THE 5 MOST COMMON MISCONCEPTIONS ABOUT SwiftUI

MATTEO MANFERDINI

A SMALL PROMISE FROM YOU

It took me months of research, many years of experience, and hours of work to produce this guide. But I am happy to offer it to you completely free of charge.

In exchange I ask you for a small promise.

If after reading this guide you will think I did a good job, I ask you to share it with someone you think would also benefit from it.

In this way, I can spend less time looking for the people I can help and more producing great free material. You will find some links to share at the end of this guide. Thank you.

Let’s start.

ARE YOU READY FOR THE FUTURE?

SwiftUI is the future of UI development.

That's a message I heard loud and clear from many of my students and readers. And this is also the consensus on Reddit and other countless internet forums on iOS development.

I agree. Apple announcement about SwiftUI took the iOS development community by storm. While we all had gripes with tools like storyboards and Auto Layout, we got used to making apps with them. They were flexible but sophisticated to master. Still, no one saw SwiftUI coming.

That's why I share the belief that SwiftUI will reshape the way we create user interfaces. And that's not only true for iOS, but also for other platforms like iPadOS, macOS, and watchOS. The best part of all this is that we will be able to share code between all these platforms.

But, reading online forums and articles, my excitement has been quickly replaced by frustration. Everyone and their dog jumped on the SwiftUI bandwagon, producing superficial, out of context, and often contradictory blog posts. These are leaving many people more and more confused, whether they are approaching SwiftUI in particular or iOS development in general.

That brings me back to a day, years ago, when I started addressing such confusion on my blog. Many iOS developers struggled to structure the code of iOS apps properly. SwiftUI is a great tool, but the hype that surrounds it is recreating many of the problems I found back then.

I am not one that gives up easily, though, so I decided to create this guide.

And no matter your experience in iOS development, I designed this guide so that anyone can take something out of it. I believe that the best way to start with SwiftUI is to head in the right direction and spare you a lot of wasted time and frustration.

I hope you will enjoy it.

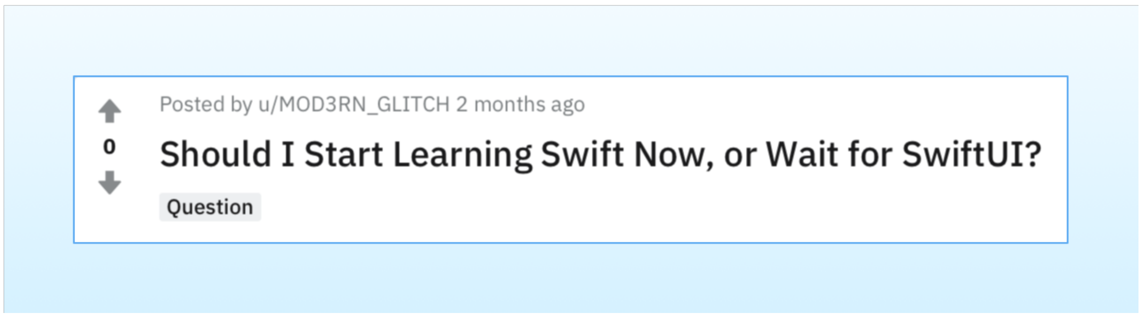
MISCONCEPTION #1: SWIFTUI WILL REPLACE STANDARD SWIFT

Not true.

SwiftUI is a UI framework, not a new language replacing Swift. I can hear some of you already thinking: *"Well, duh! Of course!"*. Especially if you are already somewhat experienced in iOS development.

But when everyone out there keeps screaming that *"SwiftUI is the new way to make iOS apps*," I don't find it surprising that beginners are confused. If SwiftUI is **the new way** of making iOS apps, many people approaching it for the first time might think that Swift, in turn, is the old way.

Look at this question I found on Reddit:



Posted by u/MOD3RN_GLITCH 2 months ago

0

Should I Start Learning Swift Now, or Wait for SwiftUI?

Question

I have seen many questions along those lines. So, if you are among those wondering which one of the two you should pick, the answer is **both**:

- *Swift* is the **programming language** you will use to make any iOS app, be it a simple to-do list or a complex social network.

- *SwiftUI* is a **UI framework** you can use to build the user interfaces of your apps. It is new because it replaces *UIKit*, the previous UI framework for iOS apps. As its name suggests, SwiftUI is based on Swift, so you have first to learn the language to use the framework.

There is also another reason why many confuse the two: the syntax of SwiftUI looks nothing like the syntax of standard Swift.

The reason is that **SwiftUI makes heavy use of some new Swift features**. These are advanced features that you probably didn't learn when learning programming. And even if you are more experienced, the chances are that you don't know them yet.

The good news is that you don’t need to understand how they work, only how to use them in SwiftUI. Large part of the code of your apps will not be related to the UI, so you will still need to write it in the standard Swift you know.

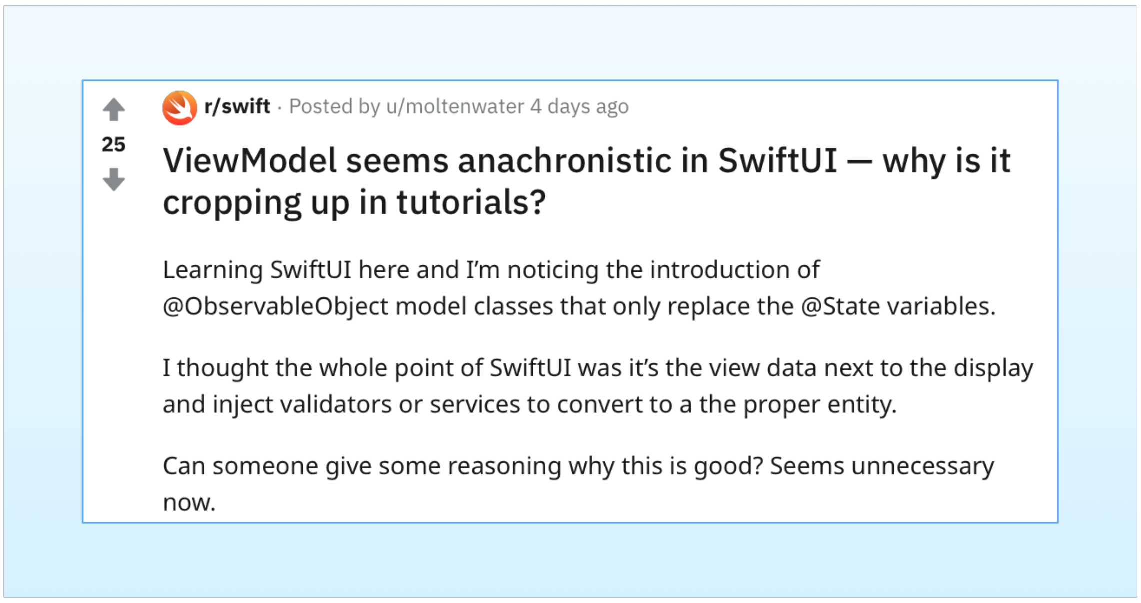
If you are interested, you find the list new Swift features in the appendix of this book.

MISCONCEPTION #2: YOU ARE SUPPOSED TO WRITE THE ENTIRE APP IN SWIFTUI

Not true.

This one is what frustrates me the most, and it does not seem only to affect beginners. Some people seem to believe that now, all the code in your app should reside inside SwiftUI views.

Look at this question I found on Reddit:



ViewModel seems anachronistic in SwiftUI — why is it cropping up in tutorials?

Learning SwiftUI here and I'm noticing the introduction of @ObservableObject model classes that only replace the @State variables.

I thought the whole point of SwiftUI was it's the view data next to the display and inject validators or services to convert to a the proper entity.

Can someone give some reasoning why this is good? Seems unnecessary now.

As I mentioned above, SwiftUI is a UI framework. That means **you should use it only to build the user interface of your app**. The fact that with SwiftUI, we now create user interfaces in code does not mean that that's where you should put all your code. The reason we did not write all code inside views in UIKit was not that we used storyboards. That was true even if you created all your UI in code. The same applies to SwiftUI.

Your apps still need to manage other tasks like business logic, storing data, performing network requests, interfacing with device sensors, and so on. Putting that code inside views violates basic software design principles like *separation of concerns*, *the principle of least knowledge*, and *don't repeat yourself*.

All that code goes into separate structures and classes. That is why SwiftUI has *bindings* and *observable objects*. Learn to use those features appropriately, or you will write unmanageable spaghetti code. And that **will prevent you from getting hired as an iOS developer**.

If you come from UIKit, you might recognize the old *massive view controller* problem, where all code ended inside view controllers. Unfortunately, the problem does not go away in SwiftUI. It just needs a new name. At least view controllers imposed some structure on UIKit code.

This second misconception is the underlying problem of a question I have seen again and again: **how do you integrate Core Data with SwiftUI**? If you read this section carefully, you should already have the answer. You do it exactly as you would in a UIKit app.

Many developers seem to have problems because they can't make some specific features of Core Data work in SwiftUI, for example predicates or fetch requests. The problem though is not that SwiftUI lacks Core Data integration.

The problem is that **you are not supposed to put Core Data code inside your UI code**. You should not do it in UIKit, and you should not do it in SwiftUI either.

MISCONCEPTION #3: SWIFTUI THROWS AWAY THE MVC AND MVVM PATTERNS

Not true.

Yes, it is true that, unlike UIKit, which uses view controllers, SwiftUI does not impose any structure on your code. What that means is that now you have more freedom to shoot yourself in the foot. So, in SwiftUI, **architectural design patterns are more important, not less**.

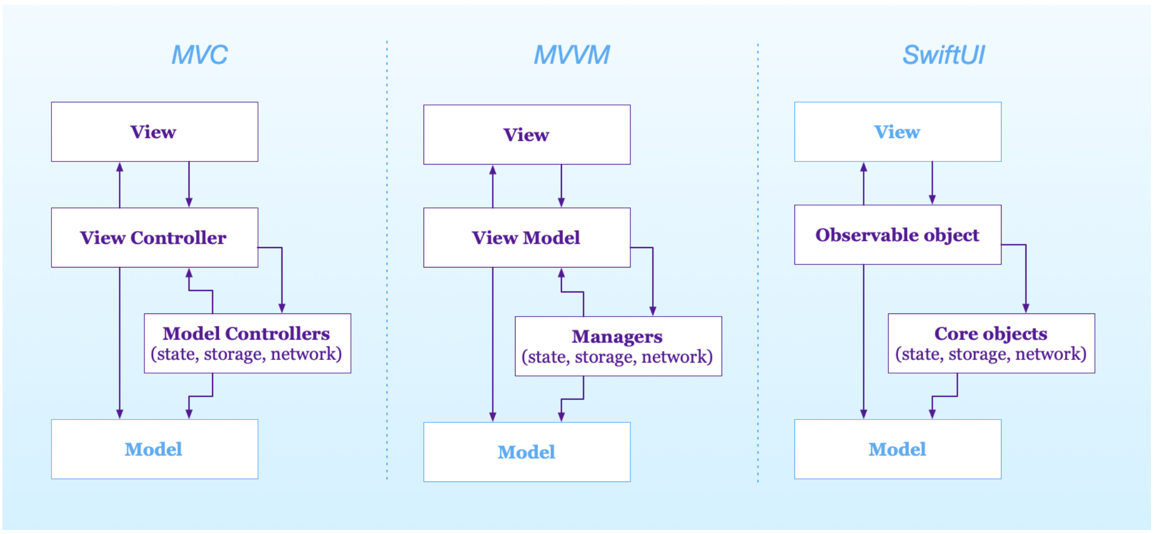
For a long time, the preferred design patterns for iOS apps have been Model-View-Controller (MVC) and Model-View-ViewModel (MVVM). The question is: **are they still valid in SwiftUI**? The short answer is: yes, they are.

The misconception here seems to be that many believe that these design patterns are tied to UIKit and its view controllers. But that's not true. Both models predate UIKit and even iOS development. Apple already used the MVC pattern for Mac development before the iPhone was invented. But it's even older than that, having been introduced in the Smalltalk language in the 1970s. The same is true for MVVM, which was invented by Microsoft architects in 2005.

So, both MVC and MVVM are here to stay. The question then becomes: **which of the two should you use**?

It does not matter. The reason is simply that, if you analyze them accurately, they are the same pattern with different names. For years I have been teaching that the MVC pattern in iOS does not have only three layers, but four. The extra layer comes from splitting the controller layer into view controllers and model controller. This is not something I invented. You can find the idea in this old Apple guide from 2012.

That also means that the idea of view controllers predates iOS and UIKit. View controllers are a structural concept, which in UIKit was implemented by the UIViewController class. So, when you put MVC and MVVM side-by-side, you don't need to be an expert in graph theory to see that they are the same pattern.



Admittedly, this is a simplistic diagram, and SwiftUI's version of MVC requires a much longer discussion. Still, the same structure roughly applies to SwiftUI as it did to UIKit. SwiftUI's *observable objects* are the equivalent of view controllers in MVC and view models in MVVM. Even Apple engineers called those objects *models* during WWDC 2019's presentations on SwiftUI. I'll write more about this topic soon.

MISCONCEPTION #4: YOU NEED TO USE THE COMBINE FRAMEWORK TOGETHER WITH SWIFTUI

Not true.

Combine is a new *functional reactive programming* framework (FRP) introduced by Apple alongside SwiftUI. Now, it is true that SwiftUI uses Combine behind the scenes. But **that does not force you to use it yourself**.

In fact, there is no reference to Combine features in SwiftUI code, and you don't need even to import Combine in any of your project files. To tell the whole truth, that was the case in the first beta versions of SwiftUI. Before the introduction of the *@Published* property wrapper, you had to use Combine publishers explicitly. This contributes in part to the misconception.

Still, some people believe that, since you should use the MVVM pattern in SwiftUI, you must also use an FRP framework like Combine. But while MVVM was initially designed with reactive programming in mind, it does not need it.

MVVM is an architectural design pattern. As such, **it only dictates the structure of your app**, not which tools you use for communication between its layers. Combine is just a tool you can use, but you don't have to. Moreover, FRP in general, and Combine in particular, are complex concepts that take time to understand even for experienced developers. If all the beginners that approach iOS development need to start with Combine, we are screwed.

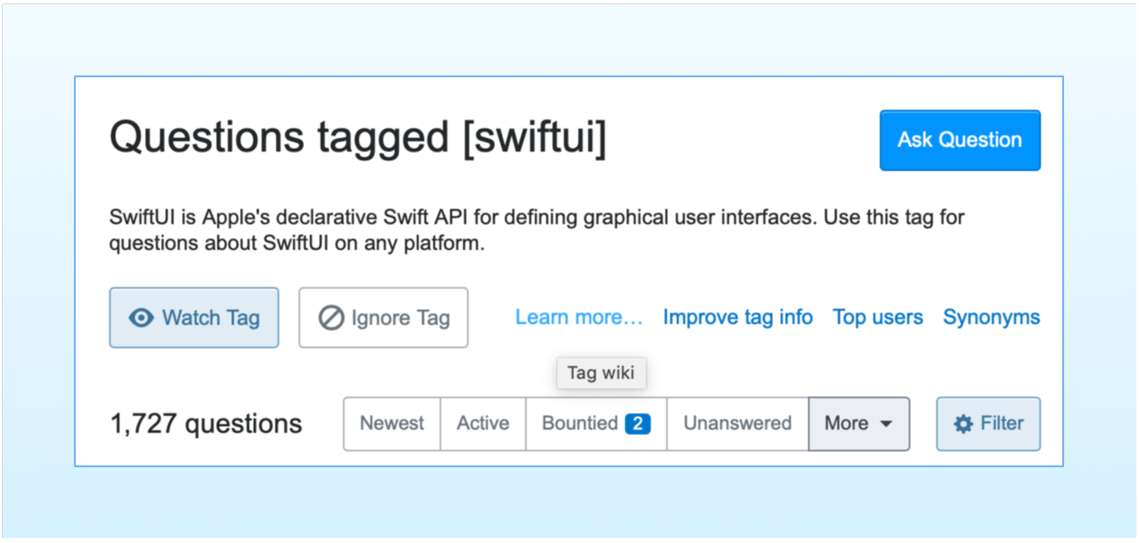
Some might argue that Combine is what the "cool kids" are using nowadays, so you have to learn it anyway. I was recently talking with one of my students, who confessed that, at his current company, they adopted some paradigms because that's what they found in online articles.

I would caution you against this line of thinking.

The blogs of a few vocal developers are not a reflection of the whole industry. I worked at many companies and had many clients. None, except one, used FRP in their projects. And that single one that did had the messiest codebase I've ever seen. Always take what you read online with a grain of salt. Even what I write.

Don't believe me yet?

At the moment of writing, there are 1.727 questions related to SwiftUI on Stack Overflow.

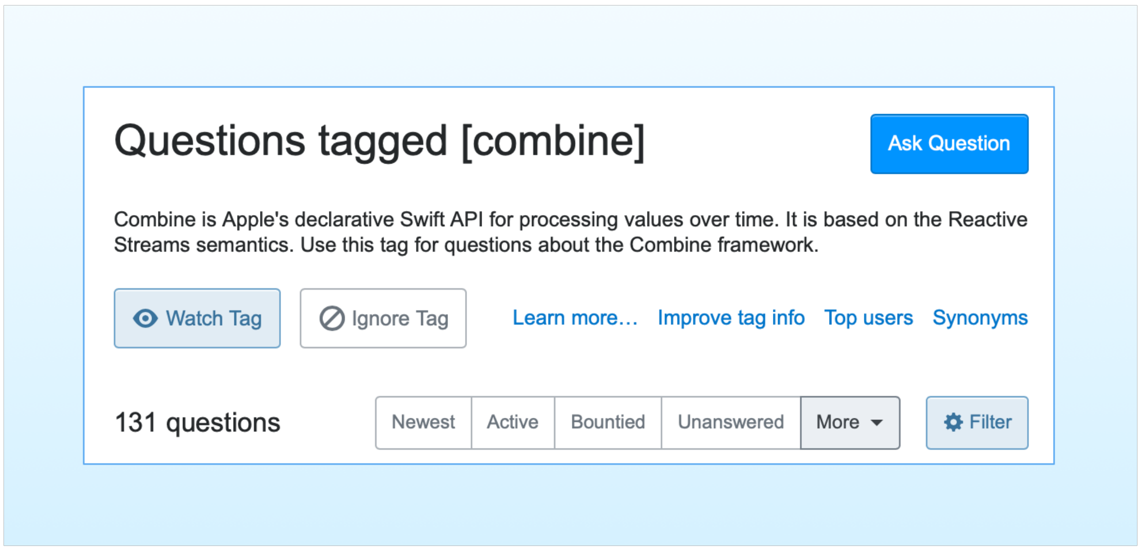


Questions tagged [swiftui]

SwiftUI is Apple's declarative Swift API for defining graphical user interfaces. Use this tag for questions about SwiftUI on any platform.

1,727 questions

The ones about Combine are not even 10% of that.



Questions tagged [combine]

Combine is Apple's declarative Swift API for processing values over time. It is based on the Reactive Streams semantics. Use this tag for questions about the Combine framework.

131 questions

But don’t take my word for it. Go and check yourself. You can find the same result if you check the two tags in Apple’s forums.

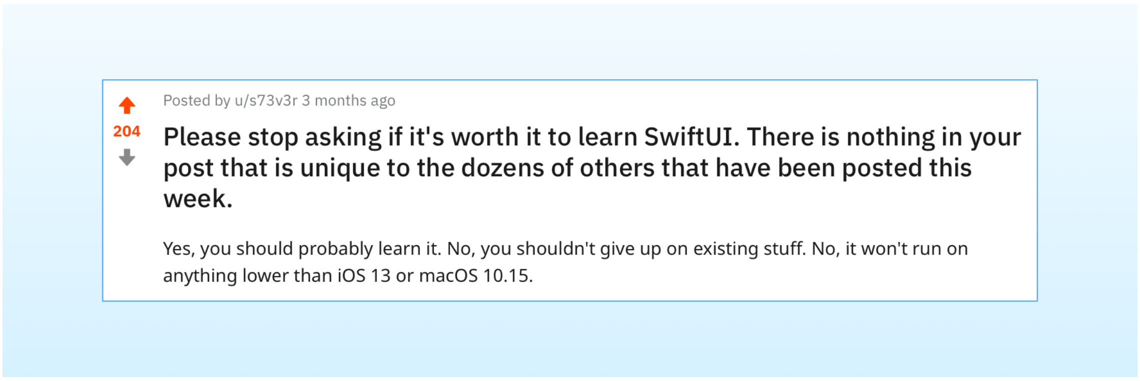
That says a lot about the popularity of Combine in the iOS community, in my opinion. Especially since it's a much harder framework to understand than SwiftUI. I expected it to have many more questions than that.

I am not saying, of course, that you should never use Combine. If you like it, go for it. But you don’t have to just because the “cool kids” do.

MISCONCEPTION #5: SWIFTUI IS NOT READY FOR PRODUCTION BECAUSE IT MISSES MANY FEATURES OF UIKIT

Well, it depends.

For starters, one thing is definitely true. **SwiftUI is available only in iOS 13, iPadOS and macOS Catalina**. That's not going to change, no matter how many times people ask about it.



Posted by u/s73v3r 3 months ago

Please stop asking if it's worth it to learn SwiftUI. There is nothing in your post that is unique to the dozens of others that have been posted this week.

Yes, you should probably learn it. No, you shouldn't give up on existing stuff. No, it won't run on anything lower than iOS 13 or macOS 10.15.

That means that most companies that already have published apps in the App Store with a large user base won't be able to use SwiftUI for a while. Users adopt new versions of iOS is slowly. Any business must serve the majority of their customers if they want to survive.

How long that takes, depends on the business. Many told me that for new apps, they plan to go with SwiftUI. But the most common policy I have seen is to support the **current version of iOS, minus two**. That means that most companies adopt SwiftUI in their apps **only when iOS 15 will be released** somewhere in 2021.

The other thing that is also true is that SwiftUI does not have all the features of UIKit. The most often cited missing feature in SwiftUI is UIKit's collection views. That's less true after the new features Apple introduced at WWDC 2020, especially the new grids. Still, those are only available in iOS/iPadOS 14 and macOS Big Sur, which pushes their adoption further into the future.

But well, it's not that surprising that SwiftUI is still incomplete. UIKit has existed for many years, so it will take a while for SwiftUI to catch up.But this is where the misconception comes.

SwiftUI can use any UIKit class, including collection views. So, while it's technically accurate that SwiftUI does not have those features, you can still use them in any SwiftUI app. All you need to do is use the UIViewRepresentable and UIViewControllerRepresentable protocols. These allow you to wrap any UIKit component in a SwiftUI view.

These protocols are not hard to adopt once you understand how they work, even though, at the time of writing, their documentation is lacking. And they work in the same way, so you need to make an effort only once.

So, **if missing features like text fields or collection views are the only thing holding you back**, you can go on and use SwiftUI for your apps. Apple did a pretty good job integrating UIKit into SwiftUI. While it's true that UIKit's collection views in SwiftUI are not any better than before, they are also not worse.

At least, you will get the benefits of SwiftUI in other parts of your app, while you wait for it to catch up with UIKit.

APPENDIX: THE NEW SWIFT FEATURES THAT MAKE SWIFTUI POSSIBLE

Some Swift developers are puzzled by SwiftUI's syntax, which looks nothing like regular Swift.

```swift
struct Title: View {
    @State var captionText: String = "Dolor sit amet"

    var body: some View {
        VStack {
            Text("Lorem ipsum")
                .font(.title)
            Text(captionText)
                .foregroundColor(.blue)
        }
    }
}
```

There is a reason. To enable the declarative syntax of SwiftUI, Apple made some behind the scenes additions to the Swift language (which sparked a lengthy discussion on the Swift forums about how much the community drives the evolution of Swift).

Since these are pure language features, they are not limited to SwiftUI only. You can use them in any part of your code.

For the more experienced developers among you, here is a list of those features. But if you are approaching SwiftUI or iOS development for the first time, feel free to skip this list. These are advanced features that few developers use and understand. You can learn SwiftUI without knowing what these features are or how they work.

Here is the list:

- **Implicit returns** allow you to omit the *return* keyword in many functions.

This feature is new but might look familiar, since it was already possible to use it inside closures. In Swift 5.1, it was extended to any function with a single expression. That's why you don't see any return statement in SwiftUI code.

- **Opaque result types** are responsible for that strange *some* keyword you see before the return type of a view body.

Opaque types allow you to use protocols as types for properties and functions, which was not previously possible.

In a way, opaque types are the opposite of generics. While generics can specify constraints on types, a generic is always resolved to a concrete type when you use it. So, a caller always needs to specify a type for a generic explicitly. For example, the array structure is generic, but you can only create arrays of specific types, like integers, strings, etc.

Opaque types allow you to hide the nature of a returned value. All the caller knows about an opaque type is that it conforms to a specific protocol and nothing else.

- **Property wrappers**, allow the *@State*, *@Binding*, *@ObservedObject*, *@Environment* and *@EmvironmentObject* modifiers you see in SwiftUI code.

Property wrappers are a powerful feature that allows you to attach custom code to any stored property. This means that each of the property wrappers of SwiftUI I listed above defines a different, custom behavior.

The good news is that you don't need to understand how property wrappers work to use SwiftUI. All you have to learn is what those property wrappers mean. For example, a property with the *@State* wrapper causes a view to re-render whenever it is changed. That's all you need to know, and you don't need to care about how it does that.

- **Result builders** are responsible for the declarative syntax of SwiftUI.

Result builders allow you to create domain-specific languages in Swift. *Domain-specific languages* (DSLs) are languages you embed inside another programming language, which help you deal with a specific domain using an ad-hoc syntax. Such syntax reflects more the particular field you are working on than a generic programming language, making it easier to write code and avoiding repetitions.

SwiftUI is a DSL, and that's why its syntax looks different from normal Swift. In SwiftUI, you *declare* how an interface should look, something that would not be possible in an imperative programming language.

- **Method chaining** is used to create the various SwiftUI view modifiers.

A *view modifier* is one of those methods you append to SwiftUI views to change some of their properties. For example, some modifiers affect the font, the color, the size, or the padding of a view.

In standard Swift code, you would assign these values to stored properties, but SwiftUI modifiers work differently. Instead of altering the view to which you apply them, they return a new view that embeds the previous one.

Admittedly, method chaining is not a new feature of Swift and was a technique that you could also use in previous versions of the language. Nevertheless, it's an advanced technique, and it requires understanding other advanced features of Swift, so the chances are that you never used it in your code.

PLEASE SHARE THIS GUIDE

I hope you enjoyed this guide and it helped you improve your understanding of SwiftUI. It took me many hours of work to put it together, but I am happy to offer it to you free of charge.

If you found it useful, please take a moment to share it with someone that could also find it useful. In this way, I can dedicate more time into creating more free articles and guides to help you in your iOS development journey.

Think about colleagues and friends that could find this guide useful and send them this link through email, on forums, or through a private message, so that they can get it together with all the other material I only share with my email list:

https://matteomanferdini.com/the-5-most-common-misconceptions-about-swiftui/

You can also use one of these links to share the guide on your favorite social network.

Click here to share it on Facebook

Click here to share it on Twitter

Click here to share it on LinkedIn