< Understanding property wrappers in Swift and SwiftUI

What is the @State property wrapper? >

All SwiftUI property wrappers explained and compared

Paul Hudson @twostraws February 9th 2021

Updated for Xcode 13.2

Updated for iOS 14

SwiftUI offers 17 property wrappers for our applications, each of which provide different functionality. Knowing which one to use and when is critical to getting things right, so in this article I'm going to introduce you to each of them, and give you clear guidance which to use.

TL;DR

I'm going to explain more in a moment, but here's the "too long; didn't read" summary that describes roughly what each wrapper does, whether it owns its data or not (i.e. whether the data belongs to it and is managed by it), along with links to more:

- `@AppStorage` reads and writes values from `UserDefaults`. This owns its data. More info.
- `@Binding` refers to value type data owned by a different view. Changing the binding locally changes the remote data too. This does not own its data. More info.
- `@Environment` lets us read data from the system, such as color scheme, accessibility options, and trait collections, but you can add your own keys here if you want. This does not own its data. More info.
- `@EnvironmentObject` reads a shared object that we placed into the environment. This does not own its data. More info.
- `@FetchRequest` starts a Core Data fetch request for a particular entity. This owns its data. More info.
- `@FocusedBinding` is designed to watch for values in the key window, such as a text field that is currently selected. This does not own its data.
- `@FocusedValue` is a simpler version of `@FocusedBinding` that doesn't unwrap the bound value for you. This does not own its data.
- `@GestureState` stores values associated with a gesture that is currently in progress, such as how far you have swiped, except it will be reset to its default value when the gesture stops. This owns its data. More info.
- `@Namespace` creates an animation namespace to allow matched geometry effects, which can be shared by other views. This owns its data.
- `@NSApplicationDelegateAdaptor` is used to create and register a class as the app delegate for a macOS app. This owns its data.
- `@ObservedObject` refers to an instance of an external class that conforms to the `ObservableObject` protocol. This does not own its data. More info.
- `@Published` is attached to properties inside an `ObservableObject`, and tells SwiftUI that it should refresh any views that use this property when it is changed. This owns its data. More info.
- `@ScaledMetric` reads the user's Dynamic Type setting and scales numbers up or down based on an original value you provide. This owns its data. More info.
- `@SceneStorage` lets us save and restore small amounts of data for state restoration. This owns its data. More info.
- `@State` lets us manipulate small amounts of value type data locally to a view. This owns its data. More info.
- `@StateObject` is used to store new instances of reference type data that conforms to the `ObservableObject` protocol. This owns its data. More info.
- `@UIApplicationDelegateAdaptor` is used to create and register a class as the app delegate for an iOS app. This owns its data. More info.

Storing temporary data

When it comes to storing data in your app, the simplest property wrapper is `@State`. This is designed to store value types that are used locally by your view, so it's great for storing integers, Booleans, and even local instances of structs.

In comparison, `@Binding` is used for simple data that you want to change, but is *not* owned by your view. As an example, think of how the built-in `Toggle` switch works: it needs to move between on and off states, but it doesn't want to store that value itself so instead it has a binding to some external value that we own. So, our view has an `@State` property, and the `Toggle` has an `@Binding` property.

There is a variation of `@State` called `@GestureState`, specifically for tracking active gestures. This isn't used so often, but it does have the benefit that it sets your property back to its initial value when the gesture ends.

For more advanced purposes – i.e., dealing with classes, or sharing data in many places – you should not use `@State` and `@Binding`, instead, you should create your object somewhere using `@StateObject`, then use it in other views with `@ObservedObject`.

A simple rule is this: if you see "state" in the name of a property wrapper, it means that views definitely owns the data.

So, `@State` means simple value type data created and managed locally but perhaps shared elsewhere using `@Binding`, and `@StateObject` means reference type data created and managed locally, but perhaps shared elsewhere using something like `@ObservedObject`.

This is important: if you ever see `@ObservedObject var something = SomeType()` it should almost certainly be `@StateObject` instead so that SwiftUI knows the view should own the data rather than just refer to it elsewhere. Using `@ObservedObject` here can sometimes cause your app to crash because the object is destroyed prematurely.

If you find yourself handing the same data from view to view to view, you'll find the `@EnvironmentObject` property wrapper useful. This lets you read a reference type object from a shared environment, rather than passing it around explicitly.

Just like `@ObservableObject`, `@EnvironmentObject` should *not* be used to create your object initially. Instead, create it in a different view and use the `environmentObject()` modifier to inject it into the environment. Although the environment will automatically keep ownership of your object, you can also use `@StateObject` to store it wherever it was originally created. This is not required, though; putting an object into the environment is enough to keep it alive without further ownership.

The final state-based property wrapper is `@Published`, which is used *inside* your reference types to annotate the properties. Any property marked with `@Published` will cause its parent class to announce that a change has occurred, which in turn will cause any view observing that object to make any changes it needs.

Storing long-term data

SwiftUI has three property wrappers designed to store and/or retrieve data.

The first is `@AppStorage`, which is a wrapper around `UserDefaults`. Every time you read or write a value from app storage, you're actually reading or writing from `UserDefaults`.

The second is `@SceneStorage`, which is a wrapper around Apple's state restoration APIs. State restoration is what allows an app to be closed and reloaded, and come back to the same state the user left off – it makes it look like our apps were always running, even though they were silently terminated.

`@AppStorage` and `@SceneStorage` *are not* secure and should *not* be used to store sensitive data.

Although `@AppStorage` and `@SceneStorage` sound the same, they are not: `@AppStorage` stores one value for your entire application, whereas `@SceneStorage` will automatically save multiple values for the same data for times when the user has your app window open multiple times – think iPadOS and macOS.

So, you might use `@AppStorage` to store global values such as "what is the user's high score?", and you might use `@SceneStorage` to store "what page is the user reading right now?"

The third data property wrapper is `@FetchRequest`, which is used to retrieve information from Core Data. This will automatically use whichever managed object context is in the environment, and update itself when the underlying data has changed.

Reading environment data

SwiftUI has two properties wrappers for reading the user's environment: `@Environment` and `@ScaledMetric`.

`@Environment` is used to read a wide variety of data such as what trait collection is currently active, whether they are using a 2x or 3x screen, what timezone they are on, and more. It also has a couple of special application actions, such as exporting files and opening a URL in the system-registered web browser.

`@ScaledMetric` is much simpler, and lets us adapt the size of our user interface based on a user's Dynamic Type settings. For example, a box that is 100x100 points might look great using the system default size, but with `@ScaledMetric` it will automatically become 200x200 when a larger Dynamic Type setting is enabled.

Referring to views

SwiftUI has provides the `@Namespace` property wrapper, which creates a new namespace for animations. Animation namespaces let us say "animate views with an ID of 9", and all views in that namespace with the ID 5 will be animated.

You can share namespaces between views by using the property type `Namespace.ID` and injecting the `@Namespace` value from whichever view created it. This allows you to created matched geometry effect animations across views, rather than storing all the data in the current view.

Application handling

If you ever need access to the old `UIApplicationDelegate` and `NSApplicationDelegate` methods and notifications, you should use the `@UIApplicationDelegateAdaptor` and `@NSApplicationDelegateAdaptor` property wrappers respectively.

You provide these with the class of your app delegate, and they will make sure an instance is created and sent all appropriate notifications.

Sources of truth

Earlier I described which property wrappers own their data, and really this comes to sources of truth in your application: wrappers that own their data are sources of truth because they create and manage the value, and wrappers that do not own their data are not sources of truth because they get the value from somewhere else.

Property wrappers that are sources of truth

These create and manage values directly:

- `@AppStorage`
- `@FetchRequest`
- `@GestureState`
- `@Namespace`
- `@NSApplicationDelegateAdaptor`
- `@Published`
- `@ScaledMetric`
- `@SceneStorage`
- `@State`
- `@StateObject`
- `@UIApplicationDelegateAdaptor`

Property wrappers that are not sources of truth

These get their values from somewhere else:

- `@Binding`
- `@Environment`
- `@EnvironmentObject`
- `@FocusedBinding`
- `@FocusedValue`
- `@ObservedObject`

If you remember nothing else...

I want to make a new property owned by the current view. You should use `@State` for value types, and `@StateObject` for reference types.

I want to refer to a value created elsewhere. You should use `@Binding` for value types, and either `@ObservedObject` or `@EnvironmentObject` for reference types.

Similar solutions...

- SwiftUI tips and tricks
- How to use Instruments to profile your SwiftUI code and identify slow layouts
- Building a menu using List
- Understanding property wrappers in Swift and SwiftUI
- Answering the big question: should you learn SwiftUI, UIKit, or both?

< Understanding property wrappers in Swift and SwiftUI

What is the @State property wrapper? >