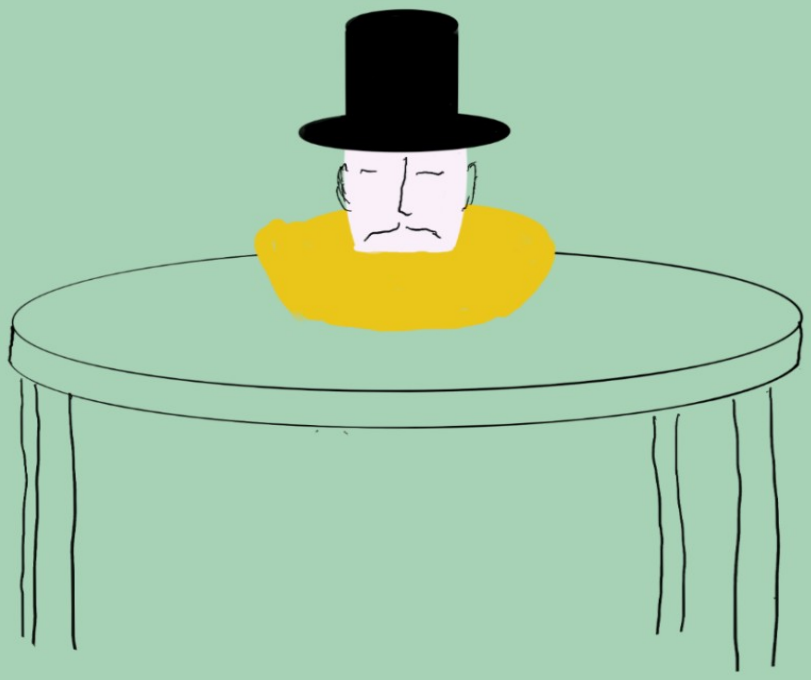
The Case of the Vanishing Bodies



An Introduction to Swift Macros

Daniel H Steinberg

The Case Of The Vanishing Bodies

An Introduction To

Swift Macros

by Daniel H Steinberg

Editors Cut

Copyright

ISBN-13: 978-1-944994-06-8

Book Version

This is the initial release of this book for Swift 5.10, Xcode 15.3, macOS Sonoma 14.4, and iOS 17.4 released April 2024. All code has been tested on Apple Silicon.

Code Download

Visit https://github.com/editorscut/ec015swiftmacros for all of the code for this book.

Run it in Xcode 15 or higher (15.3 if possible). 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

A Look Around

Swift Macros are like a compile-time magic trick that turns this bit of Swift code into that bit of Swift code.

If you've worked with macros in other languages, I want you to keep an open mind. Swift macros are not like some systems you've experienced or heard awful things about.

In this first chapter you'll see examples of macros in Swift code we use every day when writing SwiftUI and SwiftData apps and get a quick tour of the Xcode template we'll use to create our own macros.

If you've been around SwiftUI for a while you might remember writing code like this to display a preview for `ContentView`.

```
struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    ContentView()
  }
}
```

With macros you don't need to write all that boilerplate just to display `ContentView()`. Now you can write this:

```
#Preview { ContentView() }
```

That's it! The `Preview` macro is a freestanding macro that produces everything we need to display a preview of the `View` passed to it.

In the second chapter we create a simple macro from start to finish. In the remainder of the book we create a bunch of others with different features so that hopefully you can figure out how to write the macro you want to write.

Then again, I want to make it clear that it's ok if you decide that you never want to write one of your own. In fact, this is the first book I've written where I consider it a success if you get to the end and say, "Nope, not for me."

"Daniel," you say, "this is an odd introduction. It's not very informative or inviting."

True. I just want to make sure that you really want to be here despite my warnings to the contrary. In fact, I may disuade you a bit more in our first section where you are introduced, or re-introduced to our fictional characters.

This is the third book in the series that features stories of the great detective Chamfered Edges and Captain Swiftly. If you have not met them yet, you may want to read this introduction from "The Curious Case of the Async Cafe".

We begin our journey with an usual proposal from Swiftly that will certainly change his relationship with Edges during this book and perhaps beyond.

Perhaps We Can Give It A Try

Edges and I hadn't been back to the Async Cafe for many months. In some sense, it felt as if no time had passed. But that's how it would feel at the Async Cafe.

"Swiftly," said Edges, sipping their espresso, "I'm sorry but I have to leave soon. I have a new case."

"I have to say," I said looking down at my cappuccino, "I'm jealous. Ever since I left the department store I've been so bored."

"I understand," said Edges, though I was sure they didn't.

"Actually," I said as Edges rose to leave, "I wanted to ask if I might work for you."

"What," asked Edges, "a sort of Archie Goodman to my Nero Wolfe?"

"No," I said quickly, "I'm not looking to move in. And I don't need to be paid a salary. My investments have been quite good. I just need something to challenge me."

"So," said Edges, "a Doctor Watson to my Sherlock Holmes?"

I shook my head, "not exactly. I'd like to just shadow you for a bit and see if there's anything useful I could do."

"You won't like it," said Edges. It's not like the books or the movies. The actual work is very tedious and it's easy to make a wrong move."

"I know," I said. I'd been on stakeouts before with Edges and there could be long periods of time where it didn't seem like anything productive was happening. "I think I might be useful."

"Well, mon ami," said Edges, "I would say 'don't do it.'" The great detective paused and shrugged and said, "we can give it a try."

"Splendid," I said, "I will start tomorrow."

"It might be better," cautioned my friend, "if you took the rest of the chapter to just look around and acclimate yourself. We shall begin in earnest in the next chapter."

I beamed as Edges nodded to me and turned to leave.

Walking away I was sure they said, "oh mon Dieu, he wishes to be the Captain Hastings to my Hercule Poirot."

As you begin your journey, armed with experience with some of the macros that Apple has implemented and released, you are certain they are the way forward for you.

Macros look like this new shiny thing that will make your life better.

They can and will.

And, you figure, I should make my own.

You can. This book will show you how you can make your own.

But each time you have the urge to create a new macro of your own, I'd like to echo Edges words and apply them to macros.

I would say, "don't do it."

And then I would shrug and agree that, perhaps, we can give it a try.

We Begin

There are two categories of macros. Freestanding macros might accept arguments and are replaced by the compiler with the implementation of the macro which might include using the arguments to customize the resulting code. Attached macros are attached to some declaration. They might be attached to a struct, enum, class, or even a function or method. An attached macro might also accept arguments and use them to supplement and modify the declaration they are attached to.

In this section we look at macros in standard Swift code. In fact, let's start with the multiplatform template that ships with Xcode. You'll see that we can't always spot a macro by looking at our code as the macro syntax looks like other familiar constructs we use.

Before we get started, let me show you where to find the sample code for this book.

Sample Code

Head to https://github.com/editorscut/ec015swiftmacros for the code that accompanies this book.

The code download has one folder for each chapter. This chapter's folder is *Chapter01*.

Inside *Chapter01* you'll find a folder for each section where there is code. Each of the other sections contains the finished code described in the corresponding section of the book.

Chapter01 > 02 folder contains a project named *Sample*. You can either start with this project or create a fresh one using the instructions below. Either method will give you exactly the same result. We aren't going to add any code in this section or the next.

I use a two-space indent for my code. You can always reformat code by selecting all of the code in a file with Command - A and re-indenting it with your preferences using Control - I.

A Starter Project

Open Xcode and choose to create a new project. You can either choose Create New Project... from the "Welcome to Xcode" window or you can use the menu item File > New > Project....

For your project template choose Multiplatform > App and click Next.

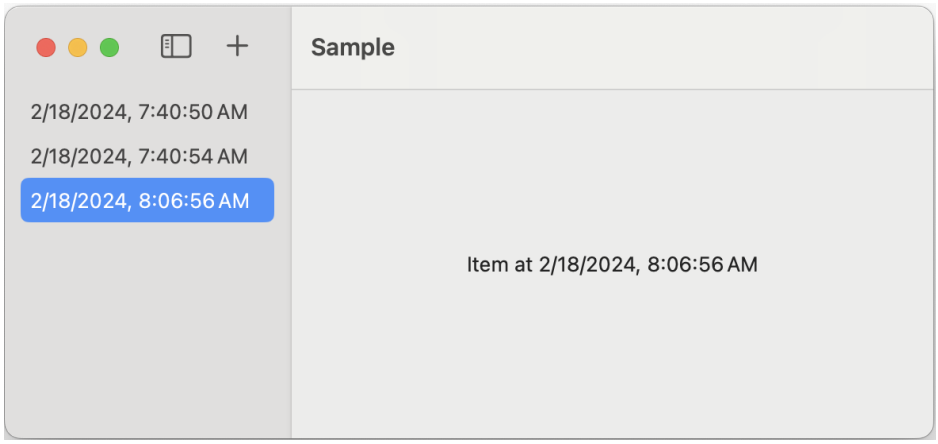
Use the product name *Sample* and *SwiftData* as the Storage type. The choices for `SwiftUI` as the Interface and `Swift` as the language have been made for you.

Click Next, pick a location for the project, and click Create.

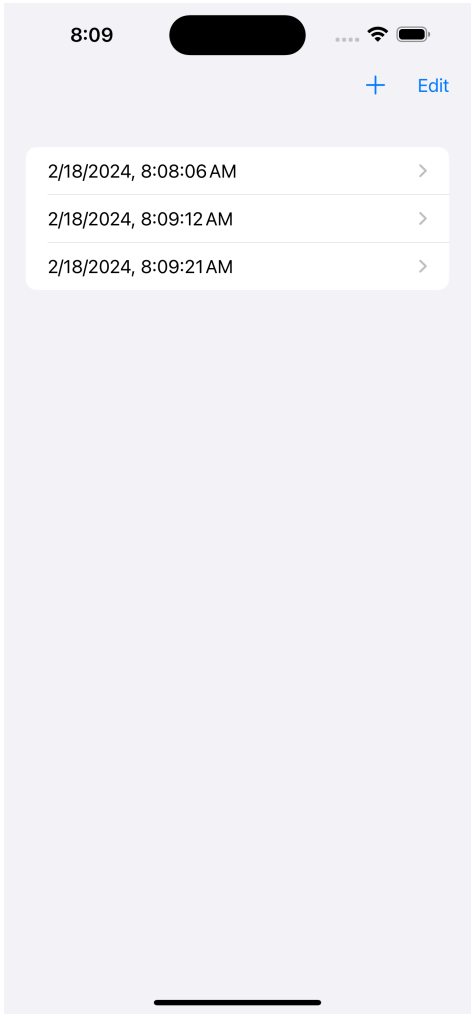
Run the app. You can choose to run it in the simulator or on your Mac or even on device.

The app should compile and build and you should see a `+` button. Tap it to add `Item`s with the current moment as the `timestamp`.

Here's what the app looks like on my Mac.



And here's what it looks like on an iPhone simulator.



Let's look at the code.

Not Freestanding Macros

In Swift, freestanding macros are denoted using `#`. Unfortunately, other things are also denoted using `#`.

As an example of something that is not a freestanding macro, look in *ContentView.swift*. Halfway down the `body` computed property you'll see the following code. I've highlighted the lines beginning with `#`.

*Sample/**ContentView.swift***

```
#if os(macOS)
      .navigationSplitViewColumnWidth(min: 180, ideal: 200)
#endif
      .toolbar {
#if os(iOS)
        ToolbarItem(placement: .navigationBarTrailing) {
          EditButton()
        }
#endif
```

In this case `#if` and `#endif` are Swift compiler directives. As you saw in the screenshots, the first conditional meant that we use a split view on the Mac but not in iOS and the second conditional resulted in an Edit button on iOS but not on the Mac.

At compile time this code will be included or excluded depending on the platform we're building for.

I mention compiler directives as you've probably seen or even used them and I want you to separate them from freestanding macros.

Actually, *ContentView.swift* contains an freestanding macro. At the bottom of the file you should see `#Preview`.

Freestanding Macros

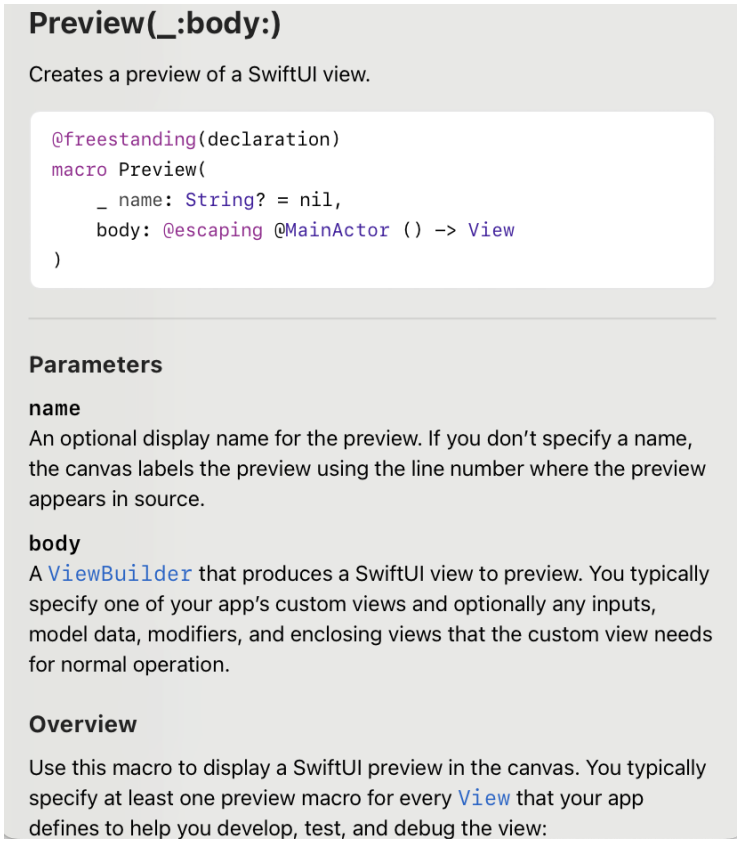
If you've worked with SwiftUI for years you've seen the standard boilerplate code for the SwiftUI preview. We had to name and declare a new struct, mark that it conformed to a particular protocol, and then implement the computed property that provided this conformance.

Apple engineers introduced the `Preview` macro that we use like this.

*Sample/**ContentView.swift***

```
#Preview {
  ContentView()
    .modelContainer(for: Item.self, inMemory: true)
}
```

Control - Click `#Preview` and select Show Quick Help to see this popover.



Preview(_:body:)

Creates a preview of a SwiftUI view.

```
@freestanding(declaration)
macro Preview(
    _ name: String? = nil,
    body: @escaping @MainActor () -> View
)
```

Parameters

name
An optional display name for the preview. If you don't specify a name, the canvas labels the preview using the line number where the preview appears in source.

body
A `ViewBuilder` that produces a SwiftUI view to preview. You typically specify one of your app's custom views and optionally any inputs, model data, modifiers, and enclosing views that the custom view needs for normal operation.

Overview

Use this macro to display a SwiftUI preview in the canvas. You typically specify at least one preview macro for every `View` that your app defines to help you develop, test, and debug the view:

`Preview` is a freestanding declaration macro. We'll create a macro of this type and also the other type of freestanding macro, the expression macro, later in the book.

`Preview` takes two parameters. The first is an optional display name for the preview that has default value `nil`. The second is the view to be displayed. It is usually entered, as in our preview above, as a trailing closure. In other words, it's the part enclosed in curly braces.

Control - Click `#Preview` again. This time select Expand Macro.

We see the code that the macro call is replaced by. The only thing I've modified in this generated code, believe it or not, is I've shortened the name of the generated struct.

I've highlighted all of the code that the macro added to its second parameter.

*Sample/**ContentView.swift***

```
struct $s6Sample33_CC6Cl7PreviewfMf_15PreviewRegistryfMu_
                          : DeveloperToolsSupport.PreviewRegistry {
    static let fileID: String = "Sample/ContentView.swift"
    static let line: Int = 56
    static let column: Int = 1

    static func makePreview() throws -> DeveloperToolsSupport.Preview {
        DeveloperToolsSupport.Preview {
          ContentView()
            .modelContainer(for: Item.self, inMemory: true)
        }
    }
}
```

The preview is named and the macro creates a struct with that name. The macro specifies that this struct conforms to the `PreviewRegistry` protocol. The macro adds properties for the file name, line number, and column number where the preview is created. It adds the `makePreview()` method that provides the conformance to `PreviewRegistry`, and it takes the closure we provided

that returned a view and uses it as the returned value in the body of `makePreview()`.

That's a lot that comes from us entering `#Preview {}`.

This is the siren call of macros. It's all of the code you don't write.

I again urge you to resist that call.

Next, let's look at an example of the other category of macros, the attached macro. But first, let's look at something that looks like an attached macro but isn't.

Not Attached Macros

We use `#` in the front of `#Preview` when we call the `Preview` macro because it is a freestanding macro.

An attached macro is used to operate on the struct, class, enum, or function it decorates. We recognize calls to an attached macro because they begin with `@`. But, of course, we already use `@` to indicate many things in Swift.

For example, we use `@MainActor` before a method to indicate that it should be run on the main actor.

A common example in SwiftUI are property wrappers.

Check out the top of *ContentView.swift* and you'll see `@Environment`.

*Sample/**ContentView.swift***
```
struct ContentView: View {
  @Environment(\.modelContext) private var modelContext
  @Query private var items: [Item]
```

Control - Click `@Environment` and choose Jump to Definition...

You'll see something like this.

```
@frozen @propertyWrapper public struct Environment<Value> : DynamicProperty {
    @inlinable public init(_ keyPath: KeyPath<EnvironmentValues, Value>)
    @inlinable public var wrappedValue: Value { get }
}
```

It's a property wrapper.

But doesn't it look just like the thing below it? Before you check out `@Query`'s definition, note that when you Control - Click it, there's no option to Expand Macro.

"Hmmm," I say to myself, "than it can't possible be a macro."

Of course this is wrong. Check out `@Query`'s definition.

Actually, there are many variations of `@Query` but this particular one has the following definition.

```
@available(iOS 17.0, macOS 14.0, tvOS 17.0, watchOS 10.0, *)
@attached(accessor)
@attached(peer, names: prefixed(`_`))
  public macro Query() = #externalMacro(module: "SwiftDataMacros",
                                        type: "QueryMacro")
```

So, in fact, `Query` consists of two different attached macros: an accessor macro and a peer macro.

We'll learn and write many types of attached macros in the course of this book but I wanted to start by pointing out that you can't always read code and assume you know whether something is a macro or not.

This simple usage of `@Query` looks similar to how we might use `@State` and `@Binding` and those are both property wrappers.

Unfortunately, we can't expand `@Query` and see what is generated for us but there is another macro in this sample code that we can expand and explore. We do that in the next section.