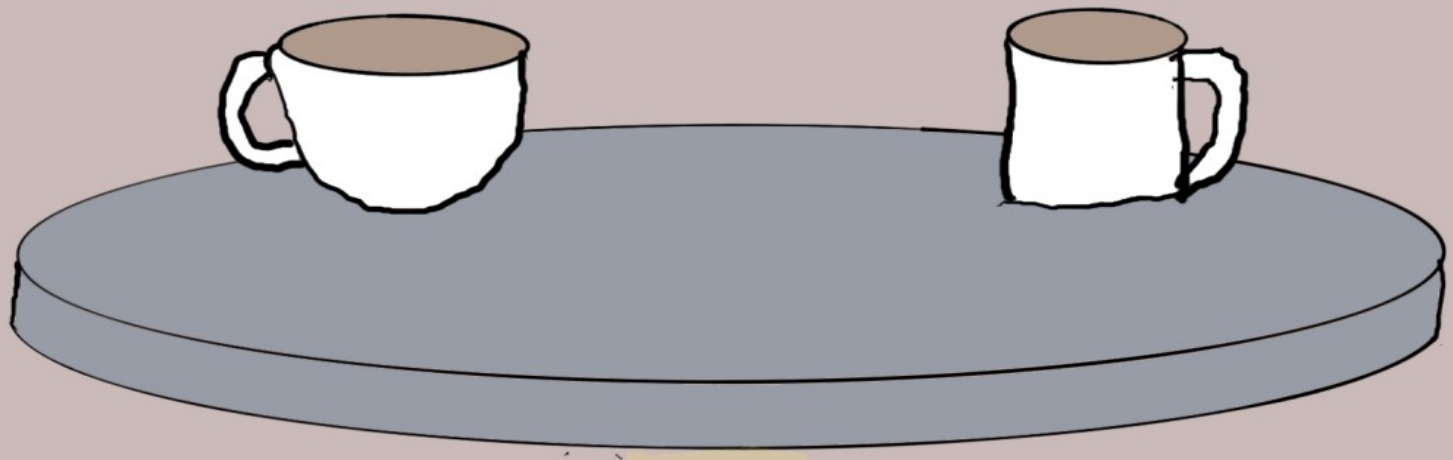
# The Curious Case of the Async Cafe

An Introduction to
Modern Concurrency in Swift



by Daniel H Steinberg

# The Curious Case Of The Async Cafe

## An Introduction To

## Modern Concurrency In Swift

by Daniel H Steinberg

Editors Cut

## Copyright



ISBN-13: 978-1-944994-04-4

## Book Version

This is version 0.5 for Swift 5.7, Xcode 14.2, macOS Ventura 13.0, and iOS 16.1 released February 2023. All code has been tested on Apple Silicon.

## Code Download

Visit https://github.com/editorscut/ec013Async for all of the code for this book.

Run it in Xcode 14.2 or higher. All code is written in Swift.

To avoid long lines and code that wraps, I've split some lines in code listings in ways that you might not in an IDE. Please feel free to not break the lines where I have.

## Recommended Settings

The ePub is best viewed in scrolling mode on an iPad. On smaller devices I also choose landscape. For some reason that I don't understand, scrolling mode is supported by Apple's Books app on

the iPad but not on the Mac. If you view this book in Apple's Books app, choose "Let lines break naturally" in Preferences > General. Finally, I've gone to great pains to make this look good in light and dark mode but Apple has foiled me yet again. I'm told that not all of the syntax coloring works in dark mode.

## Submit Errata

Submit your errata here for the book or for the source code by selecting New Issue. Please provide the book version listed above, chapter, section, and page number in your issue so that I can find it and, if possible, resolve it quickly.

## Official Links

Please check http://developer.apple.com for additional resources including videos, sample code, documentation, and forums. You'll also find information on what is required to take advantage of these resources.

Apple has posted videos, slides, and sample code from the Worldwide Developers Conference.

## Legal

The sample code is intended to be used to illustrate points made in the text. It is not intended to be used in production code.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks or service marks. Where those designations appear in this book, and Dim Sum Thinking, Inc. was aware of the trademark claim, the designations have been printed with initial capital letters or in all capitals.

This book uses terms that are registered trademarks of Apple Inc. for which the terms of use don't permit rendering them in all caps or initial caps. You can view a complete list of the trademarks and registered trademarks of Apple Inc at http://www.apple.com/legal/trademark/appletmlist.html.

The Editor's Cut name and logo are registered trademarks of Dim Sum Thinking, Inc.

# Table Of Contents

CHAPTER 1

# Async, Await, And Task

Sections:



We make all sorts of method calls when writing apps for the iPhone, iPad, Apple Watch, Mac, and other platforms that use Swift.

Some are simple and quick and return immediately and some will take a while to finish.

Over the years there've been lots of techniques for working with these asynchronous calls including delegates, closures, notifications, and more recently the Combine framework. We've used threads, queues, and other APIs. The async / await mechanism is built around Tasks and functions that can suspend and resume.

In this chapter we look at three fundamental ideas.

- If a method might take a while we label it as `async`. A method marked as `async` can suspend and resume so that the app isn't blocked by long running work.
- If we call a method that is `async` we must mark the call with `await`. Every time you see `await` you should consider it to be a possible suspension point and think about what might be happening while the task is suspended.
- Finally, we can only use `await` from an async context. That means that either the call is inside an asynchronous method that is also marked `async` or the call is in the closure passed to a `Task`.

As simple as the syntax makes it seem, there are a lot of things to watch out for when using async / await. We'll start to dig into the complexities in this chapter.

Before we start on the ideas behind async / await, we'll look back at the syntax for error handling in Swift. You'll see parallels between `throws`, `try`, and `do catch` and `async`, `await`, and `Task`.

Let's start by walking through the example we'll work with throughout this chapter.

# Introducing Async / Await

*We had yet to place our order when I suddenly noticed that Edges had turned towards me and was studying my face.*

*"You look so concerned," the great detective said, "is there something wrong?"*

*"Why yes," I replied, "it's just that I can't conceive of living in a world - even in this cafe - where everything comes to an absolute standstill every time any drink is ordered."*

*"Of course you are right, my friend," Edges smiled. "I have a better world. In this other world when the waiter orders a drink from a barista nothing stops."*

*"Nothing?"*

*"Not a thing. Instead the waiter returns immediately to our table to take the next order and we continue as usual."*

*"Then how do we get our coffee?"*

*Edges nodded, "ah, yes. When the barista finishes the drink the waiter is given the drink and proceeds to bring it to us."*

*I thought a minute and asked, "but isn't the barista the bottleneck?"*

*"It is a good point that you raise," Edges agreed. "For now assume that every drink order is given to a different barista with their own machine. We'll return to this later."*

Continue with our current project or start with the project in *Chapter01/03/*.

Change `EntryController` to use `AsyncEntryVendor` instead of `SleepingEntryVendor`.

*WhosNext/Controllers/**EntryController.swift***

```
class EntryController: ObservableObject {
  private var count = 0
  @Published private(set) var entries: [Entry] = []
  private let vendor = AsyncEntryVendor()
}
```

## async

`async` is the parallel in the concurrency world to `throws` in the error world.

It's our "Beware, this could take a while" sign that we hang so that people using our method know that we should code as if this method call might take up a significant amount of time.

The `async` keyword indicates that a method is able to suspend. The idea is the system can do other work and come back to this method later.

A familiar example is a network request. You don't want your user to have to sit while the request is sent to a server and we wait for a response of some sort. Suspending at the point that the network call is made allows the system to not be blocked while waiting for a response that could take a while. In fact, a network request is both an asynchronous call and one that can fail. We'll come back to this example later.

So the `async` label on a method lets us know that something in its body may take a while and whoever is in charge of scheduling might want to pause the execution and resume later on the current thread or on another thread.

Even though we aren't doing anything in `imageName()` that could suspend, label the `imageName()` method in `AsyncEntryVendor` with `async` in the same place that we used `throws` earlier.

*WhosNext/Controllers/EntryVendors/**AsyncEntryVendor.swift***

```
struct AsyncEntryVendor  {
  func entry(for count: Int) -> Entry {
    let imageName = imageName(for: count) // problem
    return Entry(imageName: imageName)
  }
}

extension AsyncEntryVendor {
  private func imageName(for int: Int) async -> String {
    let number = int % 51
    return "\(number).circle"
  }
}
```

You should see a compiler error at the call site for `imageName()` in the `entry()` method.

We need to be clear in our code that we are aware that the call to `imageName()` may suspend.

## await

With errors, if we call a method that `throws` we have to mark the call with `try`. That indicates that we know that the call can result in an error being thrown.

Similarly, if we call a method that is `async` we have to mark the call with `await`. That indicates that we know that calling the async function can result in the execution being suspended.

As with `try`, note that when we mark a method call with `await` we are communicating that execution could be suspended -- it's not definite that it will be.

In any case, add `await` before the call to `imageName()`.

*WhosNext/Controllers/EntryVendors/**AsyncEntryVendor.swift***

```
struct AsyncEntryVendor  {
  func entry(for count: Int) -> Entry {
    let imageName = await imageName(for: count)  //problem
    return Entry(imageName: imageName)
  }
}
```

The compiler error is now that we're using `await` but we aren't in a function that supports concurrency.

Again, let's refer back to our use of `try` to call a method that `throws`. At the point that we added the `try` we were told we either had to handle the error or rethrow it.

This is what we're being told here as well but this time in the context of concurrency because our use of `await`.

Either we have to wrap the asynchronous call somehow to shield it from the outside world, or we have to mark `entry()` as asynchronous as well.

In the current case, we can't wrap the asynchronous call as we need to return an `Entry` instance and that's going to take time.

So mark `entry()` as `async`.

*WhosNext/Controllers/EntryVendors/**AsyncEntryVendor.swift***

```
struct AsyncEntryVendor  {
  func entry(for count: Int) async -> Entry {
    let imageName = await imageName(for: count)
    return Entry(imageName: imageName)
  }
}
```

Well, that's a relief. There are no more errors in `AsyncEntryVendor`. The error has moved to the code that calls `entry()` in `EntryController`. Hopefully, that code belongs to someone else and we can ignore the error.

I'm kidding. Of course we're going to handle the asynchronous call there.

## Task

You should expect a compiler error at the call site for `AsyncEntryVendor`'s `entry()`. In fact, there are two but only one might appear.

*WhosNext/Controllers/**EntryController.swift***

```
extension EntryController {
  func next() {
    count += 1
    let entriesCopy = entries
    let newEntry = vendor.entry(for: count) // problem
    entries = entriesCopy + [newEntry]
  }
}
```

Since `entry()` is an `async` method, we add `await` before the `vendor.entry()` call.

This is going to become fairly automatic for you. We use `await` when we call an `async` method.

*WhosNext/Controllers/**EntryController.swift***

```
extension EntryController {
  func next() {
    count += 1
    let entriesCopy = entries
    let newEntry = await vendor.entry(for: count) // problem
    entries = entriesCopy + [newEntry]
  }
}
```

The remaining compiler error is:

```
'async' call in a function that does not support concurrency
```

This is the same choice we faced in `entry()`. To repeat the choice, either we have to wrap the asynchronous call somehow to shield it from the outside world, or we have to mark `next()` as asynchronous as well.

In our current case we can't mark `next()` as `async` as it is the action for a button and cannot be an asynchronous call.

So, we wrap the `await` in a `Task`. Syntactically, this looks like wrapping a `try` in a `do` block.

*WhosNext/Controllers/**EntryController.swift***

```
extension EntryController {
  func next() {
    count += 1
    let entriesCopy = entries
    Task {
      let newEntry = await vendor.entry(for: count)
      entries = entriesCopy + [newEntry]
    }
  }
}
```

The `next()` method is synchronous and returns almost immediately. Inside `next()` we increment `count`, make a copy of `entries`, Create a `Task`, and return. The work of the `Task` might take a long time but `next()` has executed and returned and will be long gone.

A `Task` is how we keep asynchronous work from blocking. Other work can happen on a thread while our `Task` is suspended. When the `Task` resumes, it might resume on the same thread or it could resume on a different thread.

There are many ways to create a `Task`. The simplest signature of an `init()` for a `Task` that can't result in an error is:

```
init(priority: TaskPriority?, operation: () async -> Success)
```

In our case we aren't going to specify the `TaskPriority?` so `priority` will have the default value `nil`.

`operation` automatically inherits the actor context. That currently doesn't mean anything, but by the end of the section we'll talk

briefly about running `next()` on the actor that runs on the main thread. Our `Task` inside of `next()` will inherit this actor context and also want to execute on the actor that runs on the main thread.

`operation` also implicitly captures `self` so we don't list `self` in a capture list at the start of the closure.

Most of us generally move `operation` to a trailing closure. So we use the following syntax to create a `Task`.

```
Task {
  // work to be done in the form of a closure
  // that accepts no input and optionally
  // returns a value of type Success
}
```

We'll explore `Task`s further throughout this book.

For now I do want to make one subtle point. Take a look at the closure for our `Task`.

*WhosNext/Controllers/**EntryController.swift***

```
Task {
  let newEntry = await vendor.entry(for: count)
  entries = entriesCopy + [newEntry]
}
```

We're covering so much new material that I want to stop and remind you that `operation` implicitly captures `self`. That's why we aren't being prompted to explicitly use `self` before `vendor`, `count`, and `entries`.

`self` is strongly retained by the closure. We can explicitly use `[weak self]` in the closure's capture list or use a variant of `Task` that we'll discuss later: a detached `Task`.

## Take a breath

Let's take a moment to take stock of what we've learned.

To sum up, if you have a method call that has `await` in front of it, you must enclose this call either in a `Task` or in a method that is marked `async`. At some point up the call stack every `async` method must be contained in a `Task`.

When you create a `Task`, its `operation` is sent to the system to execute when there's an available thread. The `TaskPriority` can help the system decide what gets done in what order.

That's a lot of information all at once. For now we'll use this simplest form of `Task` in the simplest way.

Just as `do catch` allows us to call a method that `throws` inside of a method that doesn't, `Task` allows us to call a method that is async from a method that is synchronous. Each allows us to embed an extraordinary world in our every day world.

We'll say more about `Task`s later. We'll also say a lot more about `await`. For now, every time you see `await`, think of it as a possible suspension point. Work may suspend when you call `vendor.entry()` so it might be a while before the async call to `vendor.entry` returns

and `newEntry` is assigned the return value and before the next line is executed.

On the other hand, `next()` returns immediately, letting the `Task`'s work continue in the background.

Run the app.

The good news is that it runs and is responsive. The bad news is that we're getting a lot of purple warnings:

```
Publishing changes from background threads is not allowed;
make sure to publish values from the main thread
(via operators like receive(on:)) on model updates.
```

The diagnosis is perfect. The recommendation, however, is not helpful in our situation.

## Meet the MainActor

The challenge with learning any big new concept is that there are a lot of subtopics and each depends somewhat on one or more of the other ones.

I'm only going to briefly tell you what an actor is right now and we'll dig in deeper later on.

The big idea is that if you and I hold references to the same object and we can each make changes to this shared object then we can get into trouble if we're both trying to access it at the same time.

An `actor` has reference semantics like a `class` but it restricts access to properties and methods that are explicitly part of the actor.

That means that I can't be getting a property while you're setting it. Only one of us has access to the property at a time.

There is a special actor known as the `MainActor`. All UI is done on the `MainActor` in the same way we made sure that all UI was done on the Main Queue when using Grand Central Dispatch and on the main thread before that. Actors, Queues, and Threads are all different mental models and programming constructs but no matter which one we use, all UI is done on the main thread.

All SwiftUI `View`s and UIKit `ViewController`s run on the `MainActor`.

The purple runtime issue that we're seeing is that `MainView` uses `EntryController`'s `entries` to update the UI and display the numbers surrounded by circles. Because `EntryController` is an `ObservableObject` and `entries` is `@Published`, we need to update `entries` on the `MainActor`.

We will see various techniques for doing so. One is to explicitly run the updating code on the `MainActor` using `MainActor.run{}`. A second is to mark the `next()` method as being called on the `MainActor`.

We will follow Apple's guidance and mark `ObservableObject`s, in this case the entire `EntryController`, with `@MainActor`.

*WhosNext/Controllers/**EntryController.swift***

```
import Combine

@MainActor
class EntryController: ObservableObject {
  private var count = 0
  @Published private(set) var entries: [Entry] = []
  private let vendor = AsyncEntryVendor()
}
```

This placement of `@MainActor` marks all property access and method calls as taking place on the `MainActor`.

## Point-free

You'll see a warning in `MainView` where we define the action of the button.

*WhosNext/Views/**MainView.swift***

```
extension MainView: View {
  var body: some View {
    NavigationStack {
      VStack {
        EntryGrid(entries: controller.entries)
        Button("Next",
               action: controller.next) //warning
      }
      .padding()
      .navigationTitle("Entries")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

We define the button action as being the method with name `controller.next`. This style is called point-free. Honestly, I created this issue on purpose so that we'd see this warning and be forced to address it.

If you aren't familiar with the point-free name or style then your code would have run fine as you would have more likely defined the action using a closure along with the parentheses that indicate the method call like this: `{ controller.next()}`.

I prefer the point-free version `controller.next` but this currently removes the information that `next()` runs on the `MainActor`. We need to use the pointed version `controller.next()`. Let's use a trailing closure to specify the action.

*WhosNext/Views/**MainView.swift***

```
extension MainView: View {
  var body: some View {
    NavigationStack {
      VStack {
        EntryGrid(entries: controller.entries)
        Button("Next") {
          controller.next()
        }
      }
      .padding()
      .navigationTitle("Entries")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}
```

Run the app. It runs perfectly. The button is immediately available again and the numbers appear just as they did in the first section of

this chapter.

Then again, `imageName()` doesn't take any time at all.

In the next section we re-introduce a sleep in the middle of `imageName()`. This time we'll use `Task.sleep()` instead of `Thread.sleep()` as `Task.sleep()` participates in async / await.

## The rules

Here's a quick summary of what we've seen.

- If we call a method marked `async` we must use `await` to signal that we know we are calling a method that could suspend execution and take a while.
- If we call a method using `async`, the method in which we call it must either also be `async` or we must wrap the `await` in a `Task`.

The basic mental map for now is that syntactically `async` is like `throws`, `await` is like `try`, and `Task` is like `do catch`.

There are a ton more details about the what and why but this gets us started on the how. Next let's dig into `Task` a bit.