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.

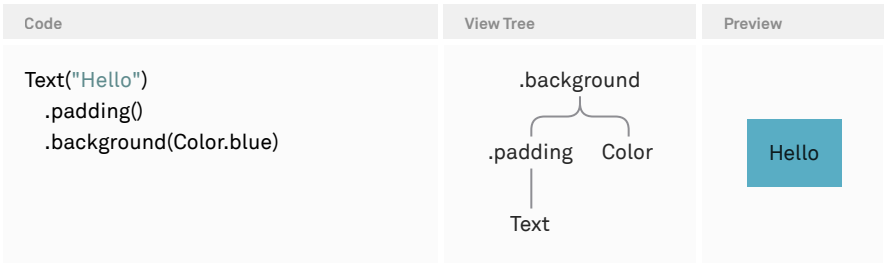
Florian and Chris

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.

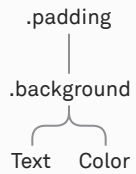
For example, consider the following view:



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:

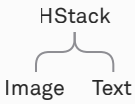
Code	View Tree	Preview
Text("Hello") .background(Color.blue) .padding()		

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

HStack {
Image(systemName: "hand.wave")
Text("Hello")
}
		

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.

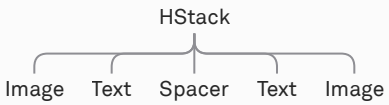
SwiftUI uses view builders in many places. All container views like stacks and grids, as well as modifiers like background and overlay, take a view builder closure to construct their subviews. Furthermore, the body property of each view is implicitly marked with `@ViewBuilder`, as is the `body(content:)` 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

```
HStack(spacing: 20) {
  Image(systemName: "hand.wave")
  Text("Hello")
  Spacer()
  Text("And Goodbye!")
  Image(systemName: "hand.wave")
}
```

View Tree



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")
  }
```

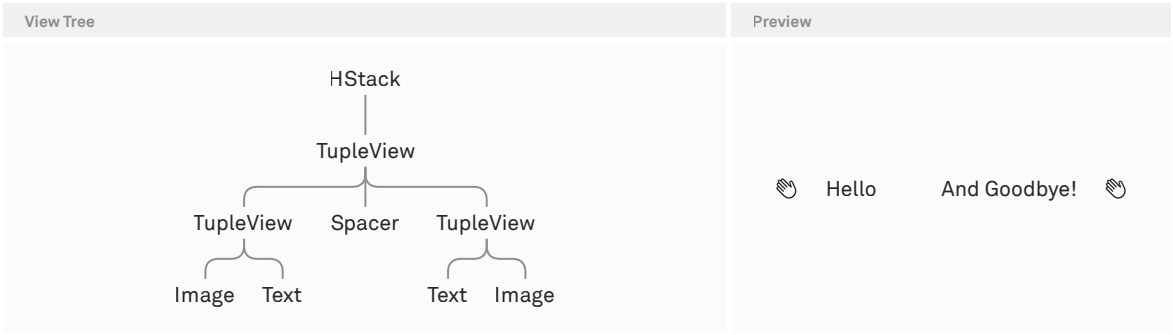
```
  @ViewBuilder var bye: some View {
    Text("And Goodbye!")
    Image(systemName: "hand.wave")
  }

  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 TupleViews 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()
    }
  }
}
```

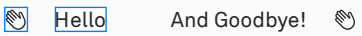
Since the body of the Hello and Bye views are themselves view lists with two elements, they get unfolded when the stack iterates over its subviews like it did before.

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

```
HStack(spacing: 20) {
  Hello()
    .border(.blue)
  Spacer()
  Bye()
}
```

Preview



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

```
struct Greeting: View {
  var body: some View {
    HStack {
      Group {
        Image(systemName: "hand.wave")
        Text("Hello")
      }
      .border(.blue)
    }
  }
}
```

Preview

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.

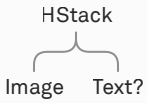
Dynamic Content

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

Code

```
HStack {
  Image(systemName: "hand.wave")
  if showText {
    Text("Hello")
  }
}
```

View Tree



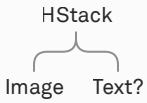
Looking at the diagram, we can see that the HStack still has two subviews: an image, and an optional text. From this view tree, SwiftUI knows that the stack will always have an image as the first subview, and perhaps a text as the second subview.

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

Code

```
HStack {
  Image(systemName: "hand.wave")
  if let g = greeting {
    Text(g)
  }
}
```

View Tree



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.

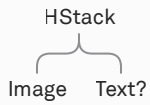
> The render tree doesn't actually exist, but it's a useful model to understand how SwiftUI works. In reality, SwiftUI has something called the attribute graph, which includes more than just the rendered views; it also contains the state and tracks dependencies. Apple calls the nodes in the render tree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

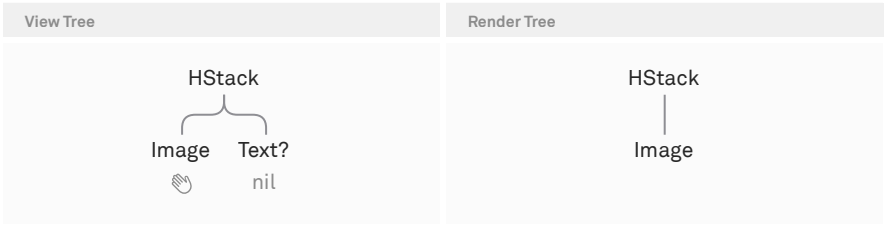
Code

```
HStack {
  Image(systemName: "hand.wave")
  if let g = greeting {
    Text(g)
  }
}
```

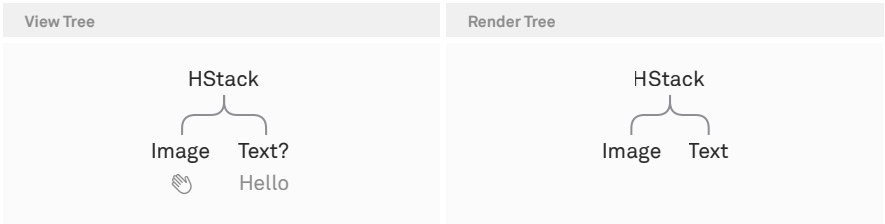
View Tree



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.

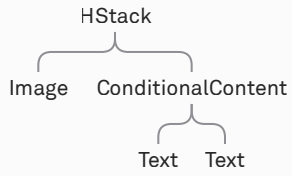
Identity

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

```
HStack {
  Image(systemName: "hand.wave")
  if let g = greeting {
    Text(g)
  } else {
    Text("Hello")
  }
}
```

View Tree



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

```
HStack {
  Image(systemName: "hand.wave") // 0
  if let g = greeting {
    Text(g) // 1.ifBranch
  } else {
    Text("Hello") // 1.elseBranch
  }
}
```

View Tree

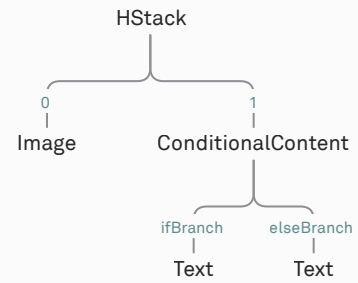


Image is "0", because it’s the first subview of the HStack. The Text in the non-nil branch of the if let statement is "1.ifBranch", because the ConditionalContent is the second subview of the HStack, and the Text is the first subview of the ConditionalContent.

> 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.

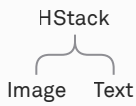
Now consider the two text views in the two branches of the if let statement. They have different identities, and therefore are considered two distinct views by SwiftUI. When the condition changes, the old text will be removed from the render tree, and a new text will be inserted. This has all kinds of 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

```
HStack {
  Image(systemName: "hand.wave")
  Text(greeting ?? "Hello")
}
```

View Tree



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

```
HStack {
   Image(systemName: "hand.wave")
   Text(greeting ?? "Hello")
      .id(greeting == nil)
}
```

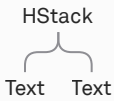
View Tree

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

HStack {
  let v = Text("Hello")
  v
  v
}
	

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:

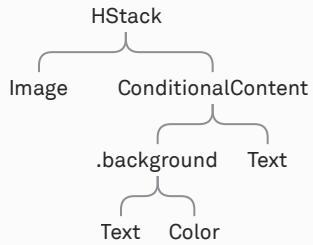
```
// Anti-pattern
extension View {
  @ViewBuilder
  func applyIf<V: View>(_ condition: Bool, transform: (Self) -> V) -> some View {
    if condition {
      transform(self)
    } else {
      self
    }
  }
}
```

The applyIf method can be used like this:

Code

```
HStack {
  Image(systemName: "hand.wave")
  Text("Hello")
    .applyIf(highlighted) {
      $0.background(.yellow)
    }
}
```

View Tree



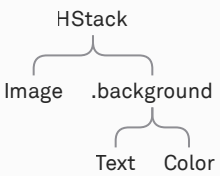
Looking at the resulting view tree, we can see that using the applyIf modifier has introduced a ConditionalContent with two subviews: an unmodified text, and a text with a background. This means that when the condition (highlighted) changes, the identity of the text onscreen will have changed as well.

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

```
HStack {
  Image(systemName: "hand.wave")
  Text("Hello")
    .background(highlighted ? .red : .clear)
}
```

View Tree



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 rerenders.

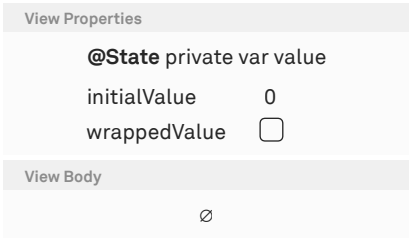
In addition to the `_value` property, we also added a computed `value` property, which makes the state easier to use: instead of having to write `_value.wrappedValue` each time we want to read or write, we can use `value`, and the computed property will forward that to `_value.wrappedValue` transparently. 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:

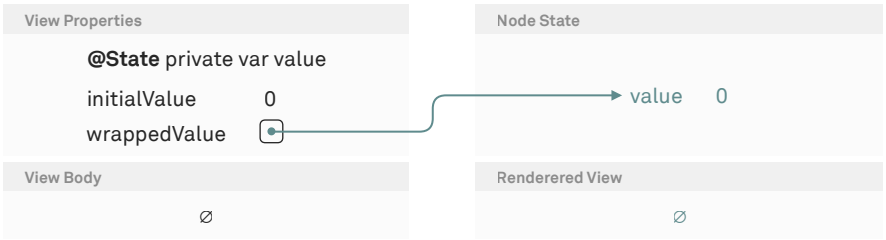
> To make the state diagrams throughout this chapter more readable, we have highlighted the changes compared to the diagram in the previous step using this color.

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.



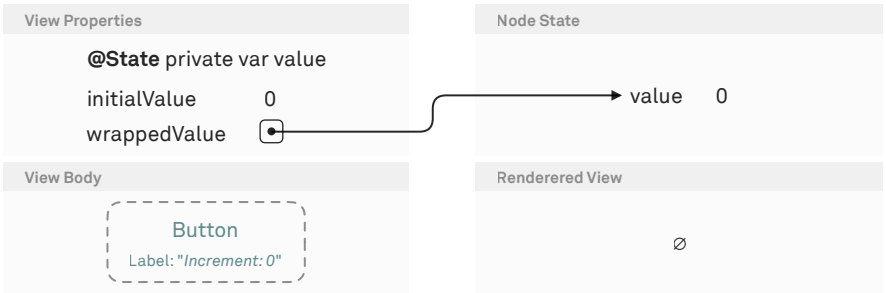
Counter View Counter Node in the Render Tree

Step 2. When SwiftUI creates the counter view's node in the render tree (on the right), it allocates memory for the view's state property. The system initializes the memory for the value property in the render node with the state property's initialValue of 0. The wrapped value of the state property now points to the memory in the render node.



Counter View Counter Node in the Render Tree

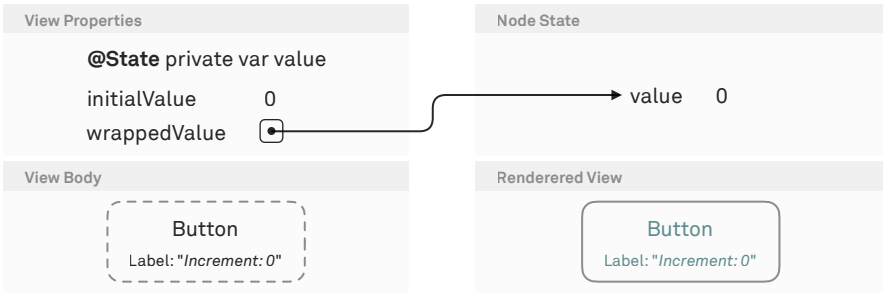
Step 3. The view's body gets executed and the Button view is created. Since the wrapped value of the state property now points to the memory in the render node, the value stored in the render node is used when constructing the button's title.



Counter View Counter Node in the Render Tree

Step 4. Using the counter view's body as a blueprint, the button view in the render

node is created.

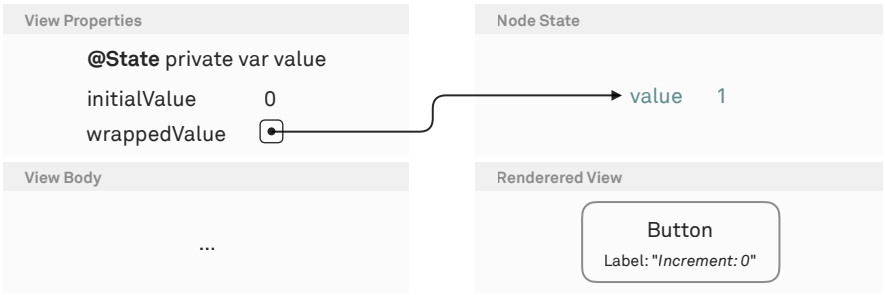


Counter View | Counter Node in the Render Tree

Because we used `value` in the body of `Counter` (for the label of the button), a dependency is added between the memory in the render tree and the body of the counter view.

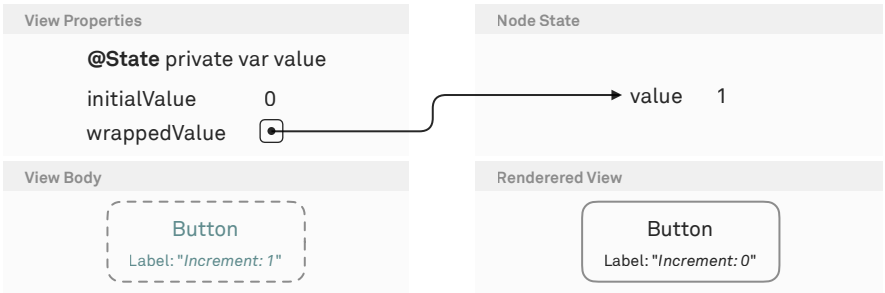
Now, let's consider the steps when the button is tapped:

Step 1. Because the value property — which is really `_value.wrappedValue` — is a pointer to the memory in the render tree, that memory will be incremented.

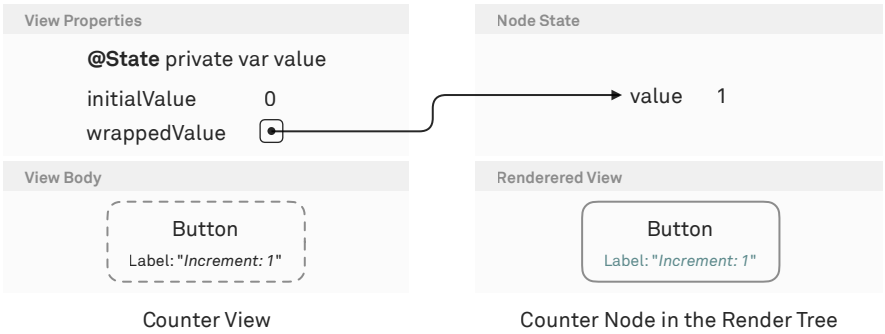


Counter View | Counter Node in the Render Tree

Step 2. Because the counter's body is dependent on that state memory, it's reexecuted, and a new button view will get constructed. Accessing the value property for the button's title will now return 1 (even though the initial value of the state property is still 0).



Step 3. Using the newly constructed view tree of the counter view as a blueprint, the button's title (in the render tree) changes to the new value.



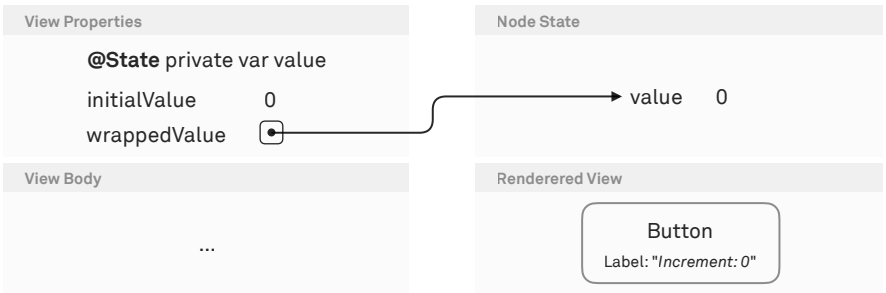
At the beginning of this section, we mentioned that `@State` is meant to be used for *private* view state, and that's why we annotated all our state properties with the `private` keyword. Although the latter isn't required, we think it's a good habit — and having peeked beneath the magic of the `@State` property wrapper, we can now make a better case for it: we saw in the code above that the initializer of `State` only takes an initial value for that state. Let's consider what would happen if we'd expose this via the view's initializer:

```
struct Counter: View {
  @State private var value: Int

  init(value: Int = 0) {
    _value = State(initialValue: value)
  }

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

Now we can pass the value in from the outside, but this only changes the *initial value* of the state property, since we don't have access to the state's current value in the view's initializer. Once the node for this view has been created in the render tree, passing in a different initial value will have no effect — or at least not the effect we might expect. All that's happening is that the initial value (not the actual value!) gets changed, which will only have an effect if the node is removed and reinserted into the render tree.

For example, here's what happens when we initially render the counter view with a value of 0 and then update it to a value of 5, which we pass in from the outside as in the code sample above:

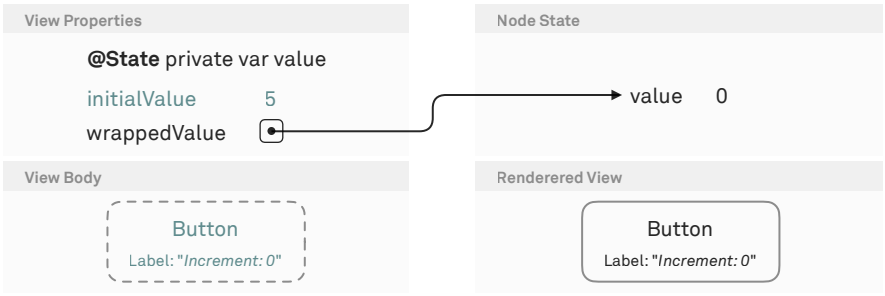
Step 1. We start out with the initially rendered counter view. The initial value is 0, and the wrapped value points to the state memory in the render node.



Counter View Counter Node in the Render Tree

Step 2. Now we pass in the value 5 to the counter view from the outside, which will construct a new counter view, but only the initial value has changed. The wrapped value still points to the state memory, which has the value 0, and so the label of the

button also displays 0.



Counter View | Counter Node in the Render Tree

If we wanted to have an initializer to pass a value in from the outside, it'd be clearer to at least change the initializer's label, like this:

```
init(initialValue: Int = 0) {
  _value = State(initialValue: initialValue)
}
```

Naming the parameter initialValue communicates the behavior more clearly, but in our experience, initializing state properties from the outside isn't really useful. Therefore, we recommend declaring all @State properties as private.

We might wonder why we can't just write the initializer in the following way, which the compiler happily accepts:

```
struct Counter: View {
  @State private var value = 0

  init(value: Int = 0) {
    self.value = value
  }
  ...
}
```

As we've seen before, the self.value = value statement actually translates to self._value.wrappedValue = value. There's nothing wrong with this statement from the compiler's point of view, but it still won't do what we'd expect. However, there's a valuable lesson in the reason why this doesn't work.

In the previous chapter, we talked about SwiftUI's concept of structural identity. In a nutshell, views have identity by nature of their position in the view tree. The reason that the assignment in the initializer above doesn't work is twofold: the state of a view is coupled to its identity, and at the time the initializer runs, the view doesn't yet

have identity. To better understand why SwiftUI *can't* know about the view's identity at this point, we can do a little thought experiment. Consider the following snippet:

```
struct ContentView: View {
  let counter = Counter()
  VStack {
    counter
    counter
  }
}
```

In this example, the initializer of `Counter` runs once when we create the struct and assign it to the local variable `counter`. However, at this point, we haven't placed the view in the view hierarchy, so the view can't have identity yet when the initializer runs, and therefore we can't access the view's state. Later on, when the counter's body is executed, the view has identity, and therefore, the state property references the correct state value in the render tree.

While the `@State` property wrapper was originally used almost exclusively for value types, this changed with the introduction of the `Observable` macro in iOS 17, as we'll see in the next section. However, on older platforms, `@State` still should only be used for value types, unless we know exactly why we're breaking this rule. We'll talk about observing objects with older versions of SwiftUI later in this chapter.

Observable Macro

The `Observable` macro is SwiftUI's mechanism for observing state in objects. It's part of the Observation framework, and it was introduced at WWDC23 as a replacement for the entire existing object-observation model (based on the Combine framework) that we've used up until this point.

The `Observable` macro does two things:

- → It adds conformance to the `Observable` *marker protocol*. This is an empty protocol used to tag a class at compile time but that doesn't have a runtime representation.
- → It changes the object's properties to track both read and write access.

The observation of an `Observable` object happens automatically just by accessing the object's properties, regardless of where it's stored, and without any special property wrappers. Therefore, we only use the `@State` property wrapper in combination with `Observable` objects to change how the objects' lifetimes are managed:

- → To couple the lifetime of an observable object to the lifetime of the view's render node (in other words, it's an object that's private to the view), we declare the property using the @State property wrapper.
- → To use an object with a lifetime independent of the view's render node (in other words, an object that we pass in from the outside), we just store it in a normal property.

In contrast to how SwiftUI handled the observation of objects in the past, the concepts of object lifetime and observation were separated in iOS 17. Before, we'd use @State to have SwiftUI manage the lifetime of a state value across view updates, and we'd use @StateObject to get the same lifetime management behavior, plus observation of the object. Since the observation part is no longer folded into the property wrappers, we're only left with @State to indicate that SwiftUI should manage the lifetime of a state object.

Here's how we'd write the counter example from above using a model object instead of a simple value:

```
@Observable final class Model {
  var value = 0
}

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

From an ownership perspective, using @State together with an Observable object works the same way as the @State example with a value type from the section above. SwiftUI allocates memory for the object in the render tree node and keeps the object alive as long as the node exists. The @State property wrapper's wrapped value points to this object.

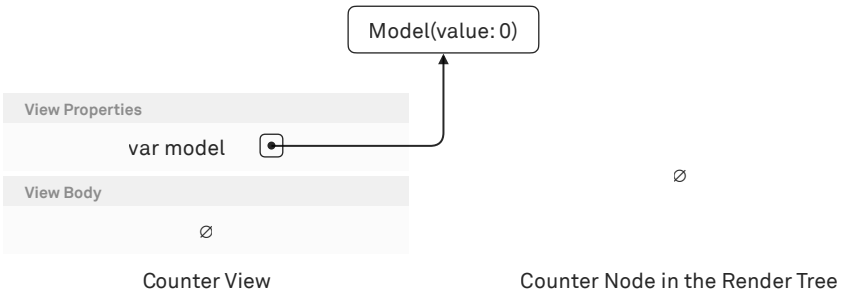
Let's first take a look at an example where we use an @Observable object without a property wrapper:

```
@Observable final class Model {
  var value = 0
  static let shared = Model()
}
```

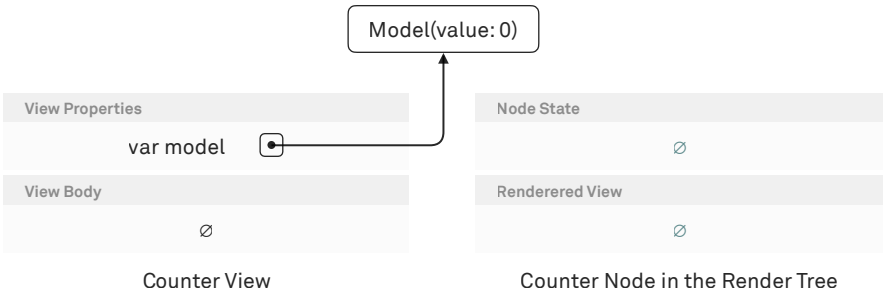
```
struct Counter: View {
  var model: Model
  var body: some View {
    Button("Increment: \(model.value)") {
      model.value += 1
    }
  }
}

struct ContentView: View {
  var body: some View {
    Counter(model: Model.shared)
  }
}
```

Now that the model property in the counter view is no longer declared as @State, we can pass it in from the outside, as we do in ContentView above. When the counter is first created, here's what happens:

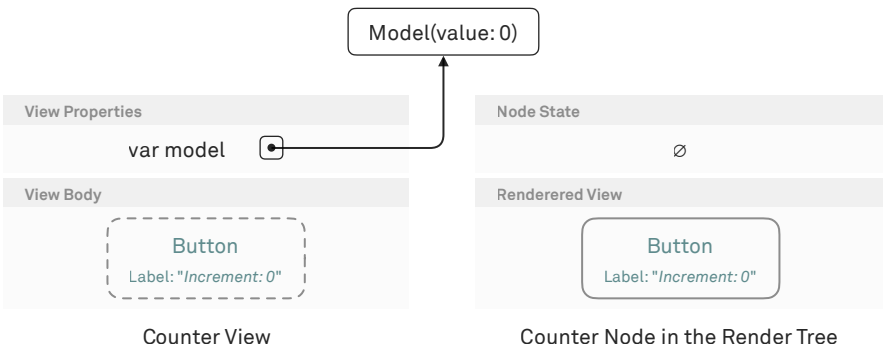
Step 1. The model property points to the model instance we passed to the counter view.



Step 2. The render node is created without any attached state, since the counter view only has plain properties.



Step 3. The body of the counter view is executed, and the button view is constructed. Inside the body, the button's title uses the external model object stored in the model property of the counter view struct. Then, the render node is updated to reflect the new view tree of the counter view.



Without @State, SwiftUI doesn't have any lifetime responsibilities with regard to this object, and observation happens automatically when the view's body is executed and the object's properties are being accessed. Therefore, the render node of the counter view doesn't need to maintain any state itself.

Of course, in this particular example, we could just write the following in the counter view instead of passing in the model from the outside:

```
struct Counter: View {
  var model = Model.shared
  ...
}
```

This does exactly the same thing, since we're always passing in the same Model.shared instance anyway. However, we could think of scenarios where we'd want to pass in different model objects to a view depending on some other state.

The new macro-based object-observation model not only introduces a more convenient syntax, but it also changes how the dependency between views and observable objects is formed: with the old property wrappers, SwiftUI would blindly subscribe to the objectWillChange publisher of any @StateObject or @ObservedObject declared in a view. With the new Observable macro, any property of an Observable object we access in the view's body will form a dependency to this view, no matter where the object is coming from.

This new model is a lot simpler. For example, accessing a global singleton (that's observable) in a view body will automatically form a dependency between the accessed property and the view, without us having to pass this singleton into the view using @ObservedObject. Likewise, observable objects can be nested in optionals, arrays, or other collections. Observation and view updates will continue to work as expected, because of the property-level tracking of dependencies in the view's body.

The new model is also more efficient. If you only use one property of an object in your view's body, changes to the other properties won't cause redraws of this view. Likewise, if you don't use a model object (for example, when it's only in one branch of your code), the model isn't observed at all. This can reduce unnecessary view updates and thus improve performance, whereas previously, we had to manually split up our model objects to get more fine-grained view updates.

State and Observable

We mostly see two common mistakes with regard to observing objects: using @State for objects that are passed in from the outside, and not using @State for objects private to the view. Phrased differently, one problem is to use @State when the lifetime of an object is managed somewhere outside of the view, while the other problem is not using @State when an object's lifetime isn't managed somewhere outside of the view.

Here's an example of the first problem, where we use an @State property for an object that's passed in from the outside:

```
struct Counter: View {
  @State var model: Model

  init(model: Model) {
    self.model = model
  }

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

As discussed in the section about @State, the problem here is that at the time the initializer runs, the view doesn't yet have identity. So, all this initializer does is change the initial value of the @State property, but it doesn't affect the state that's being used when the view is already onscreen.

The same applies if we only try to pass in a value from the outside and construct, for example, a view model object within the initializer:

```
struct Counter: View {
  @State var model: CounterViewModel

  init(value: Int) {
    self.model = CounterViewModel(value: value)
  }

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

Again, if the counter view is already onscreen and we pass a different value to the initializer, this won't change the title of the button. Only the initial value of the @State property will be changed to the new CounterViewModel instance, which will only be used the next time the view is inserted into the render tree.

As a counterexample, let's take a look at what happens if we remove the @State attribute in this view:

```
struct Counter: View {
  var model: CounterViewModel

  init(value: Int) {
    self.model = CounterViewModel(value: value)
  }
  ...
}
```

Now the counter view no longer maintains the lifetime of the model object itself. When we pass in a new value, the new view model will be constructed, and the view's body will use this new object to render itself. However, we've created another problem: this code works as long as the counter view isn't recreated due to some change further up in the view hierarchy. If it is recreated, it'll lose its state, because the counter view model will be reconstructed in the view's initializer.

We can solve this problem by passing in the view model from the outside or fetching it from some global model object, or something else along these lines. In any case, the lifetime of the view model has to be managed somewhere outside of the view if the model property in the view isn't declared with @State.

How the Observable Macro Works

The behavior of the new observation model seems very magical at first — the observation of object properties we access in a view's body just happens automatically. To accomplish this, SwiftUI leverages the new Swift macros to hide all the code necessary to make it work. However, the Observable macro is just a convenience to keep our model objects clean, and we can get the same functionality without using the macro. Let's take a look at the resulting code, step by step:

```
@Observable final class Model {
  var value = 0 {
    get {
      access(keyPath: \.value )
      return _value
    }
    set {
      withMutation(keyPath: \.value ) {
        _value  = newValue
      }
    }
  }

  @ObservationIgnored private var _value  = 0
  ...
}
```

The value property has been transformed from a stored property into a computed one, and an additional private stored property, _value, has been added. This private property is used as the backing store for the computed property. In the computed value property, the getter returns _value after recording the access to this property (by calling access with a key path). The setter mutates the private _value property within a withMutation closure, again specifying the key path of the property.

The two generated methods, access and withMutation, look like this:

```
@Observable final class Model {
  ...
  @ObservationIgnored private let _$observationRegistrar = ObservationRegistrar()

  internal nonisolated func access<Member>(keyPath: KeyPath<Model , Member>) {
    _$observationRegistrar.access(self, keyPath: keyPath)
  }

  internal nonisolated func withMutation<Member, T>(
    keyPath: KeyPath<Model , Member>,
    _ mutation: () throws -> T
  ) rethrows -> T {
    try _$observationRegistrar.withMutation(of: self, keyPath: keyPath, mutation)
  }
}
```

Both methods forward the call to a matching method on the object's observation registrar. This registrar is responsible for keeping stock of the observers interested in particular properties and notifying these observers when the properties change.

So how is the connection between an object's properties and SwiftUI views formed? There's a global function, withObservationTracking(_ apply:onChange:), which takes two closures. The first closure, apply, is executed immediately, and the observation system tracks which properties were accessed during apply. The second closure, onChange, is the observer that's called when any of the observable properties accessed in the apply closure change.

withObservationTracking stores the observation closure in a global variable and then executes apply. After apply finishes, the objects that were accessed add a dependency between the accessed properties and the observer closure. We could imagine that SwiftUI does something like the following when executing a view's body:

```
withObservationTracking {
  view.body
} onChange: {
  view.needsUpdate()
}
```

This way, any observable property we access in the view's body (directly or indirectly) goes through the object's observation registrar and forms a dependency between this specific property and the body of the view that's currently being executed.

ObservableObject Protocol

Prior to iOS 17, we'd have to use the ObservableObject protocol — together with either the @StateObject or the @ObservedObject property wrapper — to observe objects for state changes. If we need to target older platforms, this mechanism is still essential to understand, so in this section, we'll describe in detail how @StateObject and @ObservedObject work and when we should use either one.

StateObject

The @StateObject property wrapper works much in the same way as @State: we specify an initial value (an object in this case), which will be used as the starting point when the node in the render tree is created. From then on, SwiftUI will keep this object around across rerenders for the lifetime of the node in the render tree.

On top of what @State does, @StateObject observes the object for changes via the ObservableObject protocol, which is one of SwiftUI's very few interfaces to the model layer. Here's a version of the counter view from above that uses a model object instead of a simple value:

```
final class Model: ObservableObject {
  @Published var value = 0
}

struct Counter: View {
  @StateObject private var model = Model()
  var body: some View {
    Button("Increment: \(model.value)") {
      model.value += 1
    }
  }
}
```

Let's explore the various pieces involved to make this work. First, we can take a look behind the scenes of the @StateObject property wrapper. If we write the same code without the property wrapper syntax, it'd look like this:

```
struct Counter: View {
  private var _model = StateObject(wrappedValue: Model())
  private var model: Model { _model.wrappedValue }
```

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

Analogous to @State, the @StateObject property wrapper creates an underscored version of the property — _model in this example — that contains the StateObject value. Additionally, it synthesizes a computed property that forwards its getter to the StateObject's wrappedValue.

The requirement for StateObject's wrappedValue is that it conforms to the ObservableObject protocol. This protocol has only one requirement for conforming types: they have to implement an objectWillChange Combine publisher with a failure type of Never (this means that failures cannot happen). Typically, the output type of this publisher is Void , i.e. the events don't carry additional information. However, this isn't a formal requirement; the only requirement is that the objectWillChange publisher sends an event each time before a change *will* occur.

> In the early betas of SwiftUI, the requirement was called objectDidChange instead of objectWillChange. However, objectWillChange allows for better coalescing of updates.

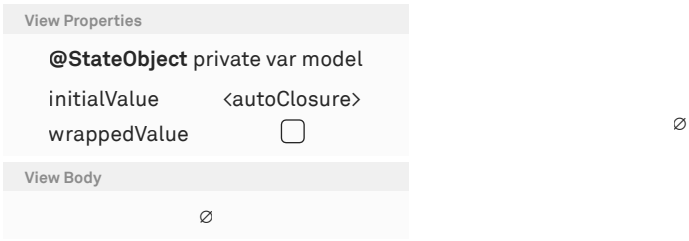
In the example above, we relied on the default implementation of the objectWillChange publisher that we get for free from the ObservableObject protocol, and we used the @Published property wrapper to send an event before our value property changes. Instead of using @Published, we could also write the following:

```
final class Model: ObservableObject {
  var value = 0 {
    willSet { objectWillChange.send() }
  }
}
```

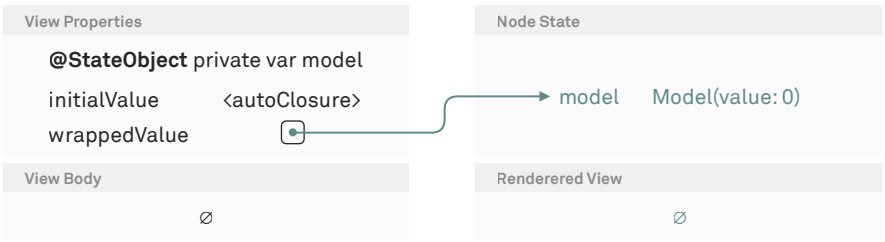
Using @Published is syntactic sugar for the code above. However, we can only use this property wrapper if we also use the default implementation of the objectWillChange publisher. If we implement our own publisher for some reason, @Published doesn't work.

When the view above is constructed, the following steps occur:

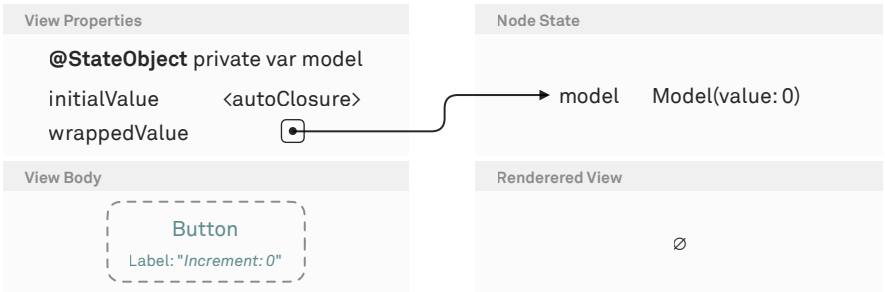
Step 1. The counter view is created, but the value of the state object isn't present yet, because the initial value is stored as an autoclosure.



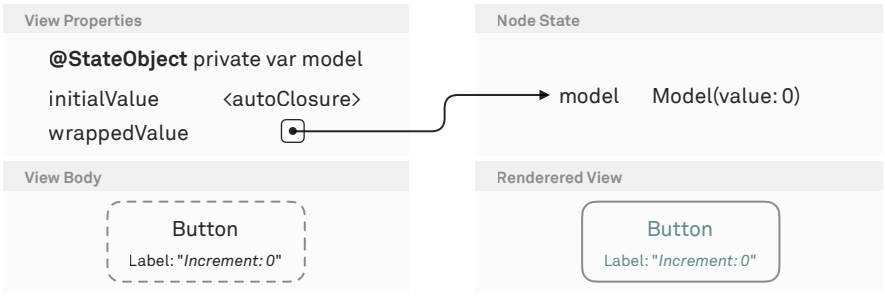
Step 2. The counter node in the render tree is created, and the memory for the state object is initialized with the value returned by the autoclosure. The wrapped value of the StateObject struct points to the model object stored in the render tree.



Step 3. The counter view's body is executed, the button view is created, and the current value of the model object from the render tree is used for the button's label.



Step 4. The node in the render tree is updated using the button view as a blueprint.



Counter View | Counter Node in the Render Tree

Just as with @State, @StateObject should be used for private view state; we shouldn't try to pass objects in from the outside or create objects in the view's initializer and assign them to the state object property. This doesn't work for the same reason we mentioned when talking about @State: when the view's initializer runs, the view doesn't yet have identity. As a rule of thumb, we should only use @StateObject when we can assign an initial value to the property on the line where we declare the property, like we did in the counter example:

```
struct Counter: View {
  @StateObject private var model = Model()
  ...
}
```

In the case of the simple model above, the @StateObject implementation is almost the same as the @State implementation, which used Observable. However, the observable implementation tracks changes at the property level, whereas the @StateObject tracks changes at the object level. Also, the @StateObject initializer takes an autoclosure, whereas the @State property will evaluate its initial value every time the view is initialized.

Observed Object

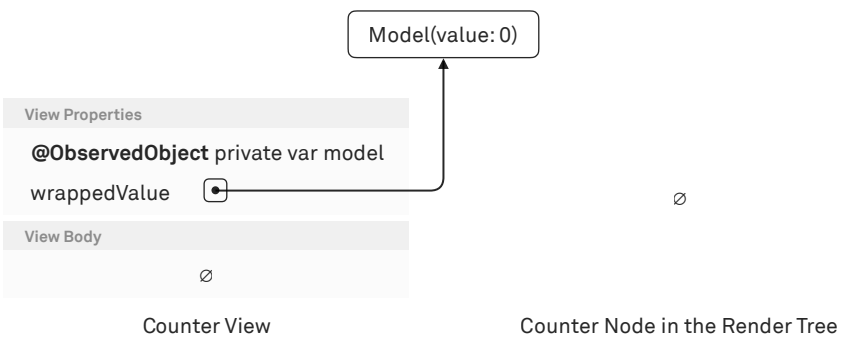
The @ObservedObject property wrapper is much simpler than @StateObject: it doesn't have the concept of an initial value, and it doesn't maintain the observed object across rerenders. All it does is subscribe to the object's objectWillChange publisher and rerender the view when this publisher emits an event. This makes @ObservedObject the only correct tool if we want to explicitly pass objects from the outside into a view (when targeting platforms before iOS 17). This is the equivalent of an Observable object within a regular property.

Here's how we could rewrite the counter example to use @ObservedObject:

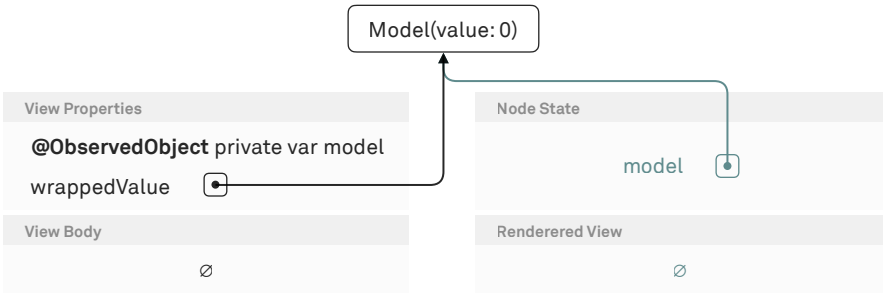
```
final class Model: ObservableObject {
  @Published var value = 0
  static let shared = Model()
}

struct Counter: View {
  @ObservedObject var model: Model
  var body: some View {
    Button("Increment: \(model.value)") {
      model.value += 1
    }
  }
}

struct ContentView: View {
  var body: some View {
    Counter(model: Model.shared)
  }
}
```

We've changed the @StateObject property in the counter view to an @ObservedObject property. Now we can pass in the model object from the outside, as we do in the ContentView. When the counter is first created, here's what happens:

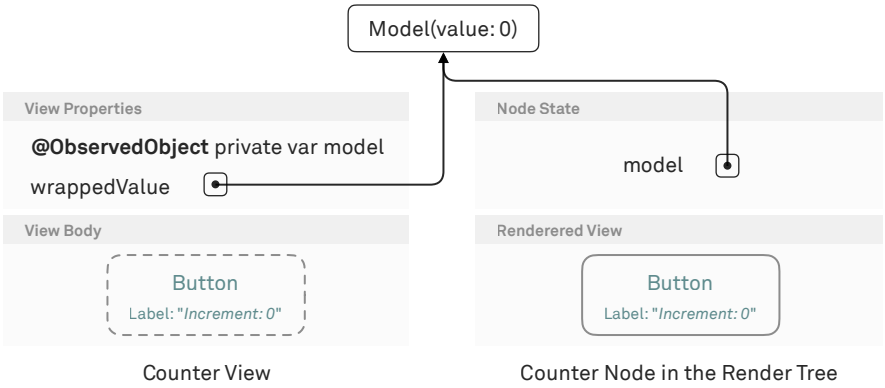
Step 1. The wrapped value of the observed object property points to the model instance we passed to the counter view.



Step 2. The render node is created. In contrast to @StateObject, the node only has a reference to the same external model object that we passed into the counter view.



Step 3. The body of the counter view is executed, and the button view is constructed. Inside the body, the button's title uses the observed object's reference to the external model object. Then the render node is updated to reflect the new view tree of the counter view.



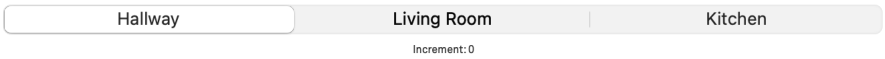
Of course, in this particular example, we could just write the following in the counter view:

```
struct Counter: View {
  @ObservedObject var model = Model.shared
  ...
}
```

This does exactly the same thing, since we're always passing in the same Model.shared instance from the outside anyway. However, we could extend this example to pass in different model objects to the counter based on some other state.

Here's a simple example of a people counter for different rooms:

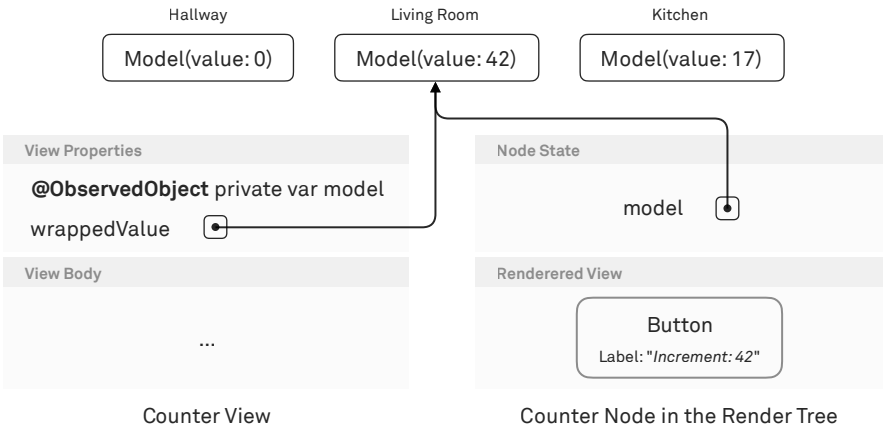
```
final class CounterModel: ObservableObject {
  @Published var value = 0
}

final class Model {
  static let shared = Model()
  var counters: [String: CounterModel] = [:]

  func counterModel(for room: String) -> CounterModel {
    if let m = counters[room] { return m }
    let m = CounterModel()
    counters[room] = m
    return m
  }
}

struct ContentView: View {
  @State private var selectedRoom = "Hallway"
  var body: some View {
    VStack {
      Picker("Room", selection: $selectedRoom) {
        ForEach(["Hallway", "Living Room", "Kitchen"], id: \.self) { value in
          Text(value).tag(value)
        }
      }
      .pickerStyle(.segmented)
      Counter(model: Model.shared.counterModel(for: selectedRoom))
    }
  }
}
```



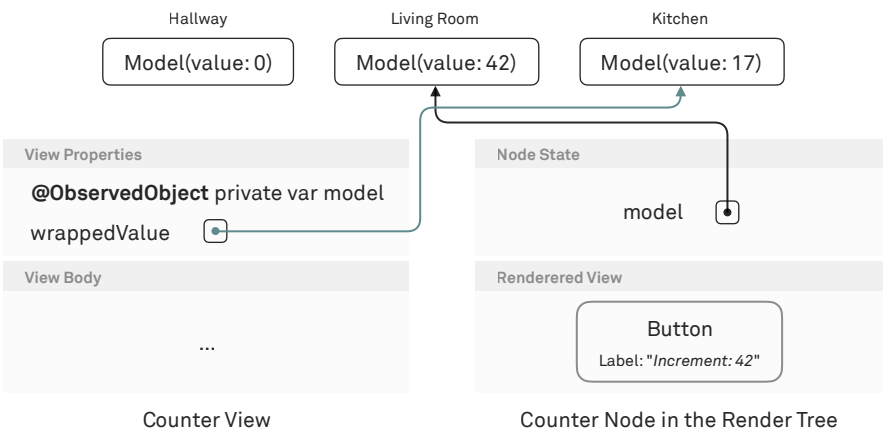
Increment: 0

We can use the picker to switch between rooms, retrieve the correct CounterModel instance from our shared model, and pass it on to the counter view. Our shared model manages the different CounterModel instances and keeps them around across view updates. The @ObservedObject's only job is to subscribe to the current instance (and to unsubscribe from the potential previous instance) to get notified of changes. Note that the node in the render tree for the counter view doesn't get recreated when we switch rooms; it just observes a different object.

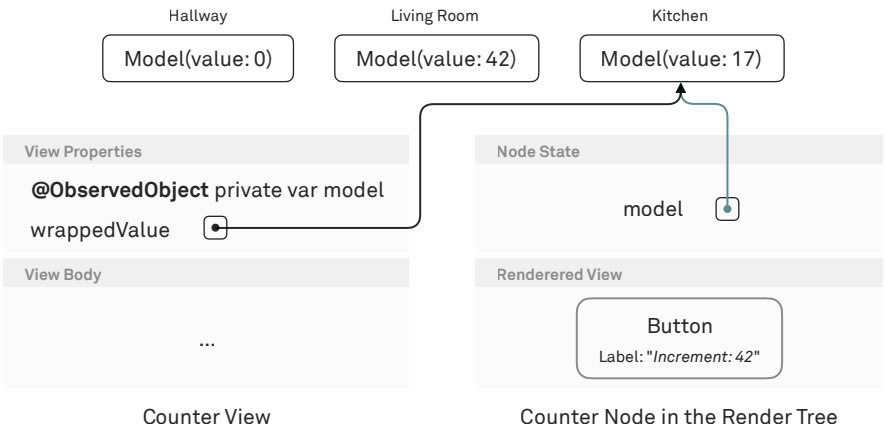
For example, let's imagine the counter is currently observing the "Living Room" object. When the Counter gets rendered, the view tree and the render tree both contain pointers to that object. Note that the render tree is responsible for observing the object, but the view tree is still just a blueprint:



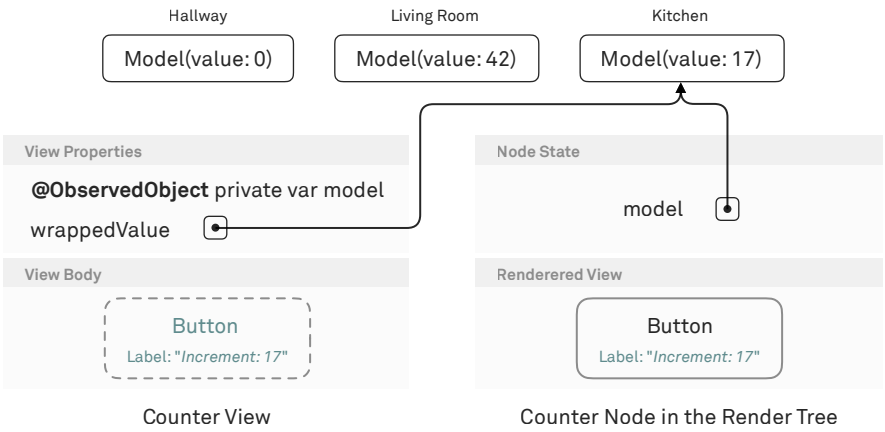
When the user selects the "Kitchen," the counter view's observed object now has a reference to a different model object.



As such, the render tree needs to be updated accordingly. First, the render tree changes its reference to the model it's observing, from the "Living Room" model to the now-current "Kitchen" model.



Finally, the body of the counter view is reexecuted to construct a new view tree, and the render tree is then updated accordingly.



Interestingly enough, when SwiftUI first came out, `@StateObject` didn't exist, and we only had `@ObservedObject` at our disposal. We could still write the same programs, but with the addition of `@StateObject`, we can replace some of the use cases of `@ObservedObject` with much less code.

Bindings

To keep an app's state consistent, it's important that each piece of state has a single *source of truth*. Otherwise, two values that are supposed to represent the same thing

might diverge. When writing components, it's often the case that we need read and write access to a value, but we don't need to know what the source of truth for this value actually is. Bindings are used exactly for this purpose, as we can see with many of SwiftUI's built-in components. For example, a Toggle takes a binding to a Bool, a TextField takes a binding to a String, and a Stepper takes a binding to a Strideable value.

Let's take the @State-based counter view from earlier and turn it into a component that takes a binding to an integer:

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

All we had to do was change the @State attribute on the value property to @Binding and remove the private keyword we previously used on the state property. After all, the whole point of switching to a binding is that it can be passed in from the outside.

To explore what bindings actually do, let's take a step back and think about how we'd implement external state without using @Binding. The counter view needs to read the current value, and it needs to be able to change the current value. We could achieve this by using a normal property for value and adding another property, setValue, that allows us to mutate the value:

```
struct Counter: View {
  var value: Int
  var setValue: (Int) -> ()
  var body: some View {
    Button("Increment: \(value)") { setValue(value + 1) }
  }
}
```

We'd use this version of the Counter view as follows:

```
struct ContentView: View {
  @State private var value = 0
  var body: some View {
    Counter(value: value, setValue: { value = $0 })
  }
}
```

Instead of writing the pair of properties, value and setValue, we can combine both using the Binding type:

```
struct Counter: View {
  var value: Binding<Int>
  var body: some View {
    Button("Increment: \(value.wrappedValue)") {
      value.wrappedValue += 1
    }
  }
}
```

The binding acts as a wrapper around a getter and a setter for the wrapped value. To avoid wrappedValue in our code, we can again use the pattern of an underscored Binding property and a non-underscored computed property that forwards the getter and setter to the binding's wrapped value:

```
struct Counter: View {
  var _value: Binding<Int>
  var value: Int {
    get { _value.wrappedValue }
    set { _value.wrappedValue = newValue }
  }
  init(value: Binding<Int>) {
    self._value = value
  }
  var body: some View {
    Button("Increment: \(value)") { value += 1 }
  }
}
```

The initializer's only job is to expose a nicer API without an underscored _value label for its parameter. We'd then use the Counter like this:

```
struct ContentView: View {
  @State private var value = 0
  var body: some View {
    Counter(value: Binding(get: { value }, set: { value = $0 }))
  }
}
```

Here, we're constructing the binding manually using the Binding(get:set:) initializer to demonstrate the nature of a binding as a wrapper around a getter and a setter.

Now that we've written all that code manually, we can simplify it again. The code around the _value and value properties in the counter view is exactly what the @Binding property wrapper generates for us. In the ContentView, we can replace the manual construction of the binding with the dollar sign syntax, $value, which constructs the binding for us:

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

struct ContentView: View {
  @State private var value = 0
  var body: some View {
    Counter(value: $value)
  }
}
```

To demystify the $value syntax, let's rewrite the ContentView once more without @State and $value:

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

Here we can see that $value is shorthand for _value.projectedValue. The dollar syntax isn't specific to SwiftUI; rather, it's a feature of Swift's property wrappers, and it's syntactic sugar for accessing the property wrapper's projectedValue.

We can use the dollar sign syntax not only on @State properties, but also on @StateObject properties or @ObservedObject properties. For example, we could store the value in the ContentView in a model object:

```
final class Model: ObservableObject {
  @Published var value = 0
}
```

```
struct ContentView: View {
  @StateObject var model = Model()
  var body: some View {
    Counter(value: $model.value)
  }
}
```

We could even create a computed property and bind to that. For example, here's a computed property on top of value that clamps the value from 0 to 10:

```
final class Model: ObservableObject {
  @Published var value = 0
  var clampedValue: Int {
    get { min(max(0, value), 10) }
    set { value = newValue }
  }
}

struct ContentView: View {
  @StateObject var model = Model()
  var body: some View {
    Counter(value: $model.clampedValue)
  }
}
```

As long as the property we're referencing after the $ has a getter and a setter, we can create a binding to it.

Observable and Bindable

SwiftUI's macro-based object-observation model poses a challenge with regard to bindings: as we saw above, the $ syntax used to construct bindings relies on the projectedValue of a property wrapper. Since we often no longer need to use a property wrapper when working with the @Observable macro, we can't construct a binding in the same way as when using, for example, the @ObservedObject property wrapper.

To solve this problem, the Bindable wrapper — which you can use either as a property wrapper or directly inline — was introduced. For example, it's used as a property wrapper here:

```
struct Counter: View {
  @Bindable var model: Model
```

```
  var body: some View {
    Stepper("\(model.value)", value: $model.value)
  }
}

struct ContentView: View {
  var model = Model.shared

  var body: some View {
    Counter(model: model)
  }
}
```

The memberwise initializer of Counter takes a plain Observable object and uses the Bindable(wrappedValue:) initializer internally to wrap that object in a Bindable value. We can also create bindable values inline by using the more succinct Bindable(_:) initializer without the label (both initializers do the same thing):

```
struct ContentView: View {
  var model = Model.shared
  var body: some View {
    Stepper("\(model.value)", value: Bindable(model).value)
  }
}
```

Similar to @StateObject and @ObservedObject, @Bindable's projected value allows us to access properties through the dynamic member lookup syntax. For example, the Counter view from above looks like this when we rewrite all the Bindable syntactic sugar:

```
struct Counter: View {
  var model: Model { _model.wrappedValue }
  var _model: Bindable<Model>

  init(model: Model) {
    _model = Bindable(wrappedValue: model)
  }

  var body: some View {
    Stepper("\(model.value)", value: _model.projectedValue[dynamicMember: \.value])
  }
}
```

In the counter view's body, we can see the dynamic member lookup syntax when we access the model's value property: model.value is translated into _model.projectedValue[dynamicMember: \.value].

View Updates and Performance

One of the main benefits of SwiftUI is that the UI is automatically kept in sync with our state. To avoid rerendering the entire view tree for any state update, SwiftUI establishes dependencies between particular state values (or objects) and the views that are dependent on them.

When the body of a view runs, SwiftUI knows which state properties were accessed while the body executed and creates a record of the dependency between the two. When a particular state value changes, SwiftUI knows which view is dependent on that value and only reexecutes the body of that view. Then it figures out which bodies in the view's subtree need to be reexecuted by comparing the view values from before and after the state change.

As we can see, SwiftUI goes to great lengths to ensure that only the absolutely necessary parts of the view tree are being rerendered when some state has changed. However, it's also important to write our code in a such a way that it doesn't counteract these efforts.

For example, if we put all our state in one large observable object (using the "old" @StateObject or @ObservedObject property wrappers), SwiftUI has to rerender all views that observe that object on any change, even if the particular change perhaps only affected a small subset of the views that observe the object. When performance becomes an issue, it can help to divide our state up into smaller units so that view updates can happen more granularly. The @Observed macro alleviates this problem to a large degree, because it shifts observation from the object level to the individual properties of the object.

Additionally, it's important to only pass the data that's really needed to subviews. For example, if we have a large struct that contains a lot of properties and we pass that struct along to a view that just needs one value out of this struct, that view will be rerendered each time anything in the struct changes. This can be avoided by being very specific about the properties on our views, which — as an added benefit — also makes it easier to show them in previews. To this end, it can make sense to break up larger views into multiple subviews that depend on distinct substates.

When we run into performance problems, there are several techniques to diagnose which view bodies are being executed. The first option is to insert a print statement in a view's body:

```
var body: some View {
  let _ = print("Executing <MyView> body")
  ...
}
```

The anonymous variable assignment `let _ = ...` is necessary because the view builder only accepts expressions of type `View`. We cannot simply call `print("...")`, because the return type of the print function is `Void`, which the view builder cannot handle. The `let _ = ...` syntax circumvents this problem.

> For a more visual indication of view rerenders, Peter Steinberger has a neat trick using random background colors.

To find out *why* a view's body was reexecuted, we can use the `Self._printChanges()` API in the view body like this:

```
var body: some View {
  let _ = Self._printChanges()
  ...
}
```

Again, we have to use the anonymous variable assignment workaround due to the `Void` return type of the function. This print statement will log the reason of the rerender to the console:

1. If the view was rerendered due to a state change, the name of the state property will be logged in its underscored form.
2. If the view value itself has changed, i.e. if the value of one of the view's properties has changed, `"@self"` will be logged.
3. If the identity of the view has changed, `"@identity"` will be logged. In practice, this usually means the view was freshly inserted into the render tree.

Finally, there's also a SwiftUI profiling template in Instruments, which we can use to diagnose which view bodies are being executed more often than we'd expect.

Which Property Wrapper for What Purpose?

We often see that the various property wrappers discussed in this chapter are overused in SwiftUI code. The first rule is to use regular properties without any property wrappers in views whenever possible. If we just want to pass a value to a

view and the view doesn't need write access to that value, a regular property is all we need.

When a view needs read and write access to a value (not an object) and it should own that value as local, private view state (which cannot be passed in from the outside), then we use `@State`. If, on the other hand, the view needs read and write access to a value but shouldn't own that value (and doesn't care where the value is actually stored), then we use `@Binding`.

If a view needs state in the form of an object and the view should own that object as local, private view state (which cannot be passed in from the outside), then we use `@StateObject` pre-iOS 17 or `@State` with an `@Observable` object post-iOS 17. However, if we need an object to be passed in from the outside, then we use `@ObservedObject` pre-iOS 17 and just a plain property post-iOS 17.

A good rule of thumb is that `@State` and `@StateObject` should only be used if a property can be initialized directly on the line where it's declared. If that doesn't work, we should probably use `@Binding`, `@ObservedObject`, or a plain property with an `@Observable` object.

Layout

4

SwiftUI's layout algorithm is straightforward: the parent view proposes a size to its subview, the subview determines its own size based on the proposal, and the subview reports that size back to the parent. The parent then places the subview within its own coordinate system. In essence, the goal of the layout process is to give each view a position and a size.

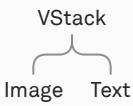
However, when we're struggling to understand why a view tree is rendering the way it does, understanding the algorithm's behavior isn't always straightforward. This chapter walks through all the details needed to mentally trace SwiftUI's layout steps and make sense of its behavior.

The first thing to keep in mind is that SwiftUI's layout algorithm proceeds top-down along the view tree. Therefore, it's important to understand the view tree produced by our view builder code (refer back to the chapter on view trees for more details on this). Let's take a look at an example to better understand how to apply the layout algorithm to a real view tree:

Code

```
VStack {
  Image(systemName: "globe")
  Text("Hello, World!")
}
```

View Tree



Since the VStack is the root view in this example, it'll receive the safe screen area as the proposed size. To determine its own size, the stack first recursively proposes sizes to its subviews. The image will report its size based on the size of the globe symbol, and the text will report its size based on the proposed size and the string it has to render (we'll go into the details of how exactly Image and Text determine their sizes later). The stack now places the two subviews beneath each other, inserting default spacing between the two. The stack computes its own size as the size of the union of its subviews' frames and reports that back to the window.

More formally, we can describe the layout algorithm like this:

1. The parent proposes a size to its subview.
2. The subview determines its own size based on the proposal, recursively starting at step 1 if it has subviews of its own.
3. The subview reports its size to the parent.
4. The parent places the subview.

> Note that the size reported by the subview in step 3 is the definitive size of the subview — the parent cannot alter this size unilaterally. It could go back to step 2 and make another proposal, but at the end of the day, the subview determines its size, taking the proposal into account as it sees fit.

While the API involved in this process isn't part of the public View protocol, we can imagine each view having a method like this:

```
extension View {
  func sizeThatFits(in: ProposedViewSize) -> CGSize {
    ...
  }
}
```

The ProposedViewSize type is public as of iOS 16 and macOS 13. It is used in the Layout protocol, which we'll discuss in the Advanced Layout chapter:

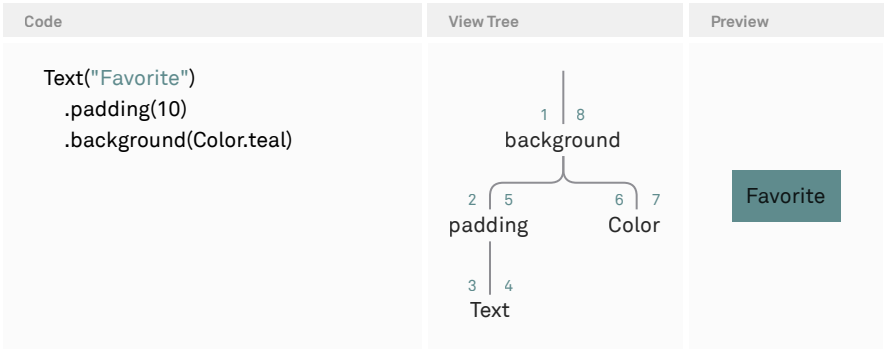
```
struct ProposedViewSize {
  var width, height: CGFloat?
}
```

The difference between ProposedViewSize and CGSize is that both components are optional in ProposedViewSize. Proposing nil for a component means that the view can become its *ideal size* in that dimension. The ideal size is different for each view, and we'll look at this in detail below.

In API terms, we can express the layout algorithm like this:

1. The parent calls sizeThatFits on its subview, passing on the proposed size.
2. The subview determines its own size based on the proposed size, perhaps calling sizeThatFits on its subviews if it has any. Some views completely disregard the proposed size (e.g. Image does so by default), while other views return the proposed size as their actual size (e.g. Rectangle). When a view returns the proposed size, we say that it *accepts* the proposed size.
3. The subview view reports its own size to its parent via the return value of the sizeThatFits method.
4. The parent places the subview according to its own alignment and the alignment guides of the subview (we'll go into detail about how alignment works later in this chapter).

Here's a more detailed example, including all the steps and specific sizing of the views involved:



For the sake of this example, we'll assume that we have a window with a safe area of 320×480.

1. The system proposes 320×480 to the background.
2. The background proposes the same 320×480 to its primary subview (the padding).
3. The padding subtracts 10 points on each edge, and it proposes 300×460 to the text.
4. The text reports its size as 51×17.
5. The padding adds 10 points on each edge, and it reports its size as 71×37.
6. The background proposes the size of the padded text (71×37) to the secondary subview (Color).
7. The color accepts and reports the proposed 71×37.
8. The background reports the size of its primary subview (71×37).

Equipped with this general understanding of how SwiftUI's layout system traverses the view tree, we'll next look at the specific layout behavior of many built-in SwiftUI views.

In general, when trying to understand how views are laid out, our favorite method is putting a border on the view. In addition, we could add an overlay with a geometry reader to also render the size of the view, but we've found the border already helps debug almost every layout problem.

While the general layout algorithm can be summarized in a sentence or two, understanding the specific layout behavior of all the built-in views is no easy task.

However, it's important so as to avoid too much guesswork with SwiftUI in the long run.

In this section, we'll go over many of SwiftUI's layout-relevant views and view modifiers and explain how they determine their own size, what size they're going to propose to their subviews, and what determines their ideal size. We'll start with leaf views (views that have no subviews), look at view modifiers next, and then talk about container views.

Leaf Views

Text

By default, `Text` views fit themselves into any proposed size. `Text` uses various strategies to make that work, in this order: it'll break the text into multiple lines (word wrapping), break up words (line wrapping), truncate, and finally clip the text. `Text` always reports back the exact size it needs to render the content, which is less than or equal to the proposed width, and at least the height of one line (with the exception of proposing 0×0). In other words, `Text` can become any width — from zero, to the size it needs to render the content in its entirety.

Here are some examples of how `Text("Hello, World!")` will render depending on the proposed size. The dashed rectangle represents the proposed size, and the solid rectangle represents the reported size:

Proposed: 25×50
Measured: 23×20

Proposed: 50×50
Measured: 44×10

Proposed: 100×50
Measured: 44×10

There are a handful of modifiers we can use on `Text` to change its behavior:

- → `.lineLimit(_ number:)` lets us specify the maximum number of lines that should be rendered, regardless of whether or not there's more vertical space proposed. Specifying `nil` means there's no line limit.
- → `.lineLimit(_ limit:reservesSpace:)` lets us specify the maximum number of lines that should be rendered, while giving us the option to always include the space for these lines in the reported size, regardless of whether or not they're empty.
- → `.truncationMode(_ mode:)` lets us specify where truncation should be applied.
- → `.minimumScaleFactor(_ factor:)` lets us specify how much `Text` is allowed to scale the font size down to make the text fit into the proposed size.

If we apply .fixedSize() to Text, it'll become its ideal size, because fixedSize proposes nil×nil to the text. The ideal size of the text is the size that's needed to render the content without wrapping and truncation. Here are some examples of how Text("Hello, World!").fixedSize() renders at different proposed sizes:

Proposed: 25×50
Measured: 44×10

Proposed: 50×50
Measured: 44×10

Proposed: 100×50
Measured: 44×10

As we can see by the solid rectangle (the reported size) rendering outside of the dotted one (the proposed size), the fixedSize modifier ensures its subview (the text) will always render at its ideal size, regardless of what was proposed. This is a good example of how a parent view in SwiftUI cannot expect its subview to respect the proposed size — in the final analysis, the subview determines its own size.

Shapes

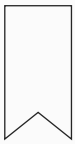
Most built-in shapes (Rectangle, RoundedRectangle, Capsule, and Ellipse) accept any proposed size from zero to infinity and fill the available space. Circle is an exception: it'll fit itself into any proposed size and report back the actual size of the circle. If we propose nil to a shape (i.e. if we wrap it in a .fixedSize), it takes on a default size of 10×10.

Along with the built-in shapes, we can also implement custom shapes. This is currently one of the few places in SwiftUI where we can write our own logic for computing the size based on the proposed size. As an example, we'll build a bookmark shape:

Code

```
struct Bookmark: Shape {
  func path(in rect: CGRect) -> Path {
    Path { p in
      p.move(to: rect[.topLeading])
      p.addLines([
        rect[.bottomLeading],
        rect[.init(x: 0.5, y: 0.8)],
        rect[.bottomTrailing],
        rect[.topTrailing],
        rect[.topLeading]
      ])
      p.closeSubpath()
    }
  }
}
```

Preview



To make writing this shape easier, we're using the following extension on CGRect, which takes a UnitPoint and returns the respective point within the rectangle:

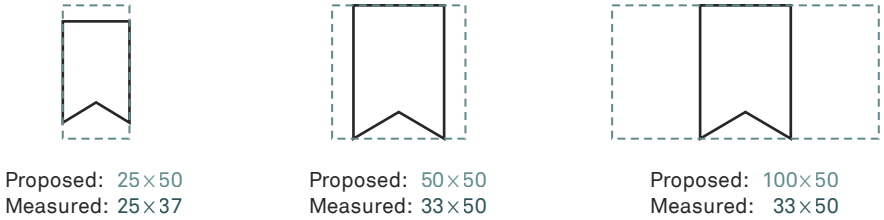
```
extension CGRect {
  subscript(_ point: UnitPoint) -> CGPoint {
    CGPoint(x: minX + width*point.x, y: minY + height*point.y)
  }
}
```

When this shape should always be rendered with an aspect ratio of 2×3, we can of course apply the .aspectRatio modifier at the site of usage, but any other consumers of the API might accidentally render our bookmark with a different aspect ratio. To ensure the bookmark shape always has an aspect ratio of 2×3, we can implement sizeThatFits. The default implementation for shapes returns the proposed size. We can override this with an implementation that hardcodes the aspect ratio to be 2×3:

```
func sizeThatFits(_ proposal: ProposedViewSize) -> CGSize {
  var result = proposal.replacingUnspecifiedDimensions()
  let ratio: CGFloat = 2/3
  let newWidth = ratio * result.height
  if newWidth <= result.width {
    result.width = newWidth
  } else {
    result.height = result.width / ratio
  }
```

```
    return result
}
```

Since the components of the proposed size might be nil, we have to start by calling replacingUnspecifiedDimensions, which gives us a CGSize. Then we compute the width and height of the shape so that its aspect ratio is 2x3 and it fits into the proposed size.



> We can also use this sizeThatFits method to better understand container views. If we add logging of the proposed and reported sizes to a shape and then use this shape within an HStack, for example, we'll see what the HStack is proposing to its subviews in the console. Note that sizeThatFits is part of the Shape protocol, and not part of the View protocol.

Colors

When using a color directly as a view, e.g. Color.red, it behaves just as Rectangle().fill(...) from a layout point of view.

However, there's a special case: if we put a color in a background that touches the non-safe area, the color will magically bleed into the non-safe area. Although this behavior doesn't affect the layout, we still wanted to mention it here since we'll all probably run into it at some point. If we want to prevent this, we can use the ignoresSafeAreaEdges parameter on .background, or use Rectangle().fill(...) instead of Color.

Image

By default, Image views report a static value: the size of the underlying image. Once we call .resizable() on an image, it makes the view completely flexible: the Image will then accept any proposed size, report it back, and squeeze the image into that size. In practice, virtually any resizable image will be combined with

an .aspectRatio(contentMode:) or .scaleToFit() modifier to prevent the image from being distorted. We'll talk about these modifiers in the next section.

The resizable modifier has two parameters. The first parameter lets us specify the capInsets, indicating the portion (at the edges) that shouldn't be resized. The second parameter lets us specify the resizing mode: the image can either stretch or tile.

Divider

When Divider is used outside of horizontal stacks, it accepts any proposed width and reports the height of the divider line. Within a horizontal stack, the divider accepts the proposed height and reports the width of the divider line. Proposing nil will result in a default size of 10 on the flexible axis, depending on the context.

Spacer

Outside of horizontal or vertical stacks, Spacer accepts any proposed size, from its minimum length to infinity. However, the behavior of Spacer changes when it's placed into a stack: within a vertical stack, Spacer accepts any height from its minimum length to infinity, but it reports a width of zero. Within a horizontal stack, it behaves the same way (with the axes swapped). The minimum length of a spacer is the length of the default padding, unless the length is specified using the minLength parameter in the spacer's initializer.

A popular use of spacers is for aligning views. For example, it's common to see code like this:

```
HStack {
  Spacer()
  Text("Some right-aligned text")
}
```

We recommend using a flexible frame with alignment instead (we'll discuss flexible frames in detail in the next section):

```
Text("Some right-aligned text")
  .frame(maxWidth: .infinity, alignment: .trailing)
```

We prefer this solution over the one with the HStack and the Spacer, because there's an edge case in the HStack solution that might not be immediately apparent: the Spacer has a default minimum length (equal to the default spacing). As a result, the text might start wrapping or truncating sooner than necessary, because the Spacer also occupies some of the proposed width of the HStack.

> One quick way to see the differences in resizing behavior is by putting both of the views above in a VStack inside the content view of a Mac application, which is resizable by default.

View Modifiers

View modifiers always wrap an existing view inside another layer: the modifier becomes the parent of the view it's applied to. While SwiftUI has the .modifier API to apply a value conforming to the ViewModifier protocol, SwiftUI's built-in modifiers are all exposed as extensions on View (which is a good practice for our own view modifiers as well). In this section, we'll describe the view modifiers that affect the layout.

Padding

The .padding modifier uses its padding value to modify the proposed size by subtracting the padding from the specified edges. This modified size is then proposed to the subview of .padding. When the subview reports back its own size, padding takes that size, adds the padding for the specified edges, and reports this extended size as its own.

There are two helpful variants of .padding:

- → .padding(_ inset:) lets us specify different padding values for different edges in one call using an EdgeInset value.
- → .padding(_ edges:_ length:) lets us specify one padding value for a set of edges (.horizontal and .vertical can be especially handy for the edges).

When we don't specify a concrete value — for example, by writing .padding() without any arguments — the default padding for the current platform is used.

Fixed Frames

The fixed frame modifier, .frame(width:height:alignment), has very simple layout behavior: it proposes exactly the specified size to its subview, and it reports exactly the specified size as its own size, independent of the reported size of its subview. In other words, a fixed frame is an invisible view of exactly the specified size we can use to propose a specific size to another view.

If we only specify the width or the height — by specifying nil for the other parameter or omitting it altogether — the other dimension will be untouched by the fixed frame. The proposed size of the frame will be forwarded to the subview in the unspecified dimension, and the frame will report the size of its subview as its own in that dimension.

With support for dynamic type and various screen sizes, using fixed frames in production code is actually quite rare. It's a good practice to avoid hardcoding magic numbers in fixed frames unless absolutely necessary, e.g. for certain design elements that intrinsically cannot or should not scale.

Flexible Frames

The API for flexible frames has a ton of parameters: we can specify not only the minimum, maximum, and ideal values for both the frame's width and height, but also an alignment. The behavior of flexible frames isn't very intuitive, but they're a useful tool, and it's important to learn their behavior. We'll first only consider the minimum and maximum boundary parameters, and we'll ignore the ideal width/height parameters for now.

Flexible frames apply the minimum and maximum boundaries twice: once for the size they propose to their subview, and once to determine their reported size.

The clamping "on the way in," i.e. to compute the size that's going to be proposed to the frame's subview, is the simpler one of the two. The flexible frame takes the proposed size and clamps it by whatever minimum and maximum parameters were specified. This clamped size is then proposed to the frame's subview.

Once the subviews' size has been computed, the flexible frame determines its own size based on the proposed size by applying the boundaries again. However, now a missing boundary value — e.g. if we only specify `minWidth`, but not `maxWidth` — will be substituted using the subview's reported size. Then the proposed size gets clamped by these boundaries and reported as the frame's size.

In practice, this means that if we only specify `minWidth`, the flexible frame will become at least `minWidth`, and at most, the width of its subview. If we only specify `maxWidth`, the flexible frame will become at least the width of its subview, and at most, `maxWidth`. The same applies for the height.

This behavior enables two common patterns in SwiftUI that are hard to parse at first, but become second nature once we've used them a few times.

The first common way of using flexible frames is like this:

```
Text("Hello, World!")
  .frame(maxWidth: .infinity)
  .background(.quaternary)
```

Proposed: 25×50
Measured: 25×20

Proposed: 50×50
Measured: 50×10

Proposed: 100×50
Measured: 100×10

The .frame(maxWidth: .infinity) pattern makes sure the flexible frame becomes at least as wide as proposed, or the width of its subview if that's wider than proposed. This is commonly used to create views that span the entire available width.

We could abstract this pattern in an extension on View:

```
extension View {
  func proposedWidthOrGreater() -> some View {
    frame(maxWidth: .infinity)
  }
}
```

Another common pattern is the following:

```
Text("Hello, World!")
  .frame(minWidth: 0, maxWidth: .infinity)
  .background(Color.teal)
```

This makes sure the frame always becomes exactly as wide as proposed, independent of the size of its subview. Again, we could write a helper function for this:

```
extension View {
  func acceptProposedWidth() -> some View {
    frame(minWidth: 0, maxWidth: .infinity)
  }
}
```

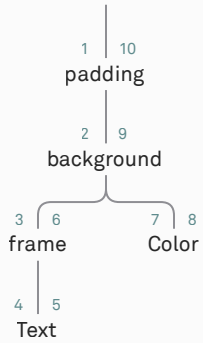
Both patterns can be applied to the height as well.

Using the previous example, let's consider what happens when the following view renders on a screen of 320×480:

Code

```
Text("Hello, World!")
  .frame(minWidth: 0, maxWidth: .infinity)
  .background(Color.teal)
  .padding(10)
```

View Tree



1. The system proposes 320×480 to the padding.
2. The padding proposes 300×460 to the background.
3. The background proposes the same 300×460 to its primary subview (the frame).
4. The frame proposes the same 300×460 to its subview (the text).
5. The text reports its size as 76×17.
6. The frame's width becomes max(0, min(.infinity, 300)) = 300. Note that the 0 and .infinity values are the arguments specified for the flexible frame.
7. The background proposes the size of the flexible frame (300×17) to the secondary subview (Color).
8. The color accepts and reports the proposed size.
9. The background reports the size of its primary subview (300×17).
10. The padding adds 10 points on each side, and it reports its size as 320×37.

The flexible frames API is the only one in SwiftUI to explicitly specify an ideal size, i.e. the size that will be adopted if nil is proposed for one or both dimensions. If the idealWidth or idealHeight parameters are specified, this size will be proposed to the frame's subview, and it'll also be reported as the frame's own size, regardless of the size of its subview.

Aspect Ratio

The aspectRatio modifier is most useful when working with completely flexible views. For example, we can use the following code to draw a rectangle with a 4/3 aspect ratio:

```
Color.secondary
  .aspectRatio(4/3, contentMode: .fit)
```

The aspectRatio modifier will compute a rectangle with an aspect ratio of 4/3 that fits into the proposed size and then propose that to its subview. To the parent view, it always reports the size of its subview, regardless of the proposed size or the specified aspect ratio.

To make this example more concrete, let's assume a proposed size of 200×200. The aspect ratio then computes the largest possible rectangle with a width-to-height ratio of 4/3 that fits into the proposed size, which is 200×150 in this case. It proposes this size to the Color, which accepts the proposal. The aspectRatio then reports 200×150 as its own size.

Let's look at what would happen if we used .fill instead of .fit as the contentMode:

```
Color.secondary
  .aspectRatio(4/3, contentMode: .fill)
```

We start again with the assumed proposed size of 200×200. Based on that, .aspectRatio computes the smallest rectangle that has a width-to-height ratio of 4/3 and still covers the entire proposed size, which is roughly 266.6×200. It proposes this size to the Color, which accepts the proposal. The .aspectRatio then returns the size of its subview, 266.6×200, as its own size.

A common use case of .aspectRatio is with images. As we discussed above, when we call .resizable on an image to make it flexible, it'll distort the image to fit itself into any size. Most of the time though, we want to maintain the aspect ratio of the underlying image while the image is scaled to fit or fill the available space. Without knowing the aspect ratio of the underlying image, we can write the following:

```
Image("header")
  .resizable()
  .aspectRatio(contentMode: .fit)
```

> The .scaleToFit and .scaleToFill modifiers are shorthand for .aspectRatio(contentMode: .fit) and .aspectRatio(contentMode: .fill), respectively.

Since we don't specify any specific aspect ratio, the .aspectRatio modifier will probe the subview for its ideal size (by proposing nil×nil) and calculate the aspect ratio based on that size. Assuming a proposed size of 200×200 and an ideal size of 100×30 for the image, here are the layout steps for this example. Note that the resizable modifier modifies the Image rather than adding another layer to the tree:

Code

```
Image("header")
  .resizable()
  .aspectRatio(contentMode: .fit)
```

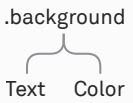
View Tree

1. The aspectRatio is proposed a size of 200×200.
2. The aspectRatio proposes nil×nil to the image.
3. The image reports its ideal size of 100×30.
4. The aspectRatio fits a rectangle with the aspect ratio of 100/30 into 200×200, which is 200×60, and proposes this size to the image.
5. The image reports its size as 200×60.
6. The aspectRatio reports the subview's size of 200×60 as its own size.

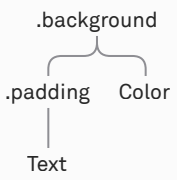
Note that the aspectRatio view doesn't necessarily end up with the aspect ratio we specify. While the modifier will propose a size of that aspect ratio to its subview, as always, it's up to the subview to determine its own size based on the proposal. If the subview isn't flexible enough to fit itself into the specified aspect ratio, the aspectRatio modifier itself won't become the specified ratio either, because it takes on the size of its subview as its own.

Overlay and Background

Overlay and background are among the most useful modifiers in SwiftUI. Layout-wise, they work exactly the same way. The only difference is that an overlay draws the secondary view on top of the primary view, whereas a background draws the secondary view behind the primary view. For example, if we want to draw a background behind some text, we can do it like so:

Code	View Tree	Preview
Text("Hello") .background(Color.teal)		

The way this works is rather elegant: the background first sizes the primary subview (in the example above, the Text) and then takes the size of the primary subview and proposes it to the secondary subview. This way, the teal color becomes exactly as large as the text. We could also change the text to include some padding:

Code	View Tree	Preview
Text("Hello") .padding(10) .background(Color.teal)		

We can now see that the teal background is exactly as large as the text, plus 10 points of padding on each side.

Note that background and overlay don't influence the layout of their primary subviews — the reported size of the overlay or background is always the reported size of the primary subview. For example, we could create a highlight modifier that draws an orange background behind a view that's slightly larger than the view itself:

```
extension View {
  func highlight(enabled: Bool = true) -> some View {
    background {
      if enabled {
        Color.orange
          .padding(-3)
      }
    }
  }
}
```

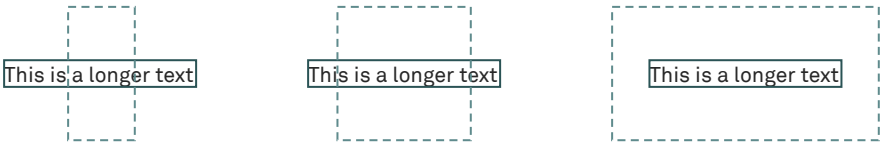
If we toggle the enabled flag, this won't cause any siblings of the view to move around: the background modifier always becomes the size of its primary subview, independent of the view in the background (the secondary subview).

Within an overlay or background, when the secondary view is a view list containing multiple views, these views are placed in an implicit ZStack.

Fixed Size

The fixedSize() modifier proposes nil for the width and height to its subview, regardless of its own proposed size. This way, the subview will become its ideal size. The fixedSize(horizontal:vertical:) overload allows us to make the subview become its ideal size in only one dimension (e.g. propose nil for the width but not the height).

One of the most common use cases for fixedSize is with Text views. As described above, Text will do anything to render itself inside the proposed size (e.g. truncation or word wrapping). One way to circumvent this is by proposing nil×nil to the text so that it becomes its ideal size. Note that, depending on the view tree, this might cause the text to draw out of bounds. For example:



Proposed: 25×50
Measured: 72×10

Proposed: 50×50
Measured: 72×10

Proposed: 100×50
Measured: 72×10

In the example above, we can see that the text draws out of the bounding rect of the frame modifier (which we use to propose certain sizes to the text). This is because, by default, views in SwiftUI aren't clipped. If we want clipping, we can enable it using the clipped modifier.

As we just saw, overlay and background have a useful property: they both propose the size of the primary subview to the secondary subview. However, we might not always want the second subview's size to be dependent on the first subview. For example, consider creating a custom badge modifier that should render a badge at the top trailing corner of a view:

Code

```
extension View {
  func badge<Badge: View>(@ViewBuilder contents: () -> Badge)
    -> some View {
    self.overlay(alignment: .topTrailing) {
      contents()
        .padding(3)
        .background {
          RoundedRectangle(cornerRadius: 5).fill(.teal)
        }
    }
  }
}
```

Preview

The above modifier works fine as long as the ideal size of the badge is smaller than the size of the view we're applying it to. For example, if we have a large button and we're trying to display a badge with just the text "1," it'll work fine. However, if we're applying a large badge to, say, a Text("Hi"), the badge won't become the size it needs to be.

Code

```
Text("Hi").badge {
  Text("2023").font(.caption)
}
```

Preview

To fix this, we can insert a fixedSize in the overlay. This will make sure the badge becomes its ideal size rather than the size of the view it's applied to:

```
extension View {
  func badge<Badge: View>(@ViewBuilder contents: () -> Badge) -> some View {
    self.overlay(alignment: .topTrailing) {
      contents()
        .padding(3)
        .background(RoundedRectangle(cornerRadius: 5))
        .fixedSize()
    }
  }
}
```

We'll improve the alignment of the badge later in this chapter when we look at alignment guides.

Container Views

HStack and VStack

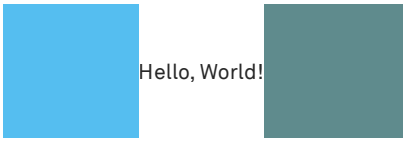
In our experience, HStacks and VStacks work pretty intuitively — until they don't. Then it can be rather complicated to figure out what's happening exactly, because the stack layout algorithm is quite complex.

> Horizontal and vertical stacks lay out their subviews in the same way, just with a different major axis. Therefore, we'll limit our discussion below to HStack.

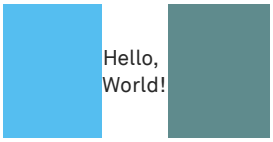
Consider the following stack:

```
HStack(spacing: 0) {
  Color.cyan
  Text("Hello, World!")
  Color.teal
}
```

If we propose a large-enough width, this renders as expected: the text "Hello, World!" with a blue color on the left, and a green color on the right. But as the width gets smaller, something interesting happens: the text wraps, even though there's enough space to show it in full (and still have some space left for the two colors).



Proposed: 150×50



Proposed: 100×50

The reason for this behavior is how the HStack algorithm divides up the available width among its subviews:

→ First, the stack determines the *flexibility* of its subviews. The two colors are infinitely flexible, so they become any size we propose. The Text, however, has an upper limit; it can become any size between zero and its ideal size, but never larger than that.

→ The stack then sorts the subviews according to flexibility, from least flexible to most flexible. It keeps track of all the remaining subviews and the available remaining width.

→ While there are remaining subviews, the stack proposes the remaining width, divided by the number of remaining subviews.

For simplicity, let's assume the text has an ideal width of 100 points. When we propose 180×180 to the `HStack`, it starts by proposing a size of 180/3 for the width to the least flexible subview — the text. The text will then wrap or truncate as needed. Let's assume it becomes 50×40 points (inserting a line break). The two rectangles then get proposed 130/2 and 65/1 points, respectively. Because of this algorithm, even though there's enough space for the text, it doesn't grow to display itself without line breaks.

There are a few ways to modify this behavior. Firstly, we could apply the `fixedSize()` modifier to the text. That way, the text ignores its proposal and always becomes the ideal size. This works well, but once the `HStack` is proposed less than the ideal width of the text, the text will still render at its full width — or, in other words, wider than proposed, and out of bounds.

Proposed: 40×40

An alternative is to give the text a *layout priority* using the `.layoutPriority` modifier. This will cause the `HStack` to first propose the full remaining width to the text and then use whatever remains after that to propose to the two colors. This way, once there isn't enough space for the text to render itself at its ideal size, it'll start word wrapping.

To better understand the stack layout's algorithm, here's an informal description:

1. Determine the flexibility of all subviews by proposing both `0×proposedHeight` and `.infinity×proposedHeight` to each subview (this process of proposing multiple sizes to a subview is also called *probing*). The flexibility is the difference of the two resulting widths.
2. Group all the subviews according to their layout priorities.
3. Start with the proposed width, and subtract the minimum widths of all subviews, as well as the spacing between them. The result is the `remainingWidth` (this might be `nil` if the proposed width was `nil`).
4. While there are remaining groups:

1. Pick the group with the highest layout priority.
2. Add the sum of all minimum widths of the subviews in this group back to the remainingWidth.
3. While there are remaining views in the group:
    1. Pick the view with the smallest flexibility.
    2. Propose (remainingWidth/numberOfRemainingViews)×proposedHeight.
    3. Subtract the view's reported width from remainingWidth.

While this algorithm isn't officially documented, it hasn't changed since we first described it in the first edition of this book, which was in 2020, and we consider it stable. It's possible to at least partially verify the behavior by placing custom shapes into the stack, which implement sizeThatFits and log the proposed size and reported size.

ZStack

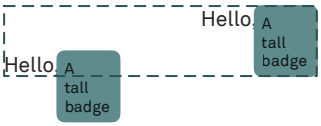
At first glance, ZStack might seem to do the same thing as an overlay or background, but it behaves differently. As we described above, overlay and background take on the size of the primary subview and discard the size of the secondary subview. When we use a ZStack, the union of the subviews' frames is used to compute the stack's own size.

Consider the following view:

```
ZStack {
  Color.teal
  Text("Hello, world")
}
```

When we use this as the root view for a new iOS application, the color stretches to the full size of the safe area. This is because the ZStack gets proposed the entire safe area, and it takes that proposal and proposes the same value to each of its subviews.

If we implemented the badge example from the overlay/background section using a ZStack instead of an overlay, the badge would influence the layout of the view it's applied to, because the size of the ZStack would include the badge. While a single view with a badge applied will look the same, its layout behavior relative to other views has changed.



The two views above are bottom aligned in an HStack. Both have a top-trailing aligned badge. The view on the left has an overlay for the badge, so the layout isn't influenced by the size of the badge view. Meanwhile, the view on the right uses a ZStack for the badge, and therefore, the bottom of this view is the bottom of the badge.

Scroll View

With scroll views, we have to distinguish between the layout behavior of the actual scroll view and the layout behavior of the scroll view's contents, which scrolls within the visible area of the scroll view.

The scroll view itself accepts the proposed size along the scroll axis, and it becomes the size of its content on the other axis. For example, if we place a vertical scroll view as the root view, it'll become the height of the entire safe area, and it'll take on the width of the scroll content.

Along the scroll axis, a scroll view essentially has unlimited space, and the content can grow as large as it wants to within that axis. Therefore, the scroll view proposes nil in the scroll axis (or axes), and it proposes the unmodified dimension for the other axis. Let's consider the following example:

```
ScrollView {
  Image("logo")
    .resizable()
    .aspectRatio(.fit)
  Text("This is a longer text")
}
```

By default, when we specify multiple views for the contents of the scroll view (as we do above), these subviews are placed inside an implicit VStack, regardless of the scroll direction. For the sake of this example, let's assume that the scroll view gets proposed a size of 320×480. It'll then propose 320×nil to both of its subviews and place them within the scroll container accordingly. The scroll view itself will always become the proposed height (480).

When we have a scroll view with text, sometimes the text might perform word wrapping and become a little less wide than the proposed width. If there are no other views in the scroll view that become the proposed width, this might mean that our scroll view becomes less wide than proposed. To fix this, we can add

a .frame(maxWidth: .infinity) to our text. The same applies to views other than Text that don't necessarily accept their proposed width.

When we put a shape into a scroll view, the result might be surprising: the built-in shapes in SwiftUI all have an ideal size of 10×10. This is due to the default argument of .replacingUnspecifiedDimensions(by:) on ProposedViewSize, which is 10×10. Since the scroll view proposed nil along the scroll axis, the shape will take on its ideal size of 10 points on that axis. If we want to change this, we can specify an explicit height using .frame(height:) or .frame(idealHeight:), or we could use .aspectRatio.

GeometryReader

Geometry readers are used to get access to the proposed size. We'll use them in the Advanced Layout chapter. A geometry reader always accepts the proposed size and reports that size to its view builder closure via a GeometryProxy, giving us access to the geometry reader's size. The geometry proxy also lets us access the current safe area insets and the frame of the view in a specific coordinate space, and it allows us to resolve anchors. For example, here's a geometry reader that shows the proposed size:

```
GeometryReader { proxy in
   Text(verbatim: "\(proxy.size)")
}
```

However, when reading Stack Overflow or perusing social media, GeometryReader is one of the views that always seems to cause trouble. Because a geometry reader always becomes the proposed size, if we want to — for example — measure the width of some Text view by putting a geometry reader around it, this will influence the layout around the text.

Once we have a good understanding of the SwiftUI layout system, we can avoid geometry readers in many places. That said, there are cases when we do need them. If we want a geometry reader to not influence the layout, here are two ways to achieve that:

- → When we wrap a completely flexible view inside a GeometryReader, it won't affect the layout. For example, when we have a scroll view, which becomes the proposed size anyway, we can wrap it using a geometry reader to access the proposed size.
- → When we put a geometry reader inside a background or overlay modifier, it won't influence the size of the primary view. Inside the background or overlay, we can then use the proxy to read out different values related to the view's geometry. This is useful to measure the size of a view, as the size of the primary subview will be proposed to the secondary subview (the geometry reader). We'll see more examples of this in the Advanced Layout chapter.

Geometry readers are special among other SwiftUI container views, in that they don't provide an alignment parameter, and they place their subviews top-leading by

default — whereas all other container views, with the exception of ScrollView, use center alignment by default.

List

List is SwiftUI's equivalent to UITableView or NSTableView. At the time of writing, List is still very limited and allows little configuration. The List view itself takes on the proposed size, and similarly to ScrollView, it proposes its own width, and nil for the height, to its subviews.

The sizing of the list's content in the vertical direction is complicated, because the list items are laid out lazily. Similar to a UITableView with non-fixed row heights, List estimates the entire height of the content based on the items that have already been laid out.

LazyHStack and LazyVStack

LazyHStack and LazyVStack behave in the same way, just with different major and minor axes. Lazy stacks share layout behavior with their non-lazy counterparts insofar as they become the size of the union of all subviews' frames. However, lazy stacks don't attempt to distribute the available space along the major axis among their subviews. For example, a LazyHStack just proposes nil×proposedHeight to its subviews, i.e. the subviews become their ideal width.

Computing the height of the lazy vertical stack is complicated by the fact that it creates its subviews lazily when it's embedded inside a scroll view (for the width, the lazy vertical stack accepts the proposed width). For example, consider the following view:

```
ScrollView {
  LazyVStack {
    ForEach(0..<100) { i in
      Text("View \(i)")
        .padding(.vertical)
        .onAppear {
          print("View \(i) appeared")
        }
    }
  }
}
```

As we scroll, more and more print messages start to appear, and views in the render tree get created as needed. Therefore, just like List, the LazyVStack needs to estimate its height based on the subviews that have already been laid out and update its own size as new subviews appear onscreen.

LazyVGrid and LazyHGrid

We'll focus on LazyVGrid in this section, as both LazyVGrid and LazyHGrid use the same underlying algorithm to calculate the size of their columns or rows.

The first step of laying out a LazyVGrid is to compute the widths of the columns based on the grid's proposed width. There are three column types: fixed columns, flexible columns, and adaptive columns. Fixed-width columns unconditionally become their specified width, flexible columns are flexible (but have lower and upper bounds on their width), and adaptive columns are really containers that host multiple subcolumns.

The grid starts by subtracting all the fixed-column widths and spacings from the proposed width. For the remaining columns, the algorithm proposes the remaining width, divided by the number of remaining columns, in order of the columns. Flexible columns clamp this proposal using their bounds (minimum and/or maximum width). Adaptive columns are special though: the grid tries to fit as many subcolumns inside an adaptive column as possible by taking the proposed column width and dividing it by the minimum width specified for the adaptive column. These subcolumns can then be stretched out to the specified maximum width to fill the remaining space.

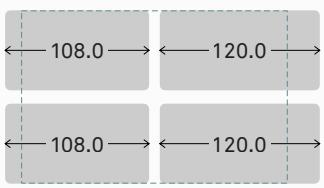
Now that the column widths have been computed, the grid computes its own width as the sum of the column widths, plus any spacing in between columns. For its height, the grid proposes columnWidth×nil to its subviews to compute the row heights and then computes its own height as the sum of the row heights plus spacing.

Contrary to HStack and VStack, grids go through the column layout algorithm twice: once during the layout pass, and then again during the render pass. During layout, the grid starts out with its proposed width. However, during the render pass, it starts out with the width that was calculated during the layout pass, and it divides that width among the columns again. This can have really surprising results. For example, consider this grid:

Code

```
LazyVGrid(columns: [
    GridItem(.flexible(minimum: 60)),
    GridItem(.flexible(minimum: 120))
], spacing: 10, content: {
    ...
})
.frame(width: 200)
.border(.teal, ...)
```

Preview



Not only does the grid render out of bounds — although the columns' minimums would happily fit into the available width of 200 points — but it also renders off-center of its enclosing 200-point-wide frame. Let's go through the steps of the grid's layout algorithm to understand what's happening.

We start at a remaining width of 200 points, minus 10 points of spacing, which gives us 190 points. For the first column, we calculate the width as 190 divided by 2 remaining columns, which equals 95 points. Since the first column has a minimum width of 60 points, the width of 95 points isn't affected by the clamping, so the remaining width stands at 95 points. The second column becomes 95 points clamped to its minimum of 120 points, i.e. 120 points wide.

However, that's not what we see in the rendering of this grid: the first column renders 108 points wide, and the second one renders 120 points wide, whereas we calculated 95 points and 120 points. That's where the second layout pass comes in.

The overall width of the grid as calculated by the first pass is 95 + 10 + 120 = 225 points. The fixed frame with a width of 200 points around the grid centers the grid, shifting it (225 − 200) / 2 = 12.5 points to the left. When it's time for the grid to render itself, it goes through the column layout again, but this time starting out with a remaining width of 225 points.

The second pass starts with 225 points minus 10 points spacing, or 215 points. The first column becomes 215 points divided by 2 remaining columns, equaling 108 points (rounded). The remaining width is now 215 minus 108, giving us 107 points. The second column becomes 107 points clamped to its minimum of 120 points. This is exactly what we see in the example above.

In addition to the surprising column widths, we can now also explain why the grid renders off-center in the fixed frame: since the frame around the grid has calculated the grid's origin based on the original width of 225 points, but the grid now renders itself with an overall width of 108 + 10 + 120 = 238 points, the grid appears out of center by about 7 points.

For more grid layout examples, take a look at this blog post.

Grid

`Grid` was introduced with iOS 16/macOS 13 and works similarly to `HStack` and `VStack`, but in two dimensions. The grid's subviews are sorted by flexibility, which can be either two-dimensional flexibility or one-dimensional flexibility (depending on the proposed size), and the available space is distributed among the subviews in two dimensions.

At the time of writing, the grid's layout behavior isn't documented. Before iOS 17, it had some bugs. In iOS 17 and up, the behavior is different but still has bugs. Because of this, we won't go into more detail here.

ViewThatFits

When we want to display different views depending on the proposed size, we can use `ViewThatFits`. It takes a number of subviews, and it displays the first subview that fits. It does so by proposing `nil` to figure out the ideal size for each subview, and it displays the first subviews (in the order the subviews appear in the code) where the ideal size fits within the proposed size. If none of the subviews fit, it picks the last subview.

Rendering Modifiers

SwiftUI has a bunch of view modifiers that influence how a view is rendered, but that don't influence the layout itself — for example, `offset`, `rotationEffect`, `scaleEffect`, etc. We can think of these modifiers as performing something like a `CGContext.translate` to modify where a view is drawn. However, from the layout system's perspective, the view is still situated in its original position.

Alignment

SwiftUI uses alignment to position views relative to each other. By default, almost all views center their subviews. For example, if we create an iOS app with just a text or a rectangle as the only view, that view will get centered both horizontally and vertically within the safe area.

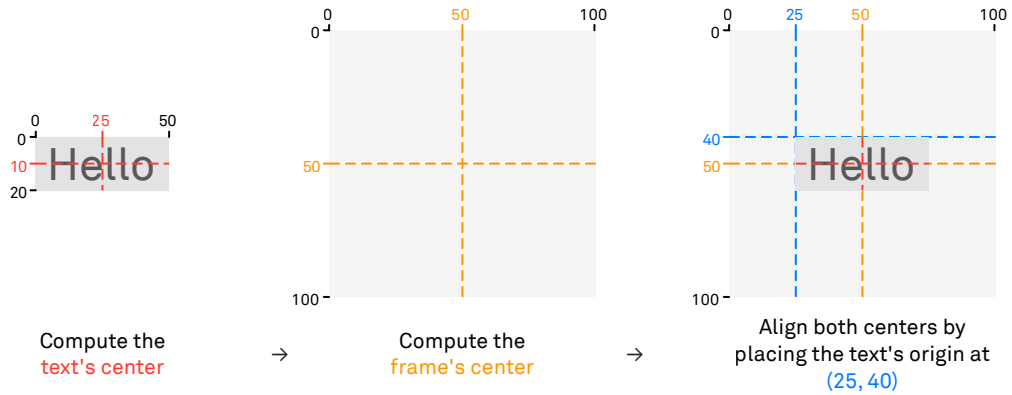
To better understand how alignment works, let's consider the following example:

```
Text("Hello")
  .frame(width: 100, height: 100)
```

The `frame` modifier has an `alignment` parameter with a default value of `.center`. As we discussed in the previous section, a fixed frame always becomes the exact dimensions we specify — 100×100 points in this case. Within that 100×100 points, the text will be centered. For the sake of simplicity, imagine the reported size of the text as 50×20 points. When placing its subview, the frame modifier takes the following steps:

1. It asks its subview for its horizontal center. The subview is 50 points wide and reports its horizontal center as 25 (in local view coordinates of the subview). This value is called the *horizontal center alignment guide* of the subview.
2. It asks its subview for its vertical center. The subview is 20 points tall and reports its vertical center as 10. This is the *vertical center alignment guide* of the subview.
3. The frame computes its own horizontal and vertical center as (50, 50).

4. Now the frame can center the subview by taking the difference between the two centers (50-25, 50-10 and placing the origin of the subview at that point (25, 40) in the frame's coordinate space.



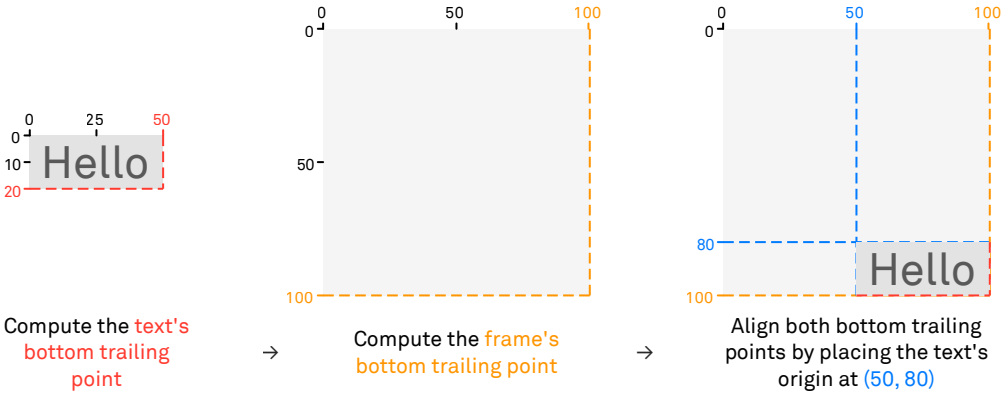
The most important point to understanding SwiftUI's alignment system is that alignment is always determined in "negotiation" between the parent and the subview. The parent view doesn't place the subview view on its own; rather, it consults the subview about the relevant alignment guides and then positions the subview relative to its own size or other subviews.

Here's the same example with a different, .bottomTrailing alignment:

```
Text("Hello")
  .frame(width: 100, height: 100, alignment: .bottomTrailing)
```

The algorithm is exactly the same as for the center alignment:

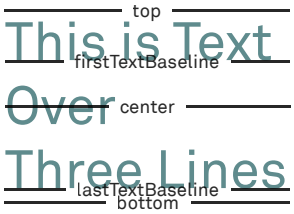
1. It asks the subview for its trailing alignment guide. The subview responds with 50, because that's the position of the trailing edge in the subview's local view coordinates.
2. It asks the subview for its bottom alignment guide. The subview responds with 20.
3. The frame computes its own trailing (100) and bottom (100) alignment guides.
4. The frame places the origin of the subview at (50, 80). This is again computed by taking the difference (100-50, 100-20).



For a view modifier like frame, alignment works in two directions. The frame's alignment parameter is of type Alignment, which is a composite struct that combines HorizontalAlignment and VerticalAlignment. Other types that align in two dimensions include .overlay, .background, and ZStack. In contrast, a VStack only has horizontal alignment, and an HStack only has vertical alignment. Note that the Alignment type isn't the alignment guide. Instead, it's what determines *which* alignment guide to use.

The HorizontalAlignment and VerticalAlignment structs have static constants for the built-in alignment guides: .leading, .center, and .trailing in the horizontal direction, and .top, .center, .bottom, .firstTextBaseline, and .lastTextBaseline in the vertical direction. The composite Alignment struct combines these into constants like .topLeading or .bottomTrailing.

The HorizontalAlignment and VerticalAlignment types all have a method to compute the default value given the view dimensions. Therefore, each view automatically defines all the built-in alignment guides. For example, here are the most important vertical alignment guides for a three-line text view:



When computing the first text baseline and last text baseline for views that don't contain any text, the height of the view is used (similar to .bottom).

Next, let's take a look at a simple ZStack example with multiple subviews. The ZStack (and all other container views besides frames) has to consult all of its subviews for their alignment guides and align them relative to each other. This is a more complicated process compared to the frame example above, because the ZStack first has to compute its own size (based on the sizes of its subviews and their alignment guides) before it can then use the alignment to place the subviews:

Code

```
ZStack {
  Rectangle()
    .fill(.teal)
    .frame(width: 50, height: 50)
  Text("Hello, World!")
}
```

Preview

Since the ZStack has a default value of .center for alignment, it aligns the centers of its two subviews in the following steps:

1. Determine the ZStack's own size:
    1. Ask the first subview (the frame with the blue rectangle) for its size and center alignment guides. It responds with a size of 50×50 and an alignment point of (25, 25).
    2. Ask the second subview (the text, which we'll assume has a size of 100×20) for its size and center alignment guides. It responds with a size of 100×20 and an alignment point of (50, 10).
    3. Compute the text's origin relative to the rectangle by subtracting the two alignment points from each other: (25, 25) - (50, 10) = (-25, 15).
    4. Determine the frames of each subview: the frame is the combination of the subview's origin and size.
    5. Compute the union of the two subview frames, which has an origin of (-25, 0) and a size of 100x50. The size of that rectangle is the ZStack's own size.
2. Place the ZStack's subviews:
    1. Based on the size computed in step 1, the ZStack computes its own center as (50, 25).
    2. Compute the rectangle's origin by subtracting its alignment point from the ZStack's center: (50, 25) - (25, 25) = (25, 0).

3. Compute the text's origin by subtracting its alignment point from the ZStack's center: (50, 25) - (50, 10) = (0, 15).

In more general terms, here's how every container view aligns its subviews:

1. Determine its own size:
    1. Determine the sizes of its subviews. The specifics depend on the particular view type, which we discussed in the first part of this chapter.
    2. Ask the subviews for their alignment guides for the container's alignment.
    3. Compute the origins for the subview's frames, using any particular subview as a reference.
    4. Compute the union of the subview's frames. The size of that rectangle is the container's size.
2. Place the subviews:
    1. Compute the alignment guides of the container itself based on its size.
    2. Compute each subview's origin by subtracting its alignment guides from the container's alignment guides.

The first step can be partially or entirely omitted, depending on the container view we're dealing with. For example, a fixed frame with both dimensions specified already knows its size. An overlay's size is only based on its primary subview, so alignment doesn't play a role in determining the overlay's size.

Modifying Alignment Guides

Using the built-in alignment guides, we can only align multiple views using the same alignment for each view. For example, we can align the top edge of one view to the top edge of another view. Or we can align the bottom trailing corner of one view to the bottom trailing corner of another view. However, we cannot align e.g. the center of one view to the top trailing corner of another view. Luckily, we can override built-in (implicit) alignment guides and even create our own alignments.

SwiftUI lets us change the implicit alignment guide for a view by providing an *explicit* alignment guide for a certain alignment. For example, we can override how the first text baseline is computed for an image:

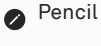
```
let image = Image(systemName: "pencil.circle.fill")
  .alignmentGuide(.firstTextBaseline, computeValue: { dimension in
    dimension.height/2
  })
```

By itself, providing an explicit alignment guide doesn't do anything. Only when this alignment guide is *used* for placement will this have an effect. For example:

Code

```
HStack(alignment: .firstTextBaseline) {
  image
  Text("Pencil")
}
```

Preview



However, if we change the stack alignment to .center, our custom alignment guide doesn't affect the alignment.

The parameter passed to the computeValue closure of the .alignmentGuide API is of type ViewDimensions. This is similar to a CGSize in that it has a width and height, but it also allows us to access the underlying view's alignment guides. For example, we could've written our image example in this way as well:

```
let image = Image(systemName: "pencil.circle.fill")
  .alignmentGuide(.firstTextBaseline, computeValue: { dim in
    $0[VerticalAlignment.center]
  })
```

Modifying alignment guides can also be very useful when we want to align views using two different alignment guides. For example, when we want to overlay the center of a badge view on the top trailing corner of another view, we can use the code below:

```
extension View {
  func badge<B: View>(@ViewBuilder _ badge: () -> B) -> some View {
    overlay(alignment: .topTrailing) {
      badge()
        .alignmentGuide(.top) { $0.height/2 }
        .alignmentGuide(.trailing) { $0.width/2 }
    }
  }
}
```

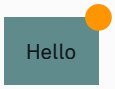
This modifies the .topTrailing alignment guides to be at the visual center of the badge. Note that the alignment specified on the overlay (.topTrailing) matches the explicit alignment guides specified on the badge (.top and .trailing).

The above modifier can be used like so:

Code

```
Text("Hello")
  .padding()
  .background(.teal)
  .badge {
    Circle()
      .fill(Color.orange)
      .frame(width: 20, height: 20)
  }
```

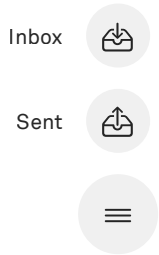
Preview



Custom Alignment Identifiers

So far, we used the built-in implicit alignments, and we used the .alignmentGuide API to specify explicit alignment guides for the built-in alignments. We can go one step further though: we can define completely custom alignments.

Let's consider a layout like this:



The larger menu button at the bottom should be horizontally centered to the circles of the individual menu items. Here's the rough structure of this menu:

```
VStack {
  HStack {
    Text("Inbox")
    CircleButton(symbol: "tray.and.arrow.down")
      .frame(width: 30, height: 30)
  }
  HStack {
    Text("Sent")
    CircleButton(symbol: "tray.and.arrow.up")
      .frame(width: 30, height: 30)
  }
```

```
    CircleButton(symbol: "line.3.horizontal")
      .frame(width: 40, height: 40)
}
```

The problem here is the horizontal alignment of the VStack: none of the built-in alignments do what we want. We could try to specify .trailing alignment on the VStack and then specify an explicit alignment guide for .trailing on both items' CircleButton. However, this doesn't work either. When the VStack places its subviews, it asks each subview for its .trailing alignment guide. Since HStack has its own .trailing alignment guide, the explicit alignment guide on CircleButton will never be used.

This is where custom alignments come in, because they can propagate up through multiple container views. The first thing we need for a custom alignment is a type conforming to the AlignmentID protocol:

```
struct MenuAlignment: AlignmentID {
  static func defaultValue(in context: ViewDimensions) -> CGFloat {
    context.width/2
  }
}
```

The only requirement of the AlignmentID protocol is the static defaultValue(in:) method. Here, we choose the default value for this particular alignment, and this default value is used unless we specify an explicit value later on. For our purposes, we specify the view's horizontal center as the default value.

Now we can define a static constant for our custom alignment on the HorizontalAlignment struct, just as SwiftUI does for the built-in alignments:

```
extension HorizontalAlignment {
  static let menu = HorizontalAlignment(MenuAlignment.self)
}
```

The .menu alignment is now identified by the type of our MenuAlignment struct. Now we can use it in the menu VStack and specify explicit alignment guides for .menu on the CircleButtons:

```
VStack(alignment: .menu) {
  HStack {
    Text("Inbox")
    CircleButton(symbol: "tray.and.arrow.down")
      .frame(width: 30, height: 30)
      .alignmentGuide(.menu) { $0.width/2 }
  }
```

```
    HStack {
      Text("Sent")
      CircleButton(symbol: "tray.and.arrow.up")
        .frame(width: 30, height: 30)
        .alignmentGuide(.menu) { $0.width/2 }
    }
    CircleButton(symbol: "line.3.horizontal")
      .frame(width: 40, height: 40)
}
```

We don't specify an explicit value for the .menu alignment on the menu button itself, because the default value of .menu is already the center, so that works for our purposes.

When the VStack now asks its subviews for the .menu alignment guide, the HStacks will consult their subviews for an explicit alignment guide value before falling back to the default value. This means that the HStack will return the explicit alignment guide we specified on the CircleButton when asked for its .menu alignment guide. In other words, even though we use the default values for our explicit alignment guide, the subview's explicit alignment guide is used instead of the stack's implicit alignment guide.

Environment

5

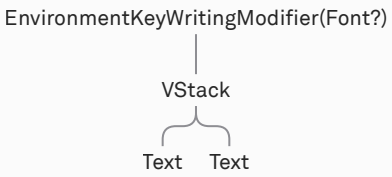
The environment is a key mechanism that makes SwiftUI code so compact. In essence, it's a built-in technique for performing dependency injection.

Here's an example: when we set the font on the VStack, it propagates down to both text views and changes their appearance:

Code

```
VStack {
  Text("Item 1")
  Text("Item 2")
}
.font(.title)
```

View Tree



In the view tree, we can see that the .font modifier gets translated to a EnvironmentKeyWritingModifier with a value of Font?.

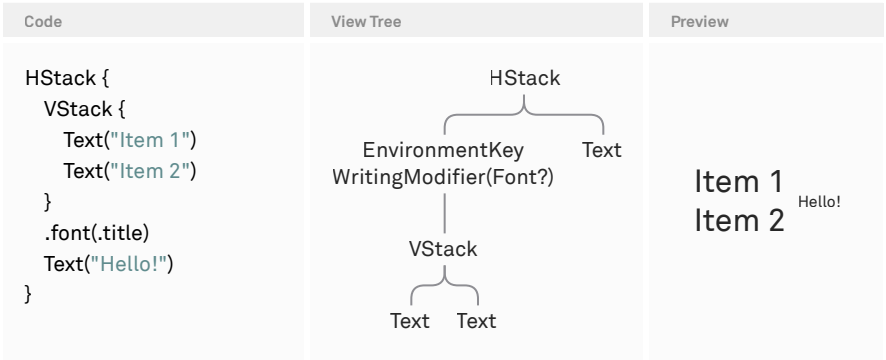
Instead of .font(.title), we could also write the following:

```
VStack {
  ...
}
.environment(\.font, .title)
```

This results in exactly the same view tree; the .font modifier is just a more discoverable and convenient syntax for writing a font into the environment.

The first parameter of the .environment(_:value:) API is a key path in EnvironmentValues, which we can think of as some sort of dictionary that stores all the environment values. We'll learn more about how this works when we define a key path for a custom value later on.

The environment is SwiftUI's mechanism for propagating values down the view tree. In our example, this means that all views downstream from the EnvironmentKeyWritingModifier — the VStack, as well as both Text views — can read the font value we specified from the environment. Views upstream from the EnvironmentKeyWritingModifier or in other branches of the view tree cannot see the custom value we set here. Here's a simple example to demonstrate this propagation behavior:



The text that's a sibling to the EnvironmentKeyWritingModifier still shows the default font, whereas the two texts within the VStack are rendered using the large title font.

We can think of the environment as a dictionary with keys and values that's passed from the root of the view tree all the way down to the leaf nodes. At any point in the view tree, we can change the value for a key by writing into the environment (as we did with the .font modifier). The subtree beneath the environment writing modifier will then receive that modified environment.

Many modifiers in SwiftUI use the environment to have an effect on the entire subtree rather than on a single view. For example, we can set the background style, the text case, and the font, among many other things (the documentation for EnvironmentValues contains a complete list). The environment is also used to make "global" settings such as the current locale or the time zone available to every view in the app.

Reading from the Environment

We can read from the environment using the @Environment property wrapper. This gives us access to a specific value and also observes the environment for any changes of that value. This means that in SwiftUI, we don't need concepts like the auto-updating current locale, as any time the locale changes, it's propagated through the environment, and all views that read the current locale are rerendered. Likewise, we can read things like the current dynamic type size:

```
struct ContentView: View {
  @Environment(\.dynamicTypeSize) private var dynamicTypeSize
  var body: some View {
    HStack {
      Text("Hello")
```

```
        if dynamicTypeSize < .large {
          Image(systemName: "hand.wave")
        }
      }
    }
  }
}
```

When we run the view above in the simulator using the default settings, we won't see the image. But when we override the dynamic type settings, we'll see that the content view automatically gets rerendered and inserts or removes the image accordingly.

Note that the content view will rerender any time the dynamic type size changes. Mostly, this isn't an issue, but sometimes we're interested in only a very specific part of the environment, so the environment property wrapper allows us to specify any key path we want. For example, if we were only interested in whether or not the dynamic type size is an accessibility size, we could write the following:

```
@Environment(\.dynamicTypeSize.isAccessibilitySize) private var isAccessibilitySize
```

This property only triggers a rerender when the value of `isAccessibilitySize` changes; it doesn't rerender on every change of `dynamicTypeSize`. This will most certainly not make a big performance difference in an app, as the dynamic type size hardly changes. But when we're putting large values into the environment, we can avoid unnecessary rerenderings by writing key paths that are more specific.

Similar to `@State` properties, it doesn't make sense to expose `@Environment` properties to the outside. Therefore, we typically mark them as private and use a linting rule to verify that they're private. Another similarity to state properties is that we can only read from the environment inside the `body` of a view; we cannot read from the environment inside a view's initializer, as the view doesn't yet have identity at that point. When we do try to read an environment property in the view's initializer, we'll get a runtime warning.

Custom Environment Keys

In the section on alignment, we showed how to build a badge view. In this section, we'll extend that badge to be styleable. For now, we'll disregard the alignment and focus only on the view that draws the badge around its content. Here's a simple implementation:

```
struct Badge: ViewModifier {
  func body(content: Content) -> some View {
    content
      .font(.caption)
      .foregroundColor(.white)
      .padding(.horizontal, 5)
      .padding(.vertical, 2)
      .background {
        Capsule(style: .continuous)
          .fill(.blue)
      }
  }
}
```

We can use the badge as follows:

Code	Preview
`Text(3000, format: .number)` `  .modifier(Badge())`	

Currently, the badge's style is hardcoded to have a blue capsule and a white text color. To make the badge's color styleable, we can switch out the blue for the tint color:

```
content
  ...
  .background {
    Capsule(style: .continuous)
      .fill(.tint)
  }
```

The badge will now render using the current tint color, falling back to the system default if we don't specify a color. For example, we can change the tint color to red for an entire subtree:

Code

```
VStack {
  Text(3000, format: .number)
    .modifier(Badge())
  Text("Test")
    .modifier(Badge())
}
.tint(.red)
```

Preview

Instead of relying on the tint color, we can also create our own environment key. Doing so takes two required steps and an optional one:

- → We need to implement a custom EnvironmentKey as the key for the badge color and associate the Color type with the key.
- → We need an extension on EnvironmentValues with a property that lets us get and set the value.
- → Optionally, we can provide a helper method on View to easily set the badge color for an entire subtree. This lets us hide the custom key and extension, and it provides a discoverable API for users.

As a first step, let's create a custom environment key. This is slightly complicated at first, but we'll get used to the pattern quickly, and it keeps the environment type-safe: an environment key isn't a value, but it's defined using a type. We'll create an empty enum type as our key — though we could also use a struct — and conform it to the EnvironmentKey protocol:

```
enum BadgeColorKey: EnvironmentKey {
  static var defaultValue: Color = .blue
}
```

The EnvironmentKey protocol requires us to implement the static defaultValue property. Because we use Color as the type, the compiler knows that the value for this key will always be of type Color. We also provide .blue as the default value. Whenever we try to read the badge color from the environment without an explicit value being set at that point in the view tree, we'll get back the default value.

In the second step, we'll add a computed property on the EnvironmentValues struct. The name badgeColor will be used in the key path when we read from and write into the environment:

```
extension EnvironmentValues {
  var badgeColor: Color {
    get { self[BadgeColorKey.self] }
    set { self[BadgeColorKey.self] = newValue }
  }
}
```

Finally, we add a helper to set the badge color:

```
extension View {
  func badgeColor(_ color: Color) -> some View {
    environment(\.badgeColor, color)
  }
}
```

Apple's documentation for EnvironmentValues also has a template for these steps.

To read out the badge color, we'll use the @Environment property wrapper inside our Badge view modifier:

```
struct Badge: ViewModifier {
  @Environment(\.badgeColor) private var badgeColor
  func body(content: Content) -> some View {
    content
      .font(.caption)
      .foregroundColor(.white)
      .padding(.horizontal, 5)
      .padding(.vertical, 2)
      .background {
        Capsule(style: .continuous)
          .fill(badgeColor)
      }
  }
}
```

Now we can style our badge using the environment:

Code

```
VStack {
  Text(3000, format: .number)
    .modifier(Badge())
  Text("Test")
    .modifier(Badge())
}
.badgeColor(.orange)
```

Preview

Note that when we don't specify the badgeColor, the default value will be used. In general, we could say that the default value from the environment is always used, unless we specifically override it.

For some components, we might want to have even more styling than this. For example, SwiftUI's buttonStyle modifier (combined with the ButtonStyle protocol) lets us completely change how buttons are rendered. To achieve this for our own components, we can use the environment as well.

Custom Component Styles

When we want to create a custom component that's styleable through the environment — similar to how SwiftUI's button can be styled with the .buttonStyle API — the general approach is the same as with regular environment values. However, we need to do a little bit more work to get the types right. As a start, we'd like to have the following API for the badge:

```
Text("Test")
  .badge {
    Text(3000, format: .number)
  }
```

The badge should render with a default badge style, but we want to be able to override the style later in the view hierarchy using an API like this:

```
someView
  .badgeStyle(.custom)
```

All the badges contained within someView should receive that new, custom badge style. To build this, we'll do the following:

1. Create a protocol BadgeStyle that defines the interface for a badge style.
2. Create an environment key for the badge style.

3. Use the custom badge style within the badge modifier.

The protocol for our badge style is similar to the ViewModifier protocol: it requires a single body method that wraps an existing view in a badge. We only want this style to be responsible for applying some sort of badge chrome to the label — it doesn't need to concern itself with positioning the badge relative to the view, as that will be done in the badge modifier. Note that the makeBody method takes an AnyView, since we need a concrete view type here to make this code compile:

```
protocol BadgeStyle {
  associatedtype Body: View
  @ViewBuilder func makeBody(_ label: AnyView) -> Body
}
```

Now we can create a default badge style:

```
struct DefaultBadgeStyle: BadgeStyle {
  var color: Color = .red
  func makeBody(_ label: AnyView) -> some View {
    label
      .font(.caption)
      .foregroundColor(.white)
      .padding(.horizontal, 5)
      .padding(.vertical, 2)
      .background {
        Capsule(style: .continuous)
          .fill(color)
      }
  }
}
```

However, when we want to create an environment key, we can't just use DefaultBadgeStyle as the type for the key's value. That would prevent us from ever using a custom badge style with a different type. Instead, we'll use an *existential* to hide the concrete type. Essentially, this wraps the concrete type in a box (similar to how AnyView wraps a concrete view in a box). Here's the environment key and the corresponding property:

```
enum BadgeStyleKey: EnvironmentKey {
  static var defaultValue: any BadgeStyle = DefaultBadgeStyle()
}
```

```
extension EnvironmentValues {
  var badgeStyle: any BadgeStyle {
    get { self[BadgeStyleKey.self] }
    set { self[BadgeStyleKey.self] = newValue }
  }
}
```

To use our badge style, we'll create a custom view modifier that reads it from the environment, transforms the label, and positions it according to the specified alignment. It also takes care of making the label its ideal size (as we don't want the badge to be dependent on the size of the view it's applied to). The layout code with the alignment guides is similar to what we discussed in the alignment section of the layout chapter:

```
struct OverlayBadge<BadgeLabel: View>: ViewModifier {
  var alignment: Alignment = .topTrailing
  var label: BadgeLabel
  @Environment(\.badgeStyle) private var badgeStyle

  func body(content: Content) -> some View {
    content
      .overlay(alignment: alignment) {
        AnyView(badgeStyle.makeBody(AnyView(label)))
          .fixedSize()
          .alignmentGuide(alignment.horizontal) { $0[HorizontalAlignment.center] }
          .alignmentGuide(alignment.vertical) { $0[VerticalAlignment.center] }
      }
  }
}
```

The OverlayBadge modifier doesn't need to be public. Instead, we can write a short helper method on the View protocol to allow for the syntax at the beginning of this section:

```
extension View {
  func badge<V: View>(alignment: Alignment = .topTrailing,
    @ViewBuilder _ content: () -> V) -> some View {
    modifier(OverlayBadge(alignment: alignment, label: content()))
  }
}
```

Now we can use our badge modifier like this, and it'll automatically use the current badge style in the environment:

Code	Preview
Text("Hello") .badge { Text(100, format: .number) }	

Let's create a custom, glossier badge style to demonstrate that the style in the environment actually gets used:

```
struct FancyBadgeStyle: BadgeStyle {
  var background: some View {
    ZStack {
      ContainerRelativeShape()
        .fill(Color.red)
        .overlay {
          ContainerRelativeShape()
            .fill(LinearGradient(colors: [.white, .clear],
              startPoint: .top, endPoint: .center))
        }
      ContainerRelativeShape()
        .strokeBorder(Color.white, lineWidth: 2)
        .shadow(radius: 2)
    }
  }
  func makeBody(_ label: AnyView) -> some View {
    label
      .foregroundColor(.white)
      .font(.caption)
      .padding(.horizontal, 7)
      .padding(.vertical, 4)
      .background(background)
      .containerShape(Capsule(style: .continuous))
  }
}
```

To make it easier to set the badge style in the environment, we'll again add an extension on View:

```
extension View {
  func badgeStyle(_ style: any BadgeStyle) -> some View {
    environment(\.badgeStyle, style)
  }
}
```

We can now use our fancy badge style by specifying .badgeStyle(FancyBadgeStyle()) anywhere in the hierarchy, and the entire subtree will be rendered using this style. To make the syntax even nicer, we can add the following extension to BadgeStyle:

```
extension BadgeStyle where Self == FancyBadgeStyle {
  static var fancy: FancyBadgeStyle {
    FancyBadgeStyle()
  }
}
```

Now we can write .badgeStyle(.fancy):

Code

```
HStack {
  Text("Inbox")
  Text("Spam")
    .badge {
      Text(3000, format: .number)
    }
}
.badgeStyle(.fancy)
```

Preview

It's unfortunate that we had to use an AnyView in our definition of BadgeStyle. However, we can't make it a generic parameter, as the associated Body type needs to be static (it can't be dependent on Label). As Kasper Lahti showed, we can at least hide that fact by creating an additional Label struct, but at the moment of writing, we cannot write this feature without using AnyView. In the same blog article, Kasper also shows how we can could get access to things like @Environment properties in our custom badge style.

Environment Objects

We can also use the environment for passing around objects, and not just for values. The way this works changed with iOS 17/macOS 14, so we'll have to distinguish between the platforms we want to target.

As of iOS 17, environment objects should use the new `@Observable` macro, and the property should be declared with the same `@Environment` property wrapper that we've used throughout this chapter. The only difference is that we can use the type of the environment object as the key without having to declare a separate key type conforming to the `EnvironmentKey` protocol.

Here's a simple example of using an environment object:

```
@Observable final class UserModel {
  ...
}

struct Nested: View {
  @Environment var userModel: UserModel?

  var body: some View {
    Text(userModel?.name ?? "default name")
    ...
  }
}
```

If some view further up in the hierarchy has set a user model in the environment, we'll be able to access it here without having to pass the object through all the levels of the view tree. If the object hasn't been set in the environment, the `userModel` property will be `nil`. Setting the object is simple:

```
struct ContentView: View {
  var body: some View {
    Nested()
      .padding()
      .environment(UserModel())
  }
}
```

We can also declare a property for an environment object as a non-optional type. In this case, we have to rely on this object being present in the environment. Otherwise, the app will crash when we try to access the environment property.Using the object's type as the environment key is convenient, but it's not strictly necessary. We could

also define an environment key, as we've done earlier in this chapter, and pass the object with this explicit key down the view tree.

If we want to target platforms prior to iOS 17, we have to use two different APIs: the `@EnvironmentObject` property wrapper is used to read an object from the environment, and the `environmentObject` modifier is used to set an object in the environment. These APIs also rely on the type of the object as the key. However, the object has to conform to the `ObservableObject` protocol (just like state objects or observed objects).

When using `@EnvironmentObject`, there's no way to declare the property with an optional type. The code will crash if no object of the specified type has been set in the environment and we access the `@EnvironmentObject` property. A helpful technique is to at least bundle up all our environment object setters into a single helper method:

```
extension View {
  func injectDependencies() -> some View {
    environmentObject(UserModel())
      .environmentObject(Database())
  }
}
```

We can use this helper not only on the app's root view, but also to inject all dependencies into our previews, while still being able to override dependencies locally:

```
struct Nested_Previews: PreviewProvider {
  static var previews: some View {
    Nested()
      .environmentObject(UserModel.mock())
      .injectDependencies()
  }
}
```

Note that environment objects work well with subclassing — if our `UserModel.mock()` returns a subclass of `UserModel`, it's still read correctly by the nested view.

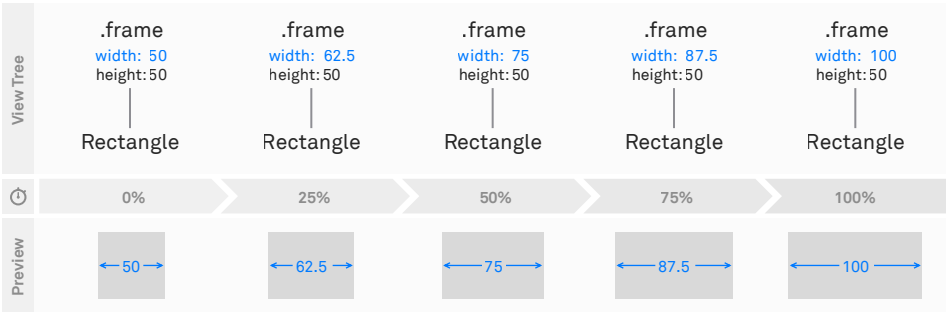
Animations

6

In SwiftUI, the only way to trigger a view update is by changing state. By default, the changes between the old and the new view tree aren't animated, but there are multiple ways to tell SwiftUI to animate some or all changes that occurred as a result of a state change. Here's an example:

```
struct ContentView: View {
 @State private var flag = false
 var body: some View {
  Rectangle()
   .frame(width: flag ? 100 : 50, height: 50)
   .onTapGesture {
    withAnimation(.linear) { flag.toggle() }
   }
 }
}
```

When the user first taps the rectangle, the flag state is changed from false to true, and the body gets reexecuted. The new view tree still has the same structure: a frame with a rectangle inside (we're ignoring the tap gesture that's also part of the view tree for the sake of brevity here).



However, the frame's width has changed from 50 to 100 points. Since the state change has been wrapped in a call to withAnimation { ... }, SwiftUI will compare the new view tree to the current render tree and look for changes that are animatable. Since the width of a frame is an animatable property (as almost all view modifier parameters are), the .default timing curve (implicit in the call to withAnimation) is used to generate progress values for the animation from 0 to 1. These progress values are then used to interpolate the width of the rectangle from the old width of 50 points to the new width of 100 points.

> While you can think of the progress starting at zero and ending at one, the values during the animation might be smaller than zero or larger than one. For example, when you use iOS 17's .bouncy timing curve, the animation *overshoots*.

Animations always start from the current state of the render tree and move toward the new state that's described by the new view tree, which is generated in response to a state change. The current state of the render tree can also be the transient state of a currently running animation. This makes animations in SwiftUI additive and cancelable by default.

While most of this chapter applies to all recent versions of iOS, iOS 17 introduced many updates, large and small, to SwiftUI's animation system. The main additions are:

- → Animations now have completion handlers that fire once an animation finishes.
- → The `animation` modifier got a new overload that allows scoping an animation to particular modifiers.
- → We can now create completely custom animation curves using the new `CustomAnimation` protocol.
- → Phase-based animations allow us to specify a series of animations that run automatically, one after the other, always returning to their starting value.
- → Keyframe-based animations provide a new abstraction around creating completely custom animations based on the interpolation of arbitrary values.
- → New APIs for transitions, which center around the `Transition` protocol, have been introduced.

Property Animations vs. Transitions

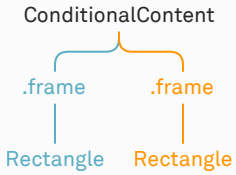
Property animations interpolate changed properties of views that exist in the view tree before and after the state change. In other words, the properties of the views change, but *which* views are onscreen doesn't. In the context of the example above, this means that it's really important that we only change the value of the frame's `width` parameter rather than show two separate views. The frame itself is stable across the state change.

Consider the following alternative (we left the tap gesture out of the diagram for brevity):

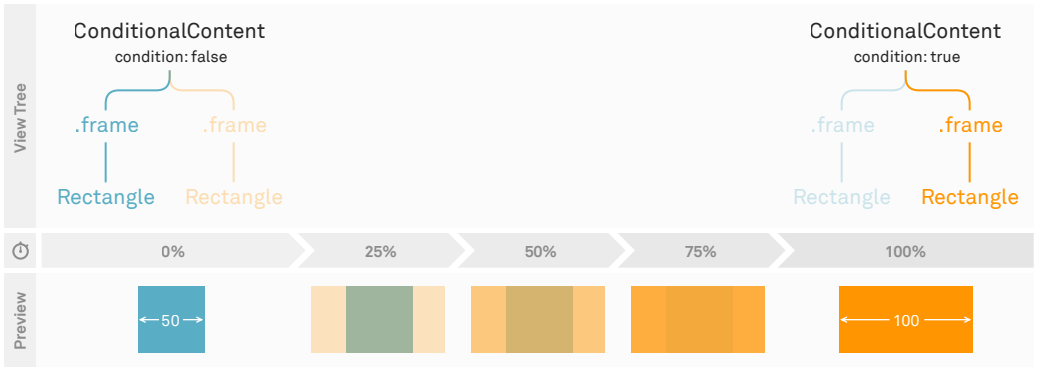
Code

```
var body: some View {
  let rect = Rectangle().onTapGesture {
    withAnimation { flag.toggle() }
  }
  if flag {
    rect
      .frame(width: 200, height: 100)
  } else {
    rect
      .frame(width: 100, height: 100)
  }
}
```

View Tree



When the flag is false, only the blue subtree will be present in the render tree. When the flag switches to true, the blue subtree is taken out of the render tree, and the orange subtree gets inserted. This means that the frame that was there before the state change is *not* the same frame as the one after the state change. These two frames have different identities, because they're located in different positions within the view tree. Therefore, SwiftUI doesn't animate the width property of the frame, because there's no frame that's part of the render tree before and after the state change!



When trying this example, the narrow rectangle fades out, and the wide rectangle fades in. What we're seeing here is what's called a *transition*. Unlike property animations, transitions are animations that are applied to views being removed from or inserted into the render tree. We'll take a closer look at transitions later in this chapter.

Controlling Animations

There are multiple ways of specifying when animations happen in SwiftUI:

1. Implicit animations occur when a particular value changes.
2. Explicit animations occur when a particular event takes place.

To specify implicit animations, the .animation(_:value:) view modifier can be applied anywhere in the view tree and has two parameters: the first one specifies the timing curve to be used for the animation, and the second one specifies the value that needs to change for the animation to be applied. The value parameter is used to restrict the scope of the animation to only certain changes. This is important to gain control over when an animation should actually take effect.

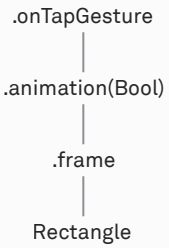
> In the first version of SwiftUI, there was the now-deprecated .animation(_:) view modifier (without the value parameter). This modifier applied animations way too broadly and often caused unexpected behaviors, like animations being erroneously applied when the device orientation changed.

When we rewrite our first example and use an implicit animation instead of an explicit animation, it looks like this:

Code

```
struct ContentView: View {
  @State private var flag = false
  var body: some View {
    Rectangle()
      .frame(width: flag ? 100 : 50, height: 50)
      .animation(.default, value: flag)
      .onTapGesture { flag.toggle() }
  }
}
```

View Tree



Implicit animations using the .animation view modifier animate everything within their view subtree, so it's important to place the modifier in the correct spot. It's generally a good idea to apply animations as locally as possible to avoid unintended side effects, especially when we make changes to our code later on. When we look at the view tree for the animation above, we can see that the frame and rectangle form the subtree of the animation modifier.

> Unfortunately, at the time of writing, there are some exceptions to the rule that .animation(_:value:) only animates changes in its subtree. For example, if we switch the .frame and .animation lines in the example above, the frame will still animate. In our understanding, this is intended behavior. Ole Begemann describes these exceptions nicely in his blog post.

As of iOS 17, the new .animation(_:body:) modifier allows us to scope implicit animations to particular modifiers, as shown in the following code:

```
Text("Hello World")
  .opacity(flag ? 1 : 0)
  .animation(.default) {
    $0.rotationEffect(flag ? .zero : .degrees(90))
  }
```

This will only animate the rotation effect, and not the opacity, when the flag value changes. In our testing, this works the same as the following code:

```
Text("Hello World")
  .opacity(flag ? 1 : 0)
  .animation(nil)
  .rotationEffect(flag ? .zero : .degrees(90))
  .animation(.default)
```

This new variant of .animation comes with two potential pitfalls:

1. Contrary to the .animation(_:value:) modifier, the new animation modifier with a body closure doesn't take a value parameter to control when the animation will take effect. This means that the animatable modifiers within the body closure will always animate when their parameters change, regardless of where this change originated.
2. As mentioned, some modifiers — like .frame, .offset, or .foregroundColor — might have unexpected behavior in conjunction with animations. These modifiers take effect at the leaf view, and not at the position in the view tree where we insert them. Therefore, these modifiers might still animate although there was no animation present at the point in the view tree where we used them. If we specify one of these "out-of-place" modifiers inside the body closure of .animation(_:body:), no animation will take place, because no animation is present at the leaf view.

Instead of defining an animation for a certain view subtree and a particular value changing, we can also scope an animation to particular state changes, which we call an *explicit animation*. In fact, that's how we started out in the first example in this chapter using the withAnimation modifier:

```
struct ContentView: View {
  @State private var flag = false
  var body: some View {
    Rectangle()
      .frame(width: flag ? 200 : 100, height: 100)
      .onTapGesture {
        withAnimation { flag.toggle() }
      }
  }
}
```

Whenever we have a callback like the onTapGesture 's perform closure, we can use withAnimation to wrap the state changes that should be animated.

Binding animations are another variant of explicit animations. We can call .animation on a binding, which wraps the setter of the binding in an explicit animation. For the sake of example, we could rewrite the code above like this:

```
struct ToggleRectangle: View {
  @Binding var flag: Bool
  var body: some View {
    Rectangle()
      .frame(width: flag ? 100 : 50, height: 50)
      .onTapGesture { flag.toggle() }
  }
}

struct ContentView: View {
  @State private var flag = false
  var body: some View {
    ToggleRectangle(flag: $flag.animation(.default))
  }
}
```

This code does exactly the same thing as the variant before, and of course, the original version is much more straightforward. However, using an animation on a binding can be a good option to apply an explicit animation without modifying the code of the event closure (or without having an event closure at all).

At first sight, implicit and explicit animations seem to accomplish the same thing. However, it's important to note that implicit animations have potentially different effects than explicit ones. For example, consider the case where a change in the model layer (perhaps by new data being pushed from the server) results in a particular value changing. When using an implicit animation that's scoped to that value, the change to the render tree will be animated, regardless of the source, but the scope within the view is well-defined. With explicit animations, we can easily

distinguish between updates from the model layer and user interactions, but we can't directly restrict the animation to certain parts of the view tree.

There's no right or wrong solution here. Whether we should use an explicit or implicit animation depends on the behavior we're trying to achieve. It's important to be aware that the different ways of defining animations do have potentially different behavior, even though this doesn't necessarily show up in immediate testing.

Timing Curves

SwiftUI comes with all the usual animation timing curves built in, e.g. linear, ease-in or ease-out, and spring timing curves. All animation APIs have a parameter to specify the timing curve and will fall back to the default timing curve when the parameter is left out.

- → .speed(_:) lets us slow down or speed up animations using a factor.
- → .delay(_:) lets us delay the start of an animation by a fixed time.
- → .repeatCount(_:autoreverses:) lets us loop an animation a certain number of times.

If the built-in timing curves don't do what we want, we can use the CustomAnimation protocol, which was introduced in iOS 17, to implement completely custom timing curves. Prior to iOS 17, the only way to create animations with custom timing curves was to implement a custom animation (using the Animatable protocol, which we'll look at below) on top of a linear timing curve.

Transactions

Under the hood, both implicit and explicit animations use the same construct: transactions. Each view update (triggered by a state change) is wrapped in a transaction, which carries the information about the animation to be applied. By default, a transaction's animation is nil.

The withAnimation API implicitly creates a transaction and sets the animation property of that transaction. The scope of the implicit transaction is the body of the closure that's passed to withAnimation. In fact, we can achieve the same effect by using the withTransaction API:

```
struct ContentView: View {
  @State private var flag = false
  var body: some View {
    Rectangle()
      .frame(width: flag ? 100 : 50, height: 50)
      .onTapGesture {
```

```
            var t = Transaction(animation: .default)
            withTransaction(t) { flag.toggle() }
        }
    }
}
```

Similarly, we can rewrite an implicit animation with the .transaction view modifier:

```
struct ContentView: View {
    @State private var flag = false
    var body: some View {
        Rectangle()
            .frame(width: flag ? 100 : 50, height: 50)
            .transaction { t in
                t.animation = .default
            }
            .onTapGesture { flag.toggle() }
    }
}
```

Note, however, that this code behaves like the deprecated .animation(_:) API without the value parameter. To mimic the .animation(animation:value:) API using .transaction, we'd have to remember the old value of flag and compare it to the new value before setting the animation on the transaction. Similar to an implicit animation, the transaction is passed down the view tree (in this case, to the frame and the rectangle).

Since both types of animation just set the animation property of the current transaction under the hood, implicit animations take precedence over explicit animations. The modifier setting the implicit animation will be executed later while evaluating the view tree, therefore overriding any explicit animation that might have been set at the start of the state change.

In the context of animation precedence, there's one more useful property on Transaction, and it's called disablesAnimations. When we set this property to true, implicit animations won't have any effect during the transaction. The disablesAnimations property isn't disabling animations in general; it just prevents implicit animations from overriding the transaction's animation.

If we change the example above to use an explicit transaction and disable implicit animations (by setting transaction.disablesAnimations to true), we can see that the green rectangle moves slowly when tapped.

In other words, we can imagine an implicit animation to be implemented something like this:

```
extension View {
  func animation(_ animation: Animation?) -> some View {
    transaction { t in
      guard !t.disablesAnimations else { return }
      t.animation = animation
    }
  }
}
```

The .transaction API can also be useful to remove the current animation. Previously, before the .animation(_:) API was deprecated, we could just write .animation(nil) to disable animations for the subtree in question. This will now result in a deprecation warning. Instead, we can write .transaction { $0.animation = nil }.

As of iOS 17, transactions are extensible — the same way the environment is extensible — through custom keys. Instead of the EnvironmentKey protocol, we use the TransactionKey protocol. This allows us to attach state to a transaction and read it later on while the transaction is processed. For example, we could add the source of a change to the transaction (whether it originated on the server or the client) and choose our implicit animation based on this value.

Completion Handlers

Animation completion handlers are part of the transaction as well, and they're available as of iOS 17. We can set completion handlers directly when using the explicit withAnimation API, or we can add them to a transaction using .addAnimationCompletion within the closure of a .transaction modifier. Generally, these work as expected, but let's look at some cases that aren't entirely intuitive.

When adding completion handlers to a transaction manually, we need to ensure that the closure that modifies the transaction is executed every time the animation gets triggered. Otherwise, the completion handler will only fire once. Consider this example:

```
struct ContentView: View {
  @State private var flag = false

  var body: some View {
    VStack {
      Button("Animate!") { flag.toggle() }
      Circle()
        .fill(flag ? .green : .red)
        .frame(width: 50, height: 50)
        .animation(.default, value: flag)
```

```
            .transaction {
              $0.addAnimationCompletion { print("Done!") }
            }
        }
      }
    }
```

Since we toggle the flag state property each time the button is tapped, the view body is reexecuted each time as well, and the fill color of the circle animates from red to green and back. However, SwiftUI seems to perform dependency tracking on the .transaction closure, and since this closure doesn't depend on any state, it doesn't get reexecuted each time the body executes. Therefore, the completion handler only fires the first time.

We can fix this issue by using the .transaction(value:_ transform:) modifier to force the transform closure to be called each time the specified value changes:

```
var body: some View {
  VStack {
    Button("Animate!") { flag.toggle() }
    Circle()
      ...
      .transaction(value: flag) {
        $0.addAnimationCompletion { print("Done!") }
      }
  }
}
```

Completion handlers can specify whether they should be called when an animation is logically complete (i.e. it "feels" complete in the eye of a human observer), or when the animation curve really has finished and the animation has been removed (which could happen significantly later, as is the case with spring animations). When implementing custom animation curves using the new CustomAnimation protocol, we can specify the point where our animation should count as logically complete by setting the isLogicallyComplete flag on the AnimationContext.

The Animatable Protocol

At the heart of SwiftUI's property animation system lies the Animatable protocol. This protocol can be adopted by views and view modifiers and exposes their animatable properties to SwiftUI.

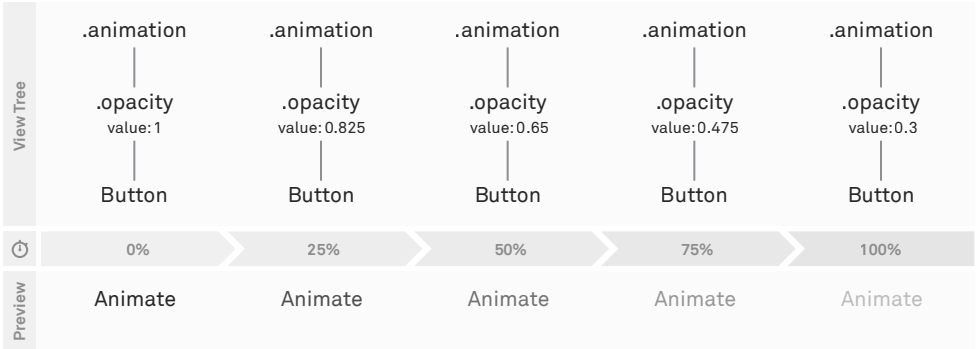
The only requirement of this protocol is the animatableData property (which has a default implementation that does nothing). The type of animatableData has to conform to the VectorArithmetic protocol, which essentially means that we can multiply the value with a float and add two values. This allows SwiftUI to interpolate the change from one value to another.

When a transaction has an associated animation, SwiftUI inspects each view and looks for animatable data properties that have changed. It then interpolates the change using the timing curve of the current animation. Each interpolated value will be set through the animatableData property, which will then reexecute the view's body or the view modifier's body.

Usually, all this happens transparently in the background, and we won't have to conform to this protocol ourselves. However, there are some use cases where we need to implement a custom animatable view or view modifier to achieve the result we want. To first get a better understanding of how the Animatable protocol is used to enable property animations in SwiftUI, let's take a look at a simple example with an opacity animation:

```
struct ContentView: View {
  @State private var flag = true
  var body: some View {
    Button("Animate") { flag.toggle() }
      .opacity(flag ? 1 : 0.3)
      .animation(.linear(duration: 1), value: flag)
  }
}
```

When we tap the button, the flag state property changes, and a new transaction with a linear animation is created. SwiftUI now traverses further down the view tree and looks for views or view modifiers in the tree that conform to the Animatable protocol and that have a different animatableData value compared to the old state. In our case, the only change is the opacity value. The opacity view modifier is animatable out of the box, i.e. it exposes the opacity value as animatable data. SwiftUI will now start to interpolate this value from 1 to 0.3, using the linear timing curve with a duration of one second. At 60 frames per second, this means that the opacity view modifier's animatable data property will be set 60 times, with values linearly decreasing from 1 to 0.3 in small increments.



For more insight into what's happening behind the scenes, we can also implement an animatable opacity modifier ourselves:

```
struct MyOpacity: ViewModifier, Animatable {
  var animatableData: Double

  init(_ opacity: Double) {
    animatableData = opacity
  }

  func body(content: Content) -> some View {
    let _ = print(animatableData)
    content.opacity(animatableData)
  }
}
```

Of course, we're using SwiftUI's opacity modifier to implement our own, but this gives us the opportunity to log the interpolated values of the animatable data property to the console. Let's change the example above to use our own opacity modifier:

```
Button("Animate Opacity") { flag.toggle() }
  .modifier(MyOpacity(flag ? 1 : 0.3))
  .animation(.linear(duration: 1), value: flag)
```

When we run this app, a lot of values will appear in the console:

```
1.0
0.988333511352539
0.9766663551330566
0.9649998664855957
...
0.33500013351440433
```

```
0.3233336448669433
0.3116664886474609
0.3
```

For each value of the animatable data property, the view modifier's body will be reexecuted, allowing us to update the view based on the current animation state.

One common case where we need to adopt the `Animatable` protocol is that of animations that should visually end up in the same state they started in. Before iOS 17, the problem with this kind of animation was that we had no way to tell SwiftUI what and how to animate if the view tree didn't change. For example, if we wanted to shake a button in response to a tap — with the button ending up in the same spot it started from — there wouldn't be changes in the view tree to animate. As of iOS 17, we can use either phase animations or keyframe animations, both of which we'll discuss at the end of this chapter.

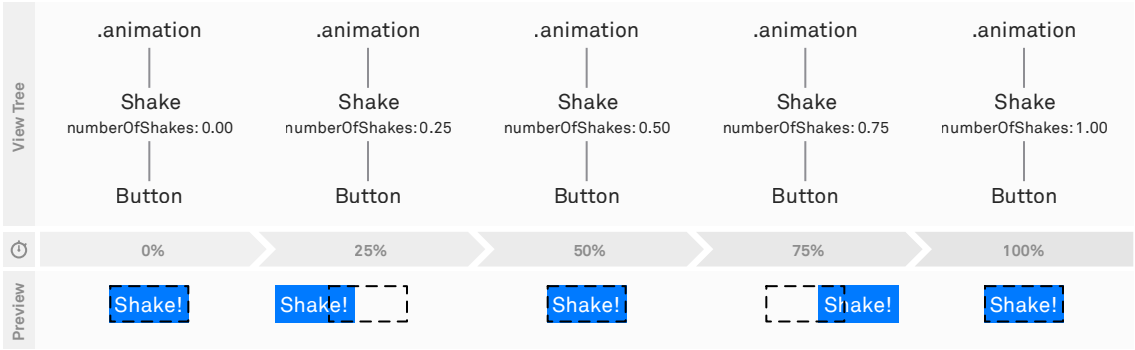
Let's take a look how we can use a custom animatable view modifier (i.e. without depending on the new iOS 17 APIs) to implement the shake animation, which shakes a view from its current position over to the left, then to the right, and back to its original position:

```
struct Shake: ViewModifier, Animatable {
  var numberOfShakes: Double
  var animatableData: Double {
    get { numberOfShakes }
    set { numberOfShakes = newValue }
  }

  func body(content: Content) -> some View {
    content
      .offset(x: -sin(numberOfShakes * 2 * .pi) * 30)
  }
}
```

We can use this modifier like this:

```
struct ContentView: View {
  @State private var shakes = 0

  var body: some View {
    Button("Shake!") { shakes += 1 }
      .modifier(Shake(numberOfShakes: Double(shakes)))
      .animation(.default, value: shakes)
  }
}
```

When the user taps the shake button, the shakes state property is incremented by one. The structure of the new view tree and the render tree is the same; only the properties are different. In the render tree, the numberOfShakes property of Shake is zero, and in the new view tree, the property is one.



Since the animatableData property on Shake simply forwards the value of the numberOfShakes property, the value of animatableData on Shake has changed from zero to one as well. Therefore, SwiftUI will interpolate this change using the specified .default timing curve from zero to one. Each time animatableData changes, the body of the view modifier will be reexecuted, and the position of the view onscreen will be updated.

In the body method of the Shake modifier, we can use the numberOfShakes value (or the value of animatableData, which is the same thing) to implement the actual shake movement using an offset. We feed the numberOfShakes value into a sine function, applying some additional math to ensure the view first moves 30 points to the left, then over 60 points to the right, and back to where it started from.

> We can use Apple's built-in Grapher app on macOS to visually reproduce what the math in this view modifier does.

If we need more than one value to be interpolated in a custom animatable view or view modifier, we can use the AnimatablePair type to compose multiple animatable values into tuples.

> Since the animatableData property has a default implementation of type EmptyAnimatableData on the Animatable protocol (which means that there's nothing to be animated), it's easy to forget the actual implementation of animatableData. Furthermore, if we add an animatableData property that doesn't conform to VectorArithmetic, the compiler won't complain because of the default implementation, but our animation won't work.

While these kind of “finish where you started” animations are a prime example of custom animations, there are many other effects we can achieve this way. Basically, we have complete freedom to build whatever animation we want based off some value that SwiftUI interpolates for us. For example, we could build animations with custom timing curves, or staggered animations that all run within a single SwiftUI animation.

Transitions

Along with animating properties of views that are already onscreen, we can also animate insertions and removals of views. SwiftUI calls these animations *transitions*. In iOS 17, there’s a new protocol-based API, but the functionality is the same as before. We’ll start by explaining the pre-iOS 17 APIs, and afterward, we’ll discuss the differences between the old and new APIs.

When we animate a state change that inserts or removes a view, SwiftUI applies the default .opacity transition. Let’s take a look at a minimal example in which a rectangle is inserted or removed depending on a flag:

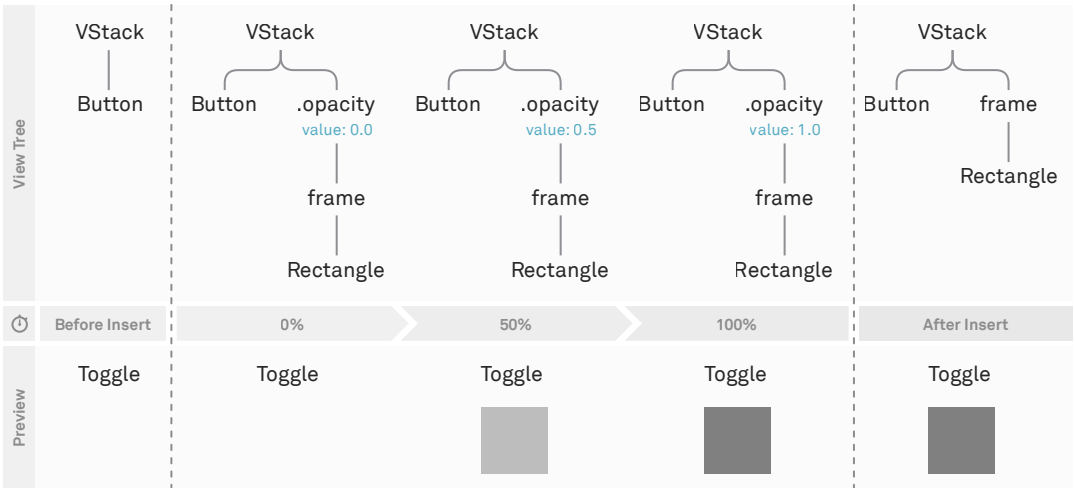
```
struct ContentView: View {
  @State private var flag = true

  var body: some View {
    VStack {
      Button("Toggle") {
        withAnimation { flag.toggle() }
      }
      if flag {
        Rectangle()
          .frame(width: 100, height: 100)
      }
    }
  }
}
```

In the button’s action, we use an explicit animation to apply an animation to the transaction. The rectangle then fades in and out, which is the default .opacity transition at work.

To specify this transition explicitly, we’d add the .transition modifier to the rectangle:

```
if flag {
  Rectangle()
    .frame(width: 100, height: 100)
    .transition(.opacity)
}
```

Transitions have two states: the *active* state, and the *identity* state. When a view is inserted, it animates from the active state of the transition to the identity state. When it's removed, it animates from the identity state to the active state. For the opacity transition, the active state applies an .opacity(0) modifier, and the identity state applies .opacity(1).



We can use these two states of transitions in SwiftUI's API to create custom transitions using AnyTransition.modifier(active:identity:). Both parameters take a view modifier that defines the state. For example, if we'd want to reimplement SwiftUI's default .opacity transition, we could write the following:

```
struct MyOpacity: ViewModifier {
  var value: Double
  func body(content: Content) -> some View {
    content.opacity(value)
  }
}
```

```
extension AnyTransition {
  static let myOpacity = AnyTransition.modifier(
    active: MyOpacity(value: 0),
    identity: MyOpacity(value: 1)
  )
}
```

Before the view with our .myOpacity transition is inserted, the MyOpacity(value: 0) view modifier is applied to put the view into the initial, active state of the transition. Then the modifier is changed to the identity one, applying the MyOpacity(value: 1) modifier to make the view fully opaque after the insertion animation has finished. For removal, the process is reversed: the view starts out at its "resting" position with the identity modifier applied, and then it gets the active modifier applied to animate it to its final state before it disappears.

> Note that it's not necessary for a view to be removed by an if or switch statement for a transition to be triggered. If the identity of the view changes, this will trigger the transition too, because from SwiftUI's perspective, the old view (with the old identity) has been removed, and a new view (with a new identity) has been inserted. Using the .id(_:) modifier to trigger transitions can be a useful tool.

SwiftUI has a whole range of built-in transitions, like .slide, .move(edge:), and .scale(scale:anchor:) — among others — and we can also combine them in parallel using the .combined(with:) transition. To specify different transitions for insertion and removal, we can use the .asymmetric(insertion:removal:) transition:

```
if flag {
  Rectangle()
    .frame(width: 100, height: 100)
    .transition(
      .asymmetric(insertion: .slide, removal: .scale)
      .combined(with: .opacity)
    )
}
```

This transition uses a slide for the insertion and a scale for the removal, combining both of them with an opacity transition.

If the built-in transitions aren't enough, we can use the .modifier(active:identity:) API we discussed above to implement completely custom transitions. For example, we could use this to build a blur transition. As a first step, we'll need a custom modifier:

```
struct Blur: ViewModifier {
  var radius: CGFloat

  func body(content: Content) -> some View {
    content
      .blur(radius: radius)
  }
}
```

Note that we don't need to mark our view modifier as Animatable, as we're relying on the fact that the built-in .blur modifier is already animatable. We can also add an extension to AnyTransition to provide a convenient API:

```
extension AnyTransition {
  static func blur(radius: CGFloat) -> Self {
    .modifier(active: Blur(radius: 5), identity: Blur(radius: 0))
  }
}
```

We can use this transition in the context of our previous example:

```
if flag {
  Rectangle()
    .frame(width: 100, height: 100)
    .transition(.blur(radius: 5))
}
```

It's important to remember that transitions always work in conjunction with animations. Just applying a .transition modifier to a view doesn't animate the change. For that to happen, we always need to apply an explicit or implicit animation next to the transition. If we use an implicit animation, the animation shouldn't be part of the subtree that gets removed or inserted. For example, this wouldn't work:

```
if flag {
  Rectangle()
    .frame(width: 100, height: 100)
    .transition(.blur(radius: 5))
    .animation(.default, value: flag)
}
```

Instead, we have to place the animation in a spot that's stable when the view gets inserted or removed — for example:

```
VStack {
  if flag {
    Rectangle()
      .frame(width: 100, height: 100)
      .transition(.blur(radius: 5))
  }
}
.animation(.default, value: flag)
```

Here, we used a VStack to wrap the if condition so that we can apply an animation from the outside, but we could've used an HStack or a ZStack as well. It really doesn't matter for this purpose — we just need something to wrap the if condition so that we can apply an animation to this part of the view tree. Of course, we could've also used an explicit animation instead.

In iOS 17, the transition APIs have changed to provide a clearer interface, while keeping the functionality the same. The built-in transitions are now types conforming to the Transition protocol. For convenience, they're accessible via static properties on the Transition protocol as well, so that we can continue to write, for example, .transition(.opacity) or .transition(.scale(2)).

To create a custom transition, we now can create a custom type conforming to the Transition protocol. The concept of active and identity states has been replaced by an explicit TransitionPhase type, which distinguishes between the willAppear, identity, and didDisappear phases. The custom blur transition now looks like this:

```
struct BlurTransition: Transition {
  var radius: CGFloat

  func body(content: Content, phase: TransitionPhase) -> some View {
    content
      .blur(radius: phase.isIdentity ? 0 : radius)
  }
}
```

To make this transition available as a static method on Transition, we can write:

```
extension Transition where Self == BlurTransition {
  static func blur(radius: CGFloat) -> Self {
    BlurTransition(radius: radius)
  }
}
```

Earlier in this section, we showed how we can combine multiple transitions and specify different transitions for insertion and removal. Using the new APIs, we can

rewrite this example — combining an opacity transition with a slide for insertion and a scale for removal — like this:

```
if flag {
 Rectangle()
  .frame(width: 100, height: 100)
  .transition(
    AsymmetricTransition(insertion: .slide, removal: .scale)
      .combined(with: .opacity)
  )
}
```

Inside a custom transition definition, we can also use the TransitionPhase to distinguish between insertion, identity, and removal instead of creating an asymmetric transition.

Overall, we think the new APIs are an improvement and feel more at home with the rest of SwiftUI. However, the old transition APIs aren't yet deprecated, so existing code will keep working without warnings, for the time being.

Phase-Based Animations

The new phase animators introduced in iOS 17 are the easiest way to have "finish where you started" animations. The phases are discrete values that the phase animator cycles through: once one phase completes, the system continues with the next one. A phase animator can either have infinite duration (once it completes the last phase, it continues with the initial phase) or loop through all phases exactly once every time the trigger value changes.

For example, we can reimplement our shake animation from before using a phase animator:

```
struct Sample: View {
  @State private var shakes = 0
  var body: some View {
    Button("Shake") {
      shakes += 1
    }
    .phaseAnimator([0, -20, 20], trigger: shakes) { content, offset in
      content.offset(x: offset)
    }
  }
}
```

The initial phase is the first element in the array (0), and the button renders with an `offset(x: 0)`. Whenever the `shakes` property changes, the phase animator changes its internal state to the next phase (-20). The internal state change is wrapped inside an explicit animation. When that animation completes, the animator changes its state to the third phase (20), again using an animation. Finally, when that animation completes, the animator changes its state back to the initial state. Because we've provided a trigger value, the animator stops here. If we would've left out the trigger value, the animator would loop.

> Note that the phase animator itself doesn't interpolate any values. While we used a `CGFloat` for our phase, we could have had an enum with three cases instead. In the example above, it's the `offset` modifier that animates the position of the view. At the time of writing, the only requirement is that `Phase` conforms to the `Equatable` protocol.

Each phase change happens in a separate animation, i.e. we can't specify one overall timing curve for the phase animator cycling through all phases, but we can specify one timing curve per phase using the `animation` parameter. The separate animations are a key difference to the custom shake animation from above: there, we had a single animation drive the entire shake effect.

Keyframe-Based Animations

Keyframe-based animations are a way of describing animations that transition between discrete values specified in keyframes. Unlike phase animations, these are built using a completely separate subsystem: they aren't built on top of regular animations.

> At the time of writing, keyframe animations still seem to be unfinished. For example, the shake animation below runs correctly the first time but not the second time.

A keyframe animation animates a *single value* over time. This value can be a single double value, but it could also be a struct containing multiple properties — SwiftUI doesn't place any constraints on the value type. A keyframe animation is built out of one or more tracks, with one track per property. For example, imagine a struct with a rotation value and an offset value. We can animate each property independently using its own track. It's helpful to think of a single track as a timeline containing multiple items: each item describes how to move to the next value. This description includes the target value, as well as the duration, and how to interpolate.

When we reimplement our shake animation using a keyframe-based animation, we only need to animate a single float (the offset). Because of this, we don't need to specify multiple tracks:

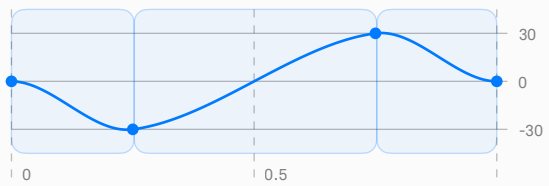
```
struct ShakeSample: View {
  @State private var trigger = 0

  var body: some View {
    Button("Shake") {
      trigger += 1
    }
    .keyframeAnimator(initialValue: 0, trigger: trigger) { content, offset in
      content.offset(x: offset)
    } keyframes: { value in
      CubicKeyframe(-30, duration: 0.25)
      CubicKeyframe(30, duration: 0.5)
      CubicKeyframe(0, duration: 0.25)
    }
  }
}
```

Any time the trigger value changes, the animation runs. The offset will start at 0, and the first keyframe animates toward -30 during the first quarter of a second. The second keyframe animates toward 30 during the next half second, and the final keyframe animates back to 0. Because we chose cubic interpolation, the animation will smoothly ramp up (starting with an initial velocity of 0) and also smoothly transition between the keyframes. Here's a diagram that shows the value plotted over time (the dots represent the keyframes):

Code

```
CubicKeyframe(-30, duration: 0.25)
CubicKeyframe(30, duration: 0.5)
CubicKeyframe(0, duration: 0.25)
```

Preview



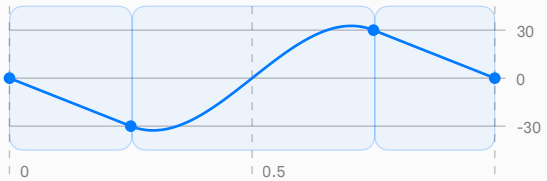
Keyframes compute their curves by inspecting their surrounding keyframes. For example, in the plot above, we see a smooth curve through the cubic keyframes, whereas below, we see that the cubic keyframe in the middle takes into account the

end velocity of the first linear keyframe, as well as the starting last linear keyframe, creating a smooth transition:

Code

```
LinearKeyframe(-30, duration: 0.25)
CubicKeyframe(30, duration: 0.5)
LinearKeyframe(0, duration: 0.25)
```

Preview



Note that while the keyframeAnimator doesn't require any constraints on the value, the individual keyframes (LinearKeyframe, CubicKeyframe, and MoveKeyframe) require that the target value conforms to Animatable.

Multiple Tracks

Above, we only animated a single CGFloat. Animating a single value is a special case; in general, keyframe animations allow us to provide multiple tracks that all run at the same time. For example, we could extend our shake animation to include a slight wobble through a rotation effect. First, we define a struct that has properties for the two values we want to animate:

```
struct ShakeData {
  var offset: CGFloat = 0
  var rotation: Angle = .zero
}
```

With the exception of the keyframe definitions, the code doesn't change all that much. We now start with an initial value of a different type, and we add the rotation effect inside the content closure:

```
struct ShakeSample2: View {
  @State private var trigger = 0

  var body: some View {
    Button("Shake") {
      trigger += 1
    }
```

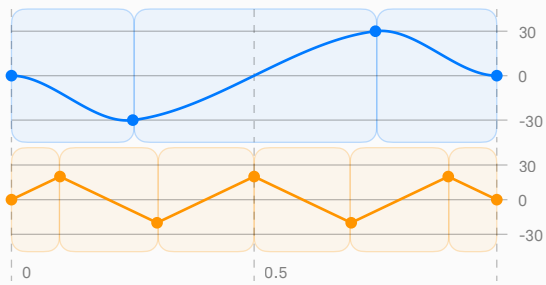
```
    .keyframeAnimator(initialValue: ShakeData(), trigger: trigger) { content, data in
      content
        .offset(x: data.offset)
        .rotationEffect(data.rotation)
    } keyframes: { value in ... }
  }
}
```

The definition of the keyframes is a bit more complicated, though. Instead of animating a single value, we now use KeyframeTrack to construct two separate tracks. SwiftUI will still animate a single ShakeData value, but it uses the first track to animate the offset and the second track to animate the rotation. While the keyframes within a single track run in sequence, the tracks themselves run in parallel. We can see this in the following diagram, where offset and rotation are animated separately but combined into a single value.

Code

```
KeyframeTrack(\.offset) {
  CubicKeyframe(-30, duration: 0.25)
  CubicKeyframe(30, duration: 0.5)
  CubicKeyframe(0, duration: 0.25)
}
KeyframeTrack(\.rotation) {
  LinearKeyframe(.degrees(20), duration: 0.1)
  LinearKeyframe(.degrees(-20), duration: 0.2)
  LinearKeyframe(.degrees(20), duration: 0.2)
  LinearKeyframe(.degrees(-20), duration: 0.2)
  LinearKeyframe(.degrees(20), duration: 0.2)
  LinearKeyframe(.degrees(0), duration: 0.1)
}
```

Preview



In general, tracks don't have to have the same duration. If a track ends while other tracks are still running, the track will report its final value for the rest of the animation. Note that in the keyframeAnimator method, we also receive the current value. Initially, this will be initialValue, but if we tap again, this will be the final value of the previous animation. If we trigger the animation before the previous animation has ended, the closure's initialValue will be the current (interpolated) value.

Thus far, we've used the keyframeAnimator modifier, but there's also a KeyframeAnimator view. Instead of modifying a view, this lets us construct a view

that should be animated. Both the keyframe modifier and the view have a variant that runs every time a trigger value changes, along with a variant that runs indefinitely.

When the built-in keyframe animators aren't good enough, we can also use the `KeyframeTimeline` struct. This takes an initial value and a keyframe description, but instead of using it to animate views, we can query it for the value at a specific time. This is great when we want to animate things other than SwiftUI views. For example, we could use it to render different stages of an animation in a snapshot test, or to animate within a `TimelineView`, or to log the animated value (which is what we did to generate the diagrams above).However, there are some limitations to keyframe animations:

- → Due to their nature, they're only available as implicit animations (there's no `withKeyframeAnimation`).
- → The individual keyframes can only animate `Animatable` values. For example, we can't animate a `Color` value directly using keyframes, as it doesn't conform to `Animatable` (but we can animate a `Color.Resolved` value).
- → Finally, at the time of writing, the trigger-based animations still suffer from some bugs.

Advanced Layout

7

Using the basic layout primitives in SwiftUI, we don't know what sizes are proposed to views, and we don't have information about the sizes of siblings or subviews. Therefore, we have to use the more low-level techniques from this chapter to build layouts that rely on this kind of information.

As of iOS 16 and macOS 13, we can use the `Layout` protocol to create custom layout containers. This protocol works in two steps:

- → First, the container size is determined using the `sizeThatFits` method. Inside that method, we get access to the subviews through their proxies. Each proxy allows us measure a subview by proposing various sizes.
- → In the second step, the subviews are placed within the container's size. Again, we have access to the subviews through their proxies.

The `Layout` protocol currently only allows us to measure and place subviews, but it doesn't allow us to filter them or add additional views (e.g. separators between items). For example, we can't easily build `ViewThatFits` using the `Layout` protocol. While there are some tricks to make this work, it's good to be aware of the limitations.

Likewise, the `Layout` protocol isn't available on platforms released prior to 2022. When we need to support these platforms, we can still build many of the custom layouts using geometry readers, preferences, and overlays. We'll discuss these techniques later in this chapter.

The Layout Protocol

Using the `Layout` protocol, we can build custom container views that lay out their subviews according to an algorithm we write. For example, we can use the protocol to build custom layouts such as masonry layouts, flow layouts, or circular layouts. We can also use the protocol to wrap and tweak existing layouts.

In this section, we'll build a flow layout. Whereas an `HStack` always puts its subviews on a single line, the flow layout line wraps when the next view no longer fits on the current line — similar to how a text view line wraps when the next word doesn't fit.

Using the `Layout` protocol, we'll implement the flow layout using the following steps:

1. Ask each of the subviews for its *ideal size* by proposing `nil×nil`.
2. Compute the positions of all subviews based on the container's own size.
3. Place each subview based on the positions computed in step 2.

The line-based layout algorithm in step 2 takes a list of sizes (the ideal sizes from step 1), spacing, and the container width, and it returns an array of frames (one rectangle for each input size). It works like this:

1. We keep track of the current position as a `CGPoint` starting at (0, 0). The `x` component is the current horizontal position within the line, and it changes with each additional view. The `y` component is the top of the current line, and it only changes when we start a new line.
2. We loop over all the subviews.
    1. If the subview doesn't fit on the current line, we start a new line by resetting the current position's `x` component to zero and incrementing the `y` component by the height of the current line plus spacing.
    2. We use the current position as the origin for this subview's rectangle.
    3. We increase the current position's `x` component by the width of the subview and add spacing.

We can write this algorithm independent of the `Layout` protocol:

```
func layout(
  sizes: [CGSize],
  spacing: CGSize = .init(width: 10, height: 10),
  containerWidth: CGFloat
) -> [CGRect] {
  var result: [CGRect] = []
  var currentPosition: CGPoint = .zero

  func startNewline() {
    if currentPosition.x == 0 { return }
    currentPosition.x = 0
    currentPosition.y = result.union().maxY + spacing.height
  }

  for size in sizes {
    if currentPosition.x + size.width > containerWidth {
      startNewline()
    }
    result.append(CGRect(origin: currentPosition, size: size))
    currentPosition.x += size.width + spacing.width
  }
  return result
}
```

Using this algorithm, our actual layout implementation becomes very short. The `sizeThatFits` method calls `layout` using all the ideal sizes and the proposed container width, and it returns the size of the union of all the frames — similar to how a `ZStack`

also reports the size of the union of all its subviews as its size. The union method is a custom extension we wrote.

Placing the subviews first computes all the frames and then places each of the subviews at their respective frame's origin:

```
struct FlowLayout: Layout {
  func sizeThatFits(proposal: ProposedViewSize, subviews: Subviews, cache: inout ())
    -> CGSize
  {
    let containerWidth = proposal.replacingUnspecifiedDimensions().width
    let sizes = subviews.map { $0.sizeThatFits(.unspecified) }
    return layout(sizes: sizes, containerWidth: containerWidth).union().size
  }

  func placeSubviews(in bounds: CGRect, proposal: ProposedViewSize,
    subviews: Subviews, cache: inout ())
  {
    let subviewSizes = subviews.map { $0.sizeThatFits(.unspecified) }
    let frames = layout(sizes: subviewSizes, containerWidth: bounds.width)
    for (f, subview) in zip(frames, subviews) {
      let offset = CGPoint(x: f.origin.x + bounds.minX, y: f.origin.y + bounds.minY)
      subview.place(at: offset, proposal: .unspecified)
    }
  }
}
```

One interesting effect here is that this layout doesn't always become exactly as wide as proposed, similar to Text when it wraps over multiple lines. If we assume 10 points spacing and three subviews with an ideal width of 100, consider what happens when we propose 250 for the width: our layout cannot fit all three items onto a single line, and it'll put the third item on a second line. It will thus report a width of 210 (the width of the widest line).

Code

```
FlowLayout {
  ForEach(Array(0...10), id: \.self) { idx in
    Rectangle()
      .fill(Color.random)
      .frame(width: .random(in: 10...35))
  }
}
.border(.blue)
.frame(width: 125)
.border(.black)
```

Preview

There are many ways in which we can extend this layout. Here are some ideas we can consider:

- → Add parameters for the alignment within a line. Currently, all our subviews in a single line are top-aligned, but we could support any vertical alignment. To do this, we can ask each of the subviews (using their proxies) for their alignment value.
- → Support horizontal alignment (currently our lines are leading-aligned).
- → Make the spacing controllable from the outside.
- → Add support for special views (e.g. headings) that always have their own line.
- → Come up with better default behavior when the proposed size is `nil` on one (or both) of the axes.

Most of these features aren't very difficult to add, but they require more code than what could reasonably be printed in a book. Therefore, we'll go over the APIs currently available and see how they can help us extend the flow layout. Let's start with the API on the `Layout` protocol:

- → We can define a custom `Cache` type, and we can use this to share computation between the different methods. For example, in `sizeThatFits`, we can use a cache to store the computed frames of the subviews and reuse them in `placeSubviews`. There's a method — `makeCache` — that we can override to either initialize or even construct the cache. However, as the documentation states, it's often not necessary to build our own cache, so we need to verify we have a performance problem before going down this route unnecessarily.

→ We can provide explicit alignment guides using the `explicitAlignment(...)` API. Unlike the `.alignmentGuide` API, we can also query our subviews (through their proxies) to compute their alignment guides. For example, we can expose alignment guides for the `firstFlowBaseline` and `lastFlowBaseline` alignments by overriding the default implementation of the `explicitAlignment` API.

The `LayoutSubview` type also gives us a number of useful APIs we can use to query our subviews:

→ We're not limited to calling `sizeThatFits` once; we can actually call it multiple times. This *probing* is what stack views do to determine the flexibility of the subviews.

→ Instead of `sizeThatFits`, we can also propose a size and get the subview's `ViewDimensions`. The `ViewDimensions` type is similar to `CGSize` in that it has a width and height, but it can also be used to query the alignment guides of the subview. We could use this (for example) to align our subviews on a horizontal line while respecting their custom alignment guides.

→ We can ask for the subview's preferred spacing value and use those values if no explicit spacing has been specified.

→ We can ask for a view's priority. This doesn't make too much sense for a flow layout, but it's useful when implementing (for example) a custom stack layout.

→ Subviews can also define custom *layout values*. These are values that the container can read using a subscript on the view proxy. For example, we could use this to have subviews declare a `heading` layout value as a Boolean and read it in the container view.

Similar to `AnyView`, there's an `AnyLayout` type that erases the concrete type of a layout. Additionally, it can be used to dynamically switch between different layouts. When the switch is done as part of an animation, the frames of the subviews are also animated.

Limitations

The `Layout` protocol was added in iOS 16 and macOS 13, so when we're targeting older platforms, we can't use it. However, we can build almost any custom layout with preferences and geometry readers — without requiring the `Layout` protocol — as we'll explore in the next section.

Another limitation is that we can't communicate directly with the subviews: for example, we can't apply any modifiers, read out preferences, or perform other tasks that require the View protocol. As mentioned in the introduction, all subviews are placed — we can't remove or add to the list of subviews. For example, we can't insert decorative views such as separators, nor can we apply a filter to skip certain subviews.

Preference-Based Layout

As we saw in the previous section, the Layout protocol lets us propose different sizes and place our views within a container. It's possible to replicate almost all of this behavior (and more) using preferences, geometry readers, and ZStack.

Geometry Readers

Using a GeometryReader, we can measure the proposed size. The GeometryReader unconditionally accepts the proposed size, and it reports that size via a GeometryProxy to its view builder closure. Using the geometry proxy, we can access the size of the geometry reader (as a CGSize — this is the proposed size, with nil replaced by the default value of 10 points). We can also read the safe area insets, read the frame (in a specific coordinate space), and resolve anchors.

Often, we see people recommend to never use a geometry reader. We believe much of the sentiment around that is because geometry readers are misunderstood. For example, consider an HStack with three text views: the moment we wrap one of the text views inside a geometry reader, our layout will look completely different. This is because the wrapped text view now accepts the proposed size rather than basing its size on its content.

When using a geometry reader to measure the size of a view, we recommend putting it inside an overlay or background. As we discussed in the Layout chapter, the secondary subview of the overlay is proposed the size of the overlay's primary subview. By putting a geometry reader inside that overlay, we can measure the size of the primary subview without influencing the layout.

Once we have the size, we can do a number of things. For example, let's say we want to display a rounded rectangle as the background and adjust the corner radius according to the geometry reader's size:

Code

```
myView
  .padding()
  .background {
    GeometryReader { proxy in
      let radius: CGFloat = proxy.size.width > 50 ? 10 : 5
      RoundedRectangle(cornerRadius: radius)
        .fill(.blue)
    }
  }
```

Preview

However, when we want to use the size of the view in one of its ancestors, we're out of luck: since the geometry reader is just another view, we can't assign a state property inside the view builder closure or modify state otherwise.

When we're dealing with measuring a single view, we can use `onAppear` together with `onChange(of:)` to work around this limitation. For example, let's imagine we want a view to become its ideal size, but we want to display it within a frame with a width of 100 points and detect when it draws beyond that frame. We create a `Text` and make sure it becomes its ideal size by applying the `fixedSize` modifier. We then apply a frame with a width of 100 to make sure the layout size is always exactly 100 points. We can measure the text's size using an overlay with a geometry reader. Because the geometry reader needs some kind of view as its content, we use a clear color. When the node in the render tree is created (and `onAppear` gets called), or when the text's size crosses the 100-point threshold (and `onChange` gets called), we assign the new value to the state property:

Code

```
struct ContentView: View {
  var text: String
  @State private var overflows: Bool = false

  var body: some View {
    Text(text)
      .fixedSize()
      .overlay {
        GeometryReader { proxy in
          let tooWide = proxy.size.width > 100
          Color.clear
            .onAppear {
              overflows = tooWide
            }
            .onChange(of: tooWide) { _ in
              overflows = tooWide
            }
        }
      }
      .frame(width: 100)
      .border(overflows ? Color.red : Color.clear)
  }
}
```

Preview

Note that we need both onAppear and onChange(of:): the former only fires when the node is first created, and the latter only fires when the value actually changes (but not the first time the view gets rendered). If we need this pattern more often, we can create a helper that combines the two modifiers:

```
extension View {
  func onAppearOrChange<Value: Equatable>(of value: Value,
    perform: @escaping (Value) -> ()) -> some View
  {
    self
      .onAppear { perform(value) }
      .onChange(of: value, perform: perform)
  }
}
```

If we target iOS 17/macOS 14, there's a new variant of onChange that takes an initial parameter, which indicates whether the perform closure should be called initially in addition to when the value changes:

```
Color.clear
  .onChange(of: tooWide, initial: true) {
    overflows = tooWide
  }
```

In many cases, this is the simplest solution to measure the size of a single view and propagate the measurement to an ancestor. However, when we need to measure multiple related views, using `onAppear` and `onChange(of:)` doesn't scale very well. Instead, we can use *preferences*.

Preferences

Preferences are the counterpart to the environment. Instead of propagating values down the tree, *preferences propagate values up the tree*. For example, in the case of a flow layout, we could measure the size of each subview and propagate it to the container using a preference. In addition, preferences let us define how these values are combined. For example, in the case of a flow layout, we'd like to combine each individual measurement into an array or dictionary of sizes.

Similar to the environment, preferences are defined using a slightly complicated syntax to ensure type-safety. The `PreferenceKey` protocol wants a default value (in case no preference was set), but it also requires a `reduce` method. This is used to combine two values. For example, if we want to measure the maximum width of multiple views, we could have a `CGFloat` value and implement `reduce` by taking the maximum. If we're interested in a single value that might be at any point in the subtree, we could model it using an optional and take the first non-`nil` value.

In our case, we want to measure the sizes of all the subviews. We'll model this as an array of `CGSize`. So our preference key looks like this:

```
struct SizeKey: PreferenceKey {
  static var defaultValue: [CGSize] = []

  static func reduce(value: inout [CGSize], nextValue: () -> [CGSize]) {
    value.append(contentsOf: nextValue())
  }
}
```

To measure a single view, we take our knowledge from the previous section and measure using a geometry reader inside an overlay. In there, we use preferences to communicate this value up the view tree:

```
extension View {
  func measureSize() -> some View {
    overlay {
      GeometryReader { proxy in
        Color.clear
          .preference(key: SizeKey.self, value: [proxy.size])
      }
    }
  }
}
```

To measure the sizes of a number of views, we generate a number of subviews using a ForEach and call measureSize on each of the subviews. We also add an onPreferenceChange(_:perform:) modifier somewhere upstream from where we measure the sizes in the view tree and print out the preference value:

```
    ZStack(alignment: .topLeading) {
      ForEach(0..<5) { ix in
        Text("Item \(ix)" + String(repeating: "\n", count: ix/2))
          .padding()
          .measureSize()
      }
    }
    .onPreferenceChange(SizeKey.self) { print($0) }
```

When running this on macOS, the print statement prints an array of all the sizes:

```
[(70.5, 48.0), (68.5, 48.0), (70.0, 64.0), (70.5, 64.0), (70.5, 80.0)]
```

The values from each .preference modifier bubble up the view tree, and the key's reduce method is used to combine values from multiple subviews. Once an .onPreferenceChange(_ key:perform:) modifier with the same key is encountered, the perform closure is called with the aggregated value from the subtree within the .onPreferenceChange. When a state change that affects the preferences happens, this process starts from a blank slate again.

The next step in building the flow layout is to use these sizes to actually lay out the subviews.

Putting it all together

Given the sizes, we can reuse the layout method from earlier in this chapter. Recall that it takes a number of input sizes and returns a frame for each input size. In addition, it requires the width of the container:

```
func layout(
  sizes: [CGSize],
  spacing: CGFloat = 10,
  containerWidth: CGFloat
) -> [CGRect]
```

As a first step, we'll need to measure the container width. When not using the Layout protocol, we can't easily measure the proposed size and compute our own size based on that, so we'll need a few workarounds to do this. First, we'll create an outer ZStack that combines a zero-height geometry reader (to not influence the vertical layout size) and our container view, ZStackFlowLayout. Inside the geometry reader, we measure the proposed size and set it to a state property. On our container, we set the minWidth to 0 and the maxWidth to .infinity. This is a quirky way of writing "become exactly the proposed width" (see the Layout chapter for more details). In other words, this outer ZStack, the GeometryReader, and the ZStackFlowLayout all become exactly the proposed width. As a result, the height of the outer view will effectively be the height of the ZStackFlowLayout:

```
struct ZStackFlowLayoutExample: View {
  @State private var containerWidth: CGFloat?
  var body: some View {
    ZStack {
      GeometryReader { proxy in
        let width: CGFloat = proxy.size.width
        Color.clear
          .onAppearOrChange(of: width) { containerWidth = $0 }
      }
      .frame(height: 0)
      ZStackFlowLayout(containerWidth: containerWidth ?? 0)
        .frame(minWidth: 0, maxWidth: .infinity)
    }
  }
}
```

Now that we have our wrapping view, we can implement the ZStackFlowLayout view. We define a state property that holds the sizes of the subviews, and we take the container width as a parameter. In addition, the subview(for:) method generates a text with a background for each index in the flow layout:

```
struct ZStackFlowLayout: View {
  @State private var sizes: [CGSize]? = nil
  var containerWidth: CGFloat
  let subviewCount = 5
```

```
  func subview(for index: Int) -> some View {
    Text("Item \(index)" + String(repeating: "\n", count: index/2))
      .padding()
      .background {
        RoundedRectangle(cornerRadius: 5)
          .fill(Color(hue: .init(index)/10, saturation: 0.8, brightness: 0.8))
      }
  }

  var body: some View {
    ...
  }
}
```

Inside the body property, we first compute all the offsets, given the sizes. We then create a ZStack with topLeading alignment, and we move each subview using alignment guides. Because the ZStack's layout size will be the union of all the subviews' frames, this will report the correct size to the layout system. This is important when combining the flow layout with other views — for example, when putting it into a scroll view. When the sizes parameter is nil, we position each view at .zero. We render each subview using fixedSize to ensure that the size of the subview is *not* dependent on anything we measure, which prevents infinite layout loops. We then measure the size of the subview and set the alignment guides. Note that to move the subview to, say, position 100×50, we have to set the leading alignment guide to -100 and the top alignment guide to -50. We then add an onPreferenceChange on the top-most view to read out all the preference values:

```
var body: some View {
  let offsets = sizes.map {
    layout(sizes: $0, containerWidth: containerWidth).map { $0.origin }
  } ?? Array(repeating: .zero, count: subviewCount)

  ZStack(alignment: .topLeading) {
    ForEach(0..<subviewCount) { ix in
      subview(for: ix)
        .fixedSize()
        .measureSize()
        .alignmentGuide(.leading) { _ in -offsets[ix].x }
        .alignmentGuide(.top) { _ in -offsets[ix].y }
    }
  }
  .onPreferenceChange(SizeKey.self) { sizes = $0 }
}
```

We might think it'd be more efficient to not store the sizes, but instead directly

compute the layout in onPreferenceChange. While this is possible, it does introduce a problem: once we do that, the view doesn't respond to changes in containerSize. By writing our body as we did, the view will get invalidated any time the container size changes or any time we measure different sizes. Even though we have static subviews, the sizes might still change when something in the environment changes (e.g. dynamic type).

The ZStack-based algorithm to build a flow layout is way more complex than the first version we showed using the Layout protocol. Additionally, it always becomes the proposed width, even if there is (for example) just a single item. But it also shows that we can build intricate layouts using the tools that have been available since the first version of SwiftUI.

Some of the subtleties might not be immediately obvious and are easy to miss when writing this code ourselves. For example, when the above view is first rendered, the onPreferenceChange happens sometime during the layout phase, and the @State property is invalidated. Before even rendering the current frame, the view's body is reexecuted. This prevents flickering and is generally very useful. However, we also have to make sure we don't introduce accidental loops in our layout. When we remove the fixedSize inside the ForEach, we might actually end up in a loop where a subview gets measured, reports a different size, and repeatedly causes a layout invalidation. When this problem happens, we typically get a "Bound preference x tried to update multiple times per frame." warning in the console.

Likewise, when we try to compute the positions inside onPreferenceChange, our code will look like it's working correctly the first time around, but it won't support a changing container width. One easy way to verify this is by running the code as part of a macOS application: resizing the window should cause items to flow.

Variadic Views

A final shortcoming of our ZStack/preference-based flow layout is that we can only support a single type of subview. Of course, we could replace the ForEach with something more complex (by looping over an array of Identifiable items, for example). When using a Layout, however, we can pass in any list of views. For example, here we add six subviews — the first five from a ForEach, and the last view as part of the view builder. This isn't easily doable using the preference-based layout:

```
FlowLayout() {
  ForEach(0..<5) { ix in
    Text("Item \(ix)")
  }
```

```
    RoundedRectangle(cornerRadius: 10)
      .frame(width: 100, height: 100)
}
```

In our preference-based layout, we *can* achieve this syntax using a semi-public (underscored) API called variadic views. These give us access to the view proxies of the underlying views as a collection we can use with `ForEach`. This lets us build an API that's very similar to the layout-based version. While the full code for that is beyond the scope of the book, a finished version is available here.

Preferences aren't just limited to layout; we can also use them for other purposes. For example, if we were to implement our own navigation view alternative, we could build the `navigationTitle` API using preferences. In general, when we build reusable components that wrap other views, those views could communicate back to the component using preferences.

Coordinate Spaces

When dealing with geometry readers, we can measure a view's geometry in different coordinate spaces. There are two built-in coordinate spaces: global and local. In addition, we can define our own coordinate spaces. Consider the following view, which has an explicit coordinate space around the stack:

```
struct ContentView: View {
  var body: some View {
    VStack {
      Text("Hello")
      Text("Second")
        .overlay { GeometryReader { proxy in
          let _ = print([
            proxy.frame(in: .global).pretty,
            proxy.frame(in: .local).pretty,
            proxy.frame(in: .named("Stack").pretty)
          ].joined(separator: "\n"))
          Color.clear
        }}
    }
    .coordinateSpace(name: "Stack")
  }
}
```

When we run the above code on an iOS simulator, it prints three different values:

- → The origin within the global coordinate space is (167, 438). This is the distance from the top-leading edge of the screen.
- → The origin within the local coordinate space is (0, 0). Local always means local to the view, which in this case is the geometry reader, which is the same coordinate space as the second text label (not the container).
- → The origin within the stack is (0, 20), as the view with the geometry reader is the stack's second subview.

The printed sizes for the view above are all the same. However, consider what happens when we apply a scale effect:

```
Text("Hello")
  .overlay { GeometryReader { proxy in
    let _ = print([
      proxy.frame(in: .global).size.pretty,
      proxy.frame(in: .local).size.pretty,
    ].joined(separator: "\n"))
    Color.clear
  }}
  .scaleEffect(0.5)
```

As we discussed in the Layout chapter, effects only change how the view is drawn, but they don't change the layout of the view. However, asking the geometry proxy for a view's frame relative to a specific coordinate space will take any effects inside of that coordinate space into account. In the example above, the size measured in the global coordinate space is half of the size measured in the local coordinate system (to be more precise: both the width and the height are half of their local counterparts).

Anchors

Anchors are built on top of geometry readers, preferences, and coordinate spaces. In essence, an anchor is a wrapper around a geometry value (a `CGRect`, `CGSize`, or `CGPoint`) measured in the *global coordinate space*. A geometry proxy has special support for anchors and lets us automatically transform the geometry value into the *local coordinate space*.

For example, consider building an onboarding flow where we want to highlight certain parts of our interface by putting an ellipse around the current item:

Code

```
VStack {
  Button("Sign Up") {}
  Button("Log In") {}
    .overlay {
      Ellipse()
        .strokeBorder(Color.red, lineWidth: 1)
        .padding(-5)
    }
}
.border(Color.green)
```

Preview

In general, the ellipse might be obscured by siblings of the login button, or by any other views that are higher up the view tree. In this case, we want to draw the ellipse at the very top, even though its position is determined by a view that's possibly deep inside the hierarchy.

To achieve this, we can propagate the frame of the login button using a preference, and then draw the ellipse as an overlay on top of the entire view tree. As a first step, we create a key for propagating our anchor:

```
struct HighlightKey: PreferenceKey {
  typealias Value = Anchor<CGRect>?
  static var defaultValue: Value

  static func reduce(value: inout Value, nextValue: () -> Value) {
    value = value ?? nextValue()
  }
}
```

Second, we replace the drawing of the ellipse by setting an anchor preference. This propagates the position and size of the second button (as measured in the global coordinate space) up the view tree:

```
VStack {
  Button("Sign Up") {}
  Button("Log In") {}
    .anchorPreference(key: HighlightKey.self, value: .bounds, transform: { $0 })
}
```

Now we add the drawing outside of our VStack. Earlier in this chapter, we used onPreferenceChange to get the value of a preference, but in this case, we use a simpler API that gives us access to the preference value inside an overlay. This lets us avoid having a separate state property. Inside the overlay, we use a geometry reader and its

proxy to resolve the anchor — the Anchor<CGRect> is transformed into a CGRect within the local coordinate space:

```
VStack {
  ...
}
.border(Color.green)
.overlayPreferenceValue(HighlightKey.self) { value in
  if let anchor = value {
    GeometryReader { proxy in
      let rect = proxy[anchor]
      Ellipse()
        .strokeBorder(Color.red, lineWidth: 1)
        .padding(-5)
        .frame(width: rect.width, height: rect.height)
        .offset(x: rect.origin.x, y: rect.origin.y)
    }
  }
}
```

Note that anchors are there for convenience; we could've written the same code either by defining a custom coordinate space (and measuring everything in terms of the coordinate space), or by measuring everything in the global coordinate space. However, both of these alternatives would have been more code and potentially more error-prone.

In our experience, anchors are most useful when we find ourselves converting coordinates or sizes between different coordinate spaces. This often happens when we work with multiple views that are at different depths in the view hierarchy. In the flow layout example further up in this chapter, all subviews shared the same parent, and as a result, we didn't need to perform any conversions between different coordinate spaces.

Matched Geometry Effect

The matchedGeometryEffect API is used to give one or more views (the targets) the same position, size, or frame (position and size) as another view (the source). This works by measuring the size and position of the source view and then proposing the same size to the target views, as well as setting offsets on them. The source and target views belonging to one matched geometry group can be located anywhere in the view tree and are identified using a namespace and a secondary identifier (which can be any hashable value).

As an example, we could reimplement the highlighting ellipse code from above using a matched geometry effect, instead of manually propagating anchors via preferences:

Code

```
struct ContentView: View {
  @Namespace var namespace
  let groupID = "highlight"

  var body: some View {
    VStack {
      Button("Sign Up") {}
      Button("Log In") {}
        .matchedGeometryEffect(id: groupID, in: namespace)
    }
    .overlay {
      Ellipse()
        .strokeBorder(Color.red, lineWidth: 1)
        .padding(-5)
        .matchedGeometryEffect(id: groupID, in: namespace, isSource: false)
    }
  }
}
```

Preview

Both views — the button in the stack, and the stroked ellipse with the padding — belong to the same geometry group because we specified the same namespace and a common identifier. In this case, we used a string as the identifier, but we can use any other hashable value, like an integer or a UUID.

Within a geometry group, only one view is allowed to be the source for the group. The isSource parameter is true by default, so we explicitly have to specify false for the ellipse in the overlay, since the matching should happen from the button to the ellipse.

> We could actually implement a matched geometry effect API ourselves using the techniques we've discussed in this chapter so far. We'd need an explicit view modifier somewhere at the top level to read the source geometries from the preferences and to propagate them down to the targets via the environment. As a framework, SwiftUI has the luxury of being able to do that implicitly. Other than that though, there's no magic to creating a matched geometry effect.

When the button's frame gets matched to the ellipse, we can think of this as if a frame and an offset were inserted (this isn't necessarily how it's implemented, but it works well as a mental model):

```
VStack {
  ...
}
.overlay {
  Ellipse()
    .strokeBorder(Color.red, lineWidth: 1)
    .padding(-10)
    .frame(width: <matchedWidth>, height: <matchedHeight>)
    .offset(x: <offsetX>, y: <offsetY>)
  }
```

Therefore, the source's size is proposed to the target view. As we discussed before, it's entirely up to the frame's subview to decide what to do with that proposed size. For example, if the target view is a fixed-size view (e.g. a non-resizable image or another fixed frame), the matched geometry effect won't influence the target view's size.

If we want to match a view's size to another view, it's important for both views to have the same flexibility; otherwise, matching might not work. Of course, we might also intentionally restrict the flexibility of the target view (e.g. by using a flexible frame with a maximum width) to constrain the target's behavior.

Matched Geometry Effect and Transitions

Matched geometry effects are applied before a view is inserted and after it has been removed, similar to how the active state of a transition is applied to a view that's being inserted or removed. This allows us to create smooth transitions from one view to another with very little code.

When we insert a new view that's part of a geometry group into the view hierarchy, its initial geometry is computed as if the `isSource` parameter on this view is set to `false`, i.e. the view's geometry is matched to the source view that's in the view tree before the state change. When we remove a view from the view tree, its final geometry is matched to the source view that's going to be in the view tree after the state change.

This allows us to write very easy transitions from one view to another. Here's an example of a transition between a 30×30 circle and a fullscreen circle:

```
struct ContentView: View {
  @State private var hero = false
  @Namespace var namespace
```

```
  var body: some View {
    let circle = Circle().fill(Color.green)
    ZStack {
      if hero {
        circle
          .matchedGeometryEffect(id: "image", in: namespace)
      } else {
        circle
          .matchedGeometryEffect(id: "image", in: namespace)
          .frame(width: 30, height: 30)
      }
    }
    .onTapGesture {
      withAnimation(.default) { hero.toggle() }
    }
  }
}
```

When we toggle the `hero` flag in the example above, we see the circle animate smoothly between the 30×30 size and the fullscreen size.

There are two parts to this animation: the matched geometry effect takes care of the position and size, and the default transition (opacity) blends between the two views. In other words, during the transition, *both* views are onscreen. When we change `hero` from `false` to `true`, the first view is inserted with an opacity of `0` and a frame that matches the second view (which was the source view of the geometry group before the state change). During that transition, the first view animates from that matched initial position toward its final position while fading in. At the same time, the second view (that got removed) fades out and animates from its former position before the state change to the position it would occupy after the state change, now that it's matched to the newly inserted source view.

Note that the order of the modifiers is both crucial and easy to get wrong. For example, if we swap the `matchedGeometryEffect` and the `frame` in the `else` branch, the smaller circle will always stay at its 30×30 size:

```
if hero {
  ...
} else {
  circle
    .frame(width: 30, height: 30)
    .matchedGeometryEffect(id: "image", in: namespace)
}
```

As we mentioned above, the `.matchedGeometryEffect` modifier essentially translates to a `.frame` and `.offset` modifier. Since the innermost frame with a width and height of

30 points always determines the size of the circle, the matched geometry effect has no effect, at least with regard to the size of the view.

Beyond removing and inserting views from the view tree entirely, matched geometry effects can also be used creatively to animate views that remain in the view hierarchy. We can insert and remove views from a geometry group by changing their namespaces or identifiers.