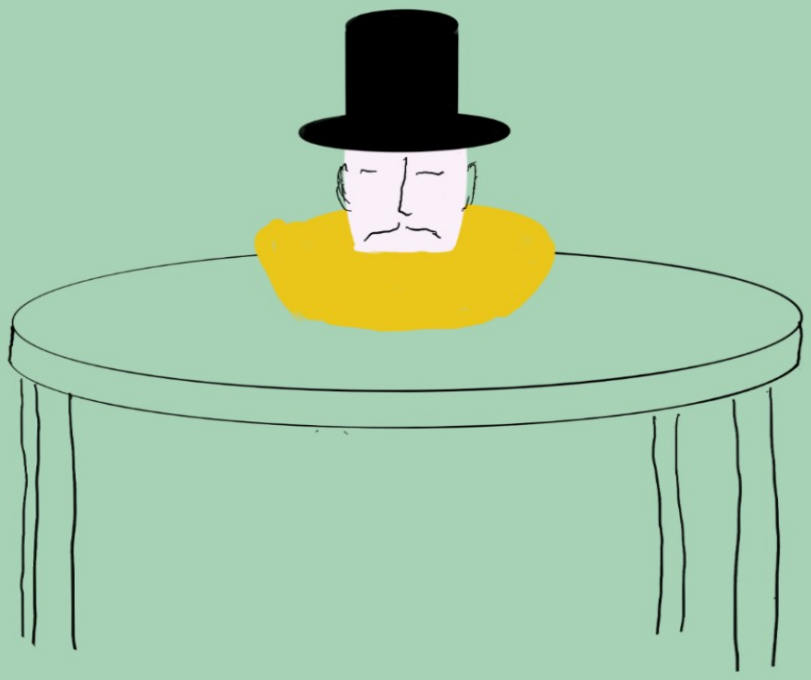
The Case of the Vanishing Bodies



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 5

More Attached Macros

In the previous chapter you were introduced to attached macros through an extended example of working with a member macro. The member macro allowed us to add a new property or method to a class, struct, enum, actor, or protocol.

In this chapter we meet many more attached macros. The extension macro allows us to create an extension that conforms to a given protocol. The accessor macro lets us add a `get`, `set`, `willSet`, or `didSet` to a property. The peer macro can be attached to a property to add a member at the same level and more. The memberAttribute macro can add annotations to members such as `@Published` and deprecation notices.

Another cool thing that we can do with attached macros that we can't do with freestanding macros is combine them. We can, and will, define a single macro that is both a peer macro and an accessor macro.

But wait, there's more.

We can use the memberAttribute macro to annotate a member with another attached macro.

I think you're going to enjoy this chapter. But first, let's revisit when it is appropriate to create a macro.

Extension Macros

"Swiftly," Edges said, "I think it is time we start billing our clients for some of your work."

"Ok," I said, trying to contain my excitement.

"We won't charge for everything," Edges continued, "only when we feel that your presence is contributing."

I nodded. "How," I asked, "will we remember what to bill for."

"We will make a notation like this," said Edges.

```
@Billable(client Name, rate: standard)
Research at Library(3 hours)
```

I added the `@Billable` *to the top of the event in our calendar. Suddenly the event transformed.*

```
Research at Library(3 hours)

Research is Billable
  client Name
  standard rate
```

"That," continued Edges, "will make it clear which items are `Billable` *and which are not."*

In this section we will create a macro that adds an extension that conforms to a given protocol.

Sometimes this conformance is automatic and there's nothing more for us to do than declare it. Other times we need to define a variable or implement a method to conform to the given protocol. The extension macro allows us to create either type.

We're going to add a `Hashable` and an `Identifiable` extension to an enumeration. Let's motivate our macro with an example.

The goal

Suppose you have an enumeration.

```
enum Sample {
  case one
  case two
  case three
}
```

It is trivial to conform `Sample` to `Identifiable`. We can add this extension.

```
enum Sample {
  case one
  case two
  case three
}

extension Sample: Identifiable {
  var id: Sample {
    self
  }
}
```

You could instead declare the type of `id` to be `Self` if you want. I'm going to leave it the way it is.

If the cases in `Sample` have associated values then `Sample` must also be `Hashable`.

```
enum Sample {
  case one(String)
  case two(Int)
  case three
}

extension Sample: Hashable {}

extension Sample: Identifiable {
  var id: Sample {
    self
  }
}
```

The two extensions are absolutely boiler plate. There is nothing interesting or dynamic about them. We're going to create an attached macro called an extension macro that adds these two extensions.

Defining our Macro

Create a new package with the Macro template named `EnumId` and make the usual changes to get it to compile. The macro will start as `#EnumID`. Alternately, start with the `EnumID` package in *Chapter05/EnumID* by double-clicking *Package.swift*. (The code in *Chapter05/02/EnumID* is the final code you'll have at the end of this section.)

Let's begin by adding a comment to the top of `EnumID` before the declaration.

*EnumID/Sources/EnumID/**EnumID.swift***
```
/// A macro that adds an extension to make an enum identifiable
/// It requires (but doesn't check)
/// that any associated values are identifiable
///
```

Change the definition to an attached macro of type extension. Also the declaration of the call can be simplified.

*EnumID/Sources/EnumID/**EnumID.swift***
```
@attached(extension)
public macro EnumID()
= #externalMacro(module: "EnumIDMacros",
                 type: "EnumIDMacro")
```

This is, of course, not sufficient. We have to add the names of the protocols we're declaring conformance to. In addition, we are adding a member named `id` which is also added to the definition.

*EnumID/Sources/EnumID/**EnumID.swift***
```
@attached(extension, conformances: Identifiable, Hashable,
          names: named(id))
public macro EnumID()
= #externalMacro(module: "EnumIDMacros",
                 type: "EnumIDMacro")
```

Next, let's implement the macro.

Implementing an Extension Macro

We've been through this next step before. Change `EnumIDMacro` to conform to `ExtensionMacro`, delete `expansion()`, and use the FixIt to stub out the appropriate version of `expansion()`.

*EnumID/Sources/EnumIDMacros/**EnumIDMacro.swift***

```
public struct EnumIDMacro: ExtensionMacro {
  public static func expansion(
    of node: AttributeSyntax,
    attachedTo declaration: some DeclGroupSyntax,
    providingExtensionsOf type: some TypeSyntaxProtocol,
    conformingTo protocols: [TypeSyntax],
    in context: some MacroExpansionContext
    ) throws -> [ExtensionDeclSyntax] {
    // ...
  }
}
```

The only thing we need to check for is that we are applying this macro to an enumeration. For simplicity, I've used a `fatalError()`. You may choose to use an `Error` or a `Diagnostic`.

*EnumID/Sources/EnumIDMacros/**EnumIDMacro.swift***

```
public struct EnumIDMacro: ExtensionMacro {
  public static func expansion(
    of node: AttributeSyntax,
    attachedTo declaration: some DeclGroupSyntax,
    providingExtensionsOf type: some TypeSyntaxProtocol,
    conformingTo protocols: [TypeSyntax],
    in context: some MacroExpansionContext
    ) throws -> [ExtensionDeclSyntax] {
    guard declaration.is(EnumDeclSyntax.self) else {
      fatalError("IdentifiableEnum can only be applied to an enum")
    }
    // ...
  }
}
```

If the type that `@EnumID` is annotating is an enum, we want to add two extensions. One conforms to `Hashable` and the other conforms to `Identifiable` and provides a computed property for `id`.

The init we're using for `ExtensionDeclSyntax()` can throw so we call each with a `try`. The `expansion()` method is declared with `throws` so there's nothing else we need do.

You'll also see that I use `type.trimmed`. This eliminates what SwiftSyntax refers to as `trivia` surrounding the type. This is the stuff that doesn't matter for our goals but would make it harder to write unit tests that are sensitive, for example, to extra spaces.

*EnumID/Sources/EnumIDMacros/**EnumIDMacro.swift***

```
public struct EnumIDMacro: ExtensionMacro {
  public static func expansion(
    of node: AttributeSyntax,
    attachedTo declaration: some DeclGroupSyntax,
    providingExtensionsOf type: some TypeSyntaxProtocol,
    conformingTo protocols: [TypeSyntax],
    in context: some MacroExpansionContext
    ) throws -> [ExtensionDeclSyntax] {
    guard declaration.is(EnumDeclSyntax.self) else {
      fatalError("IdentifiableEnum can only be applied to an enum")
    }
    return
     [
      try ExtensionDeclSyntax(
     """
     extension \(type.trimmed): Hashable {}
     """
     ),
      try ExtensionDeclSyntax(
     """
     extension \(type.trimmed): Identifiable {
       var id: \(type.trimmed) {
         return self
       }
     }
     """
     )
     ]
  }
}
```

Let's update *main.swift* and use the macro.

Using our macro

Enter our motivating example into `main`.

*EnumID/Sources/EnumIDClient/**main.swift***

```
import EnumID
import Foundation

enum Sample {
  case one(String)
  case two(Int)
  case three
}
```

Annotate `Sample` with our extension macro `EnumID`.

*EnumID/Sources/EnumIDClient/**main.swift***

```
import EnumID
import Foundation

@EnumID
enum Sample {
  case one(String)
  case two(Int)
  case three
}
```

Expand the macro and you'll see the two added extensions.

Of course, we wouldn't apply this macro if `Sample` already conformed to `Identifiable` but we might if `Sample` conformed to `Hashable`.

Although I won't work this example myself, you may want to consider how you might implement `@EnumID` so that it doesn't produce the extension with `Hashable` conformance if `Sample` already conforms to it.

For completion, here's a quick unit test that we can run.

*EnumID/Tests/EnumIDTests/**EnumIDTests.swift***

```
final class EnumIDTests: XCTestCase {
    func testMacro() throws {
        #if canImport(EnumIDMacros)
        assertMacroExpansion(
            """
            @EnumID
            enum Sample {
              case one(String)
              case two(Int)
              case three
            }
            """,
            expandedSource: """
            enum Sample {
              case one(String)
              case two(Int)
              case three
            }

            extension Sample: Hashable {
            }

            extension Sample: Identifiable {
              var id: Sample {
                return self
              }
            }
            """,
            macros: testMacros
        )
        #else
        throw XCTSkip("macros are only supported on the host platform")
        #endif
    }
}
```

This test should pass.