): 978-1-4842-8695-1
https://doi.org/10.1007/978-1-4842-8695-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Mark Powers
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. 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 booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

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 the GitHub repository: https://github.com/Apress/Modern-Concurrency-on-Apple-Platforms. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my mother Renata and to my brother Gastón, for all the support and patience they have always shown and for everything they have always done for me.

Table of Contents

About the Author

Andrés Ibañez Kautsch started writing iOS apps as a young college student in 2011. His first introduction to concurrency programming and its common pitfalls was in an operating systems class that introduced the importance (and complexity) of writing concurrent code. Since then, he has studied how this problem is solved in Apple's platforms, including iOS. Andy has worked in institutions that make use of concurrent technologies to keep their services running for their customers, including banks, applying the concepts to their mobile applications.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years.

He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

Acknowledgments

Whenever you decide to buy a technical book, you usually see one or two author names, and maybe a reviewer attached to it, but there are a lot of people involved in a single book, both directly and indirectly. It will not be possible to acknowledge everyone who has helped make this book possible, but I'd like to single out a few.

First, I'd like to give a big kudos to Alexis Marechal. Alexis has, in short, saved my career as a software developer. Alexis is not only a great teacher, but also a great friend. His teaching style has greatly inspired mine, and I hope I can do justice to that in this book. A big thank-you to Carlos Anibarro and Jose Luis Vera for their teachings and words of encouragement as well.

Writing a book is something I have always wanted to do, but Ernesto Campohermoso is the first person who ever asked me, "When are you going to write a book?" That single question was a great push for this book to come into existence.

Ivan Galarza is the first person I mentored in iOS development, and he became a very reliable iOS engineer in a short time thanks to all the talent he has. He was the ideal candidate to review a book in progress. His input helped me make sure everything makes sense and that this book will be something people would like to read.

A big thank-you to all the folks at Apress for the process that made writing this book possible. The entire experience has been a joy, from being contacted to write a book to turning in the final chapters. The final form of this book would have not been possible without their tenacity and amazing work ethic.

Finally, I'd like to give a big thanks to all the readers of my blog, andyibanez.com. The comments and feedback I received in the original tutorial series for this topic helped me greatly improve this book. I cannot reply to every message I get, but I am thankful to all my readers. Thank you all.

Preface

Concurrency is a topic a lot of developers find scary. And for good reason. Concurrency is probably one of the most complicated topics in the world of programming. When you look at it from a very high level, concurrency allows us to do a lot of work at the same time. Sometimes related, sometimes unrelated. The definition is simple, but as a programmer, it is hard to reason about concurrent programs. We learn to write code that can be read from top to bottom, and code usually makes sense when you read it in such a manner. In fact, the way developers reason about code is not too different to how non-programmers reason about a cooking recipe. Humans can understand anything they read if it is structured and makes sense.

But the very nature of concurrency breaks this top-to-bottom reading flow. Not only do we need to reason about concurrency differently, but we also need to keep in mind other factors that make it harder, such as shared mutable data, locks, threads.... Concurrency is naturally very complicated, especially when dealing with lower-level tools.

Luckily, the new async/await system, introduced in 2021, makes concurrency easier to reason about because it abstracts a lot of the complexity behind language-integrated features that are easy to use and hard to misuse. If you have written concurrent code before with anything other than async/await in any other platform (and that includes but is not limited to iOS, macOS, and other Apple platforms), you do not have to concern yourself with mutexes, semaphores, or any other low-level concurrency primitives. If you have never written concurrent code before, you can write amazing multithreaded software in a way that makes sense without ever having to concern yourself with the complexities of traditional concurrency. This new system is very powerful and easy to use for both developers familiar with concurrency as well as those who have never written concurrent code before.

Who This Book Is For

This book is aimed at intermediate iOS developers and above. You should find this book to be of your skill level if you have been writing Apple apps a bit over a year. Having experience with other concurrency tools in either Apple platforms or anything else may help you grasp this book easier, but previous concurrency knowledge is by no means necessary. You should be familiar with the basic process of writing and maintaining an iOS app to take advantage of this book.

How This Book Is Organized

I tried my best to organize this book in a way that makes sense. There were topics that recursively required the knowledge of other topics before they could be properly understood. For those situations, I spent a little bit more time explaining some concepts at a higher level so you could get by before they got properly introduced.

Chapter 1 introduces concurrency and its traditional problems when trying to implement it. It discusses low-level concurrency primitives and how they can be used. It also discusses the traditional problems you will find when you try to implement a concurrency system without using higher-level tools such as async/await.

Chapter 2 formally introduces "async" and "await" as keywords of the Swift language. These two keywords are essential to understand to use the concurrency system effectively. Every single topic makes use of async/await, so this chapter is completely dedicated to these keywords.

Chapter 3 introduces Continuations, a tool that helps you migrate closure-based or even delegate-based code to use async/await. This can help you "bridge" such code into the async/await world, making them easier to write and understand. You will also learn how to backport concurrent code to iOS 14 and 13.

Chapter 4 introduces the concept of Structured Concurrency. You will write your first concurrent code here. Structured concurrency helps you write multithreaded code that is easy to read and write.

Chapter 5 introduces the concept of Unstructured Concurrency, a topic that will help you write streamlined concurrent code with a little bit more of flexibility than Structured Concurrency.

Chapter 6 introduces the concept of Actors. Actors are reference types that isolate their own state, so they are useful when you need to write concurrent code that deals with shared mutable state. It helps you answer questions such as "What happens if two processes write to this variable at the same time?"

Chapter 7 is all about Sendable types, which are objects that can be used safely in concurrent code, either because they have built-in protection (such as actors) or because the developers took special care of these types to make them usable concurrently (like classes that implement their own synchronization mechanism).

Chapter 8 discusses Global Actors, a tool to help you write concurrent code that is spread out across different files and even frameworks. It also discusses the Main Actor, a global actor that you use when you need to update your app's UI.

Chapter 9 is all about async sequences. These sequences can help you receive values over time in an asynchronous context, helping you eliminate the usage of closures and delegates in some scenarios.

Chapter 10, the final chapter, covers the usage of a property wrapper called @TaskLocal, which you can use to share data down a concurrent tasks tree.

Before You Get Started

While Apple managed to backport the new concurrency system to iOS 13 and iOS 14, it is recommended you study this system with iOS 15. There are no native APIs that use async/await in lower iOS versions, and you would need to provide an alternative to them every time you are interested in using them.

It is recommended you have at least Xcode 13, but you should have the latest version if possible. At the time of this writing, the latest Xcode version is 13.4.1. The exercises and sample code were tested on this Xcode version. This implies your Mac will need to run macOS Monterey as Xcode 13 cannot run on anything lower than Monterey.

CHAPTER 1

Introduction

Programmers are used to writing programs that are executed in a linear fashion. As you write, test, and execute your program, you expect your instructions to run in the order that you wrote them. In Listing 1-1, you have a program that will first assign a variable `a` to the number `2`. It will then assign a variable `b` to the number `3`, followed by assigning a variable `sum`, the sum of `a + b`, and it will finally print a result to the console. There is no way this program will work if you try to *print(sum)* before you even managed to calculate the value for `sum`.

Listing 1-1. A simple program that runs from top to bottom

```
let a = 2
let b = 3
let sum = a + b // This variable depends on the values for
a and b, but the variables themselves can be assigned in
any order.
print(sum) // We can only print a variable that we have the
value of.
```

This is called *procedural programming*, because you write simple statements, and they are executed from top to bottom. Even if you add statements that can alternate the execution flow, it's still easy to follow. If you a call function when working with procedural programming, your program will "jump" to a different place in memory and execute its

A. I. Kautsch, *Modern Concurrency on Apple Platforms*,
https://doi.org/10.1007/978-1-4842-8695-1_1

contents, but the execution of these lines will also be done procedurally in the same order they were written until control flow is returned to the caller.

Even people who are not programmers can follow any instruction set if they are written in a specific order and *if they are doing one thing at a time*. Someone following a cooking recipe, or someone building a doghouse from an online tutorial may not be a programmer, but people are naturally good at doing something if they have the steps clearly laid down.

But computer programs grow and become more complex. While it is true that a lot of complex software can be written that follows such a linear execution flow, often programs will need to start doing more than one thing at once; rather than having a clear code execution path that you can follow with your bare eyes, your program may need to execute in such a way that it's not obvious to tell what's going on at a simple glance of the source code. Such programs are multithreaded, and they can run multiple (and often – but not always – unrelated) code paths at the same time.

In this book, we will learn how to use Apple's `async/await` implementation for asynchronous and multithreaded programming. In this chapter, we will also talk about older technologies Apple provides for this purpose, and how the new `async/await` system is better and helps you to not concern yourself with traditional concurrency pitfalls.

Important Concepts to Know

Concurrency and asynchronous programming are very wide topics. While I'd love to cover everything, it would go out of the scope of this book. Instead, we will define four important concepts that will be relevant while we explore Apple's `async/await` system, introduced in 2021. We will define them with as few words as possible, because it's important that you keep them in mind while you work through the chapters of this book. The concepts of the new system itself will be covered in the upcoming chapters.

Note Apple is not the original creator of the `async/await` system. The technology has been used in other platforms in the past. Microsoft announced C# would get `async/await` support in 2011, and C# with these features was officially released to the public in 2012.

Threads

The concept of *Thread* can vary even when talked about in the same context (in this case, concurrency, and asynchronous programming). In this book, we will treat a *thread* as a unit of work that can run independently. In iOS, the *Main Thread* runs the UI of your app, so every UI-related task (updating the view hierarchy, removing and adding views) must take place in the main thread. Attempting to update the UI outside of the main thread can result in unwanted behavior or, even worse, crashes.

In low-level multithreading frameworks, multithreading developers will manually create threads and they need to manually synchronize them, stop them, and do other thread management operations on their own. Manually handling threads is one of the hardest parts of dealing with multithreading in software.

Concurrency and Asynchronous Programming

Concurrency is the ability of a thread (or your program) to deal with multiple things at once. It may be responding to different events, such as network handlers, UI event handlers, OS interruptions, and more. There may be multiple threads and all of them can be concurrent.

There are different APIs throughout Apple's SDKs that make use of concurrency. Listing 1-2 shows how to request permission to use Touch ID or Face ID, depending on the device.

Listing 1-2. Biometric unlock is an asynchronous task

```
func requestBiometricUnlock() {
    let context = LAContext()

    var error: NSError? = nil

    let canEvaluate = context.canEvaluatePolicy(
    .deviceOwnerAuthenticationWithBiometrics, error: &error)

    if canEvaluate {
        if context.biometryType != .none {
            // (1)
            context.evaluatePolicy(
                .deviceOwnerAuthenticationWithBiometrics,
                localizedReason: "To access your data") {
                (success, error) in
                // (2)
                if success {
                    // ...
                }
            }
        }
    }
}
```

`(1)` calls `context.evaluatePolicy`, which is a *concurrent call*. This will ask the system to suspend your app so it can take over. The system will request permission to use biometrics while your app is suspended. The thread your app was running on may be doing something entirely different and not even related to your app while the system is running `context.evaluatePolicy`. When the user responds to the prompt, either accepting or rejecting the biometric request, it will deliver the result to your app. The system will wait for an appropriate time to notify your app with the user's selection. The selection will be delivered to your app in the completion handler (also called a *callback*) on `(2)`, at which point your app will be in control of the thread again. The selection may be delivered in a different thread than the one which launched the `context.evaluatePolicy` call – this is important to know, because if the response updates the UI, you need to do that work on the main thread. This is also called a *blocking mechanism* or *interruption*, as `evaluatePolicy` is a *blocking* call for the thread. If you have done iOS for at least a few months now, you are familiar with this way of dealing with various events. `URLSession`, image pickers, and more APIs make use of this mechanism.

People often think that asynchronous programming is the act of running multiple tasks at once. This is a different concept called *Multithreading*, and we will talk about it in the next point.

Note If you are thinking on implementing biometric unlock to resources within your app, please don't use the code above. It has been simplified to explain how concurrency works, and it doesn't have the right safety measures to protect your user's data.

Multithreading is the act of running multiple tasks at once. Multiple threads (hence its name – multithreading) are usually involved. Many tasks can be running at the same time in the context of your app. Downloading multiple images from the internet at the same time or downloading a file from your web browser while you open some tabs are some examples of multithreading. This allow us to run tasks in *parallel* and is sometimes called *parallelism*.