

objc ↕ Thinking in SwiftUI

Updated for iOS 17

By Chris Eidhof and Florian Kugler

1 Introduction 4

2 View Trees 7

View Builders 9

Render Trees 15

Identity 17

3 State and Binding 22

State 24

Observable Macro 31

ObservableObject Protocol 39

Bindings 47

View Updates and Performance 53

Which Property Wrapper for What Purpose? 54

4 Layout 56

Leaf Views 60

View Modifiers 65

Container Views 74

Alignment 82

5 Environment 91

Reading from the Environment 93

Custom Environment Keys 94

Custom Component Styles 98

Environment Objects 103

6 Animations 105

Controlling Animations 109

The Animatable Protocol 116

Transitions 120

7 Advanced Layout 131

The Layout Protocol 132

Preference-Based Layout 137

Variadic Views 144

Coordinate Spaces 145

Anchors 146

Matched Geometry Effect 148

Introduction

1

When SwiftUI came out, it was a radical departure from UIKit. We wrote the first edition of this book to help you build a mental model of the way SwiftUI works. A few years have passed since then, and we've had the opportunity to teach this material to many teams of developers, large and small. During this process, we continued to improve and refine our approach of explaining SwiftUI's fundamentals based on the feedback from the workshops. This new edition of Thinking in SwiftUI is the result of that journey: we rewrote the entire book from the ground up to be on par with the way we teach SwiftUI in our workshops.

While Apple's SwiftUI API documentation has improved a lot over the years, we still believe that there's a need for more conceptual documentation explaining how SwiftUI works. Just as with the first edition, this is still the focus of this book. We hope to facilitate a solid conceptual understanding of SwiftUI so that you can learn about the continuously expanding platform-specific APIs on your own.

We believe that a key aspect of working efficiently with SwiftUI is to understand how the code we write translates into view trees. We cover this aspect in detail in the first chapter, and then we move on to discuss how these view trees are interpreted in terms of state, layout, animations, and more.

To this end, we included more visual explanations in this edition, partly thanks to a completely revamped infrastructure for generating the book. We moved from a LaTeX-based workflow to a pure Swift/TextKit-based tool, which allows us to embed SwiftUI views directly into the book. In addition to simplifying our toolchain, this allowed us to generate many diagrams, illustrations, and previews that hopefully help explain the otherwise somewhat abstract concepts.

While we were wrapping up this edition of the book, WWDC23 took place, and Apple announced a series of new and updated SwiftUI APIs. We added explanations throughout the book for many of the new APIs, but we took care to explicitly mention wherever we use an iOS 17-only API (which also means macOS 14 or any of the other platforms released at the same time).

As we've observed many times in our workshops, the best way to learn SwiftUI is by writing code yourself. This book cannot replace that, but it aims to be a helpful companion. We encourage you to regularly put what you've learned from this book into practice. Nothing will make your insights stick better than experimenting with them and seeing for yourself how things work.

We'd like to thank everyone who helped us during the writing of this book. Thank you Natalye for proofreading, Ole for the technical review, and Marcin for helping with TextKit. We'd like to thank Robb, Ole, and Juul for helping us improve our workshops, which in turn improved this book. We'd like to thank the previous readers of our books and attendees of our workshops for all the feedback you gave us. And lastly, a big thank you to the creators of SwiftUI.

Florian and Chris

View Trees

2

View trees and render trees are perhaps the most fundamental and important concepts to understand to work with SwiftUI. To achieve the layout we want, we need to understand how view trees are constructed. To understand how state works in SwiftUI, it's important to understand the lifetime of a view and how it's related to the view tree we're building. Understanding the lifetime is equally important to writing efficient SwiftUI code that only loads data and updates views when needed. Finally, animations and transitions also require an understanding of view trees.

For example, consider the following view:

Code	View Tree	Preview
<pre>Text("Hello") .padding() .background(Color.blue)</pre>	<pre>graph TD background[.background] --- padding[.padding] background --- color[Color] padding --- text[Text]</pre>	

To the right of the code, we can see the corresponding view tree. The background modifier is at the root of our view tree. Its primary subview — the view the background is applied to — is the padded text, and it's drawn on top. The secondary subview is the blue color, and it's drawn behind the primary subview. Each time we apply a view modifier like padding or background to the text view, it gets wrapped in another layer. Looking at a chain of view modifiers like in the example above, we have to read from the bottom up to visualize the resulting view tree; the last view modifier, background in this example, becomes the topmost view in the view tree.

Note that the background view modifier itself doesn't draw anything. Even though the background modifier is the topmost view in the view tree, the actual background (the blue color) is still drawn behind the text.

Here's a slightly different version of the example, with padding and background swapped around:

Code	View Tree	Preview
<pre>Text("Hello") .background(Color.blue) .padding()</pre>	<pre> .padding .background / \ Text Color </pre>	

The background is now the immediate parent of the text, and the padding is the parent of the background. In the [Layout chapter](#), we'll go into detail on why the layout differs, but put simply, the layout changed because we constructed a different view tree.

View Builders

SwiftUI uses a special syntax for constructing lists of views, called *view builders*. View builders are built on top of Swift's result builder feature, which was added to the language specifically for this purpose. For example, here's how we can construct a view that displays an image next to a text:

Code	View Tree	Preview
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") }</pre>	<pre> HStack / \ Image Text </pre>	

The HStack initializer takes a closure as a parameter, and that closure is marked as `@ViewBuilder`. This allows us to write a number of expressions inside — each of which represents a view. In essence, the closure passed to the stack builds a *list* of views, which become subviews of the stack in this example.

Looking at the declaration of the `ViewBuilder` struct, we can see the method below for handling a list of two views:

```
extension ViewBuilder {
    public static func buildBlock<C0, C1>(_ c0: C0, _ c1: C1) ->
        TupleView<(C0, C1)> where C0 : View, C1 : View
}
```

Since the stack in our example above has two view expressions inside, the view builder's `buildBlock` method with two parameters will be called. As we can see from the return type, this constructs a `TupleView` wrapping our two views: the image and the text. We can think of a view builder as a mechanism to construct a tuple view that represent lists of views.

If we write just one view expression in the view builder closure, this won't be wrapped in a tuple view, but simply passed on as-is. However, for our mental model, we can consider this exception to be a list of exactly one view.

SwiftUI uses view builders in many places. All container views like stacks and grids, as well as modifiers like background and overlay, take a view builder closure to construct their subviews. Furthermore, the body property of each view is implicitly marked with `@ViewBuilder`, as is the `body(content:)` method of view modifiers. We can also use the `@ViewBuilder` attribute to mark our own properties and methods as view builders, as we'll soon see in an example.

To better understand how lists of views are used by SwiftUI and how they can be composed, let's extend the example from above a bit:

Code	View Tree
<pre>HStack(spacing: 20) { Image(systemName: "hand.wave") Text("Hello") Spacer() Text("And Goodbye!") Image(systemName: "hand.wave") }</pre>	<pre>graph TD HStack --> Image1[Image] HStack --> Text1[Text] HStack --> Spacer[Spacer] HStack --> Text2[Text] HStack --> Image2[Image]</pre>

Now the stack has five subviews, which are represented as a tuple view with five elements. For better readability, we might want to break up stacks that grow large — which actually happens quite frequently in practice — into separate components. Here's one way we could do that:

```
struct Greeting: View {  
  @ViewBuilder var hello: some View {  
    Image(systemName: "hand.wave")  
    Text("Hello")  
  }  
}
```

```

@ViewBuilder var bye: some View {
    Text("And Goodbye!")
    Image(systemName: "hand.wave")
}

var body: some View {
    HStack(spacing: 20) {
        hello
        Spacer()
        bye
    }
}

```

By marking a property with `@ViewBuilder`, we're using view builder syntax in the property's body, just like we would in body or within the closure of a stack.

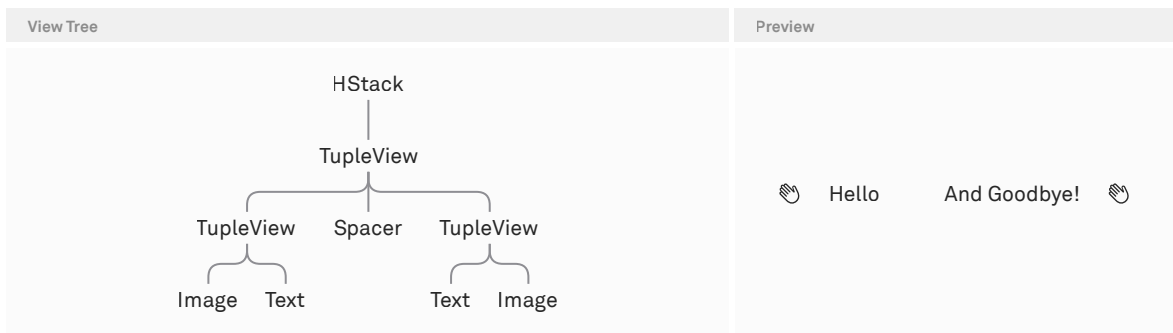
Looking at the type of the `HStack`'s view builder closure, we now have a `TupleView` with three elements — a tuple view, the spacer, and another tuple view:

```

TupleView<(
    TupleView<(Image, Text)>,
    Spacer,
    TupleView<(Text, Image)>
)>

```

However, to the `HStack`, this is exactly the same as before, when we wrote all five views directly in the stack's view builder closure. The stack still has five subviews, as we can see by the stack's spacing being applied between each of them.



With the exception of in the diagram above, we omitted the `TupleView`s in the view tree diagrams to make them more readable. We can read the lines between a parent view and its subview(s) as a tuple view.

This is a special property of view lists: when a container view like the `HStack` iterates over the view list, nested lists are recursively unfolded so that a tree of tuple views turns into a flat list of views. This even applies if we were to refactor the hello and bye view builder properties into separate views:

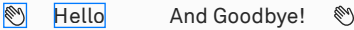
```
struct Hello: View {
  var body: some View {
    Image(systemName: "hand.wave")
    Text("Hello")
  }
}
```

```
struct Bye: View {
  var body: some View {
    Text("And Goodbye!")
    Image(systemName: "hand.wave")
  }
}
```

```
struct Greeting: View {
  var body: some View {
    HStack(spacing: 20) {
      Hello()
      Spacer()
      Bye()
    }
  }
}
```

Since the body of the `Hello` and `Bye` views are themselves view lists with two elements, they get unfolded when the stack iterates over its subviews like it did before.

We can also apply view modifiers to view lists, but the behavior might be somewhat surprising. For example, we could apply a border to the `Hello` view:

Code	Preview
<pre>HStack(spacing: 20) { Hello() .border(.blue) Spacer() Bye() }</pre>	

This will apply the border to each element of the view list, so both the image and the text have separate borders drawn around them. One common scenario where we might encounter this behavior is with using Group, which is a layout-agnostic abstraction around a view builder:

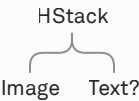
Code	Preview
<pre>struct Greeting: View { var body: some View { HStack { Group { Image(systemName: "hand.wave") Text("Hello") } .border(.blue) } } }</pre>	

Since the result of the group is a tuple view with two elements, the border will be applied to each of the two views. We can use this technique to our advantage if we want to apply the same modifiers to each view. However, we found that this can get confusing quickly if it's overused, because the behavior of the modifiers is so different from what we'd normally expect in all other contexts.

There's an exception to this and we're not sure whether this is intentional behavior: when placing the group, including the modifiers, as the root view or as the only subview within a scroll view, the group behaves like a VStack, and the modifiers aren't applied to each individual view within the group. When placing a group within an overlay or background, it behaves like an implicit ZStack, presenting another exception to the rule.

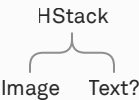
Dynamic Content

View lists constructed with view builders can be dynamic, too. Here’s how we can conditionally include a view:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if showText { Text("Hello") } }</pre>	

Looking at the diagram, we can see that the HStack still has two subviews: an image, and an optional text. From this view tree, SwiftUI knows that the stack will always have an image as the first subview, and perhaps a text as the second subview.

Instead of an if statement, we can also use other statements — such as if let, switch, or if/else — to create conditional views:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } }</pre>	

The view tree diagrams in this chapter (and the book in general) have been generated automatically from the type of the views. The opaque return type `some View`, which is used in most places in SwiftUI, hides complex nested view types, which encode the exact structure of the view tree. The type of a view also specifies exactly which parts are static and which are dynamic, giving SwiftUI full knowledge of which views can be dynamically inserted or removed.

Render Trees

SwiftUI uses the view tree to construct a persistent *render tree*. View trees themselves are ephemeral: we like to think of view trees as blueprints, since they get constructed and then thrown away over and over again. Nodes in the persistent render tree, on the other hand, have a longer lifetime: they stay around across view rerenders and are then updated to reflect the current state.

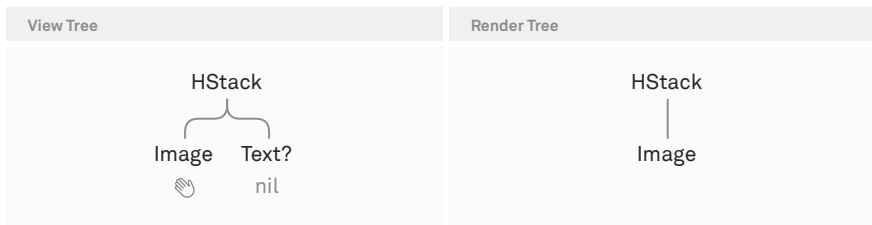
To distinguish between the two, we'll talk about *views* when talking about elements in the view tree, and *nodes* when talking about elements in the render tree. In this way, we can talk about the process of converting views into nodes as “rendering.” Note that we never deal with the render tree directly, as it's internal to SwiftUI.

The render tree doesn't actually exist, but it's a useful model to understand how SwiftUI works. In reality, SwiftUI has something called the attribute graph, which includes more than just the rendered views; it also contains the state and tracks dependencies. Apple calls the nodes in the render tree *attributes*.

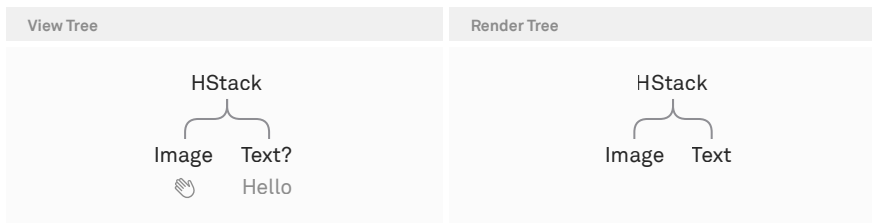
When we first display a SwiftUI view, the render tree that's constructed is mostly a one-to-one representation of the view tree. Consider the example from before:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } }</pre>	

When the greeting value is nil, the render tree for the view above only has one subview, the image node, inside the HStack.



When greeting changes to a non-nil value, the view gets reconstructed, and the render tree is then updated based on the new view tree: SwiftUI knows there will always be an HStack, so it doesn't need to touch this part of the render tree. It also knows there will always be an image as the first subview. Both of these views are completely static.



Once the view update mechanism inspects the conditional view, it knows that the condition might have changed. When the condition changes from nil to a non-nil value, SwiftUI inserts a Text node into the render tree. Likewise, when the condition changes from non-nil to nil, SwiftUI removes the Text node from the render tree. When a node is removed from the render tree, any associated state disappears as well. We'll talk more about this in the [State chapter](#).

There's one more scenario in this example for updating the render tree: we have a non-nil greeting value before and after the update, so the render tree will have the same text node before and after the update as well. However, if the value of greeting has changed, then the string of the text node will be updated.

Lifetime

As we mentioned above, the view tree itself is ephemeral — the concept of a lifetime doesn't make sense here. However, nodes in the render tree have a specific lifetime: from when they're first rendered, to when they're no longer needed for display.

However, the lifetime of nodes in the render tree isn't the same as their visibility onscreen. If we render a large VStack in a scroll view, the render tree will contain nodes for all subviews of the VStack, no matter if they're currently onscreen or not.

VStack renders its contents eagerly, as opposed to its LazyVStack counterpart. But even with a lazy stack, nodes in the render tree will be preserved when they go offscreen to maintain their state (we'll go into more detail about this in the [State chapter](#)). The bottom line is that nodes in the render tree have a lifetime, but it's not under our control.

For practical purposes, SwiftUI provides three hooks into lifetime events:

1. `onAppear` is executed each time a view appears onscreen. This can be called multiple times for one view even though the backing node in the render tree never went away. For example, if a view in a `LazyVStack` or `List` is scrolled offscreen and back onscreen repeatedly, `onAppear` will be called each time. The same is true when we switch tabs in a `TabView`: each time we switch to a tab, and not just the first time the tab is displayed, its `onAppear` will be called.
2. `onDisappear` is executed when a view disappears from the screen. This is the counterpart to `onAppear` and works using the same rules (it can be called multiple times even when the backing node doesn't go away).
3. `task` is a combination of the two used for asynchronous work. This modifier creates a new task at the point where `onAppear` would be called, and it cancels this task when `onDisappear` would be invoked.

Identity

Since view trees in SwiftUI don't consist of reference types (objects) that have intrinsic identity, SwiftUI assigns identity to views using their position in the view tree. This kind of identity is called *implicit identity*. To illustrate this, let's take a look at a slightly modified version of the example above:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") if let g = greeting { Text(g) } else { Text("Hello") } }</pre>	<pre>graph TD HStack --- Image HStack --- ConditionalContent ConditionalContent --- Text1[Text] ConditionalContent --- Text2[Text]</pre>

Instead of just an optional text, the view tree now contains a `ConditionalContent` view with two subviews: a text for the non-nil case, and another text for the nil case. Each of the views in the view tree is uniquely identifiable by its position in the tree. As an

illustration of this concept, think about constructing a “path” string to identify each view:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") // 0 if let g = greeting { Text(g) // 1.ifBranch } else { Text("Hello") // 1.elseBranch } }</pre>	

Image is "0", because it's the first subview of the HStack. The Text in the non-nil branch of the if let statement is "1.ifBranch", because the ConditionalContent is the second subview of the HStack, and the Text is the first subview of the ConditionalContent.

We're not suggesting that these path strings are how SwiftUI implements implicit identity under the hood; rather, they're just a human-friendly model to demonstrate what's meant by implicit identity.

Now consider the two text views in the two branches of the if let statement. They have different identities, and therefore are considered two distinct views by SwiftUI. When the condition changes, the old text will be removed from the render tree, and a new text will be inserted. This has all kinds of consequences in terms of state, animations, and transitions, which we'll discuss later on.

Let's take a look at the same example, but written a bit differently:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text(greeting ?? "Hello") }</pre>	

We can immediately see that the view tree now is simpler: the HStack has two static subviews, the image and the text. Now the difference between a nil and a non-nil

greeting value is just the string that's displayed by the text view. The text view itself, as described by its implicit identity (second subview of the HStack), will always be around and unaffected by any changes to the value of *greeting*.

Along with implicit identity, views can also have an *explicit identity*. This is mostly used for views in a `ForEach`, where each item in the `ForEach` is assigned an explicit identifier — for example, a unique identifier of the underlying data (either by conforming the items to the `Identifiable` protocol, or by providing a key path to a unique identifier). However, we can also assign explicit identifiers manually using the `id` modifier.

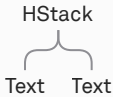
The `id` parameter can be any Hashable value. In the example below, we're using a Boolean by comparing the greeting value to `nil`. If it's `nil`, the explicit identifier is `true`. Otherwise, it's `false`. This means that SwiftUI considers the text view to be a different view when the identifier changes. Again, this will remove the previous text node from the render tree and insert a new one.

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text(greeting ?? "Hello") .id(greeting == nil) }</pre>	<pre>graph TD HStack -- 0 --> Image HStack -- 1 --> id[id] id --> true[true] true --> Text</pre>

It's important to note that an explicit identifier like the one above doesn't override the view's implicit identity, but is instead applied on top of it. In other words, SwiftUI won't be confused by using the same explicit identifiers on multiple views. As we saw, the path of the view is one way to give the implicit identity a concrete form, and we can think of explicit identifiers as "appending to the path."

With a more solid understanding of view identity in SwiftUI at hand, let's take a look at two common issues related to this topic.

First, let's consider the following example:

Code	View Tree
<pre>HStack { let v = Text("Hello") v v }</pre>	

Here, we're constructing a text view in a local variable and then using it twice in the `HStack`. What does this mean in terms of the identity of the two text views?

Clearly, the text views are located at different positions in the view tree, as the tree shows. Therefore, they have different implicit identity and are considered separate views by SwiftUI. We can also think of this in terms of the “blueprint” idea: we're creating a blueprint for a text view with the string "Hello", and then we're using this blueprint twice.

Here's another example related to view identity — it's a popular pattern for writing a little view extension that conditionally applies a view modifier:

```
// Anti-pattern  
extension View {  
  @ViewBuilder  
  func applyIf<V: View>(_ condition: Bool, transform: (Self) -> V) -> some View {  
    if condition {  
      transform(self)  
    } else {  
      self  
    }  
  }  
}
```

The applyIf method can be used like this:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") .applyIf(highlighted) { \$0.background(.yellow) } }</pre>	<pre>graph TD HStack --> Image HStack --> ConditionalContent ConditionalContent --> background[.background] ConditionalContent --> Text1[Text] background --> Text2[Text] background --> Color[Color]</pre>

Looking at the resulting view tree, we can see that using the applyIf modifier has introduced a ConditionalContent with two subviews: an unmodified text, and a text with a background. This means that when the condition (highlighted) changes, the identity of the text onscreen will have changed as well.

We strongly recommend not using this pattern, since the seemingly innocuous applyIf modifier introduces a branch in the view tree that might have unforeseen consequences downstream. Instead, the following is much safer:

Code	View Tree
<pre>HStack { Image(systemName: "hand.wave") Text("Hello") .background(highlighted ? .red : .clear) }</pre>	<pre>graph TD HStack --> Image HStack --> background[.background] background --> Text background --> Color</pre>

Most view modifiers take optionals so that we can use either the ternary operator pattern to specify a value, or nil if we don't want to specify a value. For example, the width and height of frame are optional, the color of foregroundColor is optional, and the length of padding is optional or can be set to zero. For the same reason, view modifiers like bold or disabled take a Boolean argument, although one might naively think that the argument isn't necessary.

State and Binding

3

In the previous chapter, we saw how view trees are constructed as blueprints and how they're translated into the persistent render tree. In order to build dynamic applications, we construct different view trees based on the current state and rely on SwiftUI to update the render tree accordingly. This is one of SwiftUI's greatest advantages: it observes state automatically and always keeps our views in sync with the model.

In general, the view update cycle can be summarized like this:

1. The view tree is constructed.
2. Nodes in the render tree are created, removed, or updated to match the current view tree.
3. Some event causes a state change.
4. This process repeats.

In principle, we don't have to worry about when the view tree needs to be recreated, which parts are affected by a state change, or what has to be updated onscreen to match the current view state, because SwiftUI takes responsibility for all of that. Instead, our job is to describe what should be onscreen given a specific state.

As a disclaimer, we should add that there are times when we need to think about which parts of our view tree are being rerendered and for what reason. If we run into performance problems, it's very likely that overly broad view updates play a role. We'll discuss this more at the end of this chapter.

SwiftUI comes with several different wrapper types for state, depending on whether the state is a value or an object, and whether it's private to the view or should be passed in from the outside. However, we usually don't have to deal with these wrapper types directly, since SwiftUI exposes all of them via property wrappers like `@State`, `@StateObject`, and `@ObservedObject`.

As of iOS 17, the way SwiftUI interfaces with objects has changed completely. SwiftUI no longer relies on the Combine framework for observation, and instead uses a macro-based solution, which also renders the `@StateObject` and `@ObservedObject` property wrappers superfluous. The `@State` property wrapper is now used for values and objects, whereas we usually only used it for values pre-iOS 17.

Since `@State` is relevant across all versions of SwiftUI, we'll start with an in-depth look at this property wrapper, and then we'll distinguish between the pre- and post-iOS 17 world with regard to observing objects.

State

The `@State` property wrapper is the easiest way to introduce state to a SwiftUI application. It's meant to be used for private view state values. For example, here's a simple counter view:

```
struct Counter: View {
    @State private var value = 0
    var body: some View {
        Button("Increment: \(value)") {
            value += 1
        }
    }
}
```

When the counter is rendered the first time around, the state property will have its initial value of 0. During the execution of the body property, SwiftUI notices that the state property is accessed, and it adds a dependency between the value state property and the counter view's node in the render tree. As a result, whenever value changes (e.g. because the button is tapped), SwiftUI will reexecute the counter view's body.

Note that if we don't include the value inside the button's label, SwiftUI is smart enough to figure out that it doesn't need to rerender the counter's body when the state property changes.

We might wonder how the value property can ever change, since we're assigning 0 to it each time the counter view gets initialized. To shed some light on this behavior and to take some of the magic away, here's how we could write the same code without using the `@State` property wrapper:

```
struct Counter: View {
    private var _value = State(initialValue: 0)
    private var value: Int {
        get { _value.wrappedValue }
        nonmutating set { _value.wrappedValue = newValue }
    }

    var body: some View {
        Button("Increment: \(value)") {
            value += 1
        }
    }
}
```


Instead of relying on `@State`, we're now creating a `State` value ourselves and assigning it to the `_value` property. The `State(initialValue:)` initializer makes it clear that the value `0` is just the *initial* value of the state property. This is the value that will be used when the node for the counter view is first created in the render tree. Once the node is there, the initial value of the state property will be ignored, and SwiftUI takes care of keeping the current value around across renders.

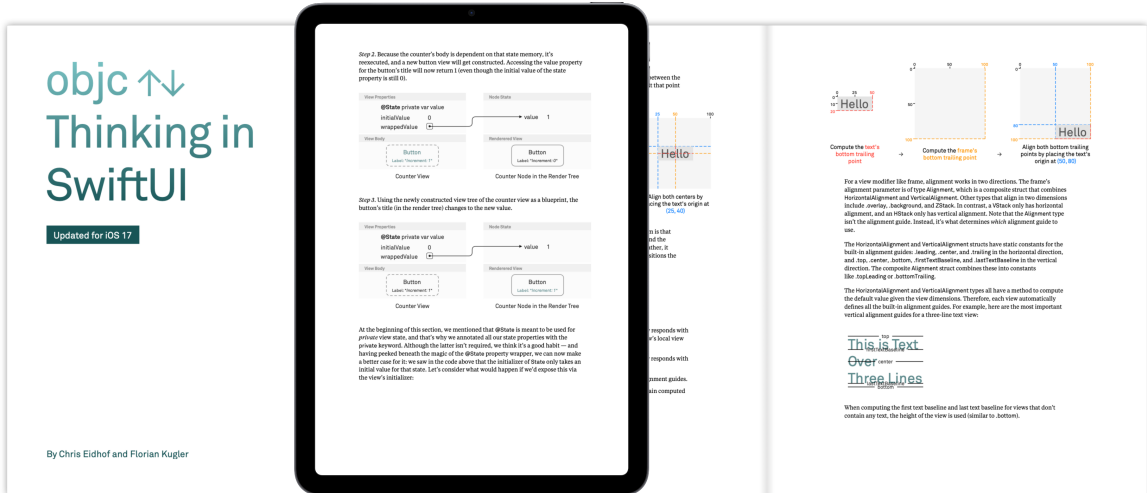
In addition to the `_value` property, we also added a computed value property, which makes the state easier to use: instead of having to write `_value.wrappedValue` each time we want to read or write, we can use `value`, and the computed property will forward that to `_value.wrappedValue` transparently. When we use the `wrappedValue` of a state property in the view's body, what we're really dealing with is a reference to the persistent state value in the render tree.

The `@State` property wrapper does all this for us: it creates the underscored version of the property (storing the actual `State` value), as well as the computed property that forwards the getter and setter to the `wrappedValue`.

Let's go through two scenarios: the initial rendering of the view above, and the second render when the button gets tapped. Here's what happens when the counter view first appears onscreen:

To make the state diagrams throughout this chapter more readable, we have highlighted the changes compared to the diagram in the previous step using [this color](#).

Step 1. When the `Counter` struct is constructed for the first time, no corresponding node in the render tree exists yet. In the diagram below, the view struct is on the left. The upper part of the view struct symbolizes the state property, which, in turn, has two internal values: the `initialValue`, which is the value we assigned during initialization of the property, and `wrappedValue`, which is the value we're interacting with in the view's body. We can think of `wrappedValue` as a pointer to the actual value of this state property, which currently doesn't point to anything yet. The lower part of the view struct symbolizes the view's body, which hasn't yet been executed and therefore is still empty.



Thanks for checking out the preview of Thinking in SwiftUI!

We hope you've enjoyed the content so far. If you'd like to read on and learn more about how SwiftUI works, please consider buying a copy of this eBook here:

<https://www.objc.io/books/thinking-in-swiftui>

You'll learn more about how SwiftUI's state system works, how you can use the environment and preferences to your advantage, all about the layout system, animations, and much more.